Algorytmy i dane

Wiemy już, że algorytmy zawierają starannie dobrane instrukcje elementarne, które określają podstawowe czynności do wykonania. Nie omawialiśmy jeszcze rozmieszczenia tych instrukcji w algorytmie, który umożliwia człowiekowi lub komputerowi ustalenie dokładnej kolejności czynności, które należy wykonać. Nie omawialiśmy też przedmiotów, którymi manipulują te działania. Algorytm może być traktowany jako wykonywany przez małego robota lub procesor (który może być odpowiednio nazwany Runaround). Procesor otrzymuje rozkazy biegania dookoła robiąc to i tamto, gdzie „to i tamto" są podstawowymi działaniami algorytmu. W algorytmie sumowania wynagrodzeń z poprzedniej części, małemu Runaroundowi kazano zanotować 0, a następnie zacząć przeglądać listę pracowników, znajdować pensje i dodawać je do zanotowanej liczby. Powinno być dość oczywiste, że kolejność wykonywania podstawowych czynności jest kluczowa. Niezwykle ważne jest nie tylko to, aby podstawowe instrukcje algorytmu były jasne i jednoznaczne, ale to samo powinno dotyczyć mechanizmu sterującego kolejnością wykonywania tych instrukcji. Algorytm musi zatem zawierać instrukcje sterujące, aby „pchać" procesor w tym lub innym kierunku, mówiąc mu, co ma robić na każdym kroku i kiedy przestać i powiedzieć „jestem jednym".

Struktury kontrolne

Sterowanie sekwencją jest zwykle realizowane za pomocą różnych kombinacji instrukcji zwanych strukturami przepływu sterowania lub po prostu strukturami sterowania. Nawet przepis na mus czekoladowy zawiera kilka typowych, takich jak:

* Sekwencjonowanie bezpośrednie, w formie „zrób A, a potem B" lub „zrób A, a potem B". (Każdy średnik lub kropka w przepisie kryje w sobie frazę „a potem", na przykład „delikatnie złóż czekoladę; [a następnie] lekko podgrzej…")

* Rozgałęzienie warunkowe w postaci „jeśli Q, to zrób A, w przeciwnym razie wykonaj B" lub po prostu „jeśli Q, to zrób A", gdzie Q jest pewnym warunkiem. (Na przykład w przepisie „w razie potrzeby lekko podgrzej, aby rozpuścić czekoladę" lub „w razie potrzeby podawaj z bitą śmietaną").

Tak się składa, że te dwie konstrukcje sterujące, sekwencjonowanie i rozgałęzianie, nie wyjaśniają, w jaki sposób algorytm o stałej - może nawet krótkiej - długości może opisywać procesy, które mogą rosnąć coraz dłużej, w zależności od konkretnego wejścia. Algorytm zawierający tylko sekwencjonowanie i rozgałęzianie może zalecić procesy tylko o pewnej ograniczonej długości, ponieważ żadna część takiego algorytmu nie jest nigdy wykonywana więcej niż raz. Konstrukty sterujące, które są odpowiedzialne za przepisywanie coraz dłuższych procesów, są rzeczywiście ukryte nawet w przepisie na mus, ale są znacznie bardziej wyraźne w algorytmach, które radzą sobie z wieloma danymi wejściowymi o różnych rozmiarach, takimi jak algorytm sumowania wynagrodzeń. Są one ogólnie nazywane iteracjami lub konstrukcjami zapętlonymi i występują w wielu odmianach. Oto dwa:

* Ograniczona iteracja w formie ogólnej „zrób A dokładnie N razy", gdzie N jest liczbą.

* Iteracja warunkowa, czasami nazywana iteracją nieograniczoną, w postaci „powtórz A do Q" lub „gdy Q do A", gdzie Q jest warunkiem. (Na przykład w przepisie „ubijaj białka do uzyskania piany").

Opisując algorytm sumowania wynagrodzeń dość nieprecyzyjnie podchodziliśmy do sposobu realizacji głównej części algorytmu; powiedzieliśmy „przejdź przez listę, dodając pensję każdego pracownika do zanotowanej liczby", a następnie „po osiągnięciu końca listy wygeneruj odnotowany numer jako wynik". Powinniśmy naprawdę użyć konstrukcji iteracyjnej, która nie tylko precyzuje zadanie procesora przechodzącego przez listę, ale także sygnalizuje koniec listy. Załóżmy zatem, że dane wejściowe do

problemu obejmują nie tylko listę pracowników, ale także jej długość; to jest całkowita liczba pracowników, oznaczona literą N. Teraz można użyć ograniczonej konstrukcji iteracyjnej, dając następujący algorytm:

(1) zanotuj 0; aby wskazać pierwszą pensję na liście;

(2) wykonaj następujące N - 1 razy:

(2.1) dodać wskazaną pensję do zaznaczonego numeru;

(2.2) wskazać następną pensję;

(3) dodać wskazaną pensję do zaznaczonego numeru;

(4) wygeneruj odnotowaną liczbę jako dane wyjściowe.

Wyrażenie „następujące" w punkcie (2) odnosi się do segmentu składającego się z podrozdziałów (2.1) i (2.2). Ta konwencja, w połączeniu z wcięciem tekstu, aby podkreślić „zagnieżdżony" charakter (2.1) i (2.2), będzie swobodnie używana w sequelu. Zachęcamy do szukania powodu używania N - 1 i dodawania końcowej pensji oddzielnie, zamiast po prostu używać N, a następnie tworzyć wyniki i zatrzymywać się. Zauważ, że algorytm zawodzi, jeśli lista jest pusta (to znaczy, jeśli N wynosi 0), ponieważ druga część klauzuli (1) nie ma sensu. Jeśli dane wejściowe nie zawierają N, całkowitej liczby pracowników, musimy użyć warunkowej iteracji, która wymaga od nas podania sposobu, w jaki algorytm może wykryć, kiedy dotarł do końca listy. Wynikowy algorytm wyglądałby bardzo podobnie do podanej wersji, ale używałby formy „powtórz następujące do osiągnięcia końca listy" w klauzuli (2). Powinieneś spróbować spisać pełny algorytm dla tego przypadku. Zwróć uwagę, jak konstrukcje iteracyjne umożliwiają krótkiej części tekstu algorytmu przypisanie bardzo długich procesów, których długość jest podyktowana wielkością danych wejściowych — w tym przypadku długością listy pracowników. Iteracja jest zatem kluczem do pozornego paradoksu jednego, ustalonego algorytmu wykonującego zadania o coraz dłuższym czasie trwania.

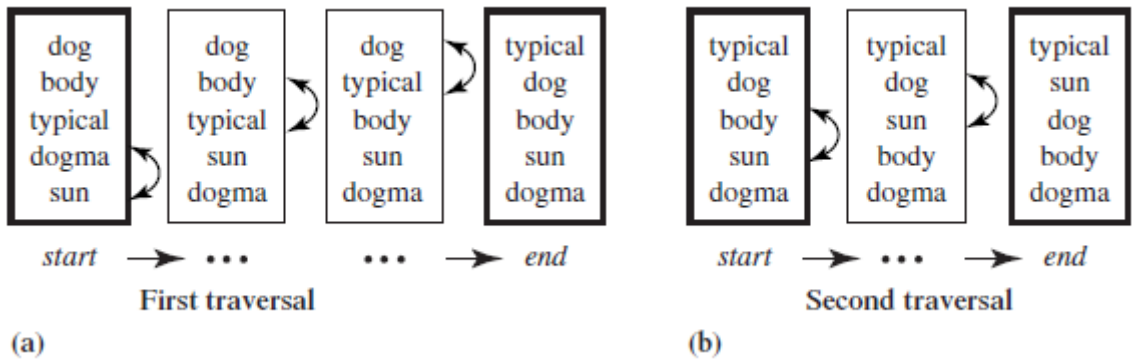
Łączenie struktur kontrolnych

Algorytm może zawierać wiele konstrukcji przepływu sterowania w nietrywialnych kombinacjach. Sekwencjonowanie, rozgałęzianie i iteracje mogą być przeplatane i zagnieżdżane w sobie. Na przykład algorytmy mogą zawierać zagnieżdżone iteracje, częściej nazywane zagnieżdżonymi pętlami. Pętla wewnątrz pętli może przybrać formę „zrób A dokładnie N razy", gdzie samo A jest, powiedzmy, postacią „powtórz B do C". Procesor realizujący taki segment musi dość ciężko pracować; za każdym razem, gdy wykonuje A, za każdym razem, gdy przechodzi pętla zewnętrzna, pętla wewnętrzna musi być przemierzana wielokrotnie, aż C stanie się prawdziwe. Tutaj zewnętrzna pętla jest ograniczona, a wewnętrzna warunkowa, ale możliwe są również inne kombinacje. Część A zewnętrznej pętli może zawierać wiele dalszych segmentów, z których każdy może z kolei wykorzystywać dodatkowe konstrukcje sekwencjonowania, rozgałęziania i iteracji, to samo dotyczy pętli wewnętrznej. Tym samym nie ma ograniczeń co do potencjalnej złożoności algorytmów. Rozważmy prosty przykład potęgi iteracji zagnieżdżonych. Załóżmy, że problemem było zsumowanie wynagrodzeń, ale nie wszystkich pracowników, tylko tych, którzy zarabiają więcej niż ich bezpośredni przełożeni. Oczywiście zakłada się, że (poza prawdziwym „szefem") w kartotece pracownika znajduje się nazwisko przełożonego tego pracownika. Algorytm rozwiązujący ten problem może być skonstruowany tak, aby pętla zewnętrzna przebiegała w dół listy, jak poprzednio, ale dla każdego pracownika „wskazanego" pętla wewnętrzna przeszukuje listę w celu znalezienia rekordu bezpośredniego przełożonego tego pracownika. Po znalezieniu menedżera stosuje się konstrukcję warunkową, aby określić, czy wynagrodzenie pracownika powinno być gromadzone w „zanotowanej liczbie", co wymaga porównania dwóch

wynagrodzeń. Po wykonaniu tej „wewnętrznej” czynności pętla zewnętrzna wznawia kontrolę i przechodzi do następnego pracownika, którego menedżer jest następnie poszukiwany, aż do osiągnięcia końca listy.

Sortowanie bąbelkowe: Przykład

Aby dokładniej zilustrować struktury kontrolne, przyjrzyjmy się algorytmowi sortowania. Sortowanie to jeden z najciekawszych tematów w algorytmice, z którym wiąże się w ten czy inny sposób wiele ważnych zmian. Dane wejściowe do problemu sortowania to nieuporządkowana lista elementów, powiedzmy liczb. Naszym zadaniem jest stworzenie listy posortowanej w porządku rosnącym. Problem można sformułować bardziej ogólnie, zastępując, powiedzmy, listami słów listy liczbowe, z zamiarem ich posortowania według ich leksykograficznej kolejności (tj. jak w słowniku lub książce telefonicznej). Zakłada się, że lista elementów poprzedzona jest jej długością N, a jedynym sposobem na uzyskanie informacji o wielkości tych elementów polega na wykonaniu porównań binarnych; to znaczy porównywać dwa elementy i działać zgodnie z wynikiem porównania. Jeden z wielu znanych algorytmów sortowania nazywa się sortowaniem bąbelkowym. W rzeczywistości sortowanie bąbelkowe jest uważane za zły algorytm sortowania z powodów wyjaśnionych w Części 6. Jest on tutaj używany tylko do zilustrowania struktur kontrolnych. Algorytm sortowania bąbelkowego opiera się na następującej obserwacji. Jeśli pomieszana lista jest przemierzana po kolei, po jednym elemencie na raz i gdy dwa sąsiednie elementy są w złej kolejności (to znaczy, że pierwszy jest większy niż drugi), są one wymieniane, to po zakończeniu przechodzenia , największy element jest na swoim miejscu; mianowicie na końcu listy. Rysunek (a) ilustruje takie przechodzenie dla prostej pięcioelementowej listy.



(Lista została sporządzona od dołu do góry: pierwszy element jest najniższy na rysunku. Strzałki pokazują tylko wymienione elementy, a nie te, które są porównywane.) Oczywiście, sortowanie może poprawić inne nieprawidłowe kolejności poza umieszczeniem maksymalnego elementu w jego ostateczna pozycja. Jednak rysunek (a) pokazuje, że jedno przejście niekoniecznie sortuje listę. Teraz drugie przejście doprowadzi drugi co do wielkości element do właściwego punktu spoczynku, przedostatniej pozycji na liście, jak widać na rysunku (b). Prowadzi to do algorytmu, który wykonuje N-1 takich przejść (dlaczego nie N?), co daje posortowaną listę. Nazwa „bubblesort” pochodzi od sposobu, w jaki duże elementy „wyskakują” na górę listy w miarę postępu algorytmu, zamieniając miejsca na mniejsze elementy, które są przesuwane niżej. Przed bardziej szczegółowym opisem algorytmu należy zauważyć, że drugie przechodzenie nie musi sięgać do ostatniego elementu, ponieważ w momencie rozpoczęcia drugiego przemierzania ostatnia pozycja na liście zawiera już prawowitego dzierżawcę - największy element na liście. Podobnie, trzecie przejście nie musi iść dalej niż pierwsze N-2 elementy. Oznacza to, że bardziej wydajny algorytm przemierzyłby tylko pierwsze N elementów w pierwszym przejściu, pierwsze N-1 w drugim, N-2 w trzecim itd. Powrócimy do algorytmu

sortowania pęcherzyków i to ulepszenie Część 6, ale na razie wystarczy wersja nieulepszona. Algorytm brzmi następująco:

(1) wykonaj następujące N - 1 razy:

(1.1) wskazać na pierwszy element;

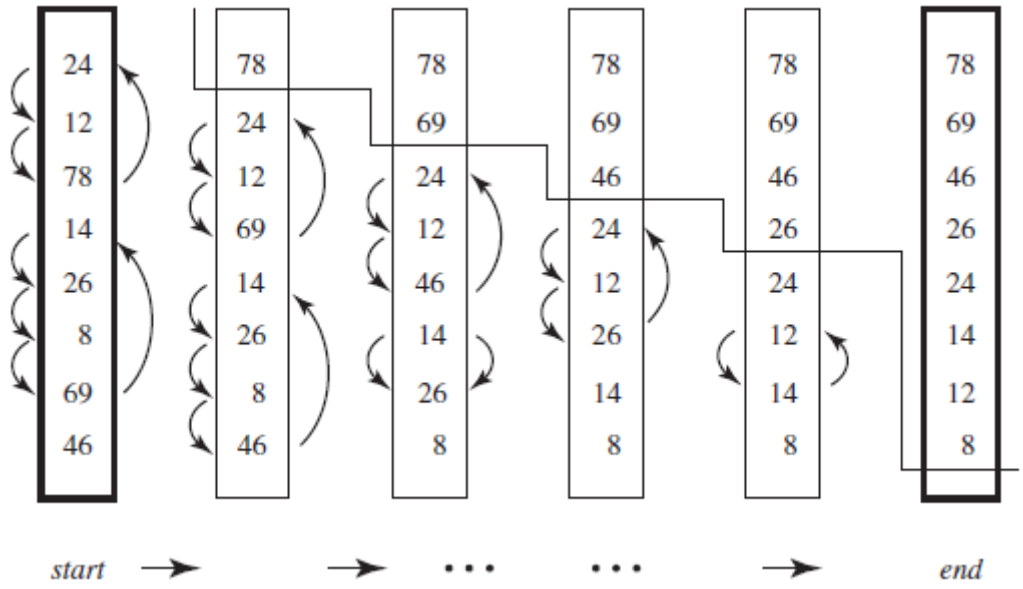
(1.2) wykonaj następujące N - 1 razy:

(1.2.1) porównać wskazany element z następnym elementem;

(1.2.2) jeśli porównywane elementy są w złej kolejności, wymienić je;

(1.2.3) wskazuje na następny element.

Zwróć uwagę, jak jest tutaj używane wcięcie dwupoziomowe. Pierwsza „następna”, w linii (1), obejmuje wszystkie linie zaczynające się od 1, a druga, w linii (1.2), obejmuje te zaczynające się od 1.2. W ten sposób wyraźnie widać zagnieżdżony charakter konstrukcji pętli. Główne kroki podejmowane przez algorytm na ośmiopunktowej liście są zilustrowane na rysunku 2.2, gdzie sytuacja jest przedstawiona tuż przed każdym wykonaniem punktu (1.2).



Elementy pojawiające się nad linią znajdują się w swoich ostatecznych pozycjach. Zauważ, że w tym konkretnym przykładzie dwa ostatnie przejścia (nie pokazane) są zbędne; lista jest sortowana po pięciu, a nie siedmiu przejściach. Należy jednak zauważyć, że jeśli na przykład najmniejszy element jest ostatnim na oryginalnej liście (czyli na górze na naszych ilustracjach), to w rzeczywistości konieczne są wszystkie przejścia N - 1, ponieważ elementy, które mają być „ bulgotać” (elbbubed? …) sprawiają więcej kłopotów niż te, które „bulgotają”.

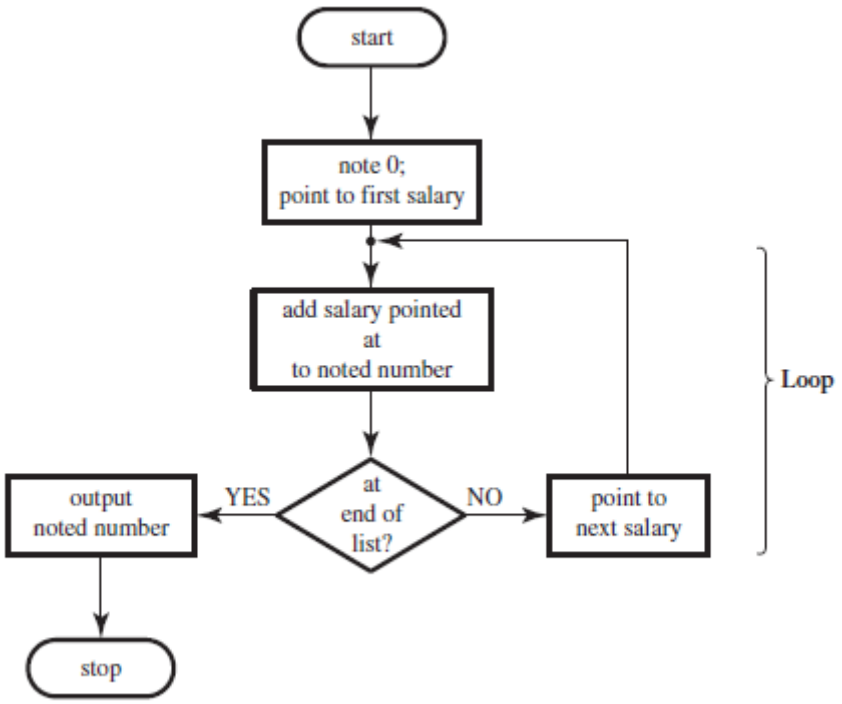
Instrukcja „Goto”

Jest jeszcze inna ważna instrukcja kontrolna, która jest ogólnie nazywana instrukcją goto. Ma ogólną postać „goto G”, gdzie G zaznacza jakiś punkt w tekście algorytmu. W naszych przykładach moglibyśmy napisać, powiedzmy, „goto (1.2)”, instrukcję, która powoduje, że procesor Runaround dosłownie przechodzi do wiersza (1.2) algorytmu i wznawia wykonywanie od tego miejsca. Konstrukcja ta jest

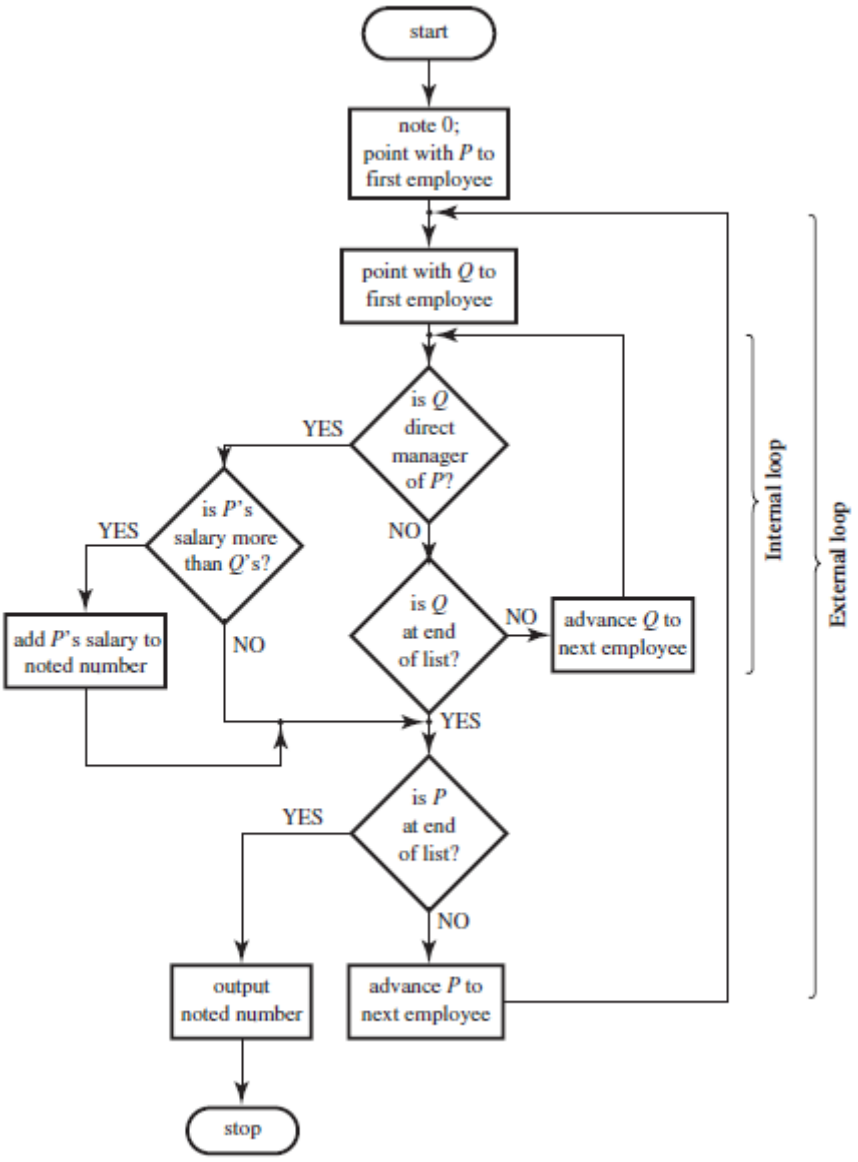
kontrowersyjna z wielu powodów, z których najbardziej oczywisty ma charakter pragmatyczny. Algorytm, który zawiera wiele instrukcji „goto” kierujących sterowanie w tę i z powrotem w splątany sposób, szybko staje się bardzo trudny do zrozumienia. Przejrzystość, jak będziemy argumentować później, jest bardzo ważnym czynnikiem w projektowaniu algorytmicznym. Poza potencjalnym ograniczeniem naszej zdolności do zrozumienia algorytmu, istrukcje „goto” mogą również powodować trudności techniczne. Co się stanie, jeśli instrukcja „goto” skieruje procesor w sam środek pętli? Przykładem tego jest wstawienie instrukcji „goto (1.2.1)” między (1.1) a (1.2) do algorytmu sortowania bąbelkowego. Czy procesor ma wykonać (1.2.1) do (1.2.3), a następnie zatrzymać się, czy też należy wykonać je N-1 razy? Co się stanie, jeśli ta sama instrukcja pojawi się w sekwencji (1.2.1)-(1.2.3)? Problem ten ma swoje źródło w niejednoznaczności wynikającej z próby dopasowania tekstu algorytmu do zaleconego przez niego procesu. Oczywiście istnieje takie dopasowanie, ale ponieważ ustalone algorytmy mogą określać procesy o różnej długości, pojedynczy punkt w tekście algorytmu może być powiązany z wieloma punktami wykonywania odpowiedniego procesu. W związku z tym istrukcje „goto” są w pewnym sensie istotami z natury niejednoznacznymi, a wielu badaczy sprzeciwia się używaniu go swobodnie w algorytmach.

Diagramy dla algorytmów

Wizualne, diagramatyczne techniki są jednym ze sposobów przedstawiania przepływu sterowania algorytmem w jasny i czytelny sposób. Istnieją różne sposoby „rysowania” algorytmów, w przeciwieństwie do ich zapisywania. Jednym z najbardziej znanych jest zapisywanie podstawowych instrukcji w prostokątnych pudełkach, a testów w rombowych pudełkach i używanie strzałek do opisania, w jaki sposób procesor Runaround wykonuje algorytm. Powstałe obiekty nazywane są schematami blokowymi. Rysunek 1 przedstawia schemat blokowy zwykłego algorytmu sumowania wynagrodzeń,



a Rysunek 2 przedstawia schemat bardziej wyrafinowanej wersji, która obejmuje tylko pracowników zarabiających więcej niż ich bezpośredni przełożeni.



Zwróć uwagę na sposób, w jaki iteracja przedstawia się wizualnie jako cykl prostokątów, rombów i strzałek, a zagnieżdżone iteracje są wyświetlane jako cykle w cyklach. To wyjaśnia użycie terminu „pętla”. Schematy blokowe na rysunkach 2 i 3 ilustrują również stosowność terminu „rozgałęzienie”, który był powiązany z instrukcjami warunkowymi. Schematy blokowe mają również wady. Jednym z nich jest fakt, że trudniej jest zachęcić ludzi do przestrzegania niewielkiej liczby „dobrze uformowanych” struktur kontrolnych. Korzystając ze schematów blokowych, łatwo jest ulec pokusie użycia wielu „goto”, ponieważ są one przedstawiane po prostu jako strzałki, takie jak te, które reprezentują pętle „while” lub instrukcje warunkowe. Tak więc to nadużywanie medium schematów blokowych spowodowało, że wielu badaczy zaleciło ich stosowanie z ostrożnością. Innym problemem jest fakt, że wiele rodzajów algorytmów po prostu nie nadaje się w naturalny sposób do graficznego, diagramowego odwzorowania oferowanego przez takie jak schematy blokowe. Powstałe artefakty będą często przypominać spaghetti, zmniejszając, a nie zwiększając, zdolność widza do zrozumienia, co naprawdę się dzieje. Niezależnie od powyższej dyskusji, w Części 14 zobaczymy, że istnieją języki diagramowe (nazwiemy je formalizmami wizualnymi), które są bardzo skuteczne, głównie w kontekście określania zachowania dużych i złożonych systemów. Problemem nie jest opisywanie obliczeń, jak robią to algorytmy, ale określenie reaktywnego i interaktywnego zachowania w czasie.

Podprogramy lub procedury

Załóżmy, że otrzymujemy obszerny tekst i chcemy dowiedzieć się, jak chciwy jest jego autor, licząc zdania, które zawierają słowo „pieniądze”. W takich przypadkach nie interesuje nas liczba wystąpień słowa „pieniądze”, ale liczba zdań, w których występuje. Można zaprojektować algorytm, który będzie przechodził przez tekst w poszukiwaniu „pieniędzy”. Po znalezieniu takiego zdarzenia biegnie naprzód, szukając końca zdania, które dla naszych celów przyjmuje się jako kropkę, po której następuje spacja; to jest ". ” kombinacja. Po znalezieniu końca zdania algorytm dodaje 1 do licznika (czyli „zanotowanej liczby”, jak w algorytmie sumowania wynagrodzeń), który został zainicjowany na 0 na początku. Następnie wznawia poszukiwanie „pieniędzy” od początku następnego zdania; to znaczy od litery następującej po kombinacji. Oczywiście algorytm musi cały czas zwracać uwagę na koniec tekstu, aby po jego osiągnięciu mógł wypisać wartość licznika. Algorytm przyjmuje postać zewnętrznej pętli, której zadaniem jest liczenie odpowiednich zdań. W tej pętli znajdują się dwa wyszukiwania, jedno dla „pieniądze”, a drugie dla „. ”, z których każda sama w sobie stanowi pętlę. Chodzi o to, że dwie wewnętrzne pętle są bardzo podobne; w rzeczywistości obaj robią dokładnie to samo – szukają sekwencji symboli w tekście. Obecność obu pętli w algorytmie wyraźnie działa, ale możemy zrobić to lepiej. Pomysł polega na napisaniu pętli wyszukiwania tylko raz, z parametrem, który z założenia zawiera konkretną kombinację poszukiwanych symboli. Ten segment algorytmiczny nazywa się podprogramem lub procedurą i jest aktywowany (lub wywoływany lub wywoływany) dwukrotnie w głównym algorytmie, raz z parametrem „pieniądze”, a raz z „. ” kombinacja. Tekst podprogramu jest dostarczany osobno i odnosi się on do zmiennego parametru poprzez nazwę, powiedzmy X. Podprogram zakłada, że wskazujemy jakieś miejsce w tekście wejściowym i może wyglądać następująco: podprogram szukaj X :

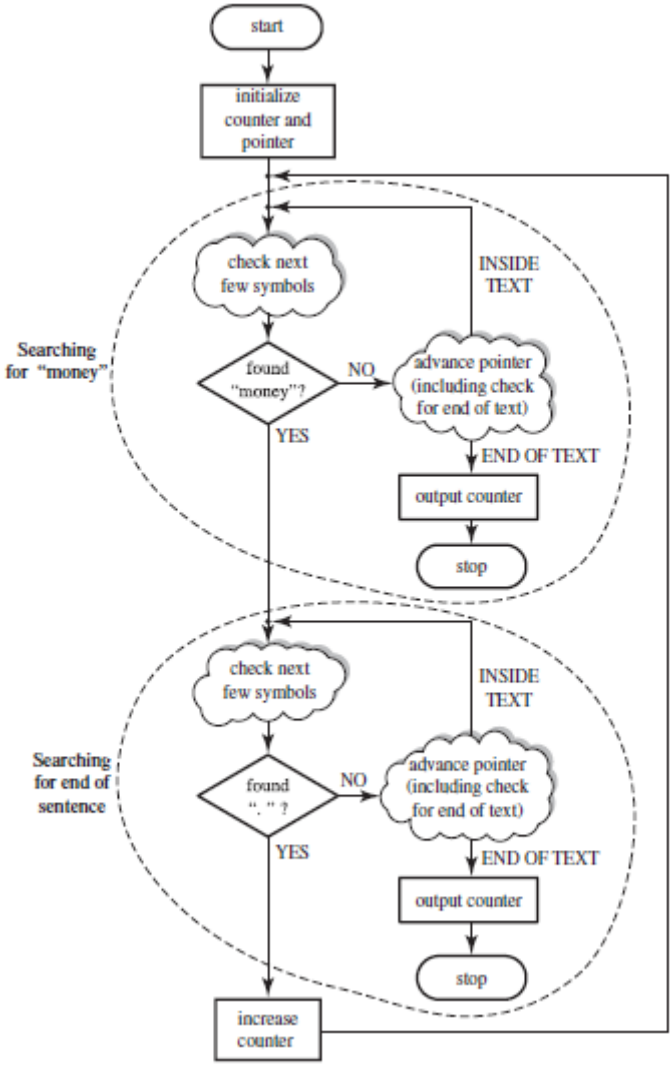
(1) wykonaj następujące czynności, aż wskażesz kombinację X lub dotrzesz do końca tekstu:

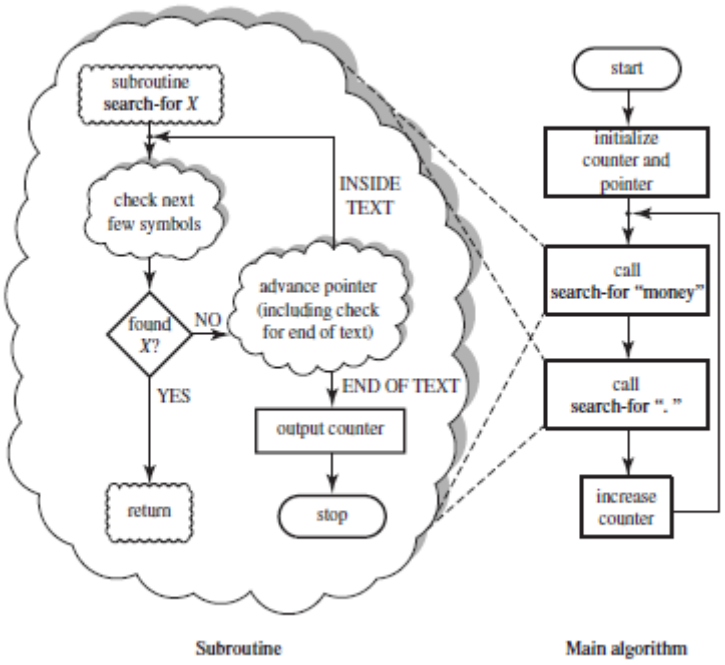
(1.1) przesunąć wskaźnik o jeden symbol w tekście;

(2) po osiągnięciu końca tekstu wyprowadź wartość licznika i zatrzymaj się;

(3) w przeciwnym razie wróć do głównego algorytmu.

Główna część algorytmu użyje podprogramu wyszukiwania dwukrotnie, za pomocą instrukcji w postaci „wywołaj wyszukiwanie pieniędzy” i „wywołaj wyszukiwanie”. ’ ” Porównaj rysunek 1 z rysunkiem 2, na którym schematycznie pokazano wersję z podprogramem.


start
initialize counter and pointer
Searching for "money"
check next few symbols
INSIDE TEXT
found "money"?
NO
advance pointer (including check for end of text)
YES
END OF TEXT
output counter
stop
Searching for end of sentence
check next few symbols
INSIDE TEXT
found ". " ?
NO
advance pointer (including check for end of text)
YES
END OF TEXT
output counter
stop
increase counter



Procesor, który działa, będzie teraz musiał być nieco bardziej wyrafinowany. Gdy zostaniesz poproszony o „wywołanie" podprogramu, zatrzyma to, co robił, zapamiętaj, gdzie to było, wybierz parametr(y), że tak powiem, i przejdzie do tekstu podprogramu. Następnie zrobi to, co każe mu podprogram, używając bieżącej wartości parametru, gdziekolwiek podprogram odwołuje się do tego parametru za pomocą swojej wewnętrznej nazwy (w naszym przykładzie X). Jeśli i kiedy podprogram każe mu powrócić, to właśnie to zrobi; mianowicie powróci do punktu następującego po „wezwaniu", które doprowadziło go w pierwszej kolejności do podprogramu, i stamtąd wznowi swoje obowiązki.

Cnoty podprogramów

Oczywiście podprogramy mogą być bardzo ekonomiczne, jeśli chodzi o rozmiar algorytmu. Nawet w tym prostym przykładzie pętla wyszukiwania jest napisana raz, ale jest używana dwa razy, podczas gdy bez niej algorytm musiałby oczywiście uwzględnić dwie szczegółowe wersje tej pętli. Ta ekonomia staje się znacznie ważniejsza dla złożonych algorytmów z wieloma podprogramami, które są wywoływane z różnych miejsc. Ponadto algorytm może zawierać podprogramy, które wywołują inne podprogramy i tak dalej. W ten sposób struktura algorytmiczna otrzymuje nowy wymiar; istnieją nie tylko zagnieżdżone pętle, ale zagnieżdżone podprogramy. Co więcej, pętle, instrukcje warunkowe, konstrukcje sekwencyjne, instrukcje „goto", a teraz podprogramy, mogą być przeplatane, dając algorytmy o rosnącej złożoności strukturalnej. Ekonomia to jednak nie jedyna zaleta podprogramów. Podprogram może być postrzegany jako „kawał" materiału algorytmicznego, pewnego rodzaju cegiełka, która po utworzeniu może być wykorzystana w innym fragmencie algorytmicznym przez pojedynczą instrukcję. To tak, jakby powiedzieć, że rozszerzyliśmy nasz repertuar dozwolonych instrukcji elementarnych. W przykładzie liczenia „pieniędzy", gdy procedura wyszukiwania jest już dostępna (a nawet wcześniej, o ile zdecydowano, że taka procedura zostanie ostatecznie napisana), instrukcja „call search-for 'abc' " jest dla każdego praktyczny cel, nowa podstawowa instrukcja. Tak więc podprogramy są jednym ze sposobów, w jaki możemy tworzyć własne abstrakcje, zgodnie z

konkretnym problemem, który próbujemy rozwiązać. To bardzo potężny pomysł, ponieważ nie tylko skraca algorytmy, ale także czyni je przejrzystymi i dobrze zorganizowanymi. Przejrzystość i struktura, jak wielokrotnie podkreślano, mają ogromne znaczenie w algorytmice i wiele wysiłku poświęca się znalezieniu sposobów narzucenia ich projektantom algorytmów.

W ten sam sposób, w jaki użytkownik programu komputerowego zwykle nie wie nic o algorytmach, których używa, podprogram może być używany jako „czarna skrzynka", nie wiedząc, jak jest zaimplementowany. Wszystko, co użytkownik podprogramu musi wiedzieć, to co robi, ale nie jak to robi. To znacznie upraszcza problem, zmniejszając ilość szczegółów, o których należy pamiętać. Za pomocą podprogramów można stopniowo, krok po kroku, rozwijać złożony algorytm. Typowy problem algorytmiczny wymaga w pełni szczegółowego rozwiązania, które wykorzystuje tylko dozwolone czynności elementarne. Projektant może stopniowo dążyć do tego celu, najpierw opracowując algorytm wysokiego poziomu, który wykorzystuje „podstawowe" instrukcje, których nie ma w książce. W rzeczywistości są to wywołania podprogramów, które projektant ma na myśli, a które zostaną napisane później (lub być może wcześniej). Te podprogramy z kolei mogą korzystać z innych instrukcji, które, nie będąc wystarczająco elementarnymi, są ponownie uważane za wywołania podprogramów, które zostaną ostatecznie napisane. W pewnym momencie wszystkie podstawowe instrukcje są na wystarczająco niskim poziomie, aby znaleźć się wśród tych wyraźnie dozwolonych. Wtedy kończy się stopniowy proces rozwoju. Takie podejście daje początek albo projektowi „z góry na dół", który, jak już opisano, idzie od ogółu do konkretu, albo konstrukcji „od dołu do góry", w której przygotowuje się potrzebne podprogramy, a następnie projektuje bardziej ogólne procedury, które je nazywają, pracując w ten sposób od szczegółowych do ogólnych. Nie ma powszechnej zgody co do tego, które z nich jest lepszym podejściem do projektowania algorytmicznego. Powszechnie uważa się, że należy użyć jakiejś mieszanki. Wracając na chwilę do gastronomii, przygotowanie „mieszanki czekoladowej" może być dobrym kandydatem na podprogram w przepisie na mus czekoladowy z Części 1. To pozwoliłoby nam opisać przepis w następującym sposób, w którym każda z czterech instrukcji jest traktowana jako wywołanie podprogramu (lub raczej podprzepisu), którego tekst byłby następnie napisany osobno:

(1) przygotować mieszankę czekoladową;

(2) mieszanka do wytworzenia mieszanki czekoladowo-żółtkowej;

(3) przygotować piankę z białek jaj;

(4) wymieszaj oba, aby uzyskać mus.

Warto zaznaczyć, że książka, z której zaczerpnięto przepis na mus, dość obszernie wykorzystuje podprogramy. Na przykład opisujemy szereg przepisów, których składniki zawierają pozycje takie jak Przepis na Słodkie Ciasto, Rogaliki czy Ciasto Francuskie, dla których użytkownik jest odsyłany do wcześniej podanych przepisów dedykowanych tym właśnie produktom. Można śmiało powiedzieć, że nie można przecenić mocy i elastyczności zapewnianych przez podprogramy

Rekurencja

Jednym z najbardziej użytecznych aspektów podprogramów, który dla wielu ludzi jest również jednym z najbardziej mylących, jest rekurencja. Rozumiemy przez to po prostu zdolność podprogramu lub procedury do wywołania samej siebie. Może to zabrzmieć absurdalnie, skoro jak, u licha, nasz procesor przybliża się do rozwiązania problemu, gdy w trakcie próby rozwiązania tego problemu mówi się, żeby zostawić wszystko i zacząć rozwiązywać ten sam problem od nowa? Poniższy przykład powinien nam pomóc w rozwiązaniu tego paradoksu i może pomóc, jeśli powiemy na początku, że rozwiązanie opiera

się na tej samej właściwości algorytmów, o których mowa wcześniej: ten sam tekst (w tym przypadku podprogram rekurencyjny) może odpowiadać wielu częściom opisanego przez nią procesu. Konstrukcje iteracyjne są jednym ze sposobów mapowania długich procesów na krótkie teksty; podprogramy rekurencyjne to kolejny. Przykład opiera się na dość starożytnej zagadce znanej jako Wieże Hanoi, wywodzącej się od hinduskich kapłanów z wielkiej świątyni Benares. Załóżmy, że otrzymaliśmy trzy wieże, a żeby być bardziej skromnym, trzy kołki, A, B i C. Na pierwszym kołku, A, są trzy pierścienie ułożone w malejącym porządku wielkości, podczas gdy pozostałe są puste. Jesteśmy zainteresowani w przenoszeniu pierścieni z A do B, być może przy użyciu C w procesie. Zgodnie z zasadami gry, pierścienie należy przesuwać pojedynczo i w żadnym momencie nie można umieszczać większego pierścienia na mniejszym. Tę prostą zagadkę można rozwiązać w następujący sposób:

przenieś A do B;

przenieś A do C;

przenieś B do C;

przenieś A do B;

przenieś C do A;

przenieś C do B;

przenieś A do B.

Zanim przejdziemy dalej, powinniśmy najpierw przekonać się, że te siedem akcji naprawdę działa, a następnie powinniśmy wypróbować tę samą łamigłówkę z czterema, a nie trzema pierścieniami na kołku A (liczba kołków się nie zmienia). Umiarkowana ilość pracy powinna wystarczyć, aby odkryć sekwencję 15 akcji „przesuń X na Y”, które rozwiązują wersję czteropierścieniową. Takie łamigłówki są intelektualnie trudne, a ci, którzy lubią łamigłówki, mogą chcieć rozważyć oryginalną hinduską wersję, która zawiera te same trzy kołki, ABC, ale z nie mniej niż 64 pierścieniami na A. Jak zobaczymy, wynalazcy układanki nie byli całkowicie oderwali się od rzeczywistości, gdy stwierdzili, że świat skończy się, gdy wszystkie 64 pierścienie zostaną prawidłowo ułożone na kołku B. Jednak nie mamy tutaj do czynienia z zagadkami, ale z algorytmiką, a w konsekwencji bardziej interesuje nas ogólny problem algorytmiczny związany z Wieżami Hanoi niż z tym czy innym konkretnym przypadkiem. Dane wejściowe to dodatnia liczba całkowita N, a pożądanym wynikiem jest lista działań „przenieś X do Y”, które, jeśli zostaną wykonane, rozwiązują zagadkę obejmującą N pierścieni. Oczywiście rozwiązaniem tego problemu musi być algorytm, który działa dla każdego N, tworząc listę działań spełniających ograniczenia; w szczególności podążanie za nimi nigdy nie powoduje umieszczenia większego pierścienia na mniejszym. To jest problem, który naprawdę powinniśmy próbować rozwiązać, ponieważ gdy algorytm jest już dostępny, każdą instancję łamigłówki, niezależnie od tego, czy jest to wersja trzy-, cztero- czy 3078-pierścieniowa, można rozwiązać po prostu uruchamiając algorytm z żądaną liczbę dzwonków jako dane wejściowe. Jak to się robi? Odpowiedź jest prosta: dzięki magii rekurencji.

Rozwiązanie dla wież Hanoi

Przedstawiony tutaj algorytm realizuje zadanie przeniesienia N pierścieni z A do B przez C w następujący sposób. Najpierw sprawdza, czy N wynosi 1, w którym to przypadku po prostu przesuwa jeden pierścień, o który został poproszony, do miejsca docelowego (a dokładniej, wyświetla opis jednego ruchu, który wykona zadanie), i następnie wraca natychmiast. Jeśli N jest większe niż 1, najpierw przesuwa górne N-1 pierścieni z A do „dodatkowego” kołka C przy użyciu tej samej procedury

rekurencyjnej; następnie podnosi jedyny pozostały pierścień w A, który musi być największym pierścieniem (dlaczego?) i przenosi go do miejsca docelowego, B; następnie, ponownie rekursywnie, przesuwa pierścienie N-1, które wcześniej „przechowywał” w C do ich ostatecznego miejsca przeznaczenia, B. Że ten algorytm przestrzega reguł gry jest trochę trudny do zauważenia, ale przy założeniu, że dwa procesy związane z pierścieniami N-1 zawierają tylko legalne ruchy, dość łatwo zauważyć, że tak samo jest z całym procesem przemieszczania pierścieni N. Oto algorytm. Jest napisany jako procedura rekurencyjna, której tytuł i parametry (jest ich cztery) mówią same za siebie:

podprogram przesuń N z X na Y za pomocą Z:

(1) jeśli N wynosi 1, to wypisz „przenieś X do Y”;

(2) w przeciwnym razie (tj. jeśli N jest większe niż 1) wykonaj następujące czynności:

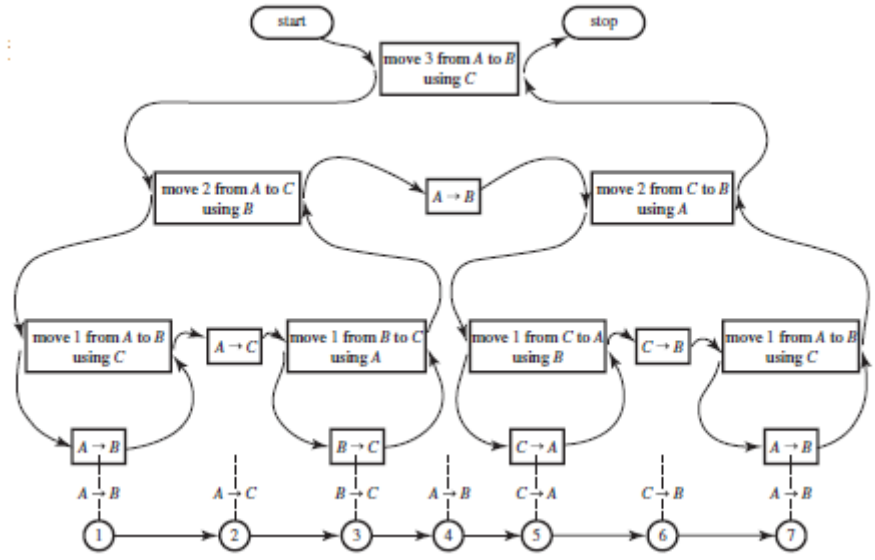
(2.1) wywołaj przeniesienie N − 1 z X do Z używając Y ;

(2.2) wyjście „przesuń X do Y”;

(2.3) wywołaj przeniesienie N − 1 z Z do Y za pomocą X;

(3) powrót.

Aby zilustrować działanie tej procedury, która na pierwszy rzut oka może wyglądać dość śmiesznie, moglibyśmy spróbować uruchomić ją, gdy N wynosi 3; czyli symulowanie pracy procesora, gdy są trzy pierścienie. Należy to zrobić wykonując „główny algorytm” składający się z pojedynczej instrukcji: wywołaj ruch 3 z A do B za pomocą C. Symulację należy przeprowadzić ostrożnie, ponieważ nazwy parametrów X, Y i Z zaczynają się dość niewinnie, jako są A, B i C, ale zmieniają się za każdym razem, gdy procesor ponownie wejdzie w procedurę. Jakby, aby było bardziej zagmatwane, wznawiają swoje stare wartości, gdy „rozmowa” się kończy, a procesor wraca do miejsca, z którego pochodzi. Rysunek



pomaga zilustrować kolejność działań w tym przypadku. Zauważ, że procesor (i my też, jeśli rzeczywiście próbujemy symulować procedurę) musi teraz pamiętać nie tylko skąd pochodzi, aby powrócić do właściwego miejsca, ale także jakie wartości nazw parametrów były przed swoje nowe zadanie, aby prawidłowo wznowić pracę. Rysunek jest zorganizowany tak, aby odzwierciedlić

głębokość rekurencji i odpowiedni sposób zmiany wartości parametrów. Strzałki reprezentują podróże procesora: strzałki w dół odpowiadają wywołaniom, strzałki w górę do powrotów, a strzałki poziome do prostego sekwencjonowania. Jak za dotknięciem czarodziejskiej różdżki okazuje się, że ostateczna sekwencja działań jest identyczna z tą, do której doszło wcześniej metodą prób i błędów. Jak się okazuje, problem Wież Hanoi dopuszcza inne rozwiązanie algorytmiczne, które wykorzystuje prostą iterację i w ogóle nie wywołuje wywołań podprogramów, i może być wykonane przez małe dziecko!

Ekspresyjna moc struktur kontrolnych

Czy możemy zrobić tylko z kilkoma prostymi strukturami kontrolnymi? Odpowiedź brzmi tak. Zidentyfikowano różne minimalne zestawy struktur kontrolnych, co oznacza, że w pewnych technicznych aspektach inne struktury kontrolne można zastąpić odpowiednimi kombinacjami tych ze zbioru minimalnego, tak że w praktyce są one jedyne potrzebne. Dobrze znany zestaw minimalny składa się z sekwencjonowania (a następnie), rozgałęzienia warunkowego (jeśli-to) i pewnego rodzaju konstrukcji pętli nieograniczonej (na przykład while-do). Nietrudno np. pokazać, że instrukcję w postaci „powtórz A, aż Q będzie prawdziwe", można zastąpić „zrób A, a potem, gdy Q jest fałszywe, wykonaj A", aby w obecności konstrukcje „podczas działania", które można wykonać bez konstrukcji „powtarzaj aż do". W podobnym duchu możliwe jest całkowite wyeliminowanie stwierdzeń „goto" z dowolnego algorytmu, kosztem jednak pewnego rozszerzenia algorytmu i zmiany jego pierwotnej struktury. Podobnie jest w ścisłym sensie, w którym wszystko, co można zrobić za pomocą podprogramów i rekurencji, można zrobić tylko za pomocą prostych pętli. Wykorzystanie tego wyniku do pozbycia się podprogramów danego algorytmu wiąże się jednak z dodaniem do algorytmu znacznej „maszyny" (w postaci nowych instrukcji elementarnych). Można wykazać, że jeśli taka maszyneria nie jest dozwolona, to podprogramy rekurencyjne są potężniejsze niż iteracja, tak że pewne rzeczy można zrobić tylko za pomocą tych pierwszych. Tematy te zostały tu poruszone tylko w sposób najbardziej powierzchowny, aby dać ci wyczucie istotnych kwestii, które cię interesują.

Typy danych i struktury danych

Wiemy więc, jak wygląda algorytm i jak, biorąc pod uwagę dane wejściowe, procesor wykonuje proces, który opisuje algorytm. Jednak byliśmy dość niejasni co do obiektów manipulowanych przez algorytm. Mieliśmy listy, słowa i teksty, a także zabawne rzeczy, takie jak „zanotowane liczby", które wzrosły i „wskaźniki" które poczyniły postępy. Jeśli chcemy posunąć się do skrajności, możemy również powiedzieć, że mieliśmy mąkę, cukier, ciasta i mus czekoladowy, a także wieże, kołki i krążki. Obiekty te stanowiły nie tylko wejścia i wyjścia algorytmu, ale także akcesoria pośrednie, które zostały skonstruowane i używane w trakcie jego życia, takie jak liczniki („zanotowane liczby") i wskaźniki. Dla wszystkich z nich używamy ogólnego terminu dane. Elementy danych występują w różnych odmianach lub mogą być różnego rodzaju. Niektóre z najczęstszych typów danych występujących w algorytmach wykonywanych przez komputery to różnego rodzaju liczby (całkowite, dziesiętne, binarne itd.) oraz słowa zapisane w różnych alfabetach. W rzeczywistości liczby mogą być również rozumiane jako słowa; Na przykład liczby całkowite dziesiętne to „słowa" w alfabecie składającym się z cyfr 0, 1, 2, . . . , 9 i liczby binarne używają alfabetu składającego się tylko z 0 i 1. Korzystne jest jednak trzymanie takich typów oddzielnie, nie tylko dla przejrzystości i porządku, ale także dlatego, że każdy typ dopuszcza własny specjalny zestaw dozwolonych operacji lub akcji. Wyliczanie samogłosek w liczbie nie ma większego sensu niż mnożenie dwóch wyrazów! I tak, naszym algorytmom trzeba będzie powiedzieć, jakimi obiektami mogą manipulować i jakie manipulacje są dozwolone. Manipulacja obejmuje nie tylko wykonywanie operacji, ale także zadawanie pytań, np. sprawdzanie, czy liczba jest parzysta lub czy dwa słowa zaczynają się na tę samą literę. Obserwacje te wydają się całkiem naturalne w świetle naszej dyskusji na temat operacji elementarnych w części 1 i emanują faktami dotyczącymi możliwości komputerów do manipulacji bitami. Tutaj przyjmujemy za pewnik podstawowe typy danych oraz

operacje i testy z nimi związane. Interesuje nas to, w jaki sposób algorytmy mogą organizować, zapamiętywać, zmieniać i uzyskiwać dostęp do zbiorów danych. Podczas gdy struktury kontrolne służą do informowania procesora, dokąd powinien się udać, struktury danych i operacje na nich organizują elementy danych w sposób, który umożliwia mu robienie tego, co powinien zrobić, gdy się tam dostanie. Świat struktur danych jest tak samo bogaty w poziomy abstrakcji, jak świat struktur kontrolnych. W rzeczywistości przydatna sztuczka mentalna, która jest podstawą paradygmatu programowania obiektowego, pokazuje, że możemy się między nimi przełączać!

Zmienne, czyli małe pudełka

Pierwszymi obiektami zainteresowania są zmienne. Na przykład w algorytmie sumowania wynagrodzeń użyliśmy „zanotowanej liczby", która została najpierw zainicjowana na 0, a następnie użyta do akumulacji sumy wynagrodzeń pracowników. Właściwie używaliśmy zmiennej. Zmienna nie jest liczbą, słowem ani jakimś innym elementem danych. Może być raczej postrzegana jako małe pudełko lub komórka, w której można przechowywać pojedynczy przedmiot. Możemy nadać zmiennej nazwę, powiedzmy X, a następnie użyć instrukcji typu „wstaw 0 w X" lub „zwiększ zawartość X o 1.". Możemy również zadawać pytania dotyczące zawartości X, na przykład „czy X zawiera liczbę parzystą?" Zmienne są bardzo podobne do pokoi hotelowych; różne osoby mogą zajmować pokój w różnym czasie, a osobę znajdującą się w pokoju można określić frazą „mieszkańca pokoju 326". Termin „326" to nazwa pomieszczenia, podobnie jak X to nazwa zmiennej. To użycie słowa „zmienna" do oznaczenia komórki, która może zawierać różne wartości w różnym czasie, jest niepodobne do znaczenia zmiennej w matematyce, gdzie oznacza pojedynczą (zwykle nieznaną) wartość. W Części 3 omówimy paradygmat programowania funkcyjnego, który nie zajmuje się komórkami, ale bezpośrednio wartościami, jak w matematyce. Algorytmy zazwyczaj wykorzystują wiele zmiennych o różnych nazwach i do bardzo różnych celów. Na przykład w algorytmie sortowania bąbelkowego szczegółowa wersja może używać jednej zmiennej do zliczania, ile razy wykonywana jest pętla zewnętrzna, innej dla pętli wewnętrznej i trzeciej, aby pomóc w wymianie dwóch elementów na liście. Aby dokonać wymiany, jeden element jest na chwilę „wkładany do pudełka", drugi na swoje miejsce, a następnie „pudełkowy" element jest umieszczany na pierwotnym miejscu drugiego elementu. Bez użycia zmiennej wydaje się, że nie ma sposobu na zachowanie pierwszego elementu bez jego utraty. Ilustruje to użycie zmiennych jako pamięci lub przechowywania w algorytmie. Oczywiście fakt, że elementy są wymieniane wielokrotnie w jednym przebiegu sortowania bąbelkowego, nie oznacza, że potrzebujemy wielu zmiennych - za każdym razem można użyć tego samego „pudełka". Wynika to z faktu, że wszystkie wymiany w algorytmie sortowania pęcherzyków są rozłączne; żadna wymiana nie rozpoczyna się przed zakończeniem poprzedniej. W ten sposób zmienne można ponownie wykorzystać. Gdy w praktyce używa się zmiennych, wyrażenie „zawartość" jest zwykle pomijane i piszemy takie rzeczy jak X ← 0 (czytaj „X dostaje 0" lub „ustaw X na 0"), aby ustawić początkową wartość (czyli zawartość z) X na 0 i X ← X + 1 (odczytaj „X otrzymuje X + 1"), aby zwiększyć wartość X o 1. Ta ostatnia instrukcja, na przykład, mówi procesorowi, aby „odczytał" liczbę znalezioną w X , zwiększ go o jeden i zastąp go wynikiem.

Wektory lub listy

Przyjrzyjmy się bliżej naszej liście pracowników. Taka lista może być postrzegana po prostu jako wiele elementów danych, które możemy zdecydować się zachować lub przechowywać w wielu zmiennych, powiedzmy X, Y, Z, . . . To oczywiście nie pozwoliłoby algorytmowi o stałym rozmiarze „przebiegać" przez listę, której długość może być różna, ponieważ każdy element na liście musiałby się odnosić w algorytmie za pomocą unikalnej nazwy. Dłuższe listy wymagałyby większej liczby nazw zmiennych, a tym samym dłuższych algorytmów. Potrzebujemy sposobu odwoływania się do wielu elementów w jednolity sposób. Potrzebujemy list zmiennych, które można „przejrzeć" lub uzyskać do nich dostęp w

inny sposób, ale bez konieczności jawnego nazywania każdego z ich elementów. aby móc „wskazywać” na elementy na tych listach, odnosić się do „następnego” lub „poprzedniego” elementu i tak dalej. Do tych celów wykorzystujemy wektory, zwane również tablicami jednowymiarowymi. Jeśli zmienna jest jak pokój hotelowy, wektor można traktować jako cały korytarz lub piętro w hotelu. Pokoje 301, 302, . . . , 346 mogą być indywidualnymi „zmiennymi”; do każdego można się odwoływać osobno, ale w przeciwieństwie do prostego zbioru zmiennych, cały korytarz lub piętro ma również nazwę (powiedzmy „piętro 3”), a dostęp do znajdujących się w nim pomieszczeń można uzyskać za pomocą ich indeksu. 15. pokój wzdłuż tego korytarza to 315, a X pokój to 3X. Oznacza to, że możemy użyć zmiennej do indeksowania wektora. Zmiana wartości X może być wykorzystana do odniesienia się do zawartości różnych elementów w wektorze. Notacja stosowana w praktyce oddziela nazwę wektora od jego indeksu nawiasami; piszemy V[6] dla szóstego elementu wektora V i analogicznie V[X] dla elementu V, którego indeksem jest bieżąca wartość zmiennej X. Możemy nawet napisać V[X + 1], który odnosi się do element następujący po V[X] na liście. (Zauważ, że V[X + 1] i V[X] + 1 oznaczają dwie zupełnie różne rzeczy!)

W algorytmie bubblesort możemy na przykład użyć zmiennej X do sterowania pętlą wewnętrzną, a jednocześnie może ona podwoić się jako wskaźnik do wektora V sortowanych elementów. Oto jak może wyglądać wynikowa (bardziej zwięzła, a także bardziej precyzyjna) wersja algorytmu, gdzie „<” oznacza „jest mniejsze niż”:

(1) wykonaj następujące N - 1 razy:

(1.1) X ← 1;

(1.2) podczas gdy X < N wykonaj następujące czynności:

(1.2.1) jeśli V[X + 1] < V[X] to je zamień;

(1.2.2) X ← X + 1.

Zachęcamy do zmodyfikowania tego algorytmu, uwzględniając wspomnianą wcześniej obserwację, tak aby przy każdym przejściu pętli zewnętrznej liczbę elementów sprawdzanych w pętli wewnętrznej można było zmniejszyć o 1. Wektory reprezentujące listy elementów mają wiele zastosowań. Książka telefoniczna to lista, podobnie jak słowniki, akta osobowe, opisy inwentarza, wymagania dotyczące kursów i tak dalej. W pewnym sensie wektor jako struktura danych jest ściśle powiązany z pętlą jako strukturą kontrolną. Przechodzenie przez wektor (w celu inspekcji, wyszukiwania, sumowania itp.) jest zwykle wykonywane za pomocą pojedynczej konstrukcji iteracyjnej. Tak jak pętla jest strukturą kontrolną opisującą długie procesy, tak wektor jest strukturą danych do reprezentowania długich list elementów danych. Istnieją również specjalne „indeksowane” wersje iteracyjnych konstrukcji sterujących, dostosowane do przechodzenia przez wektory. Na przykład możemy napisać:

dla X od 1 do 100 wykonaj następujące czynności

który jest podobny do:

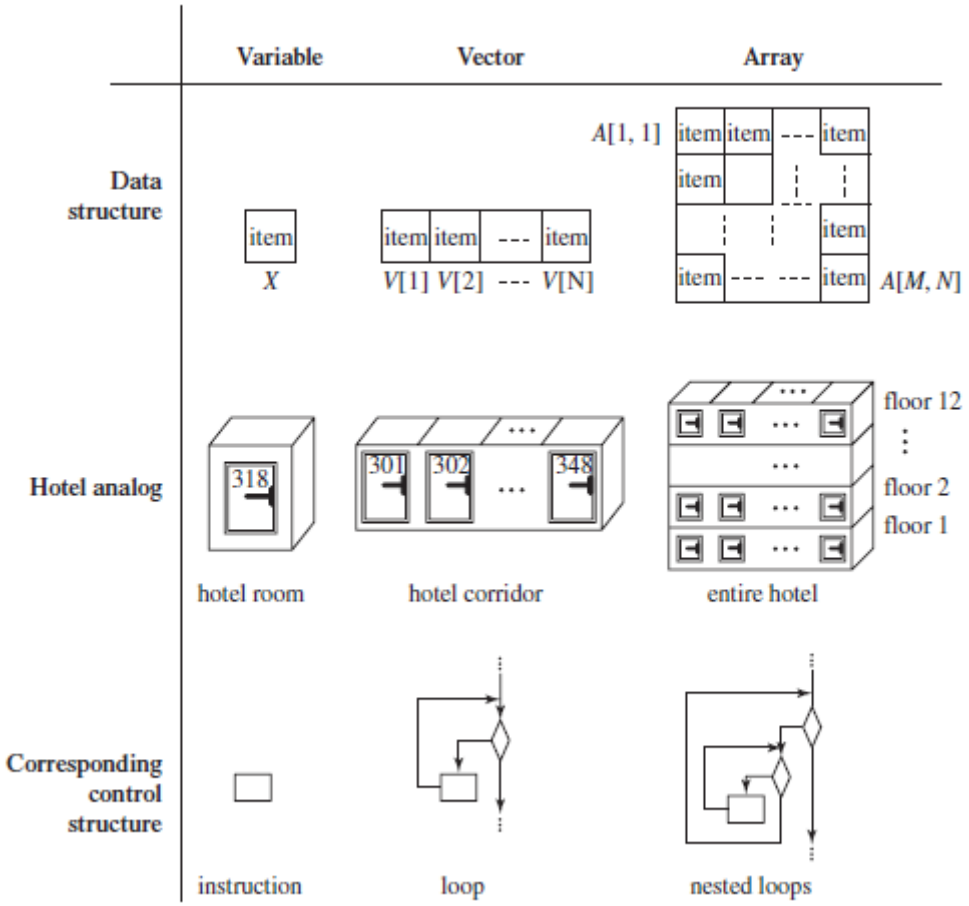
wykonaj następujące 100 razy

z tym wyjątkiem, że w przypadku pierwszego możemy odnieść się bezpośrednio do X-tego elementu w wektorze w powtarzającym się segmencie, podczas gdy w przypadku drugiego nie możemy.

Tablice lub tabele

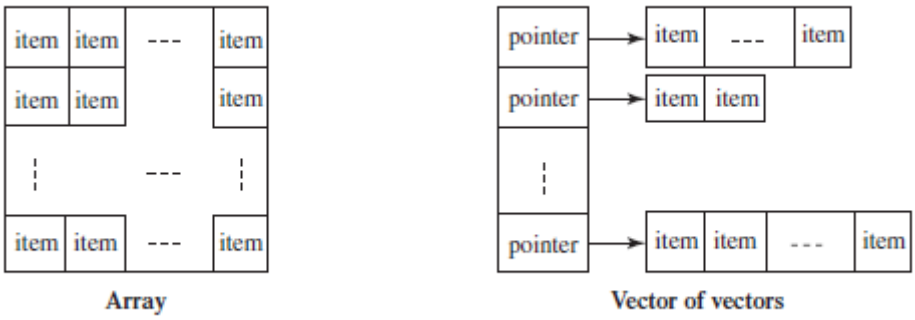
W wielu przypadkach wygodnie jest ułożyć dane nie w postaci prostej, jednowymiarowej listy, ale w formie tabeli. Odpowiednia algorytmiczna struktura danych nazywana jest macierzą, dwuwymiarową

tablicą lub po prostu tablicą. Również tutaj aplikacje są obfite. Standardowa tabliczka mnożenia drugiej klasy to tablica 10 na 10, w której element danych w każdym punkcie jest iloczynem indeksów wiersza i kolumny; listę uczniów wykreśloną na liście kursów można traktować jako tablicę, w której elementy danych są oceną ucznia z kursu; siatka szerokości i długości geograficznej Ziemi może być podstawą tablicy podającej wysokości w każdym punkcie przecięcia i tak dalej. Odwołanie się do elementu tablicy jest zazwyczaj realizowane za pomocą dwóch indeksów, wiersza i kolumny. Piszemy A[5, 3] dla elementu znajdującego się w wierszu 5 i kolumnie 3, tak że np. jeśli A jest tabliczką mnożenia, to wartość A[5, 3] wynosi 15. Tak jak poprzednio, możemy zapisać A[X, Y], a także A[X + 4, 17 − Y] i tym podobne. Jeśli zmienna jest jak pokój hotelowy, a wektor jak hotelowy korytarz/piętro, to macierz lub tablica jest jak cały hotel. Jego „rzędy" to różne piętra, a „kolumny" reprezentują lokalizacje wzdłuż korytarza/piętra. Jeśli wektor jako struktura danych odpowiada pętli jako strukturze kontrolnej, to tablica odpowiada zagnieżdżonym pętlom



Przechodzenie przez całą gamę ocen uczniów można osiągnąć za pomocą zewnętrznej pętli przebiegającej przez wszystkich uczniów i wewnętrznej, przebiegającej przez wszystkie oceny danego ucznia lub odwrotnie. Nie zawsze jest tak, że dane mogą być uporządkowane w ściśle prostokątnym formacie. Weźmy przykład studentów i kursów; różni studenci mogą być powiązani nie tylko z różnymi kursami, ale także z różną ich liczbą. Nadal możemy używać tablicy (wystarczająco szerokiej, aby pomieścić maksymalną liczbę możliwych kursów), ale pozostawiając jej części puste, gdziekolwiek uczeń pytanie nie przeszło danego kursu. Alternatywnie możemy użyć nowej struktury danych,

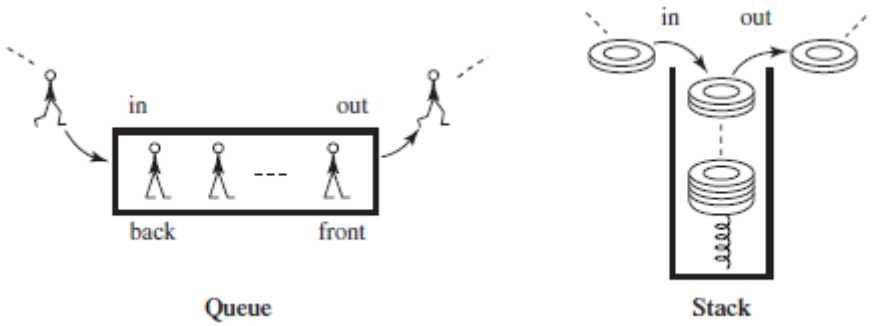
składającej się z wektora, którego elementy same w sobie wskazują na wektory o różnej długości. Różnicę ilustruje rysunek



Algorytmy mogą używać bardziej skomplikowanych tablic, na przykład o większej wymiarowości. Trójwymiarowa tablica jest jak sześcian z trzema indeksami potrzebnymi do wskazania elementu. W razie potrzeby możemy wykorzystać tylko specjalne części tablic, takie jak górna trójkątna część tablicy dwuwymiarowej, uzyskana przez ograniczenie indeksów w A[X, Y] tak, że X jest mniejsze niż Y do elementu. W razie potrzeby możemy wykorzystać tylko specjalne części tablic, takie jak górna trójkątna część tablicy dwuwymiarowej, uzyskana przez ograniczenie indeksów w A[X, Y] tak, że X jest mniejsze niż Y

Kolejki i stosy

Ciekawa wariacja na temat wektora/macierzy wynika z obserwacji, że w wielu zastosowaniach wektorów i tablic nie potrzebujemy pełnej mocy zapewnianej przez indeksy. Czasami lista służy jedynie do modelowania kolejki, w którym to przypadku wszystko, czego algorytm potrzebuje w interakcji z listą, to możliwość dodawania elementów z tyłu" i usuwania ich z „przodu". Innym razem lista ma na celu modelowanie stosu, jak te używane w restauracji do przechowywania talerzy. Tutaj algorytm wymaga dodawania i usuwania zdolności tylko na jednym końcu listy, na jej „szczycie". Kolejka jest czasami określana jako lista FIFO (pierwsze weszło-pierwsze wyszło), a stos jako lista LIFO (ostatnie weszło-pierwsze wyszło). Elementy są wpychane na stos, przy czym najwyższy element jest jedynym odsłoniętym do kontroli, a następnie stos można zdjąć, co oznacza, że najwyższy element jest usuwany.

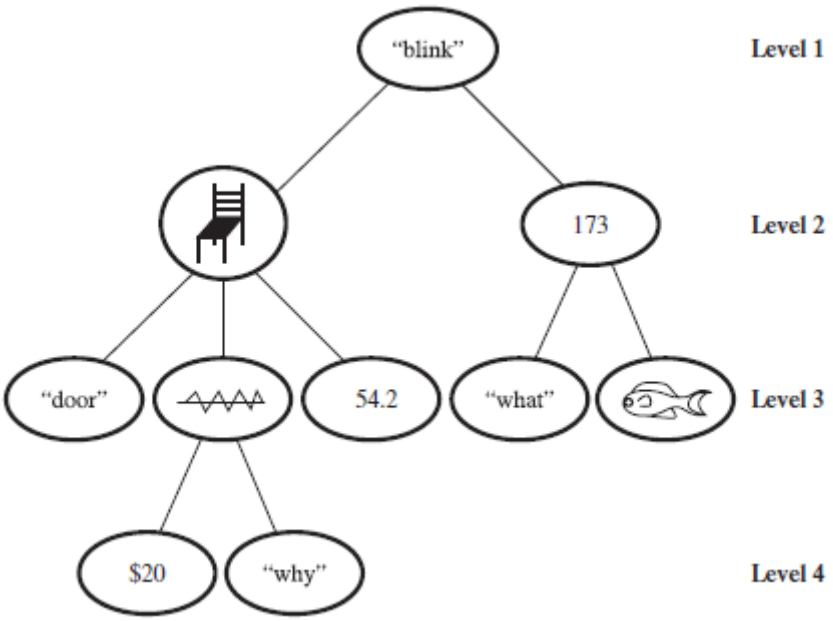


Chodzi o to, że warto, przynajmniej ze względu na przejrzystość algorytmów, myśleć o kolejkach i stosach jako o strukturach danych samych w sobie, a nie o specjalnych rodzajach list. Możemy wtedy użyć specjalnie opracowanych instrukcji elementarnych, takich jak „dodaj X do kolejki A" lub „wsuń X na stos S", zamiast niejasnych sformułowań, które wprost zawierają indeksy.

* Omawiając rekurencję wcześniej, niejawnie natknęliśmy się na potrzebę posiadania wektora, który w rzeczywistości jest używany tylko jako stos. Szczegóły nie zostaną tutaj przedstawione, ale zachęcamy do zastanowienia się nad sposobem, w jaki procesor pamięta, gdzie naprawdę się znajduje, podczas wykonywania algorytmu rekurencyjnego. Zapamiętywanie jego lokalizacji w tekście algorytmu nie stanowi problemu. To, co wymaga prowadzenia księgowości, to zarządzanie listą wywołań rekurencyjnych, które nie zostały jeszcze zakończone, i ustalenie, dokąd wrócić po zakończeniu każdego wywołania. Nietrudno zauważyć – a do zilustrowania tego możemy użyć algorytmu Wież Hanoi – że faktycznie potrzebujemy stosu. Ilekroć jest proszony o ponowne wejście do procedury przez wywołanie rekurencyjne, procesor „wpycha" swój adres zwrotny i bieżące wartości parametrów na stos. (W przykładzie z wieżami do wyboru są dwa takie możliwe adresy, odpowiadające lokalizacjom dwóch instrukcji wywołania w algorytmie.) Po zakończeniu wykonywania procedury rekurencyjnej odczytuje „pchnięte" informacje z góry stosu, przywraca stare wartości do parametrów i powraca do podanego adresu w tekście algorytmu. Jednocześnie „zrzuca" te informacje ze szczytu stosu i odrzuca je. Innym przykładem zastosowania stosów jest przemierzanie labiryntu poprzez wyczerpanie wszystkich możliwości. Takie przemierzanie wymaga utrzymywania listy odwiedzonych już skrzyżowań, dodawania do stosu nowych w miarę ich osiągania oraz usuwania tych, których wszystkie ścieżki zostały przebyte. W ten sposób stos zawsze zawiera ścieżkę od początku do aktualnie odwiedzanego punktu labiryntu.

Drzewa lub hierarchie

Jedną z najważniejszych i najbardziej wyróżniających się istniejących struktur danych jest drzewo. Nie jest to drzewo w botanicznym znaczeniu tego słowa, ale drzewo o bardziej abstrakcyjnej naturze. Wszyscy widzieliśmy takie drzewa używane do opisywania powiązań rodzinnych. Dwa najczęstsze rodzaje drzew genealogicznych to „drzewo przodków", które zaczyna się od osoby i działa wstecz przez rodziców, dziadków itd., oraz „drzewo potomków", które działa naprzód poprzez dzieci, wnuki itd. . Drzewo to zasadniczo hierarchiczny układ danych. Jeden przedmiot znajduje się w specjalnym miejscu zwanym korzeniem, a pozostałe są zorganizowane jako potomkowie korzenia. W informatyce drzewa są zwykle wizualizowane „do góry nogami" – korzeń u góry, a reszta drzewa rozłożona poniżej. Użyta terminologia jest dziwną mieszanką terminów z matematyki, botaniki i życia rodzinnego jednorodzicielskiego. Mówimy o korzeniu, o węzłach drzewa (punktach w drzewie), jego potomstwie (bezpośrednim potomstwie), jego liściach (węzły na „dno" drzewa, bez potomstwa), jego ścieżki lub gałęzie (sekwencje węzłów odpowiadające trawersom w dół w kierunku od korzenia do liścia), a także o rodzicach, przodkach, potomkach i rodzeństwie (dwa węzły są rodzeństwem, jeśli mają tego samego rodzica). Rysunek przedstawia drzewo, celowo skonstruowane bez konkretnego wyjaśnienia jego zawartości lub układu.

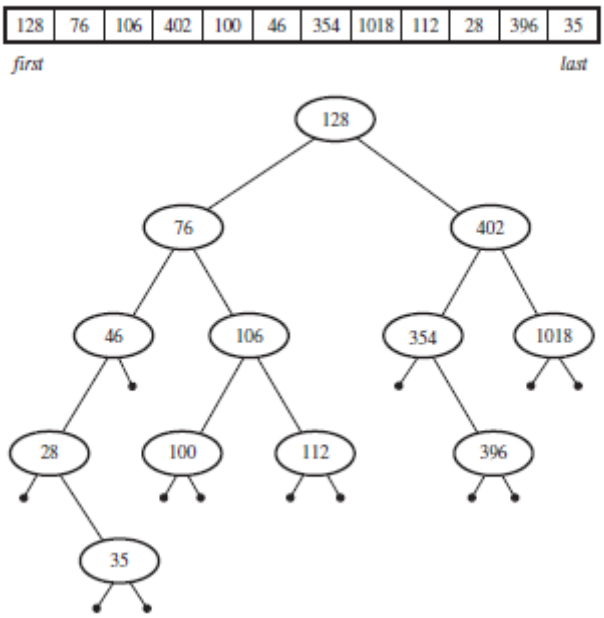


Niemniej jednak słowo „mignięcie" jest jego korzeniem lub, używając innej terminologii, jest to węzeł na poziomie 1 drzewa; Krzesło i liczba 173 to potomstwo korzenia lub, równoważnie, są to węzły na poziomie 2 drzewa i tak dalej. Przykładem drzew znanych użytkownikom komputerów jest hierarchiczna organizacja plików w katalogach, które same mogą znajdować się w innych katalogach. Drzewa można znaleźć gdzie indziej w życiu codziennym. Schematy organizacyjne większości firm to drzewa, podobnie jak schematyczne opisy rozpadu złożonych maszyn. Nawet same algorytmy często można opisać w strukturze drzewa; z grubsza mówiąc, poziomy drzewa odpowiadają poziomom reprezentowanym przez sekwencje liczb, których używamy w tej książce do oznaczania instrukcji. Inny ważny przykład dotyczy drzew łownych. Na przykład korzeń drzewa szachowego zawiera opis konfiguracji szachownicy otwierającej. Potomstwo korzenia reprezentuje 20 możliwych konfiguracji szachownicy wynikających z wykonania przez białe pierwszego ruchu, a potomstwo każdego z nich reprezentuje wyniki wszystkich możliwych odpowiedzi ze strony czarnych i tak dalej. Większość dwuosobowych drzewek do gry, podobnie jak szachy, ma szereg interesujących właściwości. Poziomy o numerach nieparzystych odpowiadają turze pierwszego gracza, podczas gdy te o numerach parzystych odpowiadają turze drugiego gracza. Ścieżki odpowiadają rzeczywistym grom, a każda możliwa gra pojawia się jako ścieżka w drzewie. Wreszcie listki reprezentują zakończenia gry (na przykład w szachach przez mata lub trzykrotne powtórzenie). W dalszej części książki będziemy mieli okazję wrócić do drzew łownych. Drzewa są używane w wielu różnych zastosowaniach i bardziej niż jakakolwiek inna strukturalna metoda łączenia części w całość można powiedzieć, że reprezentują szczególny rodzaj strukturyzacji, który obfituje w algorytmy.

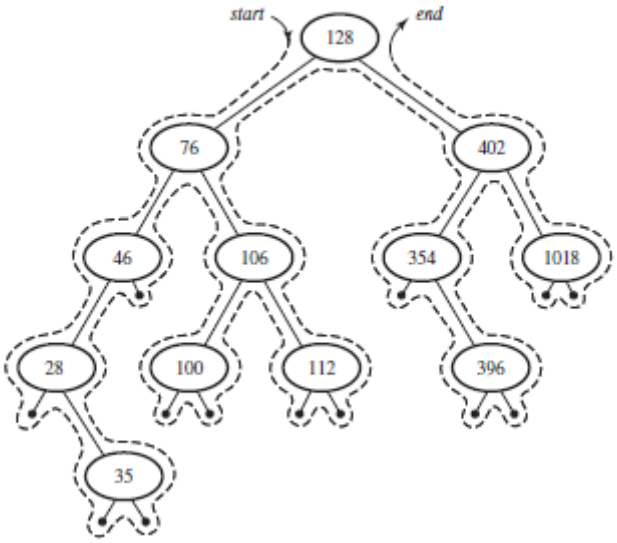
Sortowanie drzew: przykład

Aby dać pewne pojęcie o użyteczności drzew, rozważmy teraz inną procedurę sortowania, opartą na drzewach binarnych. Drzewo binarne to takie, w którym każdy węzeł ma najwyżej dwoje potomków. (Na przykład drzewo genealogiczne przodków jest binarne.) Drzewo binarne można również zdefiniować jako drzewo, którego stopień wyjściowy jest ograniczony przez 2. Zaletą tego ostatniego terminu jest jego ogólność; możemy mówić o drzewach o stopniu wyjściowym 17 lub 938, a nawet o tych z nieskończonym stopniem wyjściowym. Wracając do drzew binarnych, ponieważ stopień wyjściowy w każdym węźle wynosi co najwyżej 2, wygodnie jest rozróżnić te dwa potomstwa, nazywając je odpowiednio lewą i prawą. Treesort, jak będziemy to nazywać, składa się z dwóch

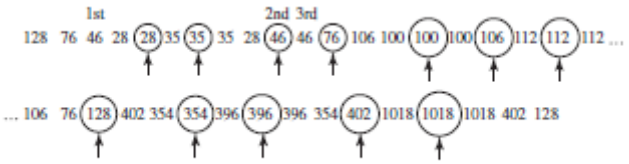
głównych kroków: (1) przekształcenia listy wejściowej w binarne drzewo poszukiwań T ; (2) przejść przez T najpierw w lewo i wyprowadzić każdy element podczas drugiej wizyty. Wyjaśnienie jest tutaj w porządku. Aby posortować listę elementów (powiedzmy liczb), algorytm najpierw organizuje elementy w specjalny rodzaj drzewa binarnego, zwanego drzewem wyszukiwania binarnego. Każdy węzeł tego drzewa ma następującą własność: wszyscy jego „lewi potomkowie" (czyli wszystkie elementy w całym jego lewym poddrzewie, nie tylko jego bezpośrednie potomstwo z lewej strony) mają mniejszą wartość niż wartość węzła samego i wszyscy jego prawi potomkowie są od niego więksi. Rysunek 1 pokazuje przykład takiego drzewa wyszukiwania binarnego.



Drzewo to można skonstruować w następujący sposób: pierwszy element na liście jest traktowany jako korzeń, a następnie każdy element jest rozpatrywany po kolei i dołączany do rosnącego drzewa jako nowy liść, prawdopodobnie „odpuszczając" wcześniej wstawione liście procesu. Znalezienie właściwego miejsca nowego elementu jako liścia w drzewie odbywa się poprzez wielokrotne porównywanie go z otrzymanymi węzłami, obracając się w lewo lub w prawo w zależności od wyniku porównania. Jeśli nowy element jest mniejszy niż element w węźle, na który patrzymy, idziemy w lewo (ponieważ wtedy należy do lewej strony tego węzła); w przeciwnym razie idziemy w prawo. Powinieneś skorzystać z rysunku, aby zapoznać się z tą procedurą, a następnie spróbować zapisać podprogram reprezentujący krok (1). Teraz nadchodzi ciekawa część. Po zbudowaniu binarnego drzewa wyszukiwania, drugi etap sortowania drzew wymaga przechodzenia przez nie w następujący sposób. Procesor zaczyna się od nasady i porusza się w dół, zawsze trzymając się lewej strony. Za każdym razem, gdy próbuje poruszyć się w lewo, ale nie może (na przykład, gdy nie ma potomstwa leworęcznego), porusza się niechętnie w prawo. Wyczerpawszy zarówno lewą, jak i prawą stronę, albo dlatego, że nie znajduje potomstwa, albo dlatego, że już je odwiedził, wycofuje się; to znaczy, że przesuwa się w górę do rodzica bieżącego węzła. Jeśli ten ruch w górę zakończy podróż z węzła macierzystego do jego lewego potomstwa, procesor skręca w prawo i ponownie opada w dół; ale jeśli właśnie wrócił z podróży w prawo, w rzeczywistości wyczerpał obie możliwości węzła rodzica, a więc przechodzi do swojego rodzica. Proces ten trwa do wyczerpania obu kierunków w dół korzenia drzewa, przemierzając w ten sposób całe drzewo. Rysunek 2 pokazuje pierwsze przejście w lewo drzewa z rysunku 1.



Dlaczego dokonujemy tego dziwnego przejścia? Cóż, najpierw zauważ, że jeśli uznamy, że nieobecność potomstwa powoduje pojawienie się krótkiego „kikuta” lub „ślepego zaułka”, który również musi zostać przebyty, to każdy węzeł na drzewie jest „odwiedzany” dokładnie trzy razy. Teraz, jeśli podczas przemierzania drzewa w ten sposób konsekwentnie wypisujemy element danych znaleziony w węźle podczas odwiedzania go dokładnie po raz drugi, lista ostatecznie zostanie wysłana do wyjścia w całości, posortowana w porządku rosnącym! Może to brzmieć jak magia, ale można to łatwo zilustrować, podążając za skręconymi strzałkami przecinającymi się na Rysunku 2, zapisując liczby podczas ich odwiedzania, a następnie zaznaczając drugie pojawienie się każdego z nich na liście wynikowej. Znaki te tworzą posortowaną listę



Oba etapy algorytmu sortowania drzew można dość łatwo opisać jako podprogramy rekurencyjne. Oto procedura dla drugiego etapu, w której używamy „left(T)” do oznaczenia lewego poddrzewa T , i podobnie do „right(T)”. Zgodnie z naszą konwencją dotyczącą obecności pniaków, gdy potomstwo nie istnieje, „left(T),” dla drzewa, które nie ma lewego poddrzewa, będzie specjalnym pustym drzewem; to znaczy drzewo nie zawierające niczego, nawet korzenia podprogramu drugiej-wizyty-przechodzenia-z T :

(1) jeśli T jest puste, to zwróć;

(2) w przeciwnym razie (tj. jeśli T nie jest puste) wykonaj następujące czynności:

(2.1) wywołaj drugą wizytę-traversal-of left(T);

(2.2) wypisz element danych znaleziony w korzeniu T ;

(2.3) zadzwoń do drugiej wizyty-przejścia w prawo(T);

(3) powrót.

Konstruując tę procedurę, wykorzystaliśmy fakt, że drugie odwiedziny węzła zawsze będą miały miejsce tuż po zakończeniu przechodzenia przez całe jego lewe poddrzewo i tuż przed rozpoczęciem przechodzenia jego prawego poddrzewa. Tak więc najpierw wywołujemy podprogram rekurencyjnie w celu ukończenia całego przejścia dla lewego poddrzewa, a kiedy to wywołanie się zakończy (to znaczy, że przejście tego poddrzewa zostało zakończone), wyprowadzamy element w korzeniu, ponieważ jest teraz odwiedzany po raz drugi (za pierwszym razem, gdy przeszliśmy w dół do jego lewego poddrzewa). Następnie przechodzimy rekursywnie w dół w prawo, aby wykonać przejście w prawo. Charakter „lewo-najpierw" tego specjalnego przejścia staje się dość oczywisty ze struktury sformułowania rekurencyjnego, ponieważ dla każdego poddrzewa (to znaczy dla każdego wywołania rekurencyjnego) można łatwo zauważyć, że procedura przechodzi najpierw w lewo, a następnie prawo. Nawiasem mówiąc, interesujące jest zobaczenie, co dzieje się z listą wyników, kiedy to odwracamy, idąc konsekwentnie najpierw w prawo, a potem w lewo. (Co się dzieje i jak odpowiednio zmieniamy podprogram?) Struktura tej procedury przypomina procedurę ruchu dla Wież Hanoi. W rzeczywistości, jeśli zastosujemy tę procedurę do przykładowego drzewa, a następnie narysujemy obraz trudów procesora, obraz będzie dokładnie odzwierciedlał kształt przykładowego drzewa. Możemy zatem stwierdzić, że jeśli wektory i tablice jako struktury danych odpowiadają pętlom, a zagnieżdżone pętle jako strukturom kontrolnym, to drzewa odpowiadają rekurencyjnym podprogramom. Drzewo jest z natury rekurencyjnym obiektem, składającym się z niczego (tj. z pustego drzewa, jak wyjaśniono wcześniej) lub z korzenia dołączonego (rekursywnie) do pewnej liczby drzew. To wyjaśnia, dlaczego przechodzenie przez drzewa, takie jak to właśnie omówione, jest stosunkowo łatwe do opisania rekurencyjnie. Istnieje kilka interesujących sposobów na wprowadzenie elastyczności do istniejących struktur danych. Dobrym przykładem jest koncepcja samoregulacji. Rozumiemy przez to, że za każdym razem, gdy element jest wstawiany lub usuwany ze struktury danych, struktura wykonuje procedurę dostosowania, która wprowadza proste zmiany, mające na celu zachowanie pewnych „ładnych" właściwości. Na przykład w wielu zastosowaniach drzew możliwe jest, że ze względu na pewną sekwencję wstawień/delecji drzewo stanie się cienkie i długie, podczas gdy ze względu na wydajność może być pożądane, aby było szerokie i krótkie. Możliwe jest - choć nie będziemy wchodzić w szczegóły - dokonywanie niewielkich lokalnych zmian w drzewie za każdym razem, gdy jest na nim wykonywana operacja, co zagwarantuje właściwość szerokości i zwięzłości.

Bazy danych i bazy wiedzy

Dla wielu aplikacji komputerowych struktury danych nie wystarczają. Nie zawsze chodzi tylko o rozważenie problemu algorytmicznego i znalezienie lub zdefiniowanie ładnych i użytecznych struktur danych do jego rozwiązania. Czasami istnieje potrzeba bardzo dużej „puli" danych, którą wiele algorytmów traktuje jako część swoich danych wejściowych, a zatem musi mieć stałą strukturę i być łatwo dostępne do wyszukiwania i manipulacji. Przykłady obejmują dane finansowe i osobowe firmy, rezerwacje i informacje o lotach linii lotniczej oraz dane indeksowe biblioteki. Takie masy danych nazywamy bazami danych. Zazwyczaj bazy danych są bardzo duże i zawierają wiele różnych rodzajów danych, od nazw i adresów po niejasne kody i symbole, a w niektórych przypadkach nawet dowolny tekst. Są one zwykle poddawane licznym rodzajom procedur wstawiania, usuwania i pobierania, wykorzystywanych do różnych celów i przez różnych ludzi. O ile dodanie nowego studenta do uniwersyteckiej bazy danych lub usunięcie starej to stosunkowo łatwe zadanie, to w przypadku przeszukiwania bazy danych w celu uzyskania z niej informacji zawiłość wydaje się nie mieć granic. Tylko dla perspektywy, oto kilka zapytań, które można skierować do bazy danych systemu rezerwacji lotów:

- wymień wszystkich pasażerów potwierdzonych lotem 123;
- wymienić wszystkie miejsca w locie 123 zajęte przez pasażerów bez bagażu rejestrowanego;

- znaleźć liczbę pasażerów, którzy zarezerwowali miejsca na dwa identycznie ponumerowane loty zaplanowane na kolejne dni marca tego roku;
- podać nazwiska i numery miejsc wszystkich pasażerów pierwszej klasy na jutrzejszych lotach do Paryża, którzy zamówili wegetariańskie posiłki i mają międzykontynentalne loty kontynuacyjne, na których usługi w klasie ekonomicznej nie zapewniają specjalnych posiłków i których międzylądowanie znajduje się w Zurychu lub Rzym.

Te przykłady ilustrują znaczenie „dobrej" organizacji bazy danych. Jeśli loty kontynuacyjne znajdują się w bazie danych w jakimś niejasnym miejscu i jeśli nie ma łatwego sposobu na zebranie informacji o trasie danego pasażera, to napisanie algorytmu, który rozwiąże ostatni wymieniony problem z odzyskaniem, stanie się naprawdę trudnym zadaniem. Podobnie jak w przypadku struktur danych, dobry projekt bazy danych jest ważny nie tylko ze względu na przejrzystość i łatwość pisania; może to mieć ogromne znaczenie, jeśli chodzi o kwestie wydajności i wykonalności zbudowania systemu bazy danych, który będzie w stanie odpowiedzieć na takie zapytania w rozsądnym czasie. Zaproponowano różne ogólne modele organizacji baz danych, które są wykorzystywane w rzeczywistych bazach danych. Modele te są przeznaczone do przechowywania dużych ilości danych przy jednoczesnym wiernym i wydajnym uchwyceniu relacji między elementami danych. Jeden z najpopularniejszych modeli baz danych, model relacyjny, umożliwia rozmieszczenie danych w dużych tabelach, przypominających strukturę danych tablicowych. Inni domagają się pewnych rodzajów układów podobnych do drzewa lub sieci, takich jak model hierarchiczny, który organizuje dane w wielowarstwowej formie przypominającej drzewo. Wydaje się, że nie ma szerokiego konsensusu co do preferowanych modeli dla poszczególnych zastosowań oraz wielu metod i języków zostały opracowane do manipulowania i odpytywania baz danych skonstruowanych w każdym z tych modeli. Istnieje jednak znaczący ruch w kierunku modelu relacyjnego, a nowsze bazy danych obiektowych czekają poza sceną na swoją kolej. Bazy danych są często wykorzystywane do obsługi systemów operacyjnych, które śledzą zmiany w niezliczonych szczegółach niezbędnych do funkcjonowania dużej organizacji. W związku z tym muszą wspierać skuteczne modyfikacje danych, a także ich zapytania, ale mniej dbają o przechowywanie informacji, które mają jedynie wartość historyczną. Jednak duże ilości danych przechowywane w bazach danych mogą być również wykorzystywane do celów analitycznych, takich jak odkrywanie trendów czy usprawnianie procesów. Na przykład bank może chcieć odkryć korelacje między niespłacaniem kredytów a innymi właściwościami posiadacza rachunku w celu opracowania lepszych narzędzi prognostycznych. Można to zrobić, stosując metody analizy statystycznej do dużej ilości starych danych dostępnych we własnych bazach danych banku. Nauka o odkrywaniu takich użytecznych bryłek informacji z ogromnych źródeł danych nazywana jest eksploracją danych i zajmuje się głównie niezmiennymi danymi historycznymi. Zazwyczaj ilość danych historycznych jest znacznie większa niż potrzebna do celów operacyjnych; często setki razy większe. W ostatnich latach biologia oferuje szczególnie kuszące rodzaje baz danych, wynikające z projektu genomu i jego produktów ubocznych, które zaczynają wymagać opracowania nowych, potężnych technik eksploracji danych. Nowy rodzaj bazy danych, zwany hurtownią danych, został opracowany w celu obsługi ogromnych ilości danych, które zmieniają się powoli lub w których w ogóle zmieniają się tylko bardzo małe porcje. Niektóre hurtownie danych wydają się użytkownikom podobne do innych baz danych; jednak organizacja wewnętrzna może używać zupełnie innego zestawu algorytmów, aby skutecznie osiągać swoje cele. Pewne rodzaje danych są lepiej postrzegane jako fragmenty wiedzy, a nie tylko liczby, nazwy czy kody. Oprócz dużej bazy danych opisującej inwentarz firmy produkcyjnej, możemy chcieć mieć dużą bazę wiedzy zawierającą informacje istotne dla prowadzenia tej firmy. Jego elementy wiedzy mogą w jakiś sposób zakodować informacje, takie jak „Zmiany wynagrodzeń to kwestie personalne", „Pan Smith jest lepszym menedżerem niż pani Brown, jeśli chodzi o problemy kadrowe, ale nie w kwestiach technicznych" lub „Jeśli cena ropy idzie w górę, w następnym miesiącu będziemy musieli

obniżyć wszystkie pensje". W przeciwieństwie do bazy danych, która przechowuje dane do późniejszego pobrania, baza wiedzy wykorzystuje swoją wiedzę w bardziej wyrafinowany sposób. Na przykład, z powyższych reguł i faktu, że ceny ropy wzrosły, moglibyśmy chcieć wnioskować, że pan Smith powinien zająć się zmianami płac. Jest oczywiste, że takie możliwości wnioskowania wymagają bardziej złożonej organizacji niż elementy danych o mniej lub bardziej stałym formacie, zwłaszcza jeśli zależy nam na wydajnym wyszukiwaniu. Bazy wiedzy stają się zatem naturalnym kolejnym krokiem po bazach danych i stanowią bogate źródło interesujących pytań dotyczących reprezentacji, organizacji i wyszukiwania algorytmicznego.

Ćwiczenia

2.1. Przedstawiony w tekście algorytm sumowania wynagrodzeń N pracowników wykonuje pętlę polegającą na dodaniu jednej pensji do sumy i przesunięciu wskaźnika na liście pracowników N -1 razy. Ostatnie wynagrodzenie doliczane jest osobno. Jaki jest tego powód? Dlaczego nie wykonamy pętli N razy?

2.2. Rozważmy algorytm bubblesort przedstawiony w tekście.

(a) Wyjaśnij, dlaczego pętla zewnętrzna jest wykonywana tylko N - 1 razy.

(b) Popraw algorytm tak, aby przy każdym powtórzonym wykonaniu pętli zewnętrznej pętla wewnętrzna sprawdzała o jeden element mniej.

2.3. Przygotuj schematy blokowe dla algorytmu sortowania bąbelkowego przedstawionego w tekście oraz dla ulepszonej wersji, którą poproszono o zaprojektowanie w ćwiczeniu 2.2.

2.4. Napisz algorytmy, które przy danej liczbie całkowitej N i liście L składającej się z N liczb całkowitych dają w S i P sumę liczb parzystych występujących odpowiednio w L i iloczyn liczb nieparzystych.

(a) Korzystanie z iteracji ograniczonej.

(b) Używanie stwierdzeń „goto".

2.5. Pokaż, jak wykonać następujące symulacje niektórych konstrukcji sterowania przez innych. Konstrukcja sekwencjonowania „a następnie" jest domyślnie dostępna dla wszystkich symulacji. W razie potrzeby możesz wprowadzić i używać nowych zmiennych i etykiet.

(a) Symuluj pętlę „for-do" za pomocą pętli „if-then-else" za pomocą pętli „while d0".

(c) Symuluj pętlę „while-do" za pomocą instrukcji „if-then" i „goto".

(d) Symuluj pętlę „while do" za pomocą pętli „repeat until" i instrukcji „if then".

2.6. Zapisz sekwencję ruchów rozwiązujących problem Wież Hanoi dla pięciu pierścieni. Silnia nieujemnej liczby całkowitej N jest iloczynem wszystkich dodatnich liczb całkowitych mniejszych lub równych N. Bardziej formalnie, wyrażenie N silnia, oznaczane przez N!, jest rekurencyjnie definiowane przez 0! = 1 i (N + 1)! = N! × (N + 1). Na przykład 1! = 1 i 4! = 3! ×4 = . . . = 1 × 2 × 3 × 4 = 24.

2.7. Napisz algorytmy, które obliczają N!, mając nieujemną liczbę całkowitą N.

(a) Korzystanie z instrukcji iteracyjnych.

(b) Korzystanie z rekurencji.

2.8. Pokaż, jak symulować pętlę „podczas wykonywania” za pomocą instrukcji warunkowych i procedury rekurencyjnej. Dla dodatniej liczby całkowitej N oznaczmy przez AN zbiór liczb całkowitych od 1 do N. Permutacja zbioru AN jest uporządkowaną sekwencją $(a_1, a_2,...,a_N)$, w której każda liczba całkowita ze zbioru A_N pojawia się dokładnie raz. Na przykład (2, 3, 1) i (1, 2, 3) to dwie różne permutacje zbioru A_3.

2.9. Udowodnij, że liczba permutacji A_N wynosi N!.

2.10. Permutację $(a_1,...,a_N)$ można przedstawić za pomocą wektora P o długości N z P[I] = aI. Zaprojektuj algorytm, który przy danej liczbie całkowitej N i wektorze liczb całkowitych P o długości N sprawdza, czy P reprezentuje jakąkolwiek permutację A_N .

2.11. Zaprojektuj algorytm, który przy dodatniej liczbie całkowitej N wytwarza wszystkie permutacje A_N . Mówimy, że permutację $\sigma = (a_1, ... , a_N)$ można uzyskać ze stosu, jeśli można rozpocząć od ciągu wejściowego (1, 2, ... , N) i pustego stosu S, i wytworzyć wynik σ stosując tylko następujące typy operacji:

read(X): Wczytaj liczbę całkowitą z wejścia do zmiennej X.

print(X): Wypisuje na wyjściu liczbę całkowitą aktualnie zapisaną w zmiennej X.

push(X, S): Włóż liczbę całkowitą aktualnie zapisaną w zmiennej X na stos S.

pop(X, S): Przenieś liczbę całkowitą ze szczytu stosu S do zmiennej X. (Ta operacja jest niedozwolona, jeśli S jest puste).

Na przykład permutację (2, 1) można uzyskać ze stosu, ponieważ następujące serie operacji

read(X), push(X, S), read(X), print(X), pop(X, S), print(X)

zastosowane do sekwencji wejściowej (1, 2) tworzy sekwencję wyjściową (2, 1). Permutację można uzyskać przez kolejkę, jeśli można ją w podobny sposób uzyskać z danych wejściowych (1, 2, ... , N), używając początkowo pustej kolejki Q i operacji read(X), print(X), i

add(X, Q): Dodaj liczbę całkowitą aktualnie przechowywaną w X na tył Q.

remove(X, Q): Usuń liczbę całkowitą z początku Q do X. (Ta operacja jest nieprawidłowa, jeśli Q jest puste.)

Podobnie możemy mówić o permutacji uzyskanej przez dwa stosy, jeśli pozwolimy na push i pop operacje na dwóch stosach S i S’ .

2.12.

(a) Pokaż, że stos można uzyskać następujące permutacje:

i. (3, 2, 1).

ii. (3, 4, 2, 1).

iii. (3, 5, 7, 6, 8, 4, 9, 2, 10, 1).

(b) Udowodnij, że stos nie może uzyskać następujących permutacji:

i. (3, 1, 2).

ii. (4, 5, 3, 7, 2, 1, 6).

(c) Ile permutacji A_4 nie może uzyskać stos?

2.13. Zaprojektuj algorytm sprawdzający, czy daną permutację można uzyskać za pomocą stosu. W przypadku odpowiedzi twierdzącej algorytm powinien również wypisać odpowiednią serię operacji. W swoim algorytmie, oprócz czytania, drukowania, wypychania i wyskakiwania, możesz użyć testu isempty(S) do testowania pustego stosu S.

2.14.

(a) Podaj serię operacji, które pokazują, że każda z permutacji podanych w ćwiczeniu 2.12(b) może być uzyskana przez kolejkę, a także przez dwa stosy.

(b) Udowodnij, że każdą permutację można uzyskać w kolejce.

(c) Udowodnij, że każdą permutację można uzyskać przez dwa stosy.

2.15. Rozszerz algorytm, który miałeś zaprojektować w ćwiczeniu 2.13, tak aby jeśli danej permutacji nie można uzyskać na stosie, algorytm wypisze serię operacji na dwóch stosach, które ją wygenerują.

2.16. Rozważmy algorytm sortowania drzewa opisany w tekście.

(a) Skonstruuj algorytm, który przekształca daną listę liczb całkowitych w drzewo wyszukiwania binarnego.

(b) Jak wyglądałby wynik funkcji sortowania drzewa, gdybyśmy odwrócili kolejność, w której podprocedura drugiej wizyty-przemierzania wywołuje się rekurencyjnie? Innymi słowy, konsekwentnie odwiedzamy prawe potomstwo węzła, zanim odwiedzimy lewe.