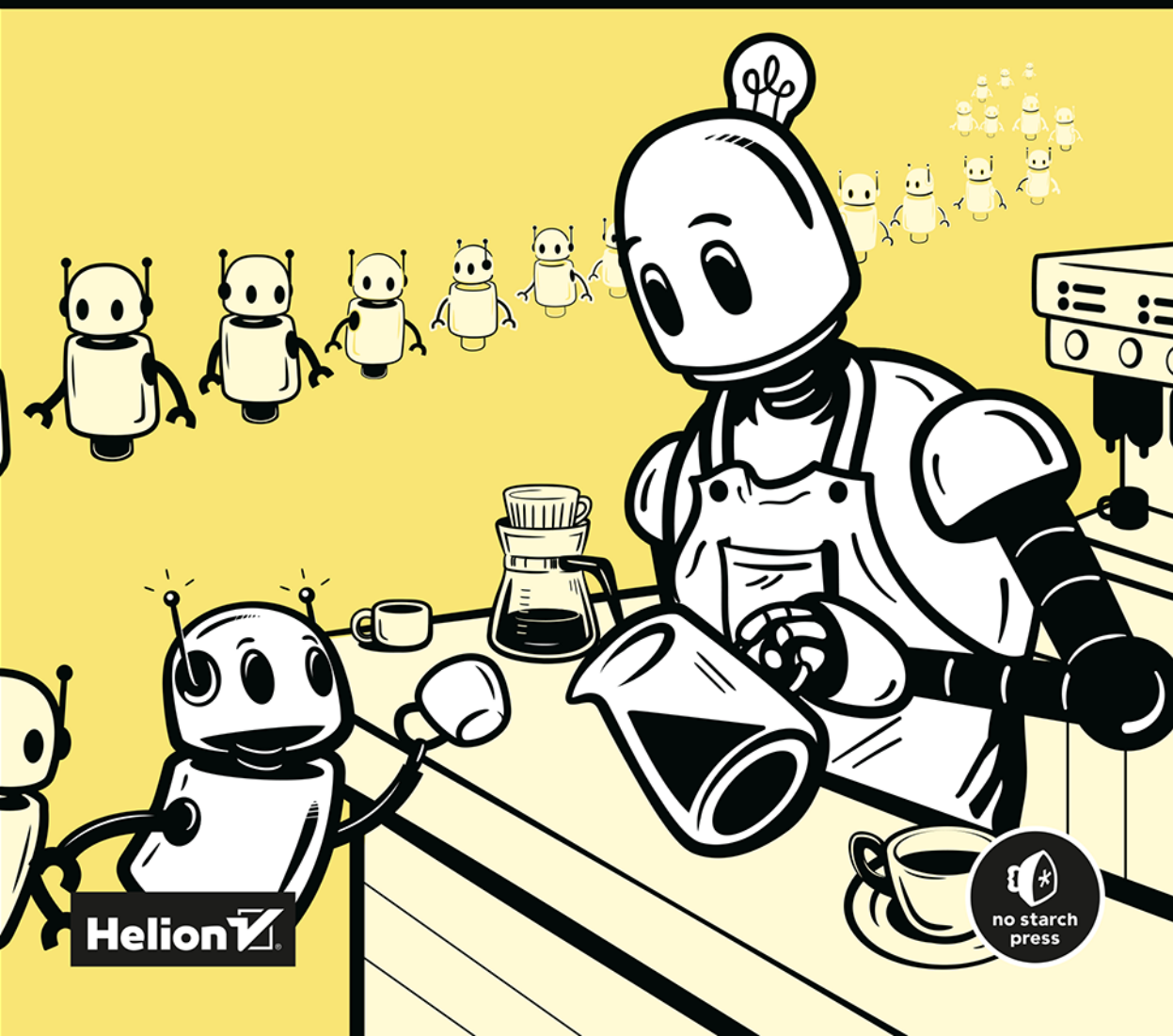
# STRUKTURY DANYCH
## Z PRZYMRUŻENIEM OKA

ZABAWNA PRZYGODA
Z PRZYKŁADAMI PACHNĄCYMI KAWĄ

JEREMY KUBICA



Helion
no starch press

Tytuł oryginału: Data Structures the Fun Way: An Amusing Adventure with Coffee-Filled Examples

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-1006-5



Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*https://helion.pl/user/opinie/stdapr*
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: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

# Spis treści

# 3

# Dynamiczne struktury danych

Ten rozdział jest wprowadzeniem do **dynamicznych struktur danych**, które zmieniają swoją strukturę wraz ze zmianą danych. To może być zwiększenie rozmiaru struktury danych na żądanie, tworzenie dynamicznych i zmiennych połączeń między różnymi wartościami i wiele więcej. Dynamiczne struktury danych leżą w sercu niemal każdego programu komputerowego na świecie i stanowią podstawę kilku najbardziej fascynujących, ciekawych i potężnych algorytmów w informatyce.

Najprostsze struktury danych wprowadzone w poprzednich rozdziałach są jak miejsca parkingowe — dają nam miejsce do przechowywania informacji, ale nie potrafią się dostosowywać. Oczywiście możemy posortować wartości w tablicy (lub samochody na parkingu) i użyć tej struktury, by wyszukiwanie binarne było wydajne, ale to jest tylko zmiana kolejności danych w obrębie tablicy. Sama struktura danych nigdy się nie zmienia ani nie reaguje na zmiany w danych. Gdy później zmienimy dane w posortowanej tablicy, na przykład przez zmianę wartości jednego z elementów, to musimy ponownie posortować tablicę. Gorzej jest, gdy jesteśmy zmuszeni zmieniać samą strukturę danych, na przykład zmniejszyć lub powiększyć tablicę. Prosta statyczna struktura danych nie pomaga.

W tym rozdziale porównano wprowadzoną w rozdziale 1. statyczną strukturę danych, tablicę, z prostą dynamiczną strukturą danych z listą, by pokazać zalety tej drugiej. Pod pewnymi względami te dwie struktury danych są podobne, gdyż obie umożliwiają programistom przechowywanie wielu wartości i uzyskiwanie dostępu do nich za pomocą

jednego odwołania, albo tablicy, albo początku listy. Jednak tablice przybierają sztywną strukturę w momencie ich tworzenia, jak rzędy miejsc parkingowych. Natomiast listy mogą rosnąć w trakcie funkcjonowania programu. Swoim działaniem przypominają kolejkę ludzi, która wydłuża się i skraca, pozwalając na dodawanie i usuwanie wartości. Gdy zrozumiesz te różnice, łatwiej Ci będzie zrozumieć bardziej zaawansowane struktury danych, które będziemy omawiać w dalszej części niniejszej książki.

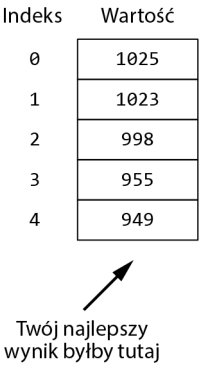
## Ograniczenia tablic

Choć tablice są świetnymi strukturami danych do przechowywania wielu wartości, mają jedno poważne ograniczenie. Ich rozmiar i układ w pamięci są ustalane na sztywno w momencie tworzenia. Jeśli chcemy przechowywać więcej wartości, niż możemy zmieścić w naszej tablicy, musimy utworzyć nową, większą tablicę i skopiować do niej dane ze starej tablicy. Taka sztywna struktura jest do zaakceptowania, gdy wiemy, że nie przekroczymy danej liczby elementów. Jeśli mamy wystarczającą liczbę pojemników, możemy ustawiać wartości pojedynczych elementów, nie martwiąc się o sztywny układ tablicy w pamięci. Jednak w wielu sytuacjach potrzebujemy dynamicznej struktury danych, która może rosnąć i zmieniać się w trakcie działania programu.

Aby wyjść naprzeciw zapotrzebowaniu na dynamiczne struktury danych, wiele współczesnych języków programowania oferuje dynamiczne „tablice", które rosną i kurczą się, gdy dodajesz i usuwasz elementy. Jednak to są tak naprawdę klasy opakowujące tablice statyczne lub inne struktury danych, które ukrywają złożoność i koszt związany z ich dynamiczną naturą. Choć jest to wygodne dla programisty, może prowadzić do ukrytych problemów z wydajnością, Gdy dodajemy elementy na koniec tablicy, program wciąż musi zwiększać zajmowaną pamięć. Robi to w tle. Aby zrozumieć, dlaczego dynamiczne struktury danych są tak ważne, musimy powiedzieć o ograniczeniach statycznych struktur danych. W niniejszej książce będziemy używać terminu *tablica*, gdy będziemy mówić o prostej statycznej tablicy.

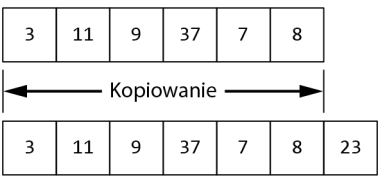
W celu zwizualizowania ograniczeń tablicy wyobraź sobie, że spędziłeś cały tydzień na opanowywaniu najnowszej retro gry komputerowej *Space Frogger 2000*. Z radością patrzysz na główny ekran z tabelą pięciu najlepszych wyników. Te niesamowite osiągi odzwierciedlają godziny potu, łez krzyku i jeszcze więcej łez. Jednak już nazajutrz Twój (wkrótce były) najlepszy przyjaciel odwiedza Cię i pobija Twoje rekordy pięć razy z rzędu. Zanim wyrzucisz zdradzieckiego eksprzyjaciela za drzwi, wracasz do swojej gry i wpatrujesz się w nowe najlepsze wyniki, widoczne na rysunku 3.1, i wykrzykujesz: „Dlaczego ta gra nie może zapisywać większej liczby wyników? Czy naprawdę takie trudne byłoby zapisywanie dziesięciu najlepszych wyników albo przynajmniej dodanie jednego na samym końcu?".

To jedno z głównych ograniczeń każdej sztywnej struktury danych i jej sztywnego układu w pamięci — nie może rosnąć wraz z danymi. Przez te ograniczenia niektóre powszechne działania stają się kosztowne. Wyobraź sobie ograniczenia edytora tekstu, w którym mógłbyś wpisać tylko ograniczoną liczbę znaków, arkusz kalkulacyjny ze stałą liczbą wierszy, program do przechowywania zdjęć z ograniczoną liczbą zdjęć lub dziennik kawosza ograniczony do 1000 wpisów.




*Rysunek 3.1. Pięcioelementowa tablica przechowująca najwyższe wyniki gry komputerowej. Niestety żaden z nich nie jest Twój*

Jako że rozmiar tablicy jest ustawiany na sztywno w momencie jej tworzenia, to jeśli chcemy poszerzyć tablicę dla większej liczby danych, musimy utworzyć nowy, większy blok pamięci. Zastanówmy się nad najprostszym dodaniem jednego elementu na koniec tablicy. Ponieważ tablica jest pojedynczym blokiem pamięci o sztywnym rozmiarze, nie możemy po prostu wcisnąć kolejnej wartości na koniec. Być może to miejsce w pamięci jest już zajmowane przez inną zmienną. Zamiast ryzykować nadpisanie wartości tej zmiennej, musimy przypisać nowy (większy) blok pamięci i na końcu zapisać tam nową wartość. To wiele zachodu dla jednego elementu, co pokazano na rysunku 3.2.




*Rysunek 3.2. Dodawanie elementu na koniec wypełnionej tablicy*

Tablicę możesz sobie wyobrazić jako ciepły bufet ze stałą liczbą pojemników. Łatwo jest wyciągnąć pusty pojemnik i włożyć nowy z jajecznicą. Ale nie dasz rady wcisnąć na koniec nowego pojemnika. Nie ma na to miejsca. Jeśli szef kuchni postanowi dodać do menu naleśniki, muszą być umieszczone gdzie indziej.

Jeżeli wiesz, że będziesz musiał dodać wiele nowych wartości, możesz *zamortyzować* koszt wielokrotnych aktualizacji. Możesz przyjąć **strategię podwajania tablicy**, co polega na podwajaniu rozmiaru tablicy podczas każdego poszerzania jej o nowe elementy. Na przykład jeśli chcemy dodać 129. element do naszej 128-elementowej tablicy, musimy najpierw utworzyć nową tablicę o rozmiarze 256 i skopiować do niej 128 elementów z pierwotnej tablicy. Dzięki temu przez pewien czas będziemy mogli dodawać nowe elementy do tablicy i nie martwić się o przypisywanie nowego obszaru w pamięci. Jednak będzie się to odbywać kosztem potencjalnie zmarnowanego miejsca w pamięci. Jeżeli w sumie potrzebujemy tylko 129 elementów, mamy 127 elementów w nadmiarze.

Podwajanie rozmiaru tablicy zapewnia rozsądną równowagę między kosztownymi kopiami tablic a zmarnowaną pamięcią. Gdy tablica rośnie, podwajanie staje się coraz rzadsze. Jednocześnie podwajanie rozmiaru wypełnionej tablicy oznacza zmarnowanie mniej niż połowy miejsca. Jednak nawet to zbalansowane podejście wyraźnie pokazuje koszt używania tablicy o sztywnym rozmiarze, zarówno jeśli chodzi o koszt kopiowania, jak i zużycie pamięci.

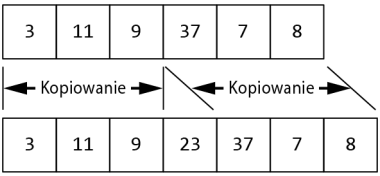
```
ArrayDouble(Array: old_array):
    Integer: length = length of old_array
    Array: new_array = empty array of size length * 2

    Integer: j = 0
    WHILE j < length:
        new_array[j] = old_array[j]
        j = j + 1
    return new_array
```

Kod podwajający rozmiar tablicy zaczyna się od przypisania nowej tablicy o rozmiarze dwukrotnie większym niż obecna tablica. Pojedyncza pętla `WHILE` przechodzi przez elementy w bieżącej tablicy i kopiuje je do nowej tablicy. Zwracana jest nowa tablica.

Wyobraź sobie, że stosujesz tę strategię dla miejsca na półce. Otwieramy księgarnię o nazwie „Struktury Danych i Nie Tylko” i wstawiamy do niej pięć skromnych półek. Dzień otwarcia pokazał zaskakującą potrzebę większego zróżnicowania, dlatego musimy zwiększyć nasze zaopatrzenie. Spanikowani przeprowadzamy się do nowej lokalizacji z 10 półkami i przenosimy książki. Potrzeba tymczasowo została zaspokojona. Jako że brak wszechstronnej struktury przechowywania jest wyraźnym problemem księgarni, nasz sklep oddala się od osiągnięcia sukcesu i wymagania cały czas rosną. Możemy się jeszcze wielokrotnie przeprowadzać do lokali z 20, 40, a nawet 80 półkami. Za każdym razem musimy zabezpieczyć nową lokalizację i przenieść wszystkie książki.

Stała lokalizacja wartości tablicy w pamięci jest kolejnym ograniczeniem. Nie możemy łatwo umieścić nowych elementów pośrodku tablicy. Nawet jeśli na końcu na naszej pierwotnej tablicy jest jeszcze miejsce na nowy element i nie musimy przenosić całej tablicy do nowego bloku pamięci, wciąż musimy przesuwać każdy element po kolei, by zrobić miejsce dla nowej wartości pośrodku. W przeciwieństwie do półki z książkami nie możemy po prostu przesunąć wszystkich elementów naraz za jednym popchnięciem. Gdybyśmy mieli 10 000 elementów i chcielibyśmy dodać coś jako drugi element, musielibyśmy przesunąć 9999 elementów. To dużo zachodu, żeby wstawić jeden element.

Problem staje się jeszcze większy, gdy chcemy wstawić nowe wartości pośrodku tablicy, która jest już pełna. Nie tylko musimy przypisać nowy blok pamięci i skopiować do niego stare wartości, ale musimy również przesunąć wszystkie wartości występujące po nowej wartości o jedną pozycję w dół, by zrobić miejsce dla nowego elementu. Załóżmy na przykład, że chcemy wstawić wartość 23 jako czwarty element istniejącej 6-elementowej tablicy, tak jak pokazano na rysunku 3.3.

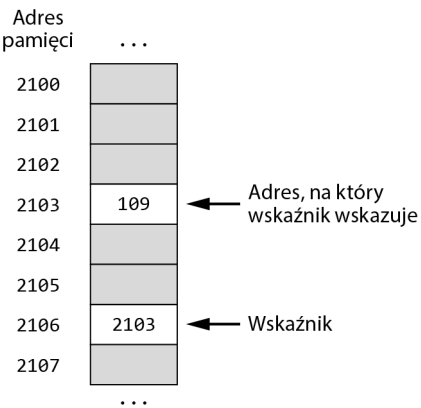



*Rysunek 3.3. Dodawanie elementu pośrodku wypełnionej tablicy*

Aby zaradzić dziurom w tablicy, musimy przejść na bardziej elastyczne struktury danych, które potrafią rosnąć w miarę dodawania nowych elementów — na dynamiczne struktury danych. Zanim poznamy szczegóły, wprowadźmy wskaźniki, typ zmiennych będący kluczem do ponownego ustawiania i poszerzania struktur danych.

# Wskaźniki i referencje

Jeden typ zmiennych stoi nad innymi, zarówno jeśli chodzi o jego czystą moc, jak i zdolność dezorientowania nowych programistów — **wskaźniki**. Wskaźnik to zmienna przechowująca tylko adresy pamięci komputera. Na rysunku 3.4 wskaźnik wskazuje na drugą lokalizację w pamięci, gdzie zapisane są dane.




*Rysunek 3.4. Wskaźnik wskazujący adres pamięci komputera*

Dociekliwy czytelnik może zapytać, jaki jest cel zmiennej, która po prostu wskazuje inną lokalizację w pamięci. Przecież do tego służy już nazwa zmiennej. Dlaczego nie zapisywać swoich danych w zmiennej jak normalni ludzie? Dlaczego zawsze trzeba tak wszystko komplikować? Nie słuchaj sceptyków. Wskaźniki są bardzo ważnym składnikiem dynamicznych struktur danych, o czym się wkrótce przekonamy.

Załóżmy, że pracujemy nad dużym projektem architektonicznym i zebraliśmy folder z przykładowymi rysunkami, którymi chcemy się podzielić z naszym zespołem. Wkrótce folder zapełnia się licznymi planami pięter, szacunkowymi kosztami i wizualizacjami

gotowej inwestycji. Zamiast kopiować opasły plik, zostawiamy naszym współpracownikom notatkę, że znajdą plik w pokoju na trzecim piętrze, w szafce numer 3, w drugiej szufladzie od dołu, w piątym folderze. Ta notatka pełni funkcję wskaźnika. Nie podaje wszystkich informacji, które są zawarte w pliku, ale raczej pomaga naszym kolegom z pracy znaleźć informacje. Co ważniejsze, możemy podzielić się tylko jednym adresem z każdym z naszych współpracowników bez konieczności kopiowania całego pliku. Każdy z nich może przeczytać tę notatkę, by znaleźć i w razie konieczności wprowadzić zmiany. Możemy nawet na biurku zostawić każdemu członkowi zespołu notatkę na karteczce samoprzylepnej, podając 10 zmiennych wskazujących na tę samą informację.

Poza zapisywaniem lokalizacji bloku pamięci wskaźniki mogą przyjmować wartość *null* (oznaczaną jako `None`, `Nil` lub `0` w niektórych językach programowania). Wartość *null* oznacza po prostu, że wskaźnik w danym momencie nie wskazuje na żadną poprawną lokalizację w pamięci. Innymi słowy, wartość *null* oznacza, że wskaźnik na nic jeszcze nie wskazuje.

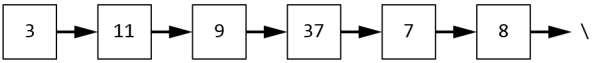
Różne języki programowania zapewniają różne mechanizmy w celu osiągnięcia tego, do czego służą wskaźniki, i nie wszystkie z nich podają programiście surowy adres pamięci. Języki programowania niższego poziomu, takie jak C i C++, dają surowe wskaźniki i umożliwiają bezpośredni dostęp do lokalizacji w pamięci. Inne języki programowania, takie jak Python, używają referencji, które mają taką samą składnię jak zwykłe zmienne, ale wciąż mogą wskazywać na inną zmienną. Te różne odmiany mają różne działania i sposoby użycia (dereferencja, arytmetyka wskaźników, postać wartości *null* i tak dalej). Aby nie komplikować, w tej książce będziemy używać pojęcia *wskaźnik* dla wszystkich zmiennych zaimplementowanych przez wskaźniki, referencje, indeksy z góry przypisanych bloków pamięci. Nie będziemy się przejmować skomplikowaną składnią potrzebną, by uzyskać dostęp do bloków pamięci (która niejednego entuzjastę programowania doprowadziła do łez). Do definiowania wskaźnika w pseudokodzie będziemy również używać wskazywanego typu (zamiast bardziej ogólnego typu wskaźnika). Nam głównie chodzi o to, że wskaźniki zapewniają mechanizm powiązania bloku pamięci tak jak w naszej pierwszej dynamicznej strukturze danych — liście.

# Listy

**Listy** są najprostszym przykładem dynamicznej struktury danych i są bliskimi krewnymi tablic. Podobnie jak tablice, są strukturami danych przechowującymi wiele wartości. Jednak w przeciwieństwie do tablic listy składają się z łańcucha węzłów połączonych ze sobą wskaźnikami. Podstawowy **węzeł** w liście jest złożoną strukturą danych zawierającą dwie części: wartość (dowolnego typu) i wskaźnik do kolejnego węzła w liście.

```
LinkedListNode {
    Type: value
    LinkedListNode: next
}
```

Listę możemy przedstawić jako szereg połączonych pojemników, tak jak na rysunku 3.5. Każdy pojemnik przechowuje pojedynczą wartość i zawiera wskaźnik do następnego pojemnika w szeregu.




*Rysunek 3.5. Lista pokazana jako szereg połączonych pojemników*

Ukośnik na końcu listy przedstawia wartość *null* i wskazuje koniec listy. W rzeczywistości mówimy, że wskaźnik `next` ostatniego węzła nie wskazuje na żaden węzeł.

Lista jest jak długa kolejka ludzi stojących po kawę w naszej ulubionej kawiarni. Ludzie rzadko znają swoje bezwzględne położenie — „Stoję na 53 kafelku od lady". Raczej przywiązują wagę do swojego względnego położenia, czyli do osoby stojącej przed nimi, którą zapisujemy we wskaźniku. Nawet jeśli kolejka zawija się na zewnątrz kawiarni, układając się w skomplikowane pętle, wciąż możemy ustalić kolejność przez zapytanie każdej osoby stojącej przed nami. Możemy przejść wzdłuż kolejki i pytać każdą osobę, kto jest przed nią.

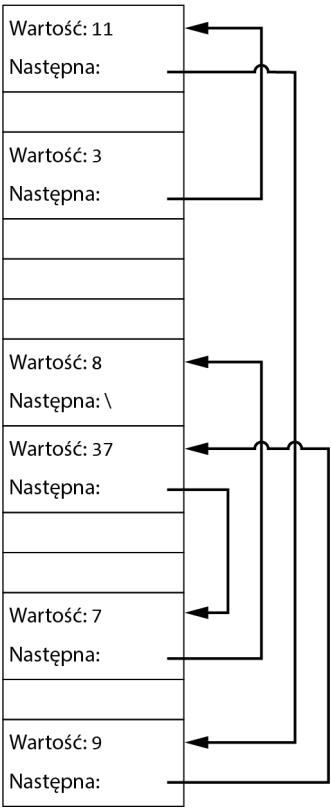
Ponieważ listy uwzględniają zarówno wskaźniki, jak i wartości, potrzebują więcej miejsca w pamięci niż tablice przechowujące dokładnie te same wartości. Mając tablicę o rozmiarze $K$ z wartościami $N$-bajtowymi, potrzebujemy tylko $K \cdot N$ bajtów. Natomiast jeśli każdy wskaźnik wymaga kolejnych $M$ bajtów, nasza struktura danych ma $K \cdot (M + N)$ bajtów. Jeżeli rozmiar wskaźników nie jest dużo mniejszy niż rozmiar naszych wartości, narzut jest znaczący. Jednak często warto poświęcić więcej pamięci, by zyskać większą elastyczność zapewnianą przez wskaźniki.

Choć podręczniki często przedstawiają listy jako zgrabne, zorganizowane struktury (jak widać na rysunku 3.5 albo na naszym przykładzie kolejki po kawę), nasza lista tak naprawdę może być rozsiana po pamięci komputera. Jak pokazano na rysunku 3.6, węzły listy są powiązane tylko za pomocą wskaźników.

To prawdziwa moc wskaźników i dynamicznych struktur danych. Nie musimy trzymać naszej listy w jednym ciągłym bloku pamięci. Możemy zapisywać nowe węzły w wolnym miejscu pamięci.

Zwykle programy zapisują listy przez zapisanie pojedynczego wskaźnika na *początek* listy. Program może uzyskać dostęp do dowolnego elementu listy przez rozpoczęcie od początku listy i przejście w pętli po wszystkich węzłach za pomocą wskaźników.

```
LinkedListLookUp(LinkedListNode: head, Integer: element_number):
    LinkedListNode: current = head ❶
    Integer: count = 0

    WHILE count < element_number AND current != null: ❷
        current = current.next
        count = count + 1
    return current
```

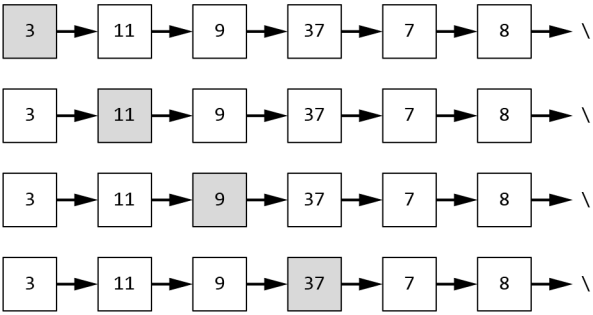



*Rysunek 3.6. Lista w pamięci komputera. Węzły niekoniecznie znajdują się obok siebie*

Kod zaczyna pracę na początku listy ❶. Definiujemy drugą zmienną `count`, by zapisywać indeks bieżącego węzła. Pętla `WHILE` przechodzi przez każdy węzeł listy, dopóki nie znajdzie poprawnej liczby, `count == element_number`, lub nie dojdzie do końca listy, `current == null` ❷. W każdym przypadku kod zwraca wartość zmiennej `current`. Jeśli pętla kończy swoje działania, ponieważ wyszła poza listę, to indeks nie znajduje się na liście i kod zwraca `null`.

Na przykład gdybyśmy chcieli uzyskać dostęp do czwartego elementu listy, program najpierw uzyskałby dostęp do pierwszego elementu listy, następnie do drugiego, trzeciego, czwartego w celu znalezienia odpowiedniego miejsca w pamięci. Rysunek 3.7 pokazuje ten proces dla listy, której pierwszą wartością jest 3.

Jednak warto nadmienić, że w przypadku list potrzebujemy większej liczby obliczeń niż w przypadku tablic. W celu uzyskania dostępu do elementu tablicy musimy jedynie obliczyć przesunięcie i sprawdzić odpowiednie miejsce w pamięci. Tablice potrzebują tylko jednego obliczenia i jednego wyszukania w pamięci bez względu na indeks wybranego przez nas elementu. Listy wymagają przechodzenia w pętli przez kolejne elementy od początku listy, dopóki nie dotrzemy do elementu, który nas interesuje. Dla dłuższych list brak możliwości bezpośredniego dostępu może powodować znaczny wzrost liczby koniecznych obliczeń.




*Rysunek 3.7. Przechodzenie przez listę wymaga przechodzenia od jednego węzła do kolejnego wzdłuż łańcucha wskaźników*

Na pierwszy rzut oka ten wyznaczony wzorzec w celu uzyskania dostępu jest głosem przeciwko listom. Bardzo wyraźnie zwiększyliśmy koszt szukania dowolnego elementu! Zastanów się, co to oznacza dla wyszukiwania binarnego. Pojedyncze wyszukiwanie wymaga przejścia w pętli przez wiele elementów i to, że lista jest posortowana, traci na znaczeniu.

Jednak pomimo tych kosztów listy mogą być prawdziwym atutem w zastosowaniach praktycznych. Struktury danych niemal zawsze wiążą się z pewnymi kompromisami pomiędzy złożonością, wydajnością i wzorcami użycia. Sposób działania struktury danych może dyskwalifikować ją dla jednego zastosowania, ale może być doskonałym wyborem dla innego algorytmu. Zrozumienie tych kompromisów jest kluczowe dla skutecznego łączenia algorytmów i struktur danych. W przypadku list w zamian za zwiększoną liczbą obliczeń potrzebną do uzyskania dostępu do elementów otrzymujemy znaczny wzrost elastyczności samej struktury danych, co zobaczymy w następnym podrozdziale.

# Działania na listach

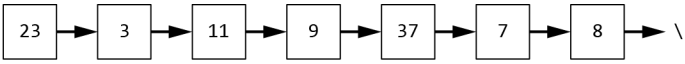
Choć niektórzy narzekają na chaos panujący w listach w porównaniu z czystym pięknem zwięzłych tablic, to właśnie możliwość łączenia różnych bloków pamięci daje tej strukturze danych taką moc, dzięki której możemy *dynamicznie* reorganizować strukturę danych. Porównajmy wstawianie nowej wartości do tablicy z dodawaniem nowej wartości do listy.

## Wstawianie do listy

Jak widzieliśmy, wstawianie nowego elementu do tablicy może wymagać od nas przypisania nowego (większego) bloku pamięci i skopiowania wszystkich wartości z pierwotnej tablicy do nowego bloku. Co więcej, ze wstawianiem nowego elementu może wiązać się konieczność przejścia przez tablicę i przesunięcia każdego elementu.

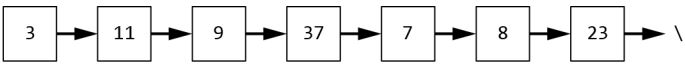
Z kolei lista nie musi być zapisana w jednym ciągłym bloku pamięci — i najprawdopodobniej nawet na początku to nie jest jeden blok. Musimy tylko znać położenie nowego węzła, zaktualizować wskaźnik `next` poprzedzającego węzła, by wskazywał na nasz nowy węzeł, i ustawić wskaźnik nowego węzła, by wskazywał na odpowiedni węzeł. Jeśli chcemy dodać węzeł z wartością 23 na początek listy pokazanej na rysunku 3.5, ustawiamy po

prostu wskaźnik `next` nowego węzła do poprzedniego początku listy (wartości równej 3). Ten proces został pokazany na rysunku 3.8. Wszystkie zmienne wskazujące wcześniej na początek listy (pierwszy węzeł) również muszą być zaktualizowane, by wskazywały na nowy pierwszy węzeł.




*Rysunek 3.8. Wydłużanie listy przez dodanie nowego węzła na początku*

W podobny sposób możemy dodać węzeł na koniec listy (patrz rysunek 3.9) przez przejście przez listę do samego końca, aktualizację wskaźnika `next` ostatniego węzła (wartość równa 8), by wskazywał na nowy węzeł, i ustawienie wskaźnika `next` nowego węzła na `null`. To podejście wymaga przejścia całej listy, by dojść od końca, ale jak się przekonamy w następnym rozdziale, są sposoby, aby uniknąć tego dodatkowego kosztu.




*Rysunek 3.9. Wydłużanie listy przez dodanie nowego węzła na końcu*

Jeśli chcemy wstawić wartość pośrodku, musimy zaktualizować dwa wskaźniki: poprzedniego węzła i wstawionego węzła. Na przykład aby dodać węzeł $N$ między węzłami $X$ i $Y$, musimy wykonać dwa kroki:

1. Ustawić wskaźnik `next` węzła $N$ na węzeł $Y$ (to samo miejsce, na które obecnie wskazuje wskaźnik `next` węzła $X$).
2. Ustawić wskaźnik `next` węzła $X$, by wskazywał na węzeł $N$.

Kolejność tych dwóch kroków jest istotna. Wskaźniki, jak wszystkie inne zmienne, mogą przechowywać tylko jedną wartość — w tym przypadku jeden adres pamięci. Gdybyśmy najpierw ustawili wskaźnik `next` węzła $X$, stracilibyśmy informację o tym, gdzie znajduje się węzeł $Y$.

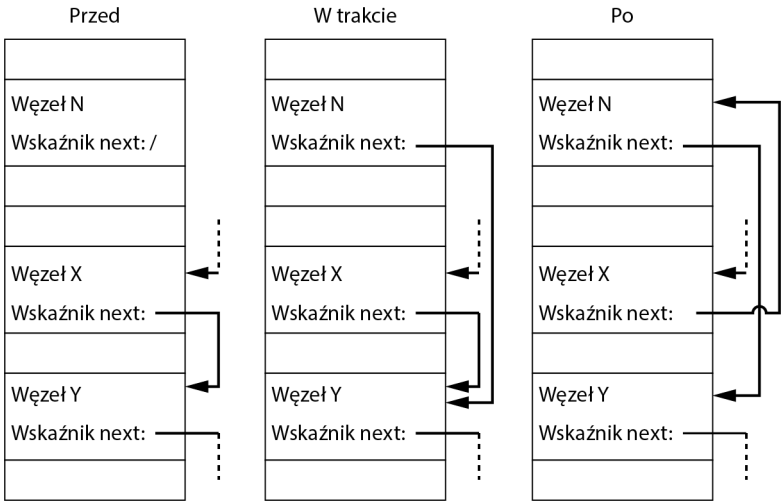
Po wykonaniu tych dwóch kroków $X$ wskazuje na $N$, a $N$ wskazuje na $Y$. Rysunek 3.10 przedstawia ten proces.

Pomimo żonglowania wskaźnikami kod dla tych operacji jest stosunkowo prosty.

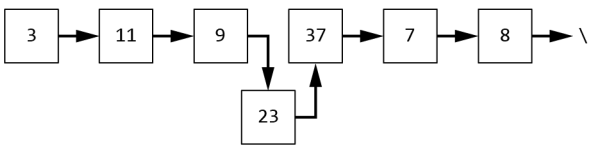
```
LinkedListInsertAfter(LinkedListNode: previous, LinkedListNode: new_node):
    new_node.next = previous.next
    previous.next = new_node
```

Powiedzmy, że jednak chcemy do naszej listy wstawić węzeł z wartością 23 między węzły 9 i 37. Na rysunku 3.11 pokazano łańcuch wskaźników, jaki będziemy mieć w efekcie tego działania.

Analogicznie gdy jeden z klientów stojących w połowie kolejki wpuści swojego przyjaciela przed siebie, zmienią się dwa wskaźniki. Każda osoba wskazuje na inną lub pamięta, kto stoi przed nią. Wspaniałomyślny klient teraz wskazuje na swojego kolegę, który stoi




*Rysunek 3.10. Proces wstawiania nowego węzła N do listy między węzłami X i Y*




*Rysunek 3.11. Wstawianie węzła 23 do listy wymaga aktualizacji wskaźników poprzedniego węzła (9) i kolejnego węzła (37)*

przed nim. A szczęśliwy klient wskazuje na osobę, która poprzednio była przed jego uprzejmym przyjacielem. Wszystkie osoby stojące za nimi w kolejce spoglądają nieprzychylnie w ich stronę i mamroczą pod nosem niemiłe rzeczy.

I znowu porównanie do kolejki w kawiarni i schematy ukrywają prawdziwe zamieszanie związane z procesem wstawiania. Nie wstawiamy nowego węzła w miejscu w pamięci sąsiadującym z ostatnim węzłem, logicznie wstawiamy go w następne wolne miejsce. Sam węzeł może być na drugim końcu pamięci komputera, obok zmiennej zliczającej nasze literówki lub kubki wypitej kawy. Dopóki aktualizujemy wskaźniki, węzły i wskaźniki, które na nie wskazują, możemy traktować jak pojedynczą listę.

Oczywiście wstawiając nowy węzeł na początku listy (indeks równy 0) lub na końcu listy, musimy zadbać o dodatkowe rzeczy. Jeśli wstawiamy węzeł przed pierwszym węzłem, musimy zaktualizować pierwszy wskaźnik, gdyż w przeciwnym razie nadal będzie wskazywał na stary pierwszy węzeł i nie będziemy w stanie uzyskać dostępu do nowego pierwszego węzła listy. Jeśli chcemy wstawić węzeł po ostatnim indeksie listy, to nie mamy odpowiedniego poprzedniego węzła o indeksie `index -1`. W tym przypadku możemy zakończyć wstawianie niepowodzeniem, zwracać błąd lub dodawać element na końcu listy (z mniejszym indeksem). Bez względu na to, jakie podejście wybrałeś, musisz to wyraźnie opisać w dokumentacji kodu. Tę dodatkową logikę możemy spakować w funkcję pomocniczą, która wykorzystuje nasz kod przeszukiwania liniowego do wstawiania nowego węzła w określonej pozycji.

```
LinkedListInsert(LinkedListNode: head, Integer: index, Type: value):
    # Przypadek szczególny — wstawianie nowego pierwszego węzła.
    IF index == 0: ❶
        LinkedListNode: new_head = LinkedListNode(value)
        new_head.next = head
        return new_head

    LinkedListNode: current = head
    LinkedListNode: previous = null
    Integer: count = 0
    WHILE count < index AND current != null: ❷
        previous = current
        current = current.next
        count = count + 1

    # Sprawdzanie, czy dotarliśmy do końca listy przed dotarciem do wymaganego indeksu.
    IF count < index: ❸
        Błąd nieważnego indeksu.

    LinkedListNode: new_node = LinkedListNode(value) ❹
    new_node.next = previous.next
    previous.next = new_node

    return head ❺
```

Kod wstawiania nowego węzła zaczyna się od szczególnego przypadku wstawiania nowego węzła pod indeksem równym 0 (`index = 0`), czyli na początku listy ❶. Tworzy nowy węzeł początkowy, ustawia wskaźnik `next` nowego węzła początkowego, wskazujący na uprzedni pierwszy węzeł, i zwraca nowy początek listy. Jako że nie ma węzła przed nowym węzłem początkowym, w tym przypadku nie musimy aktualizować wskaźnika `next` poprzedniego węzła.

Dla elementów znajdujących się w środku listy kod musi przejść przez listę, by znaleźć odpowiednią lokalizację ❷. Ten kod przypomina kod `LinkedListLookUp` dla wyszukiwania w liście. Kod idzie po wskaźnikach `next` każdego węzła, zapisując bieżący węzeł (`current`) i zliczając kroki (`count`), dopóki nie dotrze do końca listy lub szukanej lokalizacji. Kod zapisuje również dodatkowe informacje. Zmienna `previous` jest wskaźnikiem do węzła znajdującego się *przed* bieżącym węzłem. Zapisywanie tej informacji pozwala nam zaktualizować ten wskaźnik, by wskazywał na wstawiony węzeł.

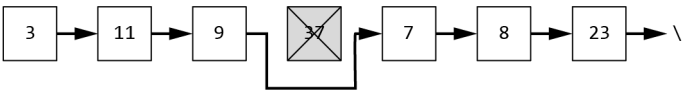
Następnie kod sprawdza, czy dotarł do szukanego indeksu, gdzie chcemy wstawić nowy węzeł ❸. Jeśli `count < index`, wciąż mamy możliwość wstawienia nowego węzła na końcu listy. Kod zwraca błąd w przypadkach, gdy próbujemy wstawić przynajmniej jedno dodatkowe miejsce *po* końcu listy.

Jeżeli kod znajdzie odpowiednią lokalizację do wstawienia nowego węzła, skleja `previous` i `current`. Kod wstawia nowe elementy przez tworzenie nowego węzła, ustawienie wskaźnika `next` tego węzła na adres wskazywany przez zmienną `previous.next`. Następnie ustawia zmienną `previous.next`, by wskazywała na nowy węzeł ❹. Ta logika sprawdza się również, gdy dołączamy nowy węzeł zaraz po ostatnim węźle listy. Ponieważ w tym przypadku `previous.next == null`, wskaźnik `next` nowego węzła będzie miał wartość `null` i będzie poprawnie wskazywał na nowy koniec listy.

Dzięki zwracaniu początku listy ❺ możemy wstawiać węzły przed pierwszym węzłem. Moglibyśmy również opakować początkowy węzeł w złożonej strukturze danych `LinkedList` i działać bezpośrednio na niej. Później w tej książce wykorzystamy to podejście do obsługi binarnych drzew poszukiwań.

## Usuwanie z listy

Aby usunąć dowolny element z listy, musimy po prostu usunąć ten węzeł i dostosować wskaźnik poprzedzającego go węzła, jak pokazano na rysunku 3.12.




*Rysunek 3.12. Usuwane węzła (37) z listy wymaga aktualizacji wskaźnika poprzedzającego węzła (9), by wskazywał na kolejny węzeł (7)*

Można to porównać do osoby, która opuszcza kolejkę, gdyż uważa, że ta kawa nie jest warta tyle czekania. Patrzy na zegarek, mówi do siebie, że ma kawę w domu, i idzie. Jeśli tylko osoba stojąca za klientem, który opuścił kolejkę, wie, za kim teraz stoi, ciągłość kolejki jest zachowana.

Usuwanie z tablicy wiąże się ze znacznie wyższym kosztem, gdyż musimy przesunąć wszystko po węźle zawierającym (37) o jeden pojemnik w stronę początku tablicy, by zlikwidować przerwę. Możliwe, że musielibyśmy przejść całą tablicę.

Podczas usuwania również musimy zwrócić szczególną uwagę na usuwanie pierwszego elementu listy lub próbę usuwania elementu poza końcem listy. Usuwając pierwszy węzeł, aktualizujemy wskaźnik na początek listy, by wskazywał nowy pierwszy węzeł, który jednocześnie staje się początkiem listy. Próba usunięcia węzła za końcem listy może zakończyć się podobnie do próby wstawiania, to znaczy możemy pominąć usuwanie lub zwrócić błąd. Kod pokazany poniżej zwraca błąd.

```
LinkedListDelete(LinkedListNode: head, Integer: index):
    IF head == null: ❶
        return null

    IF index == 0: ❷
        new_head = head.next
        head.next = null
        return new_head

    LinkedListNode: current = head
    LinkedListNode: previous = null
    Integer: count = 0
    WHILE count < index AND current != null: ❸
        previous = current
        current = current.next
        count = count + 1

    IF current != null:❹
        previous.next = current.next ❺
```

```
        current.next = null ❻
    ELSE:
        Błąd nieważnego indeksu.
    return head ❼
```

Ten kod stosuje to samo podejście co wstawianie. Tym razem zaczynamy od dodatkowego sprawdzenia ❶. Jeśli lista jest pusta, nie mamy czego usuwać i zwracamy wartość `null`, by wskazać, że lista jest wciąż pusta. W przeciwnym razie sprawdzamy, czy do usunięcia jest pierwszy węzeł ❷, a jeśli tak, usuwamy pierwszy węzeł z listy i zwracamy adres nowego początku listy.

Aby usunąć któryś z dalszych węzłów (`index > 0`), kod musi dotrzeć do odpowiedniego miejsca na liście. Przy zastosowaniu tego samego podejścia co dla wstawiania kod, przechodząc w pętli po węzłach, zapisuje `current`, `count` i `previous`, dopóki nie dotrze do szukanej lokalizacji lub do końca listy ❸. Jeśli kod znajdzie węzeł pod odpowiednim indeksem ❹, wyklucza węzeł do usunięcia przez ustawienie wskaźnika `previous.next`, by wskazywał na węzeł znajdujący się tuż za bieżącym węzłem ❺. Jednak jeśli pętla `WHILE` dojdzie do końca listy, a `current` przybierze wartość `null`, to oznacza, że nie ma już czego usuwać, i kod zwraca błąd. Ta funkcja ustawia również wskaźnik `next` usuniętego węzła na `null`, by zapewnić spójność (nie ma już następnego węzła w liście) i umożliwić językom programowania poprawne zwolnienie pamięci, która nie jest już używana ❻. Na końcu funkcja zwraca adres początkowego węzła ❼.

Możemy zmienić ten kod tak, by usuwał element listy na podstawie innej informacji niż indeks węzła. Gdybyśmy mieli wartość węzła, który chcemy usunąć, moglibyśmy zmienić warunek pętli ❸ w celu usunięcia pierwszego węzła z tą wartością.

```
WHILE current != null AND current.value != value:
```

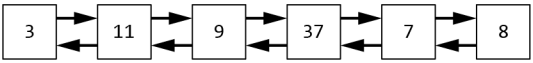
W tym przypadku musimy odwrócić kolejność porównania i sprawdzać, czy wartość zmiennej `current` to `null`, przed uzyskaniem dostępu do jej wartości. Tak samo, jeśli musimy usunąć węzeł na podstawie wskaźnika do niego, możemy porównać adres zapisany w danym wskaźniku z adresem bieżącego węzła.

Listy mają taką moc między innymi dlatego, że możemy wstawiać i usuwać elementy bez przesuwania pozostałych elementów w pamięci komputera. Możemy zostawić węzły tam, gdzie są, i tylko zaktualizować wskaźniki.

# Listy dwukierunkowe

Istnieją liczne sposoby na dodanie struktury ze wskaźnikami, a wiele z nich omówimy w następnych rozdziałach. Teraz przedstawimy jedno proste rozszerzenie listy (jednokierunkowej) — **listę dwukierunkową**, która zawiera zarówno wskaźnik na kolejny element, jak i na poprzedni, co widać na rysunku 3.13.

Dla algorytmów, które muszą przechodzić przez listę w dwóch kierunkach, albo jako programiści lubiący wyzwania i chcący zwiększyć liczbę wskaźników w swoich strukturach danych, możemy łatwo przystosować listę jednokierunkową do postaci listy dwukierunkowej.




*Rysunek 3.13. Lista dwukierunkowa zawiera wskaźniki na poprzedni i następny element*

```
DoublyLinkedListNode {
    Type: Value
    DoublyLinkedListNode: next
    DoublyLinkedListNode: previous
}
```

Kod działający na listach dwukierunkowych jest podobny do kodu dla list jednokierunkowych. Wyszukiwanie, wstawianie i usuwanie często wymagają przejścia przez listę w celu odnalezienia odpowiedniego elementu. Aktualizacja odpowiednich wskaźników `previous` poza wskaźnikami `next` wymaga dodatkowej logiki. Już tak niewielka ilość dodatkowych informacji może dać nam możliwość skrócenia niektórych działań. Mając wskaźnik do każdego węzła w liście dwukierunkowej, można uzyskać dostęp do węzła występującego przed węzłem bieżącym, bez konieczności przechodzenia przez całą listę od początku, tak jak tego wymaga lista jednokierunkowa.

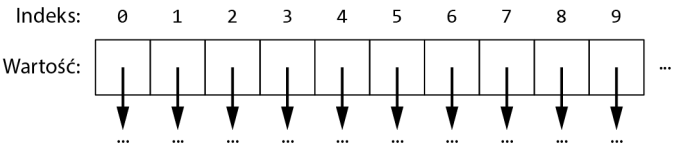
# Tablice i listy

Dotychczas używaliśmy głównie tablic do przechowywania pojedynczych wartości (liczbowych). Możemy przechowywać listę najlepszych wyników, harmonogram budzenia dla inteligentnego budzika lub prowadzić rejestr codziennego spożycia kawy. Tablice przydają się w wielu różnych sytuacjach, a te właśnie wymienione to najbardziej podstawowe zastosowania. Możemy wykorzystać wskaźniki, by zapisywać bardziej złożone elementy o różnym rozmiarze.

Wyobraź sobie, że planujesz imprezę. Załóżmy, że Twoje przyjęcie, w przeciwieństwie do tych organizowanych przez autora tej książki, jest tak popularne, że potrzebujesz listy osób, które potwierdziły swoje przybycie. Po otrzymaniu pierwszych odpowiedzi piszesz nowy program z użyciem tablicy z imionami gości. W każdym elemencie chcesz przechowywać przynajmniej jeden łańcuch znaków. Jednak szybko zdajesz sobie sprawę, że łańcuchy znaków mogą mieć różne długości, więc nie możesz mieć pewności, że zmieszczą się w pojemnikach o określonym rozmiarze. Mógłbyś ustalić taki rozmiar pojemnika, by pomieścić wszystkie możliwe łańcuchy znaków. Ale jaki rozmiar byłby odpowiedni? Czy możesz z pewnością stwierdzić, że wszyscy Twoi goście mają mniej niż 1000 znaków w swoim imieniu i nazwisku? A jeśli zezwolisz na 1000 znaków, czy to nie będzie marnowanie miejsca w pamięci? Jeśli zarezerwujemy miejsce dla 1000 znaków na jednego gościa, wtedy wpis dla *Jan Kowalski* zajmie tylko niewielką część pojemnika. A co, jeślibyś chciał uwzględnić jeszcze bardziej dynamiczne dane, takie jak lista ulubionych zespołów muzycznych gościa lub pseudonimy?

Naturalnym rozwiązaniem jest połączenie tablic i wskaźników, tak jak pokazano na rysunku 3.14. Każdy pojemnik w tablicy zapisuje jeden wskaźnik na interesujące nas

dane. W tym przypadku w każdym pojemniku znajduje się wskaźnik na łańcuch znaków zlokalizowany w innym miejscu w pamięci. Dzięki temu możemy mieć dane o różnej wielkości. Możemy przypisać tyle pamięci, ile potrzebujemy dla każdego łańcucha znaków, i wskazywać na te łańcuchy znaków z poziomu tablicy. Możemy nawet utworzyć szczegółową złożoną strukturę danych dla naszych wpisów z potwierdzeniem przybycia, a w tablicy przechowywać do nich połączenia.




*Rysunek 3.14. Tablice mogą przechowywać szereg wskaźników, dzięki którym można się łączyć z większymi strukturami danych*

Wpisy dla potwierdzeń przybycia nie muszą mieścić się w pojemnikach tablicy, gdyż ich wartości znajdują się w innym miejscu w pamięci. W pojemnikach tablicy znajdują się tylko wskaźniki (o stałej wielkości). Tak samo węzły listy mogą zawierać wskaźniki do innych danych. W przeciwieństwie do wskaźników `next` w listach, które wskazują na kolejne węzły, te wskaźniki mogą wskazywać na dowolne bloki danych.

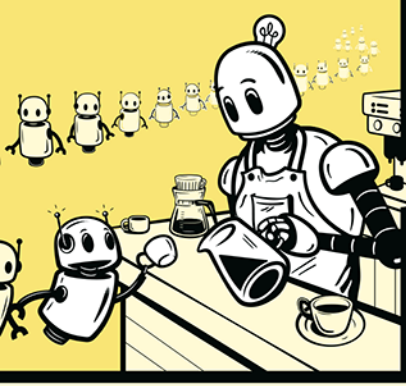
W pozostałej części książki znajdziesz wiele przykładów, gdzie pojedyncze „wartości" są tak naprawdę wskaźnikami na złożone, a nawet dynamiczne struktury danych.

# Dlaczego to takie ważne?

Listy i tablice są najprostszymi przykładami kompromisu między złożonością, wydajnością i elastycznością w naszych strukturach danych. Dzięki wskaźnikom, zmiennym przechowującym adres pamięci, możemy łączyć różne bloki pamięci. Tablica z pojemnikami o sztywnym rozmiarze może zawierać wskaźniki na złożone rekordy danych lub łańcuchy o różnej długości. Co więcej, wskaźniki mogą służyć do tworzenia dynamicznie połączonych struktur w pamięci komputera. Możemy zmieniać wartości wskaźnika, by wskazywał na nowy adres, i w ten sposób mamy możliwość zmiany naszej struktury w dowolnym czasie, zawsze wtedy, gdy jest to konieczne.

W pozostałych rozdziałach zobaczymy liczne przykłady zastosowania dynamicznych struktur danych w celu zarówno ulepszenia organizacji danych, jak i poprawy wydajności pewnych obliczeń. Jednak ważne jest, by pamiętać, co w zamian za to trzeba poświęcić. Jak widzieliśmy w przypadku tablic i list, każda struktura danych ma swoje wady i zalety, jeśli chodzi o elastyczność, wymagane miejsce w pamięci, wydajność i złożoność. W następnym rozdziale zobaczymy, jak można budować na tych podstawowych koncepcjach, by utworzyć dwie struktury danych — stosy i kolejki, które różnią się od siebie działaniem.


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



NALEJ SOBIE KAWY
I WYJDŹ POZA STANDARDOWE
PODEJŚCIE!

O strukturach danych można myśleć jako o konstruktach do organizowania i zapisywania danych. Zrozumienie, czym są, jak je tworzyć i do czego się przydają, jest jednym z fundamentów programowania. Bez tego nie można pisać efektywnego i skalowalnego kodu. Jednak dla wielu osób opanowanie struktur danych stanowi poważne wyzwanie.

Dzięki tej książce ta trudna sztuka musi Ci się udać! Znajdziesz tu gruntowne, a przy tym zabawne wprowadzenie do tworzenia i używania struktur danych. Naukę oprzesz na przejrzystych schematach i dowcipnych porównaniach, aby już wkrótce móc tworzyć wydajniejszy i elastyczny kod. Nieistotne, jakim językiem programowania się posługujesz — podczas lektury zaimplementujesz za pomocą pseudokodu kilkanaście głównych struktur danych, w tym stosy, filtry Blooma, drzewa czwórkowe i grafy. Fantazyjne przykłady ułatwią Ci intuicyjne posługiwanie się tymi strukturami danych. Dowiesz się, jak indeksować przedmioty kolekcjonerskie, optymalizować wyszukiwanie za pomocą latającej wiewiórki, a nawet jak znaleźć najbliższy kubek kawy!

## Z tą książką nauczysz się:

- znajdować równowagę między szybkością, elastycznością i zużyciem pamięci
- projektować struktury danych, które dynamicznie rosną lub maleją
- łączyć proste struktury danych, by przeprowadzać zaawansowane operacje
- znajdować i uzyskiwać dane w tabelach z haszowaniem
- przyspieszać wyszukiwanie za pomocą binarnych drzew poszukiwań
- poprawiać wydajność poszukiwań przy użyciu B-drzew

**Dr Jeremy Kubica** specjalizuje się w rozwijaniu sztucznej inteligencji i uczenia maszynowego. Na Carnegie Mellon University tworzył algorytmy wykrywające zabójcze asteroidy (niewykluczone, że w przyszłości zajmie się badaniami nad powstrzymywaniem asteroid). Jest autorem wielu książek wprowadzających do informatyki, między innymi *Komputerowego detektywa* (Helion, 2022).

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 978-83-289-1006-5
9 788328 910065

Cena: 69,00 zł

no starch press