Street Coder - Zasady do złamania i jak je złamać

tłumaczenie czatem GuPeTe

https://www.manning.com/books/street-coder

1 Na ulicy

Ten rozdział obejmuje
Rzeczywistość ulic
Kim jest programista uliczny?
Problemy współczesnego rozwoju oprogramowania
Jak rozwiązać swoje problemy za pomocą wiedzy ulicznej

Mam szczęście. Napisałem swoją pierwszą pętlę w latach 80. Wystarczyło mi tylko włączyć komputer, co zajęło mniej niż sekundę, napisać 2 linie kodu, wpisać RUN, i voila! Ekran nagle zapełnił się moim imieniem. Byłem od razu pod wrażeniem możliwości. Jeśli mogłem to zrobić za pomocą 2 linii, wyobraź sobie, co mogłem zrobić z 6 liniami, czy nawet z 20 liniami! Moje dziewięcioletnie mózgowe było zalane tyle dopaminy, że od razu uzależniłem się od programowania.

Dziś rozwój oprogramowania jest ogromnie bardziej skomplikowany. Nie ma już tej prostoty lat 80., gdy interakcje użytkownika ograniczały się do "naciśnij dowolny klawisz, aby kontynuować", chociaż czasami użytkownicy mieli problem ze znalezieniem "dowolnego" klawisza na swojej klawiaturze. Nie było okien, myszy, stron internetowych, elementów interfejsu użytkownika, bibliotek, frameworków, maszyn wirtualnych, urządzeń mobilnych. Miałeś tylko zestaw poleceń i statyczną konfigurację sprzętu.

Istnieje powód dla każdego poziomu abstrakcji, który teraz posiadamy, i nie jest tak, że jesteśmy masochistami, z wyjątkiem programistów Haskell1. Te abstrakcje są wprowadzane, ponieważ są jedynym sposobem na nadążanie za obecnymi standardami oprogramowania. Programowanie już nie polega na wypełnianiu ekranu swoim imieniem. Twoje imię musi być we właściwej czcionce i musi znajdować się w oknie, które możesz przeciągać i zmieniać jego rozmiar. Twój program musi wyglądać dobrze. Powinien obsługiwać kopiowanie i wklejanie. Musi obsługiwać różne nazwy dla konfigurowalności. Być może powinien przechowywać nazwy w bazie danych, nawet w chmurze. Wypełnianie ekranu swoim imieniem już nie jest takie zabawne.

Na szczęście mamy zasoby, które pomagają nam poradzić sobie z tą złożonością: uniwersytety, hackathony, obozy programistyczne, kursy online i kaczki gumowe.

PORADA

Kaczka gumowa debugging to ezoteryczna metoda znajdowania rozwiązań problemów programistycznych. Polega na rozmawianiu z żółtym plastikowym ptakiem. Opowiem ci więcej o tym w rozdziale o debugowaniu.

1.1 Co ma znaczenie na ulicach

Świat zawodowego rozwoju oprogramowania jest dość tajemniczy. Niektórzy klienci przysięgają, że zapłacą Ci w ciągu kilku dni za każdym razem, gdy do nich dzwonisz przez miesiące. Niektórzy pracodawcy w ogóle nie płacą Ci wynagrodzenia, ale upierają się, że zapłacą Ci "kiedy już zarobią pieniądze". Chaotyczna przypadkowość wszechświata decyduje, kto dostaje biurowe okno. Niektóre błędy znikają, gdy używasz debugera. Niektóre zespoły w ogóle nie korzystają z kontroli wersji. Tak, to przerażające. Ale musisz zmierzyć się z rzeczywistością.

Jedno jest jasne na ulicach: najważniejsza jest wydajność. Nikogo nie obchodzi twój elegancki projekt, twoja wiedza o algorytmach czy wysokiej jakości kod. Wszystko, co ich obchodzi, to ile możesz dostarczyć w określonym czasie. Wbrew intuicji, dobry projekt, dobre wykorzystanie algorytmów i wysokiej jakości kod mogą znacząco wpływać na twoją wydajność, a tego wielu programistów nie rozumie. Takie sprawy są zazwyczaj postrzegane jako przeszkody, tarcia między programistą a terminem. Taki sposób myślenia może uczynić z ciebie zombi z kulą u nogi.

W rzeczywistości niektórym ludziom zależy na jakości twojego kodu: twoim kolegom. Nie chcą pilnować twojego kodu. Chcą, żeby twój kod działał, był łatwo zrozumiały i możliwy do utrzymania. To jest coś, co im zawdzięczasz, ponieważ kiedy raz zatwierdzisz swój kod w repozytorium, staje się on kodem każdego. W zespole wydajność zespołu jest ważniejsza niż wydajność każdego z jego członków. Jeśli piszesz zły kod, spowalniasz swoich kolegów. Brak jakości twojego kodu szkodzi zespołowi, a spowolniony zespół szkodzi produktowi, a niezrealizowany produkt szkodzi twojej karierze.

Najłatwiejszą rzeczą, którą możesz napisać od zera, jest pomysł, a następną najłatwiejszą rzeczą jest projekt. Dlatego dobre projektowanie ma znaczenie. Dobre projektowanie to nie coś, co dobrze wygląda na papierze. Możesz mieć projekt w głowie, który działa. Natkniesz się na ludzi, którzy nie wierzą w projektowanie i improwizują kod. Ci ludzie nie cenią swojego czasu.

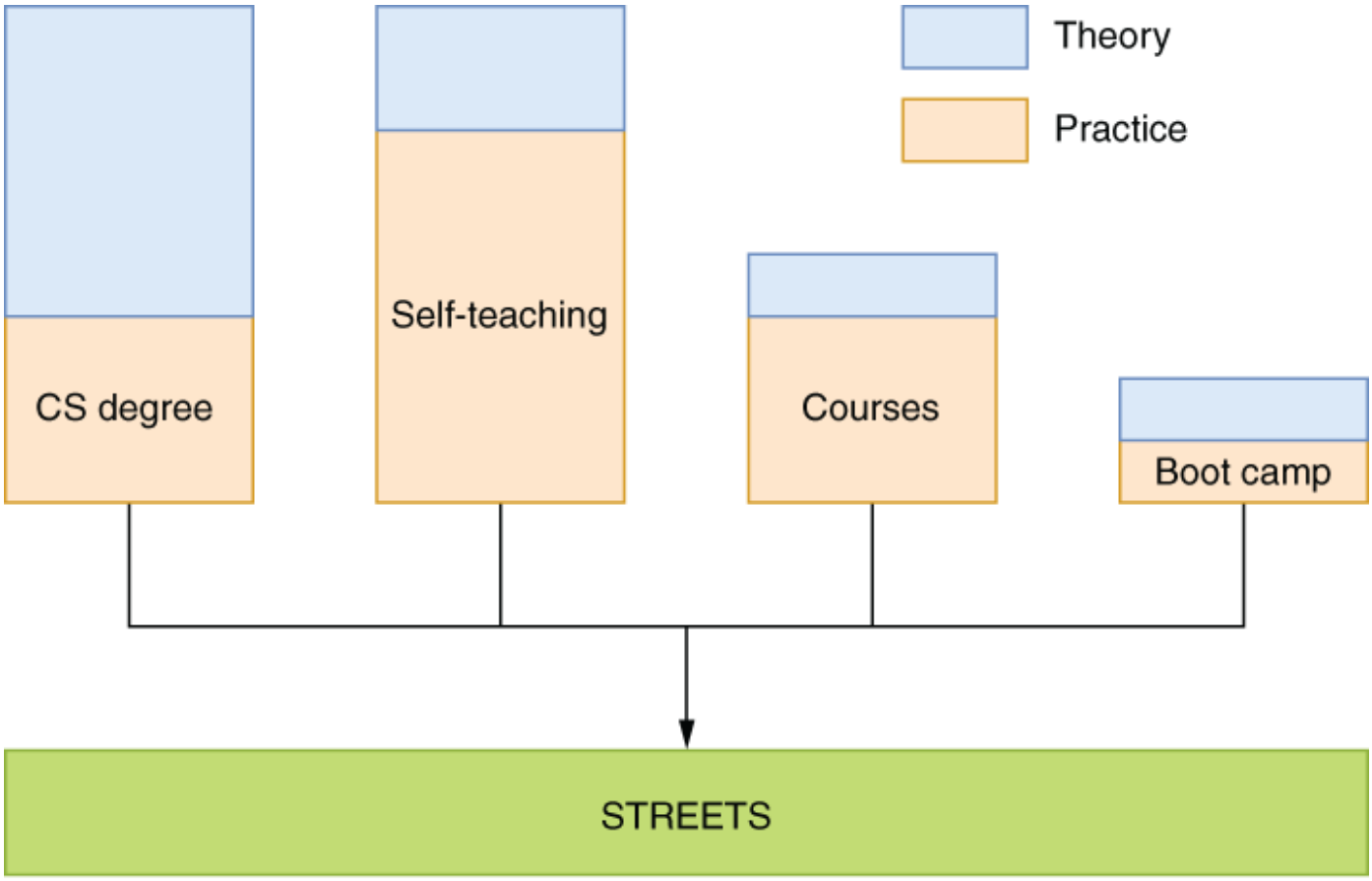
Podobnie dobry wzorzec projektowy czy algorytm może zwiększyć twoją wydajność. Jeśli nie pomaga w twojej wydajności, nie jest użyteczny. Ponieważ prawie wszystko można przyporządkować wartość pieniężną, wszystko, co robisz, można zmierzyć pod względem wydajności.

Możesz mieć wysoką wydajność przy złym kodzie, ale tylko w pierwszej iteracji. W chwili, gdy klient prosi o zmianę, zostajesz z utrzymaniem okropnego kodu. Przez całą tę książkę będę mówił o przypadkach, w których możesz zdać sobie sprawę, że wpadasz w pułapkę i wydostać się z niej, zanim stracisz zmysły.

1.2 Kim jest programista uliczny?

Microsoft rozważa dwie różne kategorie kandydatów podczas rekrutacji: absolwentów wydziałów informatyki i ekspertów branżowych, którzy posiadają znaczne doświadczenie w rozwoju oprogramowania.

Nie ważne, czy jesteś samoukiem czy ktoś, kto studiował informatykę, na początku swojej kariery brakuje ci wspólnego elementu: wiedzy ulicznej, czyli umiejętności wiedzenia, co jest najważniejsze. Samouk programista ma za sobą wiele prób i błędów, ale może brakować mu wiedzy na temat formalnej teorii i tego, jak można ją zastosować w codziennym programowaniu. Absolwent uczelni z kolei wie wiele o teorii, ale brakuje mu praktyczności i czasami podejścia krytycznego do tego, co się nauczył. Patrz rysunek 1.1.



Korpus, który zdobywasz w szkole, nie ma z nim związanej priorytetowości. Uczysz się według ścieżki edukacyjnej, a nie według ważności. Nie masz pojęcia, jakie pewne przedmioty mogą być użyteczne na ulicach, gdzie konkurencja jest bezlitosna. Terminy są nierealistyczne. Kawa jest zimna. Najlepszy framework na świecie ma ten jeden błąd, który sprawia, że tydzień twojej pracy idzie na marne. Twoja doskonale zaprojektowana abstrakcja rozpada się pod naciskiem klienta, który nieustannie zmienia swoje wymagania. Udaje ci się szybko zrefaktoryzować swój kod za pomocą kopiuj-wklej, ale teraz musisz edytować 15 różnych miejsc tylko po to, aby zmienić jedną wartość konfiguracji.

Przez lata zdobywasz nowe umiejętności, aby radzić sobie z niejasnością i złożonością. Samoucy programiści uczą się pewnych algorytmów, które im pomagają, a absolwenci uczelni w końcu rozumieją, że najlepsza teoria nie zawsze jest najbardziej praktyczna.

Programista uliczny to każdy z doświadczeniem w rozwoju oprogramowania w przemyśle, którego przekonania i teorie zostały ukształtowane przez rzeczywistość nierealistycznego szefa, który chciał, żeby tydzień pracy został zrobiony rano. Nauczyli się oni robić kopie zapasowe wszystkiego na wielu nośnikach po tym, jak stracili tysiące linii kodu i musieli to wszystko pisać od nowa. Widzieli migoczące światło C-beam w pomieszczeniu serwerowym spowodowane palącymi się dyskami twardymi i walczyli z administratorem systemu na drzwiach pomieszczenia serwerowego, tylko po to, aby uzyskać dostęp do produkcji, ponieważ ktoś właśnie wdrożył nieprzetestowany kawałek kodu. Testowali swój kod kompresji oprogramowania na własnym kodzie źródłowym, tylko po to, aby odkryć, że wszystko zostało skompresowane do jednego bajtu, a wartość tego bajtu wynosi 255. Alorytm dekompresji musi być jeszcze wynaleziony.

Właśnie skończyłeś studia i szukasz pracy, albo fascynujesz się programowaniem, ale nie masz pojęcia, co cię czeka. Wyszedłeś z obozu programistycznego i szukasz możliwości zatrudnienia, ale nie jesteś pewien, co brakuje ci w umiejętnościach. Nauczyłeś się samodzielnie jakiegoś języka programowania, ale nie wiesz, czego brakuje ci w swoim zestawie narzędzi umiejętności. Witaj na ulicach.

1.3 Wspaniali programiści uliczni

Oprócz reputacji na ulicy, honoru i lojalności, programista uliczny powinien idealnie posiadać te cechy:

1.3.1 Podejście Krytyczne

Osoba rozmawiająca z samą sobą jest uważana za nietypową, zwłaszcza jeśli nie ma odpowiedzi na pytania, które sobie zadaje. Jednakże bycie osobą krytyczną, zadawanie pytań sobie, kwestionowanie najbardziej powszechnie akceptowanych pojęć i ich dekonstrukcja mogą oczyścić twoją wizję.

Wiele książek, ekspertów od oprogramowania i Slavoj Žižek podkreśla wagę bycia krytycznym i dociekliwym, ale niewielu z nich dostarcza ci narzędzi do pracy. W tej książce znajdziesz przykłady bardzo znanych technik i najlepszych praktyk oraz jak mogą być mniej skuteczne, niż twierdzą.

Krytyka techniki nie oznacza, że jest ona bezużyteczna. Jednakże poszerzy twoje horyzonty, dzięki czemu będziesz w stanie zidentyfikować pewne przypadki użycia, w których alternatywna technika może być lepsza.

Celem tej książki nie jest omówienie każdej techniki programowania od podstaw do końca, ale przedstawienie ci perspektywy, jak traktować najlepsze praktyki, jak je priorytetyzować na podstawie wartości, i jak możesz rozważyć za i przeciw alternatywnych podejść.

1.3.2 Skoncentrowany na Wynikach

Możesz być najlepszym programistą na świecie, znającym wszystkie niuanse rozwoju oprogramowania, potrafiącym wymyślić najlepszy projekt dla własnego kodu, ale to nic nie znaczy, jeśli nie dostarczasz, jeśli nie wypuszczasz produktu.

Według paradoksu Zenona, aby osiągnąć cel końcowy, musisz najpierw osiągnąć punkt w połowie drogi. To paradoks, ponieważ aby osiągnąć punkt w połowie drogi, musisz osiągnąć punkt w ćwiartce drogi, i tak dalej, co sprawia, że nie możesz osiągnąć żadnego miejsca. Zenon miał rację: aby mieć produkt końcowy, musisz spełniać terminy i osiągać po drodze cele pośrednie. W przeciwnym razie niemożliwe jest osiągnięcie celu końcowego. Bycie skoncentrowanym na wynikach oznacza również koncentrację na celach pośrednich, na postępach.

„Jak projekt ma się stać rok po terminie? ... Dzień po dniu."

—Fred Brooks, The Mythical Man Month

Osiąganie wyników może oznaczać poświęcenie jakości kodu, elegancji i doskonałości technicznej. Ważne jest, aby mieć takie spojrzenie na rzeczywistość i zachować kontrolę nad tym, co robisz i dla kogo.

Poświęcanie jakości kodu nie oznacza poświęcania jakości produktu. Jeśli masz dobre testy, jeśli są dobre wymagania pisemne, możesz nawet pisać wszystko w PHP. Może to jednak oznaczać, że w przyszłości będziesz musiał ponieść pewne koszty, ponieważ kod niskiej jakości w końcu cię ugryzie. Nazywa się to karmą kodu.

Niektóre z technik, które przyswoisz z tej książki, pomogą ci podejmować decyzje w celu osiągnięcia wyników.

1.3.3 Wysoka Wydajność

Największymi czynnikami wpływającymi na szybkość twego rozwoju są doświadczenie, dobre i jasne specyfikacje oraz mechaniczne klawiatury. Tylko żartuję - wbrew powszechnemu przekonaniu, mechaniczne klawiatury wcale nie przyspieszają twojej pracy. Po prostu wyglądają fajnie i świetnie irytują twojego partnera. W rzeczywistości nie sądzę, że prędkość pisania ma cokolwiek wspólnego z szybkością rozwoju. Twoja pewność w prędkości pisania może nawet skłonić cię do pisania bardziej rozbudowanego kodu, niż jest to konieczne.

Część wiedzy można zdobyć, ucząc się na cudzych błędach i desperacji. W tej książce znajdziesz przykłady takich przypadków. Techniki i wiedza, które przyswoisz, sprawią, że napiszesz mniej kodu, podejmiesz szybsze decyzje i pozwoli ci mieć jak najmniejsze zadłużenie techniczne, aby nie spędzać dni na rozplątywaniu kodu, który napisałeś zaledwie pół roku temu.

1.3.4 Przyjmowanie Złożoności i Niejasności

Złożoność jest przerażająca, a niejasność jeszcze bardziej, ponieważ nie wiesz nawet, jak bardzo powinieneś się bać, co jeszcze bardziej cię przeraża.

Radzenie sobie z niejasnością to jedna z kluczowych umiejętności, o które pytają rekruterzy Microsoftu podczas rozmów kwalifikacyjnych. Zwykle obejmuje to hipotetyczne pytania, takie jak "Ile jest warsztatów naprawy skrzypiec w Nowym Jorku?", "Ile jest stacji benzynowych w Los Angeles?" lub "Ile agentów tajnej służby ma prezydent i jaki mają grafik zmianowy? Podaj ich imiona, a najlepiej pokaż ich ścieżki chodzenia na tym planie Białego Domu."

Sztuczka polega na rozjaśnianiu wszystkiego, co wiesz o problemie i opracowaniu przybliżenia na podstawie tych faktów. Na przykład możesz zacząć od ludności Nowego Jorku i od ilości osób, które mogą tam grać na skrzypcach. To da ci pojęcie o wielkości rynku i o tym, ile konkurencji może obsłużyć ten rynek.

Podobnie, gdy stajesz przed problemem z nieznanymi parametrami, takim jak oszacowanie czasu potrzebnego do opracowania funkcji, zawsze możesz zawęzić okno przybliżenia na podstawie tego, co wiesz. Możesz wykorzystać swoją wiedzę na swoją korzyść i jak najbardziej ją wykorzystać, co może zredukować niejasność do minimum.

Co ciekawe, radzenie sobie z złożonością jest podobne. Coś, co wydaje się niezwykle skomplikowane, można podzielić na części, które są znacznie bardziej zarządzalne, mniej złożone i ostatecznie prostsze.

Im więcej rzeczy wyjaśnisz, tym więcej będziesz w stanie uporać się z nieznanym. Techniki, które przyswoisz z tej książki, rozjaśnią niektóre z tych rzeczy i sprawią, że będziesz pewniejszy w radzeniu sobie z niejasnością i złożonością.

1.4 Problemy nowoczesnego rozwoju oprogramowania

Oprócz wzrastającej złożoności, licznych warstw abstrakcji i moderacji na Stack Overflow, nowoczesny rozwój oprogramowania ma również inne problemy:

1. Jest zbyt wiele technologii: zbyt wiele języków programowania, zbyt wiele frameworków, i z pewnością zbyt wiele bibliotek, biorąc pod uwagę, że npm (menedżer pakietów dla frameworka Node.js) miał bibliotekę o nazwie "left-pad" służącą tylko do dodawania spacji na końcu ciągu znaków.
2. Jest napędzany paradygmatami, a zatem konserwatywny. Wielu programistów uważa języki programowania, najlepsze praktyki, wzorce projektowe, algorytmy i struktury danych za relikty starożytnej obcej rasy i nie ma pojęcia, jak one działają.
3. Technologia staje się bardziej nieprzejrzysta, podobnie jak samochody. Ludzie kiedyś potrafili samodzielnie naprawiać swoje samochody. Teraz, gdy silniki stają się coraz bardziej zaawansowane, pod maską widzimy tylko metalową pokrywę, podobną do tej na grobowcu faraona, która uwolni przeklęte dusze na każdego, kto ją otworzy. Technologie w rozwoju oprogramowania są podobne. Mimo że prawie wszystko jest teraz open source, uważam, że nowe technologie są bardziej niejasne niż dekompilowane kody binarne z lat 90., ze względu na ogromnie wzrosłą złożoność oprogramowania.
4. Ludzie nie przejmują się nadmiernym obciążeniem swojego kodu, ponieważ mamy dostęp do rzeczywistości w skali dziesiątek tysięcy. Napisałeś nową prostą aplikację do czatu? Dlaczego by jej nie spakować z pełnoprawną przeglądarką internetową, bo wiesz, że to po prostu oszczędza ci czas, i nikt nie kiwnie powieką, kiedy i tak używasz gigabajtów pamięci?
5. Programiści skupiają się na swoim stosie technologicznym i ignorują, jak reszta działa, i słusznie: muszą zapewnić jedzenie na stole, i nie ma czasu na naukę. Nazywam to "problemem kuchennych programistów". Wiele rzeczy, które wpływają na jakość ich produktu, pozostaje niezauważonych ze względu na ograniczenia, które mają. Programista webowy zazwyczaj nie ma pojęcia, jak działają protokoły sieciowe pod warstwą internetu. Akceptują opóźnienia podczas ładowania strony takie, jakie są, i uczą się żyć z nimi, ponieważ nie wiedzą, że drobny techniczny szczegół, jak zbyt długa łańcuch certyfikatów, może spowolnić ładowanie strony internetowej.
6. Ze względu na nauczane paradygmaty, istnieje stigma przeciwko monotonnej pracy, takiej jak powtarzanie się czy kopiuj-wklej. Oczekuje się, że znajdziesz rozwiązanie DRY (Don't Repeat Yourself). Tego rodzaju kultura sprawia, że zaczynasz wątpić w siebie i swoje umiejętności, co szkodzi twojej produktywności.

> **HISTORIA NPM I LEFT-PAD**
>
> npm stało się w ostatniej dekadzie de facto ekosystemem pakietów JavaScript. Ludzie mogli przyczyniać się do tego ekosystemu swoimi własnymi pakietami, a inne pakiety mogły z nich korzystać, ułatwiając rozwijanie dużych projektów. Azer Koçulu był jednym z tych programistów. Left-pad był tylko jednym z pakietów spośród 250, które przyczynił się do ekosystemu npm. Miał tylko jedną funkcję: dodawać spacje do ciągu znaków, aby zawsze miał stały rozmiar, co jest dość trywialne.
>
> Pewnego dnia otrzymał e-mail od npm, w którym poinformowano go, że usunięto jeden z jego pakietów o nazwie "Kik", ponieważ firma o tej samej nazwie złożyła skargę. npm zdecydowało się usunąć pakiet Azer'a i przyznać nazwę innej firmie. To sprawiło, że Azer był tak zły, że usunął wszystkie swoje pakietu, w tym left-pad. Problem polegał na tym, że były setki projektów o dużych rozmiarach na całym świecie, które bezpośrednio lub pośrednio używały tego pakietu. Jego działania spowodowały zatrzymanie wszystkich tych projektów. To było dość katastrofalne i dobre nauczanie zaufania, jakie

> mamy do platform.
>
> Morał z tej historii jest taki, że życie na ulicy jest pełne niepożądanych niespodzianek.

W tej książce proponuję rozwiązania tych problemów, w tym przejście przez niektóre podstawowe koncepcje, które mogą ci się wydać nudne, priorytetyzowanie praktyczności, prostoty, odrzucanie niektórych długo utrzymywanych niepodważalnych przekonań i, co najważniejsze, kwestionowanie wszystkiego, co robimy. Wartość polega na zadawaniu pytań najpierw.

1.4.1 Zbyt wiele technologii

Nasze nieustanne poszukiwanie najlepszej technologii wynika z fałszywego przekonania o istnieniu srebrnej kuli. Myślimy, że istnieje technologia, która może zwiększyć naszą produktywność o rzędy wielkości. Nie istnieje. Na przykład Python7 to język interpretowany. Nie musisz kompilować kodu Pythona - działa od razu. Co więcej, nie musisz określać typów zmiennych, które deklarujesz, co sprawia, że jesteś jeszcze szybszy, więc Python musi być lepszą technologią niż C#, prawda? Niekoniecznie.

Ponieważ nie poświęcasz czasu na adnotowanie swojego kodu typami i kompilację, pomijasz popełniane błędy. Oznacza to, że możesz je odkryć tylko podczas testowania lub w produkcji, co jest znacznie droższe niż po prostu kompilacja kodu. Większość technologii to kompromisy, a nie zwiększacze produktywności. To, co zwiększa twoją produktywność, to jak biegły jesteś w tej technologii i jakie masz techniki, a nie to, jakie technologie używasz. Tak, są lepsze technologie, ale rzadko sprawiają one różnicę rzędu wielkości.

Kiedy chciałem stworzyć moją pierwszą interaktywną stronę internetową w 1999 roku, absolutnie nie miałem pojęcia, jak zacząć pisać aplikację internetową. Gdybym próbował najpierw znaleźć najlepszą technologię, musiałbym nauczyć się VBScript lub Perl. Zamiast tego użyłem tego, co najlepiej znałem wtedy: Pascal.8 Był to jeden z najmniej odpowiednich języków do tego celu, ale zadziałał. Oczywiście były z nim problemy. Za każdym razem, gdy się zawieszał, proces pozostawał aktywny w pamięci na losowym serwerze w Kanadzie, i użytkownik musiał dzwonić do dostawcy usług za każdym razem i prosić o ponowne uruchomienie fizycznego serwera. Mimo to Pascal pozwolił mi szybko osiągnąć prototyp, ponieważ czułem się z nim swobodnie. Zamiast uruchomić stronę internetową, którą sobie wyobrażałem po miesiącach pracy i nauki, napisałem i opublikowałem kod w trzy godziny.

Z niecierpliwością czekam, aby pokazać ci sposoby, dzięki którym możesz być bardziej efektywny, korzystając z istniejącego zestawu narzędzi, które masz pod ręką.

1.4.2 Paralotniarstwo na paradoksach

Najwcześniejszym paradygmatem programowania, z którym się spotkałem, był programowanie strukturalne w latach 80. Programowanie strukturalne polegało głównie na pisaniu kodu w strukturalnych blokach, takich jak funkcje i pętle, zamiast numerów linii, instrukcji GOTO czy wysiłku i łez. Sprawiało to, że kod był łatwiejszy do czytania i utrzymania, nie tracąc przy tym wydajności. Programowanie strukturalne zainteresowało mnie językami programowania takimi jak Pascal i C.

Następny paradygmat, z którym się zetknąłem, pojawił się co najmniej pół dekady po tym, jak nauczyłem się o programowaniu strukturalnym: programowanie obiektowe, czyli OOP (Object-Oriented Programming). Pamiętam, że wtedy magazyny komputerowe nie mogły się go doczekać. Była to kolejna wielka rzecz, która pozwoliła nam pisać jeszcze lepsze programy niż w przypadku programowania strukturalnego.

Po OOP spodziewałem się, że będę zetknął się z nowym paradygmatem co pięć lat lub coś w tym stylu. Jednakże zaczęły się one pojawiać częściej. W latach 90. poznaliśmy JIT-compiled9 zarządzane języki programowania dzięki nadejściu Javy, skrypty internetowe z użyciem JavaScript oraz programowanie funkcyjne, które powoli zaczęło przesuwać się do mainstreamu pod koniec lat 90.

Potem przyszły lata 2000. W następnych dekadach zaczęliśmy używać coraz częściej terminu aplikacje wielowarstwowe N-tier. Grube klienty. Cienkie klienty. Generyki. MVC, MVVM i MVP. Programowanie asynchroniczne zaczęło się rozpowszechniać dzięki obietnicom (promises), przyszłościom (futures) i ostatecznie programowaniu reaktywnemu. Mikroserwisy. Więcej koncepcji związanych z programowaniem funkcyjnym, takich jak LINQ, dopasowywanie wzorców (pattern matching) i niemutowalność, znalazło się w językach mainstreamowych. To jest tornado słów kluczowych.

Nawet nie wspomniałem jeszcze o wzorcach projektowych czy najlepszych praktykach. Mamy niezliczone najlepsze praktyki, porady i sztuczki dotyczące prawie każdego tematu. Napisano manifesty na temat tego, czy powinniśmy używać tabulacji czy spacji do wcięć w kodzie źródłowym, mimo że oczywista odpowiedź brzmi: spacje.10

Zakładamy, że nasze problemy można rozwiązać, stosując paradygmat, wzorzec, framework lub bibliotekę. Biorąc pod uwagę złożoność problemów, które teraz mamy do rozwiązania, to nie jest bezpodstawne. Jednak ślepe przyjmowanie tych narzędzi może przysporzyć więcej problemów w przyszłości: mogą spowolnić pracę, wprowadzając nową wiedzę dziedzinową do nauki oraz własne zestawy błędów. Mogą nawet zmusić cię do zmiany projektu. Ta książka sprawi, że będziesz bardziej pewny, że używasz wzorców poprawnie, podchodząc do nich bardziej dociekliwie, i będziesz gromadzić dobre argumenty do użycia podczas przeglądów kodu.

1.4.3 Czarne skrzynki technologii

Framework lub biblioteka to pakiet. Programiści instalują go, czytają jego dokumentację i używają. Ale zwykle nie wiedzą, jak działa. Tak samo podchodzą do algorytmów i struktur danych. Używają słownika, ponieważ wygodnie jest przechowywać klucze i wartości. Nie znają konsekwencji.

Niewzruszone zaufanie do ekosystemów pakietów i frameworków jest podatne na poważne błędy. Może nas to kosztować dni debugowania, ponieważ po prostu nie wiedzieliśmy, że dodawanie elementów do słownika o tym samym kluczu będzie różnić się od listy pod względem wydajności wyszukiwania. Używamy generatorów w C#, gdy wystarczyłby prosty tablica, i doznajemy znacznego pogorszenia wydajności, nie wiedząc dlaczego.

Pewnego dnia w 1993 roku przyjaciel podał mi kartę dźwiękową i poprosił, żebym zainstalował ją w moim komputerze PC. Tak, kiedyś potrzebowaliśmy dodatkowych kart, żeby uzyskać porządny dźwięk z PC, bo inaczej słyszeliśmy tylko beep. W każdym razie nigdy wcześniej nie otwierałem obudowy swojego komputera, i bałem się, że coś zniszczę. Powiedziałem mu: "Czy nie możesz tego zrobić za mnie?" Mój przyjaciel powiedział mi: "Musisz to otworzyć, żeby zobaczyć, jak to działa."

To rezonowało we mnie, ponieważ zrozumiałem, że moje lęki wynikały z mojej niewiedzy, a nie z mojej niezdolności. Otwarcie obudowy i zobaczenie wnętrza mojego własnego komputera mnie uspokoiło. Zawierał tylko kilka płyt. Karta dźwiękowa wchodziła do jednego z gniazd. To dla mnie już nie było tajemnicze pudełko. Później użyłem tej samej techniki, ucząc studentów szkoły artystycznej podstaw komputerów. Otworzyłem myszkę i pokazałem im jej kulkę. Myszy miały kule wtedy. No cóż, to było niestety dwuznaczne. Otworzyłem obudowę komputera. "Widzicie, to nie jest straszne, to jest płyta główna i kilka

gniazd."

To później stało się moim mottem w radzeniu sobie z czymś nowym i skomplikowanym. Przestałem się bać otwierać pudełko i zwykle robiłem to jako pierwszą rzecz, żeby móc zmierzyć się z pełnym zakresem złożoności, która zawsze była mniejsza, niż się jej obawiałem.

Podobnie szczegóły tego, jak działa biblioteka, framework czy komputer, mogą mieć ogromny wpływ na twoje zrozumienie tego, co jest na nich zbudowane. Otwarcie pudełka i przyjrzenie się częściom może pomóc ci w prawidłowym użyciu pudełka. Nie musisz od razu czytać kodu od początku do końca ani przechodzić przez tysiącstronicową książkę teorii, ale przynajmniej powinieneś wiedzieć, która część idzie gdzie i jak może wpływać na twoje przypadki użycia.

Dlatego niektóre z tematów, o których będę mówił, są fundamentalne lub niskopoziomowe. Chodzi o otwarcie pudełka i zobaczenie, jak to działa, żebyśmy mogli podejmować lepsze decyzje w programowaniu na wysokim poziomie.

1.4.4 Zbyt lekceważony nadmiar

Cieszę się, że każdego dnia widzimy coraz więcej aplikacji opartych na chmurze. Są one nie tylko opłacalne, ale także rzeczywistym sprawdzianem dla zrozumienia rzeczywistych kosztów naszego kodu. Kiedy zaczynasz płacić dodatkowego centa za każdą złą decyzję w kodzie, nadmiar nagle staje się problemem.

Frameworki i biblioteki zwykle pomagają nam unikać nadmiaru, co czyni je użytecznymi abstrakcjami. Niemniej jednak nie możemy przekazywać całego naszego procesu podejmowania decyzji frameworkom. Czasami musimy sami podejmować decyzje i musimy uwzględnić nadmiar. W przypadku aplikacji o dużym zasięgu nadmiar staje się jeszcze bardziej istotny. Każdy millisekundy, który oszczędzisz, może pomóc odzyskać cenne zasoby.

Priorytetem programisty nie powinno być eliminowanie nadmiaru. Niemniej jednak wiedza o tym, jak unikać nadmiaru w określonych sytuacjach i perspektywa jako narzędzie w twoim arsenale pomogą ci zaoszczędzić czas, zarówno dla siebie, jak i dla użytkownika, który czeka na ten kręcący się spinner na twojej stronie internetowej.

W trakcie lektury książki znajdziesz scenariusze i przykłady, jak można łatwo unikać nadmiaru, nie robiąc z tego swojego najważniejszego celu.

1.4.5 To nie moja praca

Jednym ze sposobów radzenia sobie z złożonością jest skupienie się wyłącznie na swoich obowiązkach: na komponencie, który posiadasz, na kodzie, który piszesz, na błędach, które popełniłeś, i czasami na rozsadzanym lasagne w mikrofalówce w biurze. Może się to wydawać najbardziej efektywnym sposobem wykonywania pracy, ale jak wszelkie istoty, tak i kod jest ze sobą powiązany.

Poznanie, jak działa określona technologia, jak biblioteka wykonuje swoją pracę i jakie są zależności oraz jak są ze sobą połączone, pozwala nam podejmować lepsze decyzje, gdy piszemy kod. Przykłady w tej książce dadzą ci perspektywę skupienia się nie tylko na swojej dziedzinie, ale także na jej zależnościach i problemach, które wykraczają poza twoją strefę komfortu, ponieważ odkryjesz, że one przewidują los twojego kodu.

1.4.6 Powszednie jest genialne

Wszystkie zasady nauczane w rozwoju oprogramowania sprowadzają się do jednego napomnienia: spędzaj mniej czasu wykonując swoją pracę. Unikaj powtarzających się, bezmyślnych zadań, takich jak kopiowanie i wklejanie oraz pisanie tego samego kodu od zera z drobnymi zmianami. Po pierwsze, zajmują one więcej czasu, a po drugie, jest niezwykle trudno je utrzymać.

Nie wszystkie zadania powszednie są złe. Nawet kopiowanie i wklejanie nie są złe. Istnieje silne piętno wobec nich, ale są sposoby, aby uczynić je bardziej wydajnymi niż niektóre najlepsze praktyki, które cię nauczono.

Ponadto nie cały kod, który piszesz, działa jako kod rzeczywistego produktu. Niektóry kod, który piszesz, będzie używany do opracowywania prototypu, niektóry będzie służył do testów, a niektóry będzie cię przygotowywać do właściwego zadania. Omówię niektóre z tych scenariuszy i jak możesz wykorzystać te zadania na swoją korzyść.

1.5 Czym książka nie jest

Ta książka nie jest kompleksowym przewodnikiem po programowaniu, algorytmach ani żadnym innym zagadnieniu. Nie uważam się za eksperta w konkretnych tematach, ale mam wystarczającą wiedzę na temat rozwoju oprogramowania. Książka składa się głównie z informacji, które nie są oczywiste w popularnych i znakomitych książkach dostępnych na rynku. To zdecydowanie nie jest przewodnikem do nauki programowania.

Doświadczeni programiści mogą znaleźć niewielkie korzyści z tej książki, ponieważ zdobyli już wystarczającą wiedzę i stali się już programistami znającymi ulice. Niemniej jednak mogą być zaskoczeni niektórymi spostrzeżeniami zawartymi w tej książce.

Ta książka to także eksperyment w dziedzinie tego, jak książki programistyczne mogą być zabawne w czytaniu. Chciałbym przede wszystkim przedstawić programowanie jako zabawę. Książka nie traktuje siebie zbyt poważnie, więc ty też nie powinieneś. Jeśli po przeczytaniu książki poczujesz się lepszym programistą i będziesz się dobrze bawić, uznaję siebie za spełnionego.

1.6 Tematy

Pewne tematy będą się powtarzać w całej książce:

1. Minimalna wiedza podstawowa, która wystarcza, aby poradzić sobie na ulicach. Te tematy nie będą wyczerpujące, ale mogą zainteresować cię, jeśli wcześniej wydawały ci się nudne. To zazwyczaj kluczowa wiedza, która pomaga podjąć decyzje.
2. Powszechnie znane lub akceptowane najlepsze praktyki lub techniki, które ja proponuję jako antywzorce, które w pewnych przypadkach mogą być bardziej efektywne. Im więcej przeczytasz na ich temat, tym bardziej zostanie wyostrzony twój szósty zmysł do krytycznego myślenia o praktykach programistycznych.
3. Pozornie nieistotne techniki programowania, takie jak triki optymalizacji na poziomie CPU, które mogą wpływać na twoje decyzje i pisanie kodu na wyższym poziomie. Znajomość wewnętrznych mechanizmów, "otwieranie pudełka", ma ogromną wartość, nawet jeśli nie korzystasz bezpośrednio z tej wiedzy.

4. Techniki, które uważam za przydatne w mojej codziennej pracy programisty, które mogą ci pomóc zwiększyć swoją produktywność, w tym obgryzanie paznokci i stawanie się niewidzialnym dla swojego szefa.

Te tematy będą podkreślać nową perspektywę, kiedy będziesz przyglądać się zagadnieniom programistycznym, zmieniać twoje zrozumienie pewnych "nudnych" tematów i być może zmieniać twoje podejście do pewnych dogmatów. Pomogą ci czerpać radość z pracy.

Podsumowanie

Surowa rzeczywistość "ulic", czyli świata profesjonalnego rozwoju oprogramowania, wymaga umiejętności, które nie są nauczane lub nie są priorytetem w formalnym kształceniu, a czasem zupełnie pomijane podczas samodzielnej nauki.

Nowi programiści często albo zwracają uwagę na teorię, albo zupełnie ją ignorują. Ostatecznie znajdziesz złoty środek, ale można go przyspieszyć pewną perspektywą.

Współczesny rozwój oprogramowania jest znacznie bardziej złożony niż kilka dekad temu. Aby stworzyć prostą działającą aplikację, wymaga się ogromnej wiedzy na wielu poziomach.

Programiści stoją przed dylematem między tworzeniem oprogramowania a nauką. Można to przezwyciężyć, zmieniając sposób postrzegania tematów w sposób bardziej pragmatyczny.

Brak jasności co do tego, nad czym pracujesz, sprawia, że programowanie staje się nudnym zadaniem, co w rezultacie obniża twoją rzeczywistą produktywność. Lepsze zrozumienie tego, czym się zajmujesz, przyniesie ci więcej radości.

1. Haskell to ezoteryczny język programowania, który został stworzony jako wyzwanie, aby zmieścić jak najwięcej prac naukowych w jednym języku programowania.
2. Linus Torvalds stworzył system operacyjny Linux i oprogramowanie do kontroli wersji Git oraz poparł przekonanie, że przeklinanie wolontariuszy pracujących nad projektem jest w porządku, jeśli są technicznie w błędzie.
3. Slavoj Žižek to współczesny filozof, który cierpi na schorzenie, które zmusza go do krytykowania wszystkiego na świecie, bez wyjątków.
4. Zeno był starożytnym Grekiem, który żył tysiące lat temu i nie mógł przestać zadawać frustrujących pytań. Naturalnie żadne z jego pism nie przetrwały.
5. PHP był kiedyś językiem programowania, który stanowił przykład tego, jak nie projektować języka programowania. O ile wiem, PHP przeszedł długą drogę od czasów, gdy był obiektem żartów programistycznych, i teraz jest fantastycznym językiem programowania. Niemniej jednak wciąż ma pewne problemy z wizerunkiem marki do rozwiązania.
6. DRY. Nie powtarzaj się. Przesąd, że jeśli ktoś powtarza linię kodu zamiast owijać ją w funkcję, natychmiast zostanie przekształcony w żabę.
7. Python to kolektywna próba promowania białych znaków, udająca praktyczny język programowania.
8. Wczesny kod źródłowy Ekşi Sözlük jest dostępny na GitHubie: https://github.com/ssg/sozluk-cgi.
9. JIT, kompilacja "just-in-time". Mityczne pojęcie stworzone przez firmę Sun Microsystems, twórcę języka Java, że jeśli skompilujesz kod podczas jego działania, stanie się szybszy, ponieważ optymalizator będzie miał więcej danych zebranych podczas działania. To nadal mit.

10. Napisałem o debacie dotyczącej używania tabulatorów czy spacji z pragmatycznego punktu widzenia: https://medium.com/ @ssg/tabs-vs-spaces-towards-a-better-bike-shed-686e111a5cce.
11. Spinnery są współczesnymi klepsydrami w świecie komputerów. W czasach starożytnych, komputery używały klepsydr do sprawienia, żebyś czekał przez nieokreślony czas. Spinner to nowoczesny odpowiednik tej animacji. Zazwyczaj jest to nieskończony obracający się łuk. To tylko dystrakcja, która pozwala utrzymać frustrację użytkownika pod kontrolą.

Praktyczna teoria

Rozdział ten obejmuje:

1. **Dlaczego teoria informatyki jest istotna dla twojego przetrwania**
2. **Jak wykorzystać typy danych na swoją korzyść**
3. **Zrozumienie cech algorytmów**
4. **Struktury danych i ich dziwaczne cechy, o których twoi rodzice zapomnieli ci powiedzieć**

Wbrew powszechnie przyjętemu przekonaniu, programiści są ludźmi. Mają te same błędy poznawcze co inni ludzie w praktyce tworzenia oprogramowania. Szeroko przeceniają korzyści płynące z pomijania typów danych, niezwracania uwagi na poprawne struktury danych czy zakładania, że algorytmy są ważne jedynie dla autorów bibliotek.

Ty nie jesteś wyjątkiem. Oczekuje się od ciebie, że dostarczysz produkt na czas, z dobrą jakością, i z uśmiechem na twarzy. Jak mówi przysłowie, programista to efektywny organizm, który otrzymuje kawę jako wejście i tworzy oprogramowanie jako wyjście. Możesz równie dobrze pisać wszystko na najgorszy możliwy sposób, używać kopiuj-wklej, korzystać z kodu znalezionego na Stack Overflow, używać plików tekstowych do przechowywania danych, albo nawet zawierać pakt z demonem, jeśli twoja dusza jeszcze nie podpisała umowy o poufności.1 Tylko twoi rówieśnicy naprawdę przejmują się tym, jak wykonujesz swoją pracę – wszyscy inni chcą po prostu, aby produkt działał i był dobry.

Teoria może być przytłaczająca i niezwiązana z rzeczywistością. Algorytmy, struktury danych, teoria typów, notacja Big-O i złożoność wielomianowa mogą wydawać się skomplikowane i nieistotne w kontekście tworzenia oprogramowania. Istniejące biblioteki i frameworki już zajmują się tym w zoptymalizowany i przetestowany sposób. Zazwyczaj jest zalecane, aby nigdy nie implementować algorytmu od zera, zwłaszcza w kontekście bezpieczeństwa informacji lub napiętych terminów.

To dlaczego powinieneś interesować się teorią? Ponieważ wiedza z dziedziny informatyki pozwala ci nie tylko rozwijać algorytmy i struktury danych od podstaw, ale także właściwie określać, kiedy należy zastosować daną technikę. Pomaga zrozumieć koszty podejmowanych decyzji kompromisowych. Pomaga zrozumieć charakterystyki skalowalności kodu, który piszesz. Pozwala patrzeć w przyszłość. Prawdopodobnie nigdy nie będziesz implementować struktury danych ani algorytmu od podstaw, ale wiedza na temat ich działania uczyni cię efektywnym programistą. Poprawi twoje szanse na przetrwanie na "ulicach".

Ta książka omówi tylko pewne kluczowe aspekty teorii, które mogłeś przeoczyć w czasie nauki - niektóre mniej znane aspekty typów danych, zrozumienie złożoności algorytmów i sposób działania pewnych struktur danych. Jeśli wcześniej nie uczyłeś się o typach danych, algorytmach lub strukturach danych, ten rozdział dostarczy ci wskazówek, które mogą zainteresować cię tym tematem.

2.1 Krótki kurs na temat algorytmów

Algorytm to zestaw reguł i kroków mających na celu rozwiązanie problemu. Dziękuję za udział w moim wykładzie TED. Spodziewałeś się bardziej skomplikowanej definicji, prawda? Na przykład przeglądanie elementów tablicy w celu sprawdzenia, czy zawiera określoną liczbę, jest algorytmem, nawet jeśli prostym:

```
public static bool Contains(int[] array, int lookFor) {
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

Moglibyśmy nazwać to "Algorytmem Sedata", gdybym to ja był osobą, która go wynalazła, ale prawdopodobnie był to jeden z pierwszych algorytmów, które kiedykolwiek powstały. Nie jest w żaden sposób wymyślny, ale działa i ma sens. To jeden z istotnych punktów dotyczących algorytmów: muszą one działać zgodnie z twoimi potrzebami. Niekoniecznie muszą zdziałać cuda. Kiedy wkładasz naczynia do zmywarki i ją uruchamiasz, również stosujesz algorytm. Istnienie algorytmu nie oznacza, że jest on inteligentny.

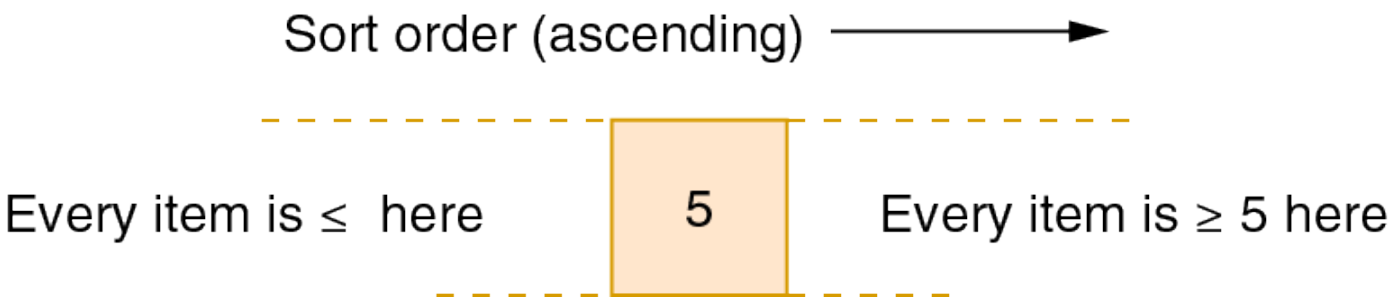
Mając powiedziane, mogą istnieć bardziej zaawansowane algorytmy, w zależności od twoich potrzeb. W poprzednim przykładzie kodu, jeśli wiesz, że lista zawiera tylko liczby całkowite, możesz dodać specjalne obsługi dla liczb niebędących dodatnimi:

```
public static bool Contains(int[] array, int lookFor) {
   if (lookFor < 1) {
       return false;
   }
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

To może znacznie przyspieszyć działanie algorytmu, w zależności od tego, jak często jest wywoływany z liczbą ujemną. W najlepszym przypadku twoja funkcja zawsze byłaby wywoływana z liczbami ujemnymi lub zerami, zwracając od razu, nawet jeśli tablica miała miliardy liczb całkowitych. W najgorszym przypadku twoja funkcja byłaby zawsze wywoływana z liczbami dodatnimi, a ty miałbyś tylko jedno dodatkowe, zbędne sprawdzenie. Typy danych mogą ci tu pomóc, ponieważ w języku C# istnieją bezznakowe wersje liczb całkowitych, nazywane uint. Dzięki nim możesz zawsze otrzymywać liczby dodatnie, a kompilator będzie sprawdzał to za ciebie, nie wprowadzając żadnych problemów z wydajnością:

```
public static bool Contains(uint[] array, uint lookFor) {
   for (int n = 0; n < array.Length; n++) {
       if (array[n] == lookFor) {
           return true;
       }
   }
   return false;
}
```

Naprawiliśmy wymaganie dotyczące liczb dodatnich, stosując restrykcje typów, zamiast zmieniać nasz algorytm, ale nadal może być on szybszy w zależności od kształtu danych. Czy posiadamy więcej informacji o danych? Czy tablica jest posortowana? Jeśli tak, możemy przypuszczać więcej o miejscu, w którym może znajdować się nasza liczba. Porównując naszą liczbę z dowolnym elementem w tablicy, możemy łatwo wykluczyć ogromną ilość elementów (zobacz rysunek 2.1).



Jeśli nasza liczba wynosi na przykład 3, i porównujemy ją z 5, możemy być pewni, że nasza liczba nie znajdzie się po prawej stronie od 5. To oznacza, że możemy natychmiast wyeliminować wszystkie elementy znajdujące się po prawej stronie listy.

Zatem, jeśli wybierzemy element ze środka listy, możemy zagwarantować, że po porównaniu możemy natychmiast wyeliminować co najmniej połowę listy. Możemy zastosować tę samą logikę do pozostałej części, wybierając środkowy punkt i kontynuując. Oznacza to, że potrzebujemy co najwyżej 3 porównań do posortowanej tablicy z 8 elementami, aby stwierdzić, czy dany element istnieje w niej. Co ważniejsze, zajmie to maksymalnie około 10 przeszukań, aby stwierdzić, czy dany element istnieje w tablicy z 1000 elementami. To jest moc, którą uzyskujesz, dzieląc na pół. Twoja implementacja mogłaby wyglądać jak w przykładzie 2.1. W zasadzie ciągle znajdujemy środkowe miejsce i eliminujemy pozostałą połowę, zależnie od tego, jak wartość, której szukamy, wpasowuje się w nią. Zapisujemy formułę w dłuższej, bardziej rozbudowanej formie, chociaż odpowiada ona (start + koniec) / 2. To dlatego, że start + koniec może przekroczyć wartość dla dużych wartości startu i końca i doprowadzić do znalezienia nieprawidłowego środka. Jeśli zapiszesz wyrażenie tak, jak w poniższym przykładzie, unikniesz tego przypadku przepełnienia.

Przykład 2.1 Wyszukiwanie w posortowanej tablicy za pomocą wyszukiwania binarnego

```
public static bool Contains(uint[] array, uint lookFor) {
  int start = 0;
  int end = array.Length - 1;
  while (start <= end) {
    int middle = start + ((end - start) / 2);
    uint value = array[middle];
    if (lookFor == value) {
      return true;
    }
```

```
    if (lookFor > value) {
      start = middle + 1;
    } else {
      end = middle - 1;
    }
  }
  return false;
}
```

Tutaj zaimplementowaliśmy wyszukiwanie binarne, znacznie szybszy algorytm niż Algorytm Sedata. Teraz, gdy możemy sobie wyobrazić, jak wyszukiwanie binarne może być szybsze niż zwykła iteracja, możemy zacząć myśleć o szanowanej notacji Big-O.

2.1.1 Big-O musi być dobrze zrozumiane

Zrozumienie wzrostu to doskonała umiejętność dla programisty. Bez względu na to, czy chodzi o rozmiar czy liczbę, kiedy wiesz, jak szybko coś rośnie, możesz zobaczyć przyszłość i ocenić, w jakiego rodzaju kłopoty się wpakowujesz, zanim na to zbyt dużo czasu poświęcisz. Jest to szczególnie przydatne, gdy światło na końcu tunelu rośnie, choć ty się nie poruszasz.

Notacja Big-O, jak sama nazwa wskazuje, jest po prostu sposobem wyjaśnienia wzrostu, ale również podlega nieporozumieniom. Kiedy pierwszy raz zobaczyłem O(N), myślałem, że to zwykła funkcja, która powinna zwracać liczbę. Ale nie jest. To sposób, w jaki matematycy opisują wzrost. Daje nam podstawowy pomysł na to, jak skalowalny jest algorytm. Przechodzenie przez każdy element sekwencyjnie (zwane także Algorytmem Sedata) wymaga liczby operacji liniowo proporcjonalnej do liczby elementów w tablicy. Oznaczamy to, pisząc O(N), gdzie N oznacza liczbę elementów. Nadal nie możemy dokładnie określić, ile kroków algorytm wykona, patrząc tylko na O(N), ale wiemy, że wzrasta to liniowo. Pozwala nam to dokonywać założeń na temat charakterystyki wydajności algorytmu w zależności od wielkości danych. Możemy przewidzieć, w którym momencie może stać się to problemem, patrząc na to.

Wyszukiwanie binarne, które zaimplementowaliśmy, ma złożoność O(log2n). Jeśli nie jesteś zaznajomiony z logarytmami, jest to przeciwieństwo funkcji wykładniczej, więc złożoność logarytmiczna jest naprawdę wspaniała, chyba że chodzi o pieniądze. W tym przykładzie, jeśli nasz algorytm sortowania magicznie miałby złożoność logarytmiczną, potrzebowałby tylko 18 porównań, aby posortować tablicę z 500 000 elementów. Nasza implementacja wyszukiwania binarnego jest więc świetna.

Notacja Big-O nie służy tylko do pomiaru wzrostu kroków obliczeniowych, czyli złożoności czasowej, ale również do pomiaru wzrostu zużycia pamięci, co nazywane jest złożonością przestrzeniową. Algorytm może być szybki, ale może mieć wzrost wielomianowy w pamięci, jak w naszym przykładzie sortowania. Powinniśmy zrozumieć tę różnicę.

> PORADA
>
> Wbrew powszechnemu przekonaniu, O(Nx) nie oznacza złożoności wykładniczej. Oznacza to złożoność wielomianową, która, choć dosyć zła, nie jest tak straszna jak złożoność wykładnicza, która jest oznaczana jako O(xn). Przy zaledwie 100 elementach O(N2) wykona 10 000 iteracji, podczas gdy O(2n) wykona jakąś nieprawdopodobną liczbę iteracji z 30 cyframi—nawet nie jestem w stanie tego wymówić. Istnieje także złożoność silniowa, która jest jeszcze gorsza niż wykładnicza, ale nie widziałem żadnych algorytmów, które by ją używały, poza obliczaniem permutacji lub kombinacji,

> prawdopodobnie dlatego, że nikt nie był w stanie jej wynaleźć do końca.

Algorytm wyszukiwania	**Złożoność**	**Czas znalezienia rekordu wśród 60 wierszy**
Domowy kwantowy komputer, który wujek Lisy ma w swoim garażu	O(1)	1 sekunda
Wyszukiwanie binarne	O(log N)	6 sekund
Wyszukiwanie liniowe (ponieważ szef poprosił cię o to godzinę przed prezentacją)	O(N)	60 sekund
Praktykant przypadkowo dodał zagnieżdżone dwie pętle.	O(N^2)	1 godzina
Jakiś kod wklejony przypadkowo ze Stack Overflow, który znajduje również rozwiązanie pewnego problemu szachowego podczas wyszukiwania, ale programista nie zadał sobie trudu, aby to usunąć	O(2^N)	36,5 miliarda lat
Zamiast znalezienia rzeczywistego rekordu, algorytm próbuje znaleźć układ rekordów, które ułożone w pewien sposób tworzą szukany rekord. Dobra wiadomość polega na tym, że ten programista już tu nie pracuje.	O(N!)	Koniec wszechświata, ale nadal przed tym, zanim małpy skończą swoje tak zwane "Shakespeare'y"

Musisz być zaznajomiony z tym, jak notacja Big-O opisuje wzrost prędkości wykonania algorytmu i zużycia pamięci, abyś mógł podejmować świadome decyzje podczas wyboru struktury danych i algorytmu do użycia. Zapoznaj się z notacją Big-O, nawet jeśli nie musisz implementować algorytmu. Uważaj na złożoność.

2.2 Struktury danych od środka

Na początku była pustka. Kiedy pierwsze sygnały elektryczne trafiły do pierwszego bitu w pamięci, zrodziły się dane. Dane były początkowo wolno unoszącymi się bajtami. Te bajty zeszły się razem i stworzyły strukturę.

— Początek 0:1

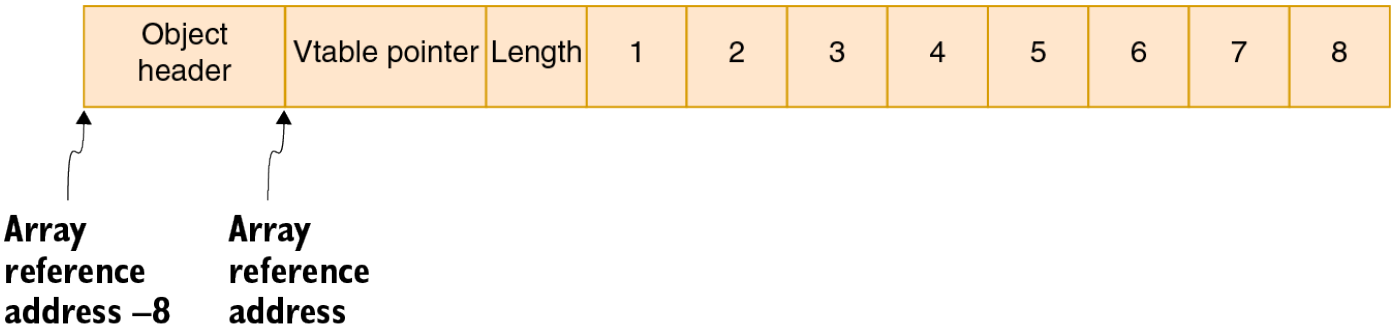
Struktury danych dotyczą tego, jak dane są rozmieszczone. Ludzie odkryli, że gdy dane są rozmieszczone w określony sposób, mogą być bardziej przydatne. Lista zakupów na kartce jest łatwiejsza do czytania, jeśli każdy przedmiot znajduje się w osobnej linii. Tabela mnożenia jest bardziej użyteczna, jeśli jest ułożona w siatkę. Zrozumienie, jak działa określona struktura danych, jest kluczowe dla twojego rozwoju jako programisty. To zrozumienie zaczyna się od zerknięcia pod maskę i przyjrzenia się, jak struktura działa.

Przyjrzyjmy się na przykład tablicom. Tablica w programowaniu to jedna z najprostszych struktur danych, a jest rozmieszczana jako ciągłe elementy w pamięci. Załóżmy, że masz taką tablicę:

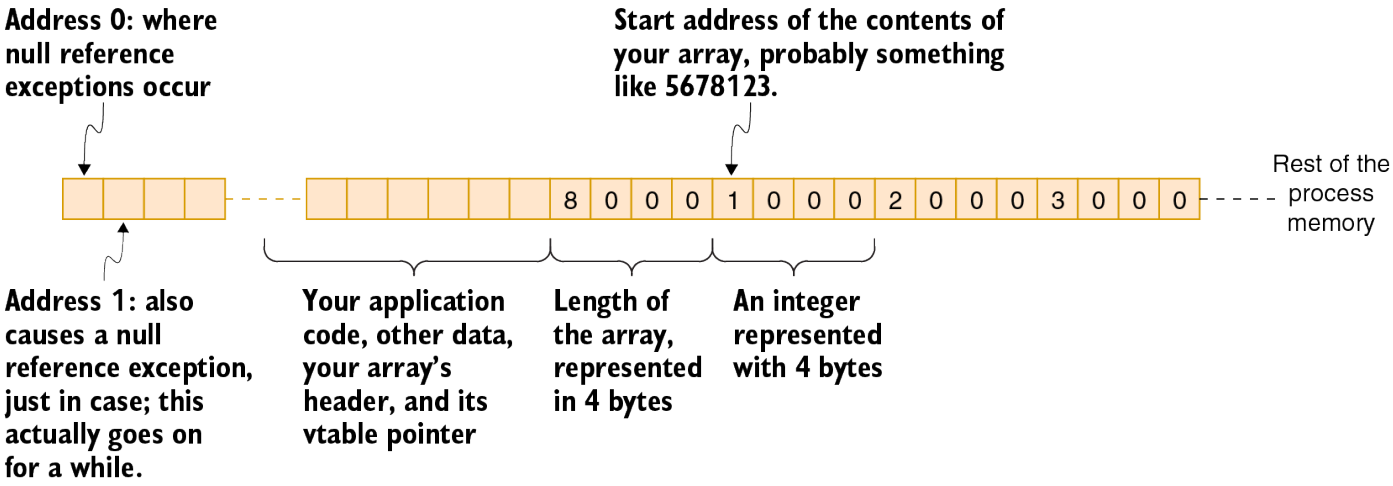
```
1 var wartosci = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };
```

Możesz sobie wyobrazić, jak wyglądałaby w pamięci, jak na rysunku 2.2.

Tak naprawdę nie wyglądałoby to w pamięci w taki sposób, ponieważ każdy obiekt w .NET ma określony nagłówek, wskaźnik do tablicy wskaźników do tabeli wirtualnych metod oraz informacje o długości, jak pokazano na rysunku 2.3.



Staje się to jeszcze bardziej realistyczne, gdy spojrzysz na to, jak jest umieszczone w pamięci RAM, ponieważ RAM nie jest zbudowany z pojedynczych liczb całkowitych, jak pokazano na rysunku 2.4. Dzielę się tymi informacjami, ponieważ chcę, abyś nie bał się tych niskopoziomowych koncepcji. Ich zrozumienie pomoże ci na wszystkich poziomach programowania.



To nie jest wygląd twojej rzeczywistej pamięci RAM, ponieważ każdy proces ma swój własny fragment pamięci do niego dedykowany, zgodnie z tym, jak działają nowoczesne systemy operacyjne. Ale ten układ będzie zawsze mieć z nim do czynienia, chyba że zaczniesz tworzyć swój własny system operacyjny lub własne sterowniki urządzeń.

Ogólnie rzecz biorąc, sposób, w jaki dane są rozmieszczone, może sprawić, że rzeczy będą działać szybciej lub bardziej efektywnie, lub wręcz przeciwnie. Ważne jest, aby znać podstawowe struktury danych i jak działają wewnętrznie.

2.2.1 Ciągi znaków

Ciągi znaków mogą być najbardziej przyjaznym typem danych w świecie programowania. Reprezentują tekst i zazwyczaj mogą być czytelne dla ludzi. Nie powinieneś używać ciągów znaków, gdy inny typ lepiej się nadaje, ale są one nieuniknione i wygodne. Korzystając z ciągów znaków, musisz znać kilka podstawowych faktów na ich temat, które nie są oczywiste od razu.

Mimo że są podobne do tablic pod względem użycia i struktury, ciągi znaków w .NET są niemodyfikowalne. Niemodyfikowalność oznacza, że zawartość struktury danych nie może być zmieniana po jej zainicjowaniu. Załóżmy, że chcemy połączyć imiona osób, aby utworzyć pojedynczy ciąg znaków oddzielony przecinkami i że wróciliśmy dwie dekady wstecz w czasie, więc nie ma lepszego sposobu na wykonanie tego zadania:

```
public static string JoinNames(string[] names) {
    string result = String.Empty;
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        result += names[i] + ", ";
    }
    result += names[lastIndex];
    return result;
}
```

Na pierwszy rzut oka może się wydawać, że mamy ciąg znaków o nazwie result i modyfikujemy ten sam ciąg znaków podczas wykonywania operacji, ale tak nie jest. Za każdym razem, gdy przypisujemy zmiennej result nową wartość, tworzony jest nowy ciąg znaków w pamięci. .NET musi określić długość nowego ciągu znaków, zaalokować nową pamięć na jego potrzeby, skopiować zawartość innych ciągów znaków do nowo zbudowanej pamięci i zwrócić ci go. To dość kosztowna operacja, a koszt rośnie w miarę wydłużania się ciągu znaków oraz śladu śmieci do zebrania.

W frameworku są narzędzia, które pozwalają unikać tego problemu. Nawet jeśli nie zależy ci na wydajności, te narzędzia są darmowe, więc nie musisz zmieniać swojej logiki ani starać się osiągnąć lepszą wydajność. Jednym z nich jest StringBuilder, z którym możesz pracować, aby budować ostateczny ciąg znaków i uzyskać go za pomocą wywołania ToString w jednym kroku:

```
public static string JoinNames(string[] names) {
    var builder = new StringBuilder();
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        builder.Append(names[i]);
        builder.Append(", ");
    }
    builder.Append(names[lastIndex]);
    return builder.ToString();
}
```

StringBuilder wewnętrznie używa kolejnych bloków pamięci zamiast realokować i kopiować za każdym razem, gdy musi zwiększyć rozmiar ciągu znaków. Dlatego zazwyczaj jest bardziej wydajny niż budowanie ciągu znaków od zera.

Oczywiście od dłuższego czasu dostępne jest idiomatyczne i znacznie krótsze rozwiązanie, ale twoje przypadki użycia nie zawsze muszą pokrywać się z tymi przypadkami:

```
String.Join(", ", names);
```

Łączenie ciągów znaków jest zazwyczaj akceptowalne podczas inicjalizacji ciągu, ponieważ obejmuje tylko jedną alokację bufora po obliczeniach wymaganej całkowitej długości. Na przykład jeśli masz funkcję, która łączy imię, drugie imię i nazwisko, oddzielając je spacją za pomocą operatora dodawania, tworzysz tylko jeden nowy ciąg znaków w jednym momencie, a nie w wielu krokach:

```
public string ConcatName(string firstName, string middleName, string lastName) {
    return firstName + " " + middleName + " " + lastName;
}
```

Może to wydawać się niewłaściwe, gdybyśmy zakładali, że `firstName + " "` stworzyłoby najpierw nowy ciąg znaków, a następnie tworzył nowy ciąg znaków z middleName i tak dalej, ale kompilator faktycznie zamienia to na pojedyncze wywołanie funkcji `String.Concat()`, która alokuje nowy bufor o długości sumy długości wszystkich ciągów znaków i zwraca go w jednym kroku. Dlatego jest to nadal szybkie. Ale gdy łączysz ciągi znaków w wielu krokach z pomiędzy klauzul if lub pętli, kompilator nie może tego zoptymalizować. Musisz wiedzieć, kiedy można łączyć ciągi znaków, a kiedy nie.

Mówiąc o tym, niemutowalność nie jest świętą pieczęcią, której nie można złamać. Istnieją sposoby na modyfikowanie ciągów znaków w miejscu lub innych niemodyfikowalnych strukturach, które głównie obejmują niebezpieczny kod i istoty astralne, ale zazwyczaj nie są one zalecane, ponieważ ciągi znaków są deduplikowane przez środowisko wykonawcze .NET, a niektóre z ich właściwości, takie jak hasze, są buforowane. Wewnętrzna implementacja silnie polega na właściwości niemutowalności.

Funkcje dla ciągów znaków domyślnie działają w kontekście bieżącej kultury (culture), co może być bolesne doświadczenie, gdy twoja aplikacja przestaje działać w innym kraju.

UWAGA

Kultura, znana również jako lokalizacja w niektórych językach programowania, to zestaw reguł dotyczących wykonywania operacji związanych z danym regionem, takich jak sortowanie ciągów znaków, wyświetlanie daty/czasu we właściwym formacie, układanie sztućców na stole i tak dalej. Bieżąca kultura to zazwyczaj to, co system operacyjny uznaje za aktualnie używaną.

Zrozumienie kultur może uczynić twoje operacje na ciągach znaków bezpieczniejszymi i szybszymi. Na przykład rozważ kod, w którym sprawdzamy, czy podana nazwa pliku ma rozszerzenie .gif:

```
public bool CzyGif(string nazwaPliku) {
   return nazwaPliku.ToLower().EndsWith(".gif");
}
```

Jesteśmy sprytni, widzisz: zamieniamy ciąg znaków na małe litery, więc radzimy sobie z przypadkiem, gdy rozszerzenie może być .GIF lub .Gif lub jakimkolwiek innym połączeniem wielkich i małych liter. Problem polega na tym, że nie wszystkie języki mają takie same zasady małych liter. W języku tureckim na przykład mała litera "I" to nie "i," ale "ı," znane jako kropkowe "i". Kod w tym przykładzie nie zadziałałby w Turcji i być może w niektórych innych krajach, takich jak Azerbejdżan. Zamieniając ciąg znaków na małe litery, faktycznie tworzymy nowy ciąg znaków, co, jak się nauczyliśmy, jest niewydajne.

.NET dostarcza wersje metod dla ciągów znaków, które są niezależne od kultury, takie jak ToLowerInvariant. Oferuje również przeciążenia tych samych metod, które przyjmują wartość StringComparison, która ma alternatywy w postaci invariant (niezależnego od kultury) i ordinal (porównanie znaków według kodów ASCII). Dlatego możemy napisać tę samą metodę w bezpieczniejszy i szybszy sposób:

```
public bool CzyGif(string nazwaPliku) {
   return nazwaPliku.EndsWith(".gif",
       StringComparison.OrdinalIgnoreCase);
}
```

Korzystając z tej metody, unikamy tworzenia nowego ciągu znaków, a także używamy szybszej i bezpiecznej metody porównywania ciągów znaków, która nie zależy od naszej bieżącej kultury i jej skomplikowanych zasad. Moglibyśmy użyć StringComparison.InvariantCultureIgnoreCase, ale w przeciwieństwie do porównania ordinal, dodaje ono kilka dodatkowych reguł tłumaczenia, takich jak traktowanie niemieckich znaków diakrytycznych lub grafemów ich odpowiednikami w alfabecie łacińskim (ß versus ss), co może powodować problemy z nazwami plików lub innymi identyfikatorami zasobów. Porównanie ordinal porównuje wartości znaków bezpośrednio, bez jakiejkolwiek translacji.

2.2.2 Tablica (Array)

Przejrzeliśmy już, jak wygląda tablica w pamięci. Tablice są praktyczne do przechowywania wielu elementów, których liczba nie będzie przekraczać rozmiaru tablicy. Są to struktury statyczne. Nie mogą rosnąć ani zmieniać swojego rozmiaru. Jeśli potrzebujesz większej tablicy, musisz stworzyć nową i skopiować zawartość starej do nowej. Istnieje kilka rzeczy, które musisz wiedzieć na temat tablic.

Tablice, w przeciwieństwie do ciągów znaków, są mutowalne. O to właśnie w nich chodzi. Możesz dowolnie manipulować ich zawartością. Tak naprawdę trudno jest uczynić je niemodyfikowalnymi, co czyni je słabymi kandydatami do interfejsów. Przyjrzyjmy się tej właściwości:

```
public string[] NazwyUżytkowników { get; }
```

Mimo że właściwość nie ma settera, jej typ wciąż jest tablicą, co sprawia, że jest mutowalna. Nic nie powstrzymuje cię przed wykonaniem poniższego kodu:

```
NazwyUżytkowników[0] = "root";
```

Co może komplikować sytuację, nawet gdy to tylko ty używasz tej klasy. Nie powinieneś pozwalać sobie na dokonywanie zmian w stanie, chyba że jest to absolutnie konieczne. Stan jest korzeniem wszelkiego zła, nie null. Im mniej stanów ma twoja aplikacja, tym mniej problemów będziesz mieć.

Stań na stanowisku, które ma najmniejszą funkcjonalność odpowiednią dla twojego celu. Jeśli potrzebujesz tylko przejść przez elementy sekwencyjnie, użyj `IEnumerable<T>`. Jeśli potrzebujesz także powtarzalnie dostępnego licznika, użyj `ICollection<T>`. Zauważ, że metoda rozszerzenia LINQ `.Count()` ma specjalny kod obsługi dla typów, które obsługują `IReadOnlyCollection<T>`, więc nawet jeśli używasz jej na `IEnumerable`, istnieje szansa, że może zwrócić wartość z pamięci podręcznej.

Tablice najlepiej sprawdzają się do użycia w lokalnym zakresie funkcji. Do każdego innego celu istnieje lepiej dostosowany typ lub interfejs do użycia obok `IEnumerable<T>`, takie jak `IReadOnlyCollection<T>`, `IReadOnlyList<T>` lub `ISet<T>`.

2.2.3 Lista (List)

Lista zachowuje się jak tablica, która może nieznacznie rosnąć, podobnie jak działa StringBuilder. Możliwe jest stosowanie list zamiast tablic prawie wszędzie, ale to spowoduje niepotrzebne obciążenie wydajności, ponieważ dostępy indeksowane są wywołaniami wirtualnymi w liście, podczas gdy tablica używa bezpośredniego dostępu.

Wiesz, programowanie obiektowe pozwala na używanie przyjemnej funkcji zwanej polimorfizmem, co oznacza, że obiekt może zachowywać się zgodnie z jego podstawową implementacją, bez zmiany jego interfejsu. Jeśli masz, na przykład, zmienną `a` z typem interfejsu `IOpenable`, `a.Open()` może otworzyć plik lub połączenie sieciowe, w zależności od typu obiektu przypisanego do niej. Osiąga się to przez przechowywanie odwołań do tabeli, która mapuje funkcje wirtualne do wywołania dla typu na początku obiektu, nazywanego tabelą metod wirtualnych lub vtable. W ten sposób, chociaż `Open` mapuje do tego samego wpisu w tabeli w każdym obiekcie tego samego typu, nie będziesz wiedzieć, dokąd prowadzi, dopóki nie sprawdzisz rzeczywistej wartości w tabeli.

Ponieważ nie wiemy, co dokładnie wywołujemy, są to wywołania wirtualne. Wywołanie wirtualne obejmuje dodatkowe wyszukiwanie w tabeli metod wirtualnych, więc jest nieco wolniejsze niż zwykłe wywołania funkcji. To może nie być problemem w przypadku kilku wywołań funkcji, ale gdy dzieje się to wewnątrz algorytmu, jego nadmiar może wzrastać w sposób wykładniczy. W rezultacie, jeśli twoja lista nie będzie rosła po inicjalizacji, możesz chcieć użyć tablicy zamiast listy w lokalnym zakresie.

Zwykle nie powinieneś zbytnio zastanawiać się nad tymi szczegółami. Ale gdy znasz różnicę, są sytuacje, w których tablica może być lepsza od listy.

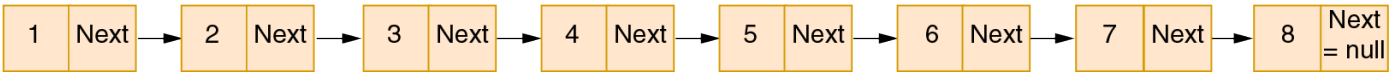
Listy są podobne do StringBuilder. Obydwa są strukturami danych o dynamicznie rosnącej liczbie elementów, ale listy są mniej wydajne w mechanice wzrostu. Kiedy lista stwierdza, że potrzebuje wzrosnąć, alokuje nową tablicę o większym rozmiarze i kopiuje do niej istniejącą zawartość. Z drugiej strony StringBuilder trzyma ze sobą połączone fragmenty pamięci, co nie wymaga operacji kopiowania. Obszar bufora dla list rośnie, gdy osiągnięty zostanie limit bufora, ale rozmiar nowego bufora podwaja się za każdym razem, co oznacza, że potrzeba wzrostu zmniejsza się w miarę czasu. To jest przykład, w którym użycie konkretnej klasy do danego zadania jest bardziej wydajne niż użycie klasy ogólnej.

Można także uzyskać doskonałą wydajność z list, określając ich pojemność. Jeśli nie określisz pojemności dla listy, zacznie ona od pustej tablicy, następnie zwiększy swoją pojemność do kilku elementów. Po osiągnięciu limitu, podwaja swoją pojemność, gdy zostanie zapełniona. Jeśli ustawisz pojemność podczas tworzenia listy, unikniesz niepotrzebnych operacji wzrostu i kopiowania. Pamiętaj o tym, gdy już wiesz, ile maksymalnie elementów będzie mieć lista.

Mając to na uwadze, nie rób nawyku określania pojemności listy bez znanego powodu. Może to spowodować zbędny nadmiar pamięci, który może się akumulować. Używaj tych funkcji świadomie.

2.2.4 Lista dwukierunkowa (Linked list)

Listy dwukierunkowe to listy, w których elementy nie są ułożone kolejno w pamięci, ale każdy element wskazuje na adres następnego elementu. Są użyteczne ze względu na ich wydajność przy operacjach wstawiania i usuwania, które mają złożoność O(1). Nie można uzyskać dostępu do pojedynczych elementów za pomocą indeksu, ponieważ mogą być one przechowywane w dowolnym miejscu w pamięci, co nie jest możliwe do obliczenia. Jednak jeśli głównie uzyskujesz dostęp do początku lub końca listy, lub jeśli potrzebujesz tylko przeglądać elementy, może być równie szybka. W przeciwnym razie sprawdzanie, czy element istnieje w liście dwukierunkowej, ma złożoność O(N), podobnie jak w przypadku tablic i list. Rysunek 2.5 przedstawia przykładowy układ listy dwukierunkowej.



To nie oznacza, że lista dwukierunkowa zawsze jest szybsza niż zwykła lista. Indywidualne przydzielanie pamięci dla każdego elementu zamiast alokacji całego bloku pamięci na raz oraz dodatkowe odwołania do referencji mogą wpływać negatywnie na wydajność.

Lista dwukierunkowa może być potrzebna, gdy potrzebujesz struktury kolejki lub stosu, ale .NET ma to już wbudowane. Idealnie rzecz biorąc, chyba że interesujesz się programowaniem systemowym, nie powinieneś potrzebować używać listy dwukierunkowej w swojej codziennej pracy, z wyjątkiem sytuacji podczas rozmów kwalifikacyjnych o pracę. Niestety, osoby przeprowadzające rozmowy kwalifikacyjne uwielbiają zagadki związane z listami dwukierunkowymi, dlatego warto z nimi się zapoznać.

NIE, NIE BĘDZIESZ ODWRACAĆ LISTY DWUKIERUNKOWEJ

Odpowiadanie na pytania związane z kodowaniem podczas rozmów kwalifikacyjnych to rytuał dla stanowisk związanych z rozwojem oprogramowania. Większość pytań dotyczy również pewnych struktur danych i algorytmów. Listy dwukierunkowe są częścią tego zestawu, więc istnieje szansa, że ktoś poprosi cię o odwrócenie listy dwukierunkowej lub odwrócenie drzewa binarnego.

Prawdopodobnie nigdy nie będziesz wykonywać tych zadań w swojej rzeczywistej pracy, ale aby oddać sprawiedliwość osobie przeprowadzającej rozmowę kwalifikacyjną, testują one twoją wiedzę z zakresu struktur danych i algorytmów, aby po prostu upewnić się, że wiesz, co robisz. Chcą sprawdzić, czy jesteś w stanie podjąć odpowiednie decyzje, gdy pojawi się potrzeba użycia odpowiedniej struktury danych we właściwym miejscu. Testują również twoją umiejętność myślenia analitycznego i rozwiązywania problemów, dlatego ważne jest, abyś myślał na głos i dzielił się swoim sposobem myślenia z osobą przeprowadzającą rozmowę.

Nie zawsze musisz rozwiązywać dane pytanie. Osoba przeprowadzająca rozmowę zwykle szuka kogoś, kto jest pasjonatem i ma wiedzę na temat pewnych podstawowych koncepcji oraz potrafi się odnaleźć, nawet jeśli czasem się zagubi.

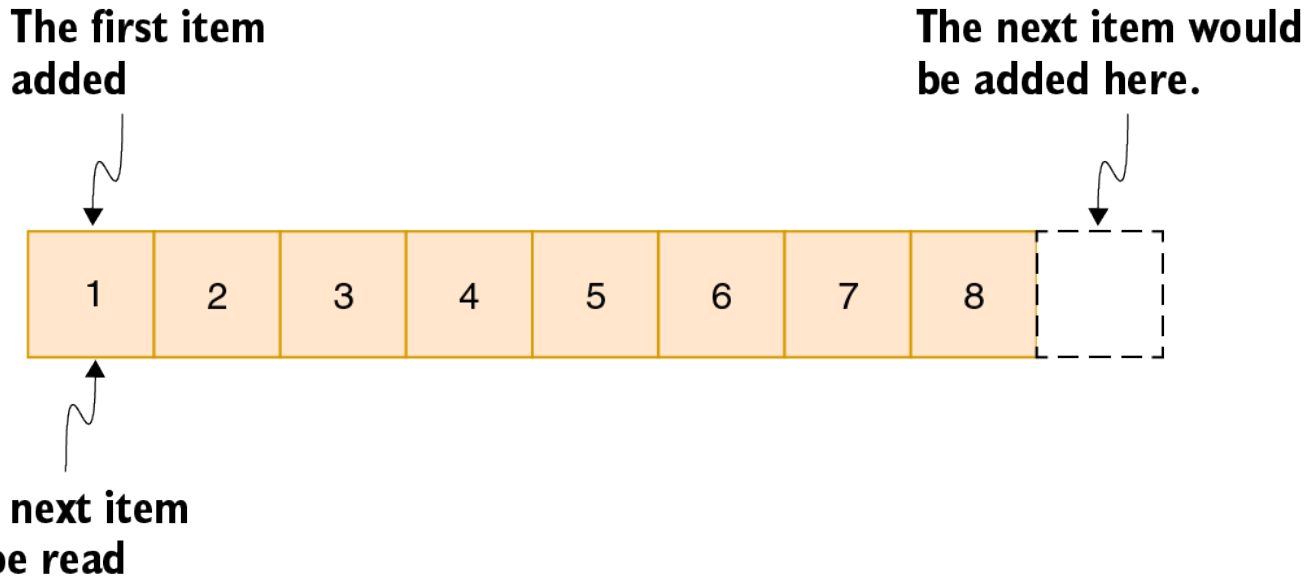
Na przykład ja zwykle dodawałem dodatkowy etap do moich pytań podczas rozmów kwalifikacyjnych w Microsoft, w którym kandydat musiał znaleźć błędy w swoim kodzie. To sprawiało, że czuli się lepiej, ponieważ mieli świadomość, że błędy są oczekiwane i nie byli oceniani na podstawie tego, jak bezbłędny jest ich kod, ale na podstawie tego, jak potrafią zidentyfikować błędy.

Rozmowy kwalifikacyjne nie polegają tylko na znalezieniu odpowiedniej osoby, ale także na znalezieniu kogoś, z kim przyjemnie będzie się pracować. Ważne jest, abyś był ciekawski, pełen pasji, wytrwały i łatwy w obyciu, kto naprawdę może pomóc im w wykonywaniu ich zadań.

Listy dwukierunkowe były bardziej popularne w czasach prehistorycznych programowania, ponieważ priorytetem było efektywne zarządzanie pamięcią. Nie mogliśmy pozwolić sobie na alokację kilobajtów pamięci tylko dlatego, że nasza lista potrzebowała rosnąć. Musieliśmy trzymać się szczelnej gospodarki pamięcią. Lista dwukierunkowa była idealną strukturą danych dla tego celu. Są nadal często używane w jądrach systemów operacyjnych ze względu na ich nieodpartą charakterystykę O(1) dla operacji wstawiania i usuwania.

2.2.5 Queue

Kolejka to struktura danych, która reprezentuje najbardziej podstawową formę cywilizacji. Pozwala odczytywać elementy z listy w kolejności ich dodawania. Kolejka może być po prostu tablicą, pod warunkiem, że trzymasz osobne miejsca na odczytanie następnego elementu i wstawienie nowego. Jeśli dodalibyśmy rosnące liczby do kolejki, wyglądałaby to mniej więcej tak jak na rysunku 2.6.



Bufor klawiatury na komputerach PC w czasach MS-DOS używał prostego tablicy bajtów do przechowywania naciśnięć klawiszy. Bufor zapobiegał utracie naciśniętych klawiszy ze względu na wolne lub niewydolne oprogramowanie. Gdy bufor był pełny, BIOS wydawał sygnał dźwiękowy, abyśmy wiedzieli, że nasze naciśnięcia klawiszy nie są już rejestrowane. Na szczęście w .NET istnieje gotowa klasa Queue, którą możemy używać bez konieczności martwienia się o szczegóły implementacji i wydajność.

2.2.6 Directory

Słowniki, nazywane również hashmapami lub czasem strukturami klucz/wartość, należą do najbardziej przydatnych i najczęściej używanych struktur danych. Zwykle nie zastanawiamy się nad ich zdolnościami, traktujemy je jak coś oczywistego. Słownik to pojemnik, który może przechowywać klucz i wartość. Później może odnaleźć wartość związana z kluczem w czasie stałym, czyli O(1). Oznacza to, że są one niezwykle szybkie w pobieraniu danych. Dlaczego są tak szybkie? Gdzie tkwi magia?

Magia tkwi w słowie "hash" (hasz). Haszowanie to proces generowania pojedynczej liczby na podstawie dowolnych danych. Wygenerowana liczba musi być deterministyczna, co oznacza, że dla tych samych danych zawsze będzie generować tę samą liczbę, ale nie musi być unikalna. Istnieje wiele różnych sposobów obliczania wartości hasza. Logika haszowania obiektu znajduje się w implementacji GetHashCode.

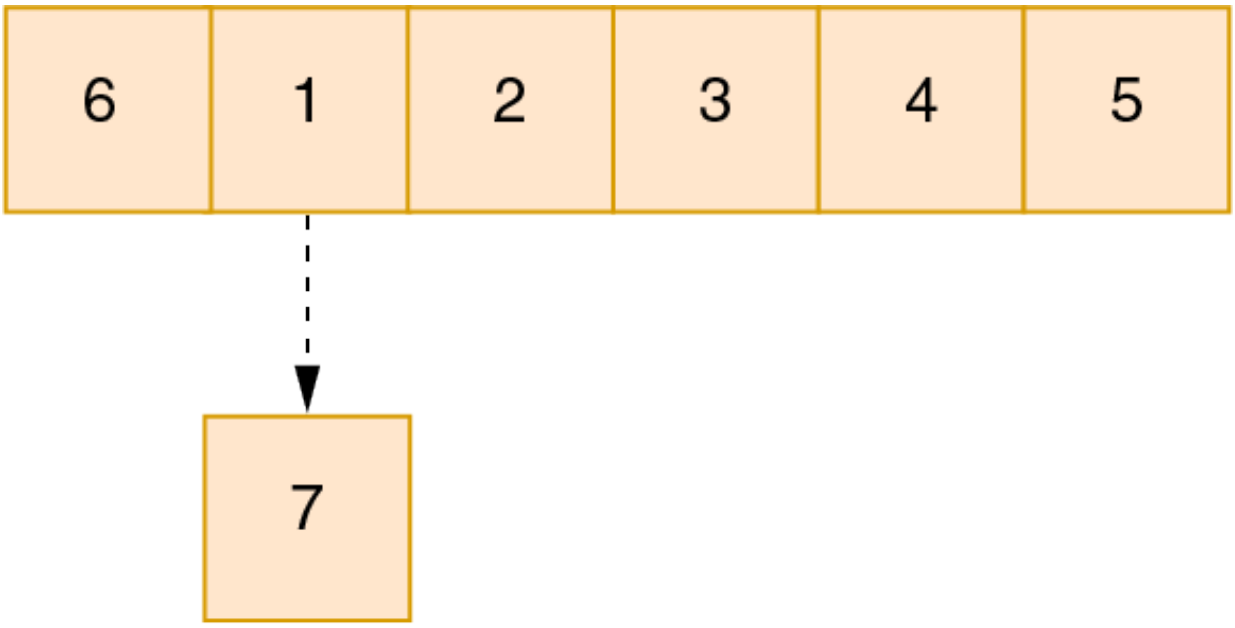
Hasze są fajne, ponieważ za każdym razem dostajesz tę samą wartość, co pozwala na ich używanie do przeszukiwania. Na przykład mając tablicę wszystkich możliwych wartości hasz, można je przeszukać, używając indeksów tablicy. Ale taka tablica zajęłaby około 16 gigabajtów dla każdego utworzonego słownika, ponieważ każda liczba całkowita zajmuje cztery bajty i może mieć około czterech miliardów możliwych wartości.

Słowniki przydzielają znacznie mniejszą tablicę i polegają na równomiernym rozłożeniu wartości hasza. Zamiast przeszukiwania wartości hasza, szukają "wartość hasza modulo długość tablicy". Załóżmy, że słownik z kluczami całkowitymi przydziela tablicę sześciu elementów, aby je indeksować, a metoda GetHashCode() dla liczby całkowitej zwraca po prostu jej wartość. To oznacza, że nasza formuła do określenia, gdzie element zostanie zmapowany, byłaby po prostu wartość % 6, ponieważ indeksy tablicy zaczynają się od zera. Tablica liczb od 1 do 6 zostałaby rozłożona, jak pokazano na rysunku 2.7.

Co się dzieje, gdy mamy więcej elementów niż pojemność słownika? Na pewno będą się nakładać, więc słowniki przechowują te nakładające się elementy w dynamicznie rosnącej liście. Jeśli przechowujemy elementy z kluczami od 1 do 7, tablica wyglądałaby tak, jak to pokazano na rysunku 2.8.

Rys 2.7

Rys 2.8

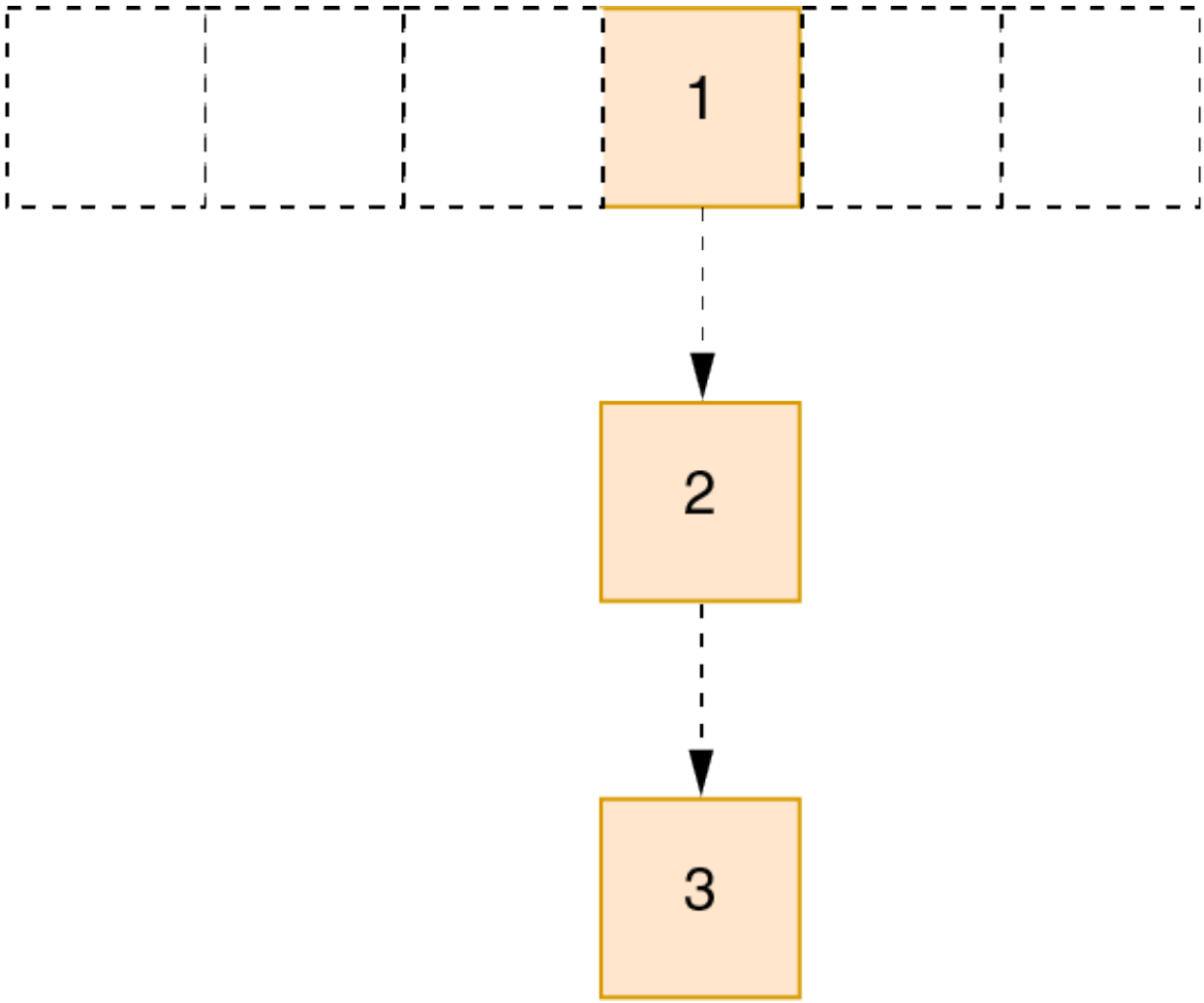


Dlaczego omawiam ten temat? Ponieważ wydajność wyszukiwania klucza w słowniku wynosi zazwyczaj O(1), podczas gdy nadmiar wyszukiwania w liście połączonej wynosi O(N). Oznacza to, że w miarę wzrostu liczby nakładających się elementów, wydajność wyszukiwania będzie się pogarszać. Jeśli na przykład miałbyś funkcję GetHashCode, która zawsze zwracałaby 4:

```
public override int GetHashCode() {
   return 4; // wybrane za pomocą uczciwego rzutu kością
}
```

Oznacza to, że wewnętrzna struktura słownika przy dodawaniu do niego elementów przypominałaby schemat przedstawiony na rysunku 2.9.

Rys 2.9



Słownik nie jest lepszy od listy połączonej, jeśli masz złe wartości funkcji skrótu (GetHashCode). Może nawet działać gorzej ze względu na dodatkowe działania, jakie musi wykonać słownik, aby obsłużyć te elementy. To przynosi nas do najważniejszego punktu: Twoja funkcja GetHashCode musi być jak najbardziej unikalna. Jeśli masz wiele nakładających się elementów, Twoje słowniki będą cierpieć, cierpiący słownik sprawi, że Twoja aplikacja będzie cierpieć, a cierpiąca aplikacja sprawi, że cała firma będzie cierpieć. W końcu to Ty będziesz cierpieć. Przez brak gwoździa, cała królestwo zostało utracone.

Czasami musisz połączyć wartości z wielu właściwości w klasie, aby obliczyć unikalną wartość skrótu. Na przykład nazwy repozytoriów na GitHubie są unikalne dla każdego użytkownika. Oznacza to, że dowolny użytkownik może mieć repozytorium o tej samej nazwie, a sama nazwa repozytorium nie jest wystarczająca, aby je uczynić unikalnym. Jeśli użyjesz tylko samej nazwy, spowoduje to większe kolizje. Oznacza to, że musisz połączyć wartości skrótu. Podobnie, jeśli nasza witryna ma unikalne wartości dla różnych tematów, mielibyśmy ten sam problem.

Aby skutecznie łączyć wartości skrótu, musisz znać ich zakresy i zajmować się ich reprezentacją bitową. Jeśli po prostu używasz operatora jak dodawanie lub proste operacje OR/XOR, nadal możesz otrzymać znacznie więcej kolizji, niż się tego spodziewasz. Musisz również używać przesunięć bitowych. Prawidłowa funkcja GetHashCode będzie używała operacji bitowych, aby uzyskać równomierne rozłożenie na pełnych 32 bitach liczby całkowitej.

Kod takiej operacji może wyglądać jak sceny hakerskie z tandetnego filmu o hakerach. Jest enigmatyczny i trudny do zrozumienia nawet dla kogoś, kto jest zaznajomiony z tym pojęciem. W zasadzie obracamy jedną z 32-bitowych liczb całkowitych o 16 bitów, więc jej najniższe bajty są przesuwane ku środkowi i XORujemy ("^") tę wartość z inną 32-bitową liczbą całkowitą, co znacznie zmniejsza szanse na kolizje. Wygląda to tak— przerażająco:

```
public override int GetHashCode() {
   return (int)(((TopicId & 0xFFFF) << 16)
       ^ (TopicId & 0xFFFF0000 >> 16)
       ^ PostId);
}
```

Na szczęście, wraz z pojawieniem się .NET Core i .NET 5, łączenie wartości skrótu w sposób minimalizujący kolizje zostało zasłonięte za pomocą klasy HashCode. Aby połączyć dwie wartości, wystarczy zrobić tak:

```
public override int GetHashCode() {
   return HashCode.Combine(TopicId, PostId);
}
```

Kody skrótu są używane nie tylko jako klucze słowników, ale także w innych strukturach danych, takich jak zbiory. Ponieważ jest znacznie łatwiej napisać odpowiednią funkcję GetHashCode za pomocą funkcji pomocniczych, nie masz wymówki, aby tego nie zrobić. Bądź czujny w tym zakresie.

Kiedy nie powinieneś używać słownika? Jeśli potrzebujesz przejść po parach klucz-wartość sekwencyjnie, słownik nie oferuje żadnych korzyści. W rzeczywistości może nawet zaszkodzić wydajności. Warto wówczas rozważyć użycie List<KeyValuePair<K, V>>, aby uniknąć niepotrzebnego narzutu.

Przepraszam za moje niedoprecyzowanie. Oto tłumaczenie nagłówka:

2.2.7 HashSet

HashSet (Zbiór) to struktura danych podobna do tablicy lub listy, z tą różnicą, że może zawierać tylko unikalne wartości. Jej przewagą nad tablicami lub listami jest to, że ma stały czas dostępu O(1) podobnie jak klucze w słowniku, dzięki mapom opartym na funkcjach skrótu, o których właśnie rozmawialiśmy. Oznacza to, że jeśli musisz często sprawdzać, czy dana tablica lub lista zawiera określony element, używanie zbioru może być szybsze. W .NET jest to nazywane HashSet i jest dostępne za darmo.

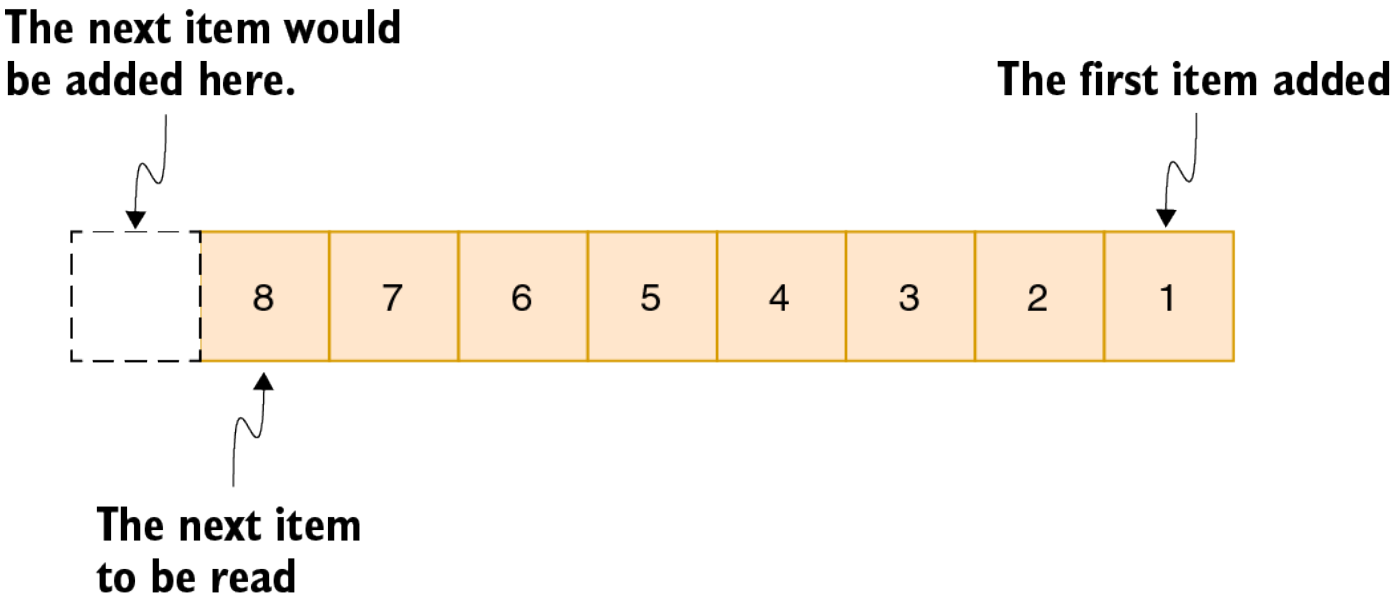
Ponieważ HashSet jest szybki podczas wyszukiwania i wstawiania, nadaje się również do operacji na przecięciach i sumach zbiorów. Nawet dostarcza metody, które umożliwiają korzystanie z tych funkcji. Aby czerpać korzyści z tych możliwości, musisz zwrócić uwagę na implementacje metody GetHashCode().

2.2.8 Stos

Stosy to kolejki LIFO (Last In First Out). Są użyteczne, gdy chcesz zapisać stan i przywrócić go w odwrotnej kolejności, w jakiej został zapisany. Kiedy odwiedzasz urząd Departamentu Komunikacji Samochodowej (DMV) w rzeczywistości, czasami musisz korzystać ze stosu. Najpierw podchodzisz do stanowiska 5, a pracownik przy stanowisku sprawdza twoje dokumenty i widzi, że brakuje ci płatności, więc odsyła cię do stanowiska 13. Pracownik przy stanowisku 13 widzi, że brakuje ci zdjęcia w dokumentach i odsyła cię do kolejnego stanowiska, tym razem do stanowiska 47, aby zrobić zdjęcie. Następnie musisz wrócić do stanowiska 13, gdzie odbierasz potwierdzenie płatności, a następnie wrócić do stanowiska 5, aby odebrać prawo jazdy. Lista stanowisk i sposób ich przetwarzania w kolejności (LIFO) to operacja przypominająca stos, która zazwyczaj jest bardziej wydajna niż w przypadku DMV.

Stos można reprezentować za pomocą tablicy. Różnica polega na tym, gdzie umieszczasz nowe elementy i skąd odczytujesz następny element. Gdybyśmy zbudowali stos, dodając liczby w kolejności rosnącej, wyglądałby jak na rysunku 2.10.

Rys 2.10



Dodawanie do stosu jest zwykle nazywane "pushing", a odczytywanie następnej wartości ze stosu nazywane jest "popping". Stosy są przydatne do cofania się po wykonanych krokach. Być może już jesteś zaznajomiony ze stosu wywołań, ponieważ pokazuje on nie tylko miejsce wystąpienia wyjątku, ale także ścieżkę wykonania, którą podążył program. Funkcje wiedzą, gdzie wrócić po zakończeniu wykonania, używając stosu. Przed wywołaniem funkcji, adres powrotu jest dodawany na stos. Gdy funkcja chce wrócić do swojego wywołującego, odczytywany jest ostatni adres dodany na stos i CPU kontynuuje wykonywanie kodu od tego adresu.

2.2.9 Stos wywołań

Stos wywołań to struktura danych, w której funkcje przechowują adresy powrotu, dzięki czemu wywołane funkcje wiedzą, gdzie wrócić po zakończeniu działania. Istnieje jeden stos wywołań na wątek.

Każda aplikacja działa w jednym lub więcej osobnych procesach. Procesy pozwalają na izolację pamięci i zasobów. Każdy proces ma jeden lub więcej wątków. Wątek jest jednostką wykonawczą. Wszystkie wątki działają równolegle na systemie operacyjnym, stąd nazwa wielowątkowość. Nawet jeśli masz procesor czterordzeniowy, system operacyjny może równocześnie uruchamiać tysiące wątków. Możliwe jest to, ponieważ większość wątków większość czasu czeka na zakończenie jakiegoś procesu, więc można zapełnić ich miejsce innym wątkiem, co daje wrażenie działania wszystkich wątków równocześnie. Dzięki temu nawet na pojedynczym procesorze możliwe jest wielozadaniowość.

Kiedyś proces był zarówno kontenerem dla zasobów aplikacji, jak i jednostką wykonawczą w starszych systemach UNIX. Chociaż podejście to było proste i eleganckie, powodowało problemy, takie jak procesy zombie. Wątki są bardziej lekkie i nie mają takiego problemu, ponieważ są związane z czasem życia wykonania.

Każdy wątek ma swój własny stos wywołań: stałą ilość pamięci. Tradycyjnie stos rośnie od góry do dołu w przestrzeni pamięci procesu, gdzie góra oznacza koniec przestrzeni pamięci, a dół oznacza nasz słynny wskaźnik zerowy: adres zero. Dodanie elementu na stos wywołań oznacza umieszczenie go tam i zmniejszenie wskaźnika stosu.

Jak każda dobra rzecz, stos ma swój koniec. Ma stały rozmiar, więc gdy przekroczy tę wielkość, CPU generuje wyjątek StackOverflowException, z którym będziesz się spotykać w swojej karierze, gdy przypadkowo wywołasz funkcję od siebie samej. Stos jest dość duży, więc zazwyczaj nie martwisz się o osiągnięcie limitu w normalnych przypadkach.

Stos wywołań przechowuje nie tylko adresy powrotu, ale także parametry funkcji i zmienne lokalne. Ponieważ zmienne lokalne zajmują tak mało pamięci, stos jest bardzo efektywny dla nich, ponieważ nie wymaga dodatkowych kroków zarządzania pamięcią, takich jak alokacja i dealokacja. Stos jest szybki, ale ma stały rozmiar i ma taki sam czas życia jak funkcja go używająca. Po zwróceniu się z funkcji, przestrzeń stosu jest oddawana. Dlatego idealne jest przechowywanie w nim małej ilości danych lokalnych. W związku z tym zarządzane środowiska wykonawcze, takie jak C# czy Java, nie przechowują danych klas w stosie, a jedynie ich odwołania.

To jest kolejny powód, dla którego typy wartości mogą mieć lepszą wydajność w pewnych przypadkach w porównaniu do typów referencyjnych. Typy wartości istnieją na stosie tylko wtedy, gdy są deklarowane lokalnie, chociaż są przekazywane przez kopiowanie.

2.3 O co chodzi z typami danych?

Programiści często biorą typy danych za pewnik. Niektórzy nawet twierdzą, że programiści są szybsi w językach o typach dynamicznych, takich jak JavaScript czy Python, ponieważ nie muszą przejmować się skomplikowanymi detalami, jak na przykład określanie typu każdej zmiennej.

UWAGA

Typ dynamiczny oznacza, że typy danych zmiennych lub elementów klas w języku programowania mogą się zmieniać podczas działania programu. W języku JavaScript możesz przypisać zmienną najpierw jako string, a później jako liczbę całkowitą, ponieważ jest to język o typach dynamicznych. Języki o typach statycznych, takie jak C# czy Swift, nie pozwoliłyby na takie działanie. Omówimy to dokładniej później.

Tak, określanie typów dla każdej zmiennej, każdego parametru i każdego elementu w kodzie może być uciążliwe, ale musisz podchodzić do tego w sposób całościowy, jeśli chcesz być szybszy. Bycie szybkim nie polega tylko na pisaniu kodu, ale także na jego utrzymaniu. Mogą być przypadki, kiedy nie musisz martwić się o utrzymanie, na przykład gdy właśnie zostałeś zwolniony i nie obchodzi cię to. Poza takimi sytuacjami, rozwijanie oprogramowania to maraton, a nie sprint.

Szybkie wykrywanie błędów to jedna z najlepszych praktyk w programowaniu. Typy danych stanowią jedną z najwcześniejszych obron przed tarciem w trakcie tworzenia kodu. Typy pozwalają ci na szybkie wykrywanie błędów i naprawę ich, zanim staną się problemem. Oprócz oczywistej korzyści z unikania przypadkowego mylenia stringa z liczbą całkowitą, typy mogą działać na twoją korzyść także w inny sposób.

2.3.1 Silne związki z typami

Większość języków programowania posiada typy danych. Nawet najprostsze języki programowania, takie jak BASIC, miały typy danych: stringi i liczby całkowite. Niektóre ich dialekty miały nawet liczby rzeczywiste. Istnieją także języki określane jako "beztroskie związki" (typeless), takie jak Tcl, REXX, Forth, i tak dalej. W tych językach operacje są wykonywane tylko na jednym typie danych, zwykle na stringach lub liczbach całkowitych. Brak konieczności myślenia o typach danych sprawia, że programowanie jest wygodniejsze, ale prowadzi do wolniejszego i bardziej podatnego na błędy kodu.

Typy danych to podstawowe narzędzie do zapewnienia poprawności kodu, więc zrozumienie systemu typów może znacznie pomóc w staniu się produktywnym programistą. Sposób, w jaki języki programowania implementują typy danych, jest silnie powiązany z tym, czy są one interpretowane czy kompilowane:

- Interpretowane języki programowania, takie jak Python czy JavaScript, pozwalają na natychmiastowe uruchamianie kodu z pliku tekstowego bez konieczności etapu kompilacji. Ze względu na swoją natychmiastową naturę, zmienne mają zwykle elastyczne typy: możesz przypisać string do zmiennej, która wcześniej przechowywała liczbę całkowitą, a nawet dodawać do siebie stringi i liczby. Są one zwykle określane jako języki o typach dynamicznych ze względu na sposób, w jaki implementują typy danych. W językach interpretowanych kod można pisać znacznie szybciej, ponieważ nie trzeba deklarować typów.
- Kompilowane języki programowania są zwykle bardziej restrykcyjne. Jak restrykcyjne są, zależy od tego, jak wiele trudności chce ci sprawić projektant języka. Na przykład język Rust może być uważany za niemieckie inżynieria wśród języków programowania: bardzo rygorystyczny, perfekcjonistyczny i dlatego wolny od błędów. Język C także można porównać do niemieckiej inżynierii, ale bardziej jak Volkswagen: pozwala ci łamać zasady i później płacić za to cenę. Oba języki są statycznie typowane. Po zadeklarowaniu zmiennej jej typ nie może się zmienić, ale Rust jest uważany za silnie typowany, podobnie jak C#, podczas gdy C uznawany jest za słabo typowany.

"Silnie typowane" i "słabo typowane" oznaczają, jak elastycznie język pozwala na przypisywanie różnych typów zmiennych do siebie nawzajem. W tym sensie C jest bardziej elastyczny: możesz przypisać wskaźnik do liczby całkowitej i odwrotnie bez problemów. Z drugiej strony C# jest bardziej restrykcyjny: wskaźniki/referencje i liczby całkowite to niezgodne typy. Tabela 2.2 pokazuje, do jakich kategorii należą różne języki programowania.

	Statycznie Typowane	Dynamicznie Typowane
	Typ zmiennej nie może się zmieniać w trakcie działania programu.	Typ zmiennej może się zmieniać w trakcie działania programu.
Silnie Typowane		
Różne typy nie mogą być zamienione nawzajem.	C#, Java, Rust, Swift, Kotlin,TypeScript, C++	Python, Ruby, Lisp
Słabo Typowane		
Różne typy mogą byćzamienione nawzajem.	Visual Basic, C	JavaScript, VBScript

Ścisłe języki programowania mogą być frustrujące. Języki takie jak Rust mogą nawet sprawić, że zaczniemy zastanawiać się nad sensem życia i istnieniem we wszechświecie. Deklarowanie typów i jawne ich konwertowanie, gdy jest to potrzebne, może wydawać się uciążliwe i pełne biurokracji. Przykładowo, w języku JavaScript nie musisz deklarować typów każdej zmiennej, argumentu czy elementu klasy. Dlaczego więc obciążamy się jawnymi deklaracjami typów, skoro wiele języków programowania może działać bez nich?

Odpowiedź jest prosta: typy mogą pomóc nam pisać kod, który jest bezpieczniejszy, szybszy i łatwiejszy do utrzymania. Dzięki nim odzyskujemy czas, który straciliśmy na deklarowaniu typów zmiennych, adnotacji naszych klas. Zyskujemy go dzięki mniejszej liczbie błędów do debugowania i mniejszej ilości problemów z wydajnością.

Oprócz oczywistych korzyści wynikających z korzystania z typów, mają one również subtelne zalety. Przeanalizujmy je.

2.3.2 Dowód poprawności

Dowód poprawności to jedna z mniej znanych korzyści płynących z posiadania predefiniowanych typów. Załóżmy, że tworzysz platformę mikroblogowania, która pozwala na określoną liczbę znaków w każdym poście, i w zamian nie jesteś oceniany za lenistwo w pisaniu czegoś dłuższego niż zdanie. Na tej hipotetycznej platformie mikroblogowania możesz wspominać innych użytkowników w poście za pomocą prefiksu @ oraz oznaczać inne posty za pomocą prefiksu #, po którym następuje identyfikator posta. Możesz nawet odnaleźć post, wpisując jego identyfikator w polu wyszukiwania. Jeśli wpiszesz nazwę użytkownika z prefiksem @ w polu wyszukiwania, pojawi się profil tego użytkownika.

Wejście użytkownika wiąże się z nowym zestawem problemów z walidacją. Co się stanie, jeśli użytkownik wpisze litery po prefiksie #? Co jeśli wpiszą dłuższą liczbę, niż jest dozwolona? Może się wydawać, że te scenariusze same się rozwiążą, ale zwykle aplikacja się zawiesi, ponieważ w jakimś miejscu ścieżki kodu coś, co nie oczekuje na nieprawidłowe dane wejściowe, spowoduje wystąpienie wyjątku. To jest najgorsze możliwe doświadczenie dla użytkownika: nie wiedzą, co poszło nie tak, i nie wiedzą nawet, co mają zrobić

dalej. Może to nawet stać się problemem związanym z bezpieczeństwem, jeśli wyświetlasz to wejście bez jego oczyszczenia.

Walidacja danych nie zapewnia dowodu poprawności w całym kodzie. Możesz zwalidować dane wejściowe po stronie klienta, ale ktoś, na przykład aplikacja stron trzecich, może wysłać żądanie bez walidacji. Możesz zwalidować kod obsługujący żądania internetowe, ale inna twoja aplikacja, na przykład kod API, może wywołać kod usługi bez koniecznej walidacji. Podobnie, kod bazy danych może otrzymywać żądania z różnych źródeł, takich jak warstwa usługi i zadanie konserwacyjne, dlatego musisz upewnić się, że wstawiasz odpowiednie rekordy do bazy danych. Rysunek 2.11 przedstawia punkty, w których aplikacja może potrzebować zwalidować dane wejściowe.