

# Spis treści

---

Przedmowa .....	9
<b>1. Podstawowe zasady analizy algorytmów .....</b>	<b>13</b>
1.1. Złożoność obliczeniowa .....	13
1.2. Równania rekurencyjne .....	20
1.3. Funkcje tworzące .....	21
1.4. Poprawność semantyczna .....	22
1.5. Podstawowe struktury danych .....	24
1.5.1. Lista .....	25
1.5.2. Zbiór .....	27
1.5.3. Graf .....	28
1.5.4. Notacja funkcyjna dla atrybutów obiektów .....	33
1.5.5. Drzewo .....	33
1.6. Eliminacja rekursji .....	36
1.7. Koszt zamortyzowany operacji w strukturze danych .....	38
1.8. Metody układania algorytmów .....	40
1.8.1. Metoda „dziel i zwyciężaj” .....	40
1.8.2. Programowanie dynamiczne .....	40
1.8.3. Metoda zachlanna .....	41
1.8.4. Inne metody .....	42
Zadania .....	42
<b>2. Sortowanie .....</b>	<b>49</b>
2.1. Selectionsort – sortowanie przez selekcję .....	50
2.2. Insertionsort – sortowanie przez wstawianie .....	51
2.3. Quicksort – sortowanie szybkie .....	53

2.4. Dolne ograniczenie na złożoność problemu sortowania .....	62
2.5. Sortowanie pozycyjne .....	66
2.6. Kolejki priorytetowe i algorytm heapsort .....	70
2.7. Drzewa turniejowe i zadania selekcji .....	78
2.8. Szybkie algorytmy wyznaczania $k$ -tego największego elementu w ciągu ..	83
2.9. Scalanie ciągów uporządkowanych .....	86
2.10. Sortowanie zewnętrzne .....	89
2.10.1. Scalanie wielofazowe z 4 plikami .....	90
2.10.2. Scalanie wielofazowe z 3 plikami .....	91
Zadania .....	94
<b>3. Słowniki .....</b>	<b>99</b>
3.1. Implementacja listowa nieuporządkowana .....	100
3.2. Implementacja listowa uporządkowana .....	100
3.3. Drzewa poszukiwań binarnych .....	105
3.3.1. Drzewa AVL .....	113
3.3.2. Samoorganizujące się drzewa BST .....	117
3.4. Mieszanie .....	120
3.4.1. Wybór funkcji mieszającej .....	121
3.4.2. Struktury danych stosowane do rozwiązywania problemu kolizji ..	122
3.5. Wyszukiwanie pozycyjne .....	126
3.5.1. Drzewa RST .....	127
3.5.2. Drzewa TRIE .....	130
3.5.3. Drzewa PATRICIA .....	131
3.6. Wyszukiwanie zewnętrzne .....	134
3.6.1. Pliki nieuporządkowane .....	135
3.6.2. Pliki z funkcją mieszającą .....	135
3.6.3. Sekwencyjne pliki indeksowane .....	135
3.6.4. B-drzewo jako wielopoziomowy indeks rzadki .....	136
3.6.5. B-drzewo jako wielopoziomowy indeks gęsty .....	138
Zadania .....	138
<b>4. Złożone struktury danych dla zbiorów elementów .....</b>	<b>143</b>
4.1. Problem sumowania rozłącznych zbiorów .....	143
4.1.1. Implementacja listowa .....	144
4.1.2. Implementacja drzewowa .....	148
4.2. Złączalne kolejki priorytetowe .....	155
Zadania .....	162
<b>5. Algorytmy tekstowe .....</b>	<b>164</b>
5.1. Problem wyszukiwania wzorca .....	165
5.1.1. Algorytm N („naiwny”) .....	165

5.1.2. Algorytm KMP (Knutha-Morrisa-Pratta) .....	166
5.1.3. Algorytm liniowy dla problemu wyszukiwania wzorca dwuwymiarowego, czyli algorytm Bakera .....	169
5.1.4. Algorytm GS' (wersja algorytmu Galila-Seifera dla pewnej klasy wzorców) .....	171
5.1.5. Algorytm KMR (Karpa-Millera-Rosenberga) .....	172
5.1.6. Algorytm KR (Karpa-Rabina) .....	174
5.1.7. Algorytm BM (Boyer-Moore'a) .....	175
5.1.8. Algorytm FP (Fishera-Patersona) .....	178
5.2. Drzewa sufiksowe i grafy podłów .....	180
5.2.1. Niezwarta reprezentacja drzewa sufiksowego .....	180
5.2.2. Tworzenie drzewa sufiksowego .....	182
5.2.3. Tworzenie grafu podłów .....	185
5.3. Inne algorytmy tekstowe .....	189
5.3.1. Obliczanie najdłuższego wspólnego podslowa .....	189
5.3.2. Obliczanie najdłuższego wspólnego podciagu .....	190
5.3.3. Wyszukiwanie słów podwójnych .....	190
5.3.4. Wyszukiwanie słów symetrycznych .....	193
5.3.5. Równoważność cykliczna .....	194
5.3.6. Algorytm Huffmmana .....	195
5.3.7. Obliczanie leksykograficznie maksymalnego sufiksu .....	196
5.3.8. Jednoznaczne kodowanie .....	199
5.3.9. Liczenie liczby podłów .....	200
Zadania .....	200
<b>6. Algorytmy równolegle .....</b>	<b>205</b>
6.1. Równolegle obliczanie wyrażeń i prostych programów sekwencyjnych .....	207
6.2. Sortowanie równolegle .....	221
Zadania .....	224
<b>7. Algorytmy grafowe .....</b>	<b>227</b>
7.1. Spójne składowe .....	229
7.2. Dwuspójne składowe .....	232
7.3. Silnie spójne składowe i silna orientacja .....	239
7.4. Cykle Eulera .....	245
7.5. 5-kolorowanie grafów planarnych .....	249
7.6. Najkrótsze ścieżki i minimalne drzewo rozpinające .....	254
Zadania .....	256
<b>8. Algorytmy geometryczne .....</b>	<b>258</b>
8.1. Elementarne algorytmy geometryczne .....	259

8.2. Problem przynależności .....	260
8.3. Wypukła otoczka .....	263
8.4. Metoda zamiatania .....	271
8.4.1. Najmniej odległa para punktów .....	272
8.4.2. Pary przecinających się odcinków .....	275
Zadania .....	282
<b>Bibliografia .....</b>	<b>284</b>
<b>Skorowidz .....</b>	<b>286</b>

## Przedmowa

Niniejsza książka jest przeznaczona dla czytelników interesujących się głębiej informatyką, w tym przede wszystkim dla studentów informatyki. W szczególności może służyć jako podręcznik do wykładów: „Algorytmy i struktury danych” i „Analiza algorytmów” dla studentów studiów informatycznych. Jej fragmenty mogą być także wykorzystane w nauczaniu przedmiotu „Metody programowania” i „Kombinatoryka i teoria grafów” w ujęciu algorytmicznym. Sądzimy, że książka może też zainteresować szersze kręgi czytelników, gdyż daje elementarne wprowadzenie do nowoczesnych metod tworzenia i analizy algorytmów. Metody te oraz bogactwo różnorodnych struktur danych, przedstawionych w książce, mogą być pomocne w projektowaniu efektywnych algorytmów dla problemów pojawiających się w praktyce programistycznej lub pracy badawczej.

Zakładamy, że czytelnik ma pewne podstawowe przygotowanie z kombinatoryki i rachunku prawdopodobieństwa (na poziomie szkoły średniej) i że umie układać algorytmy w Pascalu. Znajomość przedmiotów: „Wstęp do informatyki”, „Metody programowania” i „Analiza matematyczna I” jest pożądana przy czytaniu tej książki, ale niekonieczna.

Książka powstała z notatek do wykładów: „Algorytmy i struktury danych” oraz „Analiza algorytmów”, prowadzonych przez nas dla studentów informatyki Uniwersytetu Warszawskiego w latach 1986-1994. Pierwsza jej wersja ukazała się w postaci skryptu Uniwersytetu Warszawskiego [BDR].

Niniejsza książka składa się z 8 rozdziałów. Rozdział 1 stanowi wprowadzenie do dziedziny analizy algorytmów. Zdefiniowaliśmy w nim takie pojęcia, jak złożoność obliczeniowa i poprawność semantyczna algorytmu. Omówiliśmy rozwiązywanie równań rekurencyjnych i podstawowe struktury danych: listy, zbiory, grafy, drzewa i ich realizacje. Przedstawiliśmy także elementarne metody algorytmicznego rozwiązywania problemów.

Rozdział 2 zawiera omówienie głównych algorytmów sortowania wraz z ich analizą i zastosowaniami wprowadzonych struktur danych do rozwiązywania pokrewnych problemów. Rozdział 3 jest poświęcony zadaniu wyszukiwania elementów w zbiorze. Przedstawiliśmy w nim podstawowe struktury danych i częściową ich analizę. W rozdziale 4 omówiliśmy i zanalizowaliśmy dwie złożone struktury danych, umożliwiające szybkie wykonywanie operacji na zbiorach rozłącznych. Opisaliśmy rozwiązywanie problemu sumowania zbiorów rozłącznych i przedstawiliśmy implementację złączalnych kolejek priorytetowych za pomocą drzew dwumianowych. Rozdziały 5, 6, 7, 8 są poświęcone dziedzinom informatyki teoretycznej, w której badania nad algorytmami rozwijały się w ostatnich latach najszybciej. Przedstawiliśmy w nich algorytmy tekstowe, a także algorytmy równoległe, grafowe i geometryczne. Każdy rozdział jest zakończony zestawem zadań umożliwiających pogłębienie zdobywanej wiedzy. Celem naszym nie było dostarczenie programów gotowych do uruchomienia. Pełna implementacja niektórych z zaprezentowanych algorytmów wymaga pewnego wysiłku programistycznego. Zachęcamy Cię do podjęcia go, ponieważ dopiero pełna, poprawna implementacja świadczy o dobrym zrozumieniu przerabianego materiału.

Większość przedstawionych tu algorytmów i struktur danych uważa się już dziś za klasyczne. Można je znaleźć (rozproszone) w licznych książkach poświęconych algorytmom (niestety w większości w języku angielskim). Czytelnikowi zainteresowanemu innym spojrzeniem na omawianą tematykę oraz poszerzaniem wiedzy dotyczącej problemów poruszanych w tym opracowaniu polecamy podane niżej pozycje.

**A. V. Aho, J. E. Hopcroft, J. D. Ullman: Projektowanie i analiza algorytmów komputerowych [AHU]**

Jest to książka poświęcona algorytmom i strukturom danych z wielu różnych dziedzin informatyki teoretycznej.

**L. Banachowski, A. Kreczmar: Elementy analizy algorytmów [BK]**

W książce tej przedstawiono podstawowe zasady analizy algorytmów, zilustrowane ciekawymi przykładami.

**L. Banachowski, A. Kreczmar, W. Rytter: Analiza algorytmów i struktur danych [BKR]**

W książce tej omówiono złożone metody projektowania i analizowania algorytmów oraz struktur danych.

**T. H. Cormen, C. E. Leiserson, R. L. Rivest: Wprowadzenie do algorytmów [CLR]**

W książce tej przedstawiono najważniejsze (od elementarnych do złożonych) algorytmy i struktury danych z wielu różnych dziedzin informatyki.

**M. Crochemore, W. Rytter: Text algorithms [CR]**

Jest to książka poświęcona algorytmom na tekstach.

**A. Gibbons, W. Rytter: Efficient parallel algorithms [GR]**

W książce tej omówiono algorytmy równolegle z wielu różnych dziedzin informatyki teoretycznej.

**J. Jaja: An introduction to parallel algorithms [J]**

Jest to wprowadzenie do problematyki algorytmów i obliczeń równoległych.

**D. E. Knuth: The art of computer programming. Sorting and searching [K]**

W książce tej omówiono klasyczne metody sortowania i wyszukiwania.

**W. Lipski: Kombinatoryka dla programistów [L]**

Jest to książka poświęcona podstawowym algorytmom grafowym.

**K. Mehlhorn: Multi-dimensional searching and computational geometry [M]**

Tematem książki są problemy i algorytmy geometryczne.

**F. P. Preparata, M. I. Shamos: Computational geometry (an introduction) [PS]**

Jest to doskonale wprowadzenie do problematyki geometrii obliczeniowej.

**R. Sedgewick: Algorithms [S]**

W książce tej bardzo przystępnie omówiono różne algorytmy i struktury danych.

Na przedstawionej tu liście nie ma oczywiście wszystkich pozycji poświęconych algorytmom i strukturom danych. Wymieniliśmy tylko te, które wykorzystałyśmy do przygotowania notatek do wykładów, a następnie tej książki. Opis większości omawianych tu problemów, algorytmów i struktur danych można znaleźć w książkach z tej listy. W razie odstępstwa od powyższej zasady podajemy bezpośrednie źródło omawianego tematu.

Warszawa 1996 r.

*LBanachowski  
Lwówek Rytter  
Kunyelof Niles*

# 1 Podstawowe zasady analizy algorytmów

W tym rozdziale przedstawiamy podstawowe pojęcia stosowane przy badaniu algorytmów i struktur danych. Przede wszystkim wyjaśniamy, na czym polega analiza algorytmu w dwóch głównych aspektach: poprawności semantycznej i złożoności obliczeniowej. Omawiamy elementarne struktury danych definiowane abstrakcyjnie (jako listy, zbiory, grafy, drzewa itd.), z możliwymi różnymi konkretnymi implementacjami (reprezentacjami). Na końcu rozdziału przedstawiamy podstawowe metody konstruowania efektywnych algorytmów (metoda „dziel i zwyciężaj”, programowanie dynamiczne, metoda zachłanna, metoda kolejnych transformacji).

**Analiza algorytmów** to dział informatyki zajmujący się szukaniem najlepszych algorytmów dla zadań komputerowych. Analiza algorytmów polega między innymi na znalezieniu odpowiedzi na podane tu pytania.

1. Czy dany problem może być rozwiązywany na komputerze w dostępnym czasie i pamięci?
2. Który ze znanych algorytmów należy zastosować w danych okolicznościach?
3. Czy istnieje lepszy algorytm od rozważanego? A może jest on optymalny?
4. Jak uzasadnić, że stosując dany algorytm, rozwiąże się zamierzone zadanie?

Dokonując analizy algorytmu, zwracamy uwagę na jego poprawność semantyczną, prostotę, czas działania, ilość zajmowanej pamięci, optymalność oraz okoliczności, w jakich należy go używać, a w jakich nie.

## 1.1.

### Złożoność obliczeniowa

**Złożoność obliczeniową algorytmu** definiuje się jako ilość zasobów komputerowych, potrzebnych do jego wykonania. Podstawowymi zasobami rozważanymi w analizie algorytmów są czas działania i ilość zajmowanej pamięci.

Zauważmy, że nie jest na ogół możliwe wyznaczenie złożoności obliczeniowej jako funkcji danych wejściowych (takich jak ciągi, tablice, drzewa czy grafy). Zwykle, co naturalne, z zestawem danych wejściowych jest związany jego **rozmiar**, rozumiany – mówiąc ogólnie – jako liczba pojedynczych danych wchodzących w jego skład.

W problemie sortowania na przykład za rozmiar przyjmuje się zazwyczaj liczbę elementów w ciągu wejściowym, w problemie przejścia drzewa binarnego – liczbę węzłów w drzewie, a w problemie wyznaczenia wartości wielomianu – stopień wielomianu. Rozmiar zestawu danych  $d$  będziemy oznaczać przez  $|d|$ .

Aby móc wyznaczać złożoność obliczeniową algorytmu, musimy się jeszcze umówić, w jakich jednostkach będziemy ją liczyć. Na złożoność obliczeniową składa się złożoność pamięciowa i złożoność czasowa. W wypadku **złożoności pamięciowej** za jednostkę przyjmuje się zwykle słowo pamięci maszyny. Sytuacja jest nieco bardziej skomplikowana w wypadku **złożoności czasowej**. Złożoność czasowa powinna być własnością samego tylko algorytmu jako metody rozwiązywania problemu – niezależnie od komputera, języka programowania czy sposobu jego zakodowania. W tym celu wyróżnia się w algorytmie charakterystyczne dla niego operacje, nazywane **operacjami dominującymi** – takie, że łączna ich liczba jest proporcjonalna do liczby wykonania wszystkich operacji jednostkowych w dowolnej komputerowej realizacji algorytmu.

Dla algorytmów sortowania na przykład za operację dominującą przyjmuje się zwykle porównanie dwóch elementów w ciągu wejściowym, a czasem też przedstawienie elementów w ciągu; dla algorytmów przeglądania drzewa binarnego przyjmuje się przejście dowiązania między węzłami w drzewie, a dla algorytmów wyznaczania wartości wielomianu – operacje arytmetyczne  $+$ ,  $-$ ,  $*$  i  $/$ .

Za jednostkę złożoności czasowej przyjmuje się wykonanie jednej operacji dominującej.

Złożoność obliczeniową algorytmu traktuje się jako funkcję rozmiaru danych  $n$ . Wyróżnia się: **złożoność pesymistyczną** – definiowaną jako ilość zasobów komputerowych, potrzebnych przy „najgorszych” danych wejściowych rozmiaru  $n$ , oraz **złożoność oczekiwanej** – definiowaną jako ilość zasobów komputerowych, potrzebnych przy „typowych” danych wejściowych rozmiaru  $n$ .

Aby zdefiniować precyzyjnie pojęcia pesymistycznej i oczekiwanej złożoności czasowej, wprowadzimy następujące oznaczenia:

$D_n$  – zbiór zestawów danych wejściowych rozmiaru  $n$ ;

$t(d)$  – liczba operacji dominujących dla zestawu danych wejściowych  $d$ ;

$X_n$  – zmienna losowa, której wartością jest  $t(d)$  dla  $d \in D_n$ ;

$p_{nk}$  – rozkład prawdopodobieństwa zmiennej losowej  $X_n$ , tzn. prawdopodobieństwo, że dla danych rozmiaru  $n$  algorytm wykona  $k$  operacji dominujących ( $k \geq 0$ ).

Rozkład prawdopodobieństwa zmiennej losowej  $X_n$  wyznacza się na podstawie informacji o zastosowaniach rozważanego algorytmu. Gdy na przykład zbiór  $D_n$  jest skoń-

czony, przyjmuje się często model probabilistyczny, w którym każdy zestaw danych rozmiaru  $n$  może się pojawić na wejściu do algorytmu z jednakowym prawdopodobieństwem.

Przez **pesymistyczną złożoność czasową** algorytmu rozumie się funkcję

$$W(n) = \sup\{t(d): d \in D_n\},$$

gdzie  $\sup$  oznacza kres góry zbioru.

Przez **oczekiwana złożoność czasową** algorytmu rozumie się funkcję

$$A(n) = \sum_{k \geq 0} kp_{nk}$$

tzn. wartość oczekiwana  $\text{ave}(X_n)$  zmiennej losowej  $X_n$ .

Aby stwierdzić, na ile funkcje  $W(n)$  i  $A(n)$  są reprezentatywne dla wszystkich danych wejściowych rozmiaru  $n$ , rozważa się miary wrażliwości algorytmu: **miarę wrażliwości pesymistycznej**, czyli  $\Delta(n) = \sup\{t(d_1) - t(d_2): d_1, d_2 \in D_n\}$ , oraz **miarę wrażliwości oczekiwanej**, czyli  $\delta(n) = \text{dev}(X_n)$ , gdzie  $\text{dev}(X_n)$  jest standardowym odchyleniem zmiennej losowej  $X_n$ , tzn.  $\text{dev}(X_n) = \sqrt{\text{var}(X_n)}$  i  $\text{var}(X_n) = \sum_{k \geq 0} (k - \text{ave}(X_n))^2 p_{nk}$  ( $\text{var}(X_n)$  jest wariancją zmiennej losowej  $X_n$ ). Im większe są wartości funkcji  $\Delta(n)$  i  $\delta(n)$ , tym algorytm jest bardziej wrażliwy na dane wejściowe i tym bardziej jego zachowanie w wypadku rzeczywistych danych może odbiegać od zachowania opisanego funkcjami  $W(n)$  i  $A(n)$ .

□ **PRZYKŁAD:** Przeszukiwanie sekwencyjne ciągu

Dane wejściowe:  $L, N, a$ , gdzie  $N$  jest liczbą naturalną  $N \geq 0$ ,  $a$  jest poszukiwanym elementem,  $L[1..N + 1]$  jest tablicą, w której na miejscach od 1 do  $N$  znajdują się elementy ciągu.

**Wynik:** Zmienna logiczna  $p$  taka, że  $p = \text{true} \equiv a$  znajduje się w  $L[1..N]$

**Algorytm:**

```

begin
  j := 1;
  L[N + 1] := a;
  while L[j] ≠ a do j := j + 1;
  p := j ≤ N
end;
```

Rozmiar danych wejściowych:  $n = N$

Operacja dominująca: porównanie:  $L[j] \neq a$

Pesymistyczna złożoność czasowa:  $W(n) = n + 1$

Pesymistyczna wrażliwość czasowa:  $\Delta(n) = n$

A jaką jest oczekiwana złożoność czasowa? Założymy, że prawdopodobieństwo znalezienia  $a$  na każdym z  $n$  możliwych miejsc jest takie samo i wiadomo, że  $a$  jest w  $L[1..N]$ , tzn. że

$$p_{nk} = \frac{1}{n} \text{ dla } k = 1, 2, \dots, n$$

Wówczas

$$A(n) = \sum_{k=1}^n kp_{nk} = \frac{1}{n} \sum_{k=1}^n k = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Oczekiwana wrażliwość czasowa:

$$\begin{aligned} \text{var}(X_n) &= \sum_{k=1}^n \left( k - \frac{n+1}{2} \right)^2 \frac{1}{n} = \\ &= \frac{1}{n} \left( \frac{n(n+1)(2n+1)}{6} - \frac{2(n+1)}{2} \cdot \frac{n(n+1)}{2} + n \left( \frac{n+1}{2} \right)^2 \right) = \\ &= \frac{(n+1)(2n+1)}{6} - \frac{(n+1)^2}{4} = \frac{n+1}{12} (4n+2-3n-3) = \frac{n^2-1}{12} \equiv \frac{1}{12} n^2 \end{aligned}$$

czyli

$$\delta(n) \equiv .29n$$

Zauważmy, że zarówno funkcje wrażliwości powyższego algorytmu, jak i funkcje jego złożoności są liniowe; wynika stąd duża wrażliwość liczby operacji dominujących na dane wejściowe. ■

Faktyczna złożoność czasowa algorytmu (czas działania) w chwili jego użycia jako programu różni się od wyliczonej teoretycznie współczynnikiem proporcjonalności, który zależy od konkretnej realizacji tego algorytmu. Istotną zatem częścią informacji, która jest zawarta w funkcjach złożoności  $W(n)$  i  $A(n)$ , jest ich rzad wielkości, czyli ich zachowanie asymptotyczne, gdy  $n$  dąży do nieskończoności. Zwykle staramy się podać jak najprostsza funkcję charakteryzującą rzad wielkości  $W(n)$  i  $A(n)$ , na przykład  $n$ ,  $n \log n$ ,  $n^2$ ,  $n^3$ .

Używamy w tym celu następujących oznaczeń dla rzędów wielkości funkcji. Niech  $f, g, h: N \rightarrow R_+ \cup \{0\}$ , gdzie  $N$  i  $R_+$  oznaczają zbiory liczb – odpowiednio – naturalnych i rzeczywistych dodatnich.

Mówimy, że  $f$  jest co najwyżej rzędem  $g$ , co zapisujemy jako  $f(n) = O(g(n))$ , jeśli istnieją stała rzeczywista  $c > 0$  i stała naturalna  $n_0$  takie, że nierówność  $f(n) \leq cg(n)$  zachodzi dla każdego  $n \geq n_0$ . Oto przykład:  $n^2 + 2n = O(n^2)$ , bo  $n^2 + 2n \leq 3n^2$  dla każdego naturalnego  $n$ .

Mówimy, że  $f$  jest co najmniej rzędem  $g$ , co zapisujemy jako  $f(n) = \Omega(g(n))$ , jeśli  $g(n) = O(f(n))$ .

Mówimy, że  $f$  jest dokładnie rzędem  $g$ , co zapisujemy jako  $f(n) = \Theta(g(n))$ , jeśli zarówno  $f(n) = O(g(n))$ , jak i  $f(n) = \Omega(g(n))$ . Poprawny jest też termin  $f$  jest asymptotycznie równoważne  $g$  i oznaczenie  $f(n) \equiv g(n)$ . Oto przykład:  $n^2 + 2n \equiv n^2$ , bo zarówno  $n^2 + 2n \leq 3n^2$ , jak i  $n^2 + 2n \geq n^2$  dla każdego  $n \geq 0$ .

Będziemy także używać oznaczenia  $f(n) = g(n) + O(h(n))$ , gdy  $f(n) - g(n) = O(h(n))$ , na przykład  $(1/2)n^2 + 5n + 1 = (1/2)n^2 + O(n)$ . Zauważmy, że w ten sposób zachowujemy współczynnik proporcjonalności przy najbardziej znaczącym składniku sumy i pomijamy współczynniki przy mniej znaczących składnikach sumy.

Rzędy wielkości dwóch funkcji  $f(n)$  i  $g(n)$  mogą być porównane przez obliczenie granicy

$$E = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

Jeśli  $E = +\infty$ , to  $g(n) = O(f(n))$ , ale nie  $f(n) = O(g(n))$ .

Jeśli  $E = c > 0$ , to  $f(n) \equiv g(n)$ .

Jeśli  $E = 0$ , to  $f(n) = O(g(n))$ , ale nie  $g(n) = O(f(n))$ .

Stosując na przykład regułę de L'Hospitala, otrzymujemy

$$\lim_{n \rightarrow \infty} \frac{n \log n}{n^2} = \lim_{n \rightarrow \infty} \frac{\ln n}{n \ln 2} = \lim_{n \rightarrow \infty} \frac{1/n}{\ln 2} = 0$$

czyli  $n \log n = O(n^2)$ , ale nie  $n^2 = O(n \log n)$ .

Większość rozważanych algorytmów ma złożoność czasową proporcjonalną do jednej z podanych tu funkcji.

$\log n$  – złożoność logarytmiczna

Czas działania logarytmiczny występuje na przykład dla algorytmów typu: zadanie rozmiaru  $n$  zostaje sprowadzone do zadania rozmiaru  $n/2$  + pewna stała liczba działań, na przykład poszukiwanie binarne w ciągu uporządkowanym  $a_1 \leq a_2 \leq \dots \leq a_n$ .

Aby stwierdzić, czy  $x$  znajduje się w tym ciągu, porównujemy  $x$  najpierw z  $a_{\lfloor \frac{n}{2} \rfloor}$ <sup>10</sup>. Jeśli  $x < a_{\lfloor \frac{n}{2} \rfloor}$ , to szukamy dalej  $x$  w ciągu  $a_1 \leq a_2 \leq \dots \leq a_{\lfloor \frac{n}{2} \rfloor - 1}$ . Jeśli natomiast  $x > a_{\lfloor \frac{n}{2} \rfloor}$ , to szukamy  $x$  w ciągu  $a_{\lfloor \frac{n}{2} \rfloor + 1} \leq \dots \leq a_n$  (następny podciąg jest zawsze co najmniej o połowę krótszy).

#### $n$ – złożoność liniowa

Czas działania liniowy występuje na przykład dla algorytmów, w których jest wykonywana pewna stała liczba działań dla każdego z  $n$  elementów danych wejściowych. Przykładem takiego algorytmu jest algorytm Hornera wyznaczania wartości wielomianu.

#### $n \log n$ – złożoność $n \log n$ (liniowo-logarytmiczna)

Czas działania  $n \log n$  występuje na przykład dla algorytmów typu: zadanie rozmiaru  $n$  zostaje sprowadzone do dwóch podzadań rozmiaru  $n/2$  plus pewna liczba działań, liniowa względem rozmiaru  $n$ , potrzebnych do wykonania najpierw rozbicia, a następnie scalenia rozwiązań rozmiaru  $n/2$  w rozwiązywanie rozmiaru  $n$ . W ten sposób działa **mergesort** – algorytm sortowania przez scalanie. Aby uporządkować ciąg  $a_1, a_2, \dots, a_n$  sortujemy najpierw niezależnie podciągi:

$$\begin{aligned} a_1, a_2, \dots, a_{\lfloor \frac{n}{2} \rfloor} \\ a_{\lfloor \frac{n}{2} \rfloor + 1}, \dots, a_n \end{aligned}$$

a następnie scalamy posortowane podciągi w jeden uporządkowany ciąg.

#### $n^2$ – złożoność kwadratowa

Czas działania kwadratowy występuje na przykład dla algorytmów, w których jest wykonywana pewna stała liczba działań dla każdej pary elementów danych wejściowych (podwójna instrukcja iteracyjna).

#### $n^3, n^4, \dots$ – następne złożoności wielomianowe

#### $n^{\log n} = 2^{\log^2 n}$ – złożoność podwykładnicza

#### $2^n$ – złożoność wykładnicza $2^n$

Czas działania  $2^n$  ma na przykład algorytm, w którym jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych.

#### $n!$ – złożoność wykładnicza $n!$

Czas działania  $n!$  ma na przykład algorytm, w którym jest wykonywana stała liczba działań dla każdej permutacji danych wejściowych.

<sup>10</sup> Dla liczby rzeczywistej  $x$  zapis  $\lfloor x \rfloor$  oznacza największą liczbę całkowitą, nie większą niż  $x$ , a zapis  $\lceil x \rceil$  – najmniejszą liczbę całkowitą, nie mniejszą niż  $x$ .

Zauważmy, że algorytm o złożoności wykładniczej może być zrealizowany jedynie dla małych rozmiarów danych. Istnieje próg, od którego funkcja wykładnicza zaczyna rosnąć tak szybko, że realizacja algorytmu na komputerze staje się niemożliwa. Założymy na przykład, że dla danych rozmiaru  $n$  jest wykonywanych  $2^n$  operacji jednostkowych i że każda operacja jednostkowa zajmuje odpowiednio  $10^{-6}$  i  $10^{-9}$  sekund na dwóch różnych komputerach. Czas działania potrzebny do realizacji algorytmu jest przedstawiony w tabeli 1.1.

Tabela 1.1. Porównanie czasów realizacji algorytmu wykładniczego na dwóch komputerach

Rozmiar $n$	20	50	100	200
Czas działania ( $2^n/10^9$ )	1,04 s	35,7 lat	$4 \cdot 10^{14}$ wieków	$5 \cdot 10^{44}$ wieków
Czas działania ( $2^n/10^9$ )	0,001 s	13 dni	$4 \cdot 10^{11}$ wieków	$5 \cdot 10^{41}$ wieków

Widać, że nawet 1000-krotne przyspieszenie szybkości działania komputera niewiele pomaga algorytmowi wykładniczemu. Nierealizowalność uważa się za wewnętrzną cechę algorytmu o złożoności wykładniczej. Aby jednak mieć pełny obraz sytuacji, powinniśmy jeszcze rozważyć wrażliwość algorytmu na dane wejściowe. Może się zdarzyć, że dla danego algorytmu  $W(n) = 2^n + O(1)$ , ale także  $\Delta(n) = 2^n + O(1)$ . Wówczas nie możemy twierdzić, że algorytm jest nierealizowalny dla reprezentatywnych danych. Dane wejściowe, dla których czas działania jest wykładniczy, mogą się nigdy nie pojawić w rzeczywistych okolicznościach! Właśnie taka sytuacja zachodzi dla metody simplex programowania liniowego. Choć metoda ta ma złożoność wykładniczą dla „najgorszych” danych, dla pojawiających się w praktyce danych wejściowych działa w czasie wielomianowym, a nawet liniowym. Co więcej, w wypadku takich danych przewyższa metodę elipsoidalną, której pesymistyczna złożoność czasowa jest wielomianowa!

Przy korzystaniu z wyników analizy złożoności algorytmu należy zatem brać pod uwagę następujące uwarunkowania:

- algorytm i jego realizacja przeznaczona do wykonania są zwykle wyrażone w dwóch całkowicie różnych językach;
- wrażliwość algorytmu na dane wejściowe może spowodować, że faktyczne zachowanie się algorytmu na używanych danych będzie odbiegać od zachowania opisanego funkcjami złożoności  $W(n)$  i  $\Delta(n)$ ;
- może być trudno przewidzieć rzeczywisty rozkład prawdopodobieństwa zmiennej losowej  $X_n$ ;
- dla niektórych algorytmów nie są znane matematyczne oszacowania wielkości  $W(n)$  i  $\Delta(n)$ ; szczególnie wyznaczenie  $\Delta(n)$  dla rzeczywistego rozkładu prawdopodobieństwa może stanowić bardzo trudny problem matematyczny;
- czasami działanie dwóch algorytmów trudno jest jednoznacznie porównać; jeden działa lepiej dla pewnej klasy zestawów danych, a drugi dla innych.

Ważną cechą algorytmu jest jego prostota, z której zwykle wynika mniejszy współczynnik proporcjonalności przy złożoności obliczeniowej oraz łatwość realizacji (zaprogramowania). Szczególnie więc w dwóch przypadkach:

- program, w którym jest stosowany nasz algorytm, ma być wykonany raz lub tylko kilka razy;
- algorytm ma być stosowany tylko dla małych rozmiarów danych;

należy wybierać algorytm raczej pod kątem jego prostoty niż małej złożoności obliczeniowej (oczywiście najlepiej używać zawsze algorytmów zarówno prostych, jak i szybkich w sensie asymptotycznym).

## 1.2.

### Równania rekurencyjne

Wyznaczenie złożoności algorytmu sprowadza się często do rozwiązywania równania rekurencyjnego. Stosowane są zwykle dwie metody: rozwinięcie równania do sumy (metoda 1) i znalezienie funkcji tworzącej (metoda 2).

Metodą 2 zajmiemy się w następnym podrozdziale. Teraz pokażemy zastosowanie metody 1 do rozwiązywania trzech często pojawiających się równań rekurencyjnych ( $c$  oznacza stałą naturalną dodatnią).

$$(1) \begin{cases} T(1) = 0 \\ T(n) = T(\lfloor n/2 \rfloor) + c \text{ dla } n > 1 \end{cases}$$

(Równanie to otrzymujemy jako równanie złożoności wtedy, kiedy problem rozmiaru  $n$  sprowadza się do podproblemu rozmiaru połowę mniejszego).

Metoda rozwiązywania takiego równania polega na podstawieniu  $n = 2^k$  (tj. potęgi dwójki), rozwiązywania powstającego równania. Stąd możemy już wnioskować (zob. zad. 1.4), że rzяд wielkości rozwiązywania oryginalnego równania jest taki sam jak równania dla potęg dwójki.

Podstawmy więc  $n = 2^k$ . Wtedy

$$T(2^k) = T(2^{k-1}) + c = T(2^{k-2}) + c + c = T(2^0) + kc = kc = c \log n$$

Stąd wynika, że

$$T(n) = \Theta(\log n)$$

$$(2) \begin{cases} T(1) = 0 \\ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c \text{ dla } n > 1 \end{cases}$$

(Równanie to otrzymujemy jako równanie złożoności wtedy, kiedy problem rozmiaru  $n$  sprowadza się do dwóch podproblemów rozmiaru  $n/2$  + stała liczba działań). Podstawmy więc  $n = 2^k$ . Wtedy

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + c = 2(2T(2^{k-2}) + c) + c = 2^2T(2^{k-2}) + 2^1c + 2^0c = \\ &= 2^kT(2^0) + c(2^{k-1} + 2^{k-2} + \dots + 2^0) = 0 + c \frac{2^k - 1}{2 - 1} = c(n - 1) \end{aligned}$$

Stąd, jak poprzednio, wnioskujemy, że

$$T(n) = \Theta(n)$$

$$(3) \begin{cases} T(1) = 0 \\ T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \text{ dla } n > 1 \end{cases}$$

(Równanie to otrzymamy jako równanie złożoności wtedy, kiedy problem rozmiaru  $n$  sprowadza się do dwóch podproblemów rozmiaru  $n/2$  + liniowa liczba działań). Podstawmy  $n = 2^k$ . Wtedy

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + c2^k = 2(2T(2^{k-2}) + c2^{k-1}) + c2^k = \\ &= 2^2T(2^{k-2}) + c2^k + c2^k = 2^kT(2^0) + kc2^k = 0 + cn \log n \end{aligned}$$

Mamy zatem

$$T(n) = \Theta(n \log n)$$

## 1.3.

### Funkcje tworzące

Czasami trudno wyznaczyć rozwiązywanie równania  $T(n)$  bezpośrednio z równania rekurencyjnego (może nie istnieć związkły wzór). Można wówczas spróbować zastosować metodę funkcji tworzących, która polega na znalezieniu funkcji

$$F(z) = \sum_{n \geq 0} T(n)z^n$$

nazywanej funkcją tworzącą  $T(n)$ , i na jej podstawie wnioskować o własnościach samej funkcji  $T(n)$ .

Metodę tę stosuje się często w analizie probabilistycznej algorytmów (do wyznaczenia wartości oczekiwanej i wariancji zmiennej losowej  $X_n$ ). Rozważmy funkcję tworzącą

rozkładu prawdopodobieństwa  $p_{nk}$  zmiennej losowej  $X_n$  (z równań rekurencyjnych na  $p_{nk}$  trudno jest często wyznaczyć rozwiązanie):

$$P_n(z) = \sum_{k \geq 0} p_{nk} z^k$$

Zauważmy, że wówczas

$$\sum_{k \geq 0} p_{nk} = 1$$

Wartość oczekiwana i wariancję zmiennej losowej  $X_n$  można wyrazić za pomocą wartości pochodnych funkcji  $P_n(z)$  dla  $z = 1$  w następujący sposób:

$$\text{ave}(X_n) = P'_n(1)$$

$$\text{var}(X_n) = P''_n(1) + P'_n(1) - P'_n(1)^2$$

ponieważ

$$P'_n(z) = \sum_{k \geq 1} k p_{nk} z^{k-1}$$

$$P''_n(z) = \sum_{k \geq 2} k(k-1) p_{nk} z^{k-2}$$

$$\text{Stąd } P'_n(1) = \sum_{k \geq 1} k p_{nk} \text{ i } P''_n(1) = \sum_{k \geq 2} k(k-1) p_{nk}, \text{ a zatem}$$

$$\begin{aligned} \text{var}(X_n) &= \sum_{k \geq 0} (k - P'_n(1))^2 p_{nk} = \sum_{k \geq 0} k^2 p_{nk} - 2P'_n(1) \sum_{k \geq 0} k p_{nk} + P'_n(1)^2 \sum_{k \geq 0} p_{nk} = \\ &= \sum_{k \geq 0} k(k-1) p_{nk} + \sum_{k \geq 0} k p_{nk} - 2P'_n(1)^2 + P'_n(1)^2 = P''_n(1) + P'_n(1) - P'_n(1)^2 \end{aligned}$$

Można więc wyznaczyć wielkości  $\text{ave}(X_n)$  i  $\text{var}(X_n)$  (a co za tym idzie również złożoność oczekiwana i oczekiwana wrażliwość algorytmu), nie znając explicite rozkładu  $p_{nk}$ , a tylko jego funkcję tworzącą.

## 1.4.

### Poprawność semantyczna

Poprawność semantyczna oznacza, że program wykonuje postawione przed nim zadanie. Stosowaną metodą dowodu jest indukcja matematyczna względem liczby powtó

rzeń instrukcji iteracyjnej bądź poziomu zagnieźdżenia realizacji procedury rekurencyjnej. Rozważmy na przykład algorytm binarny potęgowania:

```

 $\{n \geq 0\}$ 
 $z := x; y := 1; m := n;$ 
while  $m \neq 0$  do
  begin  $\{\gamma: x^n = y^*z^m \wedge m > 0\}$ 
    if  $\text{odd}(m)$  then  $y := y^*z;$ 
     $m := m \text{ div } 2;$ 
     $z := z^*z$ 
  end;
 $\{y = x^n\}$ 

```

Warunek  $\gamma$ , nazywany **niezmiennikiem instrukcji iteracyjnej**, opisuje wartości zmiennych w trakcie realizacji programu. Zamieszczony warunek  $\gamma$  jest spełniony na początku, gdy rozpoczyna się realizacja instrukcji iteracyjnej, i każde powtórzenie tej instrukcji zachowuje go. Zachodzi on zatem, gdy kończy się realizacja instrukcji iteracyjnej, z czego łatwo wyprowadzić warunek końcowy  $y = x^n$ . Niezmiennik jest zwykle rozszerzeniem warunku końcowego, jak to zwykle bywa przy dowodach indukcyjnych. Chociaż przedstawiony dowód dowodzi warunku  $\{y = x^n\}$ , to jednak nie tłumaczy działania algorytmu. Aby zrozumieć, jak działa algorytm, przypatrzmy się postaci binarnej liczby  $m$  wewnętrz algorytmu.

```

 $\{n = (a_j a_{j-1} \dots a_0)_2, a_i \in \{0, 1\}, a_j = 1\}$ 
 $z := x; y := 1; m := n; k := 0;$ 
while  $m > 0$  do
  begin
     $\{\gamma': m = (a_j \dots a_k)_2 \wedge z = x^{2^k} \wedge y = x^{(a_{k-1} \dots a_0)_2} \wedge l \geq k \wedge a_j = 1\}$ 
    if  $\text{odd}(m)$  then  $y := y^*z;$ 
     $m := m \text{ div } 2; z := z^*z;$ 
     $k := k + 1$ 
  end;
 $\{y = x^n\}$ 

```

Zazwyczaj wymaga się od dowodów poprawności, aby na ich podstawie można było zrozumieć, jak faktycznie działa algorytm i dlaczego jest poprawny.

Aby dowód poprawności był kompletny, musimy jeszcze dodatkowo udowodnić dwie własności:

- wykonalność operacji częściowych, jak dzielenie, przechodzenie po dowiązaniu w drzewie lub liście, określenie zmiennej indeksowanej itp.;
- skończoność działania każdej instrukcji iteracyjnej i każdego wywołania procedury rekurencyjnej.

W wypadku algorytmu potęgowania binarnego jedyna częściowa operacja **div** jest zawsze wykonalna (gdyż dzielimy przez 2) oraz obliczenie instrukcji iteracyjnej jest kończone na mocy następującej własności liczb naturalnych: dla każdej liczby naturalnej  $n$ , wykonując dzielenie całkowite  $n$  wielokrotnie przez 2, po skończonej liczbie kroków otrzymamy 0. Co więcej, liczba wykonan instrukcji iteracyjnej jest równa liczbie dzielenów całkowitych przez 2, czyli długości binarnej  $n$ . Dla  $n > 0$  mamy zatem

$$W(n) = A(n) = \lfloor \log n \rfloor + 1 = \log n + O(1)$$

$$\Delta(n) = \delta(n) = 0$$

(rozmiarem danych jest  $n$ , a operacją dominującą – dzielenie całkowite przez 2). Widzimy, że w tym wypadku dowodzenie skończości działania instrukcji iteracyjnej jest połączane ze znajdowaniem pesymistycznej złożoności czasowej.

Jako przykład algorytmu rekurencyjnego rozważmy algorytm Euklidesa znajdowania największego wspólnego dzielnika dwóch dodatnich liczb naturalnych.

```
function NWD(x, y : integer) : integer;
var x : integer;
begin {α: x > 0 ∧ y > 0}
  r := x mod y;
  if r = 0 then NWD := y else NWD := NWD(y, r)
  {β: NWD = (x, y)}
end;
```

Przez  $(x, y)$  oznaczyliśmy największy wspólny dzielnik dodatnich liczb naturalnych  $x$  i  $y$ . Poprawność funkcji NWD względem podanych warunków pokazujemy dowodząc, że dla każdych dodatnich wartości naturalnych  $x$  i  $y$  obliczenie wywołania funkcji NWD( $x, y$ ) kończy się z wartością NWD =  $(x, y)$ . Stosujemy indukcję względem wartości  $y$ . Zasadając poprawność dla wszystkich  $0 < y < x$ , otrzymujemy, że  $x \bmod y = 0$  i wtedy  $(x, y) = y$ . Możemy też zastosować założenie indukcyjne dla pary  $(y, x \bmod y)$  i wewnętrzne wywołania rekurencyjnego. Wtedy  $(x, y) = (y, x \bmod y)$ .

## 1.5.

### Podstawowe struktury danych

Poniżej rozważamy podstawowe struktury danych: listę, graf, zbiór i drzewo, wprowadzając potrzebne w dalszej części książki oznaczenia i omawiając podstawowe metody implementacji tych struktur. Będziemy zakładać, że elementy wchodzące w skład rozważanych struktur danych pochodzą z pewnego niepustego uniwersum  $U$ . Jak wiadomo z zasad programowania strukturalnego, zagadnienia dotyczące budowy samej struktury danych i jej użycia w algorytmie wygodnie jest rozważać oddzielnie.

#### 1.5.1.

##### Lista

**Lista**<sup>b)</sup> to skończony ciąg elementów:  $q = [x_1, x_2, \dots, x_n]$ . Skrajne elementy listy  $x_1$  i  $x_n$  nazywają się **końcami** listy (odpowiednio – **lewym** i **prawym**), a wielkość  $|q| = n$  **długością** (lub **rozmiarem**) listy. Szczególnym przypadkiem listy jest lista pusta:  $q = []$ .

Weźmy dwie listy:  $q = [x_1, x_2, \dots, x_n]$  i  $r = [y_1, y_2, \dots, y_m]$ , i niech  $0 \leq i \leq j \leq n$ :

Podstawowymi abstrakcyjnymi operacjami na listach są:

- **dostęp** do elementu listy –  $q[i] = x_i$ ;
- **podlista** –  $q[i..j] = [x_i, x_{i+1}, \dots, x_j]$ ;
- **złożenie** –  $q \& r = [x_1, \dots, x_n, y_1, \dots, y_m]$ .

Za pomocą tych trzech podstawowych operacji można definiować inne operacje na listach, na przykład wstawianie elementu  $x$  za element  $x_i$  na liście  $q$ :  $q[1..i] \& [x] \& q[i+1..|q|]$ .

Listy używa się zwykle w specjalny sposób, ograniczając się do zmian jej końców:

- (a)  $front(q) = q[1]$  (pobieranie lewego końca listy);
- (b)  $push(q, x) = [x] \& q$  (wstawienie elementu  $x$  na lewy koniec listy);
- (c)  $pop(q) = q[2..|q|]$  (usunięcie bieżącego lewego końca listy);
- (d)  $rear(q) = q[|q|]$  (pobieranie prawego końca listy);
- (e)  $inject(q, x) = q \& [x]$  (wstawienie elementu  $x$  na prawy koniec listy);
- (f)  $eject(q) = q[1..|q|-1]$  (usunięcie bieżącego prawego końca listy).

Listę, na której można wykonać wszystkich sześć operacji, nazywa się **kolejką podwójną**. W szczególnych przypadkach, tzn. kiedy uwzględnia się tylko operacje *front*, *push* i *pop*, nazywa się ją **stosem**, a kiedy uwzględnia się tylko operacje *front*, *pop* i *inject* – **kolejką**. (Operacje na abstrakcyjnej strukturze danych mogą być realizowane za pomocą funkcji albo procedur).

Dwie podstawowe implementacje (reprezentacje) listy  $q = [x_1, x_2, \dots, x_n]$  to:

- **tablicowa** –  $q[i] = x_i$ , gdzie  $1 \leq i \leq n$ ,
- **dowiązaniowa** – różne warianty są przedstawione na rysunku 1.1.

W implementacjach pojedynczej liniowej i podwójnej liniowej dowiązanie prowadzące do listy wskazuje na pierwszy element na liście, a w implementacji pojedynczej cyklicznej i podwójnej cyklicznej – na ostatni. Aby mieć gwarancję, że struktura dowiązaniowa nigdy nie będzie pusta, dodaje się na początku listy element pusty, nazywany **głową** lub **wartownikiem** listy.

<sup>b)</sup> W tym sensie lista jest tym samym co w matematyce ciąg.

Pojedyncza liniowa



Pojedyncza cykliczna



Podwójna liniowa



Podwójna cykliczna



Rys. 1.1. Różne warianty implementacji dowiązanej listy

Następujące operacje na listach mają stałą złożoność czasową:

- w implementacji pojedynczej liniowej: operacje stosu, wstawianie jednego elementu za drugi, usuwanie następnego elementu;
- w implementacji pojedynczej cyklicznej: te operacje co wyżej plus złożenie oraz operacje *rear* i *inject*;
- w implementacji podwójnej cyklicznej: te operacje co wyżej plus *eject*, wstawianie jednego elementu przed drugim, usuwanie danego elementu, odwracanie listy.

Każda zatem operacja dotycząca kolejki podwójnej ma pesymistyczną złożoność czasową  $O(1)$  w implementacji podwójnej cyklicznej. Wadą tej implementacji jest użycie  $O(n)$  komórek pomocniczej pamięci na pamiętanie dowiązań ( $n$  jest rozmiarem listy).

Jeśli jest znana maksymalna długość  $m$  kolejki podwójnej, to bardziej oszczędna pamięciowo jest implementacja listy za pomocą tablicy cyklicznej  $Q[0..m-1]$ , w której następnikiem pozycji  $0 \leq i \leq m-1$  jest pozycja  $(i+1) \bmod m$ . Wówczas jeśli  $q = [x_1, x_2, \dots, x_n]$ , to  $Q[(k+i) \bmod m] = x_i$  dla  $1 \leq i \leq n$  i pewnej pozycji  $0 \leq k \leq m-1$ . Przykładowo operacja *pop*( $q$ ) ma implementację:

```
pop(k, n) :: if n = 0
  then error10
  else
    begin
      k := (k + 1) mod m;
      n := n - 1;
    end;
```

<sup>10</sup> Instrukcja error oznacza przerwanie obliczeń.a operacja *push*( $q, x$ ):

```
push(k, n, x) :: if n = m
  then error
  else
    begin
      Q[k] := x;
      k := (k - 1) mod m;
      n := n + 1;
    end;
```

### 1.5.2.

#### Zbiór

W przeciwnieństwie do elementów listy elementy w zbiorze  $S = \{x_1, x_2, \dots, x_n\}$  nie są podane w żadnym ustalonym porządku. (Zawsze będziemy zakładać, że rozważany zbiór jest skończony). Liczbę  $n$  elementów w zbiorze  $S$  oznaczamy przez  $|S|$  i nazywamy rozmiarem zbioru  $S$ . Podstawowymi operacjami na zbiorach są:

- (a) *insert*( $x, S$ )::  $S := S \cup \{x\}$  (wstawienie elementu  $x$  do zbioru  $S$ );
- (b) *delete*( $x, S$ )::  $S := S - \{x\}$  (usunięcie elementu  $x$  ze zbioru  $S$ );
- (c) *member*( $x, S$ ):: wynikiem jest wartość  $\begin{cases} \text{true}, & \text{jeśli } x \in S \\ \text{false}, & \text{jeśli } x \notin S \end{cases}$  (sprawdzenie, czy  $x$  jest elementem zbioru  $S$ );
- (d) *min*( $S$ ):: zwrócenie najmniejszego elementu w zbiorze  $S$  z uwzględnieniem pewnego ustalonego liniowego porządku  $\leq$ ;
- (e) *deletemin*( $S$ )::  $S := S - \{\min(S)\}$ ;
- (f) *union*( $S_1, S_2$ ):: obliczenie  $S_1 \cup S_2$  (przy założeniu, że zbiorы  $S_1$  i  $S_2$  są rozłączne).

Oto podstawowe implementacje zbioru  $S = \{x_1, x_2, \dots, x_n\}$ .

#### • Wektor charakterystyczny

Przy założeniu, że uniwersum  $U$  może służyć jako zbiór indeksów dla tablicy  $C$ , mamy

$$C[x] = \begin{cases} \text{true}, & \text{jeśli } x \in S \\ \text{false}, & \text{jeśli } x \notin S \end{cases}$$

Operacje *insert*, *delete* i *member* mają pesymistyczną złożoność czasową  $O(1)$ . Złożoność pamięciowa jest proporcjonalna do rozmiaru zbioru indeksów tablicy  $C$  (czyli faktycznie do rozmiaru uniwersum  $U$ ).

#### • Implementacje listowe

Oczywiście ustawiając elementy zbioru  $S$  w pewnym porządku, otrzymujemy listę. Wszystkie implementacje listy mogą być użyte do reprezentowania zbioru. Przy

rozważanych wcześniej implementacjach listy pesymistyczna złożoność czasowa podstawowych operacji na zbiorach jest proporcjonalna do rozmiaru zbiorów.

Opisując algorytmy, będziemy często stosować rozszerzenia języka Pascal. Należy to traktować jako postać roboczą – pośrednią przed ostatecznym, ścisłym zapisem algorytmu w języku programowania. Postać pośrednią ułatwia zrozumienie istoty działania algorytmu. Trzeba pamiętać, że postać ostateczna, zapisana w języku programowania, zawiera często taką liczbę szczegółów, że trudno wychwycić ideę algorytmu. Z tego właśnie względu przyjęło się stosować w dokumentacji oprogramowania pośrednią postać zapisu algorytmu. Przykładem użytecznej abstrakcji algorytmicznej jest konstrukcja

```
for each x in S do I
```

używana do opisu działań  $I$ , wykonywanych dla każdego elementu  $x$  należącego do listy lub zbioru  $S$ . Nie bierzemy tu pod uwagę mechanizmu przeglądania elementów w  $S$ ; koncentrujemy się na opisie przetwarzania każdego  $x$ , co zwykle stanowi istotę danego algorytmu. Instrukcje:

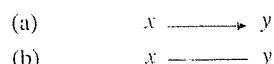
```
suma := 0;
for each x in S do suma := suma + x
```

przedstawiają sumowanie liczb w zbiorze  $S$ . Opis reprezentacji zbioru  $S$  i sposobu wyboru jego elementów odkładamy na później, koncentrując się w danej chwili tylko na jednym problemie (w tym wypadku na sumowaniu elementów zbioru  $S$ ).

W kolejnym podrozdziale podajemy więcej przykładów związanych z odkładaniem na później dokładnego opisu pewnych aspektów algorytmu.

### 1.5.3. Graf

**Graf** to system, który zapisujemy jako  $G = (V, E)$ , gdzie  $V$  oznacza zbiór skończony, którego elementy są nazywane **wierzchołkami** (gdy będzie nam zależało na podkreśleniu, że graf jest strukturą danych, używać też będziemy nazwy **węzły**), a  $E$  – zbiór krawędzi, czyli par wierzchołków ze zbioru  $V$ , przy czym, dokładniej, albo  $E$  jest podzbiorem zbioru par uporządkowanych  $\{(x, y) : x, y \in V \wedge x \neq y\}$  i wtedy graf nazywa się **zorientowany**, a krawędzie są oznaczane strzałkami łączącymi wierzchołki (rys. 1.2a), albo  $E$  jest podzbiorem zbioru wszystkich dwuelementowych podzbiorów zbioru  $V$  i wtedy graf nazywa się **niezorientowany**, a krawędzie są oznaczane liniami (rys. 1.2b).



Rys. 1.2. Dwa rodzaje krawędzi w grafach: (a) w grafie zorientowanym; (b) w grafie niezorientowanym

W obu tych wypadkach krawędź łączącą wierzchołki  $x$  i  $y$  oznaczamy jako  $(x, y)$ . Rozmiar grafu  $G = (V, E)$  jest równy sumie dwóch liczb:  $n = |V|$  i  $m = |E|$ . Dla grafu niezorientowanego (co oczywiste)  $m \leq \frac{n(n-1)}{2}$ , a dla zorientowanego  $m \leq n(n-1)$ .

Oto podstawowe implementacje grafu  $G = (V, E)$  (zakładamy, że zbiór węzłów  $V$  może być zbiorem indeksów dla tablic).

#### • Listy sąsiedztwa

Dla każdego  $x \in V$  budujemy listę (oznaczaną przez  $L[x]$ ) wierzchołków  $y$  będących sąsiadami  $x$  (tj.  $(x, y) \in E$ ).

W tej implementacji jest potrzebna pamięć  $O(n + m)$ .

#### • Macierz sąsiedztwa

$$A[x, y] = \begin{cases} 1, & \text{jeśli } (x, y) \in E \\ 0, & \text{jeśli } (x, y) \notin E \end{cases}$$

W tej implementacji jest potrzebna pamięć  $O(n^2)$ .

Definicje dotyczące grafów są zamieszczone w rozdziale poświęconym algorytmom na grafach. Teraz rozważymy tylko jeden przykład problemu grafowego.

#### □ PRZYKŁAD: Algorytm przechodzenia grafu

Niech  $G = (V, E)$ ,  $|V| = n$ ,  $|E| = m$ ,  $p \in V$ . Wychodząc od wierzchołka  $p$ , należy odwiedzić każdy wierzchołek i każdą krawędź, które są osiągalne z  $p$ . Dozwolonym ruchem jest przejście krawędzią grafu wychodzącą z odwiedzonego już wierzchołka.

W jednym kroku będziemy odwiedzać jeden z wierzchołków oraz jedną z krawędzi grafu i zaznaczać je jako odwiedzone. Na początku wszystkie wierzchołki i wszystkie krawędzie są zaznaczone jako nie odwiedzone.

1. Odwiedź wierzchołek  $p$  i zaznacz go jako odwiedzony.
2. Dopóki z jednego z odwiedzonych wierzchołków wychodzi nie odwiedzona jeszcze krawędź, wykonuj podane czynności:
  - a) wybierz odwiedzony wierzchołek, powiedzmy  $v$ , z którego wychodzi nie odwiedzona krawędź;
  - b) wybierz nie odwiedzoną krawędź, powiedzmy  $(v, w)$ , wychodzącą z wierzchołka  $v$ ;
  - c) zaznacz krawędź  $(v, w)$  jako odwiedzoną;
  - d) jeśli wierzchołek  $w$  nie został odwiedzony, odwiedź go i zaznacz jako odwiedzony.

Stosując indukcję względem odległości danego wierzchołka od wierzchołka początkowego  $p$ , możemy udowodnić, że w każdym algorytmie stosującym się do powyższego schematu musi nastąpić odwiedzenie jeden raz każdego wierzchołka i każdej krawędzi grafu  $G$  osiągalnej z  $p$ . Nie biorąc pod uwagę samej procedury odwiedzania wierzchołków i krawędzi, złożoność każdego takiego algorytmu jest proporcjonalna do łącznej liczby wierzchołków i krawędzi (a więc jest liniowa względem rozmiaru grafu).

Powyższy opis stanowi schemat klasy algorytmów. Aby otrzymać konkretny algorytm, należy wykonać podane tu kroki.

1. Przyjąć odpowiednią reprezentację grafu, na przykład  $V = \{1, 2, \dots, n\}$  i listy sąsiedztwa  $L[v]$  dla  $v \in V$ .
2. Określić, co to znaczy „zaznacz wierzchołek jako odwiedzony”. Można na przykład dodać do każdego wierzchołka  $v$  pole  $visited[v]$  i przyjąć, że  $false$  oznacza „wierzchołek nie odwiedzony”, a  $true$  „wierzchołek odwiedzony”.
3. Określić sposób wyboru odwiedzanego wierzchołka, z którego wychodzi nie odwiedzona krawędź. Można na przykład przechowywać takie wierzchołki w kolejce lub na stosie.
4. Określić sposób odróżniania (dla danego wierzchołka) krawędzi odwiedzonych od nie odwiedzonych. Można na przykład trzymać na liście sąsiedztwa danego wierzchołka  $v$  wskaźnik  $current[v]$  do pierwszej nie odwiedzonej krawędzi ( $current[v] = \text{nil}$  oznacza, że wszystkie krawędzie wychodzące z danego wierzchołka zostały odwiedzone).

Przyjmując te przykładowe ustalenia, uzyskujemy bardziej uszczegółowiony schemat przechodzenia grafu ( $visit$  jest procedurą „odwiedzania” wierzchołka; zrezygnowaliśmy z procedury odwiedzania krawędzi). Oto on:

```

for v := 1 to n do
begin
    visited[v] := false;
    ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
end;
visit(p); visited[p] := true;
if current[p] <> nil then
begin
    S := {p};
    while S <> ∅ do
    {S zawiera wszystkie odwiedzone do tej pory wierzchołki,
    z których wychodzą nie odwiedzone jeszcze krawędzie}
    begin
        wybierz wierzchołek v ze zbioru S;
        niech w będzie wierzchołkiem wskazywanym przez current[v];
        przesuń wskaźnik current[v] do następnego wierzchołka na
        liście L[v];
        if current[v] = nil then S := S - {v};
        if not visited[w] then
    end;
end;

```

```

begin
    visit(w);
    visited[w] := true;
    if current[w] <> nil then S := S ∪ {w}
end;
end;
end;

```

Dwie podstawowe implementacje zbioru  $S$  w tym schemacie algorytmów to stos i kolejka. Najpierw uszczegółowimy schemat przechodzenia grafu, przedstawiając  $S$  za pomocą stosu i interpretując operacje na  $S$  jako operacje na stosie.

(Przyjmujemy, że operacje  $pop$  i  $push$  będą realizowane jako procedury, a  $front$  jako funkcja). Otrzymujemy algorytm **przechodzenia grafu w głąb** (metoda DFS).

```

procedure dfs;
begin
    for v := 1 to n do
    begin
        visited[v] := false;
        ustaw wskaźnik current[v] na pierwszy wierzchołek na
        liście L[v]
    end;
    visit(p); visited[p] := true;
    if current[p] <> nil then
    begin
        S := []; push(S, p);
        while S <> ∅ do
        {stos S zawiera wszystkie odwiedzone do tej pory wierzchołki,
        z których wychodzą nie odwiedzone jeszcze krawędzie}
        begin
            v := front(S);
            niech w będzie wierzchołkiem wskazywanym przez
            current[v];
            przesuń wskaźnik current[v] do następnego wierzchołka
            na liście L[v];
            if current[v] = nil then pop(S);
            if not visited[w] then
            begin
                visit(w);
                visited[w] := true;
                if current[w] <> nil then push(S, w)
            end
        end;
    end;
end;

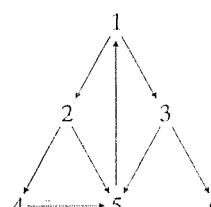
```

Zapiszemy teraz schemat przechodzenia grafu, przedstawiając  $S$  jako kolejkę i interpretując operacje na  $S$  jako operacje na kolejce. (Przyjmujemy, że operacje *pop* i *inject* będą realizowane jako procedury, a *front* jako funkcja). Otrzymujemy algorytm przechodzenia grafu wszerz (metoda BFS).

```

procedure bfs;
begin
  for v := 1 to n do
  begin
    visited[v] := false;
    ustaw wskaźnik current[v] na pierwszy wierzchołek na liście
    L[v]
  end;
  visit(p); visited[p] := true;
  if current[p] <> nil then
  begin
    S := []; inject(S, p);
    while S <> Ø do
    {kolejka S zawiera wszystkie odwiedzone do tej pory
    wierzchołki, z których wychodzą nie odwiedzone jeszcze
    krawędzie}
    begin
      v := front(S);
      niech w będzie wierzchołkiem wskazywanym przez
      current[v];
      przesuń wskaźnik current[v] do następnego wierzchołka
      na liście L[v];
      if current[v] = nil then pop(S);
      if not visited[w] then
      begin
        visit(w);
        visited[w] := true;
        if current[w] <> nil then inject(S, w)
      end
    end
  end
end;

```



Rys. 1.3. Przykładowy graf

Weźmy graf z rysunku 1.3.

Jeśli  $p = 1$ ,  $L[1] = [2, 3]$ ,  $L[2] = [4, 5]$ ,  $L[3] = [5, 6]$ ,  $L[4] = [5]$ ,  $L[5] = [1]$ ,  $L[6] = []$ , to podczas wykonywania algorytmu DFS nastąpi odwiedzenie wierzchołków grafu w następującej kolejności: 1, 2, 4, 5, 3, 6. Natomiast w wyniku realizacji algorytmu BFS nastąpi odwiedzenie wierzchołków grafu w następującej kolejności: 1, 2, 3, 4, 5, 6.

Zastosowana powyżej metoda konstrukcji algorytmów nazywa się **metodą kolejnych transformacji**. Wychodząc od ogólnego schematu algorytmu, dokonujemy kolejnych wyborów transformujących zapis algorytmu na coraz bardziej szczegółowy. W procedurach *DFS* i *BFS* zostawiliśmy jeszcze do „dopracowania” kilka spraw, na przykład kwestię reprezentacji list sąsiedztwa i wskaźników *current[v]*, przesuwających się po liście sąsiedztwa wierzchołka  $v$  w trakcie odwiedzania kolejnych krawędzi wychodzących z  $v$ . Ostatecznych (przed realizacją algorytmu) transformacji dokonuje kompilator, przekształcając program w binarny kod maszynowy.

#### 1.5.4.

#### Notacja funkcyjna dla atrybutów obiektów

Do oznaczenia atrybutów obiektów będziemy stosować notację funkcyjną. Jeśli  $X$  jest zbiorem węzłów w strukturze danych, a  $Y$  dowolnym zbiorem i jest określona funkcja  $f: X \rightarrow Y$ , to będziemy to zapisywać jako  $f(x)$ , na przykład  $f(x) := y$ .

Nie będziemy zatem stosować notacji wskazujących, jak dana funkcja jest realizowana przez konstrukcję języka programowania, to znaczy notacji obiektowej (wskaźnikowej):  $x^.f$  ( $f$  – nazwa pola rekordu), na przykład  $x^.f := y$ , albo notacji tablicowej:  $f[x]$  ( $f$  – nazwa tablicy), na przykład  $f[x] := y$ .

Jest to zgodne z zasadą przyjętą przez nas w tej książce, nakazującą oddzielanie poziomu abstrakcyjnego, charakteryzującego rodzaj i własności obiektów oraz operacji, od ich konkretnej realizacji. Funkcje określone na obiektach struktury danych są również abstrakcyjnymi strukturami danych, dopuszczającymi różne implementacje. Instrukcja przypisania  $f(x) := y$  jest abstrakcyjną operacją na obiekcie będącym funkcją (odwzorowaniem)  $f$ .

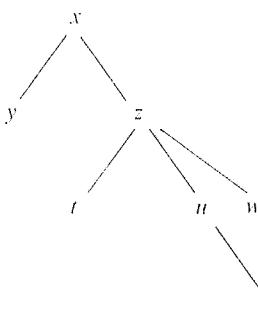
#### 1.5.5.

#### Drzewo

Drzewo to dowolny niezorientowany graf spójny i acykliczny. (Przypominamy, że spójność oznacza, iż każde dwa wierzchołki grafu są połączone ścieżką utworzoną z krawędzią grafu. Acykliczność oznacza brak cykli prostych utworzonych z krawędzią grafu). Drzewo z korzeniem to drzewo z wyróżnionym jednym wierzchołkiem nazywanym

**korzeniem**. Jeżeli z kontekstu będzie wynikać, iż chodzi o drzewo z korzeniem i korzeń drzewa jest ustalony, to będziemy po prostu pisać „drzewo”.

Zakładamy Drogie Czytelniku, że znasz podstawowe pojęcia dotyczące drzew. Przypomnijmy je na przykładzie drzewa (z korzeniem) z rysunku 1.4. Wierzchołek  $x$  ma trzy następcy:  $t$ ,  $u$  i  $w$ . Każdy wierzchołek, który ma następcę, jest **wierzchołkiem wewnętrznym**, w przeciwnym razie jest **liściem** (na przykład  $y$ ). Zbiór następców wierzchołka  $x$  w drzewie  $T$  będziemy oznaczać jako  $\text{children}_T(x)$ . (Będziemy pomijać indeks  $T$  wtedy, kiedy będzie jednoznacznie wiadomo, o które drzewo chodzi). Każdy wierzchołek z wyjątkiem korzenia ma **poprzednik** (na przykład  $v$  ma poprzednika  $u$ ). Poprzednik wierzchołka  $x$  w drzewie  $T$  będziemy oznaczać jako  $p_T(x)$ . (Będziemy pomijać indeks  $T$  wtedy, kiedy będzie jednoznacznie wiadomo, o które drzewo chodzi). **Potomkami** wierzchołka  $z$  są zarówno on sam, jak i jego następcy  $t$ ,  $u$  i  $w$ , ale także następnik  $v$  wierzchołka  $u$ . Z kolei za **przodków** wierzchołka  $v$  uważa się jego samego, jego poprzednika  $u$ , a także wierzchołki  $z$  i  $x$ , leżące na ścieżce od  $v$  do korzenia. **Głębokość** (lub **poziom**) wierzchołka w drzewie to jego odległość od korzenia ( $x$  na przykład ma głębokość 0, a  $v$  ma głębokość 3). **Wysokość** wierzchołka to maksymalna długość drogi od danego wierzchołka do liścia (na przykład wysokość  $x$  wynosi 3, a wysokość  $y$  – 0). **Wysokość drzewa** to wysokość jego korzenia (w naszym przykładzie wynosi 3).



Rys. 1.4. Drzewo o korzeniu  $x$

Drzewo reprezentuje się zwykle albo za pomocą struktury dowiązaniowej (używając rekordów i dowiązań), albo za pomocą struktury indeksowej (używając tablicy). Gdy struktura drzewa jest ustalona, stosuje się na ogół reprezentację tablicową, a gdy reprezentowane drzewo może mieć dowolny kształt – dowiązaniową. Często przy przedstawianiu algorytmów nie jest istotne, jaka reprezentacja jest używana. W takim wypadku będziemy stosować ogólną, abstrakcyjną notację funkcyjną  $p(x)$  i  $\text{children}(x)$  zamiast konkretnych  $x.p$  i  $x.\text{children}$  czy  $p[x]$  i  $\text{children}[x]$ .

□ **PRZYKŁAD:** Algorytm przechodzenia drzewa z korzeniem

W podanym tu algorytmie zakładamy, że  $\text{vertex}$  jest typem danych, reprezentującym wierzchołek drzewa, a  $\text{previsit}(v)$  i  $\text{postvisit}(v)$  to procedury specyfikujące działania wykonywane – odpowiednio – przy wejściu do wierzchołka  $v$  i przy wyjściu z niego.

```
procedure traverse(v : vertex);
var w : vertex;
begin
  previsit(v);
  for each w in children(v) do traverse(w);
  postvisit(v)
end traverse;
```

Aby przejść całe drzewo o korzeniu  $r$  wywołujemy  $\text{traverse}(r)$ . Gdy procedura  $\text{postvisit}(v)$  nie ma treści, mamy do czynienia z przejściem drzewa **metodą preorder**, a gdy procedura  $\text{previsit}(v)$  nie ma treści – **metodą postorder**. Metody preorder użyjemy na przykład do obliczenia głębokości wierzchołków drzewa, a metody postorder do obliczenia ich wysokości (zob. zadania 20 i 21).

Szczególnym przypadkiem drzewa z korzeniem jest **drzewo binarne**, w którym dla każdego wierzchołka  $v$  mamy  $|\text{children}(v)| \leq 2$ . W drzewie binarnym każdy następnik wierzchołka  $v$  jest albo **lewy**, albo **prawy** (przy czym tylko jeden może być lewy i tylko jeden prawy). Używać będziemy następujących oznaczeń:

$$\begin{aligned} \text{left}(v) &= \begin{cases} \text{lewy następnik } v, & \text{jeśli jest określony} \\ \text{nil} & \text{w przeciwnym razie} \end{cases} \\ \text{right}(v) &= \begin{cases} \text{prawy następnik } v, & \text{jeśli jest określony} \\ \text{nil} & \text{w przeciwnym razie} \end{cases} \end{aligned}$$

Dla każdego wierzchołka  $v$  zbiór potomków jego lewego następcy oraz zbiór potomków jego prawego następcy tworzą drzewa nazywane – odpowiednio – **lewym i prawym poddrzewem** wierzchołka  $v$ .

□ **PRZYKŁAD:** Algorytm przechodzenia drzewa binarnego

W podanym tu algorytmie zakładamy, jak poprzednio, że  $\text{vertex}$  jest typem danych, reprezentującym wierzchołek drzewa, a  $\text{previsit}(v)$ ,  $\text{invisit}(v)$  i  $\text{postvisit}(v)$  to procedury specyfikujące działania wykonywane – odpowiednio – przy wejściu do wierzchołka  $v$ , po rozpatrzeniu wierzchołków w lewym poddrzewie a przed rozpatrzeniem wierzchołków w prawym poddrzewie i przy wyjściu z wierzchołka  $v$ .

```
procedure b-traverse(v : vertex);
begin
  previsit(v);
  if left(v) ≠ nil then b-traverse(left(v));
  invit(v);
  if right(v) ≠ nil then b-traverse(right(v));
  postvisit(v)
end b-traverse;
```

Gdy procedury *invisit* i *postvisit* nie mają treści, mamy do czynienia z przejściem metodą *preorder*, a gdy *previsit* i *postvisit* nie mają treści – z przejściem metodą *inorder*. Gdy dotyczy to procedur *previsit* i *invisit* mamy do czynienia z przejściem metodą *postorder*.

## 1.6. Eliminacja rekursji

Algorytmy rekurencyjne są naturalne dla wielu struktur danych, a dla grafów i dla drzew w szczególności. Należy jednak pamiętać, że realizacja rekursji na maszynie wymaga użycia stosu, a to pociąga za sobą zwiększenie złożoności pamięciowej, a także współczynnika proporcjonalności w złożoności czasowej.

□ **PRZYKŁAD:** Przeglądanie drzewa binarnego metodą *inorder*

```
procedure inorder(v : vertex);
begin
  if left(v) ≠ nil then inorder(left(v));
  invisit(v);
  if right(v) ≠ nil then inorder(right(v))
end inorder;
```

Procedurę wywołujemy, przyjmując za parametr faktyczny korzeń drzewa *r*, czyli piszemy *inorder(r)*. Przy symulacji wywołania procedury rekurencyjnej *inorder* przez program nierekurencyjny należy zapisać na stosie wierzchołek bieżący *v* oraz miejsce, do którego należy wrócić po zrealizowaniu wywołania rekurencyjnego. W podanej dalej procedurze nazwa *stack* oznacza typ stosu, na którym umieszcza się pary wartości *[v, i]*, gdzie *v* jest wierzchołkiem, czyli wartością typu *vertex*, a *i* etykietą (operacji *push* i *pop* używamy jako procedur, a operacji *top* jako funkcji).

```
procedure nonrec-inorder1(r : vertex);
label 1, 2, 3;
var q : stack;
  v : vertex;
  i : 1..3;
begin
  q := []; v := r;
1: {początek symulacji wywołania rekurencyjnego}
  if left(v) ≠ nil then
  begin
    {symulacja wywołania rekurencyjnego inorder(left(v))}
```

```
push(q, [v, 2]);
{umieszczenie na stosie bieżącej wartości v i etykiety powrotu}
v := left(v); {przygotowanie nowego parametru v}
goto 1
end;
2: invisit(v);
if right(v) ≠ nil then
begin
{symulacja wywołania rekurencyjnego inorder(right(v))}
push(q, [v, 3]);
{umieszczenie na stosie bieżącej wartości v i etykiety powrotu}
v := right(v); {przygotowanie nowego parametru}
goto 1
end;
3: {powrót do miejsca, gdzie nastąpiło wywołanie rekurencyjne}
if q ≠ [] then
begin
  [v, i] := front(q);
  {przywrócenie wartości v z poprzedniej instancji
  i uzyskanie informacji o miejscu powrotu}
  pop(q); {skasowanie wierzchołka stosu}
  goto i {powrót}
end
end nonrec-inorder1;
```

Zauważmy, że złożoność pamięciowa wzrasta o maksymalną liczbę elementów na stosie *q*, czyli w wypadku procedury *inorder* o  $O(n)$ , gdzie *n* jest liczbą wierzchołków w drzewie.

Podany powyżej algorytm, będący przykładem zastosowania ogólnej metody eliminacji stosu, można nieco uproszczyć. Zauważmy, że gdy rozpoczynamy w danym węźle nową instancję, idąc w prawo, nie musimy już wracać do niego, gdyż nie pozostało już w nim nic do wykonania. Nie trzeba zatem umieszczać pary *[v, 3]* na stosie. Ponieważ powrót zawsze odbywa się w miejscu oznaczone w algorytmie etykietą 2, na stosie wystarczy przechowywać jedynie wierzchołki, do których ma nastąpić powrót.

```
procedure nonrec-inorder2(r : vertex);
label 1, 2, 3;
var q : stack;
  v : vertex;
begin
  q := []; v := r;
```

```

1: {początek symulacji wywołania rekurencyjnego}
  if left(v) ≠ nil then
    begin
      {symulacja wywołania rekurencyjnego inorder(left(v)) }
      push(q, v);
      {wstawienie na stos bieżącej wartości v; powrót zawsze do 2}
      v := left(v); {przygotowanie nowego parametru v}
      goto 1
    end;
2: invisit(v);
  if right(v) ≠ nil then
    begin
      {symulacja wywołania rekurencyjnego inorder(right(v)) }
      v := right(v); {przygotowanie nowego parametru}
      goto 1
    end;
  {powrót do instancji, gdzie nastąpiło wywołanie rekurencyjne}
  if q ≠ [] then
    begin
      v := front(q);
      {przywrócenie wartości v z poprzedniej instancji}
      pop(q); {skasowanie wierzchołka stosu}
      goto 2 {powrót}
    end
  end nonrec-inorder2;

```

## 1.7.

### Koszt zamortyzowany operacji w strukturze danych

W wypadku struktur danych jest używany jeszcze jeden rodzaj złożoności, tzw. **złożoność zamortyzowana (koszt zamortyzowany) operacji w strukturze danych**. Użycie struktury danych w algorytmie polega zwykle na wykonaniu na niej ciągu operacji  $o_1, o_2, \dots, o_m$  (jedna po drugiej, czyli w trybie on-line). Koszt  $i$ -tej operacji zapisujemy jako  $t_i$  ( $1 \leq i \leq m$ ). Na ogół to nie koszt rzeczywisty jednej operacji jest istotny, a koszt całego ciągu operacji, czyli

$$t = \sum_{i=1}^m t_i$$

Czasami, gdy nie da się bezpośrednio uzyskać zadowalającego oszacowania rzędu wielkości  $t$  (na przykład w sytuacji, kiedy koszt pewnych operacji jest mały, a innych duży),

warto skorzystać z metody wprowadzenia **kosztu zamortyzowanego operacji**. Polega ona na tym, że operacjom w wykonywanym ciągu przypisuje się koszty zamortyzowane  $a_1, a_2, \dots, a_m$  tak, żeby albo

$$\sum_{i=1}^m a_i = t$$

albo

$$t = O\left(\sum_{i=1}^m a_i\right)$$

Podstawową metodą liczenia kosztu zamortyzowanego jest **metoda potencjału**. Strukturze danych przyporządkowujemy potencjał  $\Phi$  o kolejnych wartościach nieujemnych  $\Phi_0 = 0, \Phi_1, \dots, \Phi_m$ , tak żeby

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

dla każdego  $i$  ( $1 \leq i \leq m$ ).

Sumując, otrzymujemy:

$$\sum_{i=1}^m a_i = \sum_{i=1}^m (t_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^m t_i + (\Phi_m - \Phi_0)$$

a zatem

$$t = \sum_{i=1}^m t_i \leq \sum_{i=1}^m a_i$$

#### PRZYKŁAD:

Rozważmy strukturę danych Stos z operacjami:

- (a) *empty*:: utworzenie pustego stosu – koszt rzeczywisty 1;
- (b) *push(x)*:: wstawienie  $x$  na stosie – koszt rzeczywisty 1;
- (c) *mpop(k)*:: usunięcie ze stosu  $k$  elementów – koszt rzeczywisty  $k$ .

Za potencjał stosu przyjmijmy jego rozmiar:  $\Phi(\text{Stos}) = |\text{Stos}|$ . Wtedy koszt zamortyzowany operacji *push* wynosi  $1 + 1 = 2$ , a koszt zamortyzowany operacji *mpop k*  $+ (-k) = 0$ .

Jeśli w ciągu wykonywanych operacji jest  $n$  operacji *push* oraz  $l$  operacji *mpop*, to całkowity koszt ich wykonania jest  $\leq 2n$ . Gdybyśmy szacowali go z góry przez koszty

pesymistyczne wykonania pojedynczych operacji, otrzymalibyśmy znacznie mniej dokładny wynik  $n + kl$ . Bardziej złożone przykłady użycia kosztu zamortyzowanego zamieszczymy w dalszej części książki.

## 1.8. Metody układania algorytmów

Proces układania algorytmu rozwiązuującego dane zadanie algorytmiczne ma charakter twórczy, nie dający się zalgorytmizować. Istnieją jednak ogólne metody, które w pewnych sytuacjach można zastosować. Ich użycie prowadzi często do skonstruowania szybkich algorytmów. W następnych rozdziałach zajmiemy się zastosowaniem takich właśnie ogólnych metod. W tym podrozdziale podajemy kilka z nich, z którymi powinieneś się już zetknąć przy nauce programowania.

### 1.8.1. Metoda „dziel i zwyciężaj”

Problem rozmiaru  $n$  zostaje podzielony na kilka podproblemów mniejszych rozmiarów w taki sposób, że z ich rozwiązań wynika rozwiązanie zasadniczego problemu. Naturalną konstrukcją programistyczną jest w tym wypadku rekursja. Przykładem zastosowania tej metody jest algorytm binarnego wyszukiwania elementu w liście uporządkowanych wartości. Porównanie danego elementu z elementem znajdującym się pośrodku ciągu sprawdza poszukiwanie do lewej lub prawej połówki zadanego ciągu.

### 1.8.2. Programowanie dynamiczne

Programowanie dynamiczne „poprawia” metodę „dziel i zwyciężaj” w sytuacji, kiedy wymaga ona wielokrotnego liczenia rozwiązań tych samych podproblemów. Oto funkcja rekurencyjna, licząca współczynnik dwumianowy  $\binom{n}{m}$ .

```
function wsp(n, m: integer) : integer;
{0 ≤ m ≤ n}
begin
  if ((n = m) or (m = 0)) then wsp := 1
  else wsp := wsp(n - 1, m) + wsp(n - 1, m - 1)
end
```

Wielokrotnie jest tu powtarzane rozwiązywanie tych samych podproblemów, na przykład

$$\begin{aligned} wsp(5, 3) &= wsp(4, 3) + wsp(4, 2) = wsp(3, 3) + wsp(3, 2) + wsp(3, 2) + wsp(3, 1) = \\ &= wsp(3, 3) + wsp(2, 2) + wsp(2, 1) + wsp(2, 2) + wsp(2, 1) + wsp(2, 0) = \\ &= wsp(3, 3) + wsp(2, 2) + wsp(1, 1) + wsp(1, 0) + wsp(2, 2) + wsp(1, 1) + wsp(1, 0) + \\ &+ wsp(1, 1) + wsp(1, 0) + wsp(2, 0) \end{aligned}$$

Algorytm w takiej postaci ma złożoność wykładniczą. Aby uniknąć powtarzania wyliczania tych samych wartości, możemy zacząć proces obliczeń od rozwiązywania najmniejszych podproblemów, zapisać rozwiązania w tablicy pomocniczej, a następnie użyć tych rozwiązań przy wyznaczaniu rozwiązań podproblemów większych rozmiarów – aż do otrzymania rozwiązania problemu wyjściowego. Aby na przykład obliczyć  $\binom{n}{m}$ , użyjemy tablicy pomocniczej  $pom[0..n]$ , licząc w  $i$ -tej fazie  $pom[j] = \binom{i}{j}$ , dla  $1 \leq i \leq n$ ,

```
for j := 0 to n do pom[j] := 1;
for i := 2 to n do {pom[j] =  $\binom{i-1}{j}$ , dla  $0 \leq j \leq i-1$ }
  for j := i - 1 downto 1 do pom[j] := pom[j - 1] + pom[j];
{pom[m] =  $\binom{n}{m}$ }
```

Otrzymany algorytm ma złożoność czasową  $O(n^2)$ .

### 1.8.3. Metoda zachłanna

Gdy możliwych kombinacji danych, które mogą być rozwiązaniami, jest liczba wykładnicza, rozpatrywanie danych w kolejności uporządkowanej prowadzi czasami do zadowalającego rozwiązania. W pewnych wypadkach prowadzi to do znalezienia pełnego rozwiązania, ale częściej do znalezienia rozwiązania przybliżonego (to jest nieoptymalnego).

Jako przykład rozważmy problem zapelnienia plecaka. Mamy danych  $n$  przedmiotów o nieujemnych rozmiarach – odpowiednio  $-x_1, x_2, \dots, x_n$  oraz plecak o pojemności  $c \geq 0$ . Naszym zadaniem jest wybrać pewną liczbę przedmiotów  $x_{l_1}, x_{l_2}, \dots, x_{l_k}$  tak, żeby  $x_{l_1} + x_{l_2} + \dots + x_{l_k} \leq c$  oraz żeby pozostałe w plecaku wolne miejsce  $c - (x_{l_1} + x_{l_2} + \dots + x_{l_k})$  było jak najmniejsze. W metodzie zachłannej pakujemy plecak, zaczynając od przedmiotów o najmniejszych rozmiarach. Znaleziony za pomocą tej metody wybór przedmiotów nie musi być optymalny, jak w wypadku plecaka o pojemności 3 i przedmiotach o rozmiarach – odpowiednio, 1, 1, 1, 5, 1, 5.

O innym sposobie stosowania metody zachłannej możemy się przekonać, rozwiązyując taki oto problem sortowania: dla danego ciągu wartości ze zbioru liniowo uporządkowanego

$a_1, a_2, \dots, a_n$  należy znaleźć permutację  $\sigma: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$  taką, że  $a_{\sigma(1)} \leq a_{\sigma(2)} \leq \dots \leq a_{\sigma(n)}$ . Zgodnie z metodą zachłanną znajdujemy najpierw taki indeks  $j$ , że  $a_j$  jest najmniejszym elementem w ciągu, i przyjmujemy  $\sigma(1) = j$ . Potem znajdujemy najmniejszy element z pozostałych wyrazów ciągu i otrzymujemy następny indeks  $\sigma(2)$ . Powtarzamy tę procedurę aż do otrzymania wszystkich indeksów ( $\sigma(3), \dots, \sigma(n)$ ). Problem sortowania jest tematem następnego rozdziału tej książki.

#### 1.8.4.

#### Inne metody

Wcześniej w tym rozdziale mówiliśmy już o jednej ważnej metodzie konstrukcji algorytmów: metodzie kolejnych transformacji (przy algorytmie przechodzenia grafu). Będziemy z niej też korzystać w następnych rozdziałach, ale przedstawimy również nowe metody. Warto przy okazji wspomnieć, że te same metody są stosowane przy układaniu algorytmów w różnych dziedzinach algorytmicznych. Choć następne rozdziały są pogrupowane tematycznie, powinieneś zwrócić uwagę na to, jak tych samych metod układania algorytmów i tych samych struktur danych (abstrakcyjnych typów danych) używa się w różnych sytuacjach.

### Zadania

1.1. Porównaj rzędy wielkości następujących funkcji, porządkując je od najniższych do najwyższych:

- (a)  $n^2 \log n + n\sqrt{n^2 + n^{3/2}}$
- (b)  $n^2 \log n$
- (c)  $n^{3/2} / \log n$
- (d)  $n^3 / \sqrt{n - \log n}$
- (e)  $n^{3/4} + \log n$

1.2. Dla jakich wartości  $n$

- (a)  $n^2 < 10n \log n$
- (b)  $2^n < 10n^2$

1.3. Sprawdź, czy jeżeli  $f(n) = O(g(n))$  oraz  $h(n) = O(q(n))$ , to

- (a)  $f(n) + h(n) = O(g(n) + q(n))$
- (b)  $f(n)*h(n) = O(g(n)*q(n))$

Czy podobne zależności zachodzą dla rzędów wielkości  $\Omega$  i  $\Theta$ ?

### Zadania

1.4. ([BK]) Niech  $T(n)$  będzie funkcją niemalejącą o argumentach i wartościach naturalnych, a  $f(x)$  funkcją niemalejącą o argumentach i wartościach rzeczywistych. Udowodnij, że jeśli są spełnione następujące dwa warunki:

(a)  $T(2^k) = \Theta(f(2^k))$

(b) istnieją stałe rzeczywiste  $x_0$  i  $c > 0$  takie, że dla każdego rzeczywistego  $x \geq x_0$ , mamy  $f(2x) \leq cf(x)$

to  $T(n) = \Theta(f(n))$ . Sprawdź, dla których z następujących funkcji zachodzi warunek (b):

(a)  $x$

(b)  $x \log x$

(c)  $x^2$

(d)  $2^x$

1.5. Rozwiąż (w sensie asymptotycznym) następujące równania rekurencyjne:

(a)  $T(n) = T(\lfloor n/2 \rfloor) + n$

(b)  $T(n) = 3T(\lceil n/2 \rceil) + n$

(c)  $T(n) = 2T(\lfloor n/2 \rfloor) + 1$

(d)  $T(n) = T(n-1) + \lfloor \log n \rfloor$

(e)  $T(n) = T(n-1) + n$

(f)  $T(n) = T(\lfloor n/4 \rfloor) + T(\lfloor 3n/4 \rfloor) + n$

(g)  $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor n/2 \rfloor) + n$

(h)  $T(n) = T(\lfloor \sqrt{n} \rfloor) + \lfloor \sqrt{n} \rfloor$

przy założeniu, że  $T(1) = 1$ . Czy wynik ulegnie zmianie, gdy zamiast stałej 1 weźmiemy dowolną stałą naturalną  $c$ ?

1.6. ([AHU]) Podaj rozwiązanie następującego równania rekurencyjnego:

$$T(n) = \begin{cases} b & \text{dla } n = 1 \\ aT(\lfloor n/c \rfloor) + bn & \text{dla } n > 1 \end{cases}$$

gdzie  $a, b, c$  są dodatnimi liczbami całkowitymi.

1.7. Podaj rozwiązanie następującego równania rekurencyjnego:

$$T(n) = \begin{cases} b & \text{dla } n = 1 \\ aT(\lfloor n/2 \rfloor) + b\lfloor \log n \rfloor & \text{dla } n > 1 \end{cases}$$

gdzie  $a, b$  są dodatnimi liczbami całkowitymi.

1.8. Podaj rozwiązańie następującego równania rekurencyjnego:

$$T(n) = \begin{cases} b & \text{dla } n = 1 \\ aT(\lfloor n/2 \rfloor) + \lfloor b\sqrt{n} \rfloor & \text{dla } n > 1 \end{cases}$$

gdzie  $a, b$  są dodatnimi liczbami całkowitymi.

1.9. Jak zmienią się wyniki zadań od 1.5 do 1.8, gdy zamiast równości będziemy rozpatrywać nierówność  $\leq$ ?

1.10. Zapisz algorytm wyszukiwania binarnego w ciągu uporządkowanym. Przy założeniu tego samego modelu probabilistycznego co w wypadku wyszukiwania sekwencyjnego wyznacz funkcje  $W(n)$ ,  $A(n)$ ,  $\Delta(n)$  i  $\delta(n)$ .

1.11. Stosując metodę „dziel i zwyciężaj”, ułóż algorytmy rozwiązuające następujące problemy:

- (a) sortowania;
- (b) wyznaczania dwóch największych elementów w ciągu;
- (c) wyznaczania największego i najmniejszego elementu w ciągu;
- (d) wyznaczania miejsca zerowego funkcji ciągiej  $f(x)$  w przedziale  $[a, b]$  z dokładnością  $\epsilon > 0$ ;
- (e) mnożenia dwóch liczb binarnych;
- (f) sprawdzania, czy istnieje takie  $i$ , że  $L[i] = i$  w uporządkowanym ciągu liczb całkowitych  $L[1] < L[2] < \dots < L[n]$ .

Wyznacz pesymistyczną złożoność czasową i pesymistyczną wrażliwość czasową.

1.12. Udowodnij, że  $\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$  dla każdej całkowitej dodatniej liczby  $n$ .

1.13. [BK] Niech  $x, a, b, p, q$  i  $r$  będą zmiennymi całkowitymi. Udowodnij, że podane instrukcje są poprawne względem warunku początkowego  $x \geq 0$  i warunku końcowego  $a^2 \leq x < (a+1)^2$ .

(a) **begin**  
 $a := 0;$   
**while**  $(a+1)*(a+1) \leq x$  **do**  $a := a + 1$   
**end;**

(b) **begin**  
 $a := 0; p := 1; x := 1;$   
**while**  $p \leq x$  **do**  
**begin**  
 $a := a + 1; x := x + 2; p := p + x$   
**end**  
**end;**

(c) **begin**  
 $a := 0; b := x + 1;$   
**while**  $a + 1 \neq b$  **do**  
**begin**  
 $p := (a + b) \text{div } 2;$   
**if**  $p*p > x$  **then**  $b := p$  **else**  $a := p$   
**end**  
**end;**

(d) **begin**  
 $q := 1;$   
**while**  $x \geq q*q$  **do**  $q := 2*q;$   
 $a := 0;$   
**while**  $q > 1$  **do**  
**begin**  
 $q := q \text{div } 2;$   
**if**  $(a + q)*(a + q) \leq x$  **then**  $a := a + q$   
**end**  
**end;**

(e) **begin**  
 $q := 1;$   
**while**  $x \geq q$  **do**  $q := 4*q;$   
 $r := x; a := 0;$   
**while**  $q > 1$  **do**  
**begin**  
 $q := q \text{div } 4;$   
 $a := a \text{div } 2;$   
**if**  $2*a + q \leq r$  **then**  
**begin**  
 $r := r - 2*a - q; a := a + q$   
**end**  
**end**  
**end;**

1.14. Rozważmy obliczenie iloczynu  $n$  macierzy  $M \times M_2 \times \dots \times M_n$ , gdzie  $M_i$  jest macierzą mającą  $r_{i-1}$  wierszy i  $r_i$  kolumn. Przyjmujemy, że mnożenie macierzy rozmiaru  $k \times l$  i  $l \times m$  kosztuje  $k \cdot l \cdot m$  jednostek. Korzystając z metody programowania dynamicznego, określ optymalne rozłożenie nawiasów w wyrażeniu  $M_1 \times M_2 \times \dots \times M_n$  minimalizujące sumaryczny koszt mnożenia wszystkich  $n$  macierzy.

1.15. Udowodnij, że pesymistyczna złożoność czasowa algorytmu NWD wynosi  $O(\log n)$ , gdzie  $n = \max(x, y)$ . (Wskazówka: Użyj liczb Fibonacciego).

- 1.16. Zaimportuj w Pascalu kolejkę podwójną za pomocą struktury dowiązanej tak, żeby koszt wykonania każdej operacji kolejki podwójnej był  $O(1)$ .
- 1.17. Podaj implementację listy, w której każdą operację kolejki podwójnej, złożenie dwóch list i odwrócenie listy można wykonać w czasie  $O(1)$ . Staraj się używać jak najmniej pamięci.
- 1.18. W pewnym programie ma zostać użyta tablica `a:array[1..n] of integer`. W wypadku jakiej struktury danych można nie dokonać początkowego zainicjowania wartości w tej tablicy, a jednocześnie odróżnić, czy wartość w tablicy została wstawiona tam przez instrukcję w programie, czy też jest to wartość przypadkowo znajdująca się w danym miejscu pamięci. Sprawdzenie powinno dać się wykonać w czasie  $O(1)$ . Można użyć tablic pomocniczych pod warunkiem, że się ich nie zainicjuje.
- 1.19. Dokonaj eliminacji rekursji w procedurze *b-traverse*.
- 1.20. Ułóż algorytm rekurencyjny, który w danym drzewie z korzeniem  $T$  wyznacza głębokość każdego wierzchołka. Wyeliminuj rekursję, używając standardowej metody ze stosem.
- 1.21. Ułóż algorytm rekurencyjny, który w danym drzewie z korzeniem  $T$  wyznacza wysokość każdego wierzchołka. Wyeliminuj rekursję, używając standardowej metody ze stosem.
- 1.22. Ułóż algorytmy rekurencyjne, które dla każdego wierzchołka  $v$  drzewa binarnego wyznaczają jego następnik w porządku:
- preorder,
  - inorder,
  - postorder,
- zapisując go na polu `next(v)`.
- 1.23. Napisz algorytm z procedurami rekurencyjnymi, wyznaczający dla danego drzewa binarnego  $T$  dwa wierzchołki  $x$  i  $y$ , między którymi odległość w drzewie jest największa.
- 1.24. Zapisz algorytm nonrec-inorder bez użycia instrukcji skoku.
- 1.25. Każde drzewo binarne można przejść bez korzystania z rekursji czy stosu (czyli z dodatkową pamięcią tylko  $O(1)$ ), jeśli jest dozwolona zmiana dowiązań drzewowych w trakcie wykonywania algorytmu (na koniec dowiązania mają być takie same jak na początku) oraz jeśli dopuszczać możliwość odwiedzania tego samego wierzchołka wielokrotnie. Jak tego dokonać? Ułóż algorytm.

- 1.26. Rozważmy problem przejścia drzewa binarnego metodą preorder przy założeniu, że w każdym wierzchołku  $v$  drzewa mamy pole `visited(v)` typu `boolean`, które może być użyte do zaznaczania faktu, że wierzchołek został już odwiedzony. Napisz procedurę przechodzenia takiego drzewa metodą preorder, używając  $O(1)$  dodatkowej pamięci (a więc bez użycia rekursji).
- 1.27. Zaprojektuj strukturę danych, umożliwiającą wykonywanie w czasie  $O(1)$  następujących operacji na stosie  $S$  o elementach ze zbioru liniowo uporządkowanego:
- `empty(S)`: sprawdzenie, czy stos  $S$  jest pusty;
  - `push(u, S)`: wstawienie elementu  $u$  na stos  $S$ ;
  - `pop(S)`: usunięcie z  $S$  wierzchołka stosu;
  - `findmin(S)`: wyznaczenie najmniejszego elementu znajdującego się na stosie  $S$ .
- 1.28. Podaj strukturę danych, umożliwiającą wykonywanie w czasie  $O(1)$  następujących operacji na początkowo pustym zbiorze  $S$ :
- `select(S)`: wybranie dowolnego elementu zbioru  $S$  i usunięcie go z  $S$ ;
  - `search(i, S)`: sprawdzenie, czy element  $i$  należy do  $S$ ;
  - `insert(i, S)`:  $S := S \cup \{i\}$
- przy założeniu, że  $S \subseteq \{1, 2, \dots, n\}$ .
- 1.29. Podaj strukturę danych, umożliwiającą wykonywanie w czasie  $O(1)$  następujących operacji na początkowo pustym ciągu  $q$ :
- `push(i, q)`: wstawienie elementu  $i$  na początek listy  $q$ ;
  - `pop(q)`: usunięcie elementu początkowego listy  $q$ ;
  - `search(i, q)`: sprawdzenie, czy element  $i$  znajduje się na liście  $q$ ;
  - `delete(i, q)`: usunięcie elementu  $i$  z listy  $q$
- przy założeniu, że  $q \subseteq \{1, 2, \dots, n\}$ .
- 1.30. Podaj strukturę danych, umożliwiającą wykonywanie w czasie  $O(1)$  następujących operacji na początkowo pustej liście  $q$ :
- `push(v, q)`: wstawienie elementu  $v$  na początek listy  $q$ ;
  - `pop(q)`: usunięcie elementu początkowego listy  $q$ ;
  - `uptomin(q)`: usunięcie z listy  $q$  elementu najmniejszego i wszystkich elementów wstawionych na listę  $q$  po elemencie najmniejszym.
- 1.31. **Licznikiem** nazywamy strukturę danych, umożliwiającą wykonywanie operacji `increment`, polegającej na dodawaniu jedynki do początkowo wyzerowanej wartości. Rozważmy reprezentację licznika za pomocą tablicy `var A: array[0..rA] of 0..1` oraz zmiennej  $N$  wskazującej indeks najbardziej znaczącej jedynki. Wartość

licznika ma być równa  $x = \sum_{i=0}^N A[i]2^i$ . Na początku  $A[i] = 0$  dla  $0 \leq i \leq rA$  oraz  $N = -1$ . Rozważmy następującą implementację operacji *increment*:

```

i := 0;
while i ≤ N do
  if A[i] = 1 then
    begin A[i] := 0; i := i + 1 end
  else break;
  if i ≤ rA then A[i] := 1 else error('Przepelenie licznika');
  if i > N then N := i;

```

Dobierz odpowiednio potencjał dla tej struktury danych i udowodnij, że koszt zamortyzowany operacji *increment* jest  $O(1)$ .

- 1.32. Do struktury danych z zadania 1.31 dodaj operację *reset* zerującą licznik. Przyjmij następującą jej implementację:

```
N := -1;
```

Przy odpowiednio dobranym potencjału struktury danych udowodnij, że koszt zamortyzowany każdej operacji jest  $O(1)$ .

- 1.33. Założymy, że implementujemy kolejkę za pomocą dwóch stosów, przy czym na jednym z nich umieszczamy elementy tak, jakbyśmy umieszczaли je na końcu kolejki, natomiast z drugiego stosu pobieramy elementy tak, jakbyśmy pobierali je z początku kolejki. Gdy drugi stos (służący do pobierania elementów) jest pusty, przepisujemy całą zawartość pierwszego stosu na drugi. Zapisz operacje kolejki w jej reprezentacji za pomocą dwóch stosów. Jaka jest pesymistyczna złożoność czasowa tych operacji? Zdefiniuj odpowiednio potencjał dla tej struktury danych tak, żeby koszt zamortyzowany każdej operacji kolejki był stały.

11

## 2 Sortowanie

W tym rozdziale przedstawiamy podstawowy problem informatyczny, a mianowicie sortowanie. Podajemy zarówno ogólne algorytmy sortowania, takie jak sortowanie przez selekcję (selectionsort), sortowanie przez wstawianie (insertionsort), sortowanie szybkie (quicksort), sortowanie przez kopcowanie (heapsort), sortowanie przez scalanie (mergesort), jak i algorytmy uzależnione od dziedziny, takie jak sortowanie pozycyjne (radixsort), a także algorytmy wykorzystujące pamięć zewnętrzna. Rozważamy problem, ile co najmniej trzeba wykonać porównań, żeby posortować listę  $n$  elementów. Rozpatrujemy też problemy zblżone do sortowania, jak scalanie i wyznaczanie  $k$ -tego co do wielkości elementu.

**Sortowanie** to problem bardzo często rozwiązywany na komputerach. Jego popularność wiąże się z faktem, że łatwiej jest korzystać ze zbiorów uporządkowanych niż nieuporządkowanych.

Sortowanie definiuje się następująco: dana jest lista  $q = [a_1, a_2, \dots, a_n]$  elementów zbioru liniowo uporządkowanego; trzeba dokonać permutacji ustalającej elementy w porządku niemalejącym  $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$ .

Spotyka się kilka modyfikacji problemu sortowania, które występują w praktyce:

- elementy są rekordami danych; na rekordach jest określona funkcja klucza  $key(a)$ ; uporządkować rekordy względem wartości ich klucza tak, żeby

$$key(a_{i_1}) \leq key(a_{i_2}) \leq \dots \leq key(a_{i_n})$$

- elementy są parami  $[k_i, p_i]$ , gdzie  $k_i$  jest kluczem, a  $p_i$  jest dowiązaniem do rekordu dla  $1 \leq i \leq n$ ; uporządkować elementy względem ich kluczy jak w wypadku pierwszej modyfikacji (unikamy przedstawiania rekordów, które mogą być długie);
- elementy do posortowania mogą znajdować się albo w pamięci wewnętrznej (wówczas mamy do nich dostęp bezpośredni), albo w pamięci zewnętrznej;

- czasami wymaga się dodatkowo zachowania warunku stabilności, tzn. zachowania początkowego ustawienia względem siebie elementów równych (rekordów o takich samych kluczach); gdy na przykład alfabetyczną listę studentów sortujemy względem wyników egzaminu, silą rzeczy wymagamy, aby w wypadku tej samej oceny nazwiska studentów były podawane w porządku alfabetycznym.

Za operację dominującą będziemy przyjmować porównania elementów w ciągu. Za złożoność pamięciową  $S(n)$  będziemy przyjmować ilość dodatkowej pamięci (oprócz  $n$  miejsc pamięci dla elementów w ciągu), potrzebnej do wykonania algorytmu.

Będziemy też zakładać, że elementy listy  $q$  są liczbami całkowitymi i że znajdują się w tablicy  $a$ ,  $a[i] = a_i$  dla  $1 \leq i \leq n$ . Będziemy niekiedy przyjmować, że  $a[0] = -\infty$ ,  $a[n+1] = +\infty$ , gdzie  $-\infty$ ,  $+\infty$  to liczby – odpowiednio – najmniejsza i największa w ustalonej reprezentacji liczb całkowitych. Przy analizie probabilistycznej (wielkości  $A(n)$  i  $\delta(n)$ ) będziemy przyjmować, że danymi wejściowymi są permutacje liczb  $1, 2, \dots, n$  oraz że każda taka permutacja jest jednakowo prawdopodobna.

W kolejnych podrozdziałach rozważymy podstawowe algorytmy sortujące.

## 2.1.

### Selectionsort – sortowanie przez selekcję

Sortowanie przez selekcję odbywa się w następujący sposób: trzeba wyznaczyć najmniejszy element w ciągu; zamienić go miejscami z pierwszym elementem w ciągu, wyznaczyć najmniejszy element w  $a[2..n]$  i zamienić go z drugim elementem w ciągu itd., aż cała tablica zostanie posortowana.

```
procedure selectionsort;
var i, j, min : integer;
begin
  for i := 1 to n - 1 do
    begin {a[1] ≤ ... ≤ a[i - 1] ≤ a[i ... n]}
      min := i;
      for j := i + 1 to n do
        if a[j] < a[min] then min := j;
      a[min] <-> a[i]; {zamiana miejscami a[min] z a[i]}
    end
  end selectionsort;
```

Analiza złożoności algorytmu selectionsort jest bezpośrednią.

$$W(n) = A(n) = (n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2} = \frac{1}{2}n^2 - O(n)$$

## 2.2. Insertionsort – sortowanie przez wstawianie

$\Delta(n) = \delta(n) = 0$  (w procedurze jest wykonywany zawsze ten sam ciąg operacji, niezależnie od danych wejściowych)

$S(n) = O(1)$  (mówimy, że algorytm sortuje w miejscu)

Do głównych zalet przedstawionego algorytmu należą:

- optymalność, jeśli chodzi o liczbę przestawień (tylko  $n - 1$ );
- prostota implementacji;
- zadowalająca szybkość dla małych wartości  $n$ .

Algorytm selectionsort nie jest stabilny. Znalezienie kontrprzykładowo pozostawiamy Ci, Drogi Czytelniku, jako ćwiczenie (zad. 2.2). Kosztem dodatkowego wysiłku, zwiększa- jąc jednak współczynnik proporcjonalności złożoności, można ten algorytm uczynić sta- bilnym (zad. 2.3).

## 2.2.

### Insertionsort – sortowanie przez wstawianie

Sortowanie przez wstawianie odbywa się w następujący sposób: dla każdego  $i = 2, 3, \dots, n$  trzeba powtarzać wstawianie  $a[i]$  w już uporządkowaną część listy  $a[1] \leq \dots \leq a[i-1]$ .

```
procedure insertionsort;
{a[0] = -∞, aby uniknąć testu "j > 1"}
var i, j, v : integer;
begin
  for i := 2 to n do
    begin {a[0] ≤ a[1] ≤ ... ≤ a[i - 1]}
      j := i; v := a[i];
      while a[j - 1] > v do
        begin {a[0] ≤ ... ≤ a[j - 1] ≤ a[j + 1] ≤ ... ≤ a[i]}
          j < i ⇒ v < a[j + 1], a[j] – wolne miejsce}
          a[j] := a[j - 1]; j := j - 1
        end;
      a[j] := v
    end
  end insertionsort;
```

Analiza pesymistycznej złożoności czasowej jest bezpośrednią:

$$W(n) = 2 + 3 + \dots + n = \frac{n(n + 1)}{2} - 1 = \frac{1}{2}n^2 + O(n)$$

(Liczymy również porównania z elementem  $a[0]$ ).

$$\Delta(n) = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{1}{2}n^2 + O(n)$$

$$S(n) = O(1) \text{ (algorytm działa w miejscu)}$$

Zauważmy, że liczba porównań wykonywanych w algorytmie insertionsort jest proporcjonalna do liczby inwersji na liście  $q$ , tzn. takich par, że  $a[i] > a[j]$  dla  $i < j$ . Jeśli ciąg  $q$  jest prawie uporządkowany, tzn. gdy liczba inwersji jest  $O(n)$ , to złożoność pesymistyczna tego algorytmu jest liniowa. Jest to duża jego zaleta, gdyż w praktyce sortowane dane są już często częściowo uporządkowane. Oprócz tego algorytm ten jest stabilny, prosty i łatwo implementowalny.

Przeprowadzimy teraz analizę oczekiwanej złożoności czasowej. Niech  $X_n$  będzie liczba porównań wykonywanych w algorytmie dla  $n$ -elementowej permutacji liczb  $1, 2, \dots, n$  przy założeniu, że każda permutacja jest jednakowo prawdopodobna. Element  $a[i]$  z równym prawdopodobieństwem może zająć każdą z  $i$  pozycji na liście

$$-\infty = a[0] < a[1] < a[2] < \dots < a[i-1]$$

(zad. 2.5). W  $i$ -tym kroku algorytmu wykonuje się więc średnio taka oto liczba porównań:

$$\frac{1}{i} \sum_{j=1}^i j = \frac{1}{i} \frac{(i+1)i}{2} = \frac{i+1}{2}$$

(Liczymy również porównania z elementem  $a[0]$ ).

Sumując po wszystkich  $n-1$  iteracjach, otrzymujemy

$$A(n) = \sum_{i=2}^n \frac{i+1}{2} = \frac{1}{2} \sum_{p=3}^{n+1} p = \frac{1}{2} \frac{(n+4)(n-1)}{2} = \frac{1}{4}n^2 + O(n)$$

Oczekiwana złożoność czasowa algorytmu insertionsort jest więc nieco lepsza niż algorytm selectionsort, choć tego samego rzędu wielkości.

Wyznaczenie oczekiwanej wrażliwości czasowej

$$\delta(n) = \frac{1}{6}n^3 + O(n)$$

pozostawiamy Tobie, Drogi Czytelniku, jako ćwiczenie (zad. 2.4).

### 2.3.

## Quicksort – sortowanie szybkie

Algorytm quicksort jest jednym z najczęściej używanych algorytmów sortowania. Jest uważany za najszybszy algorytm sortowania dla „losowych” danych wejściowych. Jego konstrukcja jest oparta na zasadzie „dziel i zwycięzaj”. Lista wejściowa  $q$  zostaje w specjalny sposób podzielona na dwie części, po czym obie części są sortowane niezależnie. Oto ogólny schemat tego algorytmu.

```
procedure quicksort(l, r : integer);
{parametry l, r określają podciąg do posortowania przez dane
wywołanie procedury quicksort; w wyniku działania instrukcji
  if n > 1 then quicksort(l, n);
  zostaje posortowana cała lista wejściowa}
var j : integer;
begin {l < r}
  j := partition(l, r); {podział}
  if j - 1 > l then quicksort(l, j - 1);
  if r > j + 1 then quicksort(j + 1, r)
end quicksort;
```

Decydujące znaczenie dla poprawności i efektywności algorytmu ma funkcja *partition*, która przekształca tablicę  $a[l..r]$  w ten sposób, że:

- (a) element  $v = a[j]$  znajduje się na właściwym, ostatecznym miejscu w tablicy;
- (b)  $a[l], \dots, a[j-1] \leq v$ ;
- (c)  $v \leq a[j+1], \dots, a[r]$ .

Jako element  $v$ , rozdzielający ciąg  $a[l..r]$ , wybieramy  $v = a[l]$ . Przeglądamy ciąg  $a[l+1], \dots, a[r]$  od lewej strony, dopóki nie znajdziemy elementu nie mniejszego niż  $a[l]$ . Potem przeglądamy  $a[l+1], \dots, a[r]$ , tym razem od strony prawej, dopóki nie napotkamy elementu nie większego niż  $a[l]$ . Dwa elementy, na których się zatrzymujemy, zamieniamy miejscami. Powtarzając opisane działania, zapewniamy, że elementy na lewo od wskaźnika wędrującego ze strony lewej na prawą są nie większe niż  $a[l]$ , a elementy na prawo od wskaźnika wędrującego ze strony prawej na lewą są nie mniejsze niż  $a[l]$ . Kiedy oba wskaźniki się spotykają, proces dzielenia jest zakończony – wystarczy element rozdzielający  $v = a[l]$  zamienić miejscami z ostatnim elementem lewej części.

Oto pełna postać procedury *quicksort*.

```
procedure quicksort(l, r : integer);
{parametry l, r określają podciąg do posortowania przez dane wywoła-
nie procedury quicksort; w wyniku działania instrukcji
  if n > 1 then quicksort(l, n);
  zostaje posortowana cała lista wejściowa przy założeniu, że a[n+1]=+∞}
```

```

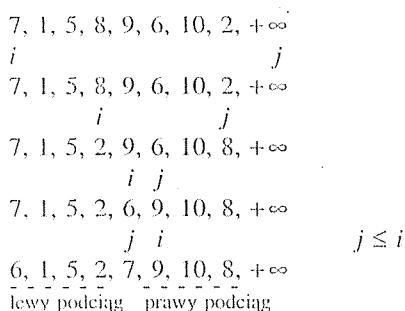
var v, i, j : integer;
begin
  {początek funkcji partition}
  { $l < r$ ,  $a[l], a[l+1], \dots, a[r] \leq a[r+1]$ }
  v := a[1]; i := 1; j := r + 1;
  repeat
    repeat i := i + 1 until a[i]  $\geq v$ ;
    repeat j := j - 1 until a[j]  $\leq v$ ;
    if i < j then a[i]  $\leftarrow a[j]$ ; {zamiana miejscami a[i] z a[j]}
  until j  $\leq i$ ;
  a[1] := a[j]; a[j] := v;
  {koniec procedury partition}
  if j - 1 > l then quicksort(l, j - 1);
  if r > j + 1 then quicksort(j + 1, r)
end quicksort;

```

(Warto zwrócić uwagę, że w wewnętrznych instrukcjach `repeat` istotne są nierówności nieostre, zobacz zad. 2.10).

### PRZYKŁAD:

Weźmy ciąg 7, 1, 5, 8, 9, 6, 10, 2 i zbadajmy działanie funkcji *partition*. Elementem dzielącym jest  $v = 7$ .



Policzmy liczbę porównań dla wywołania *quicksort*( $l, r$ ). Element dzielący  $v = a[l]$  jest porównywany z każdym elementem w ciągu  $a[l+1], \dots, a[r]$ , przy czym co najwyżej z dwoma elementami dwukrotnie, a zatem liczba porównań wynosi  $r - l + 2$ . Równanie na liczbę porównań w przypadku pesymistycznym jest więc następujące:

$$W(0) = W(1) = 0$$

$$W(n) = \max_{1 \leq j \leq n} (W(j-1) + W(n-j)) + (n+1) \text{ dla } n > 1 \quad (*)$$

### 2.3. Quicksort – sortowanie szybkie

Zagnieżdżen rekursji może być w najgorszym wypadku  $n - 1$ . Ponieważ na każdym poziomie rekursji wykonuje się co najwyżej  $n + 1$  porównań,

$$W(n) \leq (n-1)(n+1) = n^2 + O(1)$$

W poprzednich algorytmach współczynnik przy  $n^2$  był równy  $\frac{1}{2}$ . Możemy zatem podejrzewać, że otrzymane oszacowanie nie jest dość precyzyjne. Spróbujmy zgadnąć dokładniejsze rozwiązanie równań (\*), po czym przeprowadzimy dowód indukcyjny.

Możemy oczekiwać, że z największą wartością będziemy mieć do czynienia w sytuacji skrajnej (funkcja  $n^2$  jest wypukła, a zatem  $x^2 + y^2 \leq (x+y)^2 + 0^2$ ), gdy jedna z wartości  $j - 1$  albo  $n - j$  jest zawsze równa 0. Zachodzi to wtedy, kiedy algorytm quicksort jest używany do posortowania ciągu już posortowanego  $a[1] < a[2] < \dots < a[n]$ . Otrzymujemy następujące równanie na liczbę porównań:

$$\begin{cases} W_p(0) = W_p(1) = 0 \\ W_p(n) = W_p(n-1) + (n+1) \quad \text{dla } n > 1 \end{cases}$$

Stosując metodę rozwijania, mamy

$$W_p(n) = (n+1) + n + \dots + 3 + W_p(1) = \frac{(n+1)(n+2)}{2} - 3 = \frac{1}{2}n^2 + \frac{3}{2}n - 2$$

Pozostaje teraz wykazać za pomocą indukcji matematycznej, że dla każdego  $n \geq 0$

$$W(n) = \begin{cases} 0 & \text{dla } n = 0 \\ \frac{1}{2}n^2 + \frac{3}{2}n - 2 & \text{dla } n > 0 \end{cases} \quad (**)$$

Dla  $n = 0, 1$  jest to oczywiste. Założymy, że (\*\*) zachodzi dla wartości mniejszych od pewnego  $n$ . Wówczas wykorzystując wypukłość funkcji  $n^2$ , otrzymujemy

$$\begin{aligned} W(n) &= \max_{1 \leq j \leq n} (W(j-1) + W(n-j)) + (n+1) = \\ &= \max_{2 \leq j \leq n-1} (W(n-1), W(j-1) + W(n-j)) + (n+1) = \\ &= \max \left\{ \frac{1}{2}(n-1)^2 + \frac{3}{2}(n-1) - 2, \max_{2 \leq j \leq n-1} \left( \frac{1}{2}(j-1)^2 + \frac{3}{2}(j-1) - 2 + \right. \right. \\ &\quad \left. \left. + \frac{1}{2}(n-j)^2 + \frac{3}{2}(n-j) - 2 \right) + (n+1) = \right. \\ &= \max \left( \frac{1}{2}(n-1)^2 + \frac{3}{2}(n-1) - 2, \frac{1}{2}(n-2)^2 + \frac{3}{2}(n-2) - 2 \right) + (n+1) = \\ &= \frac{1}{2}(n-1)^2 + \frac{3}{2}(n-1) - 2 + n+1 = \frac{1}{2}n^2 + \frac{3}{2}n - 2 \end{aligned}$$

Z najmniejszą liczbą porównań mamy do czynienia w skrajnie przeciwej sytuacji, gdy dla każdego rekurencyjnego wywołania  $j$  znajduje się w środku przedziału  $[l..r]$ . Otrzymujemy wówczas

$$\begin{cases} W_o(0) = W_o(1) = 0 \\ W_o(n) = W_o(\lfloor (n-1)/2 \rfloor) + W_o(\lceil (n-1)/2 \rceil) + (n+1) \quad \text{dla } n > 1 \end{cases}$$

Rozwiązańiem tego równania jest funkcja  $W_o(n) = \Theta(n \log n)$ .

Pesymistyczna wrażliwość czasowa jest zatem równa

$$\Delta(n) = O(n^2 - n \log n) = O(n^2)$$

Z przeprowadzonej analizy pesymistycznej złożoności czasowej jeszcze nie widać, dla czego algorytm quicksort jest nazywany algorymem sortowania szybkiego. Okazuje się, że jego oczekiwana złożoność czasowa jest dobra, bliska funkcji  $W_o(n)$ .

Niech  $X_n$  będzie liczbą porównań, które są wykonywane w algorytmie quicksort w celu uporządkowania permutacji liczb  $1, 2, \dots, n$ , przy założeniu, że każda permutacja jest jednakowo prawdopodobna.

### Lemat 2.1.

Podciągi powstające w wyniku podziału losowej permutacji są ciągami losowymi.

Udowodnienie tego lematu pozostawiamy Ci jako zadanie 2.10.

Najpierw policzmy oczekiwana liczbę porównań przy założeniu, że elementem dzielącym jest  $s \in [1..n]$ . Wynosi ona  $(n+1) +$  oczekiwana liczba porównań dla  $quicksort(1, s-1) +$  oczekiwana liczba porównań dla  $quicksort(s+1, n)$ , co na mocy lematu 2.1 jest równe  $(n+1) + A(s-1) + A(n-s)$ .

Ponieważ każde  $s \in [1..n]$  jest jednakowo prawdopodobne (pojawia się więc z prawdopodobieństwem  $1/n$ ), oczekiwana złożoność czasowa spełnia następujące równanie:

$$\begin{cases} A(0) = A(1) = 0 \\ A(n) = (n+1) + \frac{1}{n} \sum_{s=1}^n (A(s-1) + A(n-s)) \quad \text{dla } n > 1 \end{cases}$$

Równanie na  $A(n)$ , dla  $n > 1$ , daje się sprowadzić do

$$A(n) = \frac{2}{n} \sum_{s=1}^n A(s-1) + (n+1)$$

Aby dokonać kolejnego przekształcenia, zapiszmy je w postaci dwóch równań:

$$nA(n) = 2 \sum_{s=1}^n A(s-1) + n(n+1)$$

$$(n-1)A(n-1) = 2 \sum_{s=1}^{n-1} A(s-1) + (n-1)n$$

Odejmując te równania stronami, otrzymujemy

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2n$$

a stąd

$$nA(n) = (n+1)A(n-1) + 2n$$

czyli

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}$$

Stosując do tego równania metodę rozwijania, mamy

$$\begin{aligned} \frac{A(n)}{n+1} &= \frac{A(n-1)}{n} + \frac{2}{n+1} = \\ &= \frac{A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \frac{A(1)}{2} + \frac{2}{3} + \frac{2}{4} + \dots + \frac{2}{n+1} = \\ &= 2 \left( H_{n+1} - \frac{3}{2} \right), \quad \text{gdzie } H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \end{aligned}$$

a zatem

$$A(n) = 2(n+1) \left( H_{n+1} - \frac{3}{2} \right) = 2(n+1) \left( H_n + \frac{1}{n+1} - \frac{3}{2} \right)$$

Na koniec korzystając ze związku asymptotycznego na  $H_n$

$$H_n = \ln n + \gamma + O(n^{-1}) \quad (\gamma \text{ jest stałą Eulera } \approx 0,57)$$

otrzymujemy

$$A(n) = \frac{2}{\log e} (n+1) \log n + O(n) = \frac{2}{\log e} n \log n + O(n)$$

W przybliżeniu współczynnik przy  $n \log n$  jest równy 1,4.

Jako zadanie (zad. 2.7) pozostawiamy Ci wykazanie, że oczekiwana wrażliwość czasowa algorytmu quicksort wynosi

$$\delta(n) = \sqrt{7 - \frac{2}{3}\pi^2 n} + O(\log n)$$

Współczynnik przy  $n$  jest w przybliżeniu równy 0,68.

Jako zadanie (zad. 2.8) pozostawiamy Ci też wykazanie, że oczekiwana liczba przestawień wykonywanych w algorytmie jest nie większa niż połowa oczekiwanej liczby porównań.

Zauważmy, że:

- (a) asymptotyczna oczekiwana złożoność czasowa algorytmu quicksort wynosi  $n \log n$ , a więc mniej niż dla poprzednich algorytmów;
- (b) współczynnik przy  $n \log n$  jest niewielki, około 1,4;
- (c) oczekiwana wrażliwość czasowa jest niewielka w stosunku do oczekiwanej złożoności czasowej, co mówi o silnym skupieniu rzeczywistej liczby porównań wokół wartości oczekiwanej;
- (d) liczba innych działań w algorytmie nie jest znacząco większa od liczby porównań.

Powyższe fakty tłumaczą, dlaczego algorytm quicksort jest uważany za najszybszy algorytm sortowania.

Poważnymi jednak jego wadami są niestabilność i koszt  $O(n^2)$  w przypadku pesymistycznym. Pewną wadą jest też użycie rekursji, która jest tłumaczona na operacje na stosie. W złożoności pamięciowej musimy uwzględnić pamięć potrzebną na implementację stosu. Jeśli chodzi o przypadek pesymistyczny, głębokość rekursji wynosi  $n - 1$ , a zatem

$$S(n) = O(n)$$

czyli złożoność pamięciowa jest większa niż w poprzednich algorytmach.

Na sześćce można łatwo ją zmniejszyć. Zauważmy, że:

- (a) oba wywołania rekurencyjne w treści procedury quicksort są niezależne, tzn. mogą być wykonywane w dowolnej kolejności;
- (b) wywołania te znajdują się na końcu treści procedury, a zatem:
  - jedno z nich może być zastąpione iteracją;
  - zamiast na stosie przechowywać miejsce powrotu z wywołania rekurencyjnego, można przechowywać parametry wywołania, które ma być wykonane po właściwie wykonywanym.

Na stosie będą się teraz znajdować pary indeksów  $[i, j]$  określające parametry wywołań algorytmu quicksort, które należy jeszcze wykonać. Przy realizacji wywołania  $\text{quicksort}(l, r)$  z dwóch wywołań rekurencyjnych będziemy umieszczać na stosie większy z podciągów otrzymanych w wyniku podziału (aścielj jego końca), pozostawiając mniejszy do przetwa-

rzania w kolejnej iteracji. Nazwijmy te podciągi bratnimi. Rozważmy pewien moment obliczeń i niech  $s_1, s_2, \dots, s_k$  będą długościami podciągów na stosie, a  $i_1, i_2, \dots, i_k$  rozmiarami ich bratnich podciągów. Przyjmijmy  $i_o = n$ . Zauważmy, że podciąg o długości  $i_j$ , gdzie  $j < k$ , został podzielony na dwa podciągi: jeden o długości  $s_{j+1}$ , umieszczony na stosie, i drugi o długości  $i_{j+1}$ , przetwarzany w kolejnej iteracji  $i_j = s_{j+1} + i_{j+1} + 1$  oraz  $s_{j+1} \geq i_{j+1}$ . Zachodzi wówczas:

- (a)  $i_{j+1} \leq \frac{1}{2}i_j$ ;
- (b)  $s_{j+1} \leq i_j$ .

Stosując indukcję, łatwo możemy teraz pokazać, że

$$2 \leq s_j \leq i_{j-1} \leq \frac{1}{2^{j-1}} n \quad \text{dla } 1 \leq j \leq k$$

co dowodzi, że maksymalna głębokość stosu jest ograniczona przez  $\log n$ . Mamy zatem

$$S(n) = O(\log n)$$

Oto zmieniona wersja algorytmu quicksort.

```
procedure quicksort1;
  var l, r, j : integer;
  s : stack;
begin
  if n > 1 then
    begin
      s := [[1, n]];
      repeat
        [l, r] := top(s); pop(s);
        while l < r do
          begin
            j := partition(l, r);
            if j - 1 < r - j then
              begin
                push(s, [j + 1, r]);
                r := j - 1;
              end
            else
              begin
                push(s, [l, j - 1]);
                l := j + 1
              end
          end
      until s = []
    end
  end quicksort1;
```

Algorytm quicksort był obiektem intensywnych badań, w wyniku czego zaproponowano kilka jego modyfikacji.

- Algorytm insertionsort może być używany do sortowania danych o małych rozmiarach, na przykład  $\leq 20$ .
- Gdy nie ma pewności, że ciąg wejściowy jest losowy, to lepiej albo używać generatora liczb losowych do określenia elementu dzielącego

```
u := random(1, x); {generowanie losowego indeksu z {1, x]}
a[1] <-> a[u];
```

albo jako element dzielący wybierać medianę z pewnej próbki elementów ciągu, na przykład pierwszego, środkowego i ostatniego.

- Można w ogóle pozbyć się stosu, organizując go wewnętrznie w sortowanym ciągu. Zmniejsza się w ten sposób złożoność pamięciową do  $O(1)$ , ale zwiększa się współczynnik proporcjonalności złożoności czasowej. Przy prezentacji tej metody przyjmiemy następujące założenie upraszczające algorytm: elementy listy wejściowej  $q = [a_1, \dots, a_n]$  są różnymi liczbami całkowitymi.

Zapiszmy jeszcze raz algorytm quicksort z rekurencją zastąpioną przez stos, przy założeniu, że przy podziale zawsze umieszczamy na stosie końca prawego podcięgu niezależnie od rozmiarów obu podcięgów.

```
procedure quicksort2;
  var l, r, j : integer;
  s : stack;
begin
  0:   s := [[1, n]];
  repeat {m ≥ 2}
    1:   [l, r] := top(s); pop(s);
    while r - l > m do
      begin
        niech a[k] będzie medianą z a[1], a[lfloor(l + r) / 2], a[x];
        a[k] <-> a[1]; j := partition(l, r);
        push(s, [j + 1, x]); {j + 1 ≤ x}
        r := j - 1
      end;
      wykonaj insertionsort dla a[l..r];
    3:   until s = []
  end quicksort2;
```

Etykietami zostały oznaczone instrukcje operowania na stosie. W każdej chwili realizacji algorytmu lista  $q$  jest podzielona na części przedstawione na rysunku 2.1.



Rys. 2.1. Stan w trakcie realizacji algorytmu quicksort

Zauważmy, że można wyeliminować lewy indeks z pary umieszczonej na stosie.

```
procedure quicksort3;
  var l, r, j : integer;
  s : stack;
begin
  0:   s := [n];
  r := -1;
  repeat {m ≥ 2}
    1:   l := r + 2;
    2:   r := top(s); pop(s);
    while r - l > m do
      begin
        niech a[k] będzie medianą z a[1], a[lfloor(l + r) / 2], a[x];
        a[k] <-> a[1]; j := partition(l, r);
        push(s, r);
        r := j - 1
      end;
      wykonaj insertionsort dla a[l..x];
    6:   until s = []
  end quicksort3;
```

Poprawność instrukcji przypisania  $l := r + 2$  wynika z rysunku 2.2:



Następny podcięg, dla którego indeks prawego końca bierzemy ze stosu; indeks lewego końca obliczamy, mając wartość bieżącego  $r$

Rys. 2.2. Związek między indeksami dwóch kolejnych instancji

Indeks prawego końca podcięgu do przetworzenia bierzemy ze stosu. Gdybyśmy potrafili zaznaczyć prawy koniec podcięgu w samym ciągu, stos nie byłby potrzebny (zaczynając od  $l = r + 2$ , szukalibyśmy pierwszego zaznaczonego miejsca).

Załóżmy, że ciąg do posortowania składa się wyłącznie z dodatnich liczb całkowitych. Indeksy prawych końców podciągów znaczymy, zmieniając znak liczby na ujemny.

```

procedure quicksort4;
  var l, r, j : integer;
  begin
    r := -1; a[n] := -a[n];
    repeat { $m \geq 2$ }
      l := r + 2; r := l;
      while a[r] > 0 do r := r + 1;
      a[r] := -a[r];
      while r - l > m do
        begin
          niech a[k] będzie medianą z a[l], a[ $\lfloor \frac{l+r}{2} \rfloor$ ], a[r];
          a[k] <-> a[l];
          j := partition(l, r);
          {*}
          a[r] := -a[r];
          r := j - 1;
        end;
        wykonaj insertionsort dla a[l..r];
      until r = n;
    end quicksort4;
  
```

Markowanie prawego końca można też wykonać przy użyciu samych elementów w ciągu (bez ich zmiany). Wystarczy w wierszu oznaczonym gwiazdką {\*} wstawić w miejscu  $a[r]$  największy element, powiedzmy  $a[k]$ , z podciągu  $a[l..j-1]$ , przesyłając jednocześnie  $a[r]$  w miejsce  $a[j]$ ,  $a[j]$  w miejsce  $a[j-1]$ , a  $a[j-1]$  w miejsce  $a[k]$ . Szczegóły pozostawiamy Ci jako zadanie 2.12.

## 2.4.

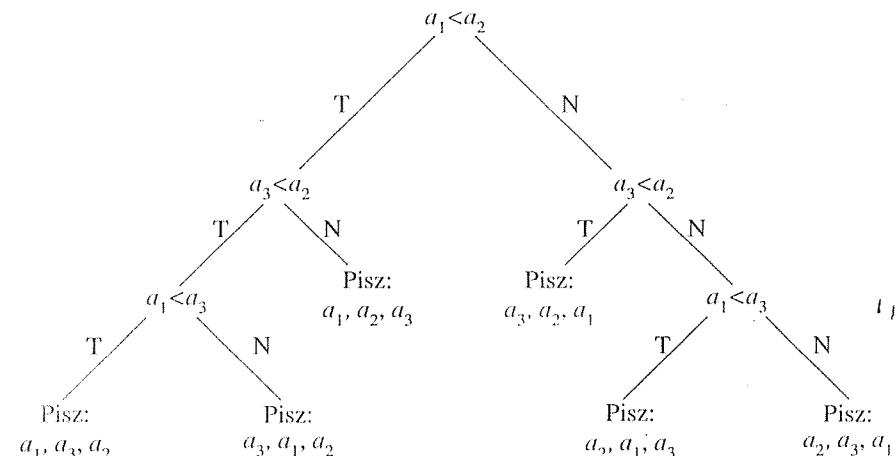
### Dolne ograniczenie na złożoność problemu sortowania

Zanim rozważymy następne algorytmy sortowania, zastanówmy się, jaka jest minimalna liczba porównań potrzebnych do posortowania ciągu  $n$  różnych elementów  $q = [a_1, a_2, \dots, a_n]$ . Za model obliczeń przyjmiemy model drzew decyzyjnych.

Drzewem sortującym listę  $q = [a_1, a_2, \dots, a_n]$  nazywamy drzewo decyzyjne, w którego węzłach wewnętrznych znajdują się porównania elementów listy, natomiast w liściach – instrukcje wypisujące ciąg w postaci uporządkowanej.

#### □ PRZYKŁAD:

Na rysunku 2.3 jest widoczne drzewo sortujące 3-elementowy ciąg  $q = [a_1, a_2, a_3]$ .



Rys. 2.3. Drzewo sortujące ciąg 3-elementowy

Każdy algorytm sortowania przez porównania (na przykład selectionsort, insertionsort, quicksort) daje się rozwinąć w drzewo sortujące przy ustalonym rozmiarze  $n$ .

Bez straty ogólności możemy założyć, że:

- jest dokładnie  $n!$  liści, tzn. tyle, ile różnych permutacji ciągu  $a_1, a_2, \dots, a_n$ ;
- każdy węzeł wewnętrzny ma dwa następcy tzn. każde porównanie jest istotne (obliczenie może dotyczyć pojścia w lewo bądź w prawo od danego węzła wewnętrznego).

Klasę drzew binarnych spełniających podane warunki oznaczamy przez  $B_n$ .

Zauważmy, że długość ścieżki od korzenia do liścia jest równa liczbie porównań wykonywanych w ramach odpowiedniego obliczenia algorytmu.

Znalezienie dolnych ograniczeń na liczbę porównań w przypadku pesymistycznym i przypadku oczekiwany sprowadza się zatem do oszacowania z dołu:

- $h(T)$  – wysokość drzewa  $T$ ;
- $d(T) = \sum_{x \text{ liść w } T} d_T(x)$  – całkowitej długości ścieżek zewnętrznych w  $T$ ;

w klasie drzew binarnych  $B_n$ . Wówczas dla każdego algorytmu sortującego przez porównania, który rozwija się w drzewo sortujące, zachodzą nierówności:

$$W(n) \geq \min_{T \in B_n} h(T)$$

$$A(n) \geq \min_{T \in B_n} \frac{d(T)}{n!}$$

□ **PRZYKŁAD:**

Dla drzewa binarnego z poprzedniego przykładu:

$$h(T) = 3$$

$$d(T) = 3 + 3 + 2 + 2 + 3 + 3 = 16$$

a zatem oczekiwana długość ścieżki zewnętrznej  $d(T)/3! = 16/6 = 2,66$ .

Zauważmy, że:

(a)  $2^{h(T)} \geq n!$  dla  $T \in B_n$  (dowód indukcyjny, że dla każdego węzła o wysokości  $h$  i liczbie potomków-liści  $l$ ,  $2^h \geq l$ );

(b)  $h(T) \geq \log n! = \sum_{i=1}^n \log i \geq \int_1^n \log x \, dx = \log e \int_1^n \ln x \, dx = \log e(x \ln x - x) \Big|_1^n = \log e(n \ln n - n + 1) = n \log n - n \log e + \log e \geq n \log n - 1,45n$ . Stąd otrzymujemy

**Twierdzenie 2.1.**

W każdym algorytmie sortującym przez porównania, w przypadku pesymistycznym, jest wykonywanych co najmniej  $n \log n - 1,45n$  porównań.

**Lemata 2.2.**

Wartość  $d(T)$  jest najmniejsza dla tych drzew  $T \in B_n$ , w których wszystkie liście znajdują się na dwóch sąsiednich poziomach.

**Dowód:** Niech drzewo  $T \in B_n$  o wysokości  $d$  nie spełnia warunku w lematce. Niech  $z$  i  $t$  będą liśćmi na poziomie  $d$ , a  $x$  liściem na poziomie  $k < d - 1$ . Dokonajmy przekształcenia drzewa  $T$  w drzewo  $T'$  przez doczepienie liści  $z$  i  $t$  do węzła  $x$ . Otrzymujemy drzewo należące do  $B_n$  i takie, że

$$\begin{aligned} d(T) - d(T') &= (2d + k) - ((d - 1) + 2(k + 1)) = \\ &\quad \text{usuwamy dwa liście} \\ &\quad \text{o głębokości } d \text{ i jeden} \\ &\quad \text{liść o głębokości } k \\ &= d - k - 1 = (d - 1) - k > 0. \end{aligned}$$

2.4. Dolne ograniczenie na złożoność problemu sortowania

Jak widać, przy przejściu od  $T$  do  $T'$  całkowita długość ścieżek zewnętrznych zmniejsza się, a zatem  $d(T)$  nie mogło być minimalne.

cbdo

**Lemat 2.3.**

Niech  $l = n!$ . Dla drzew  $T \in B_n$  zachodzi

$$d(T) \geq l \lfloor \log l \rfloor + 2(l - 2^{\lfloor \log l \rfloor})$$

**Dowód:** Na mocy poprzedniego lematu możemy założyć, że wszystkie liście (jest ich  $l = n!$ ) w  $T$  znajdują się na dwóch sąsiednich poziomach. Niech  $h$  będzie wysokością drzewa  $T$ .

**Przypadek 1:**

$$\begin{aligned} l &= 2^h \text{ (tzn. } h = 1) \\ d(T) &= lh = l \log l \end{aligned}$$

**Przypadek 2:**

$$l \neq 2^h$$

Niech  $l_i$  będzie liczbą liści o głębokości  $i$  w drzewie  $T$  ( $i = h, h - 1$ ). Wówczas zachodzą związki:  $l = l_{h-1} + l_h$ ,  $2^{h-1} < l < 2^h$  i  $h - 1 = \lfloor \log l \rfloor$ . Zauważmy, że  $2^h = l_h + 2l_{h-1} = 2l - l_h$ , czyli  $l_h = 2(l - 2^{h-1})$ . Otrzymujemy więc

$$\begin{aligned} d(T) &= hl_h + (h - 1)l_{h-1} = (h - 1)l + l_h = (h - 1)l + 2(l - 2^{h-1}) = \\ &= \lfloor \log l \rfloor l + 2(l - 2^{\lfloor \log l \rfloor}) \end{aligned}$$

cbdo

**Twierdzenie 2.2.**

W każdym algorytmie sortującym przez porównania jest wykonywanych średnio co najmniej  $n \log n - 1,45n$  porównań.

**Dowód:** Niech  $T$  będzie drzewem sortującym ciąg  $a_1, a_2, \dots, a_n$ . Wówczas  $T \in B_n$  i na mocy lematu 2.2

$$\frac{d(T)}{n!} \geq \frac{1}{n!} (n! \lfloor \log n! \rfloor + 2(n! - 2^{\lfloor \log n! \rfloor})) =$$

$$\begin{aligned}
 &= \lfloor \log n! \rfloor + 2 = 2^{1 + \lfloor \log n! \rfloor - \log n!} \geq \lfloor \log n! \rfloor \geq \\
 &\geq \lfloor n \log n - 1,45n + \log e \rfloor \geq n \log n - 1,45n
 \end{aligned}
 \quad \text{cbdo}$$

Przypomnijmy, że oczekiwana liczba porównań dla algorytmu quicksort jest  $\leq 1,4n \log n + O(n)$ , a więc bliska dolnego ograniczenia.

## 2.5. Sortowanie pozycyjne

Wszystkie wielkości w komputerze są przedstawiane za pomocą słów binarnych. W ogólnych algorytmach sortowania nie jest to brane pod uwagę. Również w językach programowania wysokiego poziomu nie ma operacji na liczbach binarnych.

Będziemy używać funkcji wycinania bitów w słowie binarnym, którą w języku Pascal można zdefiniować tak:

```

function bits(x, k, j : integer) : integer;
begin
  bits := (x div 2j) mod 2k
end bits;
  
```

Będziemy zakładać, że  $b$  jest długością słowa binarnego. Interpretację funkcji  $bits$  przedstawiamy na rysunku 2.4.



Rys. 2.4. Funkcja  $bits$  wycina  $j$  bitów, poczynając od bitu o numerze  $k$  z prawej strony

W szczególności  $bits(x, 0, 1)$  oraz  $bits(x, b-1, 1)$  oznaczają odpowiednio pierwszy bit z prawej strony i pierwszy bit z lewej strony.

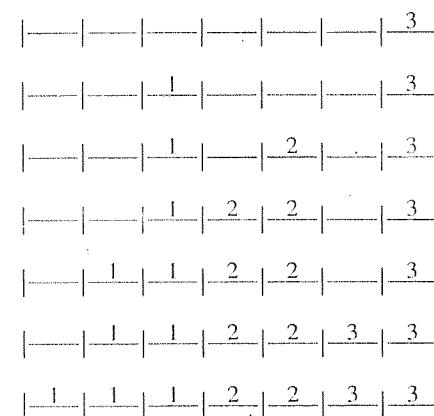
Najprostszą metodą sortowania pozycyjnego jest **metoda liczników częstości**. Zalóżmy, że  $m = 2^b$ . Dla każdego  $j = 0, 1, \dots, m-1$  liczymy, ile razy  $j$  pojawia się w ciągu wejściowym  $a_1, a_2, \dots, a_n$ . Na podstawie obliczonych liczników częstości umieszczamy każdy element  $a_i$  we właściwym miejscu w ciągu wynikowym.

```

procedure countsort;
{a[1..n], t[1..n], count[0..m-1]}
var i, j, p : integer;
begin
  for j := 0 to m-1 do count[j] := 0; {inicjowanie}
  for i := 1 to n do count[a[i]] := count[a[i]] + 1;
  {count[j] to liczba wystąpień liczby j}
  for j := 1 to m-1 do count[j] := count[j-1] + count[j];
  {count[j] to liczba wystąpień elementów ≤ j}
  for i := n downto 1 do
  begin
    p := a[i];
    t[count[p]] := p;
    count[p] := count[p] - 1
  end;
  for i := 1 to n do a[i] := t[i]
end countsort;
  
```

### □ PRZYKŁAD:

Dla ciągu  $q = [1, 3, 1, 2, 2, 1, 3]$  po drugiej instrukcji iteracyjnej  $count[1] = 3$ ,  $count[2] = 2$ ,  $count[3] = 2$ , a po trzeciej instrukcji iteracyjnej:  $count[1] = 3$ ,  $count[2] = 5$ ,  $count[3] = 7$ . W ciągu wynikowym pierwszy element zajmie pozycję  $a[1..count[1]]$ , drugi – pozycje  $a[count[1] + 1..count[2]]$ , a trzeci – pozycje  $a[count[2] + 1..count[3]]$ . Elementy ciągu  $q$  będą wpisywane do tablicy  $t$  w kolejności pokazanej na rysunku 2.5.



Rys. 2.5. Wpisywanie elementów w odpowiednie miejsca ciągu wynikowego

Analiza algorytmu countsort jest bezpośrednią:

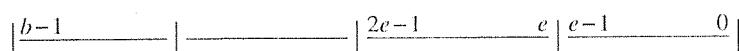
$$W(n, m) = A(n, m) = O(n + m)$$

$$\Delta(n, m) = \delta(n, m) = 0$$

$$S(n, m) = n + m + O(1)$$

Do zalet algorytmu countsort należy przede wszystkim szybkość działania, gdy  $m = O(n)$ . Wadą natomiast jest dodatkowe obciążenie pamięci. Zauważmy, że w zapisanej postaci algorytm ten jest stabilny (dla stabilności istotna jest kolejność wpisywania elementów ciągu od  $a[n]$  do  $a[1]$ ).

Algorytm countsort można stosować tylko dla niewielkich wartości  $m$  (tak, żeby można było użyć tablicy  $count$  o rozmiarze  $m$ ). Dla większych wartości  $m$  można stosować podobną metodę, ale z dzieleniem liczb do posortowania na części, na których algorytm countsort może działać. Podzielimy słowa o  $b$  bitach na  $b/e$  grup  $e$  bitowych (rys. 2.6).



Rys. 2.6. Podział słowa na *ble* grup *e*-bitowych

Najpierw sortujemy ciąg liczb, stosując algorytm countsort względem ostatniej (najmniej znaczącej) grupy bitów, po czym sortujemy ciąg liczb względem przedostatniej grupy bitów. Powtarzamy tę operację, aż dojdziemy do pierwszej (najbardziej znaczącej) grupy bitów. Istotna jest przy tym stabilność algorytmu countsort: o ostatecznej pozycji elementu decyduje pierwszych  $e$  bitów; jeśli są one takie same, to decydujące znaczenie mają dopiero następne grupy bitów.

```

procedure radixsort;
{m = 2e, a, t[1..n], count[0..m - 1]}
var i, j, pass, nofpasses, key : integer;
begin
  nofpasses := b div e;
  if nofpasses * e = b then nofpasses := nofpasses - 1;
  for pass := 0 to nofpasses do
    begin
      for j := 0 to m - 1 do count[j] := 0;
      for i := 1 to n do
        begin
          key := bits(a[i], pass * e, e);
          count[key] := count[key] + 1
        end;
      for j := 1 to m - 1 do count[j] := count[j - 1] + count[j];
      for i := n downto 1 do
        begin
          key := bits(a[i], pass * e, e);
          t[count[key]] := a[i];
        end;
    end;
  end;
end.

```

```

        count[key] := count[key] - 1
    end;
    for i := 1 to n do a[i] := t[i]
end
end radixsort;

```

W tym wypadku analiza złożoności obejmuje więcej parametrów:

$$W(n, m) = A(n, m) = O\left(\frac{b}{e}(n+m)\right) = O(n+m), \text{ gdy } \frac{b}{e} = O(1)$$

$$\Delta(n, m) \equiv \delta(n, m) \equiv 0$$

$$S(n, m) \equiv n + m + O(1)$$

---

□ PRZYKŁAD

Niech  $b = 8$ ,  $c = 2$ ,  $n = 4$  oraz

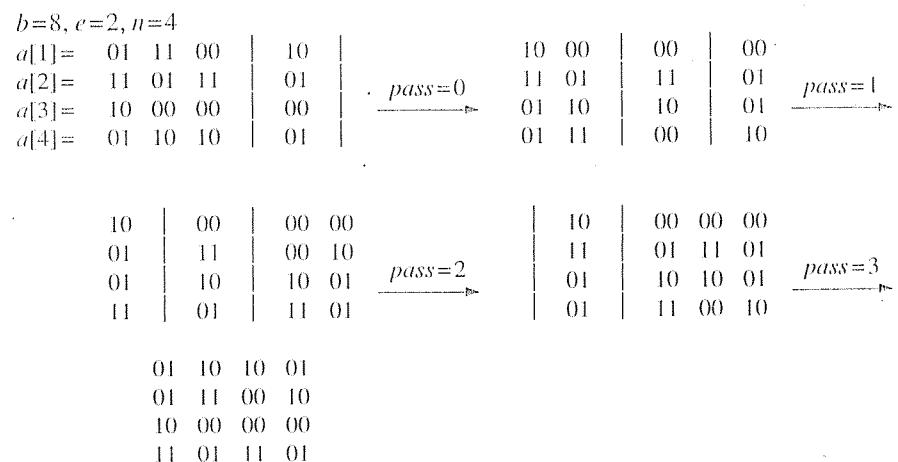
$a[1] = 01110010$

$a[2] = 11011101$

$a[3] = 10000000$

$a[4] = 01101001$

Działanie algorytmu radixsort przedstawiamy na rysunku 2.7.



Rys. 2.7. Działanie algorytmu radixsort

Gdy bity liczb są losowe, można znacznie przyspieszyć działanie algorytmu radixsort. Wystarczy

- (a) użyć radixsort dla najbardziej znaczących  $b/2$  bitów;
- (b) wykonać insertionsort.

Zauważmy, że prawdopodobieństwo, iż dwie liczby będą miały takie same połówki bitów, wynosi  $2^{-b/2}$ . Zatem algorytm insertionsort jest stosowany do ciągu prawie uporządkowanego, czyli działa szybko.

Istnieje też wersja rekurencyjna algorytmu radixsort, w której „rozrzuca się” liczby względem  $c$  najbardziej znaczących bitów i stosuje się rekurencyjnie algorytm radixsort (ewentualnie znowu insertionsort) do każdej podlisty.

Algorytm radixsort nie jest algorymem sortującym przez porównania; wymaga zapisu elementów w układzie pozycyjnym. Do jego wad trzeba też zaliczyć:

- spory współczynnik proporcjonalności, który sprawia, że algorytm jest wolny dla małych rozmiarów  $n$ ;
- spore obciążenie pamięci, szczególnie wtedy, kiedy  $n$  jest duże.

Główna zaletą algorytmu radixsort jest liniowość jego złożoności czasowej (przy odpowiednich założenях dotyczących związku  $n$  z długością słowa maszyny). Dla „średnich” wartości  $n$  może on być lepszy niż algorytm quicksort. (Po odpowiednich modyfikacjach) algorytm radixsort może być użyty do sortowania

- (a) słów w porządku alfabetycznym,
- (b) liczb rzeczywistych z pewnego przedziału.

## 2.6.

### Kolejki priorytetowe i algorytm heapsort

Przez **dynamiczny problem sortowania** rozumiemy takie oto zadanie algorytmiczne: podać strukturę danych dla elementów dynamicznego skończonego multizbioru  $S^b$ , względem którego są wykonywane następujące operacje:

- (a)  $construct(q, S)$ : utworzenie multizbioru  $S = \{a_1, \dots, a_n\}$ , dla danej listy  $q = [a_1, \dots, a_n]$ ;
- (b)  $insert(x, S)$ :  $S := S \cup \{x\}$ ;
- (c)  $deletemax(S)$ : usunięcie z  $S$  największego elementu (dualną operacją jest  $deletemin(S)$ : usunięcie z  $S$  najmniejszego elementu);

przy czym zakładamy jak poprzednio, że elementy zbioru  $S$  pochodzą z pewnego uniwersum  $U$ , które jest liniowo uporządkowane.

<sup>b</sup> W multizbiorze elementy mogą się powtarzać. Multizbiór tym różni się od listy, że elementy w multizbiorze nie są ustawione w ciąg (tak samo jak w zbiorze).

Czasami wymaga się wykonywania również następujących operacji:

- (d)  $findmax(S)$ : wyznaczenie największego elementu w  $S$  (bez usuwania go);
- (e)  $replacemax(x, S)$ : usunięcie z  $S$  największego elementu w  $S$  i wstawienie na jego miejsce  $x$ ;
- (f)  $delete(x, S)$ :  $S := S - \{x\}$ ;
- (g)  $change(x, y, S)$ : zastąpienie w  $S$  elementu  $x$  elementem  $y$ ;
- (h)  $union(S_1, S_2, S)$ :  $S := S_1 \cup S_2$  przy założeniu, że  $S_1$  i  $S_2$  są zbiorami rozłącznymi.

Pięć ostatnich operacji jest definiowalnych za pomocą trzech pierwszych. Czasami jednak lepiej jest implementować je osobno, żeby zmniejszyć ich złożoność.

Strukturę danych będącą rozwiązaniem dynamicznego problemu sortowania nazywamy **kolejką priorytetową**.

Do elementarnych implementacji kolejki priorytetowej należą:

- lista nieuporządkowana, w której  $construct$  ma złożoność  $O(n)$ ,  $insert = O(1)$ ,  $deletemax = O(n)$ ; jest ona godna polecenia wtedy, kiedy w ciągu wejściowym operacji jest dużo operacji  $insert$ , a mało  $deletemax$ ;
- lista uporządkowana, w której  $construct$  ma złożoność  $O(n \log n)$ ,  $insert = O(n)$ , a  $deletemax = O(1)$ ; jest ona godna polecenia wtedy, kiedy w ciągu wejściowym operacji jest dużo operacji  $deletemax$ , a mało  $insert$  (na przykład  $O(\log n)$ ).

Podstawową zaletą elementarnych implementacji kolejki priorytetowej jest ich prostota i niski współczynnik proporcjonalności w funkcjach złożoności.

W tym podrozdziale zajmiemy się strukturą danych, tzw. **kopcem**, dla której operacja  $construct$  ma złożoność czasową  $O(n)$ , a  $insert$  i  $deletemax = O(\log n)$ .

Algorytm heapsort, czyli algorytm sortowania przez kopcowanie, daje się zapisać za pomocą operacji kolejki priorytetowej. Oto ogólny schemat algorytmu sortowania za pomocą kolejki priorytetowej: dana jest lista  $q = [a_1, \dots, a_n]$ ; wykonaj:

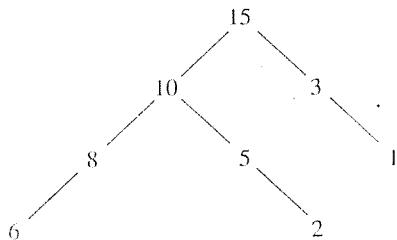
- (a)  $construct(q, S)$ ;
- (b) powtórz  $n - 1$  razy  $deletemax(S)$ .

Omawiany wcześniej algorytm selectionsort umożliwia sortowanie zgodnie z tym schematem przy użyciu jako struktury danych listy nieuporządkowanej.

Stosując implementację kopcową, otrzymamy algorytm sortowania heapsort o pesymistycznej złożoności czasowej  $O(n \log n)$ .

Przystąpimy teraz do zdefiniowania kopca.

Przez **kopiec** rozumiemy drzewo binarne, w węzłach którego znajdują się elementy reprezentowanego multizbioru  $S$  i jest spełniony tzw. **warunek kopca**, a mianowicie:



Rys. 2.8. Przykład kopca

jeśli węzeł  $x$  jest następcą węzła  $y$ , to element w węźle  $x$  jest nie większy niż element w węźle  $y$  (rys. 2.8).

Mówimy, że elementy są wpisane do węzłów kopca zgodnie z **porządkiem kopcowym**, a drzewo ma **uporządkowanie kopcowe**.

Zauważmy, że:

- w korzeniu drzewa znajduje się największy element (bądź jeden z największych elementów, jeśli jest ich kilka);
- na ścieżkach w drzewie, od korzenia do liścia, elementy są uporządkowane w porządku nierosnącym.

Poziom 0

15

2<sup>0</sup> węzłów

Poziom 1

10 15

2<sup>1</sup> węzłów

Poziom 3

8 9 10 15 6 5 2

2<sup>2</sup> węzłów

Rys. 2.9. Kopiec zupełny

Przez **kopiec zupełny** rozumiemy kopiec będący zupełnym drzewem binarnym, tzn. takim, w którym wszystkie poziomy są wypełnione całkowicie, z wyjątkiem co najwyżej ostatniego – spójnie wypełnionego od strony lewej (rys. 2.9).

Zależność między wysokością  $h$  a liczbą węzłów w kopcu zupełnym jest następująca:

$$2^h - 1 < n \leq 2^{h+1} - 1$$

czyli

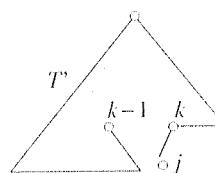
$$2^h \leq n < 2^{h+1}$$

Stąd

$$h = \lfloor \log n \rfloor$$

Kopiec zupełny ma regularny kształt, dzięki czemu może być reprezentowany w tablicy bez dodatkowych dowiązań. Rozważmy numerację węzłów kopca zupełnego poziomami (od strony lewej do prawej). Z definicji kopca zupełnego wynika, że jest ona spójna. Obliczmy numer poprzednika i następcą węzła o numerze  $k$ .

W tym celu rozważmy podkopiec zupełny  $T'$ , kończący się na drugim następcu węzła o numerze  $k - 1$  i taki, że węzeł o numerze  $k$  jest liściem. Założymy, że lewy następnik węzła  $k$  (powiedzmy o numerze  $j$ ) jest określony (rys. 2.10).



Rys. 2.10. Obliczanie numeru następcy węzła k w kopcu

Zauważmy, że  $k - 1$  to liczba węzłów wewnętrznych w  $T'$ , a  $j - 1$  to liczba wszystkich węzłów w  $T'$ .

Można łatwo udowodnić przez indukcję, że w drzewie binarnym, w którym każdy węzeł wewnętrzny ma dwa następcy, liczba wszystkich węzłów w  $T'$  jest równa  $2 \cdot$  liczba węzłów wewnętrznych w  $T' + 1$ . Stąd  $j - 1 = 2(k - 1) + 1$ , czyli  $j = 2k$ .

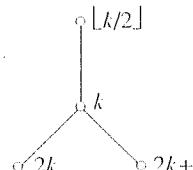
**WŁASNOŚĆ 1:**

Następni węzła  $k$  (o ile istnieją) mają odpowiednio numery  $2k$  i  $2k + 1$ .

Ponieważ  $\lfloor 2k/2 \rfloor = \lfloor (2k+1)/2 \rfloor = k$ , mamy następną własność.

**WŁASNOŚĆ 2:**

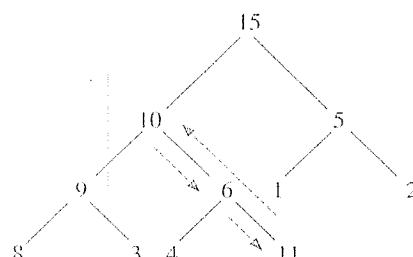
Poprzednik węzła  $k$  (różnego od korzenia) ma numer  $\lfloor k/2 \rfloor$ .



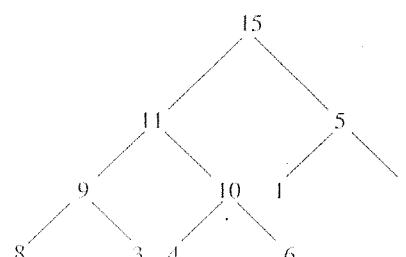
Rys. 2.11. Numery sąsiadów węzła k w kopcu

Operacja *insert*( $x, S$ ), czyli wstawienie elementu  $x$  do kopca zupełnego  $S$ , polega na umieszczeniu  $x$  w pierwszym wolnym miejscu ostatniego poziomu (lub następnego poziomu, gdy ostatni poziom jest całkowicie wypełniony) i przywracaniu zachodzenia warunku kopca, jeśli wstawiony element jest większy niż element znajdujący się w poprzedniku. Aby przywrócić zachodzenie warunku kopca, idziemy w góre w stronę

korzenia, szukając miejsca, gdzie pasowałby wstawiany element. Wstawmy na przykład element 11 do kopca zupełnego z rysunku 2.9 (rys. 2.12).



Rys. 2.12. Wstawianie elementu do kopca



Rys. 2.13. Tak wygląda kopiec po wstawieniu elementu 11

Otrzymamy kopiec zupełny, przedstawiony na rysunku 2.13.

Operacja *insert* składa się ze wstawienia elementu do liścia i dokonywanych później na ścieżce do korzenia zamian w celu przywrócenia zachodzenia warunku kopca. Użyjemy teraz procedury pomocniczej *upheap* (nad korzeniem ustawiemy wartownika, przypisując mu wartość  $+\infty$ ).

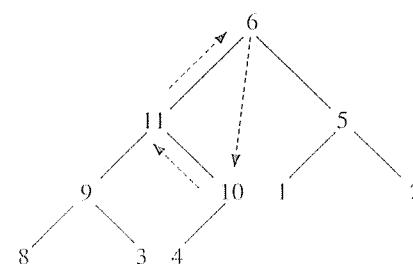
```
procedure upheap(k : integer);
var l, v : integer;
begin
  v := a[k]; a[0] := +∞;
  l := k div 2;
  {warunek kopca jest zaburzony co najwyżej tylko dla v}
  while a[l] < v do
    begin {węzeł l jest poprzednikiem węzła k}
      a[k] := a[l];
      k := l; l := l div 2
    end;
    a[k] := v
  end upheap;
procedure insert(v : integer);
begin
  n := n + 1;
  a[n] := v;
  upheap(n)
end insert;
```

Ponieważ wysokość kopca zupełnego wynosi  $h = \lfloor \log n \rfloor$ , pesymistyczna złożoność czasowa procedury *insert*, mierzona liczbą porównań, wynosi

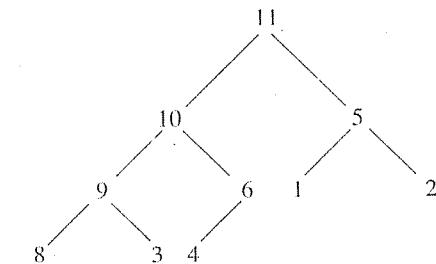
$$W_{\text{insert}}(n) = \lfloor \log n \rfloor + 1$$

gdzie  $n$  oznacza liczbę elementów w kopcu po wstawieniu  $v$  (liczone jest też porównanie z wartownikiem).

Aby wykonać operację *deletemax*( $S$ ), czyli usunąć z kopca największy element, wystarczy usunąć ten element z korzenia, wziąć prawy skrajny liść z ostatniego poziomu, usunąć go z drzewa, a element znajdujący się w tym liściu wstawić do korzenia. Ponieważ po tym wstawieniu warunek kopca może być zaburzony w jednym węźle, tym razem w korzeniu, należy tak opuścić w dół element wstawiony do korzenia, żeby przywrócić zachodzenie warunku kopca. Wykonajmy na przykład *deletemax*( $S$ ) dla drzewa z rysunku 2.13 (rys. 2.14).



Rys. 2.14. Usuwanie największego elementu w kopcu



Rys. 2.15. Wynik operacji deletemax

Otrzymujemy kopiec zupełny, przedstawiony na rysunku 2.15.

Użyjemy teraz procedury pomocniczej *downheap*.

```
procedure downheap(k : integer);
label 0;
var i, j, v : integer;
begin
  v := a[k];
  while k ≤ n div 2 do
    begin
      j := 2 * k; {j jest następcikiem k}
      if j < n then if a[j] < a[j + 1] then j := j + 1;
      if v ≥ a[j] then goto 0;
      a[k] := a[j];
      k := j
    end;
  0: a[k] := v
end downheap;
```

Zauważmy, że na każdym poziomie drzewa (z wyjątkiem zerowego) w algorytmie *downheap* mogą być wykonane dwa porównania.

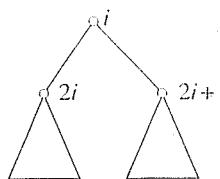
```
function deletemax : integer;
begin
  deletemax := a[1];
  a[1] := a[n];
  n := n - 1;
  downheap(1)
end deletemax;
```

Pesymistyczna złożoność czasowa operacji *deletemax*, mierzona liczbą porównań, wynosi

$$W_{deletemax}(n) = 2\lfloor \log n \rfloor$$

gdzie  $n$  oznacza liczbę elementów w kopcu po usunięciu największego elementu.

Pozostała nam jeszcze do omówienia operacja *construct*. Gdybyśmy realizowali ją za pomocą  $n$  operacji *insert*, jej złożoność czasowa byłaby  $O(n \log n)$ . Możemy jednak wykonać ją w czasie liniowym. Wystarczy konstrukcję kopca rozpocząć od dołu drzewa, tworzyć małe podkopce i łączyć je w większe – aż do powstania całego kopca. Założymy, że podkopce o korzeniach  $2i$  i  $2i+1$  zostały już skonstruowane. Aby połączyć je i wstawić kolejny element  $x$  w węźle  $i$ , wystarczy wywołać *downheap*( $i$ ) (warunek kopca jest zaburzony tylko dla węzła  $i$ ). Na rysunku 2.16 widać wynik tej operacji.



Rys. 2.16. Włączanie węzła  $i$  do kopca

Oto procedura *construct*.

```
procedure construct;
{elementy listy q=[a1, ..., an] znajdują się w tablicy a[1..n]}
var i : integer;
begin
  for i := n div 2 downto 1 do downheap(i)
end construct;
```

Zauważmy, że procedura *downheap*( $i$ ) wywołana dla węzła  $i$  z poziomu  $l$  wykonuje co najwyżej  $2(h-l)$  porównań oraz że na poziomie  $l$  jest  $2^l$  węzłów ( $0 \leq l < h$ ). Mamy zatem

## 2.6. Kolejki priorytetowe i algorytm heapsort

$$\begin{aligned} W_{construct}(n) &= \sum_{l=0}^{h-1} 2^l 2(h-l) = \sum_{l=1}^h 2^{h-l} 2l = 2^{h+1} \sum_{l=1}^h 2^{-l} l = 2^{h+1} \sum_{l=1}^h \sum_{j=1}^l 2^{-l} = \\ &= 2^{h+1} \sum_{j=1}^h \sum_{l=j}^h 2^{-l} \leq 2^{h+1} \sum_{j=1}^h 2^{-j} \sum_{s=0}^{\infty} 2^{-s} \leq 2^{h+1} \sum_{j=1}^h 2^{-j} 2 = 2^{h+2} = O(n) \end{aligned}$$

Otrzymujemy wreszcie sam algorytm sortowania przez kopcowanie.

```
procedure heapsort; {a[1..n] – lista do posortowania}
var m, i : integer;
begin
  m := n;
  construct;
  for i := m downto 2 do
    a[i] := deletemax;
  n := m
end heapsort;
```

Przeanalizowaliśmy już liczbę porównań wykonywanych w procedurze *construct*. Pozostaje nam jeszcze analiza liczby porównań wykonywanych w  $n-1$  wywołaniach procedury *deletemax*. Dla węzła  $i$ , usuwanego z poziomu  $l$ , w procedurze *deletemax* jest wykonywanych co najwyżej  $2l$  porównań. Mamy więc razem co najwyżej

$$(n-2^h+1)2h + \sum_{l=1}^{h-1} (2l)2^l$$

porównań. Obliczmy najpierw sumę:

$$\begin{aligned} \sum_{l=1}^{h-1} l2^l &= \sum_{l=1}^{h-1} \sum_{j=1}^l 2^l = \sum_{j=1}^{h-1} \sum_{l=j}^{h-1} 2^l = \sum_{j=1}^{h-1} 2^j \sum_{s=0}^{h-1-j} 2^s = \sum_{j=1}^{h-1} 2^j (2^{h-1-j+1} - 1) = \\ &= \sum_{j=1}^{h-1} 2^h - \sum_{j=1}^{h-1} 2^j = (h-1)2^h - 2^h + 2 \end{aligned}$$

Pesymistyczna złożoność czasowa algorytmu heapsort daje się zatem oszacować w następujący sposób:

$$\begin{aligned} W(n) &= O(n) + (n-2^h+1)2h + 2(h-1)2^h - 2^{h+1} + 4 = \\ &= 2h(n-2^h+1+2^h) + O(n) = 2n\log n + O(n) \end{aligned}$$

Nie jest znana pełna analiza oczekiwanej złożoności czasowej tego algorytmu. Wyniki przeprowadzonych badań wskazują, że współczynnik przy  $n\log n$  jest bliski 2. Tłumaczy

to przewagę algorytmu quicksort nad heapsort. Zaletą tego ostatniego jest to, że działa w miejscu, tzn.

$$S(n) = O(1)$$

Używając kopca reprezentowanego w tablicy, każdą operację kolejki priorytetowej, z wyjątkiem *union*, można wykonać w czasie  $O(\log n)$ . Opracowano wiele konkretnych struktur danych, które umożliwiają wykonanie również operacji *union* w czasie  $O(\log n)$ . Będzie o nich mowa w dalszej części książki.

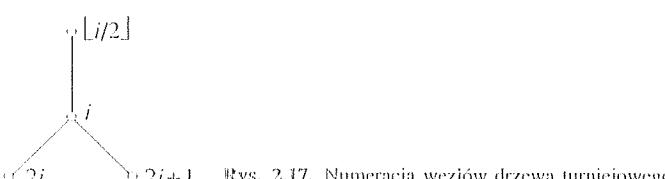
2.7.

## Drzewa turniejowe i zadania selekcji

W operacji *deletemax* na kopcu zupełnym jest wykonywana operacja *downheap(1)*, wymagająca dwóch porównań na każdym poziomie kopca. Gdybyśmy zamiast przejścia od korzenia do liścia mieli przejście od liścia do korzenia (jak w *insert*), udałoby się nam zmniejszyć liczbę porównań dwukrotnie, ale kosztem skomplikowania struktury danych. Otrzymalibyśmy strukturę danych zwaną **drzewem turniejowym**, gdyż można ją zinterpretować jako diagram rozgrywek pucharowych.

Niech  $q = [a_1, a_2, \dots, a_n]$  będzie listą różnych elementów. Przez **drzewo turniejowe dla listy  $q$**  rozumiemy takie drzewo binarne, że

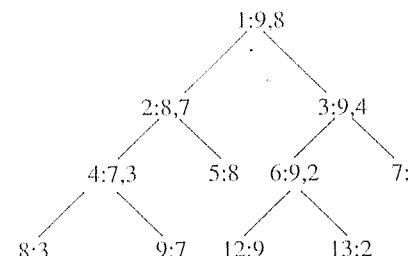
- wszystkie poziomy w drzewie są wypełnione całkowicie, z wyjątkiem co najwyżej ostatniego poziomu;
- każdy węzeł wewnętrzny ma dwa następcy;
- elementy  $a_1, a_2, \dots, a_n$  są zapisane w liściach, przy czym z każdym liściem  $x$  jest związany atrybut  $val(x)$ , czyli element zapisany w liściu  $x$ ;
- na każdym poziomie węzły wewnętrzne oraz liście są połączone w listy cykliczne zgodnie z porządkiem od lewej strony do prawej;
- uwzględniając uporządkowanie poziomami (jak w heapsort), węzeł o numerze  $i$  jest reprezentowany przez indeks  $i$  (rys. 2.17), przy czym indeksuje się także puste miejsca w ostatnim poziomie;
- dla każdego węzła wewnętrznego o numerze  $i$  są dodatkowo określone dwa atrybuty:  $mx(i)$  – numer liścia zawierającego największy element w poddrzewie węzła  $i$ , tzn. element  $\max(val(mx(2i)), val(mx(2i + 1)))$  oraz  $mn(i)$  – numer liścia zawierającego element  $\min(val(mx(2i)), val(mx(2i + 1)))$ .



Rys. 2.17. Numeracja węzłów drzewa turniejowego

### PRZYKŁAD:

Węzły wewnętrzne widoczne na rysunku 2.18 zapisujemy jako  $i$ :  $val(mx(i))$ ,  $val(mn(i))$ , a liście  $i$ :  $val(i)$ .



Rys. 2.18. Drzewo turniejowe

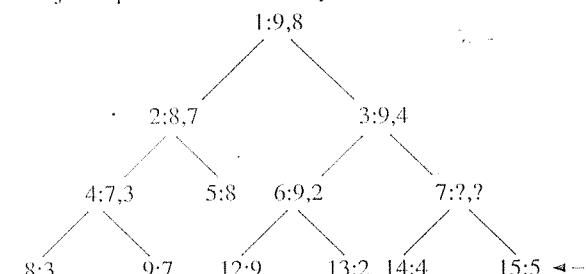
Zauważmy, że jeśli  $n$ , gdzie  $n > 1$ , jest liczbą liści w drzewie turniejowym, a  $h$  jego wysokość, to

$$2^{h-1} < n \leq 2^h$$

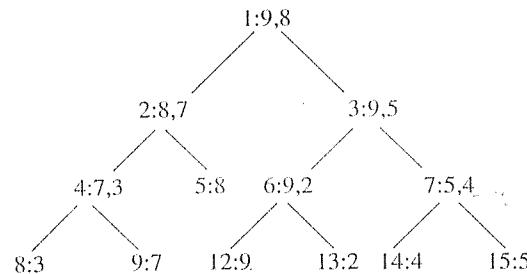
czyli

$$h = \lceil \log n \rceil$$

Operację *insert(x, S)* wykonujemy w następujący sposób: tworzymy dwa nowe liście na ostatnim poziomie (jeśli ostatni poziom jest wypełniony całkowicie, to otwieramy nowy poziom) i do jednego z nich wstawiamy  $x$ ; wybieramy liść, powiedzmy  $i$ , z poziomu sąsiedniego, zamieniamy go na węzeł wewnętrzny, a wartość  $val(i)$  wstawiamy do drugiego nowego liścia; czynimy oba nowe liście następcami  $i$ ; aktualizujemy wartości na ścieżce w drzewie, idąc w stronę korzenia. Po wykonaniu na przykład *insert(5, S)* względem drzewa z rysunku 2.18 otrzymujemy najpierw drzewo turniejowe przedstawione na rysunku 2.19, a potem, po aktualizacji wartości na ścieżce do korzenia, drzewo turniejowe przedstawione na rysunku 2.20.



Rys. 2.19. Wstawianie elementu do drzewa turniejowego



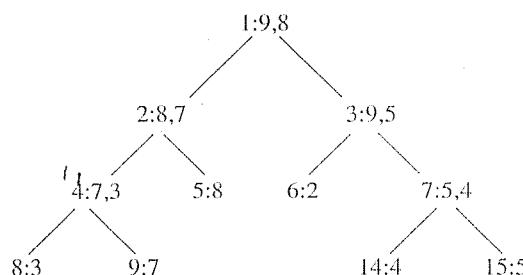
Rys. 2.20. Drzewo turniejowe po wstawieniu elementu

Zauważmy, że

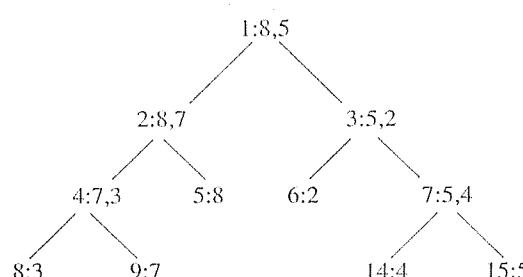
$$W_{\text{insert}}(n) = \lceil \log n \rceil$$

gdzie  $n$  jest liczbą liści po wstawieniu  $x$ .

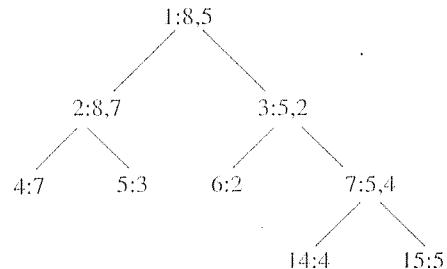
Operację  $\text{deletemax}(S)$  wykonujemy w następujący sposób: usuwamy liść zawierający największy element ( $mx(1)$  jest numerem największego elementu); aktualizujemy wartości na ścieżce od  $mx(1)$  do korzenia, przy czym jeśli  $mx(1)$  nie należy do ostatniego poziomu drzewa, to najpierw w miejscu  $mx(1)$  wstawiamy dowolny element z ostatniego poziomu, który przy pierwszym porównaniu okazał się mniejszy od swojego sąsiada. Ważne jest, by poprawianie zaczynać od dolnych poziomów drzewa. Na każdym z  $h - 1$  poziomów wykonujemy tylko jedno porównanie! Przeprowadzimy  $\text{deletemax}$  na drzewie turniejowym z rysunku 2.20. Najpierw otrzymamy drzewo przedstawione na rysunku 2.21, a następnie, po aktualizacji wartości na ścieżce, drzewo turniejowe przedstawione na rysunku 2.22.



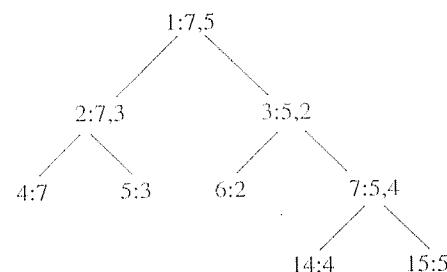
Rys. 2.21. Usuwanie największego elementu w drzewie turniejowym



Rys. 2.22. Drzewo turniejowe po usunięciu największego elementu



Rys. 2.23. Powtórne usuwanie największego elementu w drzewie turniejowym



Rys. 2.24. Drzewo turniejowe po powtórnych usuwaniach największego elementu

Wykonajmy  $\text{deletemax}(S)$  jeszcze raz. Najpierw otrzymamy (przesuwając element 3 w miejsce usuwanego elementu 8) drzewo przedstawione na rysunku 2.23, a następnie, po poprawieniu ścieżki, drzewo przedstawione na rysunku 2.24.

Zauważmy, że

$$W_{\text{deletemax}}(n) = \lceil \log n \rceil - 1$$

gdzie  $n$  jest liczbą liści przed wykonaniem  $\text{deletemax}$  (współczynnik przy  $\log n$  wynosi 1 a nie 2, jak to było w heapsort!).

Skonstruowanie algorytmu dla operacji  $\text{construct}$  tak, żeby

$$W_{\text{construct}}(n) = n - 1$$

jak również zapisanie dokładnie algorytmów dla każdej operacji kolejki priorytetowej pozostawiamy Tobie, Drogi Czytelniku. (Jakie znaczenie mają listy cykliczne węzłów wewnętrznych i liści na każdym poziomie?).

Algorytm tournamentsort, czyli algorytm sortowania turniejowego, ma ten sam schemat co heapsort: po wykonaniu  $\text{construct}(q, S)$  trzeba  $n - 1$  razy powtórzyć operację  $\text{deletemax}(S)$ . Jedyna różnica polega na użyciu do implementacji kolejki priorytetowej drzewa turniejowego zamiast kopca. Korzystając z oszacowań dla  $\text{construct}$  i  $\text{deletemax}$ , oszacujmy pesymistyczną złożoność czasową algorytmu tournamentsort:

$$W(n) = n - 1 + \sum_{i=2}^n (\lceil \log i \rceil - 1) = \sum_{i=2}^n \lceil \log i \rceil$$

Zauważmy, że różnica z dolnym ograniczeniem daje się tak oto oszacować z góry

$$\sum_{i=2}^n \lceil \log i \rceil - \sum_{i=2}^n \log i \leq n - 1$$

Algorytm tournamentsort jest prawie optymalny, jeśli chodzi o liczbę porównań. Duża jednak liczba innych działań (choćż oczywiście proporcjonalna do  $n \log n$ ) i spore wymagania dotyczące pamięci czynią ten algorytm nieprzydatnym w praktyce.

Problemami podobnymi do problemu sortowania są zadania selekcji. Dla listy  $q = [a_1, a_2, \dots, a_n]$  i parametru  $k$  ( $1 \leq k \leq n$ ) wyróżniamy trzy najważniejsze zadania selekcji:

- wyznaczenie  $k$  największych elementów (uporządkowanych);
- wyznaczenie  $k$  największych elementów (bez warunku uporządkowania);
- wyznaczenie  $k$ -tego największego elementu.

(Analogiczne zadania dotyczą najmniejszych elementów).

Do realizacji pierwszych dwóch zadań możemy użyć dowolnej implementacji kolejki priorytetowej.

Aby znaleźć  $k$  największych uporządkowanych elementów na liście  $q$ , wykonujemy najpierw  $construct(q, S)$ , następnie  $k - 1$  razy  $deletemax(S)$ , a na zakończenie wywołujemy  $findmax(S)$  (nie trzeba już ostatniego elementu usuwać z kolejki priorytetowej). Przyjmując implementację kolejki priorytetowej za pomocą drzewa turniejowego, otrzymujemy następującą pesymistyczną złożoność czasową (mierzoną liczbą porównań):

$$W(n) = (n - 1) + \sum_{i=n-k+2}^n (\lceil \log i \rceil - 1) = (n - k) + \sum_{i=n-k+2}^n \lceil \log i \rceil$$

Jako ćwiczenie pozostawiamy Ci udowodnienie, że dla  $k = 2$  można otrzymać algorytm optymalny względem liczby porównań oraz że dolne ograniczenie na liczbę porównań wynosi (zad. 2.24 i 2.25)

$$(n - k) + \lceil \sum_{i=n-k+2}^n \log i \rceil$$

Aby znaleźć  $k$  największych elementów na liście  $q$ , niekoniecznie uporządkowanych, wykonujemy najpierw  $construct(q[1..n-k+1], S)$ , następnie dla  $i = n - k + 2$  do  $n$   $replacemax(a_i, S)$  (na miejsce usuwanego największego elementu wstawiamy kolejny, nie rozpatrywany jeszcze element listy  $q$ ), a na zakończenie wykonujemy  $findmax(S)$ . Przedstawiony algorytm nosi nazwę algorytmu Hadiana-Sobela. Przyjmując implementację

kolejki priorytetowej za pomocą drzewa turniejowego, otrzymujemy następującą pesymistyczną złożoność czasową (mierzoną liczbą porównań):

$$W(n) = (n - k) + (k - 1) \lceil \log(n - k + 1) \rceil$$

(Operacja  $replacemax$  jest logicznie połączeniem  $deletemax$  i  $insert$ ; przy implementacji za pomocą drzewa turniejowego można ją zapisać za pomocą jednego przejścia od liścia do korzenia).

Do wyznaczania  $k$  największych elementów można też użyć dowolnego algorytmu wyznaczającego  $k$ -ty największy element. Po wyznaczeniu  $k$ -tego największego elementu wystarczy tylko jedno przejście po liście  $q$ , żeby wypisać wszystkie  $k$  największe elementy.

## 2.8.

### Szybkie algorytmy wyznaczania $k$ -tego największego elementu w ciągu

Aby znaleźć  $k$ -ty największy element na liście  $q$  (często  $k = \lceil n/2 \rceil$  – przypadek mediany), możemy zastosować podobną metodę jak w algorytmie quicksort. Ciąg wejściowy dzielimy na dwa podciągi względem pewnego elementu w ciągu. W zależności od liczby podciągów  $k$ -tego elementu szukamy albo w lewym, albo w prawym podciągu. Algorytm ten nosi nazwę algorytmu Hoare'a.

```
function select(l, r, k : integer) : integer;
{1 ≤ k ≤ r - l + 1; funkcja wyznacza  $k$ -ty największy element w tablicy a[l..r]; zewnętrzne wywołanie select(l, n, k)}
var j: integer;
begin
  if l < r then
    begin
      j := partition(l, r);
      if k - 1 = r - j then select := j else
        if k - 1 < r - j then select := select(j + 1, r, k) else
          select := select(l, j - 1, k - (r - j + 1));
    end
  end else select := l;
end select;
```

Ponieważ wywołanie  $select(l, r, k)$  pociąga za sobą co najwyżej jedno wywołanie rekurencyjne i to jako ostatnią instrukcję procedury, rekursję można w sposób standardowy zastąpić iteracją. Zapisanie algorytmu iteracyjnego pozostawiamy Ci jako ćwiczenie.

$$W(n) = n - 1 + \sum_{i=2}^n (\lceil \log i \rceil - 1) = \sum_{i=2}^n \lceil \log i \rceil$$

Zauważmy, że różnica z dolnym ograniczeniem daje się tak oszacować z góry

$$\sum_{i=2}^n \lceil \log i \rceil - \sum_{i=2}^n \log i \leq n - 1$$

Algorytm tournamentsort jest prawie optymalny, jeśli chodzi o liczbę porównań. Duża jednak liczba innych działań (choćż oczywiście proporcjonalna do  $n \log n$ ) i spore wymagania dotyczące pamięci czynią ten algorytm nieprzydatnym w praktyce.

Problemami podobnymi do problemu sortowania są zadania selekcji. Dla listy  $q = [a_1, a_2, \dots, a_n]$  i parametru  $k$  ( $1 \leq k \leq n$ ) wyróżniamy trzy najważniejsze zadania selekcji:

- wyznaczenie  $k$  największych elementów (uporządkowanych);
- wyznaczenie  $k$  największych elementów (bez warunku uporządkowania);
- wyznaczenie  $k$ -tego największego elementu.

(Analagiczne zadania dotyczą najmniejszych elementów).

Do realizacji pierwszych dwóch zadań możemy użyć dowolnej implementacji kolejki priorytetowej.

Aby znaleźć  $k$  największych uporządkowanych elementów na liście  $q$ , wykonujemy najpierw  $construct(q, S)$ , następnie  $k-1$  razy  $deletemax(S)$ , a na zakończenie wywołujemy  $findmax(S)$  (nie trzeba już ostatniego elementu usuwać z kolejki priorytetowej). Przyjmując implementację kolejki priorytetowej za pomocą drzewa turniejowego, otrzymujemy następującą pesymistyczną złożoność czasową (mierzoną liczbą porównań):

$$W(n) = (n - 1) + \sum_{i=n-k+2}^n (\lceil \log i \rceil - 1) = (n - k) + \sum_{i=n-k+2}^n \lceil \log i \rceil$$

Jako ćwiczenie pozostawiamy Ci udowodnienie, że dla  $k = 2$  można otrzymać algorytm optymalny względem liczby porównań oraz że dolne ograniczenie na liczbę porównań wynosi (zad. 2.24 i 2.25)

$$(n - k) + \lceil \sum_{i=n-k+2}^n \log i \rceil$$

Aby znaleźć  $k$  największych elementów na liście  $q$ , niekoniecznie uporządkowanych, wykonujemy najpierw  $construct(q[1..n-k+1], S)$ , następnie dla  $i = n - k + 2$  do  $n$   $replacemax(a_i, S)$  (na miejsce usuwanego największego elementu wstawiamy kolejny, niezrozpatrywany jeszcze element listy  $q$ ), a na zakończenie wykonujemy  $findmax(S)$ . Przedstawiony algorytm nosi nazwę algorytmu Hadiana-Sobela. Przyjmując implementację

kolejki priorytetowej za pomocą drzewa turniejowego, otrzymujemy następującą pesymistyczną złożoność czasową (mierzoną liczbą porównań):

$$W(n) = (n - k) + (k - 1) \lceil \log(n - k + 1) \rceil$$

(Operacja  $replacemax$  jest logicznie połączeniem  $deletemax$  i  $insert$ ; przy implementacji za pomocą drzewa turniejowego można ją zapisać za pomocą jednego przejścia od liścia do korzenia).

Do wyznaczania  $k$  największych elementów można też użyć dowolnego algorytmu wyznaczającego  $k$ -ty największy element. Po wyznaczeniu  $k$ -tego największego elementu wystarczy tylko jedno przejście po liście  $q$ , żeby wypisać wszystkie  $k$  największe elementy.

## 2.8.

### Szybkie algorytmy wyznaczania $k$ -tego największego elementu w ciągu

Aby znaleźć  $k$ -ty największy element na liście  $q$  (często  $k = \lceil n/2 \rceil$  – przypadek mediany), możemy zastosować podobną metodę jak w algorytmie quicksort. Ciąg wejściowy dzielimy na dwa podciągi względem pewnego elementu w ciągu. W zależności od liczbeności podciągów  $k$ -tego elementu szukamy albo w lewym, albo w prawym podciągu. Algorytm ten nosi nazwę algorytmu Hoare'a.

```
function select(l, r, k : integer) : integer;
{1 ≤ k ≤ r - l + 1; funkcja wyznacza k-ty największy element w tablicy a[l..r]; zewnętrzne wywołanie select(l, n, k)}
var j: integer;
begin
  if l < r then
    begin
      j := partition(l, r);
      if k - 1 = r - j then select := j else
        if k - 1 < r - j then select := select(j + 1, r, k) else
          select := select(l, j - 1, k - (r - j + 1));
    end
  end else select := l;
end select;
```

Ponieważ wywołanie  $select(l, r, k)$  pociąga za sobą co najwyżej jedno wywołanie rekurencyjne i to jako ostatnią instrukcję procedury, rekursję można w sposób standardowy zastąpić iteracją. Zapisanie algorytmu iteracyjnego pozostawiamy Ci jako ćwiczenie.

Równanie na pesymistyczną złożoność czasową jest następujące:

$$\begin{cases} W(1) = 0 \\ W(n) = (n+1) + W(n-1) \text{ dla } n > 1 \end{cases}$$

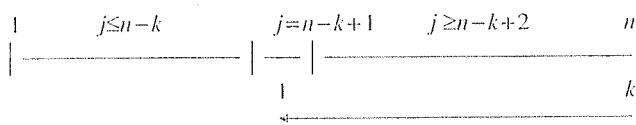
Oto jego rozwiązanie:

$$W(n) = \frac{1}{2}n^2 + O(n)$$

Pesymistyczna wrażliwość algorytmu na dane wejściowe wynosi

$$\Delta(n) = O(n^2)$$

(w przypadku optymistycznym wykonywanych jest tylko  $n+1$  porównań).



Rys. 2.25. Podział rekurencyjny w algorytmie Hoare'a

Równanie rekurencyjne na oczekiwanyą złożoność czasową w modelu permutacyjnym jest następujące (rys. 2.25):

$$\begin{cases} A(1) = 0 \\ A(n) = (n+1) + \frac{1}{n} \sum_{j=1}^{n-k} A(n-j) + \frac{1}{n} \sum_{j=n-k+2}^n A(j-1) \quad \text{dla } n > 1 \end{cases}$$

Przez indukcję względem wartości  $n$  można udowodnić, że

$$A(n) \leq 4n$$

Algorytm Hoare'a działa w miejscu, tzn.

$$S(n) = O(1)$$

Algorytm Hoare'a, podobnie jak quicksort, jest szybkim algorytmem dla danych losowych i tak jak quicksort ma pesymistyczną złożoność czasową  $O(n^2)$ .

Przedstawimy teraz szkie algorytmu liniowego na wyznaczanie  $k$ -tego co do wielkości elementu w ciągu (algorytm Bluma-Floyda-Pratta-Rivesta-Tarjana). Jego zasada działania jest taka jak algorytmu Hoare'a. Różnica polega na specjalnym, wyważonym doborze elementu dzielącego tak, żeby każda z części miała rozmiar nie większy niż  $3/4$  rozmiaru ciągu ulegającego podziałowi ( $p$  jest parametrem będącym stałą całkowitą, na przykład  $p = 50$ ).

```
function select(k : integer, q : list) : integer;
{q = [a1, a2, ..., an]}

begin
  if n < p then {w tym wypadku nie stosujemy rekursji}
  begin
    posortuj q;
    select := k-ty od końca element na liście q
  end else
  begin
    podziel q na ⌊n/5⌋ 5-elementowych podciągów i (ewentualnie)
    jeden ciąg co najwyżej 4-elementowy;
    posortuj osobno każdy podciąg;
    niech M będzie ciągiem median tych podciągów;
    m := select(⌈|M|/2⌉, M);
    niech q1, q2 i q3 będą ciągami elementów z q,
    które są odpowiednio ≤, = i ≥ m;
    if k ≤ |q1| then select := select(k, q1) else
    if |q2| + |q3| ≥ k then select := m else
    select := select(k - |q2| - |q1|, q3)
  end
end select;
```

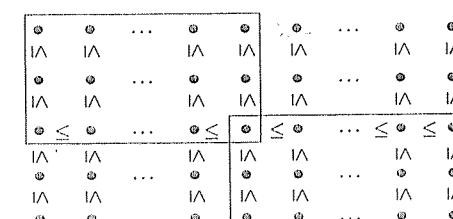
Otrzymujemy następujące równanie rekurencyjne na pesymistyczną złożoność czasową powyższego algorytmu.

$$\begin{cases} W(n) \leq c_1 n \text{ dla } n < p \\ W(n) = W(|M|) + \max(W(|q_1|), W(|q_3|)) + c_2 n \text{ dla } n \geq p \end{cases}$$

gdzie  $c_1$  i  $c_2$  są pewnymi stałymi.

Zauważmy, że co najmniej  $(1/4)n$  elementów jest mniejsza niż element dzielący lub równa mu i podobnie – co najmniej  $(1/4)n$  elementów jest większa niż ten element lub równa mu (rys. 2.26).

Elementy, które są na pewno ≤ elementu dzielącego



Elementy, które są na pewno ≥ elementu dzielącego

Rys. 2.26. Podział ciągu względem mediany median

Stąd  $|q_1|, |q_3| \leq \frac{3}{4}n$ , a zatem

$$W(n) \leq c_2 n + W\left(\frac{1}{5}n\right) + W\left(\frac{3}{4}n\right) \text{ dla } n \geq 1$$

Przez indukcję dowodzi się, że

$$W(n) \leq 20 \text{ cm}$$

gdzie  $c = \max(c_1, c_2)$

Dla liniowości rozwiązyania równania rekurencyjnego jest istotne, że  $1/5 + 3/4 < 1$ . Podział ciągu na 5-elementowe podciągi jest związany z tym warunkiem. Przy zastosowaniu podziału na 3-elementowe podciągi algorytm przestaje być liniowy!

## 2.9. Scalanie ciągów uporządkowanych

Zadaniem pokrewnym do sortowania jest **zadanie scalania**, którego definicja jest następująca: mając dane dwa ciągi uporządkowane:  $a_1 \leq a_2 \leq \dots \leq a_n$  i  $b_1 \leq b_2 \leq \dots \leq b_m$ , połączyć je (scalić) w jeden ciąg uporządkowany  $c_1 \leq c_2 \leq \dots \leq c_{n+m}$ .

Zadanie scalania można rozwiązać za pomocą dowolnego algorytmu sortowania. Jest ono jednak prostsze niż zadanie sortowania. Istnieje algorytm, za pomocą którego rozwiązuje się je w czasie liniowym względem  $n + m$ . Wystarczy rozpatrywać elementy obu ciągów w kolejności od najmniejszego do największego, porównując bieżące elementy i przesyłając mniejszy z nich do ciągu wynikowego. Wygodnie jest użyć w tym wypadku implementacji dowiązanej list. Założymy dodatkowo, że listy kończą się specjalnym elementem  $+\infty$ . Przymijmy, że każdy węzeł struktury dowiązanej ma postać

$x;key(x), next(x)$

```

function merge(a, b : list) : list
var c, d : list;
begin
    d := [+∞]; c := d;
repeat
    if key(a) ≤ key(b) then
        begin
            next(d) := a; d := a
            a := next(a)
        end
    else

```

```

begin
    next(d) := b; d := b;
    b := next(b)
end
until key(d) = maxint;
merge := next(c)
end merge;

```

Analiza złożoności jest oczywista:

$$W(n, m) = A(n, m) = O(n + m)$$

$$\Delta(n) = \delta(n) = 0$$

$S(n) = O(n)$  (miejsce na dowiązania)

Jeśli chodzi o złożoność czasową, powyższy algorytm jest optymalny wtedy, kiedy  $m = \Theta(n)$ . Gdy natomiast  $n$  i  $m$  są różnych rzędów wielkości, w optymalnym algorytmie (opartym o metodę dzielenia binarnego) jest wykonywanych  $\mathcal{O}(m(\log(n/m) + 1))$  porównań, przy założeniu, że  $n \geq m$  (zad. 2.34 i 2.35).

Scalanie może być zastosowane do rozwiązywania problemu sortowania metodą „dziel i zwyciężaj”. Oto ogólny schemat tego algorytmu.

```

procedure mergesort(q : list);
{q = [a1, ..., an]}
begin
.. if n > 1 then
begin
    podziel q na dwie części q1 i q2 o rozmiarach
    odpowiednio  $\lfloor n/2 \rfloor$  i  $\lceil n/2 \rceil$ ;
    posortuj rekurencyjnie q1 i q2;
    scal q1 i q2, używając algorytmu liniowego scalania
end
end mergesort;

```

Otrzymujemy następujące równanie rekurencyjne na pesymistyczną (a także oczekiwana) złożoność czasową:

$$\begin{cases} W(1) = 0 \\ W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n \quad \text{dla } n > 1 \end{cases}$$

Aby w rozwiązyaniu tego równania uzyskać dokładny współczynnik modyfikatywny przy  $n \log n$ , rozważmy równanie na liczbę poziomów rekursji:

$$\begin{cases} t(1) = 0 \\ t(n) = t(\lceil n/2 \rceil) + 1 \end{cases} \quad \text{dla } n > 1$$

Dla  $n = 2^k$  otrzymujemy  $t(2^k) = k$ . Liczba poziomów rekursji dla dowolnego  $n$  to  $t(n) \leq t(2^{\lceil \log n \rceil}) = \lceil \log n \rceil \leq \log n + 1$ . Na każdym poziomie rekursji jest wykonywanych  $n$  operacji porównywania (z wyjątkiem ostatniego, na którym nie wykonuje się porównań, i ewentualnie przedostatniego, na którym wykonuje się co najwyżej  $n$  porównań). Otrzymujemy zatem

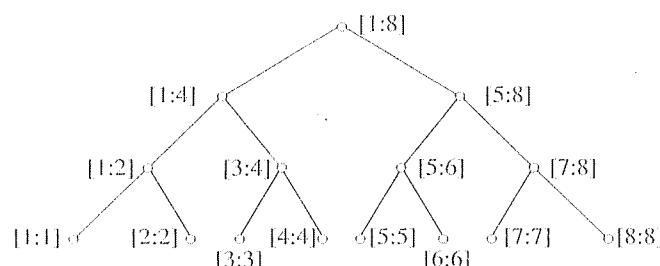
$$W(n) = A(n) = n \log n + O(n)$$

$$\Delta(n) = \delta(n) = 0$$

$$S(n) = O(n)$$

Jeśli chodzi o liczbę porównań, jest to więc algorytm bliski optymalnego. Konieczna jest jednak dodatkowa pamięć  $O(n)$ . Algorytm ten wymaga też sporo innych operacji poza porównaniami. Wyniki badań testowych stawiają go między algorytmami quicksort i heapsort.

Rekursję możemy wyeliminować, zastępując ją kolejką podciągów pozostających do posortowania. Zauważmy, że drzewo wywołań rekurencyjnych algorytmu mergesort ma regularną postać (rys. 2.27).



Rys. 2.27. Drzewo rekursji algorytmu mergesort

Algorytm składa się z dwóch kroków i jest w nim używana struktura danych kolejki list oraz liniowy algorytm scalania list uporządkowanych:

(a) przejście ciągu wejściowego  $q$  z utworzeniem kolejki  $Q$  niepustych uporządkowanych list elementów ciągu;

(b) `while |Q| ≥ 2 do`  
`begin {Q jest kolejką list uporządkowanych}`  
`q := front(Q);`  
`Q := pop(Q);`  
`r := front(Q);`  
`Q := pop(Q);`  
`s := merge(q, r);`  
`Q := inject(Q, s)`  
`end`

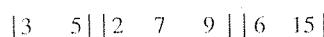
W punkcie (a) pozostał do określenia sposób tworzenia kolejki  $Q$ .

**Możliwość 1:**

$$Q = [[a_1], [a_2], \dots, [a_n]]$$

**Możliwość 2:**

Przechodząc listę wejściową  $q$ , tak długo wstawiamy kolejne elementy na jedną listę, aż natknemy się na element mniejszy od poprzedniego (rys. 2.28).



Rys. 2.28. Podział listy wejściowej na ciąg list uporządkowanych

Jeśli  $|Q| = l$ , to potrzeba  $\log l$  przejść przez kolejkę  $Q$ , żeby otrzymać jedną kolejkę. W każdym przejściu są rozpatrywane wszystkie elementy, a zatem

$$W(n) = A(n) = O(n \log l)$$

co w sytuacji, gdy ciąg wejściowy składa się z długich uporządkowanych fragmentów może wpływać na istotne przyspieszenie działania algorytmu mergesort.

## 2.10. Sortowanie zewnętrzne

Scalanie to podstawowy składnik algorytmów sortowania zewnętrzne, tzn. sortowania listy elementów  $q = [a_1, a_2, \dots, a_n]$ , gdy  $n$  przekracza rozmiar pamięci wewnętrznej. Założymy, że elementy listy  $q$  znajdują się w pewnym pliku pamięci zewnętrznej i że z jednego pliku możemy sprowadzić do bufora pamięci wewnętrznej tylko jedną stronę składającą się – powiedzmy – z  $m$  elementów.

Przez blok będziemy rozumieć dowolną posortowaną część listy. Rozważone przez nas uprzednio algorytmy scalania dają się łatwo zmodyfikować do scalania bloków zapisanych na różnych plikach zewnętrznych (elementy scalanych bloków są dostępne porcjami ściąganyymi do pamięci wewnętrznej).

W wypadku algorytmów sortowania zewnętrzne za główną operację dominującą przyjmuje się wykonanie przesłania stroną między pamięcią wewnętrzną a zewnętrzna.

Algorytmy sortowania zewnętrzne składają się z dwóch głównych kroków:

(a) tworzenia bloków początkowych i ich rozdzielenia do dwóch lub więcej plików;  
 (b) scalania wielofazowego, tzn. powtarzania scalania bloków branych z różnych plików tak długo, aż pozostało jeden blok.

Rozpoczniemy od omówienia kroku drugiego.



Potem kolejno

$P_0:$   $\cup \cup \cup \cup \cup$   
 $P_1:$   $\cup \cup \cup \cup \cup \cup$   
 $P_2:$   $\cup \cup \cup \cup \cup \cup \cup$   
 $P_0:$   $\cup \cup \cup \cup \cup$   
 $P_1:$   $\cup \cup \cup$   
 $P_2:$   $\cup \cup \cup$   
 $P_0:$   $\cup \cup$   
 $P_1:$   $\cup \cup \cup$   
 $P_2:$   
 $P_0:$   
 $P_1:$   $\cup$   
 $P_2:$   $\cup \cup$   
 $P_0:$   $\cup$   
 $P_1:$   $\cup$   
 $P_2:$   
 $a$  na końcu

$P_0:$   
 $P_1:$   $\cup$   
 $P_2:$

Można pokazać (wykorzystując własności liczb Fibonacciego; zad. 2.40), że złożoność scalania wielofazowego z 3 plikami wynosi w przybliżeniu  $1,04n \log(n/m) + O(n)$  porównania i  $2 \cdot 1,04(n/m) \log(n/m) + O(n/m)$  przesłań (każdy element jest uwzględniony w przesłaniach około  $2,08 \log(n/m)$  razy), a zatem jest tylko nieznacznie gorsza niż w wypadku scalania wielofazowego z 4 plikami.

Do omówienia pozostaje jeszcze pierwszy krok sortowania zewnętrzne, a mianowicie tworzenie bloków początkowych. Wykorzystamy tu sortowanie wewnętrzne.

#### METODA I

Powtarzać podane działania aż do osiągnięcia końca pliku wejściowego: wziąć porcję kolejnych  $m$  elementów; posortować je, używając jednego z algorytmów sortowania wewnętrzne; przesłać utworzony blok do odpowiedniego pliku.

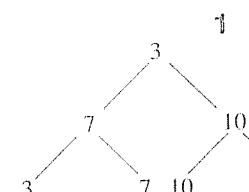
#### METODA II

Do tworzenia bloków użyć kolejki priorytetowej, na przykład drzewa turniejowego (z najmniejszym elementem w korzeniu) jak w algorytmie sortowania turniejowego, przy czym na miejsce usuwanego najmniejszego elementu wstawiać kolejny element ciągu.

Można w ten sposób tworzyć bloki o rozmiarze większym niż  $m$  (średnia długość two-rzonnego bloku wynosi faktycznie  $2m$ ). Wobec nie zmieniającej się struktury drzewa turniejowego można je całe reprezentować w tablicy o rozmiarze  $2m - 1$ .

#### □ PRZYKŁAD:

Załóżmy, że  $m = 4$  i że chcemy dokonać podziału na bloki dla następującego ciągu wejściowego: 3, 7, 10, 1, 12, 4, 8, 6, 2, 11. Budujemy drzewo turniejowe (rys. 2.29) dla pierwszych 4 elementów ciągu.

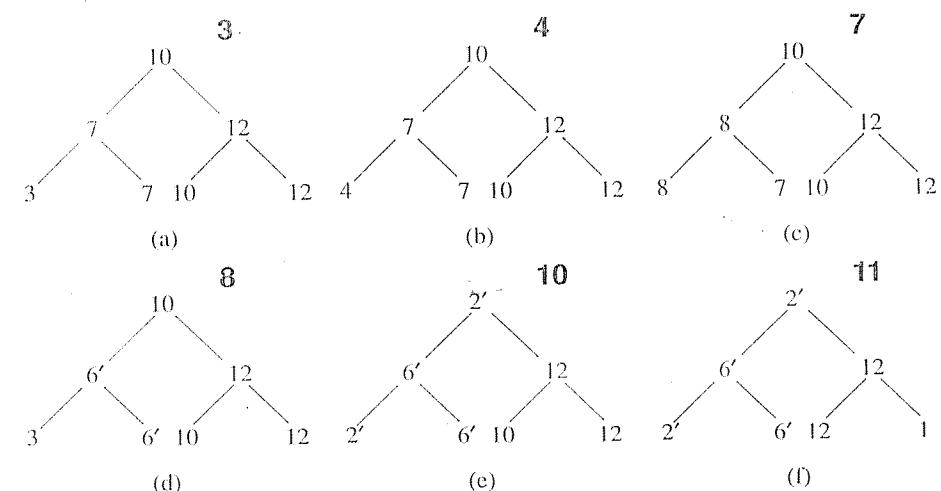


Rys. 2.29. Początkowe drzewo turniejowe

Usuwamy najmniejszy element 1, a na jego miejsce wstawiamy kolejny element 12. Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione na rysunku 2.30a.

Usuwamy najmniejszy element 3, a na jego miejsce wstawiamy kolejny element 4. Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione na rysunku 2.30b.

Usuwamy najmniejszy element 4, a na jego miejsce wstawiamy kolejny element 8. Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione na rysunku 2.30c.



Rys. 2.30. Kolejne drzewa turniejowe

Usuwamy najmniejszy element 7, a na jego miejsce wstawiamy element 6 (ponieważ ostatni element bloku bieżącego  $7 > 6$ , a 6 należy już do następnego bloku, co zaznaczamy za pomocą '). Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione na rysunku 2.30d.

Usuwamy najmniejszy element 8, a na jego miejsce wstawiamy kolejny element 2 (ponieważ ostatni element bloku bieżącego  $8 > 2$ , a element 2 należy już do następnego bloku, co zaznaczamy za pomocą '). Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione na rysunku 2.30e.

Usuwamy element 10, a na jego miejsce wstawiamy kolejny element 11 (ponieważ  $11 > 10$ , a 11 należy do bloku bieżącego). Po wykonaniu modyfikacji otrzymujemy drzewo turniejowe przedstawione rysunku 2.30f.

W ten sposób zostają utworzone dwa bloki:  $[1, 3, 4, 7, 8, 10, 11, 12]$  i  $[2, 6]$ .

Pesymistyczna złożoność czasowa pierwszego kroku sortowania zewnętrznego wynosi  $O((n/m)m \log m) = O(n \log m)$  porównań i  $2(n/m) + O(1)$  przesłań.

(Przy metodzie I zakładamy, że jest stosowany algorytm sortowania wewnętrznego działający w czasie  $O(n \log n)$ ). Wyższość metody II polega na tworzeniu bloków o długości średnio dwukrotnie większej, co prowadzi do zmniejszenia o 1 liczby przejść przez pliki.

Złożoność całego algorytmu sortowania zewnętrznego wynosi zatem  $O(n \log m) + O(n \log(n/m)) = O(n \log n)$  porównań (a więc tyle samo co dla algorytmów sortowania wewnętrznego) i  $O((n/m)(1 + \log(n/m)))$  przesłań.

W wypadku sortowania zewnętrznego czynnikiem decydującym o szybkości działania algorytmu jest liczba operacji zarządzania plikami w pamięci zewnętrznej. Liczbę tą można zmniejszyć, używając większej liczby plików do scalania, jak również drzew turniejowych do tworzenia bloków początkowych.

## Zadania

- Algorytm bubblesort polega na wielokrotnym przejściu sortowanego ciągu z dokonywaniem zamian sąsiednich elementów, gdy nie są one we właściwym porządku. Opracuj i dokonaj analizy tego algorytmu.
- Znajdź przykład na to, że algorytm selectionsort nie jest stabilny.
- Zmodyfikuj algorytm selectionsort tak, żeby był stabilny. Oszacuj, ile dodatkowego czasu i/lub pamięci wymaga taka zmiana.

- ([BK]) Używając metody funkcji tworzących, wyprowadź dla algorytmu insertionsort wzór na oczekiwanyą liczbę porównań i na odchylenie standardowe liczby porównań.
  - Udowodnij, że w algorytmie insertionsort kolejno rozpatrywany element  $a[i]$  może z jednakowym prawdopodobieństwem zająć jedną z  $i$  pozycji w uporządkowanym ciągu  $a[1] \leq a[2] \leq \dots \leq a[i-1]$ .
  - Udowodnij, że w algorytmie quicksort funkcja *partition* dzieli losową permutację liczb  $1, 2, \dots, n$  na dwa losowe podciągi.
  - ([BK]) Używając metody funkcji tworzących, wyprowadź dla algorytmu quicksort wzór na oczekiwanyą liczbę porównań i na odchylenie standardowe liczby porównań.
  - Udowodnij, że w algorytmie quicksort liczba przestawień jest równa co najwyżej połowie liczby porównań.
  - Jak zachowuje się quicksort dla listy równych elementów?
  - Czy algorytm quicksort pozostałby poprawny, gdyby w instrukcjach wewnętrznych *repeat* użyć ostrych nierówności zamiast nieostrzych?
  - Czy w zapisie nierekurencyjnym algorytmu quicksort można zamiast stosu użyć kolejki?
  - Opracuj nierekurencyjną wersję algorytmu quicksort bez stosu zasygnalizowaną na końcu podrozdziału 2.3.
  - Opracuj implementację operacji *partition*, w wyniku której dochodzi do podziału  $A[l:r]$ , dla  $l < r$ , względem elementu  $A[i] = v$  w ten sposób, żeby:
- $$A[l], \dots, A[i-1] < A[i] = \dots = A[j] = v < A[j+1], \dots, A[r]$$
- gdzie  $l \leq i \leq j \leq r$ .
- W algorytmie heapsort w fazie poprawiania kopca element  $a[1]$  jest z reguły mały. Jak można zmodyfikować fazę poprawiania kopca, aby zmniejszyć średnią liczbę porównań, uwzględniając powyższe spostrzeżenie.
  - [K] Algorytm shellsort polega na wielokrotnym zastosowaniu algorytmu insertionsort w następujący sposób: ustalić ciąg przyrostów  $k_1 > k_2 > \dots > k_t = 1$ ; w  $i$ -tej fazie, gdzie  $1 \leq i \leq t$ , sortować podciągi ciągu bieżącego złożone z co  $k_i$ -tego elementu (a więc sortować cały ciąg w ostatniej fazie). Zapisz ten algorytm.

2.16. Opracuj rekurencyjną wersję algorytmu *countsort* (w której przetwarzanie zaczyna się od najbardziej znaczących bitów).

2.17. Niech  $S_1, S_2, \dots, S_k$  będą zbiorami liczb całkowitych z przedziału od 1 do  $n$ , spełniającymi warunek:

$$\sum_{i=1}^k |S_i| = n$$

Opracuj algorytm sortujący każdy zbiór  $S_i$  w łącznym czasie  $O(n)$ .

2.18. Opracuj szybki algorytm sortujący 100 000 ułamków postaci  $p/2^q$ , gdzie  $0 \leq p, q \leq 10$ .

2.19. Niech  $x_1, x_2, \dots, x_n$  będzie ciągiem słów, z których każde ma długość  $k$  nad alfabetem  $m$ -literowym. Opracuj algorytm sortowania, działający w czasie proporcjonalnym do sumy długości tych słów (to znaczy  $O(nk)$ ), traktując  $m$  jako stałą.

2.20. Opracuj algorytm sortowania słów zmiennej długości nad alfabetem o stałej liczbie  $m$  znaków, działający w czasie proporcjonalnym do sumy długości tych słów.

2.21. Niech  $x_1, \dots, x_n$  będzie losowym ciągiem liczb rzeczywistych z przedziału  $[a, b]$ . Uwzględniając rozrzucanie i scalanie z algorytmu *radixsort*, opracuj algorytm sortowania, mający średnią złożoność czasową  $O(n)$ .

2.22. Udowodnij, że do wyznaczenia największego elementu w ciągu trzeba użyć co najmniej  $n - 1$  porównań.

2.23. Udowodnij, że do posortowania  $n$ -elementowego ciągu, który zawiera  $k$  różnych elementów  $a_1, \dots, a_k$  i każdy element  $a_i$  występuje  $n_i$  razy, czyli

$$\sum_{i=1}^k n_i = n$$

trzeba wykonać co najmniej  $\log\left(\frac{n!}{n_1! \dots n_k!}\right)$  porównań.

2.24. Udowodnij, że do wyznaczenia dwóch największych elementów w  $n$ -elementowym ciągu (uporządkowanych) trzeba wykonać co najmniej  $n + \lceil \log n \rceil - 2$  porównań.

2.25. Udowodnij, że do wyznaczenia  $k$  największych elementów w  $n$ -elementowym ciągu (uporządkowanych) trzeba wykonać co najmniej  $(n - k) + \sum_{i=n-k+2}^n \lceil \log i \rceil$  porównań.

2.26. Zaprogramuj algorytmy kolejki priorytetowej dla realizacji na drzewach turniejowych.

2.27. Ułóż algorytmy dla następujących operacji wykonywanych dla kopca:

- (a) *delete*( $i, S$ ):  $S := S - \{a[i]\}$ ;  
 (b) *change*( $i, k, S$ ): *delete*( $a[i], S$ );  $a[i] := k$ ; *insert*( $a[i], S$ );

2.28. **Drzewem lewicowym** nazywamy drzewo binarne, w którym dla każdego wierzchołka  $x$  prawa skrajna ścieżka od  $x$  do liścia jest najkrótszą ścieżką od  $x$  do liścia w poddrzewie wierzchołka  $x$ . Opracuj implementację kolejki priorytetowej, korzystając z drzew lewicowych z elementami wpisymi do nich w porządku tworzącym kopiec. Złożoność każdej z operacji: *insert*, *deletemin* i *union* powinna wynosić  $O(\log n)$ .

2.29. Podaj strukturę danych umożliwiającą wykonywanie w czasie  $O(\log n)$  następujących operacji na początkowo pustym zbiorze  $S$ :

- (a) *insert*( $v, S$ ):  $S := S \cup \{v\}$ ;  
 (b) *deletemedian*( $S$ ): usunięcie z  $S$  mediany elementów zbioru  $S$ ;

przy założeniu, że elementy zbioru  $S$  pochodzą z dowolnego zbioru liniowo uporządkowanego.

2.30. Udowodnij zachodzenie nierówności  $A(n) \leq 4n$  dla algorytmu Hoare'a.

2.31. Zaprogramuj algorytm Bluma-Floyda-Pratta-Rivesta-Tarjana.

2.32. Dla algorytmu z zadania 2.31 udowodnij zachodzenie nierówności  $W(n) \leq 20cn$ .

2.33. Czy w algorytmie z zadania 2.31 podział na 5-elementowe podciągi jest istotny?

2.34. Ułóż algorytm scalania dwóch ciągów uporządkowanych o długościach  $n$  i  $m$ , gdzie  $n \geq m$ , za pomocą  $O(m(\log(n/m) + 1))$  porównań.

2.35. Udowodnij, że w modelu drzew decyzyjnych scalenie dwóch ciągów uporządkowanych o długościach  $n$  i  $m$  wymaga wykonania  $\Omega(m(\log(n/m) + 1))$  porównań.

2.36. Opracuj stabilny algorytm sortowania ciągu zero-jedynkowego. Algorytm powinien działać w miejscu (złożoność pamięciowa –  $O(1)$ ) i mieć pesymistyczną złożoność czasową  $O(n \log n)$ .

- 2.37. Udowodnij, że w modelu drzew decyzyjnych wyznaczenie  $k$ -tego co do wielkości elementu w  $n$ -elementowym ciągu wymaga wykonania  $\Omega\left(\log\left(\binom{n}{k-1}\binom{n-k+1}{1}\right)\right)$  porównań.
- 2.38. Zaprogramuj metodę tworzenia bloków początkowych za pomocą drzewa turajewego.
- 2.39. Czy jest możliwe posortowanie  $n$  elementów za pomocą tylko jednego pliku zewnętrznego. Ile jest potrzebnych operacji wymiany bloków?
- 2.40. Podaj dokładną analizę sortowania wielofazowego z 3 plikami.

## 3 Słowniki

**D**o najczęściej wykonywanych operacji na zbiorze elementów należą wstawianie elementu do zbioru, usuwanie elementu ze zbioru i wyszukiwanie elementów o zadanych własnościach (na przykład najmniejszego elementu lub elementu o podanej wartości klucza). Struktura danych umożliwiająca wykonywanie takich operacji nazywa się **słownikiem**. Ważnymi przykładami zastosowań słownika są bazy danych, tablice identyfikatorów w kompilatorach i komputerowe słowniki języków naturalnych.

Przez **problem słownika** rozumie się następujące zadanie: podać strukturę danych umożliwiającą wykonywanie następujących operacji na zbiorze skończonym  $S$ :

- construct( $S$ )*:  $S := \emptyset$ ;
- search( $v, S$ )*: sprawdzenie, czy element  $v$  należy do  $S$ ; jeśli tak, wyznaczenie miejsca, gdzie znajduje się  $v$ ;
- insert( $v, S$ )*:  $S := S \cup \{v\}$ ;
- delete( $v, S$ )*:  $S := S - \{v\}$ .

### UWAGA!

Specyfikacja operacji *search* odbiega od pozostałych specyfikacji, gdyż odwołuje się do konkretnej implementacji zbioru  $S$ , w której ma sens mówienie o miejscu zapisania elementu zbioru.

W programach użytkowych element zbioru jest zwykle rekordem informacji, mającym swoją wewnętrzną strukturę. Wyszukiwanie elementu odbywa się wtedy względem pewnego atrybutu rekordu nazywanego **kluczem**. Gdy jest więcej rekordów o tym samym kluczu niż jeden, są one na ogół przechowywane na jednej liście. W naszym sformułowaniu problemu wyszukiwania elementu nie ma z tym kłopotów — zakładamy, że  $S$  jest zbiorem.

### 3.1.

## Implementacja listowa nieuporządkowana

Najprostsza implementacja listowa polega na zapisie elementów zbioru  $S = \{a_1, a_2, \dots, a_n\}$  w dowolnej kolejności na liście  $l = [a_1, a_2, \dots, a_n]$ . Z kolei lista może być przechowywana albo w tablicy, albo w strukturze dowiązanej. Dla każdej operacji (b)-(d) otrzymujemy (zad. 3.1)

$$W(n) = A(n) = O(n)$$

(zakładając wyszukiwanie sekwencyjne elementu na liście).

Zaletą jest tu prostota struktury danych i niewielka ilość dodatkowej pamięci; wadą natomiast jest długi czas działania. Implementację tę stosuje się, gdy dane są zawsze przetwarzane sekwencyjnie w dowolnej kolejności.

Warto zwrócić uwagę na tę odmianę bezpośredniej implementacji listowej, w której element będący argumentem operacji *search* lub *insert* jest ustawiany na początku listy. Gdy rozkład częstości dostępu do elementu nie jest równomierny, ułożenie elementów na liście staje się zgodne z rozkładem częstości; elementy będące często argumentami operacji znajdują się na ogół blisko początku listy i w związku z tym dostęp do nich jest szybki (zad. 3.2). Implementacja ta nosi nazwę **samoorganizujących się list**.

### 3.2.

## Implementacja listowa uporządkowana

Elementy na liście mogą być też uporządkowane. Wygodnie jest wówczas używać tablicy. Elementy zbioru  $S$  zapisujemy w tablicy  $a[1..n]$  tak, że  $a[1] < a[2] < \dots < a[n]$ . Wygodnie jest rozszerzyć tablicę  $a$  o dwie pozycje skrajne:  $a[0] = -\infty$  i  $a[n+1] = +\infty$ .

Operację *search* można zrealizować, używając metody „dziel i zwyciężaj”. Aby sprawdzić, czy  $v$  znajduje się w przedziale  $a[l] < a[l+1] < \dots < a[r]$ , wiedząc że  $a[l] \leq v < a[r]$ , należy wybrać indeks  $k$ , gdzie  $l < k < r$ , porównać  $v$  z  $a[k]$  i w zależności od wyniku porównania albo szukać  $v$  w  $a[l] < \dots < a[k]$ , albo w  $a[k] < \dots < a[r]$ . Niech  $cut(v, l, r)$  będzie funkcją wyznaczającą indeks  $k$  (zakładamy, że  $l+1 < r$ ).

```
function search(v : T) : integer;
var k, l, r : integer;
begin {n ≥ 1}
  l := 0; r := n + 1;
  repeat {a[l] ≤ v < a[r] ∧ r > l + 1}
    k := cut(v, l, r);
    {l < k < r}
    if v < a[k] then r := k else l := k
  until r = l + 1;
end;
```

### 3.2. Implementacja listowa uporządkowana

```
until r = l + 1;
if v = a[l] then search := l else search := 0
end search;
```

Rozważymy dwie metody wyboru funkcji *cut*.

#### METODA I

Jest to tzw. **wyszukiwanie binarne**.

$$cut(v, l, r) = \lfloor (l + r)/2 \rfloor$$

Załóżmy, że chcemy wykonać  $search(5, S)$  dla elementów  $a$  zapisanych w tabeli 3.1a. Otrzymamy wyniki przedstawione w tabeli 3.1b.

Tabela 3.1. Dane i wyniki dla algorytmu wyszukiwania

$a$	0	1	2	3	4	5	6	7	8	9	10	11
	$-\infty$	0	1	5	6	7	8	9	16	34	56	$+\infty$

(a)

$l$	0	0	2	3	3
$r$	11	5	5	5	4
$k$	5	2	3	4	

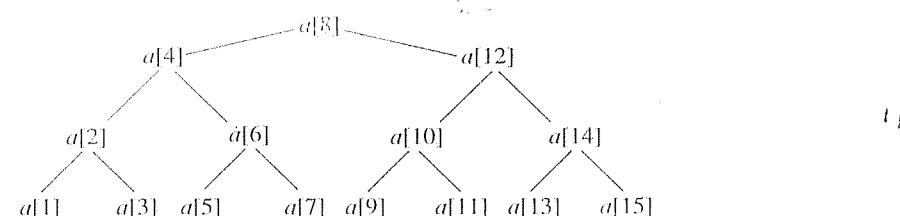
(b)

Do Ciebie należy (zad. 3.3) wykazanie, że dla operacji *search*

$$W(n) = \log n + O(1)$$

$$A(n) = \log n + O(1)$$

Działanie wyszukiwania binarnego można zobrazować za pomocą drzewa binarnego, w wierzchołkach którego znajdują się elementy tablicy  $a$ , ułożone w kolejności porównywania ich z szukanym elementem  $v$ . Dla  $n = 15$  otrzymujemy drzewo przedstawione na rysunku 3.1. Drzewo to nosi nazwę drzewa obliczeń funkcji *search*.



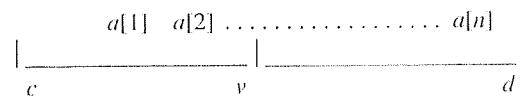
Realizacja funkcji *search* polega na przejściu jednej gałęzi od korzenia do liścia, z jednocześnie porównywaniem  $v$  z wartościami zapisanymi w wierzchołkach. Gdy  $v$  jest mniejsze, idziemy do lewego następnika danego wierzchołka w drzewie, a w przeciwnym razie – do prawego. Obliczenie funkcji *search* to ciąg par  $(l_1, v_1), \dots, (l_K, v_K)$ , gdzie  $l_j$  oznacza instrukcję, a  $v_j$  wartościowanie zmiennych.

Ponieważ każde drzewo binarne o  $n$  wierzchołkach ma wysokość nie mniejszą niż  $\lfloor \log n \rfloor$ , przedstawiona metoda wyszukiwania binarnego jest optymalna w przypadku pesymistycznym. Jeśli natomiast chodzi o przypadek oczekiwany, istnieją lepsze metody wyszukiwania, w których brana jest pod uwagę wartość  $v$  w stosunku do przedziału poszukiwań.

## METODA II

Jest to tzw. **wyszukiwanie interpolacyjne**. Niech  $S \subseteq (c, d)$ , gdzie  $c, d$  to liczby rzeczywiste,  $c < d$ ,  $|S| = n$ . Założymy, że elementy zbioru  $S$  są dobierane z rozkładem równomiernym w przedziale  $(c, d)$ . Niech  $v \in (c, d)$ . Rozważmy zmienną losową  $X$ , oznaczającą liczbę elementów zbioru  $S$ , które są  $\leq v$ , (tzn. jeśli  $v \in S$ , to  $a[X] = v$ ) (rys. 3.2). Prawdopodobieństwo, że element zbioru  $S$  jest  $\leq v$ , wynosi

$$q = \frac{v - c}{d - c}$$



Rys. 3.2. Położenie elementu w przedziale  $(c, d)$

Wybór elementów zbioru  $S$  następuje zgodnie z rozkładem Bernoulliego (mając element  $v$ , losujemy kolejno  $a[1], \dots, a[n]$  z prawdopodobieństwem sukcesu  $q$ ), a zatem

$$\text{ave}(X) = qn$$

$$\text{var}(X) = q(1 - q)n \leq \frac{1}{4}n$$

Wynika stąd, że z dużym prawdopodobieństwem wartość  $v$  jest położona w okolicy elementu  $a[\lceil qn \rceil]$  w tablicy  $a$ . W wypadku wyszukiwania interpolacyjnego kładziemy

$$q[0] = c, a[n + 1] = d$$

Wartość funkcji *cut* obliczamy w następujący sposób (zakładamy, że  $l + 1 \leq r$ ):

$$s := l + \left\lceil \frac{v - a[l]}{a[r] - a[l]} (r - l) \right\rceil;$$

$$\text{cut}(v, l, r) := \text{if } s = l \text{ then } l + 1 \text{ else if } s = r \text{ then } r - 1 \text{ else } s;$$

## 3.2. Implementacja listowa uporządkowana

Pesymistyczna złożoność czasowa i oczekiwana złożoność czasowa operacji *search* są następujące:

$$W(n) = n + O(1)$$

$$A(n) = \log \log n + O(1)$$

Załóżmy, że chcemy wykonać  $\text{search}(5, S)$  dla elementów  $a$  zapisanych w tabeli 3.2a. Otrzymamy wyniki przedstawione w tabeli 3.2b.

Tabela 3.2. Dane i wyniki dla algorytmu wyszukiwania interpolacyjnego

$a$	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	5	6	7	8	9	10	12	14	15

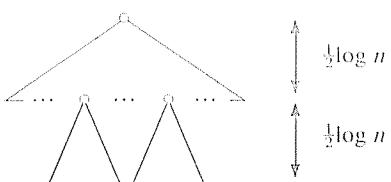
(a)

$l$	0	0	3
$r$	11	4	4
$k$	4	3	

(b)

Dowód wzoru na  $A(n)$  dla wyszukiwania interpolacyjnego jest skomplikowany matematycznie i nie przytoczymy go tutaj. Rozważmy za to pewną modyfikację wyszukiwania interpolacyjnego, dla której łatwiej jest udowodnić, że  $A(n) = O(\log \log n)$ . Najpierw pokażemy, jak można intuicyjnie wyprowadzić algorytm, korzystając z drzewa obliczeń funkcji *search*, a następnie sformułujemy algorytm, korzystając z reprezentacji tablicowej.

Wróćmy do drzewa obliczeń funkcji *search*. Założymy, że ma ono wysokość rzędu logarytmicznego. Obliczenie funkcji *search* polega na przejściu ścieżki od korzenia do liścia. Aby przejść tę ścieżkę w czasie podwójnie logarytmicznym, należy zastosować przejście binarne, jak w wyszukiwaniu binarnym (dzieląc za każdym razem na połowę długość odcinka, który pozostał do przejścia). Problem jednak polega na tym, że ta ścieżka nie jest dana z góry. Wzór interpolacyjny pozwala nam zbliżać się do szukanych wierzchołków na tej ścieżce z dodatkowym kosztem, którego wartość oczekiwana jest stała. Od korzenia będziemy schodzić na poziom środkowy drzewa, a następnie będziemy porównywać  $v$  z pewną liczbą korzeni dolnych poddrzew, która okaże się stała w przypadku oczekiwany, po czym będziemy powtarzać szukanie elementu  $v$  w odpowiednim dolnym poddrzewie (rys. 3.3).



Rys. 3.3. Podział drzewa obliczeń funkcji *search*

Zauważmy, że w poddrzewie o wysokości  $(1/2)\log n$  jest około  $2^{0.5\log n} = \sqrt{n}$  wierzchołków. Udowodnimy następujący lemat.

### Lemat 3.1.

Oczekiwana liczba porównań potrzebnych do wyznaczenia właściwego dolnego poddrzewa jest  $\leq 2,5$ .

Zanim przejdziemy do dowodu lematu, zauważmy, że wynika z niego następująca zależność:

$$\begin{cases} A(1) = 1, A(2) = 2 \\ A(n) \leq 2,5 + A(\lceil \sqrt{n} \rceil) \quad \text{dla } n \geq 3 \end{cases}$$

Stąd  $A(n) \leq 2,5 \lceil \log \log n \rceil + 2$  dla  $n \geq 1$ .

**Dowód:** Najpierw określmy nieco dokładniej (używając reprezentacji tablicowej) naszkicowany powyżej algorytm.

```
procedure quadratic binary search {szukamy  $v$  w  $a[0..n+1]$ ,  $v \in (c, d)$ }
    oblicz  $p = \frac{v - c}{d - c} n$ ;
    jeśli  $v \geq a_{\lceil p \rceil}$ , to
        wyznacz najmniejsze  $i$  takie, że  $v \leq a_{\lceil p + i \sqrt{n} \rceil}$ ;
        powtórz rekurencyjnie szukanie  $v$  w  $a[\lceil p + (i-1) \sqrt{n} \rceil \dots \lceil p + i \sqrt{n} \rceil]$ ;
    jeśli  $v < a_{\lceil p \rceil}$ , to
        wyznacz najmniejsze  $i$  takie, że  $v \geq a_{\lceil p - i \sqrt{n} \rceil}$ ;
        powtórz rekurencyjne szukanie  $v$  w  $a[\lceil p - i \sqrt{n} \rceil \dots \lceil p - (i-1) \sqrt{n} \rceil]$ ;
```

Naszym celem jest wyznaczenie wartości oczekiwanej zmiennej losowej  $Y$  oznaczającej liczbę prób (rozpatrywanych indeksów  $p$ ,  $p \pm \sqrt{n}$ , ...) prowadzących do wyznaczenia właściwego dolnego poddrzewa ( $\sqrt{n}$  elementów). Niech  $p = qn = \text{ave}(X)$ , gdzie  $X$  określa indeks elementu  $v$  w tablicy  $a[1..n]$ , a  $p$  jest indeksem interpolującym  $X$ . Zmieniąca losowa  $Y$  wiąże się ze zmieniącą losową  $X$  w następujący sposób:

$$Y \geq j \equiv |X - p| \geq (j-2)\sqrt{n} \quad \text{dla } j \geq 3$$

czyli

$$\Pr(Y \geq j) = \Pr(|X - p| \geq (j-2)\sqrt{n})$$

oraz

$$\Pr(Y \geq 1) = \Pr(Y \geq 2) = 1$$

### 3.3. Drzewa poszukiwań binarnych

Do zmiennej losowej  $X$  stosujemy nierówność Czebyszewa ( $t = (j-2)\sqrt{n}$ )

$$\Pr(|X - \text{ave}(X)| \geq t) \leq \frac{\text{var}(X)}{t^2}$$

Dla  $j \geq 3$  mamy zatem

$$\Pr(Y \geq j) \leq \frac{\text{var}(X)}{(j-2)^2 n} \leq \frac{1}{4} n \frac{1}{(j-2)^2 n} = \frac{1}{4(j-2)^2}$$

Wracając do tego, co jest naszym celem, a mianowicie oszacowania  $\text{ave}(Y)$ , otrzymujemy

$$\begin{aligned} \text{ave}(Y) &= \sum_{j \geq 1} j \Pr(Y = j) = \sum_{j \geq 1} \sum_{l=1}^j \Pr(Y = j) = \sum_{l \geq 1} \sum_{j \geq l} \Pr(Y = j) = \\ &= \sum_{l=1}^2 \Pr(Y \geq l) + \sum_{l \geq 3} \frac{1}{4(l-2)^2} = 2 + \frac{1}{4} \sum_{j \geq 1} \frac{1}{j^2} \leq 2 + \frac{1}{4} \cdot \frac{\pi^2}{6} \leq 2,5 \end{aligned}$$

cbdo

Pozostałe operacje dotyczące słownika, a mianowicie *insert* i *delete* w implementacji za pomocą tablic uporządkowanych (niezależnie od wyboru funkcji *cut*) są nadal pracochłonne:

$$W(n) = A(n) = O(n)$$

### 3.3.

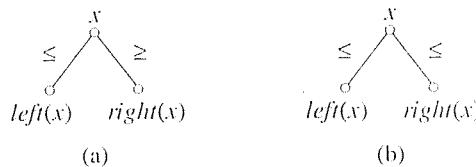
## Drzewa poszukiwań binarnych

Uogólnieniem wyszukiwania elementu w tablicy uporządkowanej jest wyszukiwanie elementu w tzw. drzewach poszukiwań binarnych, czyli w skrócie BST (od ang. *Binary Search Trees*). Przez wzbogacenie struktury danych uzyskujemy możliwość szybszego wykonywania operacji wstawiania i usuwania elementu ze zbioru.

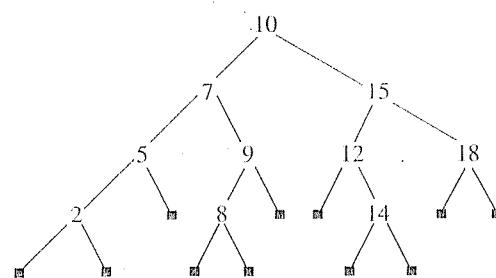
Zakładamy, że każdy węzeł  $x$  drzewa, będący obiektem typu *node*, ma trzy atrybuty:

$x$ : *left*( $x$ ), *key*( $x$ ), *right*( $x$ )

Elementy w kopcu są uporządkowane zgodnie z porządkiem kopcowym (rys. 3.4a). W drzewach BST mamy natomiast do czynienia z porządkiem symetrycznym. W porządku tym dla każdego węzła  $x$  jest spełniony następujący warunek: jeśli węzeł  $y$  leży w lewym poddrzewie  $x$ , to  $\text{key}(y) \leq \text{key}(x)$ ; jeśli  $y$  leży w prawym poddrzewie  $x$ , to  $\text{key}(x) \leq \text{key}(y)$  (rys. 3.4b).



Rys. 3.4. (a) Porządek kopekowy; (b) porządek symetryczny



Rys. 3.5. Drzewo poszukiwań binarnych z zaznaczonymi wierzchołkami zewnętrzny

**Drzewem poszukiwań binarnych (drzewem BST)** nazywamy dowolne drzewo binarne, w którym elementy zbioru są wpisane do wierzchołków zgodnie z porządkiem symetrycznym (rys. 3.5).

Wierzchołki zaznaczone na rysunku 3.5 przez ■ to **wierzchołki zewnętrzne**. W strukturze dowiązaniowej są one reprezentowane przez nil. Wierzchołkom zewnętrzny odpowiadają przedziały wartości, na które zostaje podzielona przestrzeń wszystkich kluczy przez klucze obecne w drzewie (czyli elementy znajdujące się w drzewie). Wstawiając nowy element, umieszczamy go w wierzchołku zewnętrzny reprezentującym przedział wartości, do którego należy dany element.

Operacja *search* dla drzewa BST jest uogólnieniem funkcji *search* dla tablicy uporządkowanej.

```
function search(v : T; r : node) : node;
{r jest dowolnym typem liniowo uporządkowanym; r jest korzeniem drzewa BST}
var x : node;
begin
  x := r;
  while (x ≠ nil) and (key(x) ≠ v) do
    if v < key(x) then x := left(x) else x := right(x);
    search := x;
  {jeśli element v znajduje się w drzewie, to x ≠ nil i key(x) = v;
  jeśli elementu v nie ma w drzewie, to x = nil}
end search;
```

### UWAGA

Zamiast nil można użyć wierzchołka-wartownika, powiedzmy o nazwie *sentinel*. Wtedy przed pętlą while należy wstawić instrukcję *key(sentinel) := v*. Warunek while uprości się do *key(x) ≠ v*. Zapisanie algorytmów słownika z użyciem wartownika pozostawiamy Ci jako ćwiczenie.

Pierwszą czynnością w operacjach *insert* i *delete* jest wykonanie operacji *search*. Potrzebny jest przy tym poprzednik końcowego węzła x. W tym celu zmodyfikujemy naszą operację *search*.

```
function search(v : T; r : node; var y : node) : node;
{T jest dowolnym typem liniowo uporządkowanym; r jest korzeniem drzewa BST; y jest poprzednikiem wierzchołka wyszukiwanego przez search}
var x : node;
begin
  x := r; y := nil;
  while (x ≠ nil) and (key(x) ≠ v) do
    begin
      y := x;
      if v < key(x) then x := left(x) else x := right(x);
    end;
    search := x
  end search;
```

Operacja *insert* polega na wykonaniu *search(v, r, y)*, utworzeniu nowego wierzchołka x, wstawieniu tam elementu v i dowiązaniu x do y.

```
procedure insert(v : T; var r : node);
var x, y : node;
begin
  if search(v, r, y) = nil then
    begin
      new(x);
      left(x) := nil; key(x) := v; right(x) := nil;
      if y = nil then r := x else
        if v < key(y) then left(y) := x else right(y) := x
    end
  end insert;
```

Operacja *construct* jest trywialna. Oto ona:

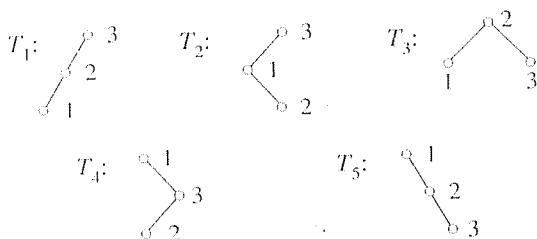
```
procedure construct(var r : node);
begin
  r := nil
end construct;
```

Drzewo przedstawione na rysunku 3.5 można skonstruować za pomocą następującego ciągu operacji:

*construct(S); insert(10, S); insert(7, S); insert(5, S); insert(9, S); insert(8, S); insert(2, S); insert(15, S); insert(12, S); insert(14, S); insert(18, S);*

To samo drzewo można też utworzyć za pomocą innych ciągów operacji. Zauważmy, że przestrzeń probabilistyczne drzew losowych BST i drzew BST tworzonych przez losowe permutacje są różne. Dla  $n = 3$  na przykład jest pięć różnych drzew BST (rys. 3.6).

W modelu drzew losowych BST każde drzewo ma więc prawdopodobieństwo  $1/5$ , natomiast w modelu permutacyjnym drzewo  $T_3$  ma prawdopodobieństwo  $1/3$ , a pozostałe  $1/6$ . Udowodniono, że w modelu drzew losowych BST oczekiwana wysokość  $n$ -wierzchołkowego drzewa BST jest  $O(\sqrt{n})$ . W naszej analizie przyjmiemy model permutacyjny drzew BST.



Rys. 3.6. Pięcioelementowe drzewa BST

Operacją *delete* zajmiemy się za chwilę, a teraz przeanalizujemy czas działania operacji *search* (a więc także *insert*). Czas działania operacji *search* jest proporcjonalny do głębokości wierzchołka  $x$  (wewnętrznego lub zewnętrznego), będącego wynikiem funkcji *search*. Stąd

$$W(n) = O(n)$$

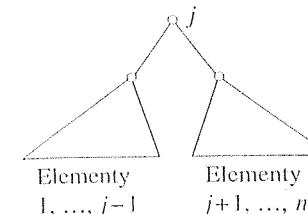
( $n$  oznacza liczbę wierzchołków w drzewie BST).

Aby obliczyć  $A(n)$ , oznaczmy przez  $G(n)$  oczekiwana sumę głębokości wierzchołków zewnętrznych w drzewie BST, utworzonym z drzewa pustego przez wykonanie ciągu  $n$  operacji *insert*

$$\text{insert}(a_1, S); \text{ insert}(a_2, S); \dots; \text{ insert}(a_n, S);$$

gdzie  $a_1, a_2, \dots, a_n$  jest losową permutacją liczb  $1, 2, \dots, n$ . Ponieważ lewe i prawe poddrzewa od korzenia drzewa losowego BST są losowe (zad. 3.6), mamy (rys. 3.7)

$$\begin{cases} G(0) = 0 \\ G(n) = (n+1) + \frac{1}{n} \sum_{j=1}^n (G(j-1) + G(n-j)) \end{cases}$$



Rys. 3.7. Struktura  $n$ -wierzchołkowego drzewa BST

Otrzymane równanie jest takie jak dla algorytmu quicksort. Jego rozwiążanie wygląda zatem tak:

$$G(n) = 1.4n \log n + O(n)$$

Oczekiwana złożoność czasowa operacji *insert* i *search* w sytuacji, kiedy nie ma elementu w zbiorze, wynosi

$$A(n) = \frac{G(n)}{n+1} = 1.4 \log n + O(1)$$

Aby wyznaczyć oczekiwana złożoność czasową operacji *search* w przypadku, gdy szukany element znajduje się w drzewie, wprowadźmy dwie zmienne losowe:  $Z_n$  oznaczającą sumę głębokości wierzchołków zewnętrznych ( $G(n) = \text{ave}(Z_n)$ ) i  $W_n$  oznaczającą sumę głębokości wierzchołków wewnętrznych.

Przez indukcję względem liczby wierzchołków w drzewie możemy pokazać (zad. 3.7), że

$$Z_n = W_n + 2n$$

Stąd

$$\text{ave}(Z_n) = \text{ave}(W_n) + 2n$$

Oczekiwana złożoność czasowa operacji *search* w przypadku, gdy element znajdzie się w drzewie, wynosi zatem

$$A(n) = \frac{\text{ave}(W_n)}{n} = \frac{\text{ave}(Z_n)(n+1)}{(n+1)n} - 2 = 1.4 \log n - O(1)$$

czyli tyle samo co w przypadku, gdy szukanego elementu nie ma w drzewie.

Nietrudno też wykazać (zad. 3.8), że

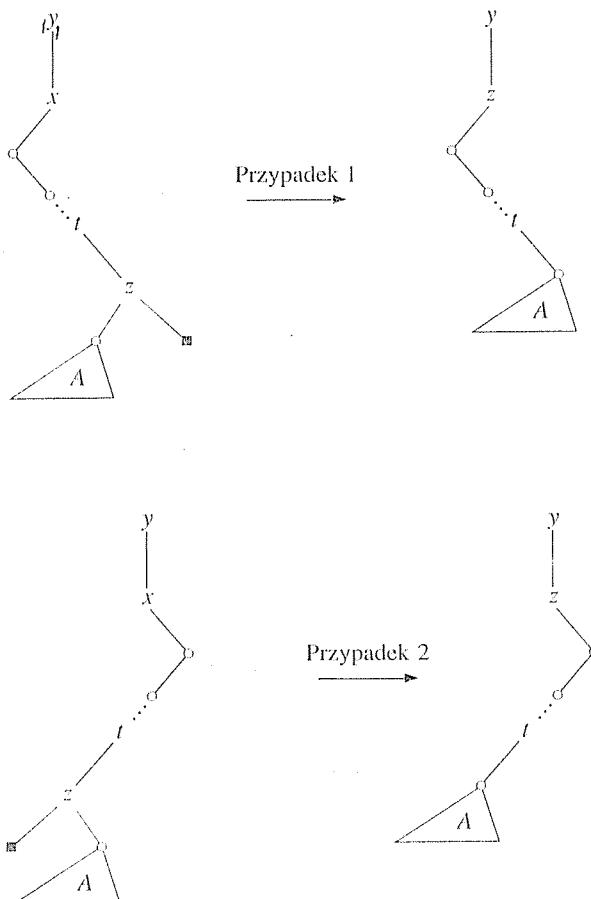
$$\delta(n) = O(\sqrt{\log n})$$

Zajmiemy się teraz operacją

$$\text{delete}(v, S):: S := S - \{v\}$$

Usunięcie elementu ze zbioru wiąże się z usunięciem wierzchołka z drzewa. Trudność polega na tym, że nie można swobodnie usuwać wierzchołków z drzewa, gdy mają one następców. Można sobie z tym poradzić, zastępując wierzchołek  $x$  zawierający element  $v$  wierzchołkiem zawierającym albo element bezpośrednio poprzedzający  $v$  w zbiorze  $S$ , albo element bezpośrednio następujący po  $v$  (rys. 3.8). Aby zagwarantować losową postać drzewa BST po usunięciu wierzchołka, losujemy, który wierzchołek wybrać. Fizycznie usuwamy wierzchołek, który ma co najwyżej jeden wewnętrzny następnik (co najmniej jeden z następców jest wierzchołkiem zewnętrznym reprezentowanym przez nil).

Oto algorytm delete. Jego tekst jest dosyć długi ze względu na dużą liczbę możliwych przypadków.

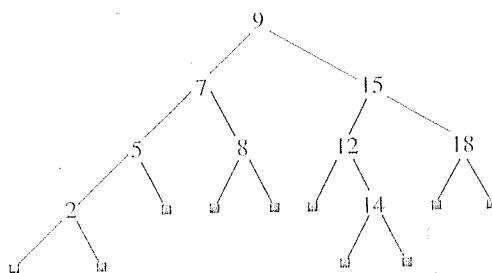


Rys. 3.8. Operacja usuwania elementu z drzewa BST

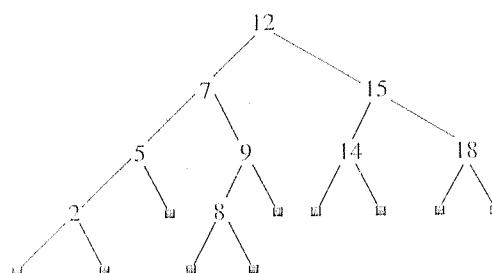
```

procedure delete(v : T; var x : node);
var x, y, z, t : node;
    b : 0..1;
begin
    x := search(v, x, y);
    if x ≠ nil then
        begin
            if (left(x) = nil)or(right(x) = nil) then
                begin
                    if (left(x) = nil)and(right(x) = nil) then z := nil else
                        if left(x) = nil then z := right(x)
                        else z := left(x);
                    if y = nil then r := z else
                        if x = left(y) then left(y) := z else right(y) := z
                end
            else
                begin {left(x) ≠ nil i right(x) ≠ nil}
                    b := random(2);
                    {jeśli b = 0, to w miejsce v wstawiamy element
                     bezpośrednio poprzedzający v w zbiorze S;
                     jeśli b = 1, to w miejsce v wstawiamy element
                     bezpośrednio następujący po v w zbiorze S}
                    if b = 0 then
                        begin
                            z := left(x);
                            if right(z) = nil then left(x) := left(z) else
                                begin
                                    repeat
                                        t := z;
                                        z := right(z)
                                    until right(z) = nil;
                                    right(t) := left(z)
                                end
                            end
                    end else
                        begin
                            z := right(x);
                            if left(z) = nil then right(x) := right(z) else
                                begin
                                    repeat
                                        t := z;
                                        z := left(z)
                                    until left(z) = nil;
                                    left(t) := right(z)
                                end
                            end
                        end;
                    key(x) := key(z)
                end
        end
    end delete;

```



Rys. 3.9. Wynik usunięcia elementu 1  
przez przesunięcie elementu 9



Rys. 3.10. Wynik usunięcia elementu 1  
przez przesunięcie elementu 12

Aby na przykład wykonać operację *delete(10, S)* względem drzewa BST podanego na rysunku 3.5, możemy w miejscu zwalnianie przez 10 przenieść 9, uzyskując drzewo widoczne na rysunku 3.9, albo 12, uzyskując drzewo widoczne na rysunku 3.10.

Pozymistyczna złożoność czasowa przedstawionego algorytmu `delete` wynosi oczywicie

$$W(\mu) \equiv O(n)$$

Z badań wynika natomiast, że oczekiwana złożoność czasowa

$$A(\mu) \equiv O(\log \mu)$$

choćiąż nie jest to fakt matematycznie udowodniony

Możliwa jest prostsza wersja operacji *delete*, nazywana **opóźnionym usuwaniem**. Zamiast pozbywać się wierzchołka  $x$  zawierającego usuwany element  $v$ , przyjmujemy wiele rzekomiego  $x$  za „usunięty” i pozostawiamy go w drzewie. Oczekiwana złożoność czasowa wszystkich operacji słownika wynosi zatem  $O(\log m)$ , gdzie  $m$  jest liczbą wykonanych operacji *insert*. Gdy liczba pozostawionych w drzewie „śmieci” staje się zbyt duża (lub gdy jeden z wierzchołków znajduje się w zbyt dużej odległości od korzenia (na przykład  $\geq c \log n$  dla pewnej stałej  $c$ )), należy przechodząc całe drzewo, utworzyć z nie usuniętych elementów nowe drzewo BST. Można się postarać, żeby miało ono jak najmniejszą wysokość (żeby było zupełnym drzewem binarnym). Zapisanie algorytmu rekonstruującego zupełne drzewo BST pozostawiamy Ci jako ćwiczenie (zad. 3.11).

### 3.3.1. Przewa AVI

Drzewa BST są prostą i efektywną implementacją słownika w przypadku oczekiwany. W przypadku pesymistycznym każda z operacji *search*, *insert* i *delete* ma jednak złożoność  $O(n)$ . Chcąc uzyskać czas działania  $O(\log n)$ , trzeba dodatkowo zadbać, żeby drzewa BST pozostawały w postaci gwarantującej wysokość  $O(\log n)$ , gdzie  $n$  jest liczbą wierzchołków. Istnieje wiele odmian takich drzew. Najprostsze z nich to drzewa AVL.

Drzewo BST jest drzewem AVL wtedy, kiedy dla każdego wierzchołka wysokości dwóch jego poddrzew różnią się co najwyżej o 1.

三

### Lemat 3.2

Wysokość drzewa AVL o  $n$  wierzchołkach ( $n \geq 1$ ) jest nie większa niż  $1,45 \log n$ .

**Dowód:** Niech  $T_h$  będzie drzewem AVL o wysokości  $h$ , które ma najmniejszą możliwą liczbę wierzchołków wśród drzew AVL o wysokości  $h$ . Niech  $n_h$  będzie liczbą węzłów wewnętrznych w  $T_h$ , a  $m_h$  liczba węzłów zewnętrznych. Oczywiście

Rys. 3.11. Konstrukcja minimalnych drzew AV

$m_h = n_h + 1$ . Zauważmy, że (rys. 3.11)  $m_h = F_{h+3}$  dla  $h \geq 1$ , gdzie  $F_h$  jest liczbą Fibonacciego o numerze  $h$ .

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_h = F_{h-1} + F_{h-2} \quad \text{dla } h \geq 2 \end{cases}$$

Z własnością liczb Fibonacciego wynika, że

$$\Theta^{h-2} \leq F_h \leq \Theta^{h-1} \quad \text{dla } h \geq 1$$

gdzie  $\Theta = \frac{1}{2}(1 + \sqrt{5})$ . Stąd

$$\Theta^{h+1} \leq m_h \quad \text{dla } h \geq 1$$

Mamy zatem

$$h + 1 \leq \log_\Theta m_h \leq \log_\Theta(n_h + 1) \leq \log_\Theta(n + 1) \leq 1,45 \log n + 1 \quad \text{dla } h \geq 1$$

cbdo

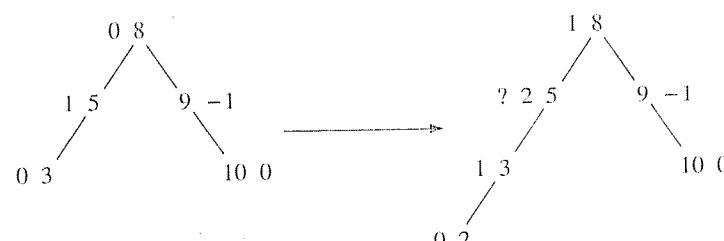
Poza atrybutami *left*, *key* i *right* każdy wierzchołek drzewa AVL ma również atrybut

$$x: bf(x) = h_L(x) - h_R(x)$$

gdzie  $h_L(x)$  i  $h_R(x)$  to odpowiednio wysokość lewego i prawego poddrzewa wierzchołka  $x$ . Z definicji drzewa AVL wynika, że dla każdego wierzchołka  $x$

$$bf(x) \in \{-1, 0, +1\}$$

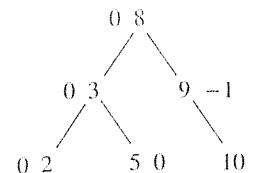
Realizacja operacji *search* dla drzewa AVL jest taka sama jak dla zwykłego drzewa BST (w wyniku *search* drzewo się nie zmienia). Natomiast operacje *insert* i *delete* dla drzewa AVL przebiegają z początku tak samo jak dla drzewa BST, po czym następuje faza przywracania struktury drzewa AVL, jeśli została zaburzona (w wyniku *insert* może się zwiększyć wysokość drzewa, a w wyniku *delete* zmniejszyć). Wstawiając na przykład



Rys. 3.12. Wstawianie elementu 2 do drzewa AVL

element 2 do drzewa AVL widocznego na rysunku 3.12 (trybuty *bf* są podane obok wierzchołków), otrzymujemy drzewo BST, w którym jeden z wierzchołków ma atrybut *bf* równy 2.

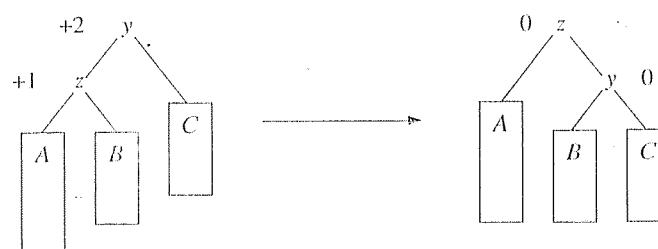
Należy dokonać przesunięcia w drzewie wierzchołków, czyli tzw. rotacji wokół wierzchołka 5, likwidując nadmierną różnicę wysokości poddrzew (rys. 3.13).



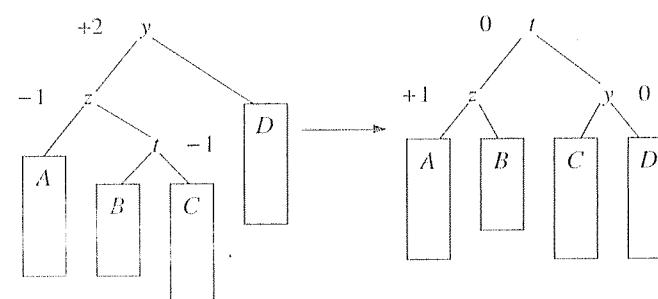
Rys. 3.13. Wynik poprawienia drzewa AVL z rysunku 3.12

Naszkicujemy teraz operację *insert(v, S)* dla drzewa AVL. Po wstawieniu nowego węzła  $x$  za pomocą zwykłego algorytmu *insert* dla drzew BST przesuwamy się z powrotem po ścieżce od  $x$  w stronę korzenia, dokonując odpowiednich zmian atrybutu *bf* (wierzchołki tej ścieżki stają się „cięższe”) do chwili napotkania takiego wierzchołka  $y$ , że albo (a)  $y$  jest korzeniem, a nowa wartość  $bf(y)$  jest różna od 2 i -2, albo (b) nowa wartość  $bf(y) = 0$ , albo (c) nowa wartość  $bf(y) = 2$  lub -2.

W przypadku (c) w wierzchołku  $y$  następuje zaburzenie struktury drzewa AVL i należy dokonać odpowiednich przesunięć wierzchołków w drzewie, aby tę strukturę przywrócić.



(a) Rotacja pojedyncza w  $y$   
(przypadek symetryczny, gdy  $bf(y) = +2$ ,  $bf(z) = +1$ )



(b) Rotacja podwójna w  $y$   
(przypadek symetryczny, gdy  $bf(y) = -2$ ,  $bf(z) = -1$ , wartości  $bf(t) \in \{-1, +1\}$ )

Rys. 3.14. Rotacje drzew AVL

Z dokładnością do symetrii mamy do czynienia z jedną z dwóch sytuacji przedstawionych na rysunku 3.14; obok zapisujemy odpowiednie operacje (nazywane rotacjami) przywracające strukturę AVL w wierzchołku  $y$ .

### Lemat 3.3.

Rotacja pojedyncza i rotacja podwójna prowadzą od drzewa BST do drzewa BST.

#### UWAGA

Należy pamiętać, żeby przy dokonywaniu rotacji doczepić do poprzednika węzła  $y$  właściwy następnik (z lub  $t$ ).

Zauważmy, że wykonanie rotacji przywraca danemu poddrzewu jego wysokość przed rozpoczęciem wykonywania operacji *insert*. Dla pozostałych wierzchołków na ścieżce do korzenia atrybut *bf* pozostaje nie zmieniony.

Aby umożliwić przejście ścieżką z powrotem do korzenia, należy podczas wykonywania operacji *search* umieszczać odwiedzane wierzchołki na stosie. Oto algorytm opisujący przechodzenie ścieżką w góre drzewa AVL z odpowiednią aktualizacją atrybutu *bf*.

```

{Stos  $S$  zawiera ścieżkę od korzenia do poprzednika wstawionego
do drzewa wierzchołka  $x$ ; stop jest nazwą procedury, która kończy
operację insert}
jeśli  $S = \emptyset$  to stop;
 $t := x$ ;
{osobno rozpatrujemy poprzednik wstawianego wierzchołka}
 $z := \text{front}(S)$ ;  $\text{pop}(S)$ ;
jeśli  $\text{bf}(z) < 0$  to begin  $\text{bf}(z) := 0$ ; stop end;
jeśli  $t$ -lewy następnik  $z$  to  $\text{bf}(z) := +1$  inaczej  $\text{bf}(z) := -1$ ;
{w poniższej pętli jest powtarzana operacja modyfikacji
atrybutu bf; wierzchołki  $t$ ,  $z$ ,  $y$  tworzą łańcuch przesuwający się
w góre drzewa}
while  $S \neq \emptyset$  do
begin
 $y := \text{front}(S)$ ;  $\text{pop}(S)$ ;
  case  $\text{bf}(y)$  of
    0: jeśli  $z$ -lewy następnik  $y$  to  $\text{bf}(y) := +1$  inaczej  $\text{bf}(y) := -1$ ;
    +1: jeśli  $z$ -prawy następnik  $y$  to begin  $\text{bf}(y) := 0$ ; stop end
      else jeśli  $\text{bf}(z) = +1$ 
        then begin rotacja pojedyncza ( $y$ ,  $z$ ); stop end
        else begin rotacja podwójna ( $y$ ,  $z$ ,  $t$ ); stop end;
```

### 3.3. Drzewa poszukiwań binarnych

```

-1: jeśli  $z$ -lewy następnik  $y$  to begin  $\text{bf}(y) := 0$ ; stop end
  else jeśli  $\text{bf}(z) = -1$ 
    then begin rotacja pojedyncza ( $y$ ,  $z$ ); stop end
    else begin rotacja podwójna ( $y$ ,  $z$ ,  $t$ ); stop end;
```

end;

$t := z$ ;  $z := y$

end;

Pelną implementacją algorytmu *insert*, jak również zaprojektowanie algorytmu *delete*, pozostawiamy Ci jako ćwiczenie (zad. 3.12-3.13). Otrzymujemy takie oto twierdzenie.

### Twierdzenie 3.1.

W wypadku drzew AVL każdą z operacji *search*, *insert* i *delete* można wykonać z pesymistyczną złożonością czasową  $O(\log n)$ . Zaimplementowanie drzewa AVL wymaga  $O(n)$  dodatkowej pamięci na atrybuty *left*, *right* i *bf*, gdzie  $n$  jest maksymalną liczbą elementów w zbiorze  $S$ .

#### 3.3.2.

### Samoorganizujące się drzewa BST

Zamiast wymagać, żeby każda operacja słownika była wykonywana w czasie  $O(\log n)$ , wystarczy zwykle zażądać żeby  $m$  operacji słownika mogło być wykonanych w łącznym czasie  $O(m \log n)$  (a więc z kosztem zamortyzowanym  $O(\log n)$  dla każdej operacji w ciągu). Czas taki zapewnia strukturę danych o nazwie **samoorganizujących się drzew** BST, w której nie trzeba używać dodatkowego atrybutu w rodzaju atrybutu *bf* dla drzew AVL. Podobnie jak dla samoorganizujących się list, wierzchołek zawierający argument operacji jest przesuwany do korzenia drzewa odpowiednio za pomocą wielokrotnego stosowania obu przedstawionych wcześniej rodzajów rotacji.

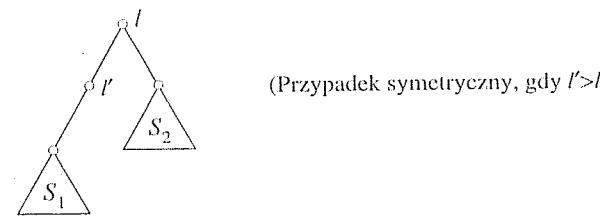
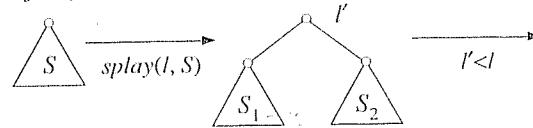
Operacje *search*, *insert* i *delete* zapisuje się za pomocą pomocniczej operacji *splay* o podanej tu specyfikacji:

*splay*( $l$ ,  $S$ ): Drzewo BST  $S$  zostaje przekształcone w drzewo BST  $S'$ , reprezentujące ten sam zbiór elementów co  $S$ . Jeśli  $l$  jest w  $S$ , to korzeń drzewa  $S'$  zawiera  $l$ . Jeśli  $l$  nie ma w  $S$ , to w korzeniu drzewa  $S'$  znajduje się taki element  $l'$ , że między wartościami  $\min(l', l)$  i  $\max(l', l)$  nie ma elementu z  $S$  (jeśli są dwa takie elementy, to jeden z nich jest wybierany dowolnie).

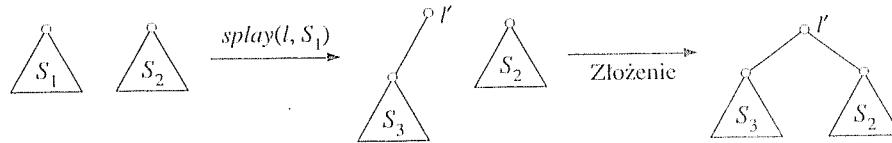
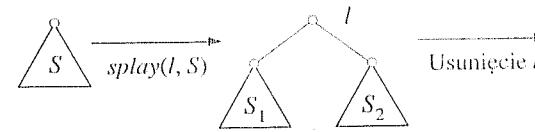
Operacje *search*, *insert* i *delete* można zrealizować, używając operacji *splay* (rys. 3.15).

$search(l, S)$ ::  $splay(l, S)$  + sprawdzenie, czy w korzeniu powstającego drzewa znajduje się  $l$

$insert(l, S)$ ::



$delete(l, S)$ ::



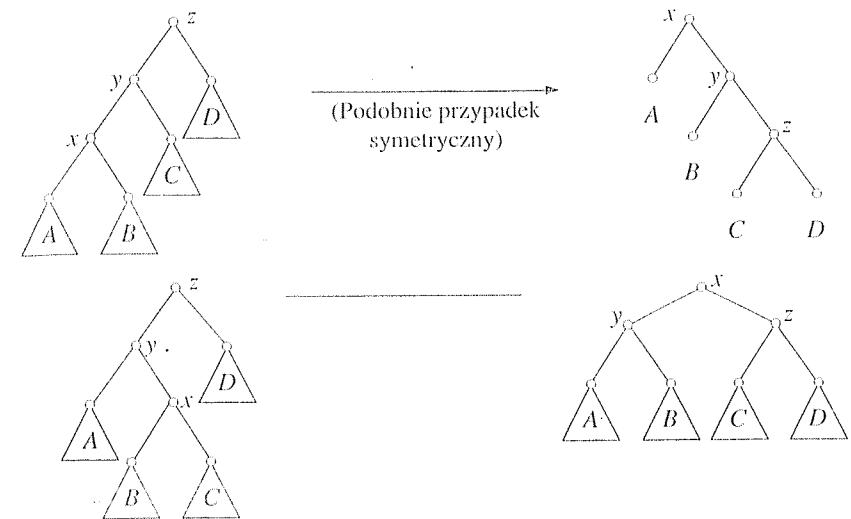
Rys. 3.15. Zapisanie operacji słownika za pomocą operacji *splay*

Operacja  $splay(l, S)$  polega najpierw na realizacji operacji  $search(l, S)$  dla zwykłych drzew BST do wyznaczenia ścieżki od korzenia do wierzchołka  $x$  zawierającego element  $l$ . Następnie przechodzimy z powrotem po ścieżce od  $x$  do korzenia drzewa, wykonując ciąg rotacji pojedynczych, w wyniku których wierzchołek  $x$  zostaje przesunięty do korzenia. Rotacje pojedyncze są wykonywane parami. Wykonanie jednej pary nazywa się **krokiem rozchylającym** (rys. 3.16).

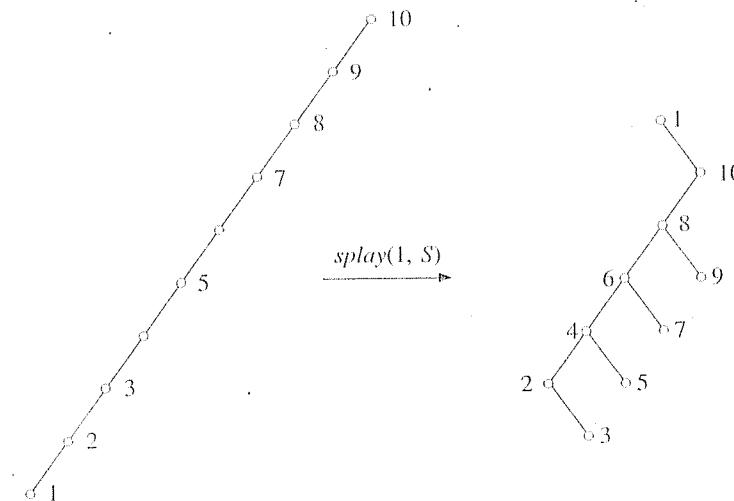
Jeśli odległość  $x$  od korzenia jest nieparzysta, to na końcu jest wykonywana jedna rotacja pojedyncza. Przykład wykonania operacji *splay* przedstawiamy na rysunku 3.17.

Jako **koszt operacji f**, co oznacza się przez  $cost(f)$ , przyjmuje się liczbę kroków rozchylających (licząc również rotacje pojedyncze kończące *splay*). Koszt jednej operacji może być nawet  $O(n)$ , ale rozłożony na cały ciąg operacji jest tylko  $O(\log n)$ . Aby to wykazać, każdej operacji przyporządkowuje się – oprócz jej kosztu – także kredyt na jej wykonanie (rzędu  $\log n$ ). Również drzewom BST przyporządkowuje się kredyt. Pełnią one funkcję banków użyczających operacjom kredytów, gdy ich koszt jest wyższy niż

### 3.3. Drzewa poszukiwań binarnych



Rys. 3.16. Krok rozchylający



Rys. 3.17. Wynik wykonania operacji *splay*

przyznany im kredyt. Gdy koszt wykonania operacji jest mniejszy niż przyznany jej kredyt, nie zużyty kredyt jest odkładany w „banku” drzewa BST.

Kredyt drzewa BST definiujemy w następujący sposób:

$$C(S) = \sum_{x \in S} \lfloor \log w(x) \rfloor$$

gdzie  $w(x)$  jest liczbą potomków  $x$  (wliczając w to  $x$ ). Zauważmy, że kredyt drzewa początkowego  $C(S_0) = 0$  i że dla dowolnego drzewa BST  $S$  mamy  $0 \leq C(S) \leq n \log n$ .

Kredyt operacji  $f$  przekształcającej drzewo BST  $S$  w drzewo BST  $S'$  definiujemy w następujący sposób:

$$c(f) = cost(f) + C(S') - C(S)$$

Można udowodnić taki oto lemat (zad. 3.14).

#### L Lemat 3.4.

Jeśli  $f = splay(l, S)$ , to  $c(f) \leq 3 \log n + 1$ .

Wypływa stąd wniosek:

kredyt operacji *search* jest  $\leq 3 \log n + 1$  (jedna operacja *splay*); kredyt *insert* kredyt operacji *search* jest  $\leq 3 \log n + 1$  (jedna operacja *splay*); kredyt *delete*  $\leq 6 \log n + 2 \leq 4 \log n + 1$  (kredyt *splay* + wzrost kredytu drzewa o  $\log n$ ); kredyt *delete*  $\leq 6 \log n + 2$  (dwie operacje *splay*).

#### T Twierdzenie 3.2.

Koszt wykonania  $m$  operacji *search*, *insert* i *delete* jest  $O(m \log n)$ .

Dowód: Niech  $\sigma = f_1; \dots; f_m$  będzie ciągiem operacji *search*, *insert* i *delete*. Niech  $f_i$  przekształca drzewo BST  $S_{i-1}$  w  $S_i$ . Wówczas

$$\begin{aligned} cost(\sigma) &= \sum_{i=1}^m cost(f_i) = \sum_{i=1}^m c(f_i) + \sum_{i=1}^m (C(S_i) - C(S_{i-1})) = \\ &= \sum_{i=1}^m c(f_i) + C(S_m) + C(S_0) = O(m \log n) \end{aligned}$$

cbdo

## 3.4. Mieszanie

Mieszanie jest zupełnie odmiennym rozwiązyaniem problemu słownika od drzew BST. Wykorzystuje się w nim własności numeryczne przechowywanych elementów (w drzewach BST jedyna informacja o elementach pochodzi z wyników porównań).

Najprostszy schemat przechowywania  $n$  elementów polega na potraktowaniu  $i$ -tego elementu zbioru  $a_i$  jako indeksu i zamarkowaniu należenia  $a_i$  do zbioru  $S$  na pozycji  $A[a_i]$  pewnej tablicy  $A$ . Aby stwierdzić, czy element  $x$  należy do  $S$ , należy sprawdzić pozycję  $A[x]$ . Metoda ta staje się niepraktyczna, gdy uniwersum możliwych elementów jest zbyt duże. Można ją jednak nieco zmodyfikować.

#### 3.4. Mieszanie

Najpierw należy obliczyć wartość pewnej funkcji odwzorowującej uniwersum elementów w zbiór indeksów tablicy. Funkcję tę nazywamy **funkcją mieszającą**. Obliczona wartość daje nam indeks w tablicy, pod którym możemy znaleźć poszukiwany element. Może się przy tym okazać, że na pozycji o danym indeksie znajduje się kilka elementów zbioru  $S$ . Zjawisko to nazywamy **kolizją**. Elementy o tej samej wartości funkcji mieszającej są trzymane na jednej liście reprezentowanej bezpośrednio lub w sposób ukryty w jednej tablicy.

Mieszanie może być uważane za metodę wypośrodkowania wymagań pamięciowych i czasowych:

- jeśli nie ma ograniczeń pamięciowych, to element  $v$  pamiętamy pod adresem  $v$ ; wyszukiwanie wymaga wówczas czasu  $O(1)$ ;
- jeśli nie ma ograniczeń czasowych, to  $a_1, a_2, \dots, a_n$  przechowujemy w postaci liniowej, używając w ten sposób minimum pamięci.

Przy mieszaniu staramy się mieć szybki dostęp do elementów zbioru i używać jak najmniej pamięci.

Mieszanie to klasyczna metoda informatyczna w tym sensie, że została dokładnie zbadana i jest powszechnie stosowana. Jej użyteczność została potwierdzona zarówno przez analizy teoretyczne, jak i testy.

#### 3.4.1. Wybór funkcji mieszającej

Funkcja mieszająca jest funkcją odwzorowującą uniwersum, z którego pochodzą elementy zbioru  $S$ , w zbiór adresów  $[0..m-1]$  dla pewnej liczby naturalnej  $m$ . Idealna funkcja mieszająca powinna być:

- łatwo obliczalna;
- losowa, tzn. każdy indeks  $[0..m-1]$  powinien być jednakowo prawdopodobny jako wartość funkcji.

Jeśli elementy zbioru  $S$  mają złożoną postać, to funkcję mieszającą definiuje się jako złożenie dwóch transformacji:

- transformacji elementu w jedno słowo maszyny;
- transformacji słowa w indeks z przedziału  $[0..m-1]$ .

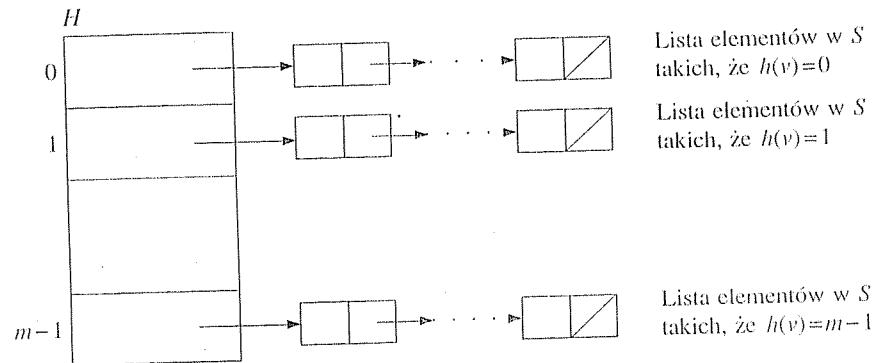
W pierwszym wypadku stosuje się takie operacje na słowach bitowych, jak różnica symetryczna słów. W drugim wypadku stosuje się często jedną z dwóch funkcji:

- $h_1(k) = k \bmod m$  (gdzie  $m$  to zwykłe liczba pierwsza);
- $h_2(k) = M$  najbardziej znaczących bitów liczby  $(ck) \bmod b$ , gdzie  $m = 2^M$ ,  $b = 2^B$ ,  $B = \text{długość słowa maszyny}$ ; postać dziesiętna stałej  $c$  nie powinna kończyć się na ...  $x21$  z cyfrą parzystą  $x$  ([K]).

### 3.4.2. Struktury danych stosowane do rozwiązywania problemu kolizji

#### METODA I

Jest to tzw. **metoda łańcuchowa**. Polega na utrzymywaniu dla każdego indeksu  $i \in [0..m-1]$  listy elementów, które mają tę samą wartość funkcji mieszającej  $h(v) = i$  (rys. 3.18). Użyjemy tablicy  $H[0..m-1]$ , nazywanej **tablicą mieszania**, do zapisu wskaźników do tych list.



Rys. 3.18. Rozwiązywanie problemu kolizji metodą łańcuchową

Operacje słownika polegają na wyznaczeniu listy  $H[h(v)]$ , a następnie na wykonaniu odpowiedniej operacji na tej liście z argumentem  $v$ . W przypadku pesymistycznym złożoność czasowa metody łańcuchowej jest zatem taka sama jak bezpośredniej implementacji listowej:

$$W(n, m) = O(n) \text{ dla operacji } search, insert \text{ i } delete;$$

$$W(n, m) = O(m) \text{ dla operacji } construct.$$

Aby wyznaczyć oczekiwany czasowy złożoność, zauważmy, że wstawianie elementów na listy odbywa się zgodnie ze schematem Bernoulliego. Przy założeniu, że prawdopodobieństwo otrzymania danej wartości funkcji mieszającej jest takie samo dla każdego elementu, prawdopodobieństwo trafienia elementu na daną listę wynosi  $1/m$ . Przy  $n$  losowanych elementach oczekiwana liczba elementów trafiających na jedną listę wynosi  $n/m$ . Mamy więc dla operacji  $search, insert$  i  $delete$

$$A(n, m) = \frac{n}{m} + O(1)$$

Ponieważ metoda łańcuchowa wymaga pamięci na przechowywanie dowiązań, złożoność pamięciowa wynosi

$$S(n, m) = m + n + O(1)$$

#### 3.4. Mieszanie

Mieszanie, a w szczególności metodę łańcuchową, stosuje się zwykle, gdy  $n \leq m$ . Dla metody łańcuchowej oczekiwana złożoność czasowa jest wtedy  $O(1)$ . Wymaga to jednak wcześniejszej znajomości maksymalnej wartości  $n$ . Jeśli maksymalna wartość  $n$  nie jest dana z góry, stosuje się dodatkowo metodę reorganizacji struktury danych, gdy  $n$  staje się większe niż  $m$ . Aby utrzymać rząd wielkości złożoności czasowej, podwaja się wartość  $m$  (tzn.  $m' = 2m$ ). Elementy z tablicy  $H$  mieszają się raz jeszcze w nowo utworzonej, dwa razy dłuższej tablicy  $H'$ .

Koszt jednej reorganizacji jest proporcjonalny do  $m' + n = O(m')$ . Koszt  $l$  reorganizacji, czyli  $m^{(l)} = 2^l m$ , jest proporcjonalny do

$$\sum_{i=1}^l m 2^i = m(2^{l+1} - 2) = O(m^{(l)})$$

Reasumując, jeśli liczba elementów w zbiorze  $S$  jest zawsze nie większa niż rozmiar tablicy bieżącej, to koszt wszystkich reorganizacji jest liniowy względem największego rozmiaru tablicy mieszania.

Zaletą metody łańcuchowej jest jej prostota i dobra oczekiwana złożoność czasowa. Wadą natomiast jest stosunkowo duże dodatkowe obciążenie pamięci. W następnych dwóch metodach rozwiązywania problemu kolizji w ogóle nie używa się dodatkowej pamięci; mają one za to trochę gorszą oczekiwana złożoność czasową. Dla ich poprawności istotny jest fakt, że  $m \geq n$ . Jest w nich przestrzegana ogólna reguła zwana **regułą adresowania otwartego**. Oto ona: elementy są przechowywane w tablicy mieszania  $H[0..m-1]$ ; w razie zaistnienia kolizji należy użyć innego wolnego miejsca w tablicy mieszania  $H[0..m-1]$ .

#### METODA II

Jest to tzw. **metoda adresowania liniowego**. Można ją opisać w następujący sposób: jeśli miejsce  $H[h(v)]$  jest zajęte i  $H[h(v)] \neq v$ , to szukaj miejsca  $v$  (lub miejsca do wstawienia  $v$ ) pod kolejnymi adresami  $h(v) + 1, h(v) + 2, \dots \pmod{m}$

Oto implementacje operacji *construct* i *insert*.

```

procedure hashconstruct;
var i : integer;
begin
  for i := 0 to m-1 do
    H[i] := +∞; {+∞ oznacza wolne miejsce w tablicy}
    n := 0
  end hashconstruct;

function hashinsert(v : integer) : integer;
var i : integer;

```

```

begin {n < m}
  i := h(v);
  while (H[i] ≠ v) and (H[i] ≠ +∞) do
    i := (i + 1) mod m;
    if H[i] ≠ v then
      begin
        H[i] := v;
        n := n + 1
      end;
      hashinsert := i
    end hashinsert;
  end;

```

W trakcie zapelniania się tablicy znajdującej się w niej elementy często grupują się razem, przez co operacje wykonują się znacznie wolniej.

Oto metoda, dzięki której unika się grupowania elementów. Zamiast przyrostu 1 używa się w niej wartości, która zależy losowo od samego elementu  $v$ .

### Metoda III

Jest to tzw. **metoda mieszania podwójnego**. Zamiast przyrostu 1 bierzemy przyrost określony przez drugą funkcję mieszającą  $h'(v)$ . Funkcja ta powinna spełniać następujące warunki:

- $h'(v) > 0$ ;
- $h'(v)$  względnie pierwsza z  $m$  (najlepiej, gdy  $m$  jest liczbą pierwszą);
- $h'$  istotnie różna od  $h$ ; na przykład dla funkcji  $h_1$  z podrozdziału 3.4.1 można wziąć funkcję

$$h'(k) = m - 2 - k \bmod (m - 2)$$

gdzie  $m$  jest liczbą pierwszą.

```

function hashinsert(v : integer) : integer;
var u, i : integer;
begin
  i := h(v); u := h'(v);
  while (H[i] ≠ v) and (H[i] ≠ +∞) do
    i := (i + u) mod m;
    if H[i] ≠ v then
      begin
        H[i] := v;
        n := n + 1
      end;
      hashinsert := i
    end hashinsert;
  end;

```

### 3.4. Mieszanie

Zajmiemy się teraz analizą oczekiwanej złożoności czasowej dla operacji  $insert(v, S)$ , gdy  $v \notin S$ , przy metodzie mieszania podwójnego, przyjmując przybliżające rzeczywisty model probabilistyczny założenie, że adresy  $(h(v) + ih'(v)) \bmod m$  tworzą losową permutację liczb  $0, 1, \dots, m - 1$  (zakładamy dodatkowo, że operacja  $delete$  nie była w ogóle wykonywana).

Niech  $C^-(n, m)$  będzie oczekiwana liczbą porównań wykonywanych w operacji  $insert(v, S)$ , gdy w tablicy  $H[0..m - 1]$  znajduje się już  $n$  elementów. Wówczas

$$C^-(0, m) = 1$$

oraz

$$C^-(n, m) = 1 + \frac{n}{m} C^-(n - 1, m - 1)$$

dla  $n > 0$ , gdyż co najmniej jedna próba jest zawsze wykonywana. Natomiast z prawdopodobieństwem  $n/m$  (oznaczającym prawdopodobieństwo, że pierwsze sprawdzane miejsce jest zajęte) branych jest pod uwagę pozostałych  $m - 1$  pozycji tablicy  $H$  oraz pozostałych  $n - 1$  elementów zbioru. Stosując indukcję matematyczną ze względu na wartość  $m$ , łatwo można pokazać, że rozwiązaniem powyższego równania jest funkcja:

$$C^-(n, m) = \frac{m + 1}{m - n + 1}$$

Mamy zatem

$$C^-(n, m) = \frac{m + 1}{m - n + 1} = \frac{1 + \frac{1}{m}}{1 - \frac{n}{m} + \frac{1}{m}} \approx \frac{1}{1 - \alpha}$$

gdzie  $\alpha = \frac{n}{m}$  oznacza współczynnik zapelnienia tablicy mieszania.

Zajmiemy się teraz analizą oczekiwanej złożoności czasowej operacji  $search(v, S)$ , gdy  $v \in S$ . Skorzystamy z powyższego wzoru dla przypadku, gdy  $v \notin S$ , i ze spostrzeżenia, że liczba prób potrzebnych do wyszukania  $v$  jest równa liczbie prób wykonywanych przy wstawianiu  $v$  do zbioru  $S$ .

Ponieważ element  $v$  mógł być z jednakowym prawdopodobieństwem wstawiany do zbioru  $S$ , gdy zbiór ten miał  $k = 0, 1, \dots, n - 1$  elementów, mamy

$$\begin{aligned}
 C^*(n, m) &= \frac{1}{n} \sum_{k=0}^{n-1} \frac{m+1}{m-k+1} = \frac{m+1}{n} (H_{m+1} - H_{m-n+1}) \equiv \\
 &\equiv \frac{1}{\alpha} (\ln(m+1) - \ln(m-n+1)) = \frac{1}{\alpha} \ln \frac{m+1}{m-n+1} \equiv \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
 \end{aligned}$$

W tabeli 3.3 podajemy zebrane wyniki analizy oczekiwanej złożoności czasowej z użyciem przedstawionych metod rozwiązywania problemu kolizji.

Tabela 3.3. Oczekiwana złożoność czasowa w wypadku różnych metod mieszania

	Liczba prób dla $\text{search}(v, S), v \in S$	Liczba prób dla $\text{search}(v, S), v \notin S$
Metoda łańcuchowa	$1 + \frac{\alpha}{2}$	$1 + \alpha$
Adresowanie liniowe	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$
Mieszanie podwójne	$-\frac{1}{\alpha} \ln(1-\alpha)$	$\frac{1}{1-\alpha}$

Podstawową zaletą metod opartych na regułach adresowania otwartego jest niewielkie obciążenie pamięciowe ( $S(n, m) = m - n$ ). Mają one jednak także poważne wady, których nie ma metoda adresowania łańcuchowego. Otóż zawsze musi zachodzić zależność:  $n \leq m$ , a poza tym gdy  $n$  jest bliskie  $m$ , algorytmy stają się podobne do wyszukiwania liniowego na liście. Operacja *delete* jest bardziej skomplikowana (szczególnie w wypadku mieszania podwójnego), zachodzi bowiem konieczność oznaczania pozycji tablicy mieszania jako „miejscu wolnego, ale kiedyś zajętego”.

## 3.5. Wyszukiwanie pozycyjne

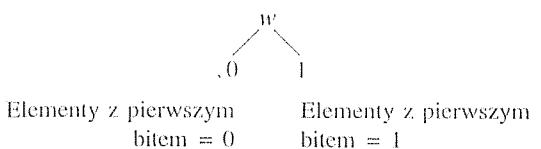
Omówimy teraz metodę wyszukiwania, w której wykorzystuje się postać elementów zbioru  $S$  jako słów binarnych, podobnie jak w metodzie sortowania pozycyjnego. Założymy, że słowa binarne mają długość  $\max b + 1$ ,  $v = (v_{\max b} v_{\max b-1} \dots v_0)_2$ ,  $v_i \in \{0, 1\}$ . Będziemy używać tej samej funkcji wycinania bitów co przy sortowaniu pozycyjnym, tzn.

$$\text{bits}(v, k, j) = (v_{k+j-1} \dots v_{k+1} v_k)_2$$

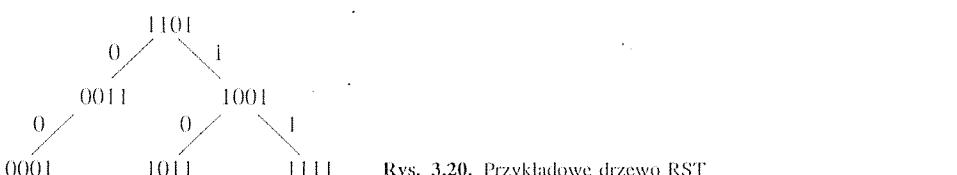
Rozważmy teraz kolejno trzy rodzaje drzew poszukiwań pozycyjnych, realizujących wyszukiwanie pozycyjne.

### 3.5.1. Drzewa RST

Drzewa poszukiwań pozycyjnych, w skrócie drzewa RST (od ang. *Radix Search Trees*), są alternatywną strukturą danych do drzew poszukiwań binarnych (BST). Rozgałęzienia są dokonywane nie przez porównanie wartości elementów, ale na podstawie wartości kolejnego bitu wyszukiwanego słowa  $v$ . Na pierwszym poziomie drzewa decydujące znaczenie ma pierwszy bit  $v_{\max b}$ , tzn. elementy zbioru  $S$  z bitem 0 na pierwszej pozycji trafiają do lewego poddrzewa, a elementy z bitem 1 do prawego (rys. 3.19).



Rys. 3.19. Rozdział elementów w drzewie RST w zależności od pierwszego bitu



Rys. 3.20. Przykładowe drzewo RST

Na drugim poziomie w drzewie decydujące znaczenie ma drugi bit itd.

Jak poprzednio, zakładamy, że każdy węzeł  $x$  drzewa RST ma atrybuty:  $\text{left}(x)$ ,  $\text{key}(x)$ ,  $\text{right}(x)$ .

Poszukiwanie elementu i wstawianie go do drzewa RST odbywa się w podobny sposób jak w wypadku drzewa BST.

```

function digitalsearch(v : integer; x : node) : node;
{ x jest korzeniem drzewa RST}
var b : integer; x : node;
begin
  b := maxb; x := x;
  while (x ≠ nil) and (key(x) ≠ v) do
  begin
    if bits(v, b, 1) = 0 then x := left(x) else x := right(x);
    b := b - 1
  end;
  digitalsearch := x
  { (v ∈ S ∧ key(x) = v) ∨ (v ∉ S ∧ x = nil) }
end digitalsearch;
  
```

```

function digitalinsert(v : integer; var x : node) : node;
{v jest korzeniem drzewa RST; v jest elementem do wstawienia w drzewie
RST}
var x, y : node; b : integer;
begin {pierwsza część algorytmu to wyszukanie miejsca w drzewie do
wstawienia elementu v}
  b := maxb; x := r; y := nil;
  while (x ≠ nil) and (key(x) ≠ v) do
  begin
    y := x;
    if bits(v, b, 1) = 0 then x := left(x) else x := right(x);
    b := b - 1
  end;
  if x = nil then {jeśli v nie ma w drzewie, to zostaje wstawiony do
nowego węzła, który jest doczepiany do y}
  begin
    new(x);
    key(x) := v; left(x) := nil; right(x) := nil;
    if y ≠ nil then
      if bits(v, b+1, 1) = 0 then left(y) := x else right(y) := x
      else r := x
    end;
    digitalinsert := x
  end digitalinsert;
end digitalinsert;

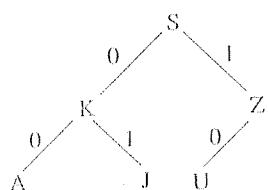
```

#### □ PRZYKŁAD:

Załóżmy, że  $i$ -tą literę alfabetu reprezentujemy za pomocą słowa binarnego odpowiadającego liczbie  $i$  i że drzewo RST tworzymy przez wykonanie wstawień liter w następującej kolejności:

S 10011  
 Z 11010  
 U 10101  
 K 01011  
 A 00001  
 J 01010

Otrzymujemy drzewo RST przedstawione na rysunku 3.21.



Rys. 3.21. Drzewo RST

#### 3.5. Wyszukiwanie pozycyjne

Operacja *delete* jest nieco prostsza niż dla drzew BST. Na miejsce usuwanego elementu można – jeśli nie jest on w liściu – wstawić element z dowolnego liścia w poddrzewie. Zapisanie tej operacji pozostawiamy Ci jako ćwiczenie (zad. 3.18).

Pesymistyczna złożoność czasowa operacji słownika na drzewach RST nie jest najlepsza. Mamy bowiem  $W(n, maxb) = O(\min(n, maxb))$  porównań słów (czyli  $O(maxb \cdot \min(maxb, n))$  porównań bitów).

Załóżmy, że elementy zbioru  $S$  są losowe, tzn. że na każdej pozycji  $b$  z jednakowym prawdopodobieństwem może wystąpić 0 lub 1. Niech  $p_j$  oznacza prawdopodobieństwo, że w  $n-1$  losowaniach bitów  $j-1$  razy padnie 0, a  $n-j$  razy 1. Gdy zostanie wylosowane 0, wstawiamy element do lewego poddrzewa, a gdy 1 – do prawego. Otrzymujemy

$$p_j = \binom{n-1}{j-1} \left(\frac{1}{2}\right)^{j-1} \left(\frac{1}{2}\right)^{n-j} = \binom{n-1}{j-1} \left(\frac{1}{2}\right)^{n-1}$$

Dla drzewa RST o  $n$  wierzchołkach z prawdopodobieństwem  $p_j$  w lewym poddrzewie znajdzie się  $j-1$  wierzchołków, a w prawym  $n-j$ . Niech  $G(n)$  oznacza sumę długości ścieżek od korzenia do wierzchołka wewnętrznego w losowym drzewie RST (o  $n$  wierzchołkach). Mamy wtedy

$$\begin{cases} G(0) = 0, \quad G(1) = 0 \\ G(n) = (n-1) + \sum_{j=1}^n p_j (G(j-1) + G(n-j)) \quad \text{dla } n > 1 \end{cases}$$

Można udowodnić [BKR], że

$$G(n) = n \log n + O(n)$$

Oczekiwana złożoność czasowa operacji *search*, gdy  $v \in S$ , wynosi zatem  $A(n) = \log n + O(1)$  porównań słów.

Tę samą oczekiwana złożoność czasową otrzymujemy się dla pozostałych operacji słownika. Złożoność pamięciowa wynosi oczywiście

$$S(n) = O(n)$$

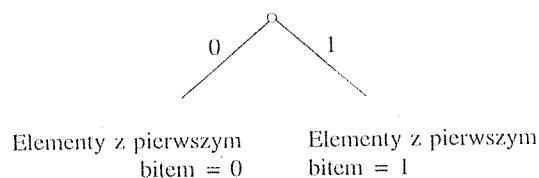
Podstawową wadą drzew poszukiwań pozycyjnych jest konieczność porównywania na każdym poziomie drzewa całych słów binarnych. W wypadku następnych dwóch struktur danych porównywanie słów odbywa się tylko raz, na zakończenie poszukiwania.

### 3.5.2. Drzewa TRIE

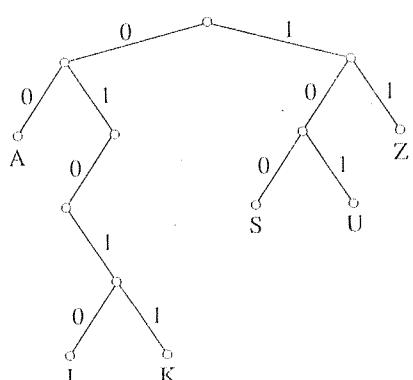
W drzewie TRIE elementy zbioru  $S$  są zapisywane w liściach. Węzły wewnętrzne mają tylko dwa atrybuty: *left* i *right*. Liść ma tylko atrybut *key*. Oprócz tego jest potrzebny atrybut logiczny typu Boolean:

$$\text{leaf}(x) = \text{true} \Leftrightarrow x \text{ jest liściem}$$

Zasada wpisywania elementów do drzewa jest taka sama jak w wypadku drzew RST (rys. 3.22).



Rys. 3.22. Zasada wpisywania elementów do drzewa TRIE ze względu na pierwszy bit



Rys. 3.23. Drzewo TRIE

Wstawiając litery z poprzedniego przykładu do pustego drzewa TRIE, otrzymujemy drzewo przedstawione na rysunku 3.23.

Oto algorytm wyszukiwania elementu w drzewie TRIE.

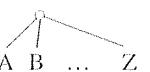
```
function triesearch(v : integer; x : node) : node;
{x jest korzeniem niepustego drzewa TRIE}
var b : integer; x : node;
begin
  b := maxb; x := r;
  while not leaf(x) do
```

### 3.5. Wyszukiwanie pozycyjne

```
begin
  if bits(v, b, 1) = 0 then x := left(x) else x := right(x);
  b := b - 1
end;
trisearch := x
{ (v ∈ S and key(x) = v) or (v ∉ S and key(x) ≠ v) }
end triesearch;
```

Jako ćwiczenie pozostawiamy Ci zapisanie pozostałych algorytmów słownika dla drzew TRIE. Zaletą drzewa TRIE jest to, że poszukiwany w drzewie element  $v$  jest porównywany tylko z jednym elementem w drzewie, mianowicie w odpowiednim liściu. Do wad drzewa TRIE należy pojawianie się gałęzi, na których węzły mają tylko jeden następnik (zwiększa złożoność pamięciową). Wadą jest także niejednorodność formatu węzłów w drzewie (węzły wewnętrzne, liście).

Zamiast rozgałęzień binarnych używa się często rozgałęzień o stopniu wyższym, na przykład względem liter alfabetu (rys. 3.24):



Rys. 3.24. Rozgałęzienia względem liter alfabetu

Rozwiążanie to daje szybkie algorytmy, ale kosztem większego obciążenia pamięci. Możliwe jest też rozwiązanie hybrydowe, które polega na tym, że na górnym poziomach drzewa stosuje się duży stopień rozgałęzienia, a na dolnych niski.

Operacje *search*, *insert* i *delete* mają następującą złożoność (konieczne jest założenie, że  $\maxb + 1 \geq \log n$ ):

$$W(n) = O(\min(\maxb, n) + \maxb) \text{ porównań bitów;}$$

$$A(n) = O(\log n) \text{ porównań bitów, gdy } \log n \leq \maxb;$$

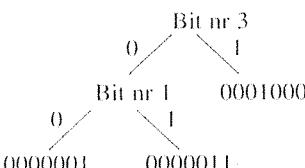
$$S(n) = O(n(\maxb - \log n + 2)) \text{ wierzchołków.}$$

Jako ćwiczenie (zad. 3.20) pozostawiamy Ci skonstruowanie drzewa TRIE o  $n$  liściach, które ma  $S(n) = \Theta(n(\maxb - \log n + 2))$  wierzchołków.

### 3.5.3. Drzewa PATRICIA

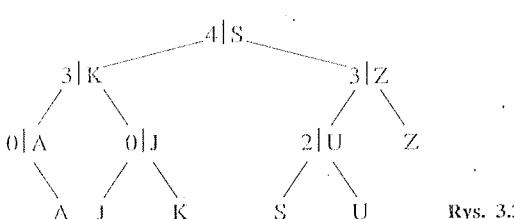
Kolejne drzewo poszukiwań pozycyjnych jest modyfikacją drzewa TRIE bez jego wad, czyli długich gałęzi i niejednorodności węzłów. W drzewie PATRICIA jest tylko  $n$  węzłów do reprezentowania  $n$  elementów. Ma ono zaletę drzewa TRIE, a mianowicie

porównanie kluczy odbywa się tylko raz (na koniec wyszukiwania). Aby skasować długie gałęzie, na ścieżce od korzenia pozostawia się tylko porównania bitów, na których różnią się reprezentowane słowa. Aby odróżnić na przykład trzy słowa binowe (przypomnijmy, że pierwszy bit z lewej strony ma numer  $maxb$ , a porównywanie zaczynamy od bitów o najwyższych numerach): 0001000, 0000011, 0000001, wystarczy najpierw porównać bity nr 3; w wypadku bitu 0 na tej pozycji kolejne bity porównujemy na pozycji nr 1 (rys. 3.25).



Rys. 3.25. Pierwsze drzewo PATRICIA

Aby uniknąć niejednorodności węzłów, utożsamiamy każdy liść z pewnym wierzchołkiem wewnętrzny, wpisując zapisany w liściu element zbioru  $S$  do odpowiadającego mu wierzchołka wewnętrznego. W ten sposób każdy węzeł wewnętrzny jest traktowany dwojako: w fazie wyszukiwania jako węzeł określający rozgałęzienie, a w fazie identyfikacji (dojścia do liścia) – jako liść. Trzeba przy tym dodać jeden węzeł, gdyż jest tylko  $n - 1$  węzłów określających rozgałęzienie. Zbudujmy drzewo PATRICIA dla zbioru  $S$  z poprzedniego przykładu (rys. 3.26).



Rys. 3.26. Drugie drzewo PATRICIA

Każdy węzeł drzewa PATRICIA ma 4 atrybuty: dowiązania drzewowe  $left$  i  $right$ , atrybut  $key$  i atrybut  $b$  (numer bitu, na podstawie którego następuje rozgałęzienie). Założamy, że element zidentyfikowany (przez wyszukiwanie) w danym węźle jest wpisany do jednego z przodków danego węzła. Łatwo jest wtedy odróżnić zwykle dowiązanie drzewowe od dowiązania identyfikującego element: dowiązanie od węzła  $x$  do  $y$  identyfikuje element  $\equiv b(x) \leq b(y)$ .

Algorytm wyszukiwania elementu w drzewie PATRICIA jest prosty.

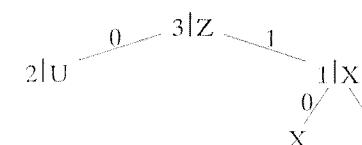
```

function patriciasearch(v: integer; x: node) : node;
{v jest korzeniem drzewa PATRICIA}
var x, y: node;
  
```

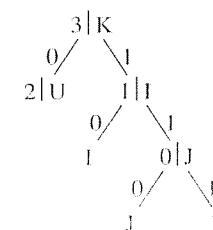
```

begin
  x := x;
  repeat
    y := x;
    if bits(v, b(x), 1) = 0 then x := left(x) else x := right(x)
  until b(y) ≤ b(x);
  patriciasearch := x
  {v jest w S ≡ key(x) = v}
end patriciasearch;
  
```

Przejdziemy teraz do algorytmu wstawiania nowego elementu do drzewa PATRICIA. Założymy, że do drzewa zbudowanego powyżej chcemy wstawić nowy element  $X = 11000$ . Stosujemy algorytm patriciasearch, żeby wyznaczyć słowo binarne, które ma te same wyróżnione bity co  $X$ . Dochodzimy do  $Z = 11010$ .  $X$  i  $Z$  różnią się dopiero na bieżąco numerze 1. Dowiązujemy węzeł zawierający  $X$  do węzła zawierającego  $Z$  (rys. 3.27).



Rys. 3.27. Wstawienie elementu X



Rys. 3.28. Wstawienie elementu I

Załóżmy teraz, że chcemy wstawić element  $I = 01001$ . Operacja patriciainsert prowadzi nas do słowa  $J = 01010$ . Pierwsze miejsce, na którym  $I$  i  $J$  się różnią, to 1. Odróżnienie słów  $I$  i  $J$  musi zatem nastąpić między węzłami zawierającymi  $K$  i  $J$  (rys. 3.28).

Dochodzimy do takiego oto algorytmu.

```

function patriciainsert(v: integer; x: node) : node;
{x jest korzeniem drzewa PATRICIA}
var t, x, y, z: node; i: integer;
begin
  t := patriciasearch(v, x);
  {element v daje te same wyniki porównania co key(t)}
  i := maxb;
  
```

```

while bits(v, i, 1) = bits(key(t), i, 1) do i := i - 1;
{ i jest numerem pierwszej pozycji, na której v i key(t) się różnią}
x := r;
repeat
  z := x;
  if bits(v, b(x), 1) = 0 then x := left(x) else x := right(x)
until (b(z) ≤ b(x)) or (b(x) < i);
{miejsce elementu v jest na dowiązaniu między z a x}
new(y); key(y) := v; b(y) := i;
if bits(v, b(y), 1) = 0 then
begin
  left(y) := y;
  right(y) := x
end else
begin
  right(y) := y;
  left(y) := x
end;
if bits(v, b(z), 1) = 0 then left(z) := y else right(z) := y;
patriciainsert := y
end patriciainsert;

```

Drzewo puste jest najwygodniej reprezentować przez drzewo składające się z jednego wierzchołka ze słowem nie występującym w zbiorze  $S$  (na przykład złożone z samych zer).

W wypadku operacji *search* i *insert* drzewa PATRICIA osiągają najlepsze rzędy wielkości złożoności z obu poprzednich rodzajów drzew:

$W(n) = O(\min(\max b, n) + \max b)$  porównań bitów;

$A(n) = O(\log n)$  porównań bitów, gdy  $\log n \leq \max b$ ;

$S(n) = O(n)$ .

Opracowanie algorytmu dla operacji *delete* pozostawiamy Ci jako ćwiczenie (zad. 3.21).

Dotychczas zakładaliśmy, że rozważane słowa binarne mają tę samą długość. Przedstawione algorytmy można uogólnić do przypadku, w którym słowa mają różną długość, pod warunkiem, że żadne słowo nie jest prefiksem drugiego. Można też zamiast alfabetu dwuliterowego użyć alfabetu o dowolnej liczbie liter.

## 3.6. Wyszukiwanie zewnętrzne

Zajmiemy się teraz problemem wyszukiwania przy założeniu, że elementy zbioru  $S$  znajdują się w pamięci zewnętrznej.

### 3.6. Wyszukiwanie zewnętrzne

Zakładamy, że pamięć zewnętrzna jest podzielona na **bloki**, a każdy blok jest identyfikowany przez swój **adres**. **Format rekordu** to lista atrybutów, a **rekord** to lista wartości, po jednej wartości dla każdego atrybutu. **Plik** jest listą  $n$  rekordów (o tym samym formacie). Stanowi on reprezentację zbioru elementów, a rekordy reprezentację elementów. **Klucz** to podlista formatu rekordu, złożona z takich atrybutów, których wartości w rekordzie identyfikują jednoznacznie rekord w pliku. Zakładamy, że jeden blok mieści  $k$  rekordów ( $k > 0$ ). Jeden plik zajmuje zwykle pewną liczbę bloków. Za operację dominującą przyjmujemy przesłanie jednego bloku między pamięcią wewnętrzną a pamięcią zewnętrzną.

Rozważymy kilka struktur danych rozwiązujących problem wyszukiwania zewnętrznego.

#### 3.6.1. Pliki nieuporządkowane

W pliku nieuporządkowanym rekordy są ulożone w dowolnej kolejności. Wykonanie operacji *search* wymaga  $n/k$  przesłań, a wykonanie operacji *insert* i *delete*  $1 + n/k$  przesłań. Podstawową wadą jest w tym wypadku długi czas działania operacji; podstawową zaletą natomiast nieużywanie dodatkowej pamięci i prostota algorytmów. Struktura danych jest użyteczna, gdy rekordy pliku są zawsze przetwarzane sekwencyjnie, bez wykorzystywania porządku.

#### 3.6.2. Pliki z funkcją mieszaną

Załóżmy, że mamy określona funkcję mieszaną:

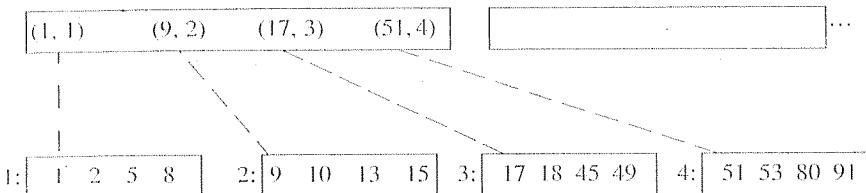
$h$ : zbiór wartości klucza  $\rightarrow \{0, 1, \dots, m-1\}$

Przyjmujemy metodę łańcuchową rozwiązywania kolizji. Tablicę mieszaną, a także poszczególne listy trzymamy w blokach pamięci zewnętrznej. Założymy, że  $m = n/k$ . Wówczas w jednym bloku mieści się średnio jedna lista. W przypadku oczekiwany operacja *search* wymaga przesłania dwóch bloków, a operacje *insert* i *delete* – trzech.

Zaletą tej metody jest jej szybkość działania w przypadku oczekiwany. Do wad natomiast należy długi czas działania w przypadku pesymistycznym oraz fakt, że nie można przetwarzać rekordów pliku w postaci uporządkowanej względem wartości klucza.

#### 3.6.3. Sekwencyjne pliki indeksowane

W tej metodzie rekordy są przechowywane w pliku głównym w postaci uporządkowanej względem wartości klucza. Oprócz pliku głównego z rekordami jest tworzony



Rys. 3.29. Sekwencyjny plik indeksowy

plik pomocniczy nazywany **indeksem rzadkim**. Dla każdego bloku pliku głównego w indeksie rzadkim znajduje się para  $(v, b)$ , gdzie  $b$  jest adresem bloku, a  $v$  najmniejszym kluczem wśród rekordów w tym bloku. Na rysunku 3.29 widać fragment indeksu rzadkiego i pliku głównego dla  $k = 4$  i wartości klucza będących liczbami naturalnymi.

Zauważmy, że plik główny zajmuje  $n/k$  bloków, a indeks rzadki –  $n/k^2$  (przyjmujemy, że każdy blok indeksu rzadkiego mieści  $k$  rekordów). Operacja *search* ma więc średni koszt:

przy wyszukiwaniu sekwencyjnym –  $1 + \frac{1}{2} \left( \frac{n}{k^2} \right)$  przesłań;

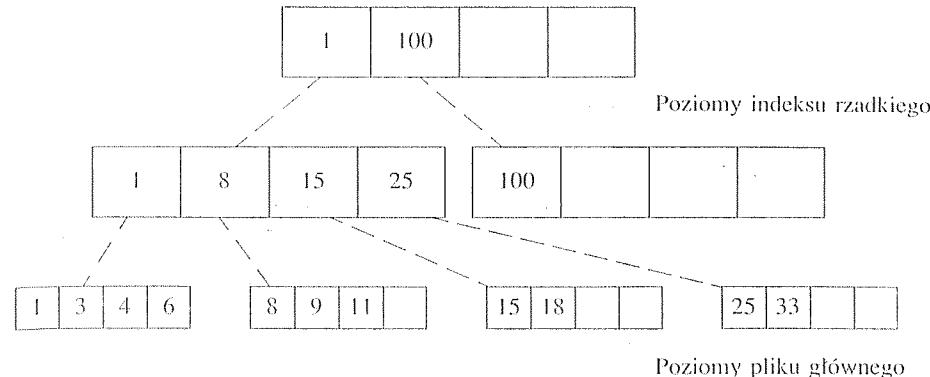
przy wyszukiwaniu binarnym –  $2 + \log \left( \frac{n}{k^2} \right)$ ;

przy wyszukiwaniu interpolacyjnym –  $2 + \log \log \left( \frac{n}{k^2} \right)$ .

Operacje *insert* i *delete* są nieco bardziej skomplikowane. Operację *delete* wykonuje się przez oznaczenie rekordu jako usuniętego. Operację *insert* wykonuje się przez dołaczanie dodatkowych bloków do już istniejących. Powoduje to skomplikowanie struktury opisanej powyżej i pogorszenie złożoności czasowej. Aby uzyskać strukturę dynamiczną, należy rozbudować indeks rzadki do postaci drzewa, przy założeniu, że bloki nie muszą być całkowicie wypełnione rekordami. Podstawową zaletą tej implementacji jest możliwość przetwarzania rekordów w pliku w postaci uporządkowanej względem wartości klucza.

### 3.6.4. B-drzewo jako wielopoziomowy indeks rzadki

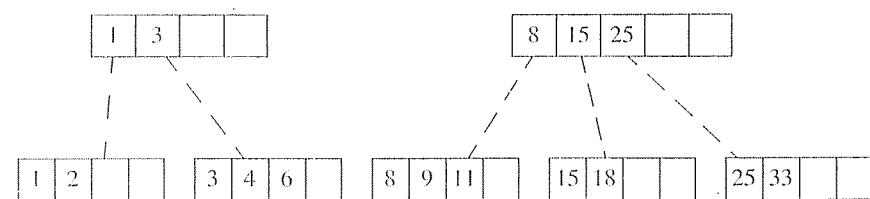
Przyjmujemy, że w jednym bloku znajduje się między  $\lceil k/2 \rceil$  a  $k$  (gdzie  $k \geq 3$ ) rekordów (zakładamy jak poprzednio, że blok pliku głównego i indeksu rzadkiego może pomieścić  $k$  rekordów), z wyjątkiem bloku znajdującego się w korzeniu drzewa, który może zawierać między 2 a  $k$  rekordów. Każdy blok indeksu jest węzłem wewnętrznym drzewa, zawierającym między  $\lceil k/2 \rceil$  a  $k$  dowiązań do węzłów poziomu niższego (z wyjątkiem korzenia). Bloki pliku głównego są liśćmi. Wszystkie liście znajdują się na tej samej głębokości w drzewie. Drzewo zbudowane w opisany właśnie sposób nazywa się **B-drzewem** (rys. 3.30).



Rys. 3.30. B-drzewo jako indeks rzadki

Załóżmy, że chcemy wstawić do drzewa element 10. Posługując się indeksem rzadkim, schodzimy do bloku zawierającego elementy 8, 9, 11. Możemy wstawić do tego bloku 10 po przesunięciu 11 w prawo.

Załóżmy teraz, że chcemy wstawić do drzewa element 2. Blok, do którego pasuje 2, jest już całkowicie zapelniony. Tworzymy nowy blok, który „stawiamy” obok bloku zawierającego 1, 3, 4, 6, i dzielmy elementy 1, 2, 3, 4, 6 między te dwa bloki. W jednym bloku, na przykład zapisujemy 1, 2, a w drugim 3, 4, 6. W ten sposób blok indeksu, zawierający klucze 1, 8, 15, 25, powinien jeszcze obejmować klucz 3. Znowu więc zachodzi konieczność utworzenia nowego bloku, tym razem na poziomie indeksu rzadkiego. W jednym bloku pozostawiamy klucze 1, 3, a do drugiego wpisujemy 8, 15, 25 (rys. 3.31).



Rys. 3.31. Wstawianie elementu do B-drzewa

Powtarzamy te kroki do czasu, aż w kolejnym bloku znajdziemy wolne miejsce albo rozbijemy korzeń na dwa nowe bloki i dołączymy je do nowego korzenia.

Koszt operacji *search*, mierzony liczbą przesłań, szacuje się przez (dla  $k > 3$ )

$$1 + \log_{k/2} \frac{n}{k/2} \leq 2 + \frac{1}{\log k - 1} \log \frac{n}{k}$$

Koszt operacji *insert* i *delete* jest natomiast trzykrotnie większy (dla każdego poziomu jest możliwe przesłanie istniejącego bloku w obie strony i przesłanie do pamięci zewnętrznej nowo utworzonego bloku).

Podstawową zaletą B-drzew jest możliwość stosunkowo łatwego wykonywania operacji *insert* i *delete* z dobrą pesymistyczną złożonością czasową. Rośnie za to obciążenie pamięci, spowodowane dostawianiem nowych poziomów w indeksie rzadkim oraz tym, że bloki mogą być teraz wypełnione tylko w połowie. Ilość dodatkowej pamięci dla pliku głównego wynosi  $nl/k$ , a dla indeksu rzadkiego  $2(n/(k/2)^2) = 8n/k^2$  (dla sekwencyjnego pliku indeksowego tylko  $n/k^2$  bloków).

B-drzewa użyte jako indeks rzadki mają jeszcze jedną wadę, a mianowicie uniemożliwiają używanie wskaźników z zewnątrz do rekordów pliku głównego, ponieważ przy wykonywaniu operacji *insert* i *delete* rekordy te są przesuwane. Kolejna omawiana struktura danych eliminuje tę ostatnią wadę, ale kosztem jeszcze większego obciążenia pamięci.

### 3.6.5. B-drzewo jako wielopoziomowy indeks gęsty

**Indeks gęsty** tworzy się z par  $(v, b)$ , gdzie  $v$  jest wartością klucza rekordu, a  $b$  identyfikatorem (adresem) rekordu, biorąc pod uwagę wszystkie rekordy pliku głównego. Plik główny pozostaje nieuporządkowany i nie wchodzi w skład B-drzewa. B-drzewo jest budowane tylko dla rekordów indeksu gęstego. Obciążenie pamięci rośnie o  $nl/k$  (dodatkowy jeden poziom liści w B-drzewie). Koszt *search* rośnie o jedno dodatkowe przesłanie, a koszt *insert* i *delete* o trzy dodatkowe przesłania (w przypadku pesymistycznym). W tej strukturze danych do rekordów mogą być kierowane dowiązania z zewnątrz (rekordy nie są przesuwane w pliku głównym). Możliwe też staje się zbudowanie kilku indeksów ze względu na różne klucze.

## Zadania

- 3.1. Opracuj bezpośrednią implementację listową słownika z wyszukiwaniem sekwencyjnym elementu na liście. Udowodnij, że dla każdej operacji *search*, *insert* i *delete* pesymistyczna złożoność czasowa i oczekiwana złożoność czasowa są  $O(n)$ .
- 3.2. ([BKR]) Opracuj samoorganizującą się implementację listową słownika, przy której argument operacji *insert* i *search* jest zawsze umieszczany na początku listy.
- 3.3. Udowodnij, że dla wyszukiwania binarnego zachodzą związki:  $W(n) = \log n + O(1)$  i  $A(n) = \log n + O(1)$  (oddziennie dla przypadku, gdy  $v \in S$ , i dla przypadku, gdy  $v \notin S$ ).

## Zadania

- 3.4. Skonstruuj dane dla wyszukiwania interpolacyjnego, wymagające wykonania  $\Omega(n)$  porównań.
- 3.5. Do pustego drzewa BST  $S$  wykonaj ciąg operacji  $insert(4, S); insert(2, S); insert(10, S); insert(6, S); insert(1, S);$
- 3.6. Udowodnij, że lewe i prawe poddrzewo drzewa BST, otrzymanego przez wykonanie  $insert(a_1, S); insert(a_2, S); \dots; insert(a_n, S)$ ; gdzie  $a_1, a_2, \dots, a_n$  jest losową permutacją liczb  $1, 2, \dots, n$ , są losowymi drzewami BST (tzn. zostały utworzone przez losowe ciągi operacji *insert*).
- 3.7. Założymy, że dla ustalonego drzewa  $T$  poszukiwań binarnych o liczbie wierzchołków wewnętrznych  $n$  zmienna  $Z_n$  oznacza sumę głębokości wierzchołków wewnętrznych w  $T$ , a zmienna  $W_n$  – sumę głębokości wierzchołków wewnętrznych w  $T$ . Udowodnij, że  $Z_n = W_n + 2n$
- 3.8. Udowodnij, że dla operacji *search*( $v, S$ ) w drzewach BST oczekiwana wrażliwość na dane wejściowe jest  $O(\sqrt{\log n})$ . Wskazówka: Rozważ osobno przypadki  $v \in S$  i  $v \notin S$ .
- 3.9. Wykonaj ciąg instrukcji  $delete(4, S); insert(3, S); delete(2, S); insert(7, S); delete(10, S);$  względem drzewa BST otrzymanego w zadaniu 3.5. (Jeśli zajdzie potrzeba, użyj metody rzutu monetą do wykonania funkcji *random(2)*).
- 3.10. Ułóż algorytm wypisujący elementy zapisane w drzewie BST w kolejności uporządkowanej. Jaka jest złożoność tego algorytmu? Jaka jest złożoność algorytmu sortowania polegającego na zapisaniu najpierw elementów listy w drzewie BST, a następnie na ich wypisaniu w kolejności uporządkowanej?
- 3.11. Ułóż algorytm rekonstruujący dane drzewo BST do postaci zupełnego drzewa BST (liście znajdują się na dwóch sąsiednich poziomach w drzewie).
- 3.12. Zaprogramuj algorytm *insert* dla drzew AVL.
- 3.13. Zaprogramuj algorytm *delete* dla drzew AVL.
- 3.14. ([BKR, lemat 2.1]) Udowodnij lemat 3.4 o kredycie operacji *splay*.

3.15. 2-3 drzewem nazywamy drzewo, w którym każdy wierzchołek wewnętrzny ma 2 lub 3 następców, a wszystkie liście leżą na tym samym poziomie. Używając 2-3 drzew, opracuj implementację słownika, w której każda z operacji: *insert*, *delete*, *search* oraz:

- (a) *concatenate* ( $S_1, S_2, S$ ): przy założeniu, że wszystkie elementy w słowniku  $S_1$  poprzedzają elementy w słowniku  $S_2$  połączenie obu słowników w jeden słownik  $S$ ;
- (b) *split* ( $S, x, S_1, S_2$ ): rozdzielenie słownika  $S$  na dwa słowniki, przy czym w słowniku  $S_1$  mają być zapisane wszystkie elementy  $\leq x$ , a w słowniku  $S_2$  wszystkie elementy  $> x$

daje się wykonać w czasie  $O(\log n)$ .

3.16. Zaprogramuj algorytm *delete* w wypadku różnych metod rozwiązywania problemu kolizji.

3.17. Istnieje metoda rozwiązywania problemu kolizji (nazywana algorytmem Brenta), która ma na celu przyspieszenie wyszukiwania elementu kosztem dodatkowej pracy przy wstawianiu. Gdy miejsce, gdzie chcemy wstawić dany element, jest już zajęte, musimy zdecydować, czy bardziej opłaca nam się kontynuować obliczanie adresów dla wstawianego elementu, czy może lepiej jest wstawić go w dane miejsce, a przesunąć element, który był tam poprzednio (minimalizując sumę długości wszystkich list kolizji). Okazuje się, że przy takim podejściu  $C^+(n, m) \leq 2,49$  ([K]). Opracuj ten algorytm dla metody mieszania podwójnego.

3.18. Zapisz algorytm *delete* dla drzew RST.

3.19. Zapisz algorytmy słownika dla drzew TRIE.

3.20. Skonstruuj drzewo TRIE o  $n$  liściach, zawierające  $O(n(\max b - \log n))$  wierzchołków.

3.21. Opracuj algorytm *delete* dla drzew PATRICIA.

3.22. Zaprogramuj operacje słownika, używając struktury danych B-drzew.

3.23. Wykonywany jest ciąg operacji słownika:

*construct*( $S$ ); *insert*(0011,  $S$ ); *insert*(1011,  $S$ ); *insert*(0010,  $S$ ); *insert*(0100,  $S$ ); *insert*(0111,  $S$ );

z użyciem następujących struktur danych:

- (a) drzewa BST (porządek alfabetyczny słów),
- (b) drzewa AVL (porządek alfabetyczny słów),

- (c) drzewa RST,
- (d) drzewa TRIE,
- (e) drzewa PATRICIA.

Narysuj odpowiednie drzewa.

3.24. Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie  $O(\log n)$  następujących operacji na zbiorze  $S$ :

- (a) *makeset*( $S$ ):  $S := \emptyset$ ;
- (b) *insert*(( $x, y$ ),  $S$ ):  $S := S \cup \{(x, y)\}$ ;
- (c) *minx*( $S$ ): usunięcie z  $S$  pary  $(x, y)$  o najmniejszej pierwszej składowej;
- (d) *miny*( $S$ ): usunięcie z  $S$  pary  $(x, y)$  o najmniejszej drugiej składowej;
- (e) *searchx*( $x, S$ ): wyznaczenie takiej pary  $(a, b)$ , że  $x = a$ ;
- (f) *searchy*( $y, S$ ): wyznaczenie takiej pary  $(a, b)$ , że  $y = b$ .

Elementy składowe par z  $S$  pochodzą ze zbiorów liniowo uporządkowanych. Wyznacz złożoność pamięciową implementacji.

3.25. Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie  $O(\log n)$  następujących operacji na zbiorze  $S$ :

- (a) *construct*( $S$ ): utworzenie ciągu pustego  $S$ ;
- (b) *insert*( $S, x$ ):  $S := S \cup \{x\}$ ;
- (c) *delete*( $S, x$ ):  $S := S - \{x\}$ ;
- (d) *search*( $S, x$ ): sprawdzenie, czy  $x$  znajduje się w zbiorze  $S$ ;
- (e) *elem*( $S, i$ ): wyznaczenie  $i$ -tego co do wielkości elementu zbioru  $S$ ;
- (f) *numb*( $S, x$ ): wyznaczenie numeru elementu  $x$  w zbiorze  $S$  (względem wielkości).

Zakładamy, że elementy zbioru  $S$  pochodzą z dowolnego liniowo uporządkowanego uniwersum  $U$ .

3.26. Zaproponuj efektywną strukturę danych do wykonywania ciągów następujących operacji (dla elementów  $x$  pochodzących z dowolnego zbioru liniowo uporządkowanego):

- (a) *initialization*:  $S_i := \emptyset$  dla  $i = 1, 2, \dots, n$ ;
- (b) *insert*( $x, S_j$ ):  $S_i := S_i \cup \{x\}$  pod warunkiem, że  $x$  nie występuje w żadnym zbiorze  $S_j$ ,  $1 \leq j \leq n$ ;
- (c) *deletemin*( $S_i$ ): usunięcie ze zbioru  $S_i$  najmniejszego elementu;
- (d) *find*( $x$ ): wyznaczenie numeru zbioru, do którego należy element  $x$ .

Jaka jest złożoność operacji?

3.27.! Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie  $O(\log n)$  następujących operacji na ciągu  $S$ :

- (a)  $construct(S)::$  utworzenie ciągu pustego  $S$ ;
- (b)  $insert(S, i, x)::$  wstawienie  $x$  na  $i$ -te miejsce w ciągu  $S$ , tzn.  $S_i := x$ , pod warunkiem, że  $i \leq |S| + 1$ ;
- (c)  $sum(S, i, j)::$  obliczenie sumy  $\sum_{k=i}^j S_k$ .

Zakładamy, że elementy ciągu są liczbami całkowitymi.

3.28. Do operacji słownika *search*, *insert* i *delete* dodajemy operację:

$$between(x, y) = |\{a \in S: x \leq a \leq y\}|$$

Podaj implementację słownika, przy której każda operacja ma pesymistyczną złożoność czasową  $O(\log n)$ .

3.29. Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie  $O(\log n)$  następujących operacji na zbiorze  $S$  zawierającym przedziały liczb rzeczywistych  $[l, r]$ :

- (a)  $empty(S)::$   $S := \emptyset$ ;
- (b)  $add(S, I)::$   $S := S \cup \{I\}$ ;
- (c)  $delete(S, I)::$   $S := S - \{I\}$ ;
- (d)  $is(S, x)::$  sprawdzenie, czy element  $x$  należy do jakiegoś przedziału zbioru  $S$ ;
- (e)  $intersect(S, I)::$  sprawdzenie, czy przedział  $I$  ma niepuste przecięcie z jakimś przedziałem należącym do  $S$ .

## Złożone struktury danych dla zbiorów elementów

### 4

W rozdziale tym przedstawimy i zbadamy dwie struktury danych umożliwiające wykonywanie różnych operacji na zbiorach rozłącznych.

#### 4.1.

##### Problem sumowania zbiorów rozłącznych

Problemem ściśle związanym z przechowywaniem i wyszukiwaniem informacji jest problem sumowania zbiorów rozłącznych. Przedstawimy go w najprostszej postaci, kiedy przechowywanymi elementami są liczby naturalne. Jeśli chodzi o inne elementy, to byłoby potrzebne wzajemnie jednoznaczne odwzorowanie tych elementów w zbiór  $\{1, 2, \dots, n\}$  dla pewnego naturalnego  $n$ .

Niech  $U = \{S_1, S_2, \dots, S_k\}$  będzie podziałem zbioru  $E = \{1, 2, \dots, n\}$ <sup>10</sup>. Problem sumowania zbiorów rozłącznych polega na podaniu struktury danych reprezentującej podziały zbioru  $E$ , umożliwiającej wykonywanie następujących operacji:

- (a)  $initialization::$   $U := \{\{1\}, \{2\}, \dots, \{n\}\}$  (utworzenie  $n$  jednolitych zbiorów);
- (b)  $union(A, B)::$   $U := (U - \{A, B\}) \cup \{A \cup B\}$ , pod warunkiem, że  $A, B \in U$  i  $A \neq B$ ;
- (c)  $find(x)::$  wyznaczenie w rodzinie zbiorów  $U$  zbioru  $A$ , do którego należy element  $x \in E$ .

<sup>10</sup> Podziałem zbioru  $E = \{1, 2, \dots, n\}$  nazywamy dowolną rodzinę  $U = \{S_1, S_2, \dots, S_k\}$  podzbiorów zbioru  $E$ , spełniającą następujące warunki:

- (1)  $S_i \neq \emptyset$  dla  $1 \leq i \leq k$ ;
- (2)  $S_i \cap S_j = \emptyset$  dla  $1 \leq i \leq j \leq k$ ;
- (3)  $\bigcup_{i=1}^k S_i = E$ .

Na poziomie implementacji będziemy zakładać, że  $x, A, B$  są wskaźnikami do węzłów w strukturze danych. Operacja *initialization* będzie wykonywana raz, na początku. Będziemy też zakładać, że jest wykonywanych  $n - 1$  operacji *union* (tworzących zbiór  $E$ ) oraz pewna liczba  $m$  operacji *find* przemieszanych z operacjami *union*.

#### □ PRZYKŁAD:

Rozważmy problem wyznaczania klas abstrakcji najmniejszej relacji równoważności, zawierającej zadany zbiór par

$$C = \{(a_i, b_i) : 1 \leq i \leq m; a_i, b_i \in E\}$$

Oto algorytm zapisany przy użyciu operacji *initialization*, *find* i *union*, dzięki któremu można ten problem rozwiązać.

```
initialization;
for i := 1 to m do
  if find(ai) ≠ find(bi) then
    union(find(ai), find(bi))
```

Aby sprawdzić, czy liczby  $x$  i  $y$  są w tej samej klasie abstrakcji, sprawdzamy predykat:

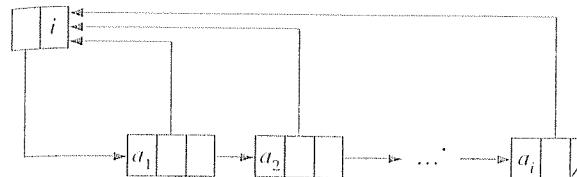
$$\text{find}(x) = \text{find}(y)$$

W szczególności możemy powyższy algorytm zastosować do wyznaczenia wszystkich spójnych składowych grafu niezorientowanego  $G = (V, C)$ . Zbiór krawędzi  $C$  traktujemy jako ciąg par równoważnych wierzchołków, tzn. mających się znaleźć w tej samej klasie abstrakcji – spójnej składowej grafu  $G$ .

Warto porównać ten algorytm z algorytmem wyznaczania spójnych składowych (klas równoważności), opartym na przejściu grafu metodą DFS, czyli w głąb (algorytm ten znajduje się w rozdziale poświęconym algorytmom grafowym). Rozwiązywanie z użyciem tej metody ma charakter statyczny off-line – wszystkie krawędzie grafu muszą być dane z góry, a spójne składowe są produkowane na koniec działania algorytmu. Rozwiązywanie za pomocą zbiorów rozłącznych ma charakter on-line – krawędzie grafu mogą być dodawane na bieżąco i w każdej chwili możemy uzyskać informację, czy dane dwa wierzchołki należą do tej samej spójnej składowej danego grafu.

#### 4.1.1. Implementacja listowa

Zbiór  $A = \{a_1, \dots, a_i\}$  reprezentujemy jako strukturę dowiązań tworzących listę jednokierunkową, z dodanym dodatkowym dowiązaniem prowadzącym od każdego elemen-



Rys. 4.1. Reprezentacja listowa zbiorn  $A = \{a_1, a_2, \dots, a_i\}$

tu listy do głowy listy. Głowa listy jest osobnym węzłem, w którym dodatkowo zapisujemy jeszcze liczbę elementów na liście. Na rysunku 4.1 jest przedstawiona taka lista.

Jako format elementu listy (element) przyjmujemy

$$x: \text{key}(x), \text{head}(x), \text{next}(x)$$

Atrybuty elementu listy oznaczają odpowiednio klucz elementu, dowiązanie do głowy listy oraz dowiązanie do następnego elementu listy.

Jako format głowy listy (*head\_f*) przyjmujemy

$$y: \text{first}(y), \text{count}(y)$$

przy czym atrybuty oznaczają odpowiednio dowiązanie do pierwszego elementu na liście oraz liczbę elementów na liście.

W wyniku operacji *initialization* powstaje  $n$  jednoelementowych list, a jej koszt jest  $O(n)$ . W algorytmie korzystającym ze zbiorów rozłącznych muszą być gdzieś zapisywane głowy bieżącego zbioru list (na przykład w tablicy `var R: array[1..n] of head_f;`). W przytoczonych tu algorytmach nie bierzemy pod uwagę tej zewnętrznej struktury danych.

```
procedure initialization;
var x : element; y : head_f; i : integer;
begin
  for i := 1 to n do
    begin
      new(x); {element}
      key(x) := i; next(x) := nil;
      new(y); {głowa listy}
      first(y) := x; count(y) := 1;
      head(x) := y;
    end
  end;
```

Operacja *find* jest bardzo prosta.

```
function find(x) : head_f;
begin
  find := head(x)
end;
```

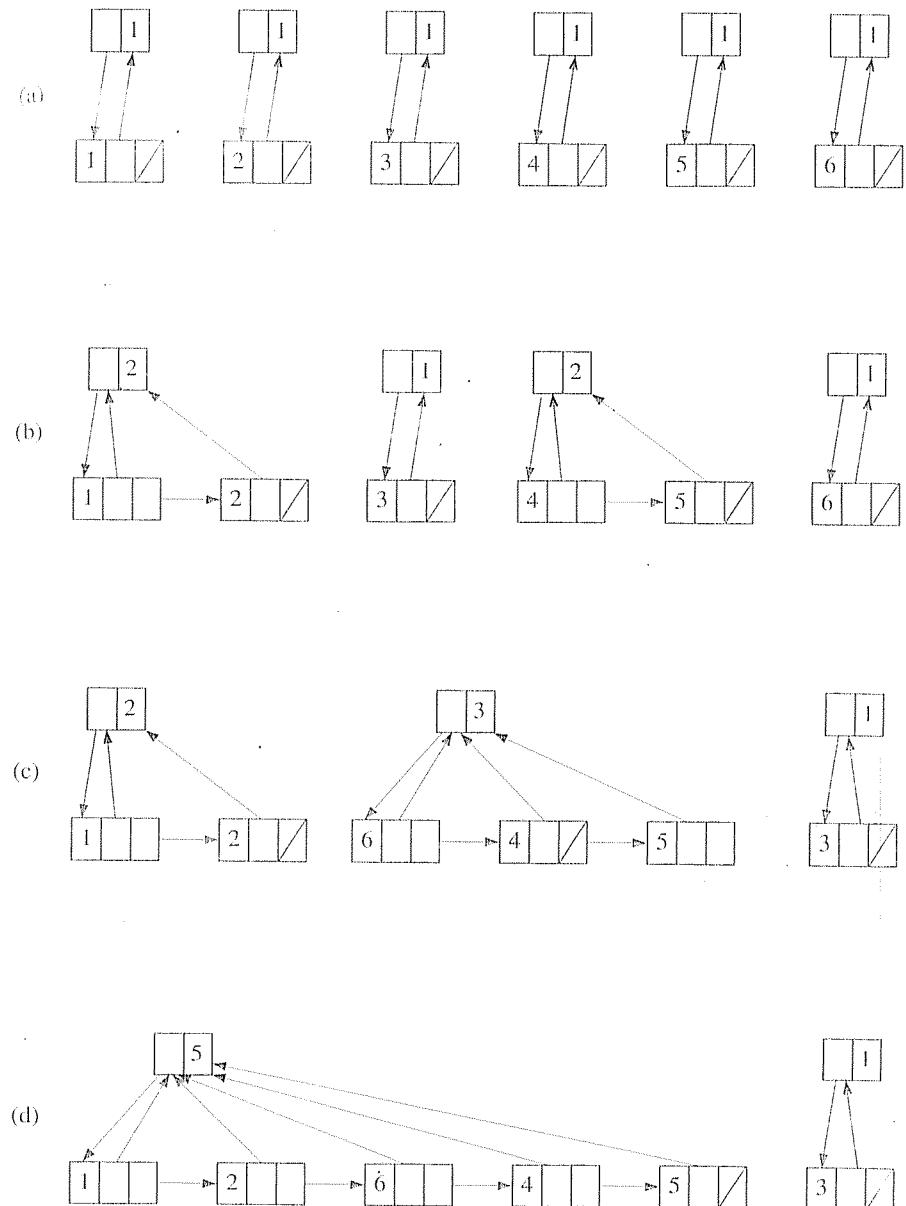
W algorytmie *union* zawsze przyłączamy listę krótszą do dłuższej. Chodzi bowiem o zmniejszenie kosztu łączenia.

```
procedure union(y, z : head_f);
var x : element;
begin
  if count(y) > count(z) then y <-> z; {zamiana y i z}
  x := first(y);
  while next(x) ≠ nil do
    begin head(x) := z; x := next(x) end;
    head(x) := z;
  {połącz dowiązaniem koniec listy y z początkiem listy z}
  next(x) := first(z);
  first(z) := first(y);
  count(z) := count(y) + count(z)
end;
```

#### □ PRZYKŁAD:

Rozwiążmy problem wyznaczania klas równoważności dla następujących danych:  $n = 6$ ,  $E = \{1, 2, 3, 4, 5, 6\}$ ,  $k = 4$ ,  $C = \{(1, 2), (4, 5), (5, 6), (4, 1)\}$ . Kolejne listy w strukturze danych są przedstawione na rysunku 4.2.

Koszt operacji *initialization* jest  $O(n)$ , a koszt *find* –  $O(1)$ . Operacją dominującą w *union* jest operacja  $head(x) := z$ , wykonywana dla każdego elementu listy krótszej, gdy jest ona przyłączana do listy dłuższej. W przypadku pesymistycznym koszt pojedynczej operacji *union* jest  $O(n)$ . Gdy jednak jest wykonywany ciąg  $n - 1$  operacji *union*, tworzących cały zbiór, koszt ten faktycznie jest  $O(n \log n)$ , a nie  $O(n^2)$ . Liczymy bowiem, ile razy dany element może mieć zmieniane dowiązanie do głowy listy. Za każdym razem element przechodzi do listy o rozmiarze co najmniej dwukrotnie większym. Zmian takich może być co najwyżej  $\log n$ . Uwzględniając wszystkie elementy, otrzymujemy koszt wykonania  $n - 1$  operacji *union*  $O(n \log n)$ . Inaczej mówiąc, koszt zamortyzowany pojedynczej operacji *union* jest  $O(\log n)$ .



Rys. 4.2. Implementacja listowa algorytmu równoważności:

- (a) zbiór list po wykonaniu operacji *initialization*;
- (b) zbiór list po wykonaniu *union*(*find*(1), *find*(2)); *union*(*find*(4), *find*(5));
- (c) zbiór list po wykonaniu *union*(*find*(5), *find*(6));
- (d) zbiór list po wykonaniu *union*(*find*(1), *find*(4))

### Twierdzenie 4.1.

Koszt ciągu operacji *initialization*,  $n - 1$  operacji *union* i  $m$  operacji *find* (zakładając, że operacja *initialization* jest wykonywana na samym początku) jest  $O(m + n \log n)$ .

4129

## Implementacja drzewowa

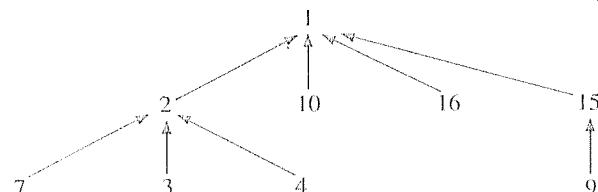
Zbiór  $A = \{a_1, \dots, a_l\}$  reprezentujemy jako strukturę dowiązań tworzących drzewo – z dowiązaniami w góre od następców do poprzednika danego wezła.

### □ PRZYKŁAD:

Zbiór

$$S = \{1, 2, 3, 4, 7, 9, 10, 15, 16\}$$

może być reprezentowany jako drzewo przedstawione na rysunku 4.3.



Rys. 4.3. Drzewo reprezentujące zbiór  $S$

Jako format elementu drzewa (*node*) przyjmujemy

$x$ :  $\text{key}(x)$ ,  $p(x)$ ,  $\text{size}(x)$

Atrybuty elementu drzewa oznaczają odpowiednio klucz wierzchołka, dowiezanie do poprzednika wierzchołka w drzewie oraz rozmiar wierzchołka – określony dla korzeni drzew. Są dwie możliwości zdefiniowania rozmiaru wierzchołka: albo utrzymywać w polu `size(x)` liczbę wierzchołków w drzewie, albo utrzymywać w tym polu wielkość mającą związek z wysokością tworzonego drzewa. Druga możliwość daje rozwiązania bardziej oszczędne pamięciowo. Do przedstawiania algorytmów wygodniej nam jednak będzie zastosować pierwszą.

W wyniku operacji *initialization* powstaje  $n$  jednoelementowych drzew, a jej koszt jest  $O(n)$ . W algorytmie korzystającym ze zbiorów rozłącznych muszą być gdzieś zapisywane korzenie bieżącego zbioru drzew (lasu), na przykład w tablicy **var R: array[1..n] of node**. W przytoczonych tu algorytmach pomijamy ten aspekt.

```

procedure initialization;
var x : node;
    i : integer;
begin
    for i := 1 to n do
        begin
            new(x);
            size(x) := 1;
            p(x) := nil
        end
    end;

```

W algorytmie union przylaczamy mniejsze drzewo do wiekszego

```

procedure union(x, y : node);
begin
    if size(x) > size(y) then x <-> y
    p(x) := y;
    size(y) := size(x) + size(y)
end;

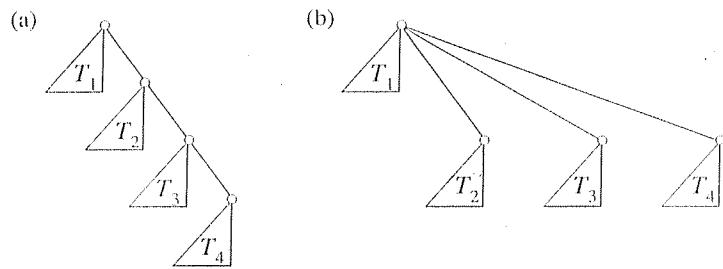
```

Koszt operacji *union* jest  $O(1)$

Lemat 4.1

- (1) Każde drzewo o wysokości  $h$  ma co najmniej  $2^h$  węzłów.
  - (2) Każde drzewo o  $n$  węzłach ma wysokość co najwyżej  $\log n$ .

**Dowód:** Stwierdzenie (2) wynika bezpośrednio z (1). Dowód (1) jest indukcyjny względem liczby wykonanych operacji *union*. Po zainicjowaniu wszystkie drzewa są jednoelementowe i mają wysokość 0. Załóżmy, że drzewo o wysokości  $h_1$  i liczbie wierzchołków  $n_1$  przyłączamy do drzewa o wysokości  $h_2$  i liczbie wierzchołków  $n_2$  ( $n_1 \leq n_2$ ). Gdy  $h_2 > h_1$ , wysokość się nie zwiększa. Gdy  $h_2 \leq h_1$  wysokość nowego drzewa zwiększa się do  $h_1 + 1$ . Korzystamy z założenia indukcyjnego  $n_1 \geq 2^{h_1}$  i  $n_2 \geq 2^{h_2}$ , otrzymując  $n_1 + n_2 \geq 2n_1 \geq 2 \cdot 2^{h_1} \geq 2^{h_1 + 1}$



Rys. 4.4. Wynik kompresji ścieżki

Operacja *find*( $x$ ) polega na przejściu od danego węzła  $x$  do korzenia drzewa, z przekazaniem korzenia jako wyniku operacji. Z lematu wynika, że drzewa mają wysokość logarytmiczną, a więc także koszt operacji *find* jest logarytmiczny. Przy okazji przechodzenia ścieżką do korzenia można spłaszczać bieżące drzewo, przyłączając wszystkie wierzchołki ścieżki bezpośrednio do korzenia drzewa (rys. 4.4). Ta dodatkowo wykonywana operacja nazywa się **kompresją ścieżki**.

```
function find(x: node) : node;
begin
  if p(x) = nil then find := x
  else
    begin
      p(x) := find(p(x));
      find := p(x)
    end
  end;
end;
```

Jako ćwiczenie pozostawiamy Ci wyeliminowanie rekursji z powyższego algorytmu.

Ponieważ wysokość tworzonych w wyniku operacji *union* drzew jest rzędu logarytmicznego, pesymistyczny koszt operacji *find* jest  $O(\log n)$ . Okazuje się jednak, że koszt zamortyzowany pojedynczej operacji *find* jest prawie stały, a mianowicie jest rzędu wielkości funkcji

$$\log^* n = \min\{i : \log^{(i)} n \leq 1\}$$

gdzie  $\log^{(i)} n$  oznacza  $i$ -krotne złożenie funkcji  $\log$ , na przykład  $\log^* 2^{65536} = 5$ ,  $\log^* 2^{2^{65536}} = 6$ . Dla rozmiarów danych, dla których stosuje się strukturę zbiorów rozłącznych,  $\log^*$  jest funkcją „praktycznie” stałą (o wartości mniejszej lub równej 5). Udowodnimy teraz takie oto twierdzenie.

### Twierdzenie 4.2. (Twierdzenie Hopcrofta-Ullmana)

Koszt ciągu operacji *initialization*, a następnie – przemieszanych –  $n - 1$  operacji *union* i  $m$  operacji *find* jest  $O((m + n) \log^* n)$ .

**Dowód:** Dowód wymaga wprowadzenia kilku pomocniczych pojęć. Przez  $i$ -tą warstwę funkcji  $\log^*$  będziemy rozumieć

$$N_i = \{n : \log^* n = i\}$$

Zauważmy, że liczba  $n$  należy do warstwy  $N_{\log^* n}$  oraz że

$$\log^* \log n \leq \log^* n - 1$$

czyli że liczba  $\log n$  należy do warstwy o numerze nie większym niż  $\log^* n - 1$ .

Potrzebna nam będzie jeszcze własność charakteryzująca warstwy funkcji  $\log^*$ . Rozważymy mianowicie funkcję  $F$ , która w przeciwnieństwie do funkcji  $\log^* n$  rośnie bardzo szybko.

$$\begin{cases} F(0) = 0 \\ F(n) = 2^{F(n-1)} \quad \text{dla } n > 0 \end{cases}$$

Z przedstawionych definicji wynika, że

$$\log^* n = i \equiv F(i) < n \leq F(i+1) \text{ dla } i > 0 \text{ oraz } i = 0, \text{ gdy } n = 0, 1$$

Stąd otrzymujemy następującą własność warstw  $N_i$ :

$$\begin{cases} N_0 = \{0, 1\} \\ N_i = \{n : F(i) < n \leq F(i+1)\} \quad \text{dla } i > 0 \end{cases}$$

Innym ważnym pojęciem potrzebnym w dowodzie twierdzenia Hopcrofta-Ullmana jest pojęcie: **rzęd wierzchołka**.

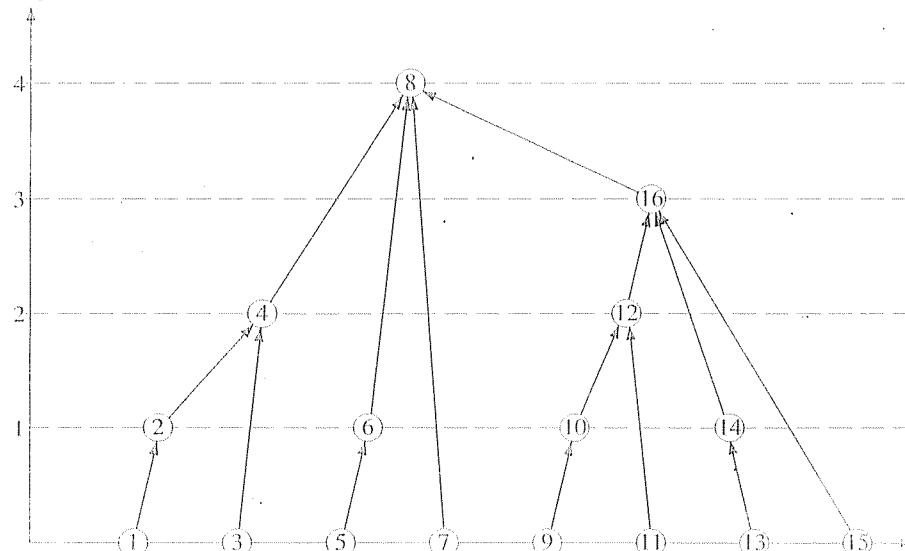
Rzędem wierzchołka  $x$ , co oznaczamy jako  $r(x)$ , nazywamy wysokość wierzchołka  $x$  w drzewie uzyskanym w wyniku wykonania danego ciągu operacji *initialization*, *union* – bez stosowania kompresji dla operacji *find*.

### PRZYKŁAD:

Prześledźmy działanie algorytmu prowadzącego do wyznaczenia klas abstrakcji przy użyciu drzewowej struktury danych. Niech  $n = 16$ ,  $C = \{(1, 2), (3, 4),$

$(2, 4), (5, 6), (7, 8), (6, 8), (4, 8), (9, 10), (11, 12), (10, 12), (13, 14), (15, 16), (14, 16), (12, 16), (1, 5), (10, 1), (9, 3)\}$ . Na rysunku 4.5 jest przedstawione końcowe drzewo, otrzymane bez stosowania kompresji ścieżek w operacjach *find*. Wysokości wierzchołków w tym drzewie to ich rzędy.

Rząd wierzchołków



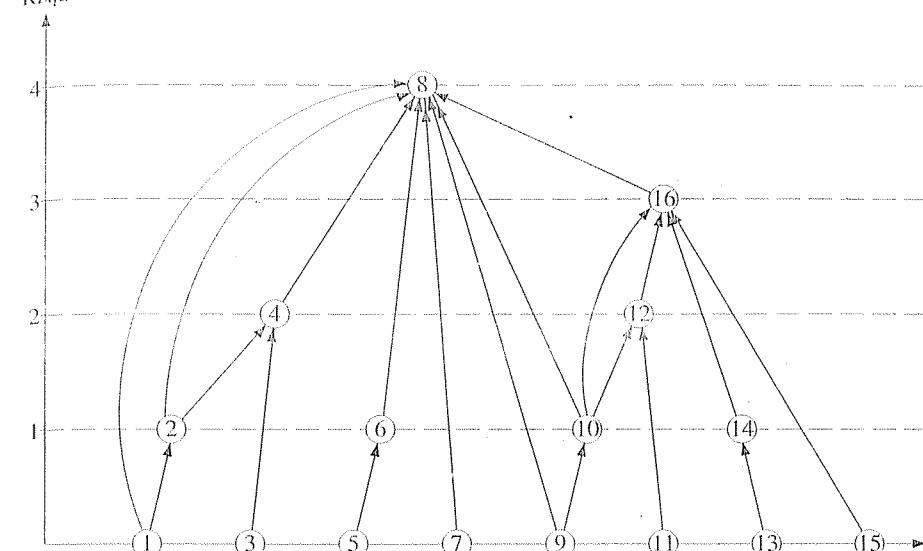
Rys. 4.5. Drzewo otrzymane bez stosowania kompresji ścieżek

Na rysunku 4.6 dla odmiany widać drzewo otrzymane z zastosowaniem kompresji ścieżek w operacjach *find*, łącznie ze wszystkimi krawędziami tworzonymi w trakcie wykonywania algorytmu. Wierzchołki są narysowane na wysokości odpowiadającej ich rzędowi.

W każdej chwili wykonywania ciągu operacji *find* i *union* rzędy wierzchołków mają następujące własności:

- dla każdego wierzchołka  $x$  rzad  $r(x)$  jest na początku równy 0, a następnie zwiększa się o 1 do czasu, aż  $p(x)$  stanie się różne od nil; kiedy do tego dojdzie,  $r(x)$  nie ulega już zmianie;
- dla każdego wierzchołka  $x$  nie będącego korzeniem  $r(x) < r(p(x))$ ;
- dla każdego wierzchołka  $x$  nie będącego korzeniem rzad  $r(p(x))$  zwiększa się przy każdej zmianie  $p(x)$ ;
- dla każdego  $k \geq 0$  liczba wierzchołków o rzędzie  $k$  wynosi co najwyżej  $n/2^k$ ;
- liczba wierzchołków w drzewie o korzeniu  $x$  wynosi co najmniej  $2^{r(x)}$ ;

Rząd wierzchołków



Rys. 4.6. Wszystkie krawędzie zostały utworzone z zastosowaniem kompresji ścieżek

- rzad każdego wierzchołka wynosi co najwyżej  $\log n$ ;
- rzędy wszystkich wierzchołków znajdują się w warstwach o numerach od 0 do  $\log^* n - 1$ .

Korzystając z podanych powyżej własności warstw funkcji  $\log^*$  i rzędu wierzchołków przeprowadzimy dowód twierdzenia Hopcrofta-Ullmana. Zastosujemy tzw. **metodę księgowania kosztu**.

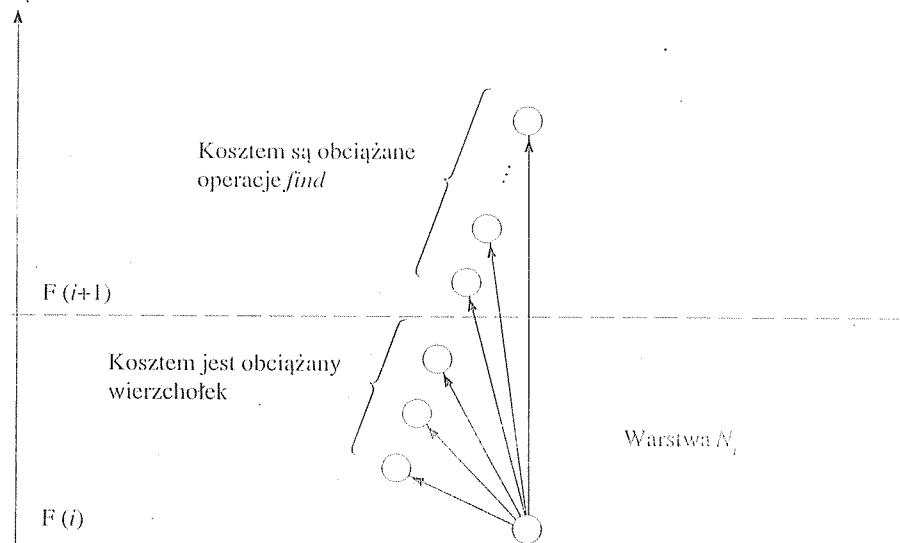
Każdej operacji przyporządkujemy pewną liczbę kredytów do pokrycia kosztu jej wykonania. Operacja *initialization* dostaje  $n$  kredytów, *union* – 1, a *find* –  $1 + \log^* n$ . Jeśli *find* wymaga większego kosztu, musi uzyskać dodatkowy kredyt z banku kredytowego. Dodatkowym kredytem są obciążane wierzchołki. Wystarczy pokazać, że łączna ilość dodatkowego kredytu, którym są obciążane wierzchołki, jest  $O(n \log^* n)$ .

Inaczej mówiąc, ze strukturą danych wiążemy potencjał  $O(n \log^* n)$  na opłacanie „nadmiernych” kosztów wykonania niektórych operacji *find*. Przyjmujemy, że operacja *find* wykorzystuje przede wszystkim przyporządkowany jej koszt, tzn. koszt zamortyzowany, a jeśli tego kosztu nie wystarczy, to część brakująca jest pokrywana z ogólnego potencjału struktury danych.

Z kosztem przydzielonego operacji *find* jest pokrywany koszt rozpatrzenia korzenia i jego bezpośredniego następnika oraz koszt rozpatrzenia każdego wierzchołka  $v$  na ścieżce związanej z realizacją *find* – takiego, że oba rzędy  $r(v)$  i  $r(p(v))$  należą do różnych warstw. Łączny koszt przydzielony operacji *find* jest zatem  $O(m \log^* n)$ .

Jeśli natomiast oba rzędy  $r(v)$  i  $r(p(v))$  należą do tej samej warstwy, to jednostką dodatkowego kredytu obciążamy wierzchołek  $v$  (rys. 4.7). Wierzchołek, którego rząd  $r(v)$  leży w  $i$ -tej warstwie, może być obciążany jednostką dodatkowego

Rząd wierzchołków



Rys. 4.7. Rozdział kosztu rozpatrywania krawędzi wychodzących z danego wierzchołka

kredytu co najwyżej  $F(i+1) - F(i) - 1$  razy (1 raz, gdy  $i = 0$ ), ponieważ wartość  $r(p(v))$  zwiększa się przy każdej kompresji  $v$  do korzenia. Obciążenie dodatkowymi kredytami wierzchołków, których rzędy leżą w  $i$ -tej warstwie, wynosi razem co najwyżej

- $n/2$  dla  $i = 0$
- 0 dla  $i = 1$
- $(F(i+1) - F(i) - 1) \sum_{q=F(i)+1}^{F(i+1)} \frac{n}{2^q} \leq F(i+1) \frac{n}{2^{F(0)+1}} \sum_{q=0}^{\infty} \frac{1}{2^q} \leq n$  dla  $i > 1$

Sumując po wszystkich warstwach, otrzymujemy oszacowanie całkowitego kredytu, którym są obciążane wierzchołki:

$$\frac{n}{2} + \sum_{i=2}^{\log^* n - 1} n = O(n \log^* n)$$

Dowód twierdzenia Hopcrofta-Ullmana został w ten sposób zakończony.

cbdo

## 4.2.

### Złączalne kolejki priorytetowe

Zbiory informacji przechowywane w komputerze mają czasami postać złączalnych ze sobą kolejek priorytetowych (elementem usuwanym ze zbioru jest element o najmniejszym bądź największym priorytecie). O kolejce priorytetowej – jako rozwiązaniu dynamicznego problemu sortowania – była już mowa w rozdziale 2. Użycie kopca zapewnia koszt  $O(\log n)$  dla operacji *insert* i *deletemin* (*deletemax*). Natomiast złączenie dwóch kopców wymaga czasu  $O(n)$ . Zajmiemy się teraz strukturą danych, zwana kolejką dwumianową, dla której wszystkie trzy operacje *insert*, *deletemin* i *union* (a także jeszcze kilka innych) dają się wykonać w czasie  $O(\log n)$ .

Elementy zbioru reprezentowanego za pomocą kolejki priorytetowej, tak jak w poprzednim punkcie, będą utożsamiane z węzłami struktury danych. Zakładamy, że każdy element (węzeł)  $x$  ma określoną wartość swojego klucza  $key(x)$ , pochodząą ze zbioru liniowo uporządkowanego.

Interesować nas będzie kolejka priorytetowa, dla elementów dynamicznego skończonego zbioru  $S$ , względem którego są wykonywane następujące operacje:

- (a) *construct*( $S$ ): utworzenie zbioru pustego  $S = \emptyset$ ;
- (b) *insert*( $x, S$ ):  $S := S \cup \{x\}$ ;
- (c) *deletemin*( $S$ ): usunięcie z  $S$  najmniejszego elementu (tzn. o najmniejszym kluczu, ewentualnie jednego z najmniejszych, gdy jest ich kilka);
- (d) *findmin*( $S$ ): wyznaczenie najmniejszego elementu w  $S$  (bez usuwania go);
- (e) *delete*( $x, S$ ):  $S := S - \{x\}$ ;
- (f) *decrease*( $x, k, S$ ): *delete*( $x, S$ );  $key(x) := k$ ; *insert*( $x, S$ ) (pod warunkiem, że  $k < key(x)$ );<sup>10</sup>
- (g) *union*( $S_1, S_2, S$ ):  $S := S_1 \cup S_2$  przy założeniu, że  $S_1$  i  $S_2$  są zbiorami rozłącznymi.

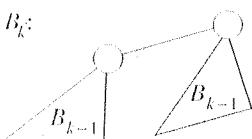
Zauważmy, że implementacja kopcową zastosowana w algorytmie heapsort (w podrozdziale 2.6) zapewnia czas  $O(\log n)$  dla wszystkich operacji z wyjątkiem (g).

Zanim zdefiniujemy kolejkę dwumianową, przytoczymy kilka innych potrzebnych pojęć:

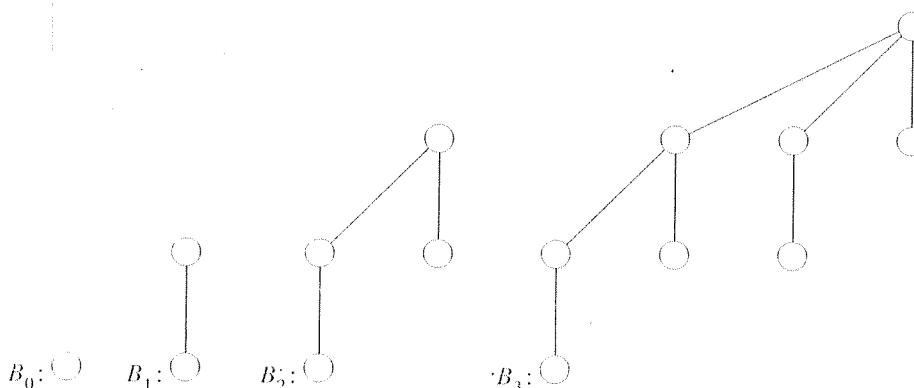
<sup>10</sup> Operacja *decrease* jest szczególnym przypadkiem operacji *change* z podrozdziału 2.6.

Drzewem dwumianowym  $B_k$  stopnia  $k \geq 0$  nazywamy

- drzewo  $B_0$  składające się z pojedynczego węzła;
- drzewo  $B_k$  powstające z dwóch drzew  $B_{k-1}$  przez dołączenie korzenia jednego z nich jako następnika korzenia drugiego (rys. 4.8).



Rys. 4.8. Struktura drzewa  $B_k$

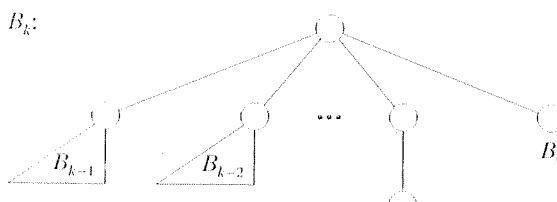


Rys. 4.9. Przykłady drzew dwumianowych dla  $k = 0, 1, 2, 3$

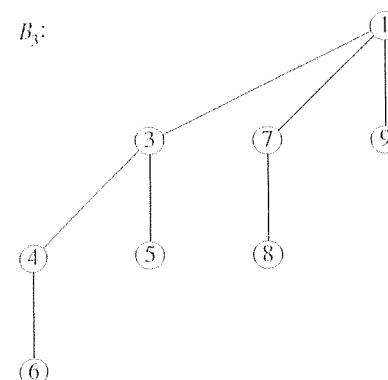
Przykłady drzew dwumianowych dla  $k = 0, 1, 2, 3$ , widać na rysunku 4.9.

Drzewa dwumianowe  $B_k$  ( $k \geq 0$ ) mają następujące własności:

- liczba węzłów w drzewie  $B_k$  wynosi  $2^k$ ;
- wysokość drzewa  $B_k$  wynosi  $k$ ;
- w drzewie  $B_k$  jest dokładnie  $\binom{k}{i}$  węzłów o głębokości  $i$  dla  $0 \leq i \leq k$ ;
- maksymalny stopień węzła w drzewie  $B_k$  wynosi  $k$ ;
- drzewo  $B_k$  daje się przedstawić tak jak na rysunku 4.10.



Rys. 4.10. Rozwinięcie drzewa  $B_k$



Rys. 4.11. Drzewo dwumianowe  $B_3$  z elementami w porządku kopcowym

Do drzewa dwumianowego elementy wpisujemy zgodnie z porządkiem kopcowym, z tym że najmniejszy element umieszczamy w korzeniu (rys. 4.11).

**Kolejką dwumianową** nazywamy zbiór drzew dwumianowych spełniających następujące warunki:

- każde drzewo dwumianowe ma uporządkowanie kopcowe;
- dla każdego  $k$  jest co najwyżej jedno drzewo dwumianowe stopnia  $k$ .

Z definicji wynikają następujące własności kolejki dwumianowej dla  $n$ -elementowego zbioru  $S$ :

- w kolejce jest  $\lfloor \log n \rfloor$  drzew dwumianowych stopnia odpowiednio

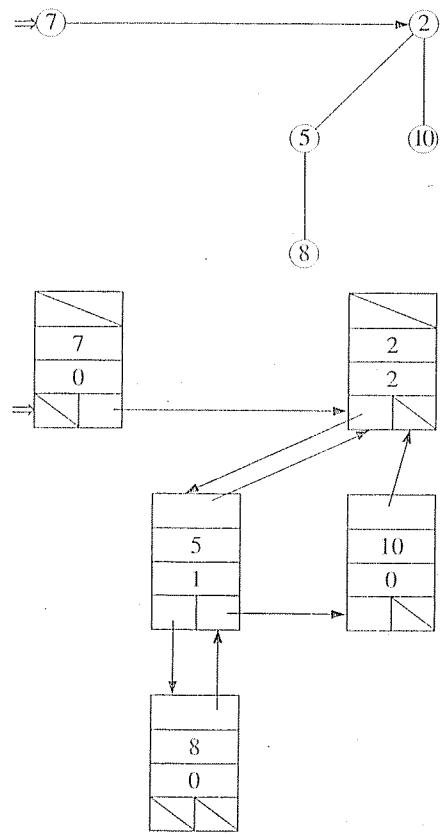
$$\lfloor \log n \rfloor = k_1 > k_2 > \dots > k_i \geq 0$$

gdzie  $n = 2^{k_1} + 2^{k_2} + \dots + 2^{k_i}$ ;

- liczba drzew w kolejce dwumianowej wynosi co najwyżej  $\lfloor \log n \rfloor + 1$ ;
- każdy węzeł ma stopień co najwyżej  $\lfloor \log n \rfloor$ ;
- najmniejszy element znajduje się w jednym z korzeni drzew dwumianowych.

Przyjmujemy następujący schemat węzła w kolejce dwumianowej (zarówno w drzewie dwumianowym, jak i w kolejce korzeni drzew dwumianowych):

- $key(x)$  – wartość klucza elementu  $x$ ,  
 $p(x)$  – poprzednik  $x$ ,  
 $child(x)$  – pierwszy następnik  $x$ ,  
 $sibling(x)$  – kolejny sąsiad  $x$  (mający ten sam poprzednik), gdy  $x$  nie jest korzeniem, bądź korzeń następnego drzewa dwumianowego w kolejce, gdy  $x$  jest korzeniem,  
 $degree(x)$  – stopień węzła  $x$ .



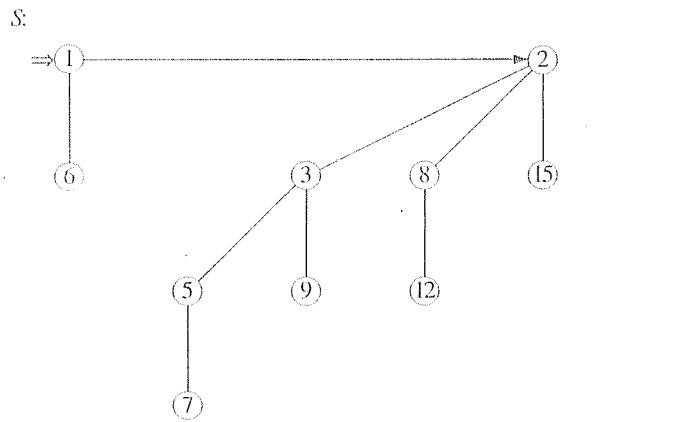
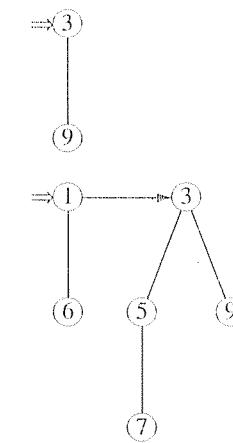
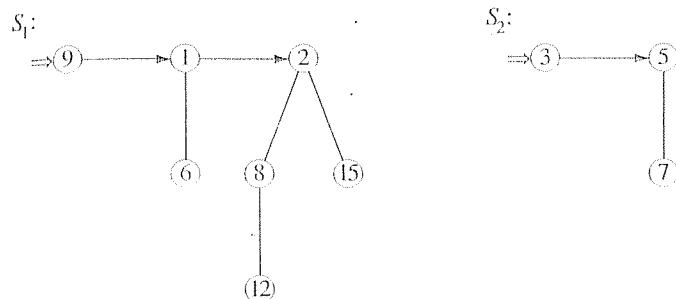
Rys. 4.12. Przykład kolejki dwumianowej i jej reprezentacji

Na rysunku 4.12 jest przedstawiona przykładowa kolejka dwumianowa i jej reprezentację w postaci struktury dowiązanej.

Zauważmy, że połączenie dwóch drzew  $B_{k-1}$  w drzewo  $B_k$  daje się wykonać w stałym czasie.

Naszkicujemy teraz implementacje operacji kolejki priorytetowej dla kolejki dwumianowej, pozostawiając Ci zapisanie ich w całości.

- $union(S_1, S_2, S)$ : Połącz drzewa dwumianowe z kolejek  $S_1$  i  $S_2$ , naśladowując binarne dodawanie liczb, tak jak to widać w przykładzie przedstawionym na rysunku 4.13; czas wykonania  $O(\log n)$ .
- $findmin(S)$ : Przejdź kolejkę korzeni drzew dwumianowych i wyznacz korzeń o najmniejszym kluczu; czas wykonania  $O(\log n)$ .



Rys. 4.13. Łączenie kopców z wykorzystaniem dodawania liczb binarnych

- *insert(x, S)::* Utwórz jednoelementową kolejkę dwumianową  $S_0$  (jedynym węzłem jest  $x$ ) i używając *union*, połącz ją z  $S$ ; czas wykonania  $O(\log n)$ .
- *deletemin(S)::*
  - Wyznacz korzeń  $x$  drzewa dwumianowego  $T$  z najmniejszym kluczem. Usuń  $T$  z głównej listy, otrzymując kolejkę dwumianową  $S'$ .
  - Usuń  $x$  z  $T$ . Odwróć porządek na liście następców  $x$ , tworząc kolejkę dwumianową  $S''$ .
  - Wykonaj *union*( $S', S'', S$ ).

Przykład działania tego algorytmu jest przedstawiony na rysunku 4.14. Operację (a) można wykonać w czasie  $O(\log n)$ , ponieważ lista korzeni kopców ma długość co najwyżej  $\lfloor \log n \rfloor + 1$ . Operację (b) też daje się wykonać w czasie  $O(\log n)$ , ponieważ każdy węzeł ma co najwyżej  $\lfloor \log n \rfloor$  następców. Operacja *union*, jak pokazaliśmy wcześniej, również daje się wykonać w czasie  $O(\log n)$ . Łączny zatem czas wykonania operacji *deletemin* jest  $O(\log n)$ .

- *decreasekey(x, k, S)::*
  - $key(x) := k$ ;
  - Wykonaj operację podniesienia  $x$  w drzewie dwumianowym tak jak dla zwykłego kopca (z algorytmu *heapsort*).

Przykład działania tego algorytmu jest widoczny na rysunku 4.15. Ścieżki do korzenia kopca mają długość co najwyżej  $\lfloor \log n \rfloor$ , a zatem koszt tej operacji też wynosi  $O(\log n)$ .

Operacja *delete* sprowadza się do wykonania najpierw operacji *decreasekey* (zwiększenia wartości klucza w węźle  $x$  do  $-\infty$ ), a następnie wykonania operacji *deletemin* (usunięcia najmniejszego elementu  $-\infty$ ). Mamy więc:

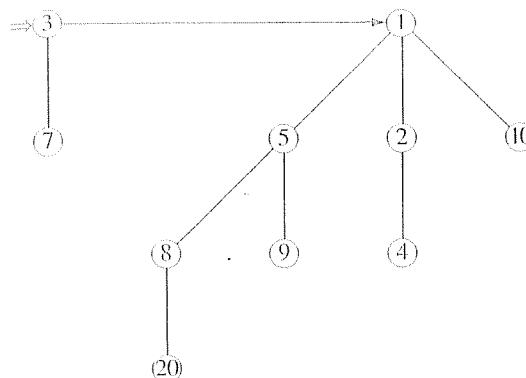
- *delete(x, S)::*
  - decreasekey(x,  $-\infty$ , S)*;
  - deletemin(S)*.

Wynika stąd, że koszt wykonania operacji *delete* też jest  $O(\log n)$ .

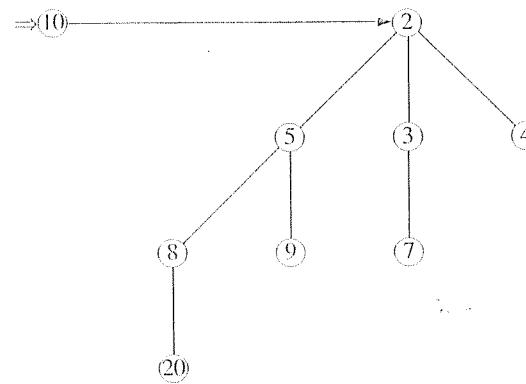
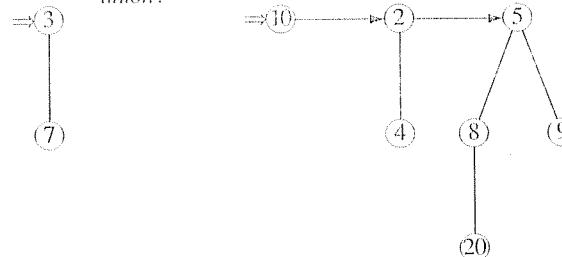
Udowodniliśmy w ten sposób, że używając kolejek dwumianowych, każda z operacji kolejki priorytetowej – *insert*, *findmin*, *deletemin*, *union*, *decreasekey* i *delete* – można wykonać w czasie  $O(\log n)$ .

Kolejki priorytetowe mają zastosowanie przy rozwiązywaniu wielu problemów. W rozdziale dotyczącym algorytmów grafowych dowiesz się, że używa się ich do wyznaczania minimalnego drzewa rozpinającego dany graf i do wyznaczania ścieżek o najmniejszej długości, łączących dany wierzchołek z pozostałymi. Korzysta się z nich także – co ważne – przy wyznaczaniu optymalnego kodu dwójkowego dla alfabetu, w którym symbolom przyporządkowano częstości ich wystąpień w komunikatach (zob. rozdział o algorytmach tekstowych).

*deletemin(S)::*

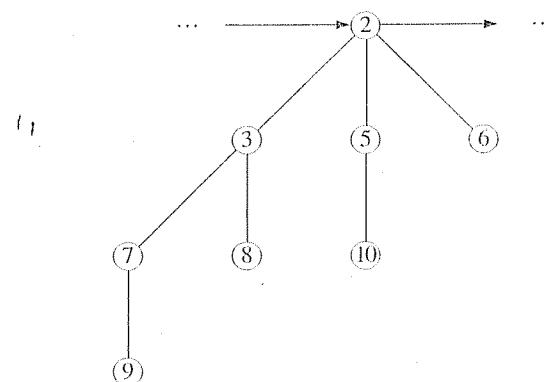
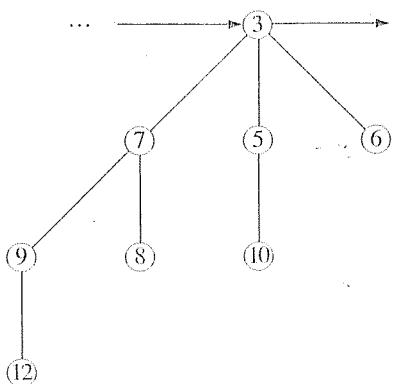


*union::*



Rys. 4.14. Usunięcie najmniejszego elementu zostaje sprowadzone do złączenia dwóch kolejek dwumianowych

Warto jeszcze wspomnieć o strukturze danych, będącej modyfikacją kolejki dwumianowej. Chodzi o kopiec Fibonacciego. Zamiast w operacji *union* od razu łączyć poszczególne drzewa w kolejce, łączy się jedynie listy. Do właściwego złączenia drzew dochodzi dopiero wtedy, kiedy trzeba wykonać operację *deletemin*. Otrzymujemy strukturę danych,

decrease( $x, 2, S$ ):Rys. 4.15. Operacja *decreasekey* polega na podniesieniu węzła w drzewie dwumianowym

w której koszt zamortyzowany wszystkich operacji kolejki priorytetowej jest  $O(1)$ . Wyjątek stanowią operacje *deletemin* i *delete*, których koszt jest  $O(\log n)$ . Zainteresowanego czytelnika odsyłamy do książki [CLR].

## Zadania

4.1. Korzystając z 2-3 drzew (zdefiniowanych w zadaniu 3.15), opracuj implementację kolejki priorytetowej, w której każda z operacji *insert*, *deletemin*, *union*, *delete* i *change* daje się wykonać w czasie  $O(\log n)$ .

4.2. Zaproponuj efektywną strukturę danych do wykonywania ciągów podanych tu operacji na dynamicznie zmieniającym się podziale zbioru  $\{1, 2, \dots, n\}$ :

- (a) *initialization*:: utworzenie podziału  $\{1\}, \{2\}, \dots, \{n\}$ ;
- (b) *union*( $A, B$ ):: połączenie dwóch zbiorów  $A$  i  $B$  bieżącego podziału w jeden;

## Zadania

## 163

- (c) *find*( $x$ ):: wyznaczenie zbioru, do którego należy element  $x$ ;
- (d) *findmin*:: wyznaczenie zbioru bieżącego podziału, który ma najmniej elementów.

Jaka jest złożoność operacji?

4.3. Używając algorytmu z przykładu z początku rozdziału, wyznacz spójne składowe grafu  $G = (V, E)$ , gdzie:

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$E = \{(1, 2), (3, 4), (5, 6), (7, 8), (1, 9), (10, 3), (4, 7), (8, 10)\}$$

Wykonaj obliczenia, raz posługując się implementacją listową, a raz posługując się implementacją drzewową z kompresją ścieżek w operacji *find*.

4.4. [AHU] Zaprojektuj strukturę danych, reprezentującą zbiory drzew (lasy), w której można efektywnie wykonywać następujące operacje:

- (a) *initialization*( $n$ ):: utworzenie  $n$  jednoelementowych drzew; zapisanie elementu  $i$  w korzeniu  $i$ -tego drzewa;
- (b) *link*( $x, y$ ):: przyłączenie krawędzią wierzchołka  $x$  (będącego korzeniem) do wierzchołka  $y$  (znajdującego się w innym drzewie niż  $x$ );
- (c) *depth*( $y$ ):: obliczenie głębokości wierzchołka  $y$ .

Ciąg  $k$  operacji *link* ( $k \leq n - 1$ ) i  $m$  operacji *depth* powinien dać się wykonać w czasie  $O((m + n)\log n)$ .

4.5. [AHU] Zaprojektuj strukturę danych, reprezentującą zbiory drzew (lasy), w której można efektywnie wykonywać następujące operacje:

- (a) *initialization*( $n$ ):: utworzenie  $n$  jednoelementowych drzew; zapisanie elementu  $i$  w korzeniu  $i$ -tego drzewa;
- (b) *link*( $x, y$ ):: przyłączenie krawędzią wierzchołka  $x$  (będącego korzeniem) do wierzchołka  $y$  (znajdującego się w innym drzewie niż  $x$ );
- (c) *lca*( $x, y$ ):: wyznaczenie najbliższego wspólnego przodka  $x$  i  $y$ .

Ciąg  $k$  operacji *link* ( $k \leq n - 1$ ) i  $m$  operacji *lca* powinien dać się wykonać w czasie  $O((m + n)\log^* n)$ .

4.6. Udowodnij, że w wypadku implementacji drzewowej z kompresją ścieżek koszt wykonania ciągu operacji: *initialization*,  $n - 1$  operacji *union*, a następnie  $m$  operacji *find*, jest  $O(m + n)$ .

4.7. Dla danej wartości  $n$  skonstruj ciąg operacji *union* tworzących drzewo o  $n - 1$  elementach i wysokości  $\lfloor \log n \rfloor$ .

# 5 Algorytmy tekstowe

Niech  $I$  będzie pewnym ustalonym skończonym zbiorem symboli (liter). O ile nie jest to wyraźnie zaznaczone, przyjmujemy na ogół, że zbiór  $I$  jest ograniczony przez pewną stałą. Zbiór ten nazywamy alfabetem, a ciągi symboli – tekstami lub słowami. Długość tekstu to liczba jego symboli. W szczególności tekst o długości zero składa się z zerowej liczby symboli; nazywamy go słowem pustym. Długość tekstu  $x$  oznaczymy przez  $|x|$ .

Dwie typowe reprezentacje tekstów to reprezentacja listowa i reprezentacja tablicowa. Reprezentacja listowa jest użyteczna zwłaszcza wtedy, kiedy dokonujemy operacji wstawiania i usuwania części tekstu. Reprezentacja tablicowa jest bardziej naturalna dla statycznych operacji, polegających na sprawdzaniu pewnych własności lub obliczaniu pewnych parametrów dla jednego lub wielu tekstów. Przyjmiemy drugą reprezentację, ponieważ w rozdziale tym będziemy się zajmować właśnie tego typu problemami.

Danymi wejściowymi będzie jeden lub kilka tekstów. Jako rozmiar  $n$  danych wejściowych będziemy przyjmować maksymalną długość tekstu, jeżeli będziemy mieć do czynienia z co najwyżej dwoma tekstami, lub sumę długości tekstów wejściowych w wypadku większej liczby tekstów.

Podstawowym problemem dotyczącym tekstów jest problem **wyszukiwania wzorca**. W dalszych rozważaniach będziemy go nazywać WW.

Problem WW polega na znalezieniu wszystkich wystąpień tekstu  $x$ , zwanego wzorcem, w tekście  $y$ . Przyjmujemy, że  $|x| = m$ ,  $|y| = n$  oraz  $n \geq m$ .

Problem WW łatwo jest uogólnić na przypadek dwuwymiarowy. Teksty  $x$  i  $y$  są wtedy tablicami składającymi się z symboli, a naszym zadaniem jest znalezienie wszystkich wystąpień  $x$  jako podtablicy  $y$ .

Problem WW daje się rozwiązać w czasie liniowym ze względu na całkowity rozmiar wejścia. (W przypadku dwuwymiarowym zakładamy, że tablice  $x$  i  $y$  są kwadratowe, a więc rozmiar wejścia jest  $O(n^2)$ ).

Wyszukiwanie wzorca było wszechstronnie badane ze względu na duże zastosowanie praktyczne. Przedstawimy kilka algorytmów dla problemu WW: N (algorytm „naiwny”; bezpośredni naturalny algorytm o złożoności teoretycznej zbyt dużej), KMP (algorytm Knutha-Morrisa-Pratta), KMR (algorytm Karpa-Millera-Rosenberga), KR (algorytm Karpa-Rabina), GS' (uproszczona wersja algorytmu Galila-Seifera), działająca jedynie dla szczególnej klasy wzorców) oraz BM (algorytm Boyera-Moore'a) i jego modyfikacje BM', działającą w czasie oczekiwany rzędu istotnie mniejszego niż liniowy. Omówimy ponadto rozszerzenie algorytmu KMP na przypadek wzorców dwuwymiarowych.

Każdy z tych algorytmów ma pewne wyróżniające go własności. Najbardziej skomplikowanym algorytmem dla problemu WW jest algorytm Galila-Seifera, który minimalizuje jednocześnie koszt czasowy i pamięciowy. My zajmiemy się pewną prostszą jego wersją – algorymem GS' (pełną znajdziesz w książce [BKR]).

Rozważymy również problem WW' dotyczący wyszukiwania wzorca z symbolami nieznaczącymi (symbole takie pasują do każdego innego symbolu). Przedstawimy algorytm FP (Fishera-Patersona) rozwiązania dla tego problemu.

## 5.1. Problem wyszukiwania wzorca

Niech  $x$  będzie wzorcem, a  $y$  tekstem, w którym szukamy wzorca. Dla słowa  $z$  przyjmijmy  $z[i..j] = z[i]z[i+1]..z[j]$ , gdzie  $i \leq j$ . Mówimy, że  $x$  występuje w  $y$  na pozycji  $i$ , gdy  $y[i..i+m-1] = x$  – inaczej mówiąc, gdy począwszy od  $i$ -tej pozycji, wzorzec „pasuje” do tekstu. Z tego powodu problem WW jest również nazywany **problemem dopasowywania wzorca**.

### 5.1.1. Algorytm N („naiwny”)

Schemat najbardziej bezpośredniego algorytmu, zwanego „naiwnym”, wygląda następująco:

```
begin
    i := 1;
    while i ≤ n - m + 1 do
        begin
            if x[1..m] = y[i..i+m-1] then write(i); i := i + 1
        end;
    end;
```

Pelny algorytm otrzymamy, rozpisując instrukcję sprawdzenia równości tekstów. Przyjmijmy dla uproszczenia, że  $x[m+1]$  i  $y[n+1]$  są specjalnymi, różnymi symbolami. Zabezpieczamy się w ten sposób przed błędem wynikającym z przekroczenia zakresu tablicy. Przyjmijmy również następujące założenie: jeżeli w czasie sprawdzania warunku logicznego  $\alpha$  zostaje przekroczyony zakres tablicy, warunkowi  $\alpha$  jest przypisywana wartość false i obliczenie jest kontynuowane.

```

 $\{$  Algorytm N (algorytm „naiwny”; zgodność wzorca jest sprawdzana od
  początku wzorca}
  begin
    i := 1;
    while i ≤ n - m + 1 do
      begin
        j := 0; while x[j + 1] = y[i + j] do j := j + 1;
        if j = m then write(i); i := i + 1; {przesunięcie = 1}
      end
    end {algorytm N};

```

Liczba wykonanych porównań symboli może być rzędu  $n^2$ , na przykład dla  $x = 0^{n/2}1$  i  $y = 0^{n-1}1$  (w tym wypadku  $m = n/2 + 1$ ; założymy, że  $n$  jest parzyste). Jednakże średnia złożoność algorytmu N jest liniowa. Oszacujmy ją dla przypadku, gdy alfabet jest dwuliterowy (dla większej liczby liter jest analogicznie).

Niech teksty  $x$  i  $y$  będą losowymi tekstami. Wówczas prawdopodobieństwo, że test  $x[j+1] = y[i+j+1]$  będzie pozytywny wynosi 1/2. Łatwo już teraz policzyć, że oczekiwana liczba takich testów, które wykonamy dla ustalonego  $i$ , nie przekracza 2. Średnia złożoność (mierzona liczbą porównań) algorytmu N nie przekracza zatem  $2n$ .

### 5.1.2.

#### Algorytm KMP (Knutha-Morrisa-Pratta)

W algorytmie N początek wzorca przesuwamy zawsze o jeden, podezas gdy możliwe jest czasami dużo większe przesunięcie. Taką wielkość przesunięcia można obliczyć, korzystając z informacji o maksymalnej wartości  $j$  (w algorytmie informacja ta w ogóle nie jest wykorzystywana przy dopasowywaniu wzorca do następnej pozycji  $i$ ; wartość  $j$  jest zerowana).

Niech  $P[j]$ , gdzie  $j > 1$ , będzie maksymalną długością właściwego sufiksu słowa  $x[1..j]$ , będącego jednocześnie jego prefiksem. Formalnie

$$P[j] = \max \{0 \leq k < j; x[1..k] \text{ jest sufiksem } x[1..j]\}.$$

Przyjmujemy  $P[1] = P[0] = 0$ . Jeśli tablica  $P$  jest już policzona, to problem WW można efektywnie rozwiązać za pomocą poniższego algorytmu KMP, którego struktura jest

### 5.1. Problem wyszukiwania wzorca

podobna do struktury algorytmu N. Wartość przesunięcia początku wzorca w  $y$  jest określona teraz wzorem:  $\text{przesunięcie}(j) = \max(1, j - P[j])$ .

```

{Algorytm KMP (Knutha-Morrisa-Pratta)}
begin
  i := 1; j := 0;
  while i ≤ n - m + 1 do
    {x[1..P[j]] = x[j - P[j] + 1..j] = y[i..i + P[j] - 1]}
    begin
      j := P[j];
      while x[j + 1] = y[i + j] do j := j + 1;
      if j = m then write(i);
      i := i + przesunięcie(j);
    end
  end {algorytm KMP};

```

W algorytmie tym jest wykonywanych co najwyżej  $2n$  porównań symboli. Wystarczy oszacować liczbę wykonanych instrukcji  $j := j + 1$ . Jeśli tą liczbą jest  $p$ , to wykonujemy co najwyżej  $n + p$  porównań (liczba  $p$  odpowiada liczbie wszystkich porównań mających wynik pozytywny; liczba porównań z wynikiem negatywnym jest oczywiście ograniczona przez  $n - m + 1$ ). Rozważmy wartość sumy  $s = i + j$ . Wartość ta nie zmniejsza się (obserwowana tuż przed wykonaniem porównania). Największą możliwą wartością  $s$  jest  $n$ , a za każdym razem, gdy wykonujemy  $j := j + 1$ ,  $s$  zwiększa się o 1. Instrukcja  $j := j + 1$  jest zatem wykonywana co najwyżej  $n$  razy. Złożoność (pesymistycznego przypadku) algorytmu KMP jest liniowa.

Pozostaje jeszcze problem obliczenia tablicy  $P$ . Dla każdego  $j \geq 2$  obliczymy  $P[j]$ , korzystając z następującej obserwacji: niech  $i \geq 1$  będzie minimalne, takie że  $x[P^{(i)}[j-1] + 1] = x[j]$ . Jeśli takie  $i$  istnieje, to  $P[j] = P^{(i)}[j-1] + 1$ ; w przeciwnym razie  $P[j] = 0$  (gdzie  $P^{(i)}$  oznacza  $i$ -krotne złożenie funkcji  $P$ ).

```

{Algorytm obliczania tablicy P}
begin
  P[0] := P[1] := 0; t := 0;
  for j := 2 to m do
    begin {obliczamy wartość P[j]}
      t := P[j - 1];
      while (t > 0) and (x[t + 1] ≠ x[j]) do t := P[t];
      if x[t + 1] = x[j] then t := t + 1; P[j] := t;
    end
  end {algorytm obliczania P};

```

Złożoność tego algorytmu można oszacować, stosując zasadę magazynu. Hekroć wykonyemy  $t := P[i]$ , pobieramy z magazynu niezerową liczbę przedmiotów  $t - P[t]$ . Do

magazynu wkładamy jeden przedmiot, gdy wykonujemy  $t := t + 1$ . Początkowo magazyn jest pusty. Inaczej mówiąc, wartość  $t$  interpretujemy jako liczbę przedmiotów w pewnym magazynie. Oczywiście w sumie włożymy co najwyżej  $m$  przedmiotów, a zatem wyjmować będziemy je również co najwyżej  $m$  razy. Wynika stąd, że liczba wykonania instrukcji  $t := P[t]$  nie przekracza  $m$ , a więc złożoność całego algorytmu jest liniowa względem  $m$ .

Tablice  $P$  można użyć w sposób bardziej bezpośredni (choć bardziej abstrakcyjny) do rozwiązywania problemu WW. Weźmy tekst  $u = x\$y$ , gdzie  $\$$  jest specjalnym znakiem występującym tylko na jednej pozycji. Obliczymy tablicę  $P$  dla tekstu  $u$ . Związek  $P[i + m + 1] = m$  zachodzi zatem wtedy i tylko wtedy, gdy  $x$  występuje w  $y$  począwszy od pozycji  $i - m + 1$  dla  $m \leq i \leq n$ .

Algorytm KMP ma bardzo istotną zaletę, a mianowicie jego złożoność nie zależy od rozmiaru alfabetu wejściowego (jest liniowa ze względu na  $n$ , i to nawet wtedy, kiedy alfabet ma rozmiar rzędu  $n$ ).

W oryginalnej wersji algorytmu KMP zamiast tablicy  $P$  używa się nieco bardziej skomplikowanej (do zdefiniowania i obliczenia) tablicy  $NEXT$ , w której  $NEXT[j]$  jest maksymalną długością  $k$  właściwego sufiku słowa  $x[1..j]$ , będącego jednocześnie jego prefiksem (jak dotyczyła te same własności spełniała wartość  $P[j]$ ), i taka, że  $x[k + 1] \neq x[j + 1]$ , jeśli  $0 < j < m$ . Przyjmijmy  $NEXT[0] = P[0] = 0$ .

Załóżmy, że w chwili zwiększania  $j$  wczytujemy kolejny symbol tekstu wejściowego  $y$ . Podobnie jak przednio przyjmujemy, że  $x[m + 1]$  jest nowym specjalnym symbolem (różnym od  $\#$ ).

Przez KMP' oznaczmy taką wersję algorytmu KMP, w której tablicę  $P$  zastąpiono tablicą  $NEXT$ .

```
{Algorytm KMP' – wersja on-line algorytmu KMP;
"na wejściu" jest ciąg  $n$  kolejnych symboli tekstu  $y$ , zakończony specjalnym symbolem #}
begin
  i := 1; j := 0; read(symbol);
  while symbol ≠ '#' do
    begin
      j := NEXT[j];
      while x[j + 1] = symbol do begin j := j + 1; read(symbol) end;
      if j = 0 then read(symbol);
      if j = m then write(i);
      i := i + max(1, j - NEXT[j]);
    end
  end
end {algorytm KMP'};
```

### 5.1. Problem wyszukiwania wzorca

Poprawność i rząd złożoności algorytmu nie ulega zmianie. Algorytm KMP' jest trochę szybszy (dla pewnych danych). Ma pewną zaletę, użyteczną w sytuacjach, kiedy cały tekst y nie jest dany od razu, a mamy jedynie kolejne jego symbole on-line. Kolejne symbole tekstu y pobieramy w czasie wykonywania instrukcji  $read(symbol)$ . Interesuje nas odpowiedź na pytanie: ile maksymalnie czasu upływa między kolejnymi instrukcjami czytania  $read(symbol)$ . Oznaczmy ten maksymalny czas przez  $delay(m)$ , a przez  $delayP(m)$  – maksymalny czas przerwy między czytaniami, gdyby zamiast  $NEXT$  była wykorzystywana poprzednia tablica  $P$ . Założymy dla uproszczenia, że alfabet wejściowy jest dwuelementowy. Drastyczną różnicę między wielkością  $delay(m)$  a  $delayP(m)$  widać na przykładzie  $x = a^m$ ,  $y = a^{m-1}ba$ . Założymy, że wczytaliśmy właśnie symbol 'b' i że  $j = m - 1$ . Mamy wtedy  $NEXT[m - 1] = 0$ , a więc kolejne czytanie odbywa się w następnym przebiegu instrukcji „dopóki” (while). Gdybyśmy zastosowali tablicę  $P$ , to zanim otrzymalibyśmy  $j = 0$ , musielibyśmy wykonać liniową liczbę iteracji. Wynika stąd, że  $delayP(m) = \Omega(m)$ . Jeżeli jednak używamy tablicy  $NEXT$ , to mamy

$$delay(m) = O(\log m)$$

Dla pewnych wzorców zachodzi ponadto  $delay(m) = \Omega(\log m)$ . Przykładem takich wzorców są słowa Fibonacciego (zdefiniowane dalej). Dla wzorców Fibonacciego o długości  $F_k$  oszacowanie  $delay(m) = \Omega(\log m)$  wynika ze wzoru:

$$NEXT[F_k - 1] = F_{k-1} - 1$$

gdzie  $F_k$  jest  $k$ -tą liczbą Fibonacciego ( $F_1 = F_2 = 1$ ,  $F_{k+2} = F_{k+1} + F_k$ ).

Oznaczmy  $k$ -te słowo Fibonacciego przez  $fib_k$ . Niech  $fib_1 = b$ ,  $fib_2 = a$  i niech

$$fib_{k+2} = fib_{k+1}fib_k \quad \text{dla } k > 0$$

Mamy wtedy  $fib_3 = ab$ ,  $fib_4 = aba$ ,  $fib_5 = abaab$ ,  $fib_6 = abaababa$ .

Słowa Fibonacciego mają wiele ciekawych własności. Po obcięciu na przykład dwóch ostatnich symboli stają się słowami symetrycznymi. Zachodzi również równanie

$$fib_{k+1}fib_k = fib_kfib_{k+1}$$

z dokładnością do zmiany kolejności dwóch ostatnich symboli.

#### 5.1.3.

### Algorytm liniowy dla problemu wyszukiwania wzorca dwuwymiarowego, czyli algorytm Bakera

Pokażemy zastosowanie algorytmu KMP do rozwiązywania problemu szukania wzorca dwuwymiarowego. Istotną własnością algorytmu KMP będzie tu niezależność kosztu od rozmiaru alfabetu (alfabet będzie rzędu  $m$ ).

Załóżmy, że mamy dane tablice  $x: \text{array}[1..m, 1..m] \text{ of } \text{char}$  i  $y: \text{array}[1..n, 1..n] \text{ of } \text{char}$ , gdzie  $n > m$ . Przypuśćmy, że mamy  $r$  różniących się między sobą kolumn tablicy-wzorca  $x$ . Oznaczmy je przez  $x_1, x_2, \dots, x_r$ . Każda kolumna jest tekstem długości  $m$ . Zastanówmy się najpierw nad rozszerzeniem algorytmu KMP na przypadek szukania wielu wzorców. Niech  $X = \{x_1, x_2, \dots, x_r\}$  będzie zbiorem wzorców o tej samej długości  $m < n$ . Dla każdej pozycji  $i$  w tekście chcemy nie tylko sprawdzić, czy któryś ze wzorców zaczyna się od tej właśnie pozycji, ale również znać numer takiego wzorca. Reprezentujemy zbiór  $X$  drzewem  $T$ , którego krawędzie są etykietowane symbolami. Każdy węzeł odpowiada prefiksowi pewnego wzorca, a prefiks jest dany ciągiem etykiet na ścieżce od korzenia do tego węzła. Korzeń drzewa  $root$  odpowiada prefiksowi pustemu, a liście drzewa – wzorców. Tablicę  $P$  definiujemy podobnie jak w algorytmie KMP. Dla węzła  $v \neq root$  (będącego prefiksem niepustym pewnego wzorca)  $P[v]$  jest maksymalnym właściwym sufiksem  $u$  słowa  $v$ , który jest prefiksem pewnego wzorca ze zbioru  $X$ . Tablicę  $P$  wyznaczamy podobnie jak w przypadku jednowymiarowego wzorca. Koszt jest liniowy względem sumarycznej długości wszystkich wzorców. Jako zadanie pozostawiamy Ci opracowanie dokładnej konstrukcji tego algorytmu.

Mając dane drzewo  $T$  i tablicę  $P$ , szukamy wzorców w tekście  $y$  podobnie jak w algorytmie KMP. Tym razem „dopasowujemy” maksymalną ścieżkę w drzewie  $T$  do tekstu. Czytając kolejne symbole tekstu, poruszamy się ścieżką, której etykiety krawędzi odpowiadają czytanym symbolom. Jeśli kolejny symbol w tekście nie jest etykietą żadnej krawędzi wychodzącej z bieżącego węzła drzewa  $T$ , to stosujemy tablicę  $P$ . Otrzymujemy algorytm umożliwiający policzenie w czasie  $O(n)$  dla każdej pozycji  $i$  numeru wzorca, którego wystąpienie zaczyna się na tej pozycji (lub stwierdzenie braku wystąpienia). Jego twórcami są A. Aho i A. Corasic. Opracowanie dokładnej konstrukcji tego algorytmu, który będziemy dalej nazywać algorytmem AC, pozostawiamy Ci jako zadanie.

Przejdźmy ponownie do szukania dwuwymiarowego wzorca. Niech  $y_1, y_2, \dots, y_n$  będą kolumnami tablicy-tekstu  $y$ . Wprowadzamy nowy alfabet  $\{0, 1, 2, \dots, r\}$ . Stosujemy algorytm AC i w każdej pozycji kolumny  $y_i$  wpisujemy numer wzorca ze zbioru  $X$  kolumn, który zaczyna się na tej pozycji („chodzimy po” kolumnach z góry na dół). Jeśli wzorzec nie występuje, to wpisujemy 0 i ostatnich  $m-1$  wierszy tablicy  $y$  odcinamy. W ten sposób z tablicy  $y$  otrzymujemy tablicę  $y^\#$ , której elementami są numery kolumn tablicy  $x$  lub zera.

Teraz istotnie korzystamy z tego, że algorytm KMP ma koszt niezależny od rozmiaru alfabetu. Niech  $j_1, j_2, \dots, j_m$  będą numerami (odpowiadającymi kolejnością słów ze zbioru  $X$ ) kolumn tablicy  $x$ . Tworzymy wzorzec  $x^\# = j_1 j_2 \dots j_m$ . W każdym wierszu tablicy  $y^\#$  szukamy wzorca  $x^\#$ . Wystąpienie tego wzorca w  $i$ -tym wierszu na  $j$ -tej pozycji oznacza, że tablica  $x$  „pasuje” do tablicy  $y$ , począwszy od pozycji  $(i, j)$ , a zatem możemy  $x$  przyłożyć do  $y$  tak, że lewy górnego róg  $x$  wypada na pozycji  $(i, j)$ .

Pokażemy działanie tego algorytmu na następującym przykładzie. Niech  $x, y$  będą (odpowiednio) tablicami:

$b\ a$	$a\ b\ a$
$a\ b$	$b\ a\ b$
	$a\ b\ b$

Numerujemy kolumny tablicy  $x$  przez 1, 2. Tablicę reprezentuje teraz tekst o długości 12 nad alfabetem  $\{0, 1, 2\}$ . Szukamy wzorców  $ba$  i  $ab$  w kolumnach tablicy  $y$  i zastępujemy elementy tej tablicy numerami wzorców, które zaczynają się na danej pozycji (z góry na dół w kolumnie). Ostatni wiersz  $y$  ocinamy. Otrzymujemy następującą tablicę  $y^\#$ :

2	1	2	1	2	1	2	1	2	1	2	1
1	2	0									

Wiemy teraz, że  $x$  zaczyna się na drugiej pozycji pierwszego wiersza i pierwszej pozycji drugiego wiersza, gdyż w tych miejscach występuje wzorzec 12.

Calkowity koszt algorytmu jest  $O(n^2)$ . Algorytm ten nazywamy algorytmem Bak, od skrótu nazwiska jego twórcy T. P. Bakera.

#### 5.1.4.

#### Algorytm GS' (wersja algorytmu Galila-Seiferasa dla pewnej klasy wzorców)

Jednym z najciekawszych algorytmów dla problemu WW jest algorytm GS (Galila-Seiferasa), który umożliwia rozwiązywanie tego problemu w czasie liniowym i jednocześnie w pamięci  $O(1)$ . Jest w nim używanych tylko kilka zmiennych całkowitych (o zakresie  $[0..n]$ ). Przez pewien czas uważano nawet, że algorytm taki w ogóle nie jest możliwy. Podamy tu pewną bardzo słabą jego wersję (algorytm GS'). Pełny algorytm znajdziesz w książce [BKR].

Mówimy, że wzorzec  $x$  jest łatwy, gdy żadne słowo niepuste postaci  $vvv$  nie jest jego prefiksem (na przykład wzorzec  $x = abababba$  nie jest łatwy, a  $x = abbababba$  tak).

W wypadku łatwych wzorców można skonstruować algorytm podobny do algorytmu KMP, w którym tablicę  $P$  zastępuje się przez pewne przybliżone oszacowania:  $P[j] < 2j/3$ , przesunięcie( $j$ )  $\geq j/3$ .

```
{Algorytm GS'; wersja algorytmu Galila-Seiferasa dla łatwych wzorców}
begin
  i := 1;
  while i ≤ n - m + 1 do
    begin
      j := 0;
      while x[j + 1] = y[i + j + 1] do j := j + 1;
      if j = m then write(i);
      i := i + max(1, ⌈ j/3 ⌉);
    end
  end {algorytm GS'};
```

Poprawność algorytmu dla łatwych wzorców wynika wprost z definicji łatwego wzorca. Złożoność algorytmu jest liniowa. Wystarczy pokazać, że liczba wykonanych instrukcji  $j := j + 1$  jest liniowa. Rozważmy sumę  $s = 3i + j$ . Można przeprowadzić podobne rozumowanie co przy omawianiu algorytmu KMP.

Algorytm GS' dla wzorców  $x$ , które nie są łatwe, może dać odpowiedź niepoprawną. Dla  $x = aaaaauaaab$  i  $y = aaaaaaaaab$  na przykład zostanie wypisanych zero wystąpień wzorca  $x$ , podczas gdy  $x$  występuje w  $y$  począwszy od pozycji  $i = 2$ . Dla  $j = 7$  otrzymamy przesunięcie równe 2 i następną badaną pozycję w  $y$  będzie  $i = i + 2 = 3$ , a pozycja  $i = 2$  zostanie pominięta jako możliwa początkowa pozycja wystąpienia  $x$  w  $y$ .

Przykładem interesującego zbioru łatwych wzorców jest zbiór słów Fibonacciego  $\{fib_1, fib_2, \dots\}$ . Dowód tego, że słowa Fibonacciego są łatwymi wzorcami, pozostawiamy Ci jako zadanie. Możesz skorzystać z następującego wzoru na elementy tablicy  $P$  (dla wzorców Fibonacciego):

$$P[j] = j - F_{k-1} \quad \text{dla } F_k \leq j < F_{k+1}, \quad \text{gdzie } k > 1$$

### 5.1.5.

#### Algorytm KMR (Karpa-Millera-Rosenberga)

Algorytm KMR (Karpa-Millera-Rosenberga) można zastosować do problemu WW w sposób pośredni. Algorytm ten umożliwia wykrycie wszystkich możliwych powtórzeń tego samego tekstu w danym słowie. Bezpośrednim zastosowaniem algorytmu KMR jest obliczanie najdłuższego powtarzającego się podslowa tekstu wejściowego  $x$  w czasie  $O(n \log n)$ , gdzie  $|x| = n$ . Problem WW daje się jednak też rozwiązać tym algorytmem w tym samym czasie, a przejście od przypadku jedno- do dwuwymiarowego jest natychmiastowe (przy stosowaniu rozszerzenia algorytmu KMP do przypadku dwuwymiarowego przejście takie jest bardziej skomplikowane). Widzimy więc, że algorytm jest gorszy od KMP o czynnik  $\log n$ ; „nadrabia” to jednak prostotą i możliwościami zastosowania w praktyce.

Załóżmy, że słowem wejściowym jest tekst  $w$  i że mamy zadaną pewną liczbę  $r$ . Jeżeli wszystkimi różnymi podslowami długości  $r$  w tekście  $w$  są słowa  $x_1, x_2, \dots, x_k$  (w kolejności leksykograficznej), to dla każdej pozycji  $1 \leq i \leq |w| - r + 1$  algorytm KMR umożliwia obliczenie wartości  $\text{numer}[i]$  równej numerowi podslowa  $w[i..i+r-1]$ . Równość  $\text{numer}[i] = k$  zachodzi wtedy, kiedy  $x_k = w[i..i+r-1]$ . Inaczej mówiąc, algorytm umożliwia obliczenie klas równoważności. Dwie pozycje  $i, j: 1 \leq i$  oraz  $j \leq |w| - r + 1$ , są równoważne wtedy, kiedy  $\text{numer}[i] = \text{numer}[j]$ . Dla  $w = ababab$  i  $r = 4$  na przykład mamy  $x_1 = abab$ ,  $x_2 = baba$  oraz (przykładowo)  $\text{numer}[3] = 1$ .

Pokażemy teraz, jak zastosować tego typu obliczenia do wyszukiwania wzorca. Weźmy  $w = yx$  i  $r = m$ . Przyjmijmy  $x_k = x$ , gdzie  $k$  jest numerem wzorca. Mamy  $\text{numer}[i] = k$

wtedy i tylko wtedy, gdy wzorzec  $x$  występuje w  $y$  na pozycji dla  $1 \leq i \leq n - m + 1$ . Algorytm KMR umożliwia zatem także rozwiązanie problemu WW.

Załóżmy dla uproszczenia, że  $r$  jest potęgą dwójki (tutaj założenie takie nie jest równie naturalne co zazwyczaj). Niech  $\text{numer}_p[i]$  będzie numerem podslowa o długości  $p$ , zaczynającego się od pozycji  $i$  w tekście  $w$ . Tablica  $\text{numer}_p$  ma długość  $|w| - p + 1$ .

Algorytm KMR umożliwia obliczenie tablic  $\text{numer}_p$  kolejno dla  $p = 1, 2, 4, \dots, r$  (dla kolejnych potęg dwójki, aż do  $r$ ). Wykorzystywany jest następujący oczywisty fakt:

$$(*) \quad \text{numer}_{2p}[i] = \text{numer}_{2p}[j] \quad \text{wtedy i tylko wtedy, gdy} \\ (\text{numer}_p[i] = \text{numer}_p[j] \& \text{numer}_p[i+p] = \text{numer}_p[j+p]).$$

Pokażemy działanie tego algorytmu na przykładzie tekstu  $w = abbababba$ . Niech  $r = 4$ . Mamy 5 podslowów długości 4. Oto one:

$$x_1 = abab, x_2 = abba, x_3 = baba, x_4 = babb \text{ i } x_5 = bbab.$$

Naszą końcową tablicą powinna być zatem następująca tablica:

$$\text{numer}_4 = [2, 5, 3, 1, 4, 2]$$

Na początku obliczamy tablicę  $\text{numer}_1$ . Obliczenie takie jest bardzo proste, gdyż  $\text{numer}_1[i]$  jest numerem symbolu  $w[i]$ . W naszym wypadku  $\text{numer}_1 = [1, 2, 2, 1, 2, 1, 2, 2, 1]$ . Pokażemy teraz, jak obliczać tablicę  $\text{numer}_{2p}$ , mając daną tablicę  $\text{numer}_p$  i korzystając z faktu (\*). Aby obliczyć tablicę  $\text{numer}_4$ , tworzymy ciąg trójkę  $(\text{numer}_2[i], \text{numer}_2[i+2], i)$ , a następnie sortujemy ten ciąg leksykograficznie w czasie liniowym. W naszym wypadku  $\text{numer}_2 = [1, 3, 2, 1, 2, 1, 3, 2]$ , a ciągiem trójkę jest ciąg:

$$(1, 2, 1), (3, 1, 2), (2, 2, 3), (1, 1, 4), (2, 3, 5), (1, 2, 6)$$

Po posortowaniu otrzymujemy ciąg:

$$(1, 1, 4) \mid (1, 2, 1), (1, 2, 6) \mid (2, 2, 3) \mid (2, 3, 5) \mid (3, 1, 2)$$

Dzielimy ten ciąg na grupy trójkę mających te same dwie pierwsze współrzędne. Jeżeli trójka  $(k_1, k_2, i)$  jest w  $k$ -tej grupie, to  $\text{numer}_4[i] = k$ , a zatem  $\text{numer}_4[4] = 1$ ,  $\text{numer}_4[1] = 2$  itd.

Schemat algorytmu KMR jest następujący:

```
{Algorytm KMR (Karpa-Millera-Rosenberga); obliczona zostaje tablica
numer}
begin
  for i := 1 to |w| do numer_1[i] := numer symbolu w[i];
  p := 1;
```

```

repeat
    oblicz tablicę numer2p, korzystając z faktu (*);
    {korzystamy z sortowania kubełkowego; koszt = O(|w|)}
    p := 2*p;
until p = r;
for i := 1 to |w| do numerr[i] := numerr[i]
end {algorytm KMR};

```

Algorytm KMR ma złożoność  $O(n \log n)$ , gdzie  $|w| = n$ . Chociaż nie jest to złożoność liniowa, algorytm KMR jest bardzo interesujący.

Jeśli  $r$  nie jest potęgą dwójki, to można wykonać algorytm dla rozmiaru  $r'$  będącego największą potęgą dwójki nie przekraczającą  $r$ . Potem należy skorzystać z faktu podobnego do faktu (\*), a mianowicie:

$$\begin{aligned} \text{numer}_r[i] &= \text{numer}_r[j] \text{ wtedy i tylko wtedy, gdy} \\ (\text{numer}_r[i] &= \text{numer}_r[j] \& \text{numer}_r[i+r-r'] = \text{numer}_r[j+r-r']). \end{aligned}$$

Algorytm KMR można bez istotnych zmian stosować dla tekstów dwuwymiarowych. Rozważmy podtablice rozmiaru  $p \times p$ , gdzie  $p$  jest potęgą dwójki. Wartość elementu  $\text{numer}_p[i, j]$  jest teraz numerem podtablicy, której lewy górny róg znajduje się na pozycji  $(i, j)$  w początkowym tekście dwuwymiarowym. Można sformułować fakt analogiczny do (\*). Trzeba będzie rozważyć numery podtablic zaczynających się na pozycjach  $(i, j)$ ,  $(i+p, j)$ ,  $(i, j+p)$  oraz  $(i+p, j+p)$  przy przejściu od rozmiaru  $p \times p$  do rozmiaru  $2p \times 2p$  (pozostawiamy to jako zadanie).

### 5.1.6.

#### Algorytm KR (Karpa-Rabina)

W algorytmie tym stosujemy funkcję mieszającą  $h$ , dzięki której można obliczyć pewną wartość (kod) podslowa  $y_i = y[i..i+m-1]$  tekstu wejściowego. Jeśli  $h(y_i) = h(x)$ , to z bardzo dużym prawdopodobieństwem zachodzi równość  $x = y_i$ , a więc wzorzec  $x$  występuje w  $y$  na  $i$ -tej pozycji. Efektywność algorytmu wynika z możliwości szybkiego obliczenia wartości  $h(y_{i+1})$ , jeżeli wartość  $h(y_i)$  jest znana. Działanie algorytmu polega na obliczaniu tych wartości kolejno dla  $i = 1, 2, \dots, n-m+1$ .

Załóżmy (dla uproszczenia), że nasz alfabet składa się z cyfr 0, 1, ...,  $r-1$ . Niech  $q$  będzie pewną, bardzo dużą liczbą pierwszą (ale nie za dużą, na przykład największą liczbą pierwszą taką, że  $q(r+1)$  nie powoduje nadmiaru). Jeśli  $z$  jest tekstem o długości  $m$ , to definiujemy

$$h(z) = (z[1]r^{m-1} + z[2]r^{m-2} + \dots + z[m]) \bmod q$$

Inaczej mówiąc, wartością funkcji mieszającej  $h$  jest wartość tekstu traktowanego jako liczba zapisana w systemie o podstawie  $r \bmod q$  (aby nie było nadmiaru). Prawdopodobieństwo, że wystąpi kolizja (wartości funkcji  $h$  będą takie same dla dwóch różnych argumentów) jest bardzo małe, gdyż wynosi  $1/q$ , a  $q$  jest bardzo duże.

```

{Algorytm KR (Karpa-Rabina)}
begin
    h1 := h(x);
    dM := rm-1; h2 := h(y[1..m]); {koszt O(m)}
    i := 1;
    while i ≤ n - m + 1 do
        begin
            if h2 = h1 then
                if x = y[i..i+m-1] then write(i) {koszt = O(m)};
                {oblicz nowe h2 = h(y[i+1..i+m]) wiedząc, że poprzednio
                h2 = h(y[i..i+m-1]) = (y[i]rm-1 + y[i+1]rm-2 + ... + y[i+m-1])
                mod q}
                h2 := ((h2 - y[i] * dM) * r + y[i+m]) mod q;
            i := i + 1;
        end
    end {algorytm KR};

```

Podobnie jak w algorytmie KMR, tak i tutaj uogólnienie na przypadek dwuwymiarowy (dla problemu WW) jest bardzo łatwe. Algorytm KR jest algorytmem praktycznym. W oryginalnej jego wersji liczba pierwsza  $q$  jest wybierana losowo.

### 5.1.7.

#### Algorytm BM (Boyer-Moore'a)

Algorytm ten, podobnie jak algorytm KMP, powstaje przez pewne ulepszenie algorytmu „naiwnego”. Ulepszenie to polega na pełniejszym wykorzystaniu gromadzonej informacji w algorytmie. W algorytmie KMP informacją tą była wartość  $j$ . Podobnie będzie i tutaj – z tą różnicą, że zacznijemy od nieco innej wersji algorytmu „naiwnego”.

```

{Algorytm N' (wersja algorytmu N); zgodność wzorca jest sprawdzana
od końca}
begin
    i := 1;
    while i ≤ n - m + 1 do
        begin
            j := m; while x[j] = y[i+j-1] do j := j - 1;
            if j = 0 then write(i); i := i + 1; {przesunięcie = 1}
        end
    end {algorytm N'} ;

```

W algorytmie tym nie jest wykorzystywana informacja związana ze zmienną  $j$ . Przed zwiększeniem wartości  $i$  poprzednia wartość  $j$  jest tracona, ale wiemy, że są wtedy spełnione warunki:

- (a)  $x[j+1..m] = y[i+j..i+m-1]$ ; {do tekstu pasuje końcowa część wzorca, począwszy od pozycji  $j+1$  we wzorcu}
- (b)  $x[j] \neq y[i+j-1]$ ;

Oznaczmy przez  $s$  wartość przesunięcia początkowej pozycji przyłożenia wzorca. Mamy obecnie  $s = 1$ . Korzystając jednak z informacji wyrażonej przez warunki (a) i (b), możemy czasami zwiększyć wartość przesunięcia  $s$ . Zastanówmy się, jakie warunki wystarczy nałożyć na wartość  $s$ , żeby zagwarantować to, iż nie zostanie pominięty żaden wzorzec zaczynający się w jednej z „przeskoczonych” pozycji. Najpierw jednak zanalizujmy, w jaki sposób warunki (a) i (b) wpływają na wystąpienie wzorca na pozycji  $i+s$ . Jeśli wzorzec zaczyna się od pozycji  $i+s$ , to:

$war1(s, j)$ : przesunięty wzorzec jest zgodny z jego częścią „pasującą” ostatnio do tekstu  $y$ , a bardziej formalnie: dla każdego  $j < k \leq m$  zachodzi  $s \geq k$  lub  $x[k-s] = y[k]$ ;

$war2(s, j)$ :  $s \geq j$  lub  $x[j-s] \neq y[j]$  (warunek ten wynika z niezgodności ostatnio czytanych symboli tekstu i wzorca).

Niech

$$\begin{aligned} d1(j) &= \min \{s \geq 1 : \text{zachodzi } war1(s, j)\} \\ d2(j) &= \min \{s \geq 1 : \text{zachodzi } war1(s, j) \text{ i } war2(s, j)\} \end{aligned}$$

Rozważmy następujący przykład:  $x = cababababa$ . Mamy  $d1(9) = 2$ ,  $d2(9) = 8$ , a więc  $d2$  daje większe przesunięcia.

Jeśli w algorytmie N' przyjmiemy wartość przesunięcia  $s = d2(j)$ , to otrzymamy algorytm BM.

{Algorytm BM (wersja algorytmu N') ; zgodność wzorca jest sprawdzana od końca}  
 begin  
 $i := 1$ ;  
 while  $i \leq n - m + 1$  do  
 begin  
 $j := m$ ; while  $x[j] = y[i+j-1]$  do  $j := j-1$ ;  
 if  $j = 0$  then write( $i$ );  $i := i + d2(j)$ ; {przesunięcie =  $d2(j)$ }  
 end  
 end {algorytm BM};

Okazuje się, że w algorytmie BM jest wykonywanych czasami znacznie mniej niż  $n$  porównań, a jednocześnie maksymalna liczba porównań jest liniowa (dowód wykracza poza ramy tej książki).

Niech  $x = cababababa$  i  $y = aaaaaaaaaababababa$ . W tym wypadku liczba porównań symboli w algorytmie BM jest równa 12. Gdybyśmy jako przesunięcia użyli  $d1(j)$  a nie  $d2(j)$ , to wykonalibyśmy 30 porównań. Okazuje się, że w wypadku użycia  $d1(j)$  liczba porównań wynosi  $\Omega(n^2)$ . Można to zobaczyć dla  $x = ca(ba)^k$  i  $y = a^{2k+2}(ba)^k$ .

Algorytm BM wymaga wcześniejszego stabilicowania wartości  $d2(j)$ . Koszt takiego obliczenia jest liniowy. Można w tym celu wykorzystać algorytm obliczania tablicy  $P$  (potrzebnej w algorytmie KMP) dla tekstu  $x^R$  (gdzie  $R$  jest operacją odwracania tekstu). Jest wtedy widoczny pewien związek między algorytmami KMP i BM. Opracowanie takiego algorytmu pozostawiamy Ci jako zadanie.

Przedstawimy teraz algorytm koncepcyjnie podobny do algorytmu BM. Nazwiemy go algorytmem  $BM'$ . Wzorzec jest przykładowy do tekstu i „dopasowanie” wzorca jest sprawdzane od końca. Przyjmijmy dla uproszczenia, że alfabet jest dwuliterowy. Niech  $r = 2 \lceil \log m \rceil$ .

{Algorytm  $BM'$ }  
 begin  
 $i := m$ ;  
 while  $i \leq n$  do  
 begin  
 $i := m$ ;  
 if  $y[i-r+1..i]$  jest pod słowem wzorca  $x$  then  
 oblicz wszystkie wystąpienia wzorca na pozycjach  
 $[i-r+1..i-r+1]$ , korzystając z algorytmu KMP {koszt =  $O(m)$ }  
 else {wzorzec nie rozpoczyna się od żadnej z pozycji  
 $[i-r+1..i-r+1]$ };  
 $i := i + m - r + 1$ ;  
 end  
 end {algorytm  $BM'$ };

Załóżmy, że koszt sprawdzania, czy  $y[i-r+1..i]$  jest pod słowem wzorca  $x$ , jest  $O(\log m)$  (jeśli korzysta się z uprzednio policzonej struktury danych  $T(x)$  lub  $G(x)$ , o których będzie mowa później).

Przypuśćmy, że  $y$  jest tekstem losowym. Wzorzec ma co najwyżej  $m$  różnych podłów długości  $r$ , a prawdopodobieństwo, że dane słowo długości  $r$  (mamy  $2^r \geq m^2$  takich słów) jest pod słowem wzorca, jest co najwyżej  $1/m$ . Oczekiwany koszt obliczeń dla ustalonego  $i$  wynosi zatem  $O(n/m + r)$ . Ponieważ mamy  $O(n/m)$  takich wartości  $i$ , oczekiwany koszt algorytmu  $BM'$  wynosi  $O((n \log m)/m)$ . Dla  $m = n^{1/2}$  na przykład otrzymujemy oczekiwany koszt algorytmu  $BM'$  rzędu  $n^{1/2} \log n$  (nawet wtedy, kiedy uwzględnimy w koszcie tworzenie  $T(x)$ ). Pominiecie czasu wstępnie obliczania  $T(x)$  ma sens przy szukaniu tego samego wzorca wielokrotnie (na przykład przy szukaniu słów kluczowych w tekście programu).

### 5.1.8. Algorytm FP (Fishera-Patersona)

W algorytmie KMR dochodziło do pewnego sprowadzenia problemu WW do problemu sortowania. W algorytmie FP (Fishera-Patersona) dla problemu WW' (wyszukiwania wzorca z symbolami nieznaczącymi) jest z kolei wykorzystywana redukcja problemu tekstu do problemu arytmetycznego. Pokażemy, że złożoność problemu WW' jest (asymptotycznie) nie większa niż złożoność mnożenia dwóch liczb całkowitych. Oznaczmy przez  $\phi$  symbol nieznaczący, który ma tę silną własność, że pasuje do każdego innego symbolu. Przez  $\equiv$  oznaczmy relację „pasowania”. Dla dwóch symboli  $z$  i  $s$  mamy  $s \equiv z$ , gdy  $s = z$  lub  $s = \phi$ , lub  $z = \phi$ . Relację  $\equiv$  rozszerzamy na teksty. Dwa teksty  $u$  i  $w$  (o tej samej długości) są w relacji  $\equiv$  (piszemy  $u \equiv w$ ), gdy dla każdej pozycji  $i$  mamy  $u[i] \equiv w[i]$ .

Relacja  $\equiv$  ma jednak nieprzyjemną własność: nie jest przechodnia. Z równości  $a \equiv b$ ,  $b \equiv c$  nie możemy wcale wnioskować, że  $a \equiv c$ . W tym momencie nietrudno uwierzyć, że poprzednio skonstruowane algorytmy dla problemu WW nie działają dla problemu WW' (gdyby sprawdzanie równości symboli zastąpić w nich sprawdzaniem relacji  $\equiv$  między symbolami). Co więcej, dla problemu WW wystarczy zawsze  $O(n)$  porównań ( $=$ ). Jeśli natomiast jedyną informację o słowach  $x$  i  $y$  otrzymujemy za pomocą porównania ( $\equiv$ ), to możemy pokazać, że potrzeba  $\Omega(n^2)$  takich porównań. Weźmy mianowicie  $x = \phi^n$  i  $y = \phi^{2n}$ . Przypuśćmy, że za pomocą pewnego algorytmu można sprawdzić, czy  $x \equiv y[i..i+n-1]$  dla  $i = 1..n+1$ , korzystając jedynie z odpowiedzi na pytania typu: „Czy  $x[p] \equiv y[q]?$ ”. Oczywiście zostaną wypisane wystąpienia na pozycjach 1, 2, ...,  $n+1$ . Przypuśćmy teraz, że w algorytmie nie ma pytania: „Czy  $x[p] \equiv y[q]?$ ” dla pewnych  $1 \leq p \leq n$  i  $n+1 \leq q \leq n+p$ . Zmieśmy symbole  $x[p]$  i  $y[q]$  w następujący sposób:  $x[p] := a$ ,  $y[q] := b$  (gdzie  $a$  i  $b$  są różnymi symbolami znaczącymi). Ponieważ zestaw odpowiedzi na zadane pytania się nie zmienił, algorytm daje ten sam wynik: wystąpienia na pozycjach 1, 2, ...,  $n+1$ . Nie jest to jednak poprawne; na pozycji  $q = p+1$  wzorzec  $x$  nie występuje (nie zachodzi  $x \equiv y[q-p+1..q-p+n]$ ). Jeśli jedynymi informacjami wejściowymi są odpowiedzi na porównania, to potrzeba  $\Omega(n^2)$  tych porównań.

Pokażemy, że korzystając z pełnej informacji o danych wejściowych oraz kodując teksty  $x$  i  $y$  jako liczby binarne, możemy osiągnąć złożoność rzędu dużo mniejszego niż  $n^2$ , a mianowicie tego co koszt mnożenia liczb. Nie będzie to jednak koszt liniowy. Problem istnienia algorytmu o złożoności liniowej dla problemu WW' jest otwarty. Wszystko wskazuje, że takiego algorytmu w ogóle nie ma.

Zacznijmy od algorytmu, w którym dwa teksty są mnożone tak jak robi się to w szkole, czyli metodą „w słupkach”. W metodzie tej podpisujemy jedną liczbę pod drugą, a następnie wymażamy cyfry jednej liczby przez cyfry drugiej, stosując operację  $*$ . Otrzymujemy kilka „pięter” poziomo zapisanych liczb. Od strony lewej do prawej sumujemy kolejno cyfry w kolumnach, stosując operację  $+$ . Można powiedzieć, że pełna operacja to para  $\langle \ast, + \rangle$ , ponieważ operacje  $\ast$  i  $+$  określają procedurę mnożenia „w słupkach”.

### 5.1. Problem wyszukiwania wzorca

Podobnie możemy zdefiniować operację  $\langle \equiv, \wedge \rangle$ . Jeśli mnożymy teksty  $u$  i  $v$ , to w wyniku otrzymujemy tekst  $w = u \equiv, \wedge v$ , taki że

$$w[k] = \bigwedge_{i+j=k+1} (u[i] \equiv v[j])$$

gdzie  $\wedge$  jest operacją koniunkcji.

Operację  $\langle \equiv, \wedge \rangle$  nazywamy **mnożeniem tekstowym**. Przypuśćmy, że mamy znaleźć wystąpienia wzorca  $x = baa$  w tekście  $y = baaba$ . Mnożymy  $x^R = aab$  przez  $y$ :

$$\begin{array}{r} b \ a \ a \ b \ a \\ \equiv \quad \quad \quad a \ a \ b \\ \hline 1 \ 0 \ 0 \ 1 \ 0 \\ 0 \ 1 \ 1 \ 0 \ 1 \\ \hline 0 \ 1 \ 1 \ 0 \ 1 \\ \hline 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \end{array} \quad \left. \begin{array}{c} \\ \\ \\ \end{array} \right\} \wedge$$

Łatwo sprawdzić, że wzorzec  $x$  występuje w tekście  $y$  od pozycji  $i$ , gdzie  $1 \leq i \leq n - m + 1$ , wtedy i tylko wtedy, gdy  $z[i+m-1] = 1$ , gdzie  $z = x^R \equiv, \wedge y$ . W naszym przykładzie  $x$  występuje w y tylko od pozycji 1. Problem WW' sprowadza się zatem do obliczania iloczynu ( $\equiv, \wedge$ ).

Podobnie definiujemy mnożenie logiczne  $\langle \wedge, \vee \rangle$  dwóch ciągów bitów. Pokażemy teraz, że obliczanie iloczynu tekstu do logicznego można sprowadzić do obliczania iloczynu logicznego.

Niech  $\text{logwektor}_i(v)$  będzie wektorem logicznym, którego  $i$ -tą składową jest true, gdy  $v[i] = a$ , gdzie  $v$  jest danym tekstem, a  $a$  symbolem alfabetu. Przez  $X_{a..b}(u, v)$  oznaczmy iloczyn logiczny:

$$X_{a..b}(u, v) = \text{logwektor}_a(u) \equiv, \wedge \text{logwektor}_b(v)$$

Wartość iloczynu tekstu do logicznego  $v \equiv, \wedge u$  jest równa negacji alternatywy logicznej wszystkich wektorów  $X_{a..b}(u, v)$ , gdzie  $(a, b)$  są wszystkimi parami wzajemnie różnych ( $a \neq b$ ) symboli znaczących (różnych od  $\phi$ ).

Problem WW' ma zatem ten sam rzad złożoności co obliczanie iloczynu logicznego wektorów. Pokażemy, w jaki sposób można łatwo sprowadzić obliczanie iloczynu logicznego wektorów do obliczania iloczynu arytmetycznego dwóch liczb. Niech  $n$  będzie długością mnożonych wektorów  $u$  i  $v$ . Każdy bit  $b$  tych wektorów zastąpmy ciągiem  $\lceil \log n \rceil + 1$  bitów 00..0b. Otrzymamy ciągi  $u'$  i  $v'$  o długości  $O(n \log n)$ ; potraktujmy je jako liczby (w zapisie binarnym). Mnożymy arytmetycznie  $u'$  i  $v'$ . Weźmy  $i$ -tą grupę bitów o długości  $\lceil \log n \rceil + 1$ . Zauważmy, że  $i$ -ty bit iloczynu logicznego wektorów jest niezerowy wtedy i tylko wtedy, gdy wartość takiej  $i$ -tej grupy bitów jest niezerowa (nie składa się z samych zer).

Ponieważ istnieją algorytmy mnożenia liczb binarnych w czasie  $O(n^r)$ , gdzie  $r < 2$  [AHU], problem  $WW'$  można rozwiązać w czasie  $O((n \log n)^r)$ , a zatem asymptotyczny koszt problemu  $WW'$  jest rzędu mniejszego niż  $n^2$ . Nie jest znany algorytm o koszcie liniowym (z czego nie wynika wcale nieistnienie takiego szybkiego algorytmu).

### 5.2.

## Drzewa sufiksowe i grafy podłów

W tym podrozdziale przedstawimy dwie interesujące struktury danych dla tekstów, a mianowicie drzewo sufiksowe  $T(x)$  i graf podłów  $G(x)$ . Struktury te służą do reprezentacji zbioru wszystkich podłów danego tekstu  $x$ . Rozmiar reprezentacji będzie liniowy, chociaż liczba wszystkich podłów może być nieliniowa. Dla liniowości czasowej procesu tworzenia tych struktur danych istotne jest, żeby rozmiar alfabetu był ograniczony przez pewną stałą. Zakładamy w tym podrozdziale, że tak jest. Jeśli rozmiar alfabetu jest „duży” i wynosi  $k$ , to należy wymożyc złożoność każdego algorytmu przez  $\log k$ .

Drzewa sufiksowe i grafy podłów mogą być użyte do tworzenia algorytmów liniowych dla problemu  $WW$ . Można je ponadto wykorzystać do rozwiązywania innych problemów.

Zbiór wszystkich podłów  $Sub(x)$  danego tekstu  $x$  o długości  $n$  może się składać z liczby rzędu  $n^2$  słów. Okazuje się jednak, że jest możliwa zwarta reprezentacja takiego zbioru. Naszym przykładowym tekstem w tym podrozdziale będzie tekst  $x = aaccacd$ .

### 5.2.1.

#### Niezwarsta reprezentacja drzewa sufiksowego

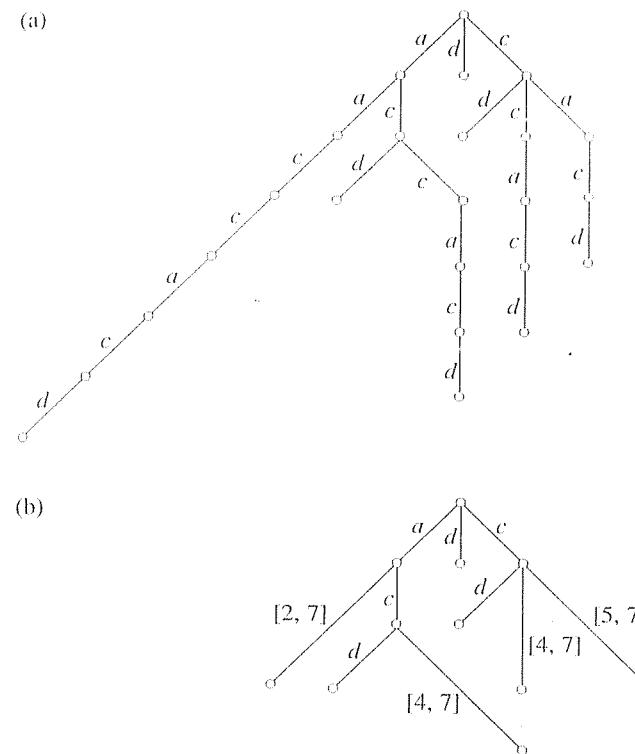
Będziemy rozważać acykliczne grafy zorientowane, których krawędzie są etykietowane symbolami alfabetu lub podłownymi tekstu  $x$ . Etykiety te nazywamy wartościami krawędzi. Grafy te będą mieć wierzchołek zwany korzeniem, z którego prowadzi ścieżka do każdego innego węzła. Przez  $wart(p)$  oznaczamy dla każdej ścieżki  $p$  tekst będący złożeniem etykiet (wartości) krawędzi na ścieżce  $p$  ( $wart(p)$  jest wartością ścieżki  $p$ ).

Powiemy, że graf  $G$  reprezentuje  $Sub(x)$ , jeżeli  $Sub(x) = \{wart(p) : p \text{ jest ścieżką o początku w korzeniu grafu } G\}$  i różne ścieżki (zaczynające się w korzeniu) mają różne wartości.

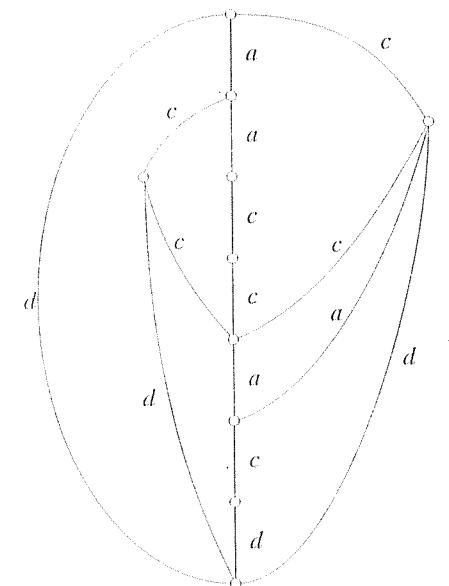
Przez  $G(x)$  oznaczamy graf reprezentujący  $Sub(x)$ . Zajmiemy się teraz tworzeniem go.

Niech  $TI(x)$  będzie drzewem reprezentującym  $Sub(x)$ , inaczej mówiąc grafem, który reprezentuje  $Sub(x)$  i w którym dwie różne ścieżki nie prowadzą od korzenia do tego samego wierzchołka.

Grafy  $TI(x)$  i  $G(x)$  dla przykładowego tekstu są przedstawione na rysunkach 5.1 i 5.2. Zauważmy, że  $G(x)$  ma tylko 10 węzłów, podczas gdy  $TI(x)$  ma ich 24.



Rys. 5.1. (a) Drzewo  $TI$ ;  
(b) drzewo  $T$



Rys. 5.2. Graf  $G$  (krawędzie są zorientowane z góry na dół)

Drzewo  $TI(x)$  ma takie oto własności. Liczba podłów  $x$  jest równa liczbie węzłów drzewa. Dla każdego węzła  $v$  przyjmijmy, że  $wart(v) = wart(p)$ , gdzie  $p$  jest ścieżką od korzenia do  $v$ . Wartościami liści tego drzewa są zatem sufiksy tekstu  $x$ . Ponieważ mamy  $n$  sufiksów, liczba liści wynosi co najwyżej  $n$ . Drzewo  $TI(x)$  nazywamy **niezwartym drzewem sufiksowym**. Drzewo sufiksowe  $T(x)$  jest zwaną (skróconą) reprezentacją  $TI(x)$ . Łańcuchem nazywamy maksymalną ścieżkę w  $TI(x)$  (być może nie od korzenia), której wszystkie węzły, z wyjątkiem końców, mają stopień równy 1 (mają jeden następnik). Zauważmy, że liczba krawędzi  $TI(x)$  wynosi 23, natomiast liczba łańcuchów jedynie 10. W ogólności liczba ta jest zawsze  $O(n)$ .

Drzewo  $T(x)$  jest drzewem, które powstaje z  $TI(x)$  przez zastąpienie każdego łańcucha  $p$  przez pojedynczą krawędź, której wartością (etykietą) jest para  $[i, j]$ , gdzie  $x[i..j] = wart(p)$ . Stosujemy tu zatem zwykły zapis podłów; gdybyśmy bowiem wpisywali cały tekst  $wart(p)$ , to rozmiar reprezentacji  $T(x)$  byłby dalej rzędu  $n^2$ . Etykietę  $[i, j]$  będziemy utożsamiać z podłówem  $x[i..j]$ .

Drzewo  $T(x)$  (dla naszego przykładowego tekstu  $x$ ) jest przedstawione na rysunku 5.1. Drzewo to ma 11 węzłów.

## 5.2.2.

### Tworzenie drzewa sufiksowego

Zajmiemy się teraz złożonością procesu tworzenia drzewa sufiksowego  $T(x)$ . Dla uproszczenia przyjmijmy, że dany tekst kończy się zawsze specjalnym symbolem, występującym jedynie na końcu (w naszym przykładzie symbolem  $d$ ). Niech  $suf(i)$  będzie  $i$ -tym sufiksem  $x$ , czyli niech  $suf(i) = x[i..n]$  dla  $i = 1, \dots, n$ . Bezpośrednim algorytmem jest skonstruowanie najpierw  $TI(x)$ , a następnie przekształcenie  $TI(x)$  na  $T(x)$ . Można to łatwo wykonać w czasie  $O(n^2)$ . Wąskim gardłem tej metody jest rozmiar  $TI(x)$ ; dla  $x = a^n b^n a^n b^n d$  bowiem rozmiar ten jest rzędu  $n^2$ . Jeśli nawet przyjmiemy, że rozmiar  $TI(x)$  jest liniowy (dla szczególnej klasy tekstów  $x$ ), to projekt algorytmu tworzenia  $TI(x)$  w czasie proporcjonalnym do rozmiaru  $TI(x)$  jest nietywialny.

Będziemy budować  $T(x)$ , nie korzystając z  $TI(x)$ , ale cały czas  $T(x)$  będziemy traktować jako reprezentację  $TI(x)$ . Algorytm polega na tworzeniu drzew sufiksowych  $T(suf(i))$  w kolejności  $i = n, n-1, \dots, 1$ . Zakładamy, że w danym momencie mamy już drzewo  $T = T(suf(i+1))$ .

Przez  $\cdot$  oznaczamy operację konkatenacji (złożenia) tekstów. Na ogół teksty pisze się obok siebie, ale czasami dzięki użyciu  $\cdot$  zapis tej operacji staje się jaśniejszy.

Niech  $LINK$  i  $TEST$  będą tablicami pomocniczymi, takimi że:

- $LINK[a, u] = v$ , gdy  $wart(v) = a \cdot wart(u)$  dla węzłów  $u$  i  $v$  drzewa  $T$  i każdego symbolu  $a$ ; jeśli takiego węzła  $v$  nie ma, to przyjmujemy  $LINK[a, u] = \text{nil}$ ;

## 5.2. Drzewa sufiksowe i grafy podłów

- $TEST[a, u] = \text{true}$ , gdy słowo  $a \cdot wart(u)$  jest prefiksem wartości pewnego węzła drzewa  $T$ ; w przeciwnym razie  $TEST[a, u] = \text{false}$ .

Wprowadzimy też pewne funkcje pomocnicze, określone częściowo (gdy istnieje węzeł spełniający zadane warunki) dla symboli z alfabetu i węzłów drzewa  $T$ . Oto one:

- $firsttest(a, w) = \text{pierwszy węzeł } u \text{ na drodze od węzła } w \text{ do korzenia, dla którego } TEST[a, u] = \text{true}$ ;
- $firstlink(a, w) = \text{pierwszy węzeł } u \text{ na drodze od węzła } w \text{ do korzenia, dla którego } LINK[a, u] \neq \text{nil}$  ( $\text{nil}$ , jeśli nie ma takiego  $u$ );
- $nast(u, x', x) = \text{następny węzeł } u \text{ do którego prowadzi krawędź o wartości zaczynającej się na tę samą literę co wartość pierwszej krawędzi na ścieżce od } x' \text{ do } x$ ;
- $breakpoint(u, v, z) = w$ , gdzie  $w$  jest takim nowym węzłem, że  $wart(w) = z$ ; funkcja ta jest jedynie określona dla takich węzłów  $u, v$ , że  $v$  jest następcą (*synem*)  $u$ ;  $z$  jest właściwym prefiksem  $wart(v)$ , a  $wart(u)$  jest właściwym prefiksem  $z$ .

Efektem ubocznym działania funkcji  $breakpoint$  jest utworzenie odpowiednich krawędzi z  $u$  do  $w$  oraz z  $w$  do  $v$ , jak również nadanie tym krawędziom właściwych wartości (uzyskane drzewo ma reprezentować  $Sub(suf(i))$ ). Inaczej mówiąc, operacja  $breakpoint$  polega na podzieleniu krawędzi od  $u$  do  $v$  na dwie krawędzie przez utworzenie nowego węzła, który jest wartością tej operacji. Ścieżka prowadząca do tego nowego węzła odpowiada słowu  $z$ .

```

{Algorytm tworzenia drzewa  $T(x)$ }
begin
   $T := T(x[n])$ ;  $v := \text{jedyny liść } T$ ;
  inicjalizowanie tablic  $LINK$  i  $TEST$  dla  $T$ ; {koszt  $O(1)$ }
  for  $i := n-1$  down to 1 do
     $\{T = T(suf(i+1)), \text{wart}(v) = suf(i+1); \text{obliczamy } T(suf(i))\}$ 
    begin
       $a := x[i]$ ;
      if  $a$  nie występuje w  $suf(i+1)$  then
        begin
          dla każdego węzła  $w$  na ścieżce od  $v$  do korzenia
          wykonaj  $TEST[a, w] := \text{true}$ ;  $v1 := v$ ;
           $v := \text{nowy węzeł o wartości } suf(i)$ , będący następcą
          korzenia;  $LINK[a, v1] := v$ ;
        end
      else
        begin
           $x := firsttest(a, v)$ ;  $x' := firstlink(a, v)$ ;
          if  $x' = \text{nil}$  then  $head1 := \text{korzeń}$ 
          else  $head1 := LINK[a, x']$ ;
          if  $x = x'$  then  $head := head1$  else
        end
    end
  end
end

```

```

begin
  if  $x' = \text{nil}$  then  $head2 :=$  następnik korzenia, taki
  że etykieta prowadzącej do niego krawędzi
  zaczyna się symbolem  $a$ 
  else  $head2 := \text{next}(head1, x', x)$ ;
   $head := \text{breakpoint}(head1, head2, a * \text{wart}(x))$ ;
end;
{aktualizacja tablic  $LINK$  i  $TEST$ }
 $TEST[b, head] := TEST[b, head1]$  dla każdego symbolu  $b$ ;
 $TEST[a, v] := \text{true}$  dla każdego węzła  $v$  na ścieżce od  $v$  do  $x'$ ;
 $LINK[a, x] := head$ ;  $v1 := v$ ;
 $v :=$  nowy węzeł będący następcą  $head$ ;
 $\text{wart}(v) := \text{suf}(i)$ ;
 $LINK[a, v1] := v$ 
end;
end {algorytm};

```

Niech  $depth(v)$  oznacza głębokość węzła  $v$  w drzewie  $T$ . Można udowodnić, że  $depth(LINK[a, v]) \leq depth(v) + 1$ , co pozostawiamy Ci jako ćwiczenie. Wynika to stąd, że  $T = T(\text{suf}(i + 1))$ . Przyjmijmy, że rozmiar alfabetu jest niewielki (ograniczony przez stałą). Niech  $v(i)$  będzie równe wartości  $v$  po zakończeniu iteracji dla danego  $i$ . Łatwo zauważać, że koszt jednej iteracji jest proporcjonalny do różnicy  $|depth(v(i + 1)) - depth(v(i))|$  z dokładnością do pewnej stalej addytywnej. Ponieważ suma takich różnic jest rzędu  $n$ , koszt algorytmu jest liniowy. Koszt czasowy (jak również pamięciowy) zależy istotnie od rozmiaru alfabetu i jest rzędu  $sn$ , gdzie  $s$  jest rozmiarem alfabetu.

W następnym podrozdziale podamy kilka przykładów zastosowania struktury  $T(x)$ . Najpierw wymienimy dwa przykłady bezpośredniego zastosowania, a mianowicie do obliczania leksykograficznie maksymalnego sufiksu danego słowa i do obliczania najdłuższego powtarzającego się pod słowa.

Pierwszy z tych problemów można rozwiązać, idąc maksymalną leksykograficznie ścieżką w dół drzewa  $T(x)$ . Drugi problem sprowadza się do znalezienia w drzewie sufiksowym węzła  $v$  o najdłuższym  $\text{wart}(v)$  – węzła, który ma co najmniej dwa następcy.

Przypuśćmy, że mamy bardzo dużo wzorców  $x$ , a tekst  $y$  jest ustalony. Czy można tak przygotować strukturę danych dla tekstu  $y$ , żeby w czasie  $O(|x|)$  można było sprawdzić wystąpienie zadanego wzorca  $x$  w tekście.

Drzewo  $T(y)$  umożliwia rozwiązanie tego problemu. Można jednak sformułować bardziej skomplikowaną wersję problemu: znaleźć pierwsze wystąpienie  $x$  lub wszystkie wystąpienia (w drugim przypadku koszt ma być proporcjonalny do  $|x|$  plus liczba wystąpien).

Strukturę  $T(y)$  można też zastosować do obliczenia tak zwanego drzewa pozycyjnego. Niech dla każdej pozycji  $i$  słowo  $ident(i)$  będzie najkrótszym słowem, które zaczyna się

w tekście  $x$  na pozycji  $i$  i nie występuje na żadnej innej. Mówimy, że słowo takie identyfikuje pozycję  $i$ . Drzewo pozycyjne jest drzewem, którego krawędzie są etykietowane symbolami lub pod słowami, a zbiór wartości ścieżek od korzenia do liści jest równy zbiorowi identyfikatorów pozycji. Utworzenie drzewa pozycyjnego, przy założeniu, że drzewo sufiksowe jest dane, pozostawiamy Tobie, Drogi Czytelniku. Z grubsza biorąc, dla każdego liścia  $v$  wystarczy zastąpić etykietę  $x[i, j]$  krawędzi prowadzącej do  $v$  przez  $x[i]$ .

### 5.2.3.

## Tworzenie grafu pod słów

Przejdziemy teraz do utworzenia grafu  $G(x)$  w czasie  $O(|x|)$ . Najpierw oszacujemy rozmiar tego grafu i zbadamy jego strukturę (omijając szczegóły algorytmiczne).

Niech  $end\text{-pos}(z)$  oznacza zbiór pozycji, na których kończy się wystąpienie słowa  $z$  w  $x$ . Rozważać będziemy tylko słowa z należące do  $Sub(x)$ . Niech na przykład  $z = ac$  i niech  $x$  będzie naszym przykładowym tekstem  $aaccacd$ . Wtedy  $end\text{-pos}(z) = \{3, 6\}$ . Założymy, że dla słowa pustego zbiór końcowych pozycji jest równy zbiorowi wszystkich pozycji. Dla pod słów tekstu  $x$  mamy 10 różnych zbiorów końcowych pozycji:  $\{1, 2, 5\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{4\}$ ,  $\{5\}$ ,  $\{6\}$ ,  $\{7\}$ ,  $\{3, 6\}$ ,  $\{3, 4, 6\}$  oraz zbiór wszystkich pozycji. Ten ostatni zbiór odpowiada korzeniowi grafu acyklicznego  $G(x)$ . Zbiór  $V$  węzłów grafu  $G(x)$  można utożsamić ze zbiorem wszystkich zbiorów  $end\text{-pos}$  dla pod słów  $x$ . Jeżeli  $v = end\text{-pos}(z)$ ,  $w = end\text{-pos}(za)$ , to etykietą krawędzi od  $v$  do  $w$  jest symbol  $a$ . Podstawową właściwością grafów  $G(x)$  jest ich mały rozmiar. Graf pod słów ma bowiem co najwyżej  $2n - 1$  węzłów.

Wynika to stąd, że rodzina zbiorów  $end\text{-pos}$  ma strukturę drzewiastą. Jeśli weźmiemy dwa różne zbiory z tej rodziny, to albo są one rozłączne, albo jeden zawiera się w drugim. Możemy zatem zdefiniować drzewo podzbiorów odpowiadających węzłom z  $V$ . Podzbiór  $V1$  jest synem  $V2$ , gdy  $V1 \subset V2$ . Ponieważ liście odpowiadają podzbiorom wzajemnie rozłącznym, a takich podzbiorów może być co najwyżej  $n$ , mamy co najwyżej  $2n - 1$  wszystkich podzbiorów.

Pozostawiamy Ci udowodnienie (dowód nietrywialny), że liczba krawędzi grafu  $G(x)$  jest również liniowa względem  $|x|$ , bez względu na rozmiar alfabetu.

Można obliczać  $G(x)$  wprost – korzystając z odpowiedniości między zbiorami  $end\text{-pos}$  i węzłami  $G(x)$ . Choć rozmiar grafu jest liniowy, algorytm taki może mieć jednak koszt rzędu  $n^2$  ( $|x| = n$ ), gdyż samo wypisanie wszystkich zbiorów  $end\text{-pos}$  może być tego rzędu (zbiorów te mogą być za duże). Nie można więc reprezentować węzłów grafu przez zbiory pozycji końcowych. Bardziej oszczędną metodą reprezentacji jest utożsamianie węzła z najdłuższym słowem  $x[i, j]$ , który jest wartością ścieżki od korzenia do danego węzła. Zamiast  $x[i, j]$  możemy pamiętać jedynie parę  $[i, j]$ .

Dany węzeł  $v = x[i..j]$  odpowiada zbiorowi pozycji  $P(v) = \text{end-pos}(x[i..j])$ . Ze wszystkich słów o tym samym zbiorze pozycji końcowych  $P(v)$  słowo  $x[i..j]$  jest najdłuższe. Zbiór takich słów (mających ten sam zbiór  $\text{end-pos}$ ) jest postaci:  $\{x[i..j], x[i+1..j], \dots, x[k..j]\}$ .

Niech  $\text{suf}(v) = x[k+1..j]$ , jeżeli  $k < j$ . W przeciwnym wypadku  $\text{suf}(v)$  będzie korzeniem grafu  $G(x)$ , a więc słowem pustym. Zauważmy, że posługujemy się tutaj trzema reprezentacjami węzłów grafu podków: zbiorami pozycji  $P(v)$ , słowami  $x[i..j]$  i numerami węzłów  $v$ . Pierwsza reprezentacja jest teoretyczna; służy jedynie do wyjaśnienia działania algorytmu tworzenia grafu  $G(x)$ .

Rozważmy dla przykładu węzeł  $v = aac = x[1..3]$  w grafie  $G(aaccacd)$ . Mamy wtedy  $P(v) = \{3\}$ ,  $\text{suf}(v) = v1 = ac = x[2..3]$ ;  $P(v1) = \{3, 6\}$ ,  $\text{suf}(v1) = v2 = c = x[3..3]$ ;  $P(v2) = \{3, 4, 6\}$  i  $\text{suf}(v2)$  jest korzeniem.

Dla węzła  $v = x[i..j]$  oznaczmy przez  $\text{length}(v)$  liczbę  $j - i + 1$ . Inaczej mówiąc,  $\text{length}(v)$  jest długością najdłuższej ścieżki od korzenia do węzła  $v$ .

Algorytm, który teraz podamy, będzie typu on-line. Oznacza to, że tworzymy kolejno grafy  $G(a_1)$ ,  $G(a_1a_2)$ ,  $G(a_1a_2a_3)$  ..., wczytując kolejne symbole tekstu  $a_1a_2\dots a_n$ . Na koniec tego podrozdziału przedstawimy nieformalny schemat prostego algorytmu działającego offline.

W algorytmie tworzenia grafu podków będziemy dla każdego węzła obliczać wartość  $\text{suf}(v)$  i umieszczać ją w tablicy  $SUF[v]$ . Przez  $\text{nast}(v, a)$  oznaczmy taki bezpośredni następnik  $w$  węzła  $v$ , że krawędź od  $v$  do  $w$  jest etykietowana symbolem  $a$ . Jeżeli takiego następnika nie ma, to przyjmujemy  $\text{nast}(v, a) = \text{nil}$ .

```
{Algorytm tworzenia grafu podków}
begin
  a := x[1]; oblicz G(a); root := korzeń grafu G(a);
  last := liść grafu G(a); {węzeł bez następców}
  for i := 2 to |x| do
    begin
      a := x[i]; utwórz nowy węzeł last1 nie mający następców;
      p := SUF[last];
      while (p ≠ root) and (nast(p, a) = nil) do
        begin nast(p, a) := last1; p := SUF[p] end;
      w := nast(p, a);
      if w = nil then {p = root}
        begin nast(root, a) := last1; SUF[last1] := root end
      else
        if length(p) + 1 = length(w) then SUF[last1] := w
      else
    end
```

```
begin
  utwórz kopię r węzła w; x ma te same następcy co w;
  length(x) := length(p) + 1; SUF[w] := r; SUF[last1] := x;
  niech wl będzie poprzednikiem węzła w, takim że
  length(wl) = length(w) - 1;
  skieruj krawędzie prowadzące do w (oprócz tej
  prowadzącej z wl) do nowego węzła r; v1 := SUF[p];
  while (length(v1) + 1 ≠ length(w)) and (nast(v1, a) = w) do
    begin nast(v1, a) := r; v1 := SUF[v1] end;
  end;
  length(last1) := length(last) + 1; nast(last, a) := last1;
  last := last1
end
end {algorytm konstrukcji G(x)};
```

Pewnych dodatkowych wyjaśnień wymaga operacja tworzenia nowych węzłów, ponieważ dla każdego węzła chcielibyśmy znać odpowiadający mu tekst  $x[i..j]$ . Oczywiście można pominąć taką informację i identyfikować węzły przez ich numery, ale identyfikator tekstowy może się przydać w różnych zastosowaniach. Jest on również pomocny do zrozumienia działania algorytmu. Jeżeli tworzymy nowy węzeł  $last1$ , to odpowiada mu tekst  $x[1..i]$ . Natomiast węzowi  $r$  odpowiada  $x[i - l + 1..i]$ , gdzie  $l$  jest wartością  $\text{length}(r)$ , obliczoną przy tworzeniu węzła  $r$ . Kluczową rolę w algorytmie odgrywa tabela  $SUF$ . Po zakończeniu danego przebiegu instrukcji „dla” (for)  $SUF[last]$  jest węzłem odpowiadającym najdłuższemu sufiksowi  $x[1..i]$ , który występuje wcześniej w słowie  $x[1..i]$  (nie tylko na końcu). Zastanówmy się, w jaki sposób liczymy wartość  $SUF[last1]$ . Rozpoczynamy kolejny przebieg instrukcji iteracyjnej „dla” (for) dla danego  $i$ . Obliczamy węzeł  $p$ . Przypuśćmy, że  $nast(p, a) \neq \text{nil}$ . Wtedy  $p$  odpowiada najdłuższemu sufiksowi  $x[s..i-1]$  słowa  $x[1..i-1]$ , dla którego słowo  $x[s..i] = x[s..i-1] \cdot a$  występuje w  $x[1..i]$  nie tylko na końcu. Nowy węzeł  $r = \text{nast}(p, a)$  jest wartością  $SUF[last1]$ . W jednym przebiegu instrukcji „dla” tworzymy jeden lub dwa nowe węzły (węzły  $r$  i  $last1$ ). Oczywiście jest znaczenie nowego węzła  $last1$ , gdyż odpowiada on zbiorowi pozycji, zawierającemu ostatnią pozycję  $i$ . Kluczowe znaczenie dla zrozumienia algorytmu ma tworzenie nowego węzła  $r$ . Węzeł taki tworzymy, gdy  $\text{length}(p) + 1 < \text{length}(w)$ , gdzie  $p$  i  $w$  są takie jak w algorytmie. Niech  $\text{wart}(w)$  będzie słowem odpowiadającym węzlowi  $w$ . Wiemy, że sufiks  $y$  tego słowa o długości  $\text{length}(p) + 1$  jest taki, że  $i \in \text{end-pos}(y)$ , podczas gdy  $i \notin \text{end-pos}(\text{wart}(w))$ .

Ścieżki o wartościach  $\text{wart}(w)$  i  $y$ , prowadzące od korzenia w dół grafu, muszą zatem prowadzić do różnych węzłów (obecnie obie prowadzą do  $w$ ). Dlatego też trzeba utworzyć nowy węzeł  $r$ , do którego prowadzi ścieżka z etykietą  $y$ . Wiemy, że  $y = y'a$  oraz  $y'$  prowadzi do  $p$ . Wystarczy jedynie skierować krawędź o etykietie  $a$  z węzła  $p$  do nowego węzła  $r$ .

Algorytm ten ma złożoność liniową. Wystarczy jedynie pokazać, że całkowita liczba wykonanych instrukcji  $p := SUF[p]$  jest liniowa. Zauważmy, że każde wykonanie tej

instrukcji powoduje zmniejszenie głębokości węzła  $p$  w grafie. Jednocześnie w jednym przebiegu instrukcji „dla” zwiększamy tę głębokość o 1, gdyż w chwili, kiedy wykonujemy  $p := SUF[last]$ , głębokość  $SUF[last]$  jest co najwyżej o 1 większa od bieżącej głębokości  $p$ . Możemy teraz zastosować zasadę magazynu, podobnie jak przy obliczaniu tablicy  $P$  w algorytmie Knutha-Morrisa-Pratta. Koszt konstrukcji grafu  $G(x)$  jest zatem  $O(|x|)$ .

Graf  $G(x)$ , podobnie jak drzewo  $T(x)$ , możemy stosować do obliczania  $npowt(x)$  – najdłuższego powtarzającego się podslowa  $x$ ,  $maxsufix(x)$  – maksymalnego leksykograficznego sufiksu oraz liczby podslowa słowa  $x$ . Pokażemy Ci teraz, jak stosować  $G(x)$  do znalezienia liczby podslow. Liczba ścieżek prowadzących do węzła  $v$ , nie będącego korzeniem, jest równa  $length(v) - length(SUF[v])$ . Wystarczy zatem zsumować te liczby po wszystkich węzłach.

Algorytm tworzenia  $G(x)$  ma jeszcze inne nieoczekiwane zastosowanie. Założymy, że tekst  $x$  zaczyna się symbolem występującym jedynie na pierwszej pozycji. Jako efekt uboczny algorytm wyznacza drzewo sufiksowe  $T(x^R)$ , a więc daje alternatywny liniowy algorytm obliczania drzew sufiksowych. Korzeniem drzewa  $T(x^R)$  jest korzeń  $G(x)$ . Poprzednikiem (ojeem) węzła  $w$  jest węzeł  $v = SUF[w]$ . Jeśli  $v \neq w$  i węzłowi  $w$  odpowiada słowo  $b_1 \dots b_m$ , a węzłowi  $v$  słowo  $b_{i+1} \dots b_m$ , to krawędź od  $v$  do  $w$  jest etykietowana słowem  $b_1 b_{i+1} \dots b_i$ . Pozostawiamy Ci obliczanie tych etykiet w postaci  $x^R[p..q]$ , jak również pełną konstrukcję algorytmu obliczania  $T(x^R)$  na podstawie algorytmu obliczania  $G(x)$ .

Graf  $G(x)$  nie zawsze jest grafem minimalnym, reprezentującym zbiór słów  $Sub(x)$ . Weźmy tekst  $x = ab^{n-1}$ . Mamy  $2n-1$  zbiorów *end-pos* dla tego tekstu, a więc  $G(x)$  ma  $2n-1$  węzłów. Latwo natomiast podać graf reprezentujący  $Sub(x)$  i mający jedynie  $n+1$  węzłów. Jest to minimalna możliwa liczba węzłów. Algorytm tworzenia  $G(x)$  można zmodyfikować i otrzymać algorytm tworzenia grafu minimalnego w czasie liniowym. Jest to jednak dość skomplikowane, a liczba węzłów niewiele się zmienia ( $G(x)$  ma co najwyżej  $2n-1$  węzłów, a graf minimalny co najmniej  $n+1$ ). W wielu wypadkach  $G(x)$  jest grafem minimalnym, reprezentującym podslowa  $x$ .

Opisujemy teraz nieformalną konstrukcję typu off-line (oznacza to, że najpierw wczytujemy całe słowo wejściowe) grafu  $G(x)$ . Założymy, że tekst  $x$  kończy się specjalnym symbolem, nie występującym nigdzie indziej w  $x$ . Niech  $TI$  będzie niezwartym drzewem sufiksowym grafu  $G(x)$ . Przypomnijmy, że każdemu prefiksowi każdego sufiksu  $x$  w  $TI$  odpowiada dokładnie jeden węzeł.

Wprowadzamy następującą relację równoważności na zbiorze wierzchołków drzewa  $TI$ . Dwa wierzchołki są równoważne, gdy izomorficzne są poddrzewa, których są one korzeniami. Oznaczmy przez  $compress(TI)$  graf, który powstaje z  $TI$  przez sklejenie wszystkich równoważnych wierzchołków (wierzchołek w budowanym grafie odpowiada klasie równoważności wierzchołków  $TI$ ). Zauważymy następujący fakt (którego udowodnienie

pozostawiamy Tobie, Drogi Czytelniku):  $compress(TI)$  jest (z dokładnością do izomorfizmu) grafem podslowa słowa  $x$ .

W celu otrzymania efektywnej konstrukcji nie możemy zaczynać od grafu  $TI$ , ponieważ może on mieć rozmiar kwadratowy. Możemy natomiast zrobić prawie taką samą konstrukcję, posługując się (zwartym) drzewem sufiksowym  $T = T(x)$ .

Niech  $GI = compress(T)$ . Graf  $GI$  różni się od  $G(x)$  jedynie tym, że etykiety (będące podslowami  $x$ ) niektórych krawędzi w grafie  $GI$  mogą mieć długość większą niż 1, podczas gdy w grafie podslow  $G$  wszystkie etykiety są pojedynczymi literami. Opisujemy (nieformalnie) procedurę *Update(GI)*, która zamienia  $GI$  na  $G$  przez dodanie pewnej liczby nowych wierzchołków tak, żeby wszystkie etykiety były jednoliterowe. Procedura ta działa lokalnie dla każdego wierzchołka grafu  $GI$ . Przetwarzanie wierzchołka oznaczamy przez *LocalUpdate(v)*. Z krawędzi w grafie  $GI$ , prowadzących do wierzchołka  $v$ , wybierzmy krawędź  $(w, v)$  o najdłuższej etykiecie  $z = b_1 b_2 \dots b_k$ . Wtedy etykieta każdej krawędzi prowadzącej w  $GI$  do  $v$  jest sufiksem  $z$ . Utwórzmy  $k-1$  nowych wierzchołków  $v_1, v_2, \dots, v_{k-1}$  oraz krawędzie  $(v_i, v_{i+1})$  z etykietami  $b_{i+1}$  dla  $i = 0, \dots, k-1$ , gdzie  $v_0 = w$ ,  $v_k = v$ . Rozważmy kolejno każdą krawędź  $(w', v)$ . Jeśli etykieta tej krawędzi jest  $b_1 b_{j+1} \dots b_k$ , to poprowadźmy krawędź  $(w', v_{j+1})$  etykietowaną  $b_j$ . Usuńmy następnie krawędź  $(w', v)$ , jeśli ma ona etykię dłuższą niż 1. Algorytm ten umożliwia tworzenie grafu podslow w czasie liniowym przez obliczenie izomorficznych poddrzew w drzewie sufiksowym. Klasę izomorficznych poddrzew można obliczyć w drzewie w czasie liniowym.

## 5.3.

### Inne algorytmy tekstowe

Poza wyszukiwaniem wzorca można rozważyć wiele innych problemów związanych z tekstem. W tym podrozdziale zajmiemy się algorytmami umożliwiającymi ich rozwiązanie.

#### 5.3.1.

##### Obliczanie najdłuższego wspólnego podslowa

Mając dwa teksty:  $x$  i  $y$ , oznaczamy przez  $nwtext(x, y)$  najdłuższe podslowo występujące jednocześnie w  $x$  i w  $y$ .

Żeby obliczyć  $nwtext(x, y)$ , utwórzmy drzewo sufiksowe (lub graf podslow) dla słowa  $x\#y\#$ . Niech  $n = |y|$ , a  $m = |x|$ . Wystarczy znaleźć węzeł  $v$  o najdłuższym  $wart(v)$ , z którego prowadzi ścieżka do liścia odpowiadającego pewnemu sufiksowi  $suf(i)$  dla  $i < m$  oraz ścieżka do liścia  $suf(j)$  dla  $j > m$ . Ścieżka prowadząca do  $v$  odpowiada słowi  $nwtext(x, y)$ .

## 5.3.2.

## Obliczanie najdłuższego wspólnego podciagu

Przez  $nwpodciag(x, y)$  oznaczmy najdłuższy wspólny podciąg tekstów  $x$  i  $y$ . Problem ten jest tylko pozornie podobny do poprzedniego. Nie znamy dla niego algorytmu liniowego. Tworzymy tablicę  $A$  rozmiaru  $n \times m$ , taką że  $A[i, j]$  jest długością słowa  $nwpodciag(y[1..i], x[1..j])$ . Niech  $x = a_1 \dots a_m$ ,  $y = b_1 \dots b_n$ . Elementy tablicy wyznaczamy kolejno ze wzoru:

$$A[i, j] = \max(A[i-1, j], A[i, j-1], A[i-1, j-1] + \delta(b_i, a_j))$$

gdzie  $\delta$  jest funkcją dającą 1 dla równych argumentów, a 0 dla różnych. Koszt obliczania tablicy jest  $O(n \times m)$ . Po obliczeniu tablicy cofamy się od elementu  $A[n, m]$  zgodnie z wyborem maximum we wzorze na  $A[i, j]$ . W ten sposób obliczamy  $nwpodciag(x, y)$ . Dodatkowy koszt jest liniowy, jednakże całkowity koszt algorytmu (zdominowany przez koszt tworzenia tablicy  $A$ ) jest  $O(nm)$ .

Inicjowanie elementów  $A[i, 0]$ ,  $A[0, j]$  pozostawiamy Tobie jako zadanie.

W podobny sposób można obliczyć odległość redakcyjną  $edit(x, y)$  między tekstami  $x$  i  $y$ . Odległość taka jest równa minimalnej liczbie operacji „redakcyjnych”, dotyczących pojedynczych symboli, potrzebnych do przekształcenia  $x$  w  $y$ , a zatem operacji usuwania i wstawiania symbolu oraz operacji zamiany jednego symbolu na drugi. W celu policzenia odległości redakcyjnej tworzymy podobną tablicę  $A$  jak w rozwiązaniu problemu obliczania najdłuższego podsłowa.

## 5.3.3.

## Wyszukiwanie słów podwójnych

Problem wyszukiwania słów podwójnych (słów typu  $xx$ ) w czasie liniowym jest nietrywialny (ma raczej znaczenie dydaktyczne).

Bezpośredni algorytm sprawdzania w wypadku każdego podsłowa danego tekstu, czy jest ono słowem podwójnym, czy nie, ma złożoność rzędu  $n^3$ . Stosując algorytm KMP, można tę złożoność zmniejszyć do  $n^2$ . Można to zrobić w następujący sposób. Niech  $pref(i, u, v)$  oznacza długość najdłuższego prefiksu słowa  $u[i..|u|]$ , który jest prefiksem  $v$ . Inaczej mówiąc, jeżeli potraktujemy  $u$  jako tekst, w którym szukamy wzorca  $v$ , to definiowana wartość jest równa długości maksymalnego prefiksu wzorca, który „pasuje” do tekstu, począwszy od pozycji  $i$ . Wartości tej funkcji łatwo umieścić w tablicy o tej samej nazwie w czasie liniowym jako efekt uboczny algorytmu KMP. Niech  $x$  będzie słowem długości  $n$ . Słowo to zawiera podsłowo podwójne wtedy i tylko wtedy, gdy dla pewnych dwóch pozycji  $i < k$  zachodzi  $pref[i, x, x[k..n]] \geq k - i$ . Problem obliczania

## 5.3. Inne algorytmy tekstowe

## 191

predykatu  $squarefree$  ( $x$  nie zawiera słowa podwójnego) można zatem sprowadzić do liczenia liniowej liczby tablic typu  $pref$ .

Opiszymy teraz algorytm dotyczący rozwiązywania tego problemu w czasie  $O(n \log n)$ . Oznaczamy go przez ML od nazwiska jego autorów M. G. Maina i R. J. Lorentza.

W algorytmie ML podstawową operację jest  $test(u, v)$ . Chodzi tu o sprawdzenie, czy słowo  $uv$  zawiera podsłowo podwójne  $ww$ , które zaczyna się w  $u$ , a kończy w  $v$ . Opiszymy jedynie tę część działania tej operacji, która prowadzi do wykrycia takiego słowa  $ww$ , którego środek znajduje się w  $v$  i które ma ustaloną długość  $2k$  ( $k = |w|$ ). Tę nową (pod)operację opiszymy jako  $righttest(u, v, k)$ . Będzie nam potrzebna funkcja  $suf$  analogiczna do funkcji  $pref$ . Wartość  $suf[i, v, u]$  jest długością maksymalnego sufiksu słowa  $v$  kończącego się na  $i$ -tej pozycji, będącego jednocześnie sufiksem słowa  $u$ . Funkcję tę można stabilicować (umieszczając wartości w tablicy o tej samej nazwie  $suf$ ) w czasie liniowym.

Łatwo zauważać, że  $righttest(u, v, k) = \text{true}$  wtedy i tylko wtedy, gdy  $pref[k, v, v] > 0$  oraz  $pref[k, v, v] + suf[k, v, u] \geq k$ .

Mając już tablice  $pref$  i  $suf$ , możemy obliczyć predykat  $righttest(u, v, k)$  w czasie  $O(1)$ . Podobnie możemy zdefiniować predykat  $lefttest(u, v, k)$  umożliwiający wykrycie słowa podwójnego długości  $2k$  w  $uv$  o środku w  $u$ . Ponieważ predykat  $test$  ma wartość true, gdy któryś z predykatów  $lefttest$  i  $righttest$  ma wartość true dla pewnego  $k$ , można go policzyć w czasie liniowym (obliczając  $righttest$  i  $lefttest$  dla  $k = 1..n$ ). Oto nieformalny schemat algorytmu ML.

```
{Algorytm ML; sprawdzanie, czy tekst x zawiera słowo podwójne}
begin
  if |x| ≤ 2 then sprawdź bezpośrednio; {czas O(1)}
  else
    begin
      sprawdź rekurencyjnie, czy tekst x[1..└ n/2 ┘] zawiera słowo podwójne;
      sprawdź rekurencyjnie, czy x[└ n/2 ┘ + 1..n] zawiera słowo podwójne;
      sprawdź, czy zachodzi test(x[1..└ n/2 ┘], x[└ n/2 ┘ + 1..n])
      {czas O(n)}
    end
  end {algorytm ML};
```

Złożoność algorytmu jest  $O(n \log n)$ . W algorytmie są wykorzystywane dodatkowo tablice rozmiaru  $O(n)$ . Pokażemy teraz, jak obliczać predykat  $righttest$  w czasie liniowym z wykorzystaniem dodatkowej pamięci rozmiaru jedynie  $O(1)$ . Prowadzi to do algorytmu o tej samej strukturze co algorytm ML, działającego w tym samym czasie  $O(n \log n)$ , ale wymagającego dużo mniejszej pamięci. W algorytmie są wykorzystane pewne proste własności kombinatoryczne struktury wystąpień słów podwójnych.

Niech  $u$  i  $v$  będą dwoma słowami nie zawierającymi słowa podwójnego i niech  $right(u, v)$  będzie takim predykatem, że  $right(u, v) = \text{true}$  wtedy i tylko wtedy, gdy słowo  $uv$  zawiera słowo podwójne o środku w tekście  $v$ . Podobnie możemy zdefiniować  $left(u, v)$ . Zauważmy, że:  $test(u, v) = (left(u, v) \vee right(u, v))$ .

```
function right(u, v : tekst) : boolean; {u = a1a2...am, v = b1...bn}
begin
  i := n; t := n + 1; right := false;
  while i ≥ 1 and not right do
  begin
    j := i;
    while (j ≥ 1) and (m - i + j ≥ 1) and (am-i+j = bj) do j := j - 1;
    if j = 0 then right := true
    else
    begin
      k := i + j;
      if k < t then
      begin
        t := k;
        while (t > i) and (bt = bj+k-t) do t := t - 1;
        if t = i then right := true;
      end;
      if not right then i := max(j, ⌈i/2⌉) - 1
    end;
  end;
end;
```

Pozostawiamy Ci udowodnienie, że podany algorytm ma złożoność liniową ze względu na długość  $v$ . Poprawność algorytmu wynika z dwóch własności słów podwójnych.

- Słowo podwójne  $zz$  występujące w  $uv$  ma środek w słowie  $v$ , gdy  $uv = y_1zzy_2$  i  $|y_1z| \geq |u|$  dla pewnych słów  $y_1$  i  $y_2$ .
- Niech  $u$  i  $v$  będą tekstami nie zawierającymi słowa podwójnego i takimi, że złożenie  $uv$  zawiera słowo podwójne o środku w  $v$ . Założymy ponadto, że  $zzy$  jest najkrótszym prefiksem  $v$ , takim że  $y$  jest niepustym sufiksem  $u$  (a więc  $yzy$  jest słowem podwójnym o środku w  $v$ ). Wówczas słowo  $z$  jest najdłuższym słowem, które jest jednocześnie prefiksem i sufiksem  $zzy$ , a  $y$  jest najdłuższym wspólnym sufiksem tekstów  $u$  i  $z$ . Założymy dodatkowo, że  $v_1, v_2, y', z'$  są takimi słowami, że  $|v_1| = |v_2|$ ,  $v_1z'y'v_2z'$  jest prefiksem  $z$ ,  $zy$  jest właściwym prefiksem  $v_1z'y'$ ,  $y'$  jest sufiksem  $u$ , a  $z'$  jest najdłuższym wspólnym sufiksem słów  $v_1z'$  i  $v_1z'y'v_2z'$ . Wtedy  $zy$  jest właściwym prefiksem  $v_1z'$  lub  $|zy| < |v_1z'y'|/2$ , a ponadto  $zy$  jest właściwym prefiksem  $v_1z'y'v_2$ .

Dowód tych technicznych własności słów podwójnych pozostawiamy Tobie, Drogi Czytelniku.

Pokażemy teraz, jak można przedstawić problem słów podwójnych, używając struktur danych  $T(x)$  i  $G(x)$ . Zakładamy, że alfabet jest rozmiar  $O(1)$ . Do rozwiązania problemu wystarczy znaleźć w drzewie  $T(x)$  taki węzeł  $v$ , że w poddrzewie o korzeniu  $v$  znajdują się dwa różne liście  $suf(i)$  i  $suf(j)$ , takie że wartość  $|i - j|$  jest równa długości słowa odpowiadającego ścieżce od korzenia  $T(x)$  do węzła  $v$ . Jeśli takiego węzła nie ma, to słowo  $x$  nie zawiera słowa podwójnego ( $\text{squarefree}(x) = \text{true}$ ). Zamiast drzewa  $T(x)$  trzeba użyć zwartej reprezentacji  $T(x)$  danej drzewem sufiksowym. Znalezienie takiego węzła w czasie liniowym jest jednak dosyć trudne.

Pokażemy Ci teraz, jak zastosować graf  $G(x)$  do znajdowania słów podwójnych w czasie liniowym.

Załączamy, że rozmiar alfabetu jest ograniczony przez stałą. Rola  $G(x)$  sprawdza się tu do faktoryzacji tekstu  $x$ , która polega na obliczeniu ciągu niepustych słów  $(v_1, v_2, \dots, v_m)$ , zdefiniowanego w następujący sposób: niech  $|v_1 \cdot \dots \cdot v_{k-1}| = i$ , gdzie  $v_1 = x[1]$ , a  $v_k$  jest najdłuższym prefiksem  $u$  tekstu  $x[i+1..n]$ , występującym co najmniej dwa razy w tekście  $x[1..i]u$ ; jeśli takiego  $u$  nie ma, to  $v_k = x[i+1]$ . Niech  $pos(v_k)$  będzie ostatnią pozycją  $l < i$ , taką że wystąpienie  $v_k$  w  $x$  zaczyna się na pozycji  $l$ ; jeśli takiej pozycji nie ma, to przyjmujemy  $l = 0$ .

Faktoryzację tekstu i obliczenie wartości  $pos$  można wykonać w czasie liniowym, korzystając z grafu  $G(x)$ . Graf ten będzie tworzyć stopniowo, konstruując kolejno  $G(v_1 \cdot \dots \cdot v_k)$  dla  $k = 1, 2, \dots, m$ . Znalezienie  $v_k$ , jeżeli mamy  $G(v_1 \cdot \dots \cdot v_{k-1})$ , sprawdza się do przejścia pewnej ścieżki w  $G(v_1 \cdot \dots \cdot v_{k-1})$ .

Można teraz pokazać, że  $x$  zawiera słowo podwójne, gdy dla pewnego  $1 \leq k \leq m$  zachodzi  $pos(v_k) + |v_k| \geq |v_1 \cdot \dots \cdot v_{k-1}|$  ( $v_k$  „nakłada” się na siebie, przez co otrzymujemy słowo podwójne) lub  $left(v_{k-1}, v_k)$ , lub  $right(v_{k-1}, v_k)$ , lub  $right(v_1 \cdot \dots \cdot v_{k-2}, v_{k-1} \cdot v_k)$ .

Algorytm obliczania  $left$  jest analogiczny do obliczania  $right$ . Przypominamy, że koszt  $right(u, v)$  był liniowy ze względu na długość  $v$ . Fakt ten ma kluczowe znaczenie dla liniowości czasu całego algorytmu. Calkowity koszt jest zatem liniowy ze względu na sumaryczną długość  $v_1, \dots, v_k$ , a więc liniowy względem długości  $n$  tekstu wejściowego  $x$ . Dokładną konstrukcję algorytmu pozostawiamy Ci jako zadanie.

#### 5.3.4.

#### Wyszukiwanie słów symetrycznych

Niech  $w^R$  oznacza odwrócenie słowa. Przez **słowa symetryczne** (tzw. **palindromy**) rozumiemy słowa niepuste postaci  $ww^R$ .

Do znajdowania słów symetrycznych – podobnie jak do znajdowania słów podwójnych – możemy skorzystać z drzewa sufiksowego lub grafu podłów. Istnieje jednak prostszy

algorytm, umożliwiający obliczenie dla każdej pozycji  $i$  maksymalnego promienia  $R[i]$  palindromu o środku na pozycji  $i$  (promieniem palindromu  $vv^R$  jest długość  $v$ ), a dokładniej

$$R[i] = \max\{k \geq 0: x[i - k + 1..i] = (x[i + 1..i + k])^R\}$$

{Algorytm Manachera; obliczanie promieni słów symetrycznych}

```

begin
  {przyjmijmy, że  $x[1] = $$ ,  $x[n] = #$ }
   $R[1] := 0$ ;  $i := 2$ ;  $j := 0$ ;
  while ( $i \leq n$ ) do
    begin
      while  $x[i - j] = x[i + j + 1]$  do  $j := j + 1$ ;
       $R[i] := j$ ;
       $k := 1$ ;
      while ( $R[i - k] \neq R[i] - k$ ) and ( $k \leq j$ ) do
        begin  $R[i + k] := \min(R[i - k], R[i] - k)$ ;  $k := k + 1$  end;
         $j := \max(j - k, 0)$ ;  $i := i + k$ 
      end
    end {algorytm Manachera};
  
```

Poprawność algorytmu wynika z następującej własności promieni palindromów: jeśli dla  $k = 1..R[i]$  mamy  $R[i - k] \neq R[i] - k$ , to  $R[i + k] = \min(R[i - k], R[i] - k)$ .

Operacjami dominującymi w algorytmie są porównania  $x[i - j] = x[i + j + 1]$ . Porównań takich z wynikiem negatywnym jest wykonywanych co najwyżej  $n$ , a dla każdej wartości  $i$  co najwyżej jedno. W porównaniach z wynikiem pozytywnym wzrasta wartość sumy  $i + j$ . Wartość ta (przy porównywaniu symboli) nie maleje. Ponieważ maksymalną wartością  $i + j$  jest  $n$ , całkowita liczba wykonanych porównań symboli nie przekracza  $2n$ .

### 5.3.5. Równoważność cykliczna

Dwa słowa są cyklicznie równoważne, gdy są równe w sensie list cyklicznych, co zapisujemy jako  $x \equiv y$ . Problem polega na sprawdzeniu, czy dwa dane słowa są cyklicznie równoważne.

Wystarczy skorzystać z następującego faktu:  $x \equiv y$  wtedy i tylko wtedy, gdy  $x$  występuje (jako wzorzec) w tekście  $yy$ . Zakładamy tutaj, że  $|x| = |y|$ . Problem sprowadza się więc do problemu WW i jego koszt jest liniowy. Istnieje jednak prostszy algorytm, który nie wymaga wyszukiwania wzorca i korzystania z dodatkowej tablicy (pamięć  $O(1)$ ). Niech  $<$  będzie dowolnym porządkiem liniowym na zbiorze symboli.

{Algorytm sprawdzania, czy  $u \equiv w$ ; niech  $x = ww\$, y = null$ ,  $n = |u|$ }

```

begin
   $i := 0$ ;  $j := 0$ ;  $k := 1$ ;
  while ( $i < n$ ) and ( $j < n$ ) and ( $k \leq n$ ) do
    begin
       $k := 1$ ;
      while  $x[i + k] = y[j + k]$  do  $k := k + 1$ ;
      if ( $k \leq n$ ) then
        begin if  $x[i + k] > y[j + k]$  then  $i := i + k$  else  $j := j + k$  end;
        {niewmiennik *}
      end;
      if ( $k > n$ ) then return ( $u \equiv w$ ) else return (nie zachodzi  $u \equiv w$ )
    end {algorytm};
  
```

Niech  $u^{(k)} = u[k + 1..n]u[1..k]$ , a więc niech  $u^{(k)}$  powstaje przez cykliczne przesunięcie  $u$ . Niech  $D1 = \{1 \leq k \leq n: u^{(k-1)} \gg u^{(0)}$  dla pewnego  $j\}$  i niech  $D2 = \{1 \leq k \leq n: u^{(k-1)} \gg u^{(0)}$  dla pewnego  $j\}$ , gdzie  $\gg$  oznacza rozszerzenie  $>$  na słowa (leksykograficznie). Poprawność algorytmu wynika stąd, że jeśli  $D1$  lub  $D2$  jest zbiorem  $\{1, 2, \dots, n\}$ , to nie zachodzi  $u \equiv w$ . Ponadto zachodzi niewmiennik \*:  $\{1, 2, \dots, i\} \subseteq D1, \{1, 2, \dots, j\} \subseteq D2$ . Algorytm ma złożoność liniową. Największa liczba porównań symboli jest wykonywana dla słów  $u$  i  $w$  postaci (odpowiednio) 11...1201, 111...120.

### 5.3.6. Algorytm Huffmmana

Kompresję tekstów można rozumieć na różne sposoby. Dla nas będzie ona oznaczać redukcję binarnego zapisu danego tekstu. Założymy, że dla danego tekstu  $x = a_1a_2\dots a_n$  i alfabetu  $I$  chcemy znaleźć jednoznaczny kod  $h$  symboli  $a_i$  alfabetu słowami binarnymi  $h(a_i)$  tak, żeby długość tekstu  $h(a_1) + \dots + h(a_n)$  była minimalna. Rozważamy tutaj klasę takich kodowań, że  $h(a_i)$  nie jest prefiksem  $h(a_j)$  dla żadnych dwóch różnych symboli  $a_i$  i  $a_j$ . Zbiór kodów symboli może być reprezentowany drzewem, którego ścieżki odpowiadają kodom poszczególnych symboli. Bit 0 oznacza „idź w lewo” w drzewie, a bit 1 – „idź w prawo”.

Rozważmy przykład  $x = abcdcdcd$ . Jeżeli zakodujemy każdy symbol ciągiem dwubitowym, to otrzymany kod będzie miał długość  $2n = 20$ . Przedstawimy teraz algorytm opracowany przez D. Huffmmana. Istota tego algorytmu sprowadza się do kodowania częściej występujących symboli krótszymi ciągami binarnymi, a rzadziej występujących symboli – dłuższymi. Przedstawimy na naszym przykładzie działanie algorytmu w wersji rekurencyjnej. Częstość występowania w słowie  $x$  poszczególnych symboli jest następująca:

$$\text{częstość}(a) = 1, \text{częstość}(b) = 2, \text{częstość}(c) = 3, \text{częstość}(d) = 4.$$

Znajdujemy dwa symbole, których suma częstości jest minimalna. Są to symbole  $a$  i  $b$ . Zamieniamy je oba na jeden nowy symbol, na przykład na  $e$ . Otrzymujemy tekst, w którym częstość występowania symboli jest następująca:

$$\text{częstość}(e) = 3, \text{częstość}(c) = 3, \text{częstość}(d) = 4.$$

(Częstość występowania symbolu  $e$  jest sumą częstości występowania „sklejonych” symboli).

Obliczamy teraz rekurencyjnie minimalny kod dla symboli  $e$ ,  $c$  i  $d$ . Nietrudno zauważać, że jest on następujący:  $h(e) = 00$ ,  $h(c) = 01$  i  $h(d) = 1$ , gdyż częstość  $d$  jest maksymalna. Następnie „rozklejamy” symbol  $e$ . Kody syniboli  $a$  i  $b$  są takie same jak symbolu  $e$ , z wyjątkiem dodania jednego bitu rozróżniającego te symbole. Mamy więc  $h(a) = h(e)0$ ,  $h(b) = h(e)1$ , czyli ostatecznie:  $h(a) = 000$ ,  $h(b) = 001$ ,  $h(c) = 01$ ,  $h(d) = 1$ .

Po zakodowaniu długość tekstu binarnego wynosi 19, a więc mniej niż długość uzyskana kodowaniem bezpośrednim (gdy każdy symbol ma tę samą liczbę bitów).

Schemat algorytmu Huffmana w wersji rekurencyjnej wygląda następująco:

```
{Algorytm Huffmana; kompresja zapisu binarnego tekstów nad alfabetem o liczbie symboli  $\geq 2$ }
begin
  if liczba symboli = 2 then kod każdego symbolu składa się z jednego bitu
  else
    begin
      niech  $a$  i  $b$  będą symbolami o najmniejszej częstości; zastąp je nowym symbolem  $a'$  o częstości będącej sumą częstości symboli  $a$  i  $b$ ;
      wywołanie rekurencyjne algorytmu Huffmana;
      niech  $h$  będzie otrzymanym kodem;
       $h(a) := h(a')0$ ;  $h(b) := h(a')1$ 
    end {algorytm Huffmana};
```

Operacjami dominującymi w algorytmie są: pobranie elementu o minimalnej częstości i wstawienie nowego elementu o danej częstości. Operacje te można wykonywać w czasie  $O(\log n)$  za pomocą kopca jako pomocniczej struktury danych. Całkowity koszt algorytmu Huffmana jest  $O(n \log n)$ .

### 5.3.7.

#### Obliczanie leksykograficznie maksymalnego sufiksu

Przez  $\text{maxsufiks}(x)$  oznaczmy leksykograficznie największy sufiks słowa  $x$ , na przykład  $\text{maxsufiks}(ababbaaaaa) = bbaaaaaa$ .

Bardzo łatwo można go obliczyć, stosując strukturę danych  $T(x)$  lub  $G(x)$ . Wystarczy tylko przejść leksykograficznie największą ścieżką. Istnieje jednak znacznie prostszy algorytm, w którym nie korzysta się z żadnej z tych struktur – nawet z tablicy pomocniczej (jedynie z kilku zmiennych). Niech  $<$  będzie relacją porządku liniowego w alfabetie.

```
{Algorytm Duvala; obliczanie maxsufiks(x)}
begin
  i := 0; j := 1; k := 1; p := 1;
  while (j + k  $\leq |x|$ ) do
    begin
      a := x[i + k]; b := x[j + k];
      if a < b then begin i := j; j := j + 1; k := 1; p := 1 end
      else if a > b then begin j := j + k; k := 1; p := j - i end
      else if (a = b) and (k  $\neq$  p) then k := k + 1
      else begin j := j + p; k := 1 end;
    end {instrukcji while};
    maxsufiks := suf(i + 1)
  end {algorytm Duvala};
```

Poprawność algorytmu wynika z pewnych własności kombinatorycznych funkcji  $\text{maxsufiks}$ . Niech  $\text{maxsufiks}(x) = z = (u)v$ , gdzie  $u$  jest najkrótszym okresem  $z$ , a  $v$  jest prefiksem właściwym  $u$  (być może pustym). Niech  $\text{per}(x) = u$  i  $\text{rest}(x) = v$ . Zauważmy, że maksymalny sufiks tekstu  $xa$  jest zawsze sufiksem tekstu  $\text{maxsufiks}(x)a$ . Niech  $a'$  będzie takim symbolem, że  $\text{rest}(x)a'$  jest prefiksem  $\text{per}(x)$ .

Funkcje  $\text{per}$  i  $\text{rest}$  mają następujące własności:

- $\text{maxsufiks}(xa) = \text{if } a \leq a' \text{ then } \text{maxsufiks}(x)a \text{ else } \text{maxsufiks}(\text{rest}(x)a);$
- $\text{per}(xa) = \text{if } a < a' \text{ then } \text{maxsufiks}(x)a \text{ else if } a = a' \text{ then } \text{per}(x) \text{ else } \text{per}(\text{rest}(x)a);$
- $\text{rest}(xa) = \text{if } a < a' \text{ lub } (a = a' \text{ oraz } \text{rest}(x)a = \text{per}(x)) \text{ then słowo puste}$   
 $\text{else if } (a = a' \text{ oraz } \text{rest}(x)a < \text{per}(x)) \text{ then } \text{rest}(x)a \text{ else } \text{rest}(\text{rest}(x))a.$

Poprawność algorytmu wynika z tych własności, ponieważ za jego pomocą można obliczyć iteracyjnie  $\text{maxsufiks}(x'u)$ , mając policzony  $\text{maxsufiks}(x')$ , gdzie  $x' = x[1..j + k]$  i  $a$  jest następnym symbolem tekstu ( $a = x[j + k + 1]$ ). W danym momencie  $\text{maxsufiks}(x') = x[i + 1..j + k]$ ,  $\text{rest}(x') = x[j + 1..j + k]$  oraz  $p$  jest długością okresu  $\text{per}(x')$ .

Czas działania algorytmu jest liniowy; można to udowodnić, korzystając z faktu, że wartość sumy  $i + j + k$  wzrasta w każdym kroku. Jej minimalną wartością jest 2, a maksymalną  $2|x|$ .

Efektem ubocznym algorytmu Duvala jest dokładne obliczenie okresu słowa  $x$  ( $\text{per}(x)$ ) w przypadku, gdy  $x$  ma okres krótszy niż połowa długości słowa  $x$ . Takie słowa  $x$  nazywamy **mocno określonymi**. Niech  $\text{maxsufiks}(x) = u'v$ , gdzie  $u$  jest najkrótszym okresem maksymalnego sufiksu. W algorytmie Duvala oblicza się również  $u$ ,  $e$  i  $v$ , a zatem  $x = wu'v$ . Czwórkę wartości ( $w$ ,  $u$ ,  $e$ ,  $v$ ) nazywamy **dekompozycją sufiksową**. Oto jej własności.

- (1) Jeśli słowo  $x$  jest mocno okresowe, to  $per(x) = |u|$ .
- (2) Jeśli dana jest dekompozycja sufiksowa, to sprawdzenie, czy  $|u|$  jest okresem całego słowa  $x$ , można wykonać w czasie  $O(|u|)$ .
- (3) Jeśli słowo  $x$  jest mocno okresowe i mamy dekompozycję sufiksową  $x$ , to dekompozycję sufiksową słowa  $x$  obciętego o  $per(x)$  można obliczyć w czasie stałym.

Korzystając z powyższych własności, można stosunkowo łatwo napisać algorytm dla problemu wyszukiwania wzorca, działający w czasie liniowym i z wykorzystaniem (dodatkowej) pamięci stałej. Algorytm ten jest symulacją algorytmu KMP bez dodatkowej tablicy  $P$ . Symuluje algorytm Duvala. Oblicza się w nim dekompozycję sufiksową słowa  $x[1..j+1]$ , mając daną dekompozycję sufiksową słowa  $x[1..j]$ . Został opracowany przez M. Crochemore'a. W algorytmie tym przyjmujemy:

$$\text{przesunięcie}'(j) = \begin{cases} per(x[1..j]), & \text{jeśli słowo } x[1..j] \text{ jest mocno okresowe} \\ \lfloor j/2 \rfloor & \text{w przeciwnym wypadku} \end{cases}$$

```
{Algorytm Crochemore'a}
begin {x jest wzorcem}
  i := 1; j := 0;
  while i ≤ n - m + 1 do
    {niezmienik: znałmy dekompozycję sufiksową słowa x[1..j]}
    begin
      while x[j + 1] = y[i + j] do
        begin
          j := j + 1;
          oblicz dekompozycję sufiksową x[1..j], korzystając
          z dekompozycji sufiksowej słowa x[1..j - 1]
          {częściowa symulacja algorytmu Duvala}
        end;
        if j = m then write(i); {wzorzec znaleziony}
        oblicz przesunięcie'(j), korzystając z dekompozycji
        sufiksowej x[1..j];
        i := i + przesunięcie'(j);
        if przesunięcie'(j) < [j/2] {przesunięcie = okres słowa
          x[1..j]}
        then
        begin
          oblicz dekompozycję sufiksową
          x[1..j - przesunięcie'(j)], korzystając z własności
          (3); j := j - przesunięcie'(j);
        end
        else j := 0
      end
    end {algorytm Crochemore'a};
```

## 5.3.8.

## Jednoznaczne kodowanie

Niech dany będzie ciąg tekstów:  $kod(1), kod(2), \dots, kod(k)$ . Możemy go traktować jako funkcję  $kod$ , która każdemu elementowi z alfabetu  $\{1, \dots, k\}$  przyporządkowuje tekst.

Funkcję tę możemy rozszerzyć na zbiór wszystkich słów nad alfabetem  $\{1, \dots, k\}$  w następujący sposób:

$$kod^*(i_1 i_2 \dots i_r) = kod(i_1)kod(i_2) \dots kod(i_r)$$

Kodowanie jest jednoznaczne wtedy, kiedy funkcja  $kod^*$  jest różnowartościowa. Chcemy sprawdzić, czy zadany ciąg tekstów jest kodem jednoznaczny. Rozmiarem problemu jest suma długości tekstów  $kod(1), \dots, kod(k)$ , którą oznaczamy przez  $n$ .

Sprawdzimy ten problem do znajdowania ścieżki w pewnym grafie skierowanym. Węzłami grafu są sufiksy słów  $kod(1), kod(2), \dots, kod(k)$ . W szczególności przyjmujemy, że węzłem jest słowo puste. Przyjmijmy również, że  $suf_{p+1}(u)$  jest słowem pustym, gdy  $|u| = p$ .

Oznaczmy zbiór węzłów przez  $V$ . Zauważmy, że  $|V| \leq n + 1$ , gdzie  $n$  zostało zdefiniowane jako sumaryczna długość słów kodowych.

Niech zapis  $x//y$  oznacza tutaj operację obcinania słowa  $x$  o słowo  $y$ , tzn.  $x//y = z$ , jeżeli  $x = yz$ . Operacja ta jest zdefiniowana częściowo; wynik jest określony, gdy  $y$  jest prefiksem  $x$ .

Niech  $suf_k(i)$  oznacza sufiks rozpoczynający się od pozycji  $k$  w słowie  $kod(i)$ . Dla każdego elementu  $v = suf_k(i)$  i słowa  $kod(j)$  tworzymy krawędź skierowaną:

- (a)  $(suf_k(i), suf_r(j))$ , jeżeli  $suf_r(j) = kod(j) // v$  i operacja ta jest określona;
- (b)  $(suf_k(i), suf_r(i))$ , jeżeli  $suf_r(i) = v // kod(j)$  i operacja ta jest określona.

Niech  $V_0$  oznacza zbiór węzłów początkowych. Są to węzły odpowiadające wszystkim słownom niepustym postaci  $kod(i) // kod(j)$  dla  $i \neq j$ . Niech węzłem końcowym będzie słowo puste. Nietrudno zauważyc, że kod nie jest jednoznaczny wtedy i tylko wtedy, gdy w odpowiadającym mu grafie istnieje ścieżka od węzła początkowego do węzła końcowego.

Otrzymany graf  $G$  ma co najwyżej  $n^2$  krawędzi. Koszt przeszukania  $G$  jest więc tego samego rzędu. Mniej oczywisty jest koszt tworzenia grafu  $G$ . Musimy umieć szybko odpowiedzieć na pytanie, czy para słów  $(v, u)$  jest krawędzią, co sprowadza się jednak do problemu WW. Istnienie na przykład krawędzi na mocy warunku (b) ( $suf_r(i) = v // kod(j)$  i operacja  $//$  jest określona) jest równoważne występowaniu wzorca  $kod(j)$ , od pozycji  $r$  w słowie  $kod(i)$ . Łatwo obliczyć wszystkie problemy WW dla tekstu  $kod(i)$  i wzorców

*kod(j)* w (sumarycznym) czasie  $O(n^2)$ , stosując wielokrotnie jeden z algorytmów wyszukiwania wzorca w czasie liniowym. Reasumując: problem jednoznaczności kodowania możemy rozwiązać w czasie  $O(n^2)$ .

### 5.3.9.

## Liczenie liczby podłów

Załóżmy, że rozmiar alfabetu jest stały. Pokażemy, jak obliczać w czasie liniowym liczbę wszystkich różnych podłów danego słowa  $x$ . Oznaczmy tę liczbę przez  $count(x)$ . Pokażemy najpierw, jak można skorzystać z grafu podłów  $G(x)$ . Ponieważ liczba ścieżek prowadzących do węzła  $v$  nie będącego korzeniem jest równa  $length(v) \cdot length(SUF[v])$ , wystarczy zsumować te liczby po wszystkich węzłach  $v$ . W ten sposób otrzymamy liczbę  $count(x)$ .

Jeśli mamy  $G(x)$  bez tablicy  $SUF$ , to  $count(x)$  możemy obliczyć jako liczbę wszystkich ścieżek w grafie  $G(x)$ , prowadzących od korzenia do jakiegokolwiek innego węzła. Liczbę ścieżek w grafie skierowanym acyklicznym łatwo obliczyć, korzystając z sortowania topologicznego takiego grafu. Liczbę ścieżek rozpoczętymi się od każdego węzła liczymy, przeglądając węzły „od końca”.

Ten sam problem można rozwiązać za pomocą drzewa sufiksowego. Niech  $T(x)$  będzie drzewem sufiksowym. Wagę krawędzi definiujemy jako długość słowa odpowiadającego tej krawędzi. Liczba  $count(x)$  jest sumą wag wszystkich krawędzi drzewa sufiksowego.

## Zadania

- 5.1. Posługując się tablicą  $P$ , skonstruj algorytm sprawdzania w czasie liniowym, czy dany tekst zaczyna się słowem postaci  $ww$  lub słowem postaci  $www$ .
- 5.2. Skonstruj algorytm liniowy obliczania tablic  $pref^*, u, v]$ ,  $suf^*, u, v]$ , zdefiniowanych przy omawianiu słów podwójnych. (Zmodyfikuj algorytm KMP).
- 5.3. Skonstruj algorytm, który umożliwia sprawdzenie w czasie liniowym i z wykorzystaniem pamięci pomocniczej  $O(1)$ , czy dany tekst jest słowem Fibonacciego.
- 5.4. Udowodnij, że  $delay(m) = O(\log m)$ , gdzie  $delay(m)$  jest funkcją zdefiniowaną przy analizowaniu algorytmu KMP'.
- 5.5. W algorytmie KMP' czasami „przeczekujemy”  $O(\log m)$  kroków przed wczytaniem kolejnego tekstu. Przerób ten algorytm tak, żeby odstępy czasowe między kolejnymi wczytaniami były ograniczone przez pewną stałą i żeby po wczytaniu

## Zadania

- 5.6. Udowodnij, że po obcięciu dwóch ostatnich symboli słowa Fibonacciego są palindromami oraz że  $fib_n fib_{n+1} = fib_{n+1} fib_n$  z dokładnością do zamiany dwóch ostatnich symboli.
- 5.7. Niech wzorzec  $x$  będzie słowem Fibonacciego. Udowodnij, że  $NEXT[F_k - 1] = F_{k-1} - 1$  oraz  $P[j] = j - F_{k-1}$  dla  $F_k \leq j < F_{k+1}$ , gdzie  $k > 1$ .
- 5.8. Skonstruj algorytm liniowy obliczania tablicy  $P$  dla drzewa  $T$  prefiksów wzorców ze zbioru wzorców  $X$ .
- 5.9. Napisz pełną wersję algorytmu AC (Aho-Corasicka) szukania wielu wzorców w czasie liniowym ze względu na sumaryczną długość wszystkich wzorców i tekstu do sprawdzenia.
- 5.10. Napisz pełną wersję algorytmu Bak (opracowanego przez T. Bakera) szukania wzorców dwuwymiarowych w czasie liniowym ze względu na całkowity rozmiar danych ( $n^2$  dla tablicy o wymiarach  $n \times n$ ).
- 5.11. Udowodnij, że algorytm GS' ma koszt liniowy.
- 5.12. Zastosuj algorytm KMR do znalezienia maksymalnej długości  $r$  takiej, że w tekście występuje dwukrotnie ( $k$ -krotnie) to samo słowo o tej długości.
- 5.13. Napisz wersję algorytmu KMR dla przypadku dwuwymiarowych i – ogólnie –  $k$ -wymiarowych wzorców ( $k$  jest stałą całkowitą). Koszt algorytmu ma być  $O(n \log n)$ , gdzie  $n$  jest całkowitym rozmiarem danych wejściowych.
- 5.14. Skonstruj algorytm liniowy obliczania tablic  $d1$  i  $d2$ , wykorzystywanych w algorytmie BM (skorzystaj z modyfikacji algorytmu KMP i liczenia tablicy  $P$ ).
- 5.15. Podaj wzór na liczbę wykonanych porównań symboli w algorytmie BM dla  $x = ca(ba)^k$  i  $y = a^{2k+2}(ba)^k$ . Ille zostanie wykonanych porównań, gdy zamiast  $d2$  użyjemy w algorytmie jako przesunięcia wartości  $d1$ ?
- 5.16. Udowodnij, że obliczanie iloczynu logicznego ma ten sam rzad złożoności co obliczanie iloczynu tekstowego (definicje podane przy opisie algorytmu FP).
- 5.17. Udowodnij, że liczba krawędzi grafu podłów  $G(x)$  nie przekracza  $3|x|$ .

- 5.18. Skonstruj pełny algorytm liniowy obliczania drzewa sufiksowego na bazie algorytmu tworzenia  $G(x)$  (jako efekt uboczny tego algorytmu). Założmy, że rozmiar alfabetu jest  $O(1)$ .

- 5.19. Niech  $T$  będzie drzewem sufiksowym dla wzorca  $x$  kończącego się specjalnym symbolem. Niech  $sext$  będzie taką tablicą, że  $sext[a, v]$  jest węzłem odpowiadającym minimalnemu słowi  $y$ , dla którego  $a \cdot \text{wart}(v)$  jest prefiksem (niekoniecznie właściwym). Jeżeli takiego węzła nie ma, to  $sext[a, v] = \text{nil}$ .

Oblicz tablicę  $sext$  w czasie liniowym. Skonstruj algorytm obliczania drzewa sufiksowego  $T(x)$ , w którym zamiast z tablic  $TEST$  i  $LINK$  korzysta się jedynie z tablicy  $sext$  (liczonej w czasie algorytmu). Koszt algorytmu ma być liniowy.

- 5.20. Niech  $x$  kończy się specjalnym symbolem. Udowodnij, że tablica  $sext$  dla  $T(x)$  daje reprezentację grafu  $G(x^k)$  w sensie  $nast(a, v) = sext[a, v]$ . Napisz alternatywny algorytm liniowy tworzenia grafu  $G(x)$ , będącego efektem ubocznym obliczania drzewa sufiksowego.

- 5.21. Niech  $G$  będzie grafem podłów tekstu  $\$x$ , gdzie  $\$$  występuje jedynie na początku tekstu, a więc  $G = G(\$x)$ . Podaj prostą transformację grafu  $G$  w graf  $G' = G(x)$ .

- 5.22. Podaj pełny algorytm liniowy sprawdzania, czy dany tekst zawiera słowo podwójne.

- 5.23. Algorytm Manacher'a umożliwia obliczanie promieni palindromów parzystych (o długości parzystej, postaci  $vv^k$ ). Podaj wersję tego algorytmu dla obliczania promieni palindromów nieparzystych (postaci  $vav^k$ ) w czasie liniowym.

- 5.24. Skonstruj algorytm liniowy sprawdzania, czy dany tekst jest złożeniem pewnej liczby palindromów o długości parzystej (słów postaci  $ww^k$ ). Skorzystaj z tablicy promieni palindromów.

- 5.25. Niech  $depth(v)$  oznacza głębokość węzła  $v$  w drzewie sufiksowym  $T$ . Udowodnij nierówność:  $depth(LINK[a, v]) \leq depth(v) + 1$ .

- 5.26. Skonstruj algorytm liniowy sprawdzania, czy tekst jest złożeniem trzech palindromów parzystych.

- 5.27. Podaj pełny algorytm rozwiązywania w czasie liniowym problemu obliczania odległości redakcyjnej. Oblicz w czasie liniowym minimalny ciąg operacji redakcyjnych.

- 5.28. Zmodyfikuj algorytm obliczania odległości redakcyjnej tak, żeby dla danych dwóch tekstów  $x$  i  $y$  była liczona minimalna odległość między  $x$  i pewnym pod-

slowem  $y$ . Znajdź podslowo  $y$  o minimalnej odległości od  $x$  (jest to dopasowywanie wzorca z błędami).

- 5.29. Udowodnij poprawność algorytmu Crochemore'a obliczania wartości  $\text{maxsufiks}(x)$ .

- 5.30. Oszacuj dokładnie maksymalną liczbę porównań symboli w algorytmie sprawdzania równoważności cyklicznej dwóch tekstów ( $u \equiv w$ ) za pomocą podanego na str. 195 algorytmu.

- 5.31. Udowodnij, że aby sprawdzić jednoznaczność kodu  $kod(1)$  i  $kod(2)$  dwóch elementów, wystarczy jedynie sprawdzić, czy nie zachodzi równość:  $kod(1)kod(2) = = kod(2)kod(1)$ .

- 5.32. Napisz iteracyjną wersję algorytmu Huffmmana. Udowodnij, że wyznaczone w wyniku działania algorytmu kodowanie jest optymalne.

- 5.33. Udowodnij, że  $\text{rightest}(u, v, k) = \text{true}$  wtedy i tylko wtedy, gdy  $\text{pref}[k, v, v] > 0$  oraz  $\text{pref}[k, v, v] + \text{suf}[k, v, u] \geq k$ . Definicje tablic  $\text{pref}$  i  $\text{suf}$  oraz definicja predykatu  $\text{rightest}$  są podane przy omawianiu słów podwójnych.

- 5.34. Skonstruj algorytm rozwiązywania problemu wyszukiwania słów podwójnych w czasie  $O(n \log n)$  i z wykorzystaniem pamięci  $O(1)$ .

- 5.35. Udowodnij własności słów podwójnych podane przy omawianiu algorytmu obliczania  $\text{right}(u, v)$  w czasie liniowym i z wykorzystaniem pamięci stałej. Pokaż również, że algorytm jest poprawny i jego złożoność jest liniowa względem  $|v|$ .

- 5.36. Udowodnij, że  $x$  zawiera słowo podwójne, gdy dla pewnego  $1 \leq k \leq m$  zachodzi  $\text{pos}(v_k) + |v_k| \geq |v_1 \cdot \dots \cdot v_{k-1}|$  (v\_k „nakłada” się na siebie, przez co otrzymujemy słowo podwójne) lub  $\text{left}(v_{k-1}, v_k)$ , lub  $\text{right}(v_{k-1}, v_k)$ , lub  $\text{right}(v_1 \cdot \dots \cdot v_{k-2}, v_{k-1}v_k)$ . Zapis  $v_1 \cdot \dots \cdot v_{k-1}v_k$  oznacza faktoryzację tekstu  $x$ .

- 5.37. Podaj pełny algorytm faktoryzacji tekstu.

- 5.38. Udowodnij, że jeśli  $R[i-k] \neq R[i] - k$  dla  $k = 1..R[i]$ , to  $R[i+k] = \min(R[i-k], R[i] - k)$ .

- 5.39. Udowodnij, że algorytm Manachera ma złożoność liniową.

- 5.40. Skonstruj algorytm liniowy obliczania minimalnego grafu reprezentującego zbiór  $\text{Sub}(x)$  wszystkich podłów tekstu  $x$ . „Sklej” równoważne węzły grafu  $G(x)$ . Węzły  $v1, v2$  odpowiadające podslowom  $x1$  i  $x2$  (w takim sensie, że  $\text{wart}(v1) = x1$ , a  $\text{wart}(v2) = x2$ ) są równoważne, gdy  $\text{Sub}(x)/x1 = \text{Sub}(x)/x2$ . Klasa równoważności są tutaj co najwyżej dwuelementowe. Zakładamy, że alfabet ma rozmiar  $O(1)$ .

- 5.41. Założmy, że alfabet jest „mały” i że niektóre pary symboli są przemienne (na przykład  $ab \equiv ba$ ). Dwa słowa są równoważne, gdy możemy otrzymać jedno z drugiego za pomocą pewnej liczby zamian sąsiadujących ze sobą symboli, które są wzajemnie przemienne. Jeśli na przykład w alfabetie  $\{a, b, c\}$  jedyną parą przemienią jest para symboli  $a$  i  $b$ , to  $abbacb \equiv baabcb$  i nie zachodzi  $acb \equiv bca$ . Skonstruuj algorytm liniowy sprawdzania, czy dwa słowa długości  $n$  są równoważne. Zauważmy, że dla danego tekstu możemy mieć wykładniczą liczbę tekstów równoważnych.
- 5.42. Założmy, że wagi elementów są posortowane. Pokaż, jak zaimplementować wtedy algorytm Huffmmana tak, żeby działał w czasie liniowym (wskazówka: skorzystaj ze stosu; nowe elementy są przesyłane na stos, który automatycznie jest sortowany).
- 5.43. Napisz dokładną implementację algorytmu Crochemore'a rozwiązywania problemu WW w czasie liniowym i z wykorzystaniem (dodatkowej) pamięci stałej (w rozdziale tym opisaliśmy jedynie nieformalny schemat tego algorytmu). Jest to przykład konstrukcji algorytmu metodą „transformacyjną”. Algorytm Duvala jest przekształcony na algorytm dopasowywania wzorca. W algorytmie Duvala liczymy leksyko-graficznie maksymalny sufiks, podczas gdy w problemie WW to nas nie interesuje (korzystamy jedynie z ubocznego efektu algorytmu Duvala). W swoim programie powinieneś użyć jedynie kilku zmiennych całkowitych. Zakładamy, że wzorzec  $x$  i tekst  $y$  są umieszczone w tablicach, które mogą być tylko odczytywane (nie mogą być modyfikowalne). Tablice te nie są traktowane jako „pamięć”, a jako dane wejściowe.

## 6 Algorytmy równoległe

Standardowym modelem obliczeń sekwencyjnych jest maszyna ze swobodnym dostępem do pamięci. Model ten jest oznaczany jako RAM (skrót od ang. *Random Access Machine*). Niestety w wypadku obliczeń równoległych nie ma podobnego (w porównaniu z sytuacją w obliczeniach sekwencyjnych) standardu. Spowodowane jest to tym, że istnieje niewiele komputerów naprawdę równoległych (z dużą liczbą procesorów), a poza tym nie wiadomo, jaka będzie w przyszłości najlepsza technologia tych maszyn. Jeśli jednak chodzi o przedstawianie i analizowanie algorytmów (mniej o realizację), to najbardziej rozpowszechnionym modelem obliczeń równoległych jest maszyna równoległa ze swobodnym dostępem do pamięci, oznaczana jako PRAM (skrót od ang. *Parallel Random Access Machine*).

PRAM jest „wyidealizowanym” modelem obliczeń. W modelu tym pomija się wiele szczegółów technicznych, zwłaszcza te, które są związane ze wzajemną komunikacją i synchronizacją procesorów. Przyjmuje się, że każdy procesor może komunikować się z innym w stałym czasie przy użyciu wspólnej pamięci. Wynikająca stąd bardzo duża liczba połączeń nie jest realizowalna przy współczesnej technologii. Jednakże model ten jest bardzo przydatny do opisu algorytmów równoległych, właśnie z powodu pominięcia tych szczegółów technicznych. Co więcej, PRAM daje się symułować (implementować) na bardziej realistycznych modelach. Złożoność niewiele się wtedy pogarsza, czas zwiększa się o czynnik  $\log^k n$ , a w wielu typowych sytuacjach rząd złożoności w ogóle się nie zmienia (na przykład w wypadku sortowania i mnożenia macierzy).

Model obliczeniowy PRAM składa się z pewnej (z reguły dużej) liczby procesorów. Każdy z nich jest maszyną ze swobodnym dostępem do pamięci (RAM). Procesory pracują synchronicznie; w jednym kroku jest wykonywana jedna instrukcja dla każdego procesora. Żeby uniknąć wchodzenia w szczegóły technologiczne, opiszymy jedynie ogólną konstrukcję programistyczną, umożliwiającą bardzo proste wyrażanie równoległości obliczeń. Konstrukcją tą jest równoległa wersja instrukcji „dla” (for).

```
for each  $x \in X$  do in parallel  $akcja(x)$ ;
```

gdzie  $akcja(x)$  jest pewną operacją zależną od parametru  $x$ .

W wyniku wykonania na przykład instrukcji

```
for each  $i \in [1..n]$  do in parallel  $A[i] := 2 * i$ ;
```

każdemu z elementów tablicy  $A$  zostanie przypisana wartość  $2*i$ , gdzie  $i$  jest pozycja w tablicy.

Istnieją pewne niebezpieczeństwa natury semantycznej, związane z tak ogólnym modelem. Jeżeli na przykład  $i$ -ty procesor, gdzie  $i \in [1..n]$ , chce przypisać tej samej zmiennej  $x$  wartość  $2*i$ , to nie wiadomo, jaka wartość będzie faktycznie przypisana (co może prowadzić do niedeterminizmu). Sytuacje takie nazywają się **konfliktami zapisu**. Jest wiele sposobów poradzenia sobie z nimi. Można na przykład przyjąć zasadę, że z procesorów usiłujących dokonać zapisu w to samo miejsce dokonuje zapisu procesor o mniejszym numerze. Najprościej jednak założyć, że konflikty zapisu są w ogóle zabronione. Nie ma natomiast większego problemu (z punktu widzenia semantyki programów) z **konfliktami odczytu**: wiele procesorów może jednocześnie czytać wartość samej zmiennej. (Należy jednak pamiętać, że przy implementacji na bardziej realistycznych komputerach równoległych oba typy konfliktów stwarzają problemy o podobnej skali trudności).

W rozdziale tym za podstawowy model obliczeniowy przyjmujemy maszynę PRAM bez konfliktów zapisu.

Główym zagadnieniem w teorii algorytmów równoległych jest rozstrzygnięcie, czy problem obliczany w czasie  $Sekw(n)$  sekwencyjnie (na jednym procesorze) da się rozwiązać w czasie istotnie mniejszym przy użyciu wielu procesorów.

Problem ten można nazwać **problemem efektywności zrównolegania algorytmów sekwencyjnych**. Przez „istotnie mniejszy” rozumiemy zwykle czas wielomianowo-logarytmiczny  $T(n) = \log^k(Sekw(n))$ , a przez „wiele” procesorów – wielomianową (względem  $n$ ) liczbę procesorów  $P(n)$ . Całkowita liczba operacji w algorytmie równoległym wynosi  $T(n)*P(n)$ , a optymalność tego algorytmu wyraża się ilorazem  $Sekw(n)/(T(n)P(n))$ . Im większy jest ten iloraz, tym lepszy jest algorytm równoległy. **Algorytmy optymalne** można zdefiniować jako algorytmy ze stałym ilorazem optymalności. Są to algorytmy równolegle, których symulacja na jednym procesorze daje algorytm o złożoności równej asymptotycznie najszybszemu znanemu algorytmowi sekwencyjnemu dla danego problemu.

W tym rozdziale zajmiemy się najpierw zrównoleganiem możliwie najprostszych algorytmów sekwencyjnych. W tym celu za model obliczeń sekwencyjnych przyjmiemy (tak zwane) proste programy sekwencyjne. Są to programy iteracyjne (nierekurencyjne)

– ciągi instrukcji przypisania, bez instrukcji warunkowych. Okazuje się, że tego typu uproszczone obliczenia sekwencyjne są wystarczająco ogólne do badania problemu efektywności zrównolegania algorytmów sekwencyjnych.

Algorytmy sekwencyjne o bardziej skomplikowanej strukturze (rekursja, instrukcje warunkowe) są w ogólnym przypadku bardzo trudne do analizy możliwości ich zrównoleglenia. Są jednak pewne łatwe szczególne przypadki. Algorytmy sekwencyjne o strukturze rekurencyjnej dają się łatwo zrównoleglić, gdy głębokość rekursji jest logarytmiczna oraz wywołania rekurencyjne są niezależne (moga być wykonywane jednocześnie). Typowym przykładem jest obliczenie iloczynu  $n$  liczb. Przyjmijmy dla uproszczenia, że  $n$  jest potęgą dwójki. W algorytmie rekurencyjnym jest osobno obliczany iloczyn pierwszych  $n/2$  liczb i iloczyn ostatnich  $n/2$  liczb. Następnie w jednym kroku jest obliczany wynik końcowy z zastosowaniem jednej operacji mnożenia. Równoległa wersja różni się od sekwencyjnej jedynie tym, że wywołania rekurencyjne są wykonywane jednocześnie.

Zamiast mnożenia dwóch liczb możemy wziąć dowolną operację łączną i wykonywalną na RAM w czasie jednostkowym, a zatem obliczanie iloczynu, sumy, maksimum i minimum z  $n$  liczb możemy wykonać na maszynie PRAM w czasie  $O(\log n)$ , korzystając z  $n$  procesorów.

Pozostawiamy Ci jako ćwiczenie redukcję procesorów o czynnik  $\log n$  bez zmiany złożoności czasowej (asymptotycznie). Zamiast maszyny PRAM można tutaj przyjąć znacznie prostszy model obliczeniowy: drzewo procesorów (komunikacja tylko między procesorami poprzednik (ojciec) – następnik (syn) w drzewie). W rzeczywistości taka komunikacja odpowiada strukturze rekurencyjnej obliczania iloczynu  $n$  liczb.

Obliczanie iloczynu dwóch macierzy o rozmiarze  $n \times n$  sprawdza się do obliczenia  $n^2$  iloczynów skalarnych. Każdy taki iloczyn liczy się jak sumę  $n$  liczb, a więc wystarczy  $n$  procesorów do policzenia jednego iloczynu skalarnego w czasie  $O(\log n)$ . Ponieważ mamy  $n^2$  takich iloczynów do policzenia, możemy stwierdzić, że iloczyn dwóch macierzy o rozmiarze  $n \times n$  można obliczyć na maszynie PRAM w czasie  $O(\log n)$ , korzystając z  $n^3$  procesorów.

## 6.1.

### Równoległe obliczanie wyrażeń i prostych programów sekwencyjnych

Przykładem programu sekwencyjnego dającego się łatwo zrównoleglić jest instrukcja jednoczesnego (wielokrotnego) przypisania:

$$(x_1, x_2, \dots, x_n) := (val_1, val_2, \dots, val_n)$$

gdzie  $val_i$  są wyrażeniami zawierającymi jedynie stałe. W algorytmie sekwencyjnym jest wykonywanych kolejno  $n$  elementarnych instrukcji przypisania:

```
x1 := val1;
x2 := val2;
⋮
xn := valn;
```

Algorytm równoległy składa się z jednej instrukcji:

```
for each i ∈ {1, …, n} do in parallel xi := vali
```

Zajmiemy się teraz nieco bardziej skomplikowanymi programami. Programy te, jak również sposób ich zrównoleglenia, będą pewnym uogólnieniem instrukcji jednoczesnego przypisania.

Ciąg sekwencyjny instrukcji przypisania  $x_i := W_i$ , gdzie  $1 \leq i \leq n$ , nazwiemy **prostym programem sekwencyjnym**, jeżeli  $W_i$  jest wyrażeniem zawierającym jedynie zmienne o numerach mniejszych niż  $i$  (zmienne o wartościach wcześniej obliczonych). Założymy dla uproszczenia, że wyrażenia takie zawierają co najwyżej dwie zmienne i operacje +, -, \* lub /.

Dla prostego programu sekwencyjnego  $P$  przez  $graf(P)$  oznaczamy graf acykliczny, którego korzeniem (wierzchołkiem bez poprzedników) jest zmienna  $x_n$ , a następnikami danego węzła  $x_i$  są zmienne występujące w wyrażeniu  $W_i$ . Zbiorem węzłów  $V$  grafu są jedynie węzły osiągalne z korzenia (zmienne „biorące” faktyczny „udział” w obliczaniu wyniku). Zmienne odpowiadające tym węzłom (jak również te węzły) nazywamy **aktualnymi**.

Na początku zajmiemy się jedynie prostymi programami sekwencyjnymi, obliczającymi wartości wyrażeń arytmetycznych. Założymy (do odwołania), że każdy rozważany prosty program sekwencyjny  $P$  ma strukturę drzewiastą, tzn.  $graf(P)$  jest drzewem. Program taki oblicza wartość wyrażenia, przechodząc drzewo wyrażenia metodą „z dołu do góry”. Węzły drzewa odpowiadają zmiennym programu. Ponieważ zmienna  $x_n$  odpowiada korzeniowi drzewa wyrażenia, a jej wartość daje wartość całego wyrażenia, naszym celem jest obliczenie jedynie wartości  $x_n$ .

Działanie algorytmu przedstawimy na następującym przykładzie programu  $P$  obliczania wartości wyrażenia

$$W = 2(3(x1 + x2) + 2(2 * x3) + 2) + x4$$

z następującymi wartościami  $x1, x2, x3, x4$ :  $x1 = 1, x2 = 2, x3 = 3, x4 = 4$ . Program  $P$  obliczania  $W$  jest takim oto prostym programem sekwencyjnym:

```
program P
x1 := 1; x2 := 2; x3 := 3; x4 := 4; x5 := 2 * x3; x6 := x5 + 2;
x7 := x1 + x2; x8 := 3 * x7 + 2 * x6; x9 := 2 * x8 + x4
```

Algorytm równoleglego wykonywania  $P$  będzie podobny do równoleglego wykonywania programu wynikającego z instrukcji jednoczesnego przypisania. Będziemy starali się wykonać jednocześnie obliczenia wyrażeń  $W_i$  stojących po prawej stronie instrukcji przypisania. Jednakże nie zawsze (nie dla każdego  $i$ ) będzie to w pełni możliwe. Jeżeli wartości pewnych zmiennej występujących w  $W_i$  nie są policzone, to wydaje się, że nie można obliczyć wyrażenia  $W_i$ . Co to znaczy dokładnie, że wartość  $x_k$  jest policzona? Otóż po prawej stronie  $k$ -tego przypisania występuje stała. Gdybyśmy chcieli w jednym równoległym kroku algorytmu obliczać jedynie wyrażenia  $W_i$ , w których wszystkie zmienne są policzone, to w niektórych wypadkach taki algorytm równoległy byłby nie-wiele lepszy od sekwencyjnego, na przykład w sytuacji, kiedy drzewo wyrażenia byłoby bardzo „chude” i miało wysokość liniową.

Przyjmijmy, że będziemy obliczać wyrażenia  $W_i$  w inny sposób. Zmienną  $x_i$  nazywamy bezpieczną wtedy, kiedy wyrażenie  $W_i$  zawiera co najwyżej jedną zmienną. Nasz algorytm równoległy będzie polegać na „częściowym” obliczaniu wyrażeń  $W_i$ . Wszystkie zmienne bezpieczne, występujące w  $W_i$ , będą zastąpione przez odpowiadające im wyrażenia, a następnie otrzymane w ten sposób wyrażenie będzie uproszczone jak to tylko możliwe. Operację taką nazywamy redukcją wyrażenia  $W_i$ .

W naszym programie  $P$  zmiennymi bezpiecznymi są wszystkie zmienne oprócz  $x7, x8$  i  $x9$ . Jeżeli operację redukcji zastosujemy do wyrażenia odpowiadającego  $x8$ , to otrzymamy

$$x8 := 3 * x7 + 2 * x5 + 4$$

Teraz wyrażenie odpowiadające  $x8$  bierze się stąd, że w wyrażeniu  $3 * x7 + 2 * x6$  zmienną bezpieczną  $x6$  zastąpiliśmy wyrażeniem  $x5 + 2$ . Zauważmy, że zmienna  $x6$  jest „bezpieczna” w tym sensie, że liczba zmiennych w wyrażeniu dla  $x8$  się nie zwiększa. Rozmiar wyrażeń stojących po prawej stronie instrukcji przypisania jest ograniczony przez stałą. W każdym wyrażeniu występują co najwyżej dwie zmienne, a ta sama zmienna występuje co najwyżej raz w jednym wyrażeniu. Wartość danej zmiennej  $x_i$  zostanie w pełni obliczona, gdy wyrażenie  $W_i$  będzie stałą.

{Algorytm jednoczesnych podstawień; równoległe obliczanie wyrażeń zapisanych jako proste programy sekwencyjne}

begin

repeat

for each  $i ∈ {1, …, n}$  do in parallel

dokonaj redukcji wyrażenia  $W_i$

{zastąp wszystkie zmienne bezpieczne w  $W_i$  przez ich wyrażenia}

until  $x_n$  obliczone {wyrażenie  $W_n$  jest stałą}

end {algorytm jednoczesnych podstawień};

Przedstawimy teraz działanie tego algorytmu dla naszego przykładowego prostego programu sekwencyjnego  $P$ . Po pierwszym wykonaniu instrukcji iteracyjnej (iteracji) otrzymujemy program  $P1$ .

PL:  $x5 := 6; x7 := 3; x8 := 3 * x7 + 2 * x5 + 4; x9 := 2 * x8 + 4;$

Wypisujemy tylko wyrażenia dla zmiennych aktywnych (odpowiadające im węzły należą do grafu  $graf(PI)$ ).

Po drugim wykonaniu instrukcji iteracyjnej otrzymujemy:

P2:  $x8 := 25; x9 := 2 * x8 + 4;$

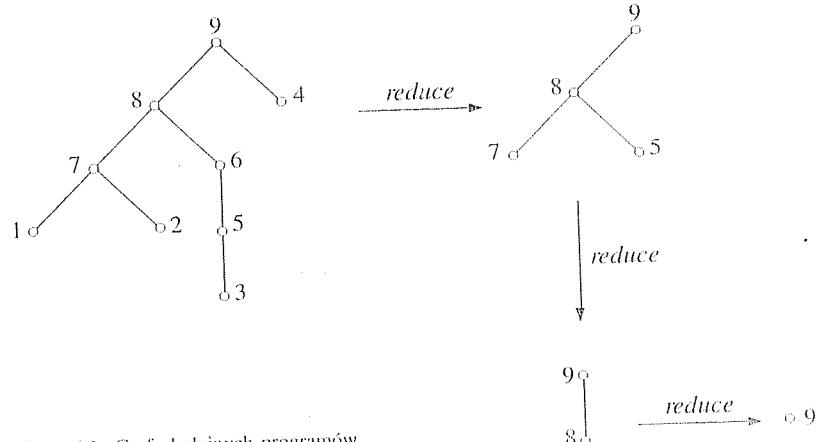
Po kolejnym wykonaniu mamy:

P3; x9 = 54

Wartość  $r9$  zostaje obliczona i kończy się działanie algorytmu.

W naszym przykładzie były potrzebne tylko trzy kroki na obliczenie  $x9$ . Kilka procesorów znacznie przyspieszyło obliczenia. Ciąg grafów kolejnych programów  $G0 = \text{graf}(P)$ ,  $G1 = \text{graf}(P1)$ ,  $G2 = \text{graf}(P2)$  oraz  $G3 = \text{graf}(P3)$  jest przedstawiony na rysunku 6.1.

Okazuje się, że rozmiar tych grafów sukcesywnie maleje; w ogólnym przypadku liczba iteracji jest logarytmiczna. Żeby to udowodnić, przejdziemy do języka teorii grafów (grafy, które tu występują, są w istocie drzewami).



Rys. 6.1. Grafy kolejnych programów

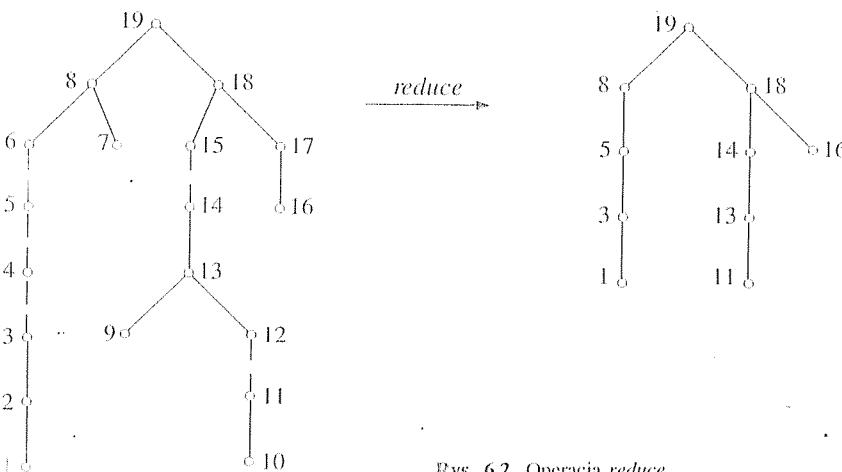
Przez  $reduce(T)$  będziemy oznaczać drzewo binarne  $T'$  powstałe z  $T$  w wyniku następującej operacji: każda krawędź prowadząca do liścia jest usunięta; każda krawędź prowadząca od węzła  $v$  do węzła  $w$  jest zastąpiona przez krawędź od  $v$  do  $nastepnik(w)$ , jeśli  $w$  ma tylko jeden następnik. Drzewo  $T'$  jest podgrafem otrzymanego grafu, zawierającym tylko węzły osiągalne z korzenia (połączone z korzeniem ścieżką).

Niech teraz  $T = \text{graf}(P)$  i niech  $P'$  będzie programem sekwencyjnym (ciągiem przypisów), otrzymanym z  $P$  za pomocą jednego wykonania instrukcji iteracyjnej algorytmu jednoczesnych podstawień. Mamy  $\text{reduce}(T) = \text{graf}(P')$ , ponieważ podstawieniu stałej na zmienną odpowiada usunięcie krawędzi prowadzącej do właściwego dla niej liścia, a wymianie krawędzi od  $v$  do  $w$  na krawędź od  $v$  do  $\text{następnik}(w)$  odpowiada zastąpienie  $x_k$  przez  $W_k$ , gdy  $W_k$  zawiera dokładnie jedną zmienną.

Niech  $|T|$  oznacza liczbę węzłów (rozmiar) drzewa  $T$ . Prawdziwa jest następująca nierówność:

$|reduce(T)| \leq 2/3 |T|$ , jeśli  $|T| > 1$

W dowodzie posłużymy się rysunkiem 6.2. Niektóre węzły drzewa  $T$  „znikają” w drzewie  $T' = \text{reduce}(T)$ , ponieważ nie są osiągalne z korzenia po usunięciu/wymianie pewnych krawędzi. Oszacujemy, jak wiele węzłów pozostaje w  $T'$ . Przez łańcuch rozumiemy maksymalną ścieżkę (w kierunku liści drzewa  $T$ ), zawierającą co najmniej dwa węzły, na której każdy węzeł ma w  $T$  dokładnie jeden następnik. (Liście nie są zatem elementami łańcuchów). Dodatkowo żądamy, żeby ostatni węzeł w takiej ścieżce nie należał do łańcucha wtedy, kiedy ścieżka ma nieparzystą liczbę węzłów, a następnikiem węzła  $v$  jest liść (zob. zaznaczone łańcuchy na rysunku 6.2). Podzielimy zbiór  $V$  wszystkich węzłów drzewa  $T$  na dwa podzbiory  $V_1$  i  $V_2$ , gdzie  $V_1$  składa się z węzłów zawartych



Rys. 6.2. Operacja *reduce*

w łańcuchach, a  $V_2$  z pozostałych węzłów. Wystarczy udowodnić, że z każdego ze zbiorów  $V_1$  i  $V_2$  pozostaje (po redukcji) co najwyżej  $2/3$  węzłów. Jest to dosyć oczywiste dla zbioru  $V_1$  (w rzeczywistości z każdego łańcucha „zniknie” co najmniej połowa węzłów, a więc w  $V_1$  zostanie jedynie co najwyżej połowa węzłów). Jeżeli chodzi o zbiór  $V_2$ , to zauważmy, że ma on co najwyżej  $3m$  węzłów, gdzie  $m$  jest liczbą liści drzewa  $T$ . Do zbioru  $V_2$  należą liście oraz co najwyżej  $m$  węzłów, których następcami są liście. Na pewno też należą przodkowie liści z dwoma następcami. Takich przodków może być co najwyżej  $m - 1$ . Jeżeli do  $V_2$  należą jednocześnie liście  $v$  i jego poprzednik, którego  $v$  jest jedynym następciem, to poprzednik liścia  $v$  jest usuwany. W przeciwnym razie jest usuwany sam liść  $v$ . Stąd wynika, że w  $V_2$  pozostało co najwyżej  $2/3|V_2|$  węzłów. W sumie pozostało w  $T'$  co najwyżej  $2/3$  węzłów początkowego zbioru  $V$  wszystkich węzłów. Kończy to dowód poprawności naszego oszacowania.

Jeśli  $T$  jest drzewem składającym się z trzech węzłów  $v_1, v_2, v_3$ , gdzie  $v_2$  jest następciem  $v_1$ , a  $v_3$  jest następciem  $v_2$ , to  $|\text{reduce}(T)| = 2/3|T|$ , a zatem nasze oszacowanie jest dokładne.

Z faktu, że  $|\text{reduce}(T)| \leq 2/3|T|$ , wynika, że liczba iteracji jest logarytmiczna. Algorytm obliczania równoleglego prostych programów sekwencyjnych o strukturze drzewiastej działa więc w czasie  $O(\log n)$  przy użyciu  $n$  procesorów równoleglego komputera PRAM. W ten sposób doszliśmy do istotnego wniosku, a mianowicie, że wyrażenia arytmetyczne (rozpisane jako ciąg  $n$  instrukcji przypisania) można obliczać w czasie  $O(\log n)$ , używając  $n$  procesorów.

Liczba procesorów można zmniejszyć do  $O(n/\log n)$  przy użyciu innego (choć o podobnej strukturze) algorytmu. Żeby się o tym przekonać, sięgnij do [GR].

Przedstawiony tu algorytm możemy również stosować wtedy, kiedy mamy do czynienia z operacjami w algebrze o nośniku (zbiorze wszystkich argumentów i wartości) rozmiaru  $O(1)$ . Rząd złożoności nie ulegnie zmianie. Trzeba jedynie zmodyfikować procedurę zastępowania zmiennych przez odpowiadające im wyrażenia. Zamiast wyrażeń możemy przechowywać tablice, na podstawie których możemy się dowiedzieć, jaką będzie wartość wyrażenia dla zadanych wartości zmiennych. Wartości te przebiegają tu cały nośnik algebry; ponieważ jest on rozmiaru  $O(1)$ , rozmiary tablic i koszt operacji na nich są ograniczone przez stałą.

Rozważmy na przykład problem obliczania najliczniejszego zbioru niezależnego  $Z$  w drzewie binarnym. Zbiór  $Z$  ma następującą własność: żadne dwa różne jego węzły nie mogą być połączone krawędzią. Naszymi operacjami są teraz operacje logiczne `and`, `not` i `or`. Wartościami zmiennych odpowiadających węzłom drzewa są wartości logiczne. Zmienne o wartości `true` zaliczamy do zbioru  $Z$ , a liściom przypisujemy wartość `true`. Jeżeli węzeł  $x_i$  ma następcy  $x_k$  i  $x_l$ , to piszemy  $x_i := \text{not}(x_k \text{ or } x_l)$ . Wykonujemy prosty program sekwencyjny i w ten sposób obliczamy również najliczniejszy zbiór niezależny  $Z$ . Poprawność powyższego algorytmu wynika z faktu, że wśród najliczniejszych zbiorów niezależnych istnieje zbiór zawierający wszystkie liście.

Innym przykładem jest obliczanie minimalnego (co do liczności) zbioru dominującego w drzewie binarnym. Zbiór  $D$  jest dominujący, jeżeli każdy węzeł drzewa należy do  $D$  lub sąsiaduje z elementem ze zbioru  $D$ . Teraz możliwymi wartościami węzłów są 0, 1 lub 2. Do zbioru  $D$  zaliczamy węzły o wartości 2, a liściom przypisujemy wartość 0. Pozostałym węzłom  $v$  przypisujemy: 0, gdy wartości następców  $v$  są równe 1; 1, gdy jeden z następców ma wartość 2, a drugi nie ma wartości 0; 2, w pozostałych przypadkach. Jeżeli po wykonaniu powyższych obliczeń wartość korzenia jest 0, to zamieniamy ją na 2. Prosta indukcja ze względu na wysokość drzewa dowodzi poprawności algorytmu. Wynika stąd, że zarówno problem obliczania maksymalnego zbioru niezależnego, jak i problem obliczania minimalnego zbioru dominującego można rozwiązać dla drzew w czasie  $O(\log n)$ , używając  $O(n)$  procesorów.

Przypuśćmy, że mamy danych  $n$  liczb  $a_1, a_2, \dots, a_n$  i chcemy obliczyć wszystkie sumy częściowe  $x_i = a_1 + \dots + a_i$ . Problem ten można łatwo sprowadzić do obliczania prostego programu sekwencyjnego:  $x_1 := a_1; x_2 := x_1 + a_2; \dots; x_n := x_{n-1} + a_n$ . Można zatem zastosować algorytm równoleglego obliczania wyrażeń i rozwiązać problem sum częściowych w czasie  $O(\log n)$  za pomocą  $n$  procesorów. Liczbę procesorów można łatwo zmniejszyć do  $O(n/\log n)$ . Operację  $+$  można zastąpić przez  $*$ ,  $\max$  lub  $\min$ . Można zatem obliczyć maksimum z każdych  $k$  początkowych elementów, dla  $k = 1..n$ , korzystając z algorytmu o takiej samej złożoności.

Problem obliczenia sum częściowych można rozwiązać również prostym algorymem rekurencyjnym. Obliczamy równolegle sumy częściowe dla ciągów  $(a_1, \dots, a_{\lfloor n/2 \rfloor})$  oraz  $(a_{\lfloor n/2 \rfloor + 1}, \dots, a_n)$ . Mając policzone te sumy i manipulując nimi, możemy w jednym kroku równolegle obliczyć sumy częściowe dla całego ciągu. Opracowanie pełnego schematu tego typu algorytmu pozostawimy Ci jako zadanie.

Zalożenie, że program sekwencyjny ma strukturę drzewiastą, jest istotne. Gdybyśmy na przykład chcieli obliczyć wartość  $n$ -tej liczby Fibonacciego  $x_n$ , stosując nasz algorytm do programu sekwencyjnego

$$x_1 := 1; x_2 := 1; x_3 := x_2 + x_1; \dots; x_n := x_{n-1} + x_{n-2};$$

to liczba iteracji nie będzie logarytmiczna. Można jednakże obliczać wartości liczb Fibonacciego innym algorymem w czasie logarytmicznym, używając  $n$  procesorów (przy założeniu, że dodawanie i mnożenie dużych liczb jest wykonywalne w czasie stałym). Taki algorytm otrzymamy wtedy, kiedy zamiast  $x_n$  będziemy od razu obliczać wektory  $z_n = (x_n, x_{n-1})$ . Wektor  $z_n$  powstaje z wektora  $z_{n-1}$  przez pomnożenie przez pewną macierz o rozmiarze  $2 \times 2$ . Stosujemy teraz taki algorytm jak w wypadku obliczania sum częściowych – z tą tylko różnicą, że liczby zastępujemy (małymi) macierzami i zamiast operacji  $+$  mamy mnożenie tych macierzy.

Rozważmy teraz bardziej ogólny przypadek, a mianowicie, gdy graf programu niekoniecznie jest drzewem. Przyjmijmy jednak pewne ograniczenia. Jedynymi operacjami niech

będą  $+ i *$ . Jeżeli założymy, że liście grafu obliczenia odpowiadają zmiennym wejściowym, to prosty program sekwencyjny  $P$  obliczy wartość pewnego wielomianu, z reguły wielu zmiennych. (Zmienne wejściowe są tymi zmiennymi  $x_i$ , dla których  $W_i$  zawiera same stałe). Przez  $\deg(P)$  oznaczmy stopień tego wielomianu i nazwijmy go stopniem programu  $P$ . Rozważmy na przykład taki oto program  $P$ :

```

x1 := c1; x2 := c2; x3 := c3; x4 := x1 * x2; x5 := x2 + x4;
x6 := x3 * x5; x7 := x4 + x6;

```

Zmiennymi wejściowymi są  $x_1, x_2, x_3$ . Program  $P$  oblicza wartość wielomianu  $x_1 * x_2 + x_3 * x_2 + x_1 * x_2 * x_3$  dla  $x_1 = c_1, x_2 = c_2$  i  $x_3 = c_3$ , a zatem  $\deg(P) = 3$ .

Inna definicja stopnia programu może być podana w terminach związanych z grafem obliczeń. Stopień każdego liścia jest równy jeden. Stopień węzła typu  $+$  jest równy maximum ze stopni jego następców, a stopień węzła typu  $*$  jest równy sumie stopni lewego i prawego następców. (Dokładną definicję grafu programu sekwencyjnego podamy później). Stopień całego programu jest równy stopniowi korzenia grafu obliczeń.

Do wykonania programu  $P$  stosujemy ten sam algorytm jednoczesnych podstawień. Zmieniamy jedynie definicję zmiennej bezpiecznej. Mówimy, że zmienna  $x_i$  jest bezpieczna, jeżeli wyrażenie  $W_i$  jest stopnia co najwyżej jeden. Przez redukcję wyrażenia rozumiemy doprowadzenie wyrażenia do postaci sumy jednomianów stopnia jeden lub iloczynu dwóch takich sum plus pewna stała. Założymy, że początkowy program zawiera jedynie wyrażenia tej postaci.

Przedstawimy najpierw działanie algorytmu dla prostego programu sekwencyjnego stopnia jeden. Weźmy program obliczania wartości piątej liczby Fibonacciego:

```
x1 := 1; x2 := 1; x3 := x2 + x1; x4 := x3 + x2; x5 := x4 + x3;
```

Po pierwszym wykonaniu instrukcji iteracji otrzymujemy:

```
x1 := 1; x2 := 1; x3 := 2; x4 := x2 + x1 + 1; x5 := x3 + 2 * x2 + x1;
```

a po następnej (ostatniej):

```
x1 := 1; x2 := 1; x3 := 2; x4 := 3; x5 := 5;
```

W powyższym przykładzie mamy dwa równolegle kroki, podczas gdy wykonując kolejno instrukcje przypisania za pomocą jednego procesora, potrzebujemy trzech kroków (obliczenie  $x3$ ,  $x4$ ,  $x5$ ). Przykład ten jest jednak trochę mylący; sugeruję bowiem, że jedną iterację łatwo wykonać w czasie stałym, mając po jednym procesorze dla każdej zmiennej. Wykażemy, że w ogólnym przypadku wykonanie jednej iteracji jest bardziej skomplikowane. Sprowadza się ono do mnożenia macierzy (operacja ta jest łatwo wyko-

nalna w czasie  $O(\log n)$  za pomocą  $n^3$  procesorów). W tym celu działanie algorytmu przedstawimy jeszcze raz na następującym, bardziej skomplikowanym przykładzie:

```

P: x1 := 1; x2 := 2 * x1 + 2; x3 := 3 * x1 + 2 * x2 + 3;
  x4 := x1 + 3 * x2 + 2 * x3 + 2;
  x5 := (x1 + 2 * x2 + x4) * (2 * x1 + 3);

```

Początkowo zmiennymi bezpiecznymi są wszystkie zmienne oprócz  $x_5$ . Po pierwszej iteracji otrzymujemy program  $PI$ :

```

x1 := 1; x2 := 4; x3 := 4 * x1 + 10; x4 := 12 * x1 + 4 * x2 + 15;
x5 := 25 * x1 + 15 * x2 + 10 * x3 + 35;

```

Wyrażenie na  $x5$  powstało w ten sposób, że podstawiłyśmy wyrażenie odpowiadające  $x1$ ,  $x2$ ,  $x4$  w dwóch wyrażeniach:  $(x1 + 2 * x2 + x4)$  i  $(2 * x1 + 3)$ . Ponieważ po podstawieniu i redukcji drugie z wyrażeń zmieniło się w stałą 5, to pierwsze wyrażenie (po zrobieniu w nim podstawień) pomnożyliśmy przez 5.

Кluczowym problemem przy zrównoleganiu takich obliczeń jest więc efektywność transformacji jednego wyrażenia przez równolegle wykonanie w nim wszystkich podstawień. Sprowadzimy ten problem do mnożenia wektora przez macierz. Reprezentacją wektorową wyrażenia  $a_1x_1 + a_2x_2 + \dots + a_nx_n + c$  jest wektor  $[a_1, a_2, \dots, a_n, c]$ . Reprezentacją wyrażenia  $x1 + 3 * x2 + 2 * x3 + 2$  odpowiadającego  $x4$  w początkowym programie  $P$  jest wektor  $[1, 3, 2, 0, 0, 2]$ .

Rozważmy macierz  $A$ , której  $i$ -ty wiersz jest reprezentacją wektorową wyrażenia  $W_i$ , gdy zmienna  $x_i$  jest bezpieczna; w przeciwnym wypadku  $i$ -ty wiersz jest  $i$ -tym wersorem (na  $i$ -tym miejscu jest 1, a na pozostałych zera). Jako ostatni wiersz dodajemy wersor z 1 na końcu. Dla naszego poczatkowego programu macierz  $A$  jest nastepujaca:

```

0 0 0 0 0 1
2 0 0 0 0 2
3 2 0 0 0 3
1 3 2 0 0 2
0 0 0 0 1 0
0 0 0 0 0 1

```

Teraz transformacja wyrażenia  $x_1 + 3 * x_2 + 2 * x_3 + 2$  polega na pomnożeniu reprezentacji wektorowej tego wyrażenia przez  $A$ :  $[1, 3, 2, 0, 0, 2] * A = [12, 4, 0, 0, 0, 15]$ . Otrzymany wektor jest reprezentacją wyrażenia  $12 * x_1 + 4 * x_2 + 15$ . Jest to więc zgodne z tym, co poprzednio dostaliśmy w programie  $PI$ , uzyskanym za pomocą jednej iteracji algorytmu z programu  $P$ . Po każdym wykonaniu instrukcji iteracyjnej macierz  $A$  może się zmienić, gdyż pewne zmienne  $x_i$  stają się bezpieczne, a następnie  $i$ -ty wiersz macierzy się zmienia (z  $i$ -tego wersora na reprezentację wektorową  $W$ ).

Jako zadanie pozostawiamy Ci opracowanie algorytmu mnożenia wektora przez macierz w czasie  $O(\log n)$  przy użyciu  $n^2$  procesorów. W jednej iteracji mamy  $O(n)$  takich mnożeń.

Algorytm jednoczesnych podstawień (przy ostatniej definicji zmiennych bezpiecznych) działa w czasie  $(\log n) \cdot$  (liczba iteracji) i wykorzystuje  $O(n^3)$  procesorów.

Założyliśmy, że w każdym z wyrażeń  $W_i$  jest co najwyżej jedna operacja  $*$  (mnożenie wielomianów) i że jeśli operacja  $*$  występuje, to  $W_i$  ma postać:

$$(a_1x_{i_1} + a_2x_{i_2} + \dots + a_kx_{i_k} + a_{k+1}) * (b_1x_{j_1} + b_2x_{j_2} + \dots + b_lx_{j_l} + b_{l+1}) + c$$

gdzie  $a_i, b_j, c$  są stałymi. Udowodnimy teraz, że jeśli stopień programu  $P$  jest wielomianowy, to liczba iteracji jest logarytmiczna, a dokładniej – nie przekracza  $\lceil \log_{3/2}(nd) \rceil$ , gdzie  $d$  jest stopniem  $P$ , a  $n$  liczbą zmiennych.

Podobnie jak w analizie algorytmu jednoczesnych podstawień dla wyrażeń, tak i tutaj przejdziemy na język teoriografowy. Niech  $graf(P)$  będzie grafem programu i niech węzły odpowiadają zmiennym. Mamy dwa typy węzłów wewnętrznych: odpowiadające zmiennym bezpiecznym (węzły typu  $+$ ) i odpowiadające pozostałym zmiennym (węzły typu  $*$ ). Dla każdego węzła (podobnie jak w drzewie) określmy jego następki. Dla węzła  $x_i$  typu  $+$  następcami są wszystkie zmienne występujące w  $W_i$ ; dla węzła typu  $*$  rozróżniamy dwa typy następców: lewy i prawy. Jeżeli  $W_i$  ma postać

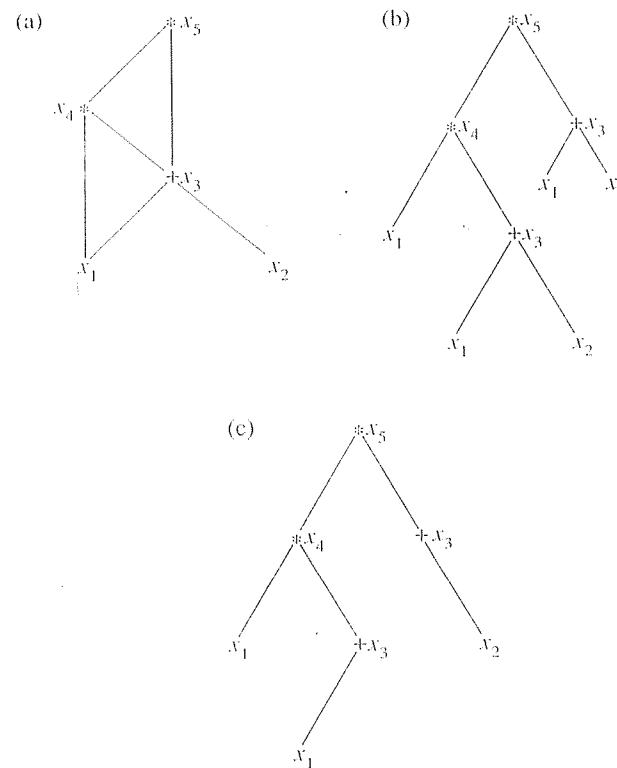
$$(a_1x_{i_1} + a_2x_{i_2} + \dots + a_kx_{i_k} + a_{k+1}) * (b_1x_{j_1} + b_2x_{j_2} + \dots + b_lx_{j_l} + b_{l+1}) + c$$

gdzie  $a_i, b_j, c$  są stałymi, to lewymi następcami są węzły odpowiadające zmiennym  $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ , a prawymi  $x_{j_1}, x_{j_2}, \dots, x_{j_l}$ . Zauważmy, że pewien węzeł może być jednocześnie lewym i prawym następcikiem tego samego węzła.

Jedna iteracja w algorytmie odpowiada transformacji *reduce* grafu programu, polegającej na jednoczesnym wykonaniu opisanych poniżej operacji (a) i (b), a następnie wykonaniu operacji (c):

- usunięcie krawędzi prowadzących do liści;
- każda krawędź prowadząca od węzła  $v$  do węzła typu  $+$  w jest usunięta i są tworzone krawędzie prowadzące od  $v$  do każdego następcnika  $w$ ; jeżeli  $v$  jest węzłem typu  $*$ , to nowe krawędzie są tego samego typu co krawędź, którą zastępują (w sensie lewy i prawy następnik);
- jeśli po wykonaniu wszystkich operacji (a) i (b) węzeł typu  $*$  w ma tylko lewe lub tylko prawe następciki (z powodu usunięcia pewnych krawędzi), to w staje się węzłem typu  $+$ .

Graf  $reduce(G)$  dla grafu  $G$  z rysunku 6.3a jest przedstawiony na rysunku 6.4a. Zauważmy, że  $x_1$  jest zarówno lewym, jak i prawym następcikiem  $x_5$ .



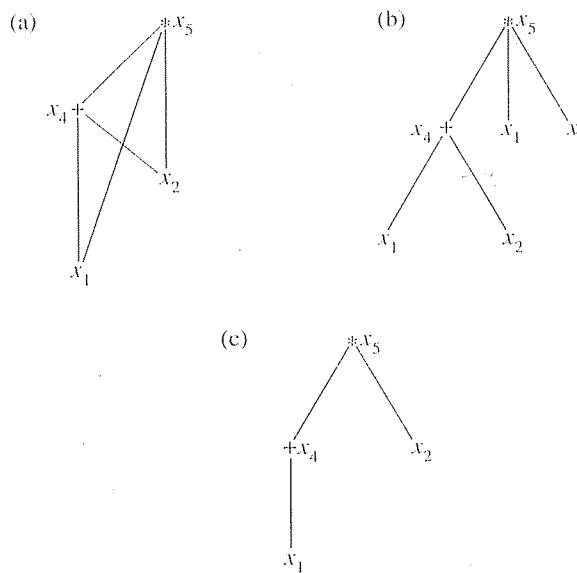
Rys. 6.3. (a) Graf  $G$ ; (b) drzewo  $tree(G)$ ; (c) drzewo ze zbioru  $sub(T)$

Wprowadzimy teraz operację „udrzewiania” grafu  $G$ , zapisywaną jako  $tree(G)$ . Z definicji drzewo  $tree(G)$  ma węzły, które odpowiadają zmiennym, ale tej samej zmiennej może odpowiadać kilka węzłów. Niech  $paths(G)$  będzie zbiorem ciągów zmiennych na ścieżkach od korzenia do liści grafu  $G$ . Drzewo  $tree(G)$  jest takim drzewem  $T$ , że  $paths(G) = paths(T)$ . Inaczej mówiąc,  $T$  jest drzewem wyrażenia  $W$ , które powstałoby, gdyby program sekwencyjny  $P$  zapisać jako jedno duże wyrażenie.

Zauważmy, że jeżeli  $G$  jest podanym wcześniej grafem programu liczącego liczby Fibonacciego, to  $tree(G)$  ma wykładniczą liczbę węzłów, chociaż stopień programu jest bardzo mały (równy 1).

Niech  $T$  będzie drzewem odpowiadającym grafowi programu  $P$ . Przez  $sub(T)$  oznaczmy zbiór wszystkich poddrzew  $T'$  spełniających warunki:

- korzeń  $T'$  jest taki sam jak korzeń  $T$ ;
- jeśli węzeł wewnętrzny  $v$  drzewa  $T'$  jest typu  $+$ , to dokładnie jeden z jego następców w  $T$  należy do  $T'$ ;

Rys. 6.4. (a) Graf  $reduceI(G)$ ; (b)  $tree(reduceI(G))$ ; (c) drzewo ze zbioru  $sub(tree(reduceI(G)))$ 

(c) jeśli  $v$  jest węzłem typu  $*$  z  $T'$ , to do  $T'$  należą tylko jeden lewy i jeden prawy następnik  $v$  w  $T$ .

Niech  $rank(G)$  będzie maksymalnym rozmiarem (liczbą węzłów) drzewa ze zbioru  $sub(tree(G))$ . Łatwo zauważyc, że stopień programu  $P$  jest równy maksymalnej liczbie liści w drzewie ze zbioru  $sub(tree(G))$ , gdzie  $G = graf(P)$ . Jednocześnie długość każdej ścieżki w drzewie nie przekracza rozmiaru  $n$  grafu  $G$  (liczby zmiennych), a więc  $rank(G) \leq nd$ , gdzie  $d$  jest stopniem  $P$ .

Wystarczy teraz udowodnić, że jeśli  $|G| > 1$ , to  $rank(reduceI(G)) \leq 2/3(rank(G))$ .

Dowód przeprowadzimy, korzystając z redukcji drzew za pomocą operacji  $reduce$ . Niech  $reduceI(G) = G1$  i niech  $T1 = tree(G1)$  oraz  $T = tree(G)$ . Dowód opiera się na następującej strukturalnej własności operacji  $reduce$  i  $reduceI$ : dla każdego drzewa  $T1'$  ze zbioru  $sub(T1)$  istnieje drzewo  $T'$  w zbiorze  $sub(T)$  takie, że  $T1' = reduce(T')$ .

Możemy ten fakt zapisać bardziej formalnie:

$$sub(tree(reduceI(G))) \subseteq reduce(sub(tree(G)))$$

(zauważmy jednak, że może się zdarzyć, iż dla pewnego  $T'$  z  $sub(T)$  drzewo  $reduce(T')$  nie będzie elementem  $sub(T1)$ , a więc odwrotna inkluzyja nie będzie zachodzić).

Powyższa własność jest intuicyjnie oczywista; trzeba tylko przypomnieć sobie definicje operacji  $sub$ ,  $tree$ ,  $reduce$  i  $reduceI$ . W tym momencie nasze oznaczenie:

### 6.1. Równoległe obliczanie wyrażeń i prostych programów

$$rank(reduceI(G)) \leq 2/3 \ rank(G)$$

wynika bezpośrednio z faktu, że

$$|reduce(T')| \leq 2/3 \ |T'|$$

co udowodniliśmy wcześniej, analizując równoległe obliczanie wyrażeń. Kończy to dowód, że liczba iteracji w algorytmie jednocześnie podstawień z drugą definicją zmiennych bezpiecznych nie przekracza  $\lceil \log_{M2}(nd) \rceil$ .

Omówimy teraz zastosowanie algorytmu jednocześnie podstawień do programowania dynamicznego. Programowanie dynamiczne jest ogólną metodą rozwiązywania zadań optymalizacyjnych. Metoda ta polega na rozbijaniu danego zadania na podzadania, a jej istotą jest to, że rozwiązania podzadań również mają być optymalne, co daje możliwość zapisu rekurencyjnego. Korzystając z takiego zapisu, obliczamy tablicę optymalnych rozwiązań dla wszystkich podzadań; dany element tablicy wyraża się prostym wzorem zależnym od innych elementów. Istotne jest również to, że liczba podzadań jest wielomianowa. Taki opis programowania dynamicznego nie jest zbyt formalny; dokładniejszy opis przedstawimy dla trzech wybranych problemów.

#### PROBLEM 1. Obliczanie optymalnej kolejności mnożenia macierzy

Przypuśćmy, że mamy dany ciąg rozmiarów macierzy  $M_1, M_2, \dots, M_n$ . Macierz  $i$ -ta ma  $r_{i-1}$  wierszy i  $r_i$  kolumn. Zakładamy, że koszt mnożenia macierzy o rozmiarach  $p \times q$  i  $q \times r$  wynosi  $pqr$ . Chcemy wyznaczyć minimalny koszt obliczenia macierzy  $M = M_1 \times M_2 \times \dots \times M_n$ . Koszt ten zależy od kolejności mnożenia.

Niech  $m_{i,j}$  będzie minimalnym kosztem obliczenia macierzy  $M_{i,j} = M_{i+1} \times M_{i+2} \times \dots \times M_j$ . Przyjmujemy  $m_{i,i+1} = 0$ . Otrzymujemy system równań:

$$\begin{cases} m_{i,i+1} = 0 \text{ dla } i = 0, \dots, n-1 \\ m_{i,j} = \min\{m_{i,k} + m_{k,j} + r_i r_k r_j : i < k < j\} \end{cases}$$

Minimalnym kosztem obliczenia macierzy  $M$  jest  $m_{0,n}$ .

Z powyższego systemu równań wynika sekwencyjny algorytm obliczania tego kosztu w czasie  $O(n^3)$ . Wystarczy pomocnicze wartości  $m_{i,j}$  umieszczać w tablicy, którą należy obliczyć po przekątnych (z dolu do góry) lub kolejno po kolumnach (kolumny od lewej do prawej, w każdej kolumnie z dolu do góry).

Zastanówmy się teraz, jaką jest złożoność obliczeniowa  $m_{0,n}$  w modelu obliczeń równoległych. W tym celu zapiszmy system równań w postaci prostego programu sekwencyjnego. Wystarczy jedynie rozpisać równanie

$$m_{i,j} = \min\{m_{i,k} + m_{k,j} + r_i r_k r_j : i < k < j\}$$

w postaci takiego programu. Można to zrobić, wprowadzając nowe zmienne  $m_{i, j, k}$  dla  $i < k < j$ :

$$\begin{aligned} m_{i, j, i+1} &:= m_{i, i+1} + m_{i+1, j} + r_i r_{i+1} r_j \\ m_{i, j, i+2} &:= \min(m_{i, j, i+1}, m_{i, i+2} + m_{i+2, j} + r_i r_{i+2} r_j); \\ &\vdots \\ m_{i, j, j-1} &:= \min(m_{i, j, j-2}, m_{i, j-1} + m_{j-1, j} + r_i r_{j-1} r_j); \\ m_{i, j} &:= m_{i, j, j-1}; \end{aligned}$$

Wartości  $r_i r_k r_j$  można obliczyć równolegle wcześniej i przyjąć, że są one stałymi. W ten sposób otrzymujemy prosty program sekwencyjny. Przyjmijmy, że wcześniej operacji  $+$  odpowiada teraz operacja  $\min$ , a operacji  $*$  – operacja  $+$ . Łatwo możemy wykazać, że stopień programu jest liniowy ( $\leq n$ ). Możemy zastosować algorytm jednocześnie podstawień (z operacjami  $+$  i  $*$  zastąpionymi odpowiednio przez  $\min$  i  $+$ ). Tak jak poprzednio, korzystamy z reprezentacji wektorowej wyrażeń.

Długość naszego programu sekwencyjnego jest  $O(n^3)$ , a zatem otrzymujemy równoległy algorytm obliczania minimalnego kosztu mnożenia ciągu macierzy na maszynie PRAM w czasie  $O(\log^2 n)$  przy użyciu  $O(n^9)$  procesorów.

Podamy jeszcze dwa analogiczne problemy związane z programowaniem dynamicznym.

#### PROBLEM 2. Obliczanie minimalnego kosztu drzewa poszukiwań binarnych (BST)

Mamy danych  $n$  kluczy  $K_1, K_2, \dots, K_n$  w porządku rosnącym. Niech  $p_i$  będzie częstością dostępu do elementu  $K_i$ . Chcemy umieścić nasze elementy w liściach drzewa binarnego tak, żeby średni koszt (długość ścieżki od korzenia) do danych elementów był minimalny. Podobnie jak poprzednio, możemy zdefiniować  $m_{i,j}$  jako minimalny koszt BST dla ciągu elementów  $K_{i+1}, \dots, K_j$  podając odpowiedni system równań, a następnie wypisać prosty program sekwencyjny obliczania  $m_{0,n}$ . Tak zdefiniowany problem optymalnego BST możemy zatem rozwiązać równolegle w czasie  $O(\log^2 n)$ , używając  $O(n^9)$  procesorów.

#### PROBLEM 3. Minimalna triangulacja wielokąta wypukłego

Dany jest ciąg wierzchołków  $v_0, v_1, \dots, v_n$  (w kolejności zgodnej z ruchem wskazówek zegara) wielokąta wypukłego na płaszczyźnie. Należy tak wybrać  $n-2$  cięciw kątowych dane punkty, aby podzieliły wielokąt na trójkąty i aby koszt takiej triangulacji był minimalny. Przez koszt rozumiemy sumę długości wszystkich cięciw.

Znowu możemy zdefiniować wielkości  $m_{i,j}$  jako koszt triangulacji wielokąta  $v_i, v_{i+1}, \dots, v_j$  i wypisać – podobnie jak poprzednio – prosty program sekwencyjny. Złożoność tego problemu jest zatem tego samego rzędu co poprzednich dwóch problemów.

Rozważaliśmy jedynie obliczenie minimalnego kosztu. Celowe jest jednak obliczenie danych do realizacji takiego minimum (drzewo reprezentujące kolejność mnożenia macierzy, optymalne BST lub też, w ostatnim problemie, minimalny zbiór cięciw). Zanalizowanie sekwencyjnej i równoległawej złożoności tak rozszerzonego problemu pozostawiamy Ci jako zadanie. W algorytmie należy korzystać z uprzednio wyliczonych wartości  $m_{i,j}$ .

## 6.2. Sortowanie równolegle

Pokażemy teraz, w jaki sposób można szybko sortować ciąg  $n$  liczb, używając tylko  $n$  procesorów. Przez „szybko” rozumiemy tutaj – w czasie  $O(\log^2 n)$ .

Zacznijmy od stwierdzenia, że łatwo można posortować  $n$  liczb w czasie  $\log n$  za pomocą  $n^2$  procesorów. Mianowicie przydzielimy każdemu elementowi  $n$  procesorów, które w czasie  $\log n$  obliczają liczbę elementów mniejszych od tego elementu (bez straty ogólności możemy założyć, że sortowane elementy są parami różne). Po obliczeniu w ten sposób pozycji każdego elementu w posortowanym ciągu wynikowym wystarczy w jednym równoległym kroku wstawić każdy element na jego miejsce wynikowe.

Redukcja liczby procesorów do  $n^2/\log n$  jest trywialna, natomiast redukcja do liniowej liczby procesorów (przy zachowaniu czasu  $O(\log n)$ ) jest zadaniem trudnym. Zadowolimy się więc dalej czasem  $T(n) = \log^2 n$ .

Zacznijmy od algorytmu, w którym jest używana liniowa liczba procesorów i który działa w czasie  $\log^2 n$  na maszynie PRAM (później rozważymy znacznieuboższy, chociaż dosyć tradycyjny, model dla problemu sortowania, a mianowicie sieci sortujące). Algorytm ten jest równoległą wersją algorytmu sortowania przez scalanie (mergesort). Wystarczy jedynie umieć skalać dwa posortowane ciągi długości  $n$  w czasie  $\log n$  przy użyciu  $n$  procesorów. Projekt algorytmu mającego takie parametry złożonościowe nie nastręcza większych trudności na maszynie PRAM. W celu posortowania ciągów  $x$  i  $y$  wystarczy przypisać procesor każdemu elementowi ciągu  $x$ . Procesor ten znajdzie, metodą wyszukiwania binarnego, miejsce  $pos(el)$  odpowiadające danemu elementowi  $el$  w ciągu  $y$  – przy założeniu, że tylko ten element ma być wstawiony do ciągu  $y$ . Rzeczywiście miejsce elementu  $el$  będzie równe  $pos(el) + rank(el)$ , gdzie  $rank(el)$  jest liczbą elementów poprzedzających  $el$  na liście  $x$ . Szczegółowy zapis algorytmu pozostawiamy Tobie, Drogie Czytelniku.

Podobnie możemy obliczyć miejsce każdego elementu ciągu  $y$  w posortowanym ciągu wynikowym. W ten sposób udowodniliśmy, że scalenie dwóch posortowanych ciągów o łącznej długości  $n$  można wykonać w czasie  $\log n$  przy użyciu  $n$  procesorów.

Jeśli korzysta się z równoległawej wersji algorytmu mergesort, to sortowanie  $n$  elementów jest wykonalne w czasie  $\log^2 n$  przy użyciu  $n$  procesorów.

Jednym z najważniejszych wyników dotyczących sortowania jest równoległa implementacja algorytmu mergesort w czasie  $O(\log n)$ . W implementacji tej operacje scalania ciągów rozmiaru liniowego są wykonywane w czasie stałym dzięki pomocniczym strukturom danych. Algorytm taki, o nazwie parallel mergesort, został podany przez R. Cole'a [GR].

Tradycyjnym modelem obliczeń równoległych dla algorytmów sortowania są sieci sortujące. Sieć sortująca jest algorymem sortowania, w którym jedynym sposobem zdobywania informacji o danych wejściowych są porównania dwóch elementów, a elementarną operacją jest zamiana dwóch elementów: *compare-exchange*. Jeśli zastosujemy tę operację względem pary elementów  $(x, y)$ , to otrzymamy parę  $(\min(x, y), \max(x, y))$ . Inaczej mówiąc, operacja *compare-exchange* jest elementarną operacją sortowania ciągów długości 2. Najważniejszą własnością, jakiej wymaga się od sieci sortujących, jest pewna „sztywność” obliczeń. Dla ustalonego  $n$  (liczby elementów do posortowania) ciąg wykonanych porównań i operacji *compare-exchange* jest „sztywny”, to znaczy nie zależy od danych wejściowych (jest taki sam dla wszystkich ciągów wejściowych długości  $n$ ). Inaczej (bardziej formalnie) mówiąc, sieć sortująca jest przyporządkowaniem  $n \rightarrow S_n$ , gdzie  $S_n$  jest ciągiem operacji *compare-exchange* sortowania wszystkich ciągów długości  $n$ .

Podobnie można zdefiniować sieć scalającą (zamiast sortowania mamy teraz ciągi operacji *compare-exchange* scalania dwóch danych posortowanych ciągów).

W jednym momencie możemy wykonać pewną liczbę kolejnych, niezależnych operacji *compare-exchange* z ciągu  $S_n$ . Niezależność polega na tym, że pary elementów, biorące udział w dwóch różnych operacjach *compare-exchange*, są rozłączne. Liczba procesorów jest zatem liniowa (liniowa liczba par rozłącznych). Podciąg kolejnych niezależnych operacji nazywamy fazą. Czas równoległy odpowiada minimalnej liczbie faz, na jakie można podzielić  $S_n$ .

Opiszemy teraz dwie sieci sortujące, oparte na równoległym zaimplementowaniu algorytmu mergesort. Wystarczy jedynie opisać sieci scalające. Przedstawimy dwie takie sieci: scalającą metodą odd-even i scalającą metodą bitoniczną.

Niech  $odd(L)$  będzie podciągiem ciągu  $L$  składającym się z elementów o indeksach nieparzystych, a  $even(L)$  podciągiem składającym się z pozostałych elementów. Dla dwóch ciągów  $L1 = (a_1, \dots, a_n)$ ,  $L2 = (b_1, \dots, b_n)$  definiujemy:

$$interleave(L1, L2) = (a_1, b_1, a_2, b_2, \dots, a_n, b_n)$$

Zdefiniujmy ponadto operację  $odd-even(L)$ . Operacja ta polega na jednoczesnym zastosowaniu operacji *compare-exchange* do każdej pary elementów  $x_i, x_{i+1}$  ciągu  $L$  dla parzystych indeksów  $i$ . Niech  $L1 \& L2$  będzie operacją dopisania (konkatenacji) listy  $L2$  do  $L1$ .

```
function odd-even-merge(L1, L2);
{L1, L2 są posortowanymi ciągami długości n, gdzie n jest potęga
dwójką}
begin
  if n = 1 then zastosuj jedną operację compare-exchange else
  begin
    do in parallel
    begin
      Lodd := odd-even-merge(odd(L1), odd(L2));
      Leven := odd-even-merge(even(L1), even(L2));
    end;
    L := interleave(Lodd, Leven);
    odd-even-merge := odd-even(L)
  end
end;
```

Przejdziemy teraz do opisu algorytmu bitonic merge. Podstawową operacją będzie tutaj operacja przepłotu.

```
procedure przeplot(L);
begin
  L1 := ciąg pierwszych n/2 elementów ciągu L;
  L2 := ciąg ostatnich n/2 elementów ciągu L;
  L := interleave(L1, L2)
end;
```

Zakładamy, że mamy dwa posortowane ciągi  $L1$  i  $L2$ . Tworzymy ciąg  $L = L1 \& L2^R$ , gdzie  $L2^R$  jest ciągiem  $L2$  z odwróconą kolejnością elementów. Ciąg  $L$  jest tzw. ciągiem bitonicznym; jego elementy najpierw rosną, a następnie maleją. Scalanie bitoniczne polega na wielokrotnym stosowaniu operacji przepłotu razem z operacjami *compare-exchange* na sąsiednich elementach ciągu.

```
procedure bitonic-merge(L1, L2);
{L1 i L2 są posortowanymi ciągami długości n, gdzie n jest potęga
dwójką}
begin
  L = L1 & L2R; {L = (x1, ..., x2n)}
  repeat log n + 1 times
  begin
    przeplot(L);
    for each i do in parallel
      if odd(i) then compare-exchange(xi, xi+1);
  end
end;
```

Złożoność obu algorytmów scalania jest oczywista. Mniej oczywista jest poprawność tych algorytmów. Przy dowodzie poprawności bardzo pomocna jest tzw. *zasada zero-jedynkowa*. Oto ona: sieć sortująca (scalająca) jest poprawna dla wszystkich ciągów wejściowych wtedy i tylko wtedy, gdy jest poprawna dla ciągów zero-jedynkowych (składających się jedynie z zer i jedynek).

Wystarczy jedynie udowodnić, że jeżeli sieć nie sortuje pewnego ciągu  $L$ , to również nie sortuje pewnego ciągu zero-jedynkowego  $L'$ . Przypuśćmy, że  $a$  i  $b$ , gdzie  $a < b$ , są dwoma elementami  $L$ , które po posortowaniu są „złe” położone względem siebie ( $b$  znajduje się wcześniej niż  $a$ ). Ciąg  $L'$  otrzymujemy z ciągu  $L$  w ten sposób, że każdy element  $x$  zastępujemy przez  $f(x)$ , gdzie

$$f(x) = \begin{cases} 0 & x < b \\ 1 & \text{w przeciwnym razie} \end{cases}$$

Łatwo zauważyc, że za pomocą tego algorytmu nie zostanie również posortowany ciąg  $L'$ . W rzeczywistości, jeśli wynikiem działania operacji *compare-exchange* względem pary  $x, y$  jest para  $x', y'$ , to wynikiem działania *compare-exchange* względem pary  $f(x), f(y)$  jest para  $f(x'), f(y')$ . Kończy to dowód zasady zero-jedynkowej.

Stosując zasadę zero-jedynkową, można pokazać, że scalanie metodą bitoniczną oraz metodą odd-even jest poprawne. Wystarczy jedynie zanalizować działanie algorytmów scalania dla ciągów postaci  $0^k 1^n$  (posortowanych ciągów zero-jedynkowych). Pozostawiamy Ci taką analizę jako ćwiczenie.

## Zadania

- 6.1. Skonstruj algorytm obliczania sumy i iloczynu pierwszych  $k$  liczb spośród danych  $n$  liczb, dla każdego  $k = 1..n$ , na maszynie PRAM w czasie  $O(\log n)$ , korzystając z  $O(n/\log n)$  procesorów.
- 6.2. Skonstruj algorytm rozwiązywania tego samego problemu (z tym samym rzędem złożoności) na binarnym drzewie procesorów. Komunikacja odbywa się jedynie między poprzednikiem (ojetem) i następnikiem (synem) w drzewie.
- 6.3. Podaj pełny zapis algorytmu mnożenia dwóch macierzy na maszynie PRAM.
- 6.4. Udowodnij, że dla nieskończoność wielu wartości  $n$  istnieją takie drzewa  $T_n$ , że  $|\text{reduce}(T_n)| = \lfloor 2/3 |T_n| \rfloor$ .
- 6.5. Skonstruj algorytm obliczania najliczniejszego zbioru niezależnego w dowolnym drzewie w czasie  $O(\log n)$  przy użyciu  $n$  procesorów (w rozdziale tym podaliśmy algorytm dla drzew binarnych).

## Zadania

- 6.6. Rozwiąż zadanie 6.5 dla problemu minimalnego zbioru dominującego.
- 6.7. Skonstruj podobny (jak w zadaniach 6.5 i 6.6) algorytm kolorowania krawędzi drzewa minimalną liczbą kolorów. Krawędzie o tym samym kolorze nie mogą mieć wspólnego węzła.
- 6.8. Oszacuj rozmiar grafu  $\text{graf}(P)$ , gdzie  $P$  jest prostym programem sekwencyjnym obliczającym liczby Fibonacciego (zgodnie z równaniem rekurencyjnym definiującym te liczby).
- 6.9. Oszacuj liczbę iteracji w algorytmie jednoczesnych podstawień (z pierwszą definicją zmiennych bezpiecznych), zastosowanym do programu  $P$  z poprzedniego zadania.
- 6.10. Podaj algorytm liczenia  $n$ -tej liczby Fibonacciego w czasie  $O(\log n)$  przy użyciu  $n$  procesorów, przy założeniu, że operacje arytmetyczne mają koszt jednostkowy (pomimo tego, że liczby Fibonacciego są bardzo duże).
- 6.11. Skonstruj algorytm sprawdzania w czasie  $O(\log n)$  i przy użyciu  $n$  procesorów, czy dane słowo jest słowem Fibonacciego (zob. rozdział 5).
- 6.12. Opisz pełną implementację jednej iteracji w algorytmie jednoczesnych podstawień z drugą definicją zmiennych bezpiecznych. Pokaż, że koszt jednej iteracji jest proporcjonalny do mnożenia macierzy liczbowych o rozmiarze  $n \times n$ .
- 6.13. Udowodnij, że dla nieskończoność wielu wartości  $n$  istnieje graf obliczeń  $G_n$ , taki że  $|\text{reduce}(G_n)| = \lfloor 2/3 |G_n| \rfloor$ .
- 6.14. Udowodnij, że rozmiar prostego programu sekwencyjnego  $P$  dla problemów programowania dynamicznego jest  $O(n^3)$  i że  $\deg(P) = O(n)$ .
- 6.15. Napisz dokładnie wzory rekurencyjne na obliczanie optymalnego BST metodą programowania dynamicznego.
- 6.16. Zrób to samo dla problemu minimalnej triangulacji wielokąta wypuklego.
- 6.17. Podaj dokładny zapis algorytmu scalania dwóch posortowanych ciągów długości  $n$  w czasie  $O(\log n)$  na PRAM. Zgrubny opis takiego algorytmu został podany w podrozdziale o sortowaniu, przed opisami sieci sortujących.
- 6.18. Korzystając z zasady zero-jedynkowej, udowodnij poprawność sieci scalających metodą odd-even.
- 6.19. Wykaż poprawność sieci scalających metodą bitoniczną.

- 6.20. Maszyna przeplotowa jest równoległą maszyną składającą się z  $n$  procesorów (RAM). Procesor  $i$ -ty ( $i < n$ ) jest połączony z procesorem  $(i + 1)$  dla każdego parzystego  $i$ . Oprócz tego procesor  $i$ -ty jest połączony z procesorem  $j_i$ , gdzie  $(j_1, j_2, \dots, j_n) = \text{przeplot}(1, 2, \dots, n)$ . Podaj implementację scalania metodą bitoniczną na maszynie przeplotowej.
- 6.21. Skonstruuj algorytm sumowania  $n$  liczb w czasie  $\log n$  na maszynie przeplotowej.
- 6.22. Korzystając z tego, że sortowanie można wykonać w czasie  $O(\log^2 n)$  za pomocą  $n$  procesorów, skonstruuj algorytm rozwiązywania problemu WW i obliczania najdłuższego powtarzającego się podslowa w czasie  $O(\log^3 n)$  przy użyciu  $n$  procesorów (zaprojektuj równoległą wersję algorytmu KMR).
- 6.23. Rozwiąż zadanie 6.22 dla przypadku dwuwymiarowego (zamiast najdłuższego powtarzającego się podslowa szukaj maksymalnej powtarzającej się podtablicy).
- 6.24. Zdefiniujmy drzewa Fibonacciego  $T_n$  w następujący sposób. Drzewa  $T_1$  i  $T_2$  składają się tylko z korzeni. Drzewo  $T_{n+2}$  składa się z korzenia, którego jedynym synem jest sklejony korzeń drzew  $T_n$  i  $T_{n+1}$ . Wtedy  $T_3$  składa się z dwóch wierzchołków, a  $T_4$  z trzech. Jaki jest związek liczby wierzchołków z liczbami Fibonacciego? Zastanów się, jak działa operacja *reduce* na drzewach Fibonacciego. Udowodnij, że potrzeba co najmniej  $(\log_f n - c)$  iteracji operacji *reduce* dla drzew Fibonacciego, gdzie  $n$  jest liczbą wierzchołków,  $f = (\sqrt{5} + 1)/2$ , a  $c$  pewną stałą.
- 6.25. Udowodnij, że dla każdego drzewa mającego  $n$  wierzchołków wystarczy  $\log_f n + c$  operacji *reduce*, żeby zredukować drzewo do pojedynczego wierzchołka, gdzie  $c$  jest pewną stałą (niekoniecznie taką samą jak w poprzednim zadaniu).

## 7

## Algorytmy grafowe

Algorytmiczna teoria grafów jest dziedziną informatyki teoretycznej, w której w ostatnich latach nowe algorytmy pojawiały się najczęściej. Algorytmy grafowe można spotkać w wielu różnych dziedzinach informatyki, na przykład w złożoności obliczeniowej, geometrii obliczeniowej, grafice komputerowej, metodach translacji sieciach komputerowych, obliczeniach równoległych i rozproszonych. W tym rozdziale zajmiemy się podstawowymi metodami konstrukcji efektywnych algorytmów grafowych. Nie jest naszym zamiarem podanie pełnego przeglądu najważniejszych problemów i algorytmów grafowych. Skoncentrujemy się raczej na przedstawieniu pewnych ogólnych metod przetwarzania grafów, w tym tak ważnych jak metody systematycznego przeszukiwania grafu i syntezy informacji o grafie z jego drzewa rozpinającego. Zwróciemy także uwagę na możliwość sprowadzania jednego problemu do drugiego, a tym samym wykorzystywania istniejących algorytmów do rozwiązywania nowych zadań. Nasze rozważania zilustrujemy, rozwiązując następujące problemy:

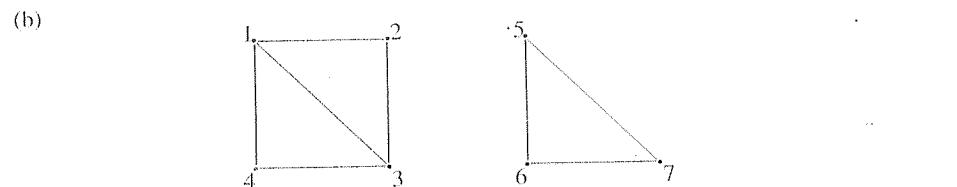
- spójnych, dwuspójnych i silnie spójnych składowych;
- silnej orientacji grafów nieorientowanych;
- cyklu Eulera;
- najkrótszych ścieżek i minimalnego drzewa rozpinającego;
- 5-kolorowania grafów planarnych.

Zrozumienie materiału z tego rozdziału ułatwi Ci znajomość podstawowego kursu z teorii grafów. W naszych rozważaniach skupimy się głównie na algorytmicznej stronie omawianych zagadnień, pozostawiając Ci często udowodnienie prostszych faktów teoriografowych.

Niech  $G = (V, E)$  będzie  $n$ -wierzchołkowym grafem nieorientowanym. Pisząc „graf”, będziemy zawsze mieli na myśli graf nieorientowany. Przypomnijmy, że liczbę krawędzi grafu  $G$  oznaczamy przez  $m$ . Będziemy zakładać, że  $V = \{1, 2, \dots, n\}$ , chyba że zaznaczymy, że jest inaczej. Najprościej jest reprezentować graf na płaszczyźnie.

Wierzchołkom grafu odpowiadają wtedy punkty płaszczyzny, a krawędziami – odcinki łączące punkty reprezentujące ich końce. Niestety, sposób ten nie może być wykorzystany do implementacji grafu w komputerze. W tym wypadku grafy reprezentujemy zazwyczaj za pomocą list lub macierzy sąsiedztwa (zob. rozdział 1). Należy podkreślić różnice w złożoności konstruowania obu reprezentacji. O ile listy sąsiedztwa można zbudować w czasie  $O(n+m)$ , o tyle budowa macierzy sąsiedztwa wymaga czasu  $\Omega(n^2)$ . Jeśli liczba krawędzi w grafie jest istotnie mniejsza od  $n^2$ , to listy sąsiedztwa są oszczędniejsze – zarówno jeśli chodzi o czas budowy, jak i o zużycie pamięci. Zaletą macierzy sąsiedztwa jest jednak możliwość sprawdzania w czasie stałym, czy istnieje krawędź między dowolną parą wierzchołków. W prezentowanych algorytmach przyjmujemy, że graf jest dany przez listy sąsiedztwa. Na rysunku 7.1 przedstawiamy przykładowy graf i różne jego reprezentacje.

(a)  $G = (\{1, 2, 3, 4, 5, 6, 7\}, \{1-2, 2-3, 3-4, 4-1, 1-3, 5-6, 6-7, 5-7\})$



(c) Wierzchołki      Sąsiedzi

1	2, 3, 4
2	1, 3
3	4, 1, 2
4	3, 1
5	7, 6
6	5, 7
7	6, 5

(d)

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	1	0	1	0	0	0	0
3	1	1	0	1	0	0	0
4	1	0	1	0	0	0	0
5	0	0	0	0	0	1	1
6	0	0	0	0	1	0	1
7	0	0	0	0	1	1	0

Rys. 7.1. a) Graf  $G$ ; b) reprezentacja graficzna; c) listy sąsiedztwa; d) macierz sąsiedztwa

Przypomnijmy teraz podstawowe pojęcia dotyczące grafów, którymi będziemy się posługiwać w dalszej części rozdziału.

**Drogą** (albo **ścieżką**) w grafie  $G$  nazywamy każdy skończony ciąg wierzchołków  $v_0, v_1, \dots, v_k$ , taki że dla każdego  $i = 0, \dots, k-1$ ,  $v_i - v_{i+1}$  ( $v_i \rightarrow v_{i+1}$  w wypadku grafu zorientowanego) jest krawędzią grafu. Liczba  $k$  jest **długością** ścieżki. O ścieżce  $v_0, v_1, \dots, v_k$  mówimy, że łączy wierzchołki  $v_0$  i  $v_k$  (lub prowadzi od  $v_0$  do  $v_k$ , gdy  $G$  jest zorientowany). Wierzchołki  $v_0$  i  $v_k$  nazywamy – odpowiednio – **początkiem** i **końcem** ścieżki. O krawędzi łączącej dwa kolejne wierzchołki na ścieżce mówimy, że **należy do tej ścieżki**. Długość najkrótszej ścieżki łączącej dwa wierzchołki  $v$  i  $w$  nazywamy ich **odległością** w grafie (lub

odległośćą od  $v$  do  $w$  w wypadku grafu zorientowanego). Ścieżkę nazywamy **prostą**, jeżeli żadne dwa wierzchołki na ścieżce się nie powtarzają. O ścieżce mówimy, że jest **cyklem**, jeżeli pierwszy wierzchołek i ostatni są takie same. Jeśli wszystkie wierzchołki, z wyjątkiem  $v_0$  i  $v_k$ , są parami różne oraz  $k > 2$  ( $k > 1$  dla grafów zorientowanych), to cykl nazywamy **prostym**. Graf  $G$  jest **spójny**, jeżeli każde dwa jego wierzchołki można połączyć ścieżką. Graf  $G_1 = (V_1, E_1)$ , w którym  $V_1$  jest podzbiorem zbioru wierzchołków  $V$ , a  $E_1$  jest podzbiorem zbioru krawędzi grafu  $G$  łączących tylko wierzchołki z  $V_1$ , nazywamy **podgrafem**  $G$ . Graf  $G_1$  jest **podgrafem rozpinającym**, jeżeli  $V_1 = V$ . W sytuacji, kiedy  $E_1$  zawiera wszystkie krawędzie grafu  $G$  o końcach w wierzchołkach z  $V_1$ , mówimy o  $G_1$ , że jest **indukowany** przez  $V_1$ . Graf  $G_1$  jest **spójną składową** grafu  $G$ , jeżeli jest jego maksymalnym (w sensie zawierania zbiorów wierzchołków i krawędzi) spójnym podgrafem.

#### □ PRZYKŁAD:

Graf z rysunku 7.1 nie jest grafem spójnym. Ma dwie spójne składowe: jedną o wierzchołkach 1, 2, 3, 4 i drugą o wierzchołkach 5, 6, 7.

### 7.1.

## Spójne składowe

W bardzo wielu zadaniach obliczeniowych dla grafów zakłada się, że przetwarzany graf jest spójny. Stąd jednym z podstawowych problemów napotykanych przy rozwiązywaniu problemów grafowych jest problem znajdowania spójnych składowych danego grafu  $G$ . Można go sformułować następująco:

#### PROBLEM 7.1.

Dla danego grafu  $G = (V, E)$  mamy obliczyć (znaleźć) funkcję  $C: V \rightarrow \{1, 2, \dots, n\}$ , taką że dwa wierzchołki  $v$  i  $w$  należą do tej samej spójnej składowej  $G$  wtedy i tylko wtedy, gdy  $C(v) = C(w)$ .

Rozwiązywanie powyższego problemu opiera się na spostrzeżeniu, że dwa wierzchołki są w tej samej spójnej składowej wtedy i tylko wtedy, gdy można je połączyć ścieżkami z tym samym wierzchołkiem w grafie. Do znajdowania spójnych składowych wykorzystamy metodę przechodzenia grafu w głąb (DFS), o której była już mowa w rozdziale 1. Przechodzenie w głąb zrealizujemy za pomocą procedury rekurencyjnej *search*, opisanej nieco dalej. Tak jak w rozdziale 1, zakładamy, że z każdym wierzchołkiem  $v$  są związane dwa pola: *visited(v)* i *current(v)*. Pierwsze pole służy do zaznaczania stanu wierzchołka: odwiedzony (true) lub nie odwiedzony (false). Drugie pole – *current(v)* – jest wskaźnikiem do pierwszego wierzchołka  $w$  na liście sąsiedztwa  $v$ , takiego że krawędź  $v - w$  nie była jeszcze odwiedzona z  $v$ . Listę sąsiedztwa  $v$  oznaczamy przez *L(v)*. W opisie procedury *search* opuściliśmy pewien fragment, który zaznaczyliśmy trzema kropkami.

Fragment ten zależy od rozwiązywanego problemu. Za chwilę pokażemy, jak on wygląda w wypadku algorytmu obliczania spójnych składowych.

```

procedure search(v : wierzchołek);
var
  w : wierzchołek;
begin
  visited(v) := true; {zamarkuj wierzchołek v jako odwiedzony}
  :
  while current(v) ≠ nil do
  begin
    w := wierzchołek na liście  $L(v)$  wskazywany przez
    current(v);
    if not visited(w) then search(w) {***};
    current(v) := wskaźnik do następnego wierzchołka na
    liście  $L(v)$ 
  end
end;

```

Jako zadanie pozostawiamy Ci udowodnienie, że wywołanie  $search(u)$  spowoduje odwiedzenie wszystkich wierzchołków połączonych w grafie ścieżką z  $u$ . Mając do dyspozycji procedurę  $search$ , łatwo już rozwiązać problem spójnych składowych. Funkcję  $C$  określającą w ten sposób, że  $C(v)$  jest równe najmniejszemu wierzchołkowi w spójnej składowej zawierającej  $v$ . Identyfikator wykrywanej właśnie spójnej składowej (jej najmniejszy wierzchołek) jest pamiętany w zmiennej globalnej  $id$ . Teraz, żeby obliczyć  $C(v)$ , wystarczy wpisać w opuszczone miejsce w opisie procedury  $search$  instrukcję

$C(v) := id$

Pelny algorytm obliczania spójnych składowych został zapisany za pomocą procedury  $c\text{-components}$ .

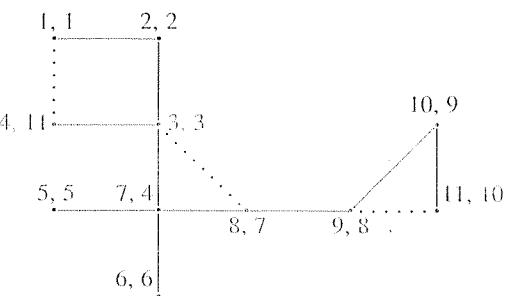
```

procedure c-components;
begin
  for v := 1 to n do
  begin
    current(v) := wskaźnik do pierwszego wierzchołka na liście
     $L(v)$ ;
    visited(v) := false
  end;
  for v := 1 to n do
  begin
    if not visited(v) then
    begin
      id := v; search(v)
    end
  end
end;

```

W procedurze  $c\text{-components}$  lista sąsiedztwa każdego wierzchołka jest przeglądana jeden raz. Ponieważ łączna długość wszystkich list sąsiedztwa wynosi  $2m$ , czas działania procedury  $c\text{-components}$  jest liniowy, a dokładniej  $O(n + m)$ . Przeglądanie listy sąsiedztwa każdego wierzchołka (instrukcja `while` w procedurze  $search$ ) może być także rozumiane jako przeglądanie listy krawędzi grafu o jednym z końców właśnie w tym wierzchołku. Jak już wspominaliśmy, metoda przechodzenia grafu, realizowana za pomocą procedury  $search$ , nosi nazwę metody przechodzenia grafu w głąb. Kierujemy się w głąb grafu tak dugo, jak dugo można. Metoda ta jest podstawą konstrukcji bardzo wielu efektywnych algorytmów grafowych, działających często w czasie liniowym. Umożliwia ona zebranie potrzebnych informacji dotyczących struktury grafu przy tylko jednokrotnym przejrzeniu go. Zdobyte informacje wykorzystuje się następnie do obliczania właściwego rozwiązania rozpatrywanego problemu.

Załóżmy, że badany graf jest spójny. W trakcie wykonywania procedury  $search$  krok zaznaczony trzema gwiazdkami odpowiada przejściu po krawędzi od zamarkowanego wierzchołka  $v$  do nie zamarkowanego wierzchołka  $w$ . Wyróżnijmy wszystkie takie krawędzie i nich  $F$  będzie zbiorem tych krawędzi. Nietrudno zauważać, że podgraf  $(V, F)$  jest drzewem rozpinającym badanego grafu (spójnym podgrafem bez cykli prostych, zawierającym wszystkie wierzchołki). Drzewo to będziemy traktować jak drzewo z korzeniem w wierzchołku, od którego rozpoczynamy przechodzenie. Jeżeli w kroku {\*\*\*} procedury  $search$  przechodzimy od wierzchołka  $v$  do  $w$ , to w tak otrzymanym drzewie  $v$  jest poprzednikiem  $w$ , a  $w$  jest następcą  $v$ . Omawiane drzewo będziemy nazywać **drzewem przechodzenia w głąb**. Przechodząc graf w głąb, można dodatkowo numerować wierzchołki w kolejności odwiedzania ich. Numer wierzchołka  $v$  będziemy oznaczać przez  $nr(v)$ . Przykładowy graf z ponumerowanymi w głąb wierzchołkami i z zaznaczonymi krawędziami drzewa przechodzenia (linie ciągłe) jest przedstawiony na rysunku 7.2. Liczby przy każdym wierzchołku oznaczają, odpowiednio, jego identyfikator oraz numer w głąb. Dla przejrzystości przykładu przyjmujemy, że wierzchołki na listach sąsiedztwa są uporządkowane rosnąco. Należy tu jednak podkreślić, że kolejność wierzchołków na listach sąsiedztwa może być dowolna. Przechodzenie grafu rozpoczyna się od wierzchołka 1.



Rys. 7.2. Graf ponumerowany w głąb z wyróżnionymi krawędziami drzewa rozpinającego (linie ciągłe)

7.2.

## Dwuspójne składowe

Pokażemy teraz, jak można wykorzystać zdobytą informację (numerację i drzewo) do wyznaczenia dwuspójnych składowych grafu  $G$ . Przypomnijmy najpierw definicję dwuspójnych składowych. Niech  $R$  będzie relacją na krawędziach grafu  $G$  zdefiniowaną następująco:  $e_1Re_2$  wtedy i tylko wtedy, gdy  $e_1 = e_2$  lub istnieje cykl prosty zawierający jednocześnie  $e_1$  oraz  $e_2$ . Można udowodnić, że  $R$  jest relacją równoważności (zadanie 7.8). Podgrafy grafu  $G$ , indukowane przez zbiory wierzchołków incydentnych z krawędziami z klas abstrakcji relacji  $R$ , nazywamy **dwuspójnymi składowymi**, a wierzchołki należące do dwóch lub więcej różnych dwuspójnych składowych – **wierzchołkami rozdzielającymi**. Wierzchołki rozdzielające charakteryzują się tym, że usunięcie ich „rozspójnia” graf. O krawędziach należących do jednoelementowych klas abstrakcji relacji  $R$  mówimy, że są **mostami**. Usunięcie mostu także „rozspójnia” graf. Zaobserwujmy, że jeśli dwuspójna składowa zawiera więcej niż 2 wierzchołki (nie jest mostem), to każde jej dwa różne wierzchołki leżą na cyklu prostym złożonym tylko z wierzchołków tej dwuspójnej składowej. Graf zawierający tylko jedną dwuspójną składową nazywamy **grafem dwuspójnym**.

### □ PRZYKŁAD:

Dwuspójnymi składowymi grafu z rysunku 7.2 są podgrafy indukowane przez zbiory wierzchołków  $\{1, 2, 3, 4\}$ ,  $\{3, 7, 8\}$ ,  $\{5, 7\}$ ,  $\{6, 7\}$ ,  $\{8, 9\}$ ,  $\{9, 10, 11\}$ . Wierzchołkami rozdzielającymi są 3, 7, 8 i 9. Mostami są krawędzie 5—7, 6—7 oraz 8—9.

Formalnie problem znajdowania dwuspójnych składowych definiuje się następująco:

### PROBLEM 7.2.

Dany jest spójny graf  $G = (V, E)$ . Mamy obliczyć funkcję  $B: E \rightarrow \{1, \dots, m\}$ , taką że  $B(e_1) = B(e_2)$  wtedy i tylko wtedy, gdy  $e_1$  i  $e_2$  należą do tej samej dwuspójnej składowej.

W celu wykorzystania drzewa przechodzenia w głąb do znajdowania dwuspójnych składowych dokonajmy pewnych obserwacji.

### OBSERWACJA 1:

Każda krawędź grafu łączy zawsze potomka z przodkiem w drzewie przechodzenia w głąb. Krawędź drzewowa łączy poprzednika z następnikiem.

### OBSERWACJA 2:

Numer w głąb wierzchołka  $v$  jest mniejszy od numeru każdego jego właściwego (różnego od  $v$ ) potomka.

## 7.2. Dwuspójne składowe

### OBSERWACJA 3:

Korzeń drzewa przechodzenia w głąb jest wierzchołkiem rozdzielającym wtedy i tylko wtedy, gdy ma co najmniej dwa następniaki.

### OBSERWACJA 4:

Niech  $low: V \rightarrow V$  będzie funkcją zdefiniowaną rekurencyjnie w następujący sposób:  $low(v) = \min(\{nr(v)\} \cup \{nr(w): v \text{ --- } w \text{ jest krawędzią niedrzewową w } G\} \cup \{low(u): u \text{ jest następnikiem } v \text{ w drzewie przechodzenia}\})$ .

Mniej formalnie,  $low(v)$  jest najmniejszym numerem w głąb  $nr(w)$ , takim że wierzchołek  $w$  jest przodkiem  $v$  w drzewie przechodzenia w głąb i istnieje ścieżka o początku w  $v$  i końcu w  $w$ , na której wszystkie wierzchołki, poza  $w$ , leżą w poddrzewie o korzeniu w  $v$ .

### □ PRZYKŁAD:

Dla grafu z rysunku 7.2  $low(1) = 1$ ,  $low(2) = 1$ ,  $low(3) = 1$ ,  $low(4) = 1$ ,  $low(5) = 5$ ,  $low(6) = 6$ ,  $low(7) = 3$ ,  $low(8) = 3$ ,  $low(9) = 8$ ,  $low(10) = 8$ ,  $low(11) = 8$ .

### OBSERWACJA 5:

Wierzchołek  $v$ , różny od korzenia, jest rozdzielający wtedy i tylko wtedy, gdy dla jego pewnego następnika  $u$  w drzewie przechodzenia w głąb  $low(u) \geq nr(v)$ . (Jeśli nierówność zachodzi, to każda ścieżka łącząca  $u$  z korzeniem drzewa zawiera  $v$ . Stąd wynika, że  $v$  jest wierzchołkiem rozdzielającym).

### □ PRZYKŁAD:

W grafie z rysunku 7.2 następujące wierzchołki są rozdzielające: 3, ponieważ  $low(7) = 3$ ; 7, ponieważ  $low(5) = 5$  (i  $low(6) = 6$ ); 8, ponieważ  $low(9) = 8$ ; oraz 9, ponieważ  $low(10) = 8$ .

### OBSERWACJA 6:

Założymy, że badany graf ma co najmniej dwie dwuspójne składowe. Zauważmy, że w takim grafie co najmniej jedna (faktycznie co najmniej dwie) dwuspójna składowa zawiera tylko jeden wierzchołek rozdzielający. Niech  $B$  będzie taką dwuspójną składową,  $b$  jedynym wierzchołkiem rozdzielającym w  $B$ , a  $x$  jedynym następnikiem  $b$  w drzewie przechodzenia w głąb, należącym do  $B$ . W trakcie przechodzenia grafu metodą w głąb, po przejściu z  $b$  do  $x$ , ponowny powrót do wierzchołka  $b$  następuje dopiero po przejściu list sąsiedztwa wszystkich wierzchołków składowej  $B$ , różnych od  $b$ . Jeśli na przeglądanie listy sąsiedztwa dowolnego wierzchołka  $v$  patrzmy jak na odwiedzanie kolejnych krawędzi o jednym z końców w tym wierzchołku, to krawędzie z listy sąsiedztwa wierzchołków dwuspójnej składowej  $B$ , różnych od  $b$ , są przeglądane kolejno zaraz po odwiedzeniu krawędzi  $b \rightarrow x$  na liście  $b$ .Więcej, jeśli usuniemy z grafu  $G$  krawędzie składowej  $B$ , to kolejność przeglądania krawędzi w pozostałej części grafu się nie zmieni.

Przedstawimy teraz zmodyfikowaną wersję procedury *search*, w której – jednocześnie – będziemy numerować wierzchołki w kolejności odwiedzania w głęb, obliczali funkcję *low* oraz wyznaczali dwuspójne składowe. Do tego celu wykorzystamy trzy struktury pomocnicze: stos *S*, na który będziemy odkładali kolejno przeglądane krawędzie, oraz zmienne *id* i *l*, na których będziemy pamiętały liczby, odpowiednio, już wygenerowanych dwuspójnych składowych i już odwiedzonych (zamarkowanych) wierzchołków. Każda krawędź *v*–*w* znajduje się zarówno na liście sąsiedztwa *v* (reprezentowana przez *w*), jak i na liście sąsiedztwa *w* (reprezentowana przez *v*). Niech *v* będzie przodkiem *w* w drzewie przechodzenia. Jeśli *v* jest poprzednikiem *w*, to krawędź *v*–*w* będzie odkładana na stos, gdy napotkamy ją na liście sąsiedztwa *v*. Jeśli *v* nie jest poprzednikiem *w*, to krawędź tę odkładamy na stos, gdy napotykamy ją na liście sąsiedztwa *w*. Żeby łatwo sprawdzić, czy *v* jest poprzednikiem *w*, przyjmiemy, że jest znany poprzednik odwiedzanego właśnie wierzchołka. Początkowo *id* = 0 i *l* = 0.

```

procedure search-b(v, u : wierzchołek);
{ v – odwiedzany właśnie wierzchołek; u – jego poprzednik w drzewie
przechodzenia; jeśli v jest korzeniem, to u = 0}
var
  w : wierzchołek;
  e : krawędź;
begin
  visited(v) := true;
  l := l + 1; nr(v) := l; low(v) := l;
  {przeglądanie listy sąsiedztwa wierzchołka v}
  /*,while current(v) ≠ nil do
  begin
    w := wierzchołek na liście L(v), wskazywany
    przez current(v);
    if not visited(w) then
      begin
        {w jest następnikiem v w drzewie przechodzenia}
        push(S, v–w);
        search-b(w, v);
        if low(w) ≥ nr(v) then
          begin
            {wykryta została dwuspójna składowa zawierająca
            krawędź v–w; wszystkie jej krawędzie znajdują
            się kolejno na szczycie stosu, przy czym v–w
            jest położona najgłębiej}
            id := id + 1;
            repeat
              e := front(S); B(e) := id; pop(S)
            until e = v–w
          end
        end
      end
    end;
  end;
end;

```

## 7.2. Dwuspójne składowe

```

else
  if low(w) < low(v) then low(v) := low(w)
end
else
  if (nr(w) < nr(v)) and (w ≠ u) then
begin
  {w jest przodkiem v w drzewie różnym od poprzednika}
  push(S, v–w);
  if nr(w) < low(v) then low(v) := nr(w)
end;
current(v) := wskaznik do następnego wierzchołka na
liście L(v)
end;
end;

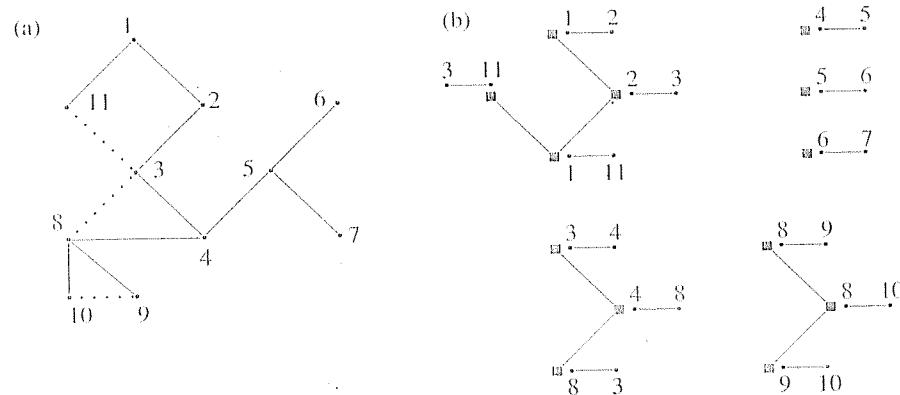
```

Procedurę *search-b* można wywołać dla dowolnego wierzchołka *v* i *u* = 0. Jej poprawność wynika bezpośrednio z obserwacji 1-6 (zad. 7.9). Podobnie jak poprzednio, każda lista sąsiedztwa jest przeglądana tylko raz. Dodatkowo każdą krawędź tylko raz wstawiamy na stos i tylko raz z niego usuwamy. Stąd wynika, że złożoność powyższego algorytmu jest liniowa ( $O(n + m)$ ).

W ostatnich latach nastąpił szybki rozwój badań nad algorytmami równolegymi. Okazało się, że stosowane w tej dziedzinie metody umożliwiają konstruowanie efektywnych, alternatywnych do istniejących, algorytmów sekwencyjnych. Poszukiwanie takich metod do rozwiązywania problemów grafowych było konieczne, ponieważ przechodzenie w głęb okazało się trudne do zrównoleglenia. Pokażemy, że dwuspójne składowe daje się wyznaczyć w czasie liniowym, nie przechodząc grafu w głęb. W tym celu sprowadzimy problem dwuspójnych składowych do problemu spójnych składowych. Dla danego spójnego grafu *G* zbudujemy pomocniczy graf *G'*, którego spójne składowe będą odpowiadały dwuspójnym składowym *G*. Zbiorem wierzchołków grafu *G'* będzie zbiór krawędzi *G*. Dwie krawędzie *e*<sub>1</sub> i *e*<sub>2</sub> będą należały do tej samej spójnej składowej *G'* wtedy i tylko wtedy, gdy należą do tej samej dwuspójnej składowej *G*. Omawiany algorytm jest przykładem zastosowania ważnej metody używanej w projektowaniu algorytmów, a polegającej na sprowadzeniu rozwiązywanego problemu do problemu, dla którego rozwiązanie (algorytm) jest już znane. Oczywiście, żeby otrzymany w ten sposób algorytm był efektywny, zarówno sprowadzenie go do znanego rozwiązania, jak i znane rozwiązanie muszą być efektywne. Niech *T* będzie dowolnym drzewem rozpinającym grafu *G* o korzeniu w dowolnie wybranym wierzchołku *r*. W przeciwieństwie do poprzedniego algorytmu, *T* nie musi być drzewem przechodzenia w głęb. Założymy także, że wierzchołki *T* są ponumerowane w kolejności odwiedzania ich metodą preorder. W rozdziale 1 pokazaliśmy, w jaki sposób obliczyć taką numerację w czasie liniowym. Od tej chwili będziemy utożsamiać wierzchołki z ich numerami. Korzeń ma zawsze numer 1. Niech *p*(*v*) będzie poprzednikiem *v* w drzewie *T* dla każdego wierzchołka *v*, różnego od korzenia.

Jak już powiedzieliśmy wcześniej, wierzchołkami pomocniczego grafu  $G'$  są krawędzie grafu  $G$ , a krawędzie w  $G'$  są tworzone według następujących reguł (zob. rys. 7.3):

- $(u \rightarrow v) \rightarrow (v \rightarrow w)$  jest krawędzią w  $G'$ , jeżeli  $u$  jest poprzednikiem  $v$  w drzewie  $T$ , krawędź  $v \rightarrow w$  nie jest krawędzią drzewową i  $v \rightarrow w$ ;
- $(u \rightarrow v) \rightarrow (x \rightarrow w)$  jest krawędzią w  $G'$ , jeżeli  $u$  jest poprzednikiem  $v$ ,  $x$  jest poprzednikiem  $w$  ( $u \rightarrow v$  i  $x \rightarrow w$  są krawędziami drzewowymi),  $v \rightarrow w$  jest krawędzią nie-drzewową oraz  $v \rightarrow w$  nie są w relacji potomek-przodek w drzewie  $T$ ;
- $(u \rightarrow v) \rightarrow (v \rightarrow w)$  jest krawędzią w  $G'$ , jeżeli  $u$  jest poprzednikiem  $v$  w drzewie,  $v$  jest poprzednikiem  $w$  oraz pewna krawędź grafu  $G$  łączy potomka  $w$  w drzewie  $T$  z wierzchołkiem nie będącym potomkiem  $v$ .



Rys. 7.3. a) Graf z zaznaczonymi krawędziami drzewa rozpinającego (linie ciągłe), w którym wierzchołki są ponumerowane metodą preorder; b) graf  $G'$

Zauważmy, że jeśli dwie krawędzie z  $G$  są połączone bezpośrednio w  $G'$ , to leżą na tym samym cyklu prostym w  $G$ , a więc należą do tej samej dwuspójnej składowej. Co więcej, zachodzi twierdzenie Tarjana-Vishkina [TV].

### Twierdzenie 7.1.

Dwie krawędzie należą do tej samej dwuspójnej składowej grafu  $G$  wtedy i tylko wtedy, gdy jako wierzchołki należą do tej samej spójnej składowej grafu  $G'$ .

**Dowód:** Każda niedrzewowa krawędź  $x \rightarrow y$  wyznacza w  $G$  cykl prosty składający się z tej krawędzi i jedynie ścieżki prostej w  $T$ , łączącej  $x$  i  $y$ . Dla ustalonego drzewa  $T$  cykle takiej postaci nazywają się cyklami bazowymi. Cykle bazowe charakteryzują się tym, że zbiór krawędzi dowolnego cyklu prostego w grafie charakteryzuje się tym, że zbiór krawędzi cyklu bazowego wyznaczanego przez krawędź  $x \rightarrow y$ .

<sup>10</sup> Różnicą symetryczną skończonych zbiorów  $X_1, \dots, X_k$  nazywamy zbiór zawierający dokładnie te elementy, które należą do nieparzystej liczby zbiorów spośród  $X_1, \dots, X_k$ .

bazowych. Niech  $Q$  będzie relacją na krawędziach grafu  $G$  zdefiniowaną następująco:  $e_1 Q e_2$  wtedy i tylko wtedy, gdy  $e_1$  i  $e_2$  należą do tego samego cyklu bazowego. Przez  $Q'$  oznaczmy zwróci i przekonie domknięcie relacji  $Q$ . Mówimy, że dwie krawędzie  $e$  i  $f$  są w relacji  $Q'$  wtedy i tylko wtedy, gdy  $e = f$  lub istnieje taki ciąg  $e_1, e_2, \dots, e_k$  krawędzi grafu  $G$ , gdzie  $k > 1$ , że  $e_1 Q e_2, e_2 Q e_3, \dots, e_{k-1} Q e_k$  oraz  $e_1 = e$  i  $e_k = f$ . Pokażemy, że  $Q' = R$ , gdzie  $R$  jest relacją równoważności zdefiniowaną wcześniej. Przypomnijmy, że klasy abstrakcji relacji  $R$  wyznaczają dwuspójne składowe grafu  $G$ . Dwie krawędzie są w relacji  $R$  wtedy i tylko wtedy, gdy są identyczne lub należą do tego samego cyklu prostego. Wynika stąd, że  $Q' \subseteq R$ . Pozostaje nam zatem pokazać, że  $R \subseteq Q'$ . Niech  $e_1, e_2$  będą różnymi krawędziami w  $G$ , należącymi do pewnego cyklu prostego  $C$ . Z tego co powiedzieliśmy wcześniej,  $C$  można przedstawić jako różnicę symetryczną pewnych, parami różnych, cykli bazowych  $C_1, C_2, \dots, C_k$ , dla pewnego  $k > 0$ . Bez straty ogólności możemy założyć, że kolejność  $C_1, C_2, \dots, C_k$  jest taka, że dla każdego  $1 < i \leq k$  cykl  $C_i$  ma wspólną krawędź z pewnym cyklem  $C_j$ , gdzie  $1 \leq j < i$ . (W przeciwnym razie krawędzie ze zbioru będącego różnicą symetryczną  $C_1, C_2, \dots, C_k$  nie mogłyby należeć do tego samego cyklu prostego). Teraz przez indukcję względem  $k$  możemy udowodnić, że każda para krawędzi należących do cykli  $C_1, C_2, \dots, C_k$  jest w relacji  $Q'$ . W szczególności  $e_1 Q' e_2$ , co implikuje  $R \subseteq Q'$ . W ten sposób wykazaliśmy, że  $Q' = R$ . Żeby udowodnić twierdzenie, wystarczy tylko pokazać, że każde dwie krawędzie sąsiadujące w  $G'$  należą do wspólnego cyklu bazowego w grafie  $G$  oraz że krawędzie każdego cyklu bazowego z grafu  $G$  są wierzchołkami tej samej spójnej składowej grafu  $G'$ .

Niech  $u \rightarrow v$  oraz  $x \rightarrow w$  będą krawędziami, które sąsiadują w  $G'$ . Rozważmy trzy przypadki konstrukcji krawędzi grafu  $G'$ .

- Krawędź  $u \rightarrow v$  jest krawędzią drzewową należącą do cyklu bazowego wyznaczanego przez krawędź  $x \rightarrow w$  (tutaj  $x = v$ ).
- Krawędzie drzewowe  $u \rightarrow v$  i  $x \rightarrow w$  należą do cyklu bazowego wyznaczanego przez krawędź  $v \rightarrow w$ .
- Wierzchołki  $x$  i  $v$ , są takie same ( $x = v$ ),  $u \rightarrow v$ ,  $v \rightarrow w$  są krawędziami drzewowymi,  $u$  jest poprzednikiem  $v$ ,  $v$  jest poprzednikiem  $w$  oraz istnieje krawędź  $y \rightarrow z$ , taka że  $y$  jest potomkiem  $w$ , a  $z$  nie jest potomkiem  $v$ . Krawędzie  $u \rightarrow v$  i  $v \rightarrow w$  są krawędziami cyklu bazowego wyznaczanego przez krawędź  $y \rightarrow z$ .

Rozważmy teraz cykl bazowy wyznaczony przez krawędź niedrzewową  $x \rightarrow y$ . Bez straty ogólności możemy założyć  $x < y$ . Niech  $z$  będzie najbliższym wspólnym przodkiem wierzchołków  $x$  i  $y$  w drzewie  $T$ , tzn. niech będzie pierwszym wspólnym wierzchołkiem na ścieżkach łączących  $x$  i  $y$  z korzeniem drzewa. Rozpatrując przypadki (1), (2), (3), otrzymujemy, że w grafie  $G'$  są następujące krawędzie:  $(x \rightarrow y) \rightarrow (y \rightarrow p(y))$ , gdzie  $p(y)$  jest poprzednikiem  $y$  w drzewie

$T$  (przypadek (1));  $(v \rightarrow u) \rightarrow (u \rightarrow w)$ , jeżeli tylko  $(v \rightarrow u)$  i  $(u \rightarrow w)$  są dwiema kolejnymi krawędziami drzewowymi na ścieżce od  $z$  do  $y$  lub od  $z$  do  $x$  (przypadek (3));  $(x \rightarrow p(x)) \rightarrow (y \rightarrow p(y))$ , jeżeli tylko  $x$  i  $y$  nie są w relacji przodek-potomek, tzn.  $x \neq z$  (przypadek (2)). Łatwo teraz zauważać, że między dwiema dowolnymi krawędziami każdego cyklu bazowego w  $G'$  istnieje ścieżka, która implikuje ich przynależność do tej samej spójnej składowej.

cbdo

Dzięki poczynionym obserwacjom możemy sformułować następujący algorytm obliczania dwuspójnych składowych.

#### Krok 1:1:

- 1.1. Znaleźć w grafie  $G$  dowolne drzewo rozpinające  $T$ .
- 1.2. Ponumerować wierzchołki drzewa metodą preorder i utożsamić je z tymi numerami.
- 1.3. Dla każdego wierzchołka  $v$  obliczyć  $nd(v)$ , tzn. liczbę potomków  $v$  w drzewie. Zadanie to można wykonać, obchodząc drzewo metodą postorder i korzystając z następującego równania rekurencyjnego:  $nd(v) = 1 + \sum nd(w)$ , gdzie sumowanie konkuje się po następnikach  $v$  w drzewie. Zauważmy, że wierzchołek  $w$  jest potomkiem  $v$  w  $T$  wtedy i tylko wtedy, gdy  $v \leq w \leq v + nd(v) - 1$ .

#### Krok 2:

Dla każdego wierzchołka  $v$  obliczyć  $low(v)$  i  $high(v)$ , gdzie  $low(v)$  ( $high(v)$ ) jest najmniejszym (największym) wierzchołkiem  $w$ , takim że  $w = v$  lub  $w$  jest połączony z potomkiem  $v$  krawędzią niedrzewową. Formalnie:

$$low(v) = \min(\{v\} \cup \{w: v \rightarrow w \text{ jest krawędzią niedrzewową}\} \cup \{low(u): u \text{ jest następnikiem } v \text{ w } T\})$$

$$high(v) = \max(\{v\} \cup \{w: v \rightarrow w \text{ jest krawędzią niedrzewową}\} \cup \{high(u): u \text{ jest następnikiem } v \text{ w } T\}).$$

Funkcje  $low$  i  $high$  oblicza się podobnie jak  $nd$  – obchodząc  $T$  metodą postorder.

#### Krok 3:

Skonstruować krawędzie grafu  $G'$  według podanych reguł 1-3.

##### Reguła 1:

Dla każdej niedrzewowej krawędzi  $v \rightarrow w$ , takiej że  $v < w$ , dodać do  $G'$  krawędź  $(p(w) \rightarrow w) \rightarrow (v \rightarrow w)$ .

##### Reguła 2:

Dla każdej niedrzewowej krawędzi  $v \rightarrow w$ , takiej że  $v + nd(v) \leq w$  (gdzie  $v$  i  $w$  nie są w relacji przodek-potomek), dodać do  $G'$  krawędź  $(p(v) \rightarrow v) \rightarrow (p(w) \rightarrow w)$ .

## 7.3. Silnie spójne składowe i silna orientacja

### Reguła 3:

Dla każdej drzewowej krawędzi  $v \rightarrow w$ , takiej że  $v$  jest poprzednikiem  $w$  w  $T$  i  $v$  nie jest korzeniem ( $v \neq 1$ ), dodać do  $G'$  krawędź  $(p(v) \rightarrow v) \rightarrow (v \rightarrow w)$ , jeżeli tylko  $low(w) < v$  lub  $high(w) \geq v + nd(v)$ .

### Krok 4:

Obliczyć spójne składowe grafu  $G'$ .

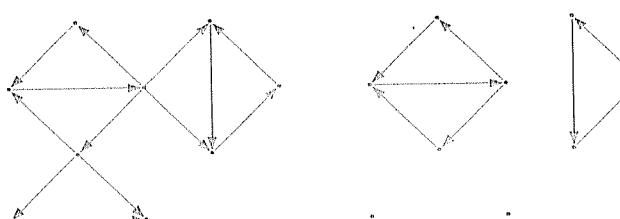
Zauważmy, że ponieważ każda krawędź z  $G$  wnosi co najwyżej dwa połączenia do grafu  $G'$ , to graf ten ma  $m$  wierzchołków i co najwyżej  $2m$  krawędzi. Stąd wynika, że jeśli każdy z czterech kroków można wykonać w czasie liniowym, to cały algorytm działa w czasie  $O(n + m)$ . Działania 1.2 i 1.3 oraz kroki 2 i 3 w oczywisty sposób wykonuje się w czasie  $O(n + m)$ . Złożoność czynności 1.1 i kroku 4 zależy od doboru odpowiednich algorytmów. Jeżeli drzewo rozpinające jest dane, to czynność 1.1 można pomieścić. Spójne składowe i drzewo rozpinające grafu można wyznaczyć w czasie liniowym w sposób alternatywny do przechodzenia w głąb, stosując na przykład przechodzenie wszerz (zob. rozdział 1). Generalnie, jeśli zapewnimy wykonanie czynności 1.1 i kroku 4 w czasie liniowym, to prezentowany algorytm obliczania dwuspójnych składowych będzie działał w czasie  $O(n + m)$ .

Wynika z powyższego, że informację o strukturze grafu można czasami uzyskać z jego dowolnego drzewa rozpinającego, niekoniecznie drzewa przechodzenia w głąb. Przedstawiamy jeszcze jeden algorytm przetwarzający dowolne drzewo rozpinające danego grafu.

## 7.3.

### Silnie spójne składowe i silna orientacja

Kolejnym problemem, który rozważamy, jest problem silnej spójności grafów zorientowanych. Powiemy, że graf zorientowany  $G$  jest **silnie spójny**, jeżeli dla każdych dwóch wierzchołków  $v$  i  $w$  w grafie  $G$  istnieją ścieżki zorientowane z  $v$  do  $w$  oraz z  $w$  do  $v$ . Jeżeli  $G$  nie jest silnie spójny, to możemy podzielić go na co najmniej dwie **silnie spójne składowe**, tzn. maksymalne (w sensie zawierania wierzchołków i krawędzi) podgrafy silnie spójne. Na rysunku 7.4 widać przykładowy graf i jego silnie spójne składowe.



Rys. 7.4. Graf zorientowany i jego silnie spójne składowe

Zanim przystąpimy do rozwiązywania problemu znajdowania silnie spójnych składowych zastanówmy się, czy krawędzie każdego grafu nieorzutowanego można tak zorientować (tzn. zamienić każdą krawędź nieorzutowaną  $u \rightarrow v$  na jedną z dwóch zorientowanych  $u \rightarrow v$  lub  $v \rightarrow u$ ), żeby otrzymać graf silnie spójny. Odpowiedź oczywiście jest negatywna. Grafu niespójnego lub z mostami nie można zorientować w ten sposób. Okazuje się jednak, że jeśli graf nieorzutowany jest spójny i bez mostów, to taka orientacja jest możliwa. Można ją znaleźć w czasie liniowym, używając bardzo podobnych metod do tych z poprzedniego algorytmu dla problemu dwuspójnych składowych.

### PROBLEM 7.3.

Dany jest spójny graf nieorzutowany bez mostów  $G = (V, E)$  i jego drzewo rozpinające  $T$ . Naszym zadaniem jest zorientować krawędzie grafu  $G$  w ten sposób, żeby otrzymany graf był silnie spójny.

Jak poprzednio założymy, że wierzchołki drzewa  $T$  zostały ponumerowane metodą preorder i utożsamione z tymi numerami oraz że obliczyliśmy  $low(v)$  dla każdego wierzchołka  $v$ . Wszystkie te czynności można wykonać w czasie liniowym. Niech  $v \rightarrow w$  będzie krawędzią w grafie  $G$ . Bez straty ogólności będziemy zakładać, że  $v < w$ . Będzie krawędzią w tylu, jeżeli  $v$  i  $w$  są w relacji przodek-potomek i  $v$  nie jest poprzednikiem  $w$ . Jeżeli  $v$  i  $w$  nie są ze sobą w relacji przodek-potomek, to  $v \rightarrow w$  nazywamy krawędzią poprzeczną. Zauważmy, że krawędź  $v \rightarrow w$  jest krawędzią w tylu wtedy i tylko wtedy, gdy jest niedrzewowa i  $w \leq v + nd(v) - 1$ . Zarówno krawędź w tylu, jak i krawędzie poprzeczne są niedrzewowe. Z kolei każda krawędź niedrzewowa jest albo krawędzią w tylu, albo krawędzią poprzeczną. Powyższe rozważania wskazują, że krawędzie niedrzewowe można sklasyfikować w czasie liniowym.

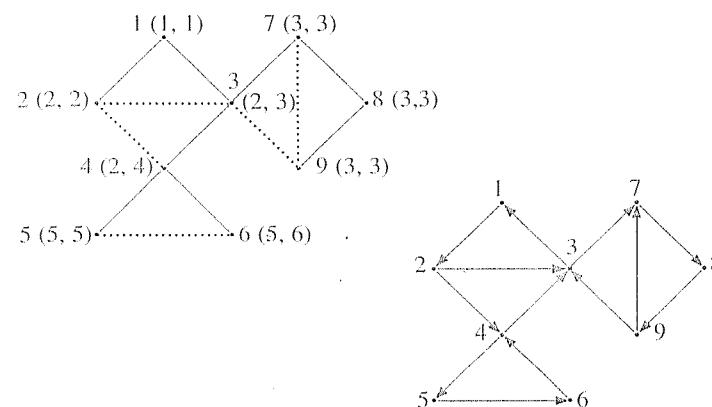
Dla każdego wierzchołka  $v$  zdefiniujemy funkcję  $lowback(v)$  analogicznie do  $low(v)$ , lecz z użyciem tylko krawędzi w tylu:  $lowback(v) = \min(\{v\} \cup \{w: v \rightarrow w \text{ jest krawędzią w tylu}\} \cup \{lowback(u): u \text{ jest następnikiem } v \text{ w } T\})$ . Funkcję  $lowback$  można liczyć jednocześnie z  $low$ .

Żeby przekształcić  $G$  w graf silnie spójny orientujemy krawędzie w następujący sposób:

- jeśli  $v \rightarrow w$  jest krawędzią w tylu ( $v < w$ ), to  $v \rightarrow w$  zamieniamy na  $w \rightarrow v$ ;
- jeśli  $v \rightarrow w$  jest krawędzią poprzeczną ( $v < w$ ), to  $v \rightarrow w$  zamieniamy na  $v \rightarrow w$ ;
- jeśli  $v \rightarrow w$  jest krawędzią drzewową ( $w$  jest następnikiem  $v$ ), to  $v \rightarrow w$  zamieniamy na  $v \rightarrow w$ , o ile tylko  $lowback(w) < w$  lub  $low(w) \geq w$ , a w przeciwnym razie na  $w \rightarrow v$ .

Na rysunku 7.5 widać graf z obliczonymi funkcjami  $low$  i  $lowback$  oraz jego orientację.

Ponieważ funkcje  $low$  i  $lowback$  można obliczyć w czasie liniowym ( $O(m)$ ), graf  $G$  można następnie zorientować w czasie liniowym. Poniżej udowodnimy, że otrzymany w ten sposób graf jest silnie spójny, a zatem, że zachodzi twierdzenie Tarjana-Vishkina [TV].



Rys. 7.5. a) Graf spójny bez mostów, drzewo rozpinające, obliczone funkcje  $low$  i  $lowback$  (para liczb w nawiasach); b) graf silnie spójny

### Twierdzenie 7.2.

Graf  $G'$ , otrzymany z  $G$  zgodnie z regułami a-c, jest grafem silnie spójnym.

**Dowód:** Żeby udowodnić, że otrzymany graf zorientowany jest rzeczywiście silnie spójny, wystarczy wykazać istnienie ścieżek (zorientowanych) z wierzchołka 1 do każdego innego wierzchołka grafu oraz z każdego wierzchołka do 1. Wykażemy tylko, że do każdego wierzchołka prowadzi ścieżka z 1. Dowód drugiej części jest analogiczny do dowodu pierwszej i pozostawiamy go Tobie, jako zadanie. Przypomnijmy, że wierzchołki są utożsamiane z ich numerami otrzymanymi metodą preorder. Dowód przeprowadzimy metodą indukcji względem numerów wierzchołków. Oczywiście 1 jest osiągalny ścieżką o długości 0. Rozważmy wierzchołek  $v > 1$  i założmy, że do wierzchołków 1, 2, ...,  $v-1$  prowadzą ścieżki z 1. Niech  $u$  będzie poprzednikiem  $v$  w drzewie ( $u < v$ ). Jeśli krawędź  $u \rightarrow v$  jest zorientowana od  $u$  do  $v$ , to ścieżka z 1 do  $u$  rozszerzona o krawędź  $u \rightarrow v$  prowadzi do  $v$ . Założymy zatem, że  $u \rightarrow v$  jest zorientowana od  $v$  do  $u$ . Na mocy reguły (c)  $lowback(v) \geq v$  (\*) oraz  $low(v) < v$ . Stąd wynika, że musi istnieć krawędź poprzeczna łącząca pewnego potomka  $x$  wierzchołka  $v$  z  $low(v)$ . Krawędź ta jest zorientowana od  $low(v)$  do  $x$  (reguła (b)). Ponieważ  $low(v) < v$ , na mocy założenia indukcyjnego do  $low(v)$  prowadzi ścieżka z 1 i dalej krawędź  $low(v) \rightarrow x$  można dotrzeć do  $x$ . Pokażemy teraz, że istnieje ścieżka zorientowana z  $x$  do  $v$  złożona z krawędzi drzewowych lub z krawędzi w tylu. Założymy przeciwnie. Niech  $y$ , gdzie  $y \neq v$ , będzie najbliższym przodkiem  $x$  i niech jednocześnie będzie potomkiem  $v$ , do którego prowadzi taka ścieżka. W takim wypadku krawędź  $p(y) \rightarrow y$  musi być zorientowana od  $p(y)$  do  $y$ . Ponieważ  $low(y) = low(x) = low(v) < v \leq y$ , to zgodnie z regułą (c)  $lowback(y) < y$ . Niech  $z$  będzie takim potomkiem  $y$ , że  $z \rightarrow lowback(y)$  jest krawędzią w tylu. Zgodnie

z regułami (c) i (a), wszystkie krawędzie drzewowe na ścieżce od  $y$  do  $z$  są zorientowane od poprzednika do następnika, a krawędź w tył  $z$  —  $lowback(y)$  — od  $z$  do  $lowback(y)$ . Zauważmy teraz, że  $y$  jest tak dobrany, że  $lowback(y) < v$ . Stąd wynika, że  $lowback(v) < v$ , wbrew założeniu (\*), czyli istnieje ścieżka zorientowana z  $x$  do  $v$ , co implikuje istnienie ścieżki z  $1$  do  $v$ .

cbdo

Jeśli drzewo rozpinające jest drzewem przechodzenia w głąb, to krawędzie niedrzewowe są tylko krawędziami w tył. W takim wypadku  $lowback(w) = low(w)$  dla każdego wierzchołka  $w$ . Stąd każdą krawędź drzewową  $v \rightarrow w$ , gdzie  $v < w$ , zamieniamy zawsze na  $v \rightarrow w$ , a każdą krawędź  $v \rightarrow w$  w tył tak jak poprzednio, tzn. na  $w \rightarrow v$ .

Przystąpimy teraz do rozwiązywania problemu znajdowania silnie spójnych składowych.

**PROBLEM 7.4**

Dany jest graf zorientowany  $G = (V, E)$ . Mamy obliczyć funkcję  $S: V \rightarrow \{1, 2, \dots, n\}$ , taką że dwa wierzchołki  $v$  i  $w$  należą do tej samej silnie spójnej składowej wtedy i tylko wtedy, gdy  $S(v) = S(w)$ .

Graf  $G$  będziemy reprezentować za pomocą list sąsiedztwa. Z każdym wierzchołkiem  $v$  wiążemy dwie listy  $L^+(v)$  i  $L^-(v)$ . Są to – odpowiednio – lista wierzchołków, do których prowadzą krawędzie o początku w  $v$ , oraz lista wierzchołków będących początkami krawędzi o końcach w  $v$ . Grafy zorientowane można przehodzić w głąb podobnie jak grafy niezorientowane. Przechodzenie nazywamy **przechodzeniem w przód**, gdy po krawędziach przechodzi się zgodnie z ich zwrotami, a **przechodzeniem w tył**, gdy po krawędziach przechodzi się przeciwnie do ich zwrotów.

Łatwiejszym zadaniem do rozwiązywania od znajdowania silnie spójnych składowych jest sprawdzanie, czy dany graf jest silnie spójny. W tym celu wystarczy dla dowolnego wierzchołka  $v$  sprawdzić, czy każdy wierzchołek  $u$  w grafie jest osiągalny z  $v$  (tzn. czy istnieje ścieżka zorientowana od  $v$  do  $u$ ) i czy z każdego wierzchołka  $u$  można osiągnąć  $v$ . W pierwszym przypadku można to zrobić, przechodząc graf w przód, a w drugim – przechodząc go w tył, za każdym razem rozpoczynając przechodzenie od  $v$ . Jeśli w obu tych przypadkach wszystkie wierzchołki grafu zostaną odwiedzone, to graf jest silnie spójny. Złożoność tego algorytmu jest oczywiście liniowa.

Powyższą ideę wykorzystamy teraz do znajdowania silnie spójnych składowych. Posłużymy się dwiema procedurami rekurencyjnymi *search-forward* oraz *search-back*. Pierwsza z nich powoduje przechodzenie grafu  $G$  w przód, z jednoczesnym obliczaniem dla każdego wierzchołka  $v$  numeru w kolejności odwiedzania, czyli  $nr(v)$ , i liczby potomków w poddrzewie przechodzenia w przód o korzeniu w  $v$ , czyli  $nd(v)$ . W procedurze wykorzystujemy zmienną globalną  $l$ , na której jest pamiętaana liczba dotychczas odwiedzonych wierzchołków. Początkowo  $l = 0$ . Podobnie jak przy przechodzeniu w głąb grafu niezo-

orientowanego zakładamy, że z każdym wierzchołkiem  $v$  są związane dwa pola: *visited-f(v)* i *current-f(v)*. Pierwsze pole służy do zaznaczania stanu wierzchołka przy przeszukiwaniu w przód. Drugie pole jest wskaźnikiem do takiego pierwszego wierzchołka w na liście  $L^t(v)$ , w wypadku którego krawędź  $v \rightarrow w$  nie była jeszcze odwiedzona.

```

procedure search-forward(v : wierzchołek);
var
w : wierzchołek;
begin
  visited-f(v) := true; {zamarkuj v jako wierzchołek odwiedzony
  przy przechodzeniu w przód}
  l := l + 1; nr(v) := l; nd(v) := 1;
  while current-f(v) ≠ nil do
  begin
    w := wierzchołek na liście  $L^+(v)$ , na który
    wskazuje current-f(v);
    if not visited-f(w) {w nie był odwiedzony przy
    przechodzeniu w przód} then
    begin
      {v → w zaliczamy do krawędzi drzewa przechodzenia
      w przód o korzeniu w wierzchołku, od którego
      rozpoczęliśmy poszukiwanie}
      search-forward(w); nd(v) := nd(v) + nd(w)
    end;
    przesuń wskaźnik current-f(v) do następnego wierzchołka
    na liście  $L^+(v)$ 
  end
end;

```

Jeśli graf nie jest silnie spójny, to wywołanie `search-forward(1)` może nie spowodować odwiedzenia wszystkich wierzchołków grafu. Żeby zagwarantować poprawne przejście całego grafu, wystarczy wykonać następujące instrukcje:

```

l := 0;
for v := 1 to n do
begin
    visited-f(v) := false;
    current-f(v) := pierwszy wierzchołek na liście  $L^+(v)$ 
end;
for v := 1 to n do
    if not visited-f(v) then search-forward(v);

```

Bez straty ogólności będziemy teraz utożsamiać każdy wierzchołek  $v$  z jego numerem  $nr(v)$ , tzn. przyjmiemy, że  $v = nr(v)$ . Funkcję  $S$  obliczymy w ten sposób, że wartość  $S(v)$

będzie równa najmniejszemu wierzchołkowi w silnie spójnej składowej zawierającej  $v$ . Zauważmy, że przy przechodzeniu grafu w przód wierzchołek najmniejszy ze wszystkich wierzchołków tej samej silnie spójnej składowej jest odwiedzany jako pierwszy. Niech  $u$  będzie właśnie takim wierzchołkiem, tzn.  $S(u) = u$ . Wszystkie wierzchołki silnie spójnej składowej zawierającej  $u$  leżą w poddrzewie przechodzenia w przód o korzeniu w tym wierzchołku. Oczywiście, w tym poddrzewie mogą być także wierzchołki spoza silnie spójnej składowej, do której należy  $u$ . Żeby wyznaczyć tylko te wierzchołki poddrzewa, z których w grafie  $G$  prowadzą ścieżki do  $u$ , wystarczy sprawdzić, które z nich można odwiedzić z  $u$ , przechodząc graf w tył. Zadanie to można zrealizować za pomocą procedury *search-back*. W jej opisie znaczenia pól *visited-b(v)* oraz *current-b(v)* są podobne do, odpowiednio, *visited-f(v)* i *current-f(v)*.

```

procedure search-back(v : wierzchołek);
var
  w : wierzchołek;
begin
  S(v) := u;
  visited-b(v) := true {v markujemy jako wierzchołek odwiedzony
  przy przechodzeniu w tył}
  while current-b(w) ≠ nil do
    begin
      w := wierzchołek na liście  $L^-(v)$ , na który
      wskazuje current-b(v);
      if  $u \leq w < u + nd(u)$  {w jest w poddrzewie przechodzenia
      w przód o korzeniu u}
      and not visited-b(w) {w nie był odwiedzony przy
      przechodzeniu w tył}
      then search-back(w);
      przesuń wskaźnik current-b(v) do następnego wierzchołka
      na liście  $L^-(v)$ 
    end
  end;
end;

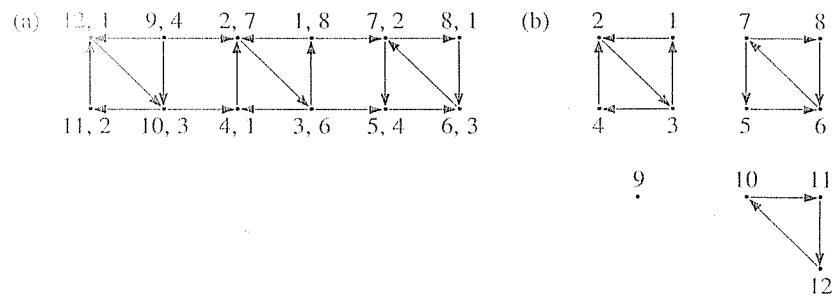
```

Silnie spójne składowe możemy teraz wyznaczyć w następujący sposób:

```

for u := 1 to n do
begin
  visited-b(u) := false;
  current-b(u) := pierwszy wierzchołek na liście  $L^-(u)$ 
end;
for u := 1 to n do
  if not visited-b(u) {u nie był odwiedzony przy przechodzeniu
  w tył} then
    search-back(u);

```



Rys. 7.6. a) Graf zorientowany z wierzchołkami ponumerowanymi (pierwsza liczba w każdej parze) za pomocą procedury *search-forward*; druga liczba w parze jest równa liczbie wierzchołków w poddrzewie przechodzenia w przód o korzeniu w danym wierzchołku; b) silnie spójne składowe

Złożoność tego algorytmu, podobnie jak poprzednich, wynosi  $O(n + m)$ . Wynika to z faktu, że w opisany algorytmie po każdej krawędzi przechodzi się dwa razy: raz w procedurze *search-forward* i raz w procedurze *search-back*. Działanie powyższego algorytmu przedstawiamy na rysunku 7.6.

## 7.4. Cykle Eulera

Rozwiążemy teraz problem znajdowania cykli Eulera w grafach. Powiemy, że cykl  $C$  w grafie  $G$  jest **cyklem Eulera**, jeśli każda krawędź grafu występuje w nim tylko raz. Przedstawimy najpierw algorytm znajdowania cykli Eulera w grafach zorientowanych, a następnie sprowadzimy zadanie dla grafów niezorientowanych do rozwiązanego zadania dla grafów zorientowanych.

Niech  $G = (V, E)$  będzie grafem zorientowanym. Istnieje bardzo proste kryterium, na podstawie którego można stwierdzić, czy graf  $G$  ma cykl Eulera.

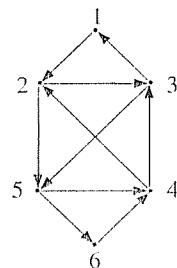
Graf  $G$  ma cykl Eulera wtedy i tylko wtedy, gdy jest spójny (każda krawędź z  $E$  traktujemy jako niezorientowaną) i liczba krawędzi wchodzących do każdego wierzchołka  $v$  (o końcu w  $v$ ) jest równa liczbie krawędzi go opuszczających (o początku w  $v$ ).

Jeśli chodzi o grafy niezorientowane, to oprócz spójności żąda się, żeby liczba sąsiadów każdego wierzchołka była parzysta. Dowody powyższych faktów nie są specjalnie skomplikowane, a poza tym można je znaleźć w prawie każdym podręczniku z teorii grafów. Więcej, wynikają one także bezpośrednio z algorytmu, który za chwilę przedstawimy.

Ponieważ warunki na istnienie cyklu Eulera łatwo sprawdza się w czasie liniowym, w dalszych rozważaniach założymy, że są spełnione.

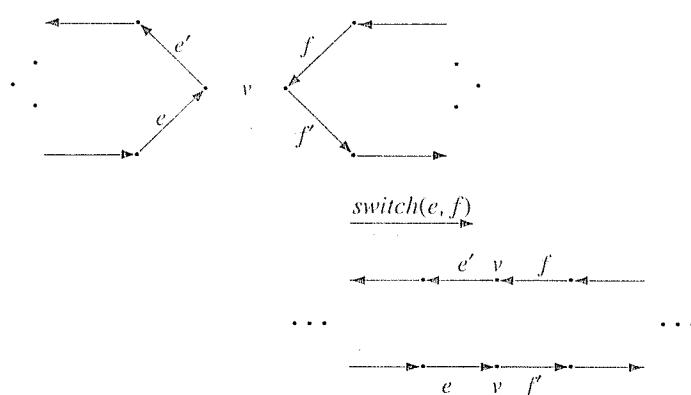
#### PROBLEM 7.5.

Dany jest spójny graf zorientowany  $G$ , taki że dla każdego wierzchołka  $v$  liczba krawędzi wchodzących do  $v$  jest równa liczbie krawędzi z niego wychodzących. Dla każdej krawędzi  $e$  mamy obliczyć  $\text{next}(e)$  – następną po  $e$  krawędź w pewnym cyklu Eulera  $C$ . Na rysunku 7.7 widać graf mający cykl Eulera.



Rys. 7.7. Grafi mający cykl Eulera, na przykład 1, 2, 3, 5, 6, 4, 2, 5, 4, 3, 1

Zanim przystąpimy do opisu algorytmu znajdowania cyklu Eulera, zdefiniujmy operację  $\text{switch}$ , która umożliwi łączenie dwóch rozłącznych krawędziowo cykli w jeden. Niech  $C_1$  i  $C_2$  będą rozłącznymi krawędziowo cyklami o wspólnym wierzchołku  $v$  ( $C_1$  i  $C_2$  mogą mieć więcej niż jeden wspólny wierzchołek). Założymy, że dla każdej krawędzi  $e$  z  $C_1$  ( $C_2$ ) krawędź  $\text{next}(e)$  występuje po  $e$  w  $C_1$  ( $C_2$ ). Niech  $e$  i  $f$  będą krawędziami, wchodzącej do  $v$  w cyklu  $C_1$  i wchodzącej do  $v$  w  $C_2$ . Procedura  $\text{switch}(e, f)$  umożliwia łączenie cykli  $C_1$  i  $C_2$  w jeden z ustanowieniem  $\text{next}(e)$  jako kolejnej krawędzi po  $f$ , a  $\text{next}(f)$  jako kolejnej krawędzi po  $e$  w nowo powstającym cyklu (rys. 7.8).



Rys. 7.8. Łączenie cykli

Formalnie procedurę  $\text{switch}$  definiuje się następująco:

```
procedure switch( $e, f$  : krawędź);
var
   $e'$  : krawędź;
begin
   $e' := \text{next}(e)$ ;  $\text{next}(e) := \text{next}(f)$ ;  $\text{next}(f) := e'$ 
end;
```

Operację  $\text{switch}$  wykorzystamy w następującym algorytmie obliczania cyklu Eulera.

#### KROK 1:

Znaleźć w  $G$  dowolny cykl  $C_0$  bez powtarzających się krawędzi.

#### KROK 2:

Usunąć z  $G$  krawędzie cyklu  $C_0$ .

Niech  $G_1, \dots, G_k$  będą spójnymi składowymi tak otrzymanego grafu, o co najmniej 2 wierzchołkach, a  $v_1, \dots, v_k$  wierzchołkami  $C_0$  należącymi, odpowiednio, do  $G_1, \dots, G_k$ . Zauważmy, że każdy z otrzymanych grafów ma cykl Eulera, ponieważ usuwanej krawędzi wchodzącej do wierzchołka odpowiada zawsze usuwana krawędź opuszczająca go.

#### KROK 3:

Znaleźć rekurencyjnie cykle Eulera  $C_1, \dots, C_k$  w grafach  $G_1, \dots, G_k$ .

#### KROK 4:

Za pomocą operacji  $\text{switch}$  połączycy  $C_0$  z  $C_1, \dots, C_k$  w jeden wspólny cykl.

Żeby efektywnie zrealizować powyższy algorytm, potrzebujemy właściwych struktur danych. Zakładamy, że z każdym wierzchołkiem  $v$  jest związana lista  $L^i(v)$  wierzchołków, do których prowadzą krawędzie o początku w  $v$ . W trakcie wykonywania algorytmu sąsiedzi  $v$  są przeglądani kolejno, od pierwszego do ostatniego na liście. Zmienna  $\text{current}(v)$  wskazuje na pierwszy wierzchołek na liście sąsiedztwa  $v$ , który nie był jeszcze odwiedzony z  $v$ . Początkowo  $\text{current}(v)$  wskazuje na pierwszy wierzchołek na liście  $L^i(v)$ . Będziemy uważali, że rozważany właśnie graf (na danym poziomie rekursji) zawiera tylko te krawędzie, które prowadzą od każdego wierzchołka  $v$  do wierzchołków z jego listy sąsiedztwa, położonych między  $\text{current}(v)$  a końcem listy. Cykl Eulera konstruujemy za pomocą procedury rekurencyjnej  $\text{Euler-cycle}(v)$ . Dzięki niej znajdziemy cykl Eulera w spójnej składowej bieżącego grafu, która zawiera  $v$ . Dla każdej krawędzi  $e$  pamiętamy nie tylko  $\text{next}(e)$ , ale i krawędź  $\text{pred}(e)$ , poprzedzającą  $e$  w konstruowanym cyklu.

```
procedure Euler-cycle( $v$  : wierzchołek);
var
   $x, w$  : wierzchołek;
   $e, e', f, f'$  : krawędź;
```

```

begin
  {znajdowanie cyklu  $C_0$ }
  w := wierzchołek, na który wskazuje  $current(v)$ ;
  {usunięcie krawędzi  $v \rightarrow w$  z grafu}
   $current(v) :=$  wskaźnik do następnego wierzchołka na liście
   $L^+(v)$ ;
  e :=  $v \rightarrow w$ ; f := e;
repeat
  x := w; w := wierzchołek, na który wskazuje  $current(x)$ ;
   $current(x) :=$  wskaźnik do następnego wierzchołka na
  liście  $L^+(x)$ ;
  next(f) :=  $x \rightarrow w$ ; pred( $x \rightarrow w$ ) := f; f :=  $x \rightarrow w$ 
until w = v;
next(f) := e; pred(e) := f;
{znajdowanie cykli w spójnych składowych i łączenie ich z  $C_0$ }
x := v; e := f;
repeat
  e' := next(e); {e' – kolejna krawędź po e w  $C_0$ }
  if  $current(x) \neq \text{nil}$  then
    begin
      {x należy do spójnego podgrafa zawierającego więcej
      niż jeden wierzchołek}
      y := wierzchołek, na który wskazuje  $current(x)$ ;
      f' :=  $x \rightarrow y$ ; Euler-cycle(x); f := pred(f');
      switch(e, f')
    end;
    e := e';
    x := końcowy wierzchołek krawędzi e
  until x = v
end;

```

Latwo zauważyc, że złożoność algorytmu zależy od łącznej sumy długości wszystkich cykli generowanych w czasie pierwszego wykonania instrukcji „powtarzaj” (**repeat**) w procedurze *Euler-cycle*. Ponieważ cykle te są krawędziowo rozłączne, ich łączna długość wynosi  $m$ . Wynika stąd, że za pomocą tego algorytmu możemy znaleźć cykl Eulera w grafie zorientowanym w czasie liniowym  $O(n + m)$ .

Powstaje naturalne pytanie, czy ten algorytm można wykorzystać do znajdowania cykli Eulera w grafach niezorientowanych. Graf niezorientowany ma cykl Eulera wtedy i tylko wtedy, gdy jest spójny i liczba sąsiadów każdego wierzchołka jest parzysta. Pokażemy, że krawędzie takiego grafu można zorientować w ten sposób, żeby liczba krawędzi wchodzących do każdego wierzchołka była równa liczbie krawędzi z niego wychodzących. W tym celu zamieniamy każdą krawędź niezorientowaną  $u \rightarrow v$  na dwie zorientowane:  $u \rightarrow v$  i  $v \rightarrow u$ , po czym na krawędziach zorientowanych budujemy cykle w taki

sposób, żeby dla każdego cyklu  $C$  istniał odpowiadający mu cykl  $C'$ , zawierający te same krawędzie, lecz zorientowane przeciwnie. Żeby znaleźć cykl Eulera, wystarczy pozostawić tylko krawędzie zorientowane jednego cyklu z każdej pary odpowiadających sobie cykli. W celu zbudowania takich cykli postępujemy w następujący sposób. Niech  $u_1, \dots, u_k$  będą sąsiadami wierzchołka  $v$  w  $G$  (gdzie  $u_i$  jest  $i$ -tym wierzchołkiem na liście sąsiadów  $v$ ). Dla każdego nieparzystego  $i$  następną krawędzią w cyklu po  $u_i \rightarrow v$  będzie *krawędź*  $v \rightarrow u_{i+1}$ , a następną po  $u_{i+1} \rightarrow v$  krawędź  $v \rightarrow u_i$ . Otrzymane w ten sposób cykle spełniają żądane warunki. Powyższą operację można w oczywisty sposób przeprowadzić w czasie liniowym. Do otrzymanego grafu stosujemy wcześniej omówiony algorytm obliczania cyklu Eulera w grafach zorientowanych.

## 7.5.

### 5-kolorowanie grafów planarnych

Graf  $G = (V, E)$  jest **grafem planarnym**, gdy można go przedstawić na płaszczyźnie w taki sposób, że dwóm różnym wierzchołkom odpowiadają dwa różne punkty płaszczyzny, a każdej krawędzi  $e$  – krzywa zwyczajna o końcach w punktach odpowiadających wierzchołkom incydentnym z  $e$ . Ponadto dla pary różnych krawędzi  $e$  i  $f$ , jeśli  $e$  i  $f$  nie sąsiadują ze sobą, to przyporządkowane im krzywe nie mają punktów wspólnych. Jeśli są to krawędzie sąsiednie, to mają tylko jeden punkt wspólny, który jest jednocześnie punktem odpowiadającym wspólnemu wierzchołkowi  $e$  i  $f$ .

Wiadomo, że liczba krawędzi  $m$  w każdym grafie planarnym zależy liniowo od liczby jego wierzchołków  $n$ . Dokładniej, jeśli  $n \geq 3$ , to  $m \leq 3n - 6$ . Z powyższego faktu wynika natychmiast, że pełny graf 5-wierzchołkowy nie jest planarny. Prawdziwy jest podany tu lemat.



#### Lemat 7.1.

Każdy graf planarny ma wierzchołek o stopniu co najwyżej 5.

**Dowód:** Bez straty ogólności możemy rozważać tylko grafy spójne i o co najmniej 6 wierzchołkach. Jeżeli każdy wierzchołek w takim grafie miałby stopień co najmniej 6, to zawierałby co najmniej  $3n$  krawędzi, a przeczyłoby temu, co powiedzieliśmy wcześniej o grafach planarnych.

ebdo

Mówimy, że graf  $G$  jest **k-kolorowalny**, jeżeli każdemu wierzchołkowi możemy przypiąć jeden z  $k$  różnych kolorów w taki sposób, iż żadne dwa sąsiadne wierzchołki nie otrzymują tego samego koloru.

Dosyć łatwo można udowodnić, że każdy graf planarny jest 5-kolorowalny. Przez ponad sto lat wielu badaczy staralo się dociec, czy każdy graf planarny (a dokładniej – każda

mapę narysowaną na kartce papieru) można pokolorować 4 kolorami. W 1976 r. Kenneth Appel i Wolfgang Haken podali pozytywną odpowiedź na to pytanie. Ponieważ jednak dowód tego faktu i wynikający z niego algorytm 4-kolorowania wykracza poza zakres tej książki, wróćmy do problemu 5-kolorowania.



#### Twierdzenie 7.4.

Każdy graf planarny jest 5-kolorowalny.

**Dowód:** Dowód przeprowadzimy metodą indukcji względem liczby wierzchołków grafu planarnego  $G$ . Jeżeli  $n \leq 5$ , to twierdzenie jest oczywiście prawdziwe. Założymy, że  $n > 5$  i twierdzenie jest prawdziwe dla każdego grafu planarnego z mniejszą niż  $n$  liczbą wierzchołków. Na mocy lematu 7.1 istnieje w  $G$  wierzchołek  $v$  o stopniu co najwyżej 5. Rozważmy dwa przypadki.

**PRZYPADEK 1:** Stopień wierzchołka  $v$  jest mniejszy niż 5.

Niech  $G'$  będzie grafem otrzymanym z  $G$  przez usunięcie wierzchołka  $v$  i krawędzi z nim incydentnych. Graf  $G'$  jest oczywiście planarny i z założenia indukcyjnego 5-kolorowalny. Rozważmy 5-pokolorowanie  $G'$ . Na pokolorowanie sąsiadów wierzchołka  $v$  używa się w  $G'$  co najwyżej 4 kolorów. Stąd wynika, że można rozszerzyć pokolorowanie  $G'$  do  $G$ , kolorując  $v$  kolorem nie wykorzystanym do pokolorowania jego sąsiadów.

**PRZYPADEK 2:** Stopień wierzchołka  $v$  jest równy 5.

Zauważmy, że w takim przypadku istnieją dwaj sąsiedzi  $v$ , powiedzmy  $u$  i  $w$ , nie połączeni krawędzią w  $G$ . W przeciwnym bowiem razie graf indukowany przez zbiór sąsiadów  $v$  byłby pełnym grafem 5-wierzchołkowym i stąd – wbrew założeniu –  $G$  nie byłby planarny. Niech  $G'$  będzie grafem otrzymanym z  $G$  przez usunięcie  $v$  i utożsamienie  $u$  z  $w$ . utożsamienie polega na usunięciu wierzchołków  $u$  i  $w$  z grafu i zastąpieniu ich przez wierzchołek  $x$ , którego sąsiadami stają się wszyscy sąsiedzi usuniętych wierzchołków. Otrzymany graf jest oczywiście planarny i z założenia indukcyjnego 5-kolorowalny. Rozważmy dowolne 5-pokolorowanie  $G'$ . Można je rozszerzyć do 5-pokolorowania  $G$  w następujący sposób: wierzchołki  $u$  i  $w$  dziedziczą kolor wierzchołka  $x$  (pamiętajmy, że nie są sąsiadami w  $G$ ). Zauważmy, że na pokolorowanie sąsiadów  $v$  ponownie używa się co najwyżej 4 kolorów. Wolnym kolorem kolorujemy  $v$ .

ebdo

Na podstawie dowodu ostatniego twierdzenia otrzymujemy już pewien algorytm kolorowania grafów planarnych 5 kolorami. Pokażemy, w jaki sposób można zrealizować ten algorytm, żeby działał w czasie liniowym. Założmy, że wejściowy graf planarny jest dany przez listy sąsiedztwa (z pewnymi modyfikacjami, o których piszemy dalej). Naj-

bardziej pracochłonną operację opisaną w dowodzie-algorytmie jest utożsamianie wierzchołków. W operacji tej trzeba z dwóch list sąsiedztwa wierzchołków  $u$  i  $w$  stworzyć jedną listę sąsiadów  $x$ . Ponieważ na takiej liście wierzchołki nie mogą się powtarzać, łączenie dwóch list wymaga co najmniej czasu proporcjonalnego do długości krótszej z nich, która w pesymistycznym przypadku może być rzędu liczby wszystkich wierzchołków w grafie. Wynika stąd, że bezpośrednią realizację dowodu twierdzenia mogłaby dać algorytm o koszcie kwadratowym. Pokażemy jednak, że zawsze można tylko utożsamiać wierzchołki z krótkimi listami sąsiedztwa. W tym celu wykorzystamy taki oto lemat.



#### Lemat 7.2.

Niech  $G$  będzie grafem planarnym, w którym każdy wierzchołek ma stopień co najmniej 5. Istnieje wówczas wierzchołek o stopniu 5, mający dwóch nie połączonych ze sobą sąsiadów o stopniach  $\leq 11$ .

**Dowód:** **Wystarczy pokazać, że w  $G$  istnieje wierzchołek  $v$  o stopniu 5, mający co najwyżej jednego sąsiada o stopniu większym niż 11, ponieważ graf indukowany przez  $v$  i dowolnych jego 4 sąsiadów nie może być pełnym grafem 5-wierzchołkowym.**

Oznaczmy przez  $n_d$  liczbę wierzchołków w grafie o stopniu  $d$ . Ponieważ

$$2m = \sum_{d \geq 5} dn_d \text{ i } m \leq 3n - 6 = 3 \left( \sum_{d \geq 5} n_d \right) - 6$$

mamy

$$n_5 \geq \sum_{d \geq 7} (d - 6)n_d + 12$$

Założymy, że każdy wierzchołek stopnia 5 ma co najmniej dwóch sąsiadów o stopniu większym niż 11. Wówczas

$$2n_5 \leq \sum_{d \geq 12} dn_d$$

Z powyższych nierówności wynika, że

$$\sum_{d \geq 7} (2d - 12)n_d + 24 \leq \sum_{d \geq 12} dn_d$$

$$\sum_{7 \leq d \leq 11} (2d - 12)n_d + \sum_{d \geq 12} (d - 12)n_d + 24 \leq 0$$

a to jest sprzeczność.

ebdo

Algorytm 5-kolorowania działa w dwóch fazach.

#### FAZA 1:

Graf wejściowy jest iteracyjnie redukowany do grafu planarnego o co najwyżej 5 wierzchołkach, który następnie w oczywisty sposób jest kolorowany 5 kolorami. W każdej iteracji należy wykonać następujące działania: (1) jeśli w grafie bieżącym jest wierzchołek  $v$  o stopniu  $\leq 4$ , to należy zredukować graf w wierzchołku  $v$ , wywołując procedurę  $reduce(v)$ ; (2) w przeciwnym razie należy znaleźć wierzchołek  $v$  o stopniu 5 z dwoma nie połączonymi sąsiadami  $x$  i  $y$  o stopniu  $\leq 11$ , a następnie wykonać  $reduce(v)$  i wywołać procedurę  $identify(x, y)$  utożsamiania  $x$  z  $y$ .

Każdy wierzchołek spełniający (1) lub (2) nazywamy **redukowalnym**. Graf wejściowy  $G$  jest reprezentowany za pomocą dwukierunkowych list sąsiedztwa. Dodatkowo zakłada się, że dwa wystąpienia tej samej krawędzi  $u$  —  $w$  na listach sąsiedztwa  $u$  (reprezentowanej przez  $w$ ) i  $w$  (reprezentowanej przez  $u$ ) są połączone, tzn. z wystąpieniem  $w$  na liście  $u$  jest dane dowiązanie do wystąpienia  $u$  na liście  $w$  i odwrotnie — z wystąpieniem  $u$  na liście  $w$  jest dane dowiązanie do wystąpienia  $w$  na liście  $u$ . Te dodatkowe dowiązania nazywamy **krzyżowymi**. Zakłada się także, że każdy element listy sąsiedztwa ma wskaźnik do jej początku. Niech  $L(v)$  będzie listą sąsiedztwa  $v$ . Do realizacji fazy 1 używa się dwóch kolejek  $Q$  i  $R$  (realizowanych także przez listy dwukierunkowe), w których pamięta się wierzchołki redukowalne. W kolejce  $Q$  są przechowywane wszystkie wierzchołki o stopniu  $\leq 4$ , a w kolejce  $R$  — wszystkie wierzchołki redukowalne o stopniu równym 5. Z lematu 7.2 wynika, że zawsze ktośś z kolejek,  $Q$  lub  $R$ , jest niepusta. W realizacji algorytmu wykorzystamy także stos  $S$  do pamiętania wierzchołków usuwanych z grafu wraz z ich bieżącymi listami sąsiedztwa i do pamiętania piątek  $[t, x, L(x), y, L(y)]$ , gdzie  $t$  jest wierzchołkiem powstającym przez utożsamienie  $x$  z  $y$ .

Jeśli kolejka  $Q$  jest niepusta, to wierzchołek, w którym nastąpi redukcja grafu, pobiera się z  $Q$ . W przeciwnym razie pobiera go się z  $R$ .

Pozostaje jeszcze opisać procedury  $reduce$  oraz  $identify$ . Najpierw jednak zdefiniujemy wykorzystywane w nich operacje na kolejkach:

- (a)  $insert(u, P)$ : if  $u \notin P$  then wstaw  $u$  do  $P$ ;
- (b)  $delete(u, P)$ : if  $u \in P$  then usuń  $u$  z  $P$ ;

Jeśli przyjmiemy, że dany jest wskaźnik do wystąpienia wierzchołka  $u$  w kolejce  $P$ , to obie operacje —  $insert$  i  $delete$  — można wykonać w czasie stałym.

Wykorzystana zostanie także pomocnicza procedura  $consider$ , umożliwiająca właściwe przetwarzanie wierzchołków redukowalnych ( $deg(u)$  oznacza stopień wierzchołka  $u$ ). Oto ta procedura:

```
procedure consider(u);
  case
    deg(u) ≤ 4 : delete(u, R); insert(u, Q);
    deg(u) = 5 : if u jest redukowalny then insert(u, R) else
      delete(u, R);
    deg(u) = 11 : for each x — sąsiad u do
      if deg(x) = 5 and x jest redukowalny then
        insert(x, R)
  endcase
end;
```

Procedura  $consider$  działa w czasie stałym. Zdefiniujemy teraz procedury  $reduce$  i  $identify$ .

```
procedure reduce(v);
  push({v, L(v)}, S);
  usuń v z grafu;
  for each u ∈ L(v) do consider(u)
end;

procedure identify(x, y);
  utwórz nowy wierzchołek t, którego sąsiadami są sąsiedzi x i y;
  push([t, x, L(x), y, L(y)], S);
  usuń x, y z grafu;
  consider(t);
  for each z ∈ L(t) do consider(z)
end;
```

Procedury „usuń” i „utwórz” można łatwo zapisać w taki sposób, żeby listy sąsiedztwa, stopnie wierzchołków oraz kolejki  $Q$  i  $R$  w czasie stałym były właściwie aktualizowane. Ponieważ obie procedury  $reduce(v)$  i  $identify(x, y)$  są wywoływane tylko dla wierzchołków o stopniu co najwyżej 11, czas ich działania wynosi  $O(1)$ . Wynika stąd, że faza 1 może być zrealizowana w czasie liniowym ( $O(n)$ ).

#### FAZA 2:

W tej fazie zdejmuję się kolejne elementy ze stosu  $S$ . Jeśli jest zdejmowany wierzchołek  $v$  (i jego lista sąsiedztwa), to jest on kolorowany kolorem różnym od kolorów jego sąsiadów. Jeśli ze stosu jest zdejmowana piątka  $[t, x, L(x), y, L(y)]$ , to są odtwarzane wierzchołki  $x$  i  $y$ , a następnie jest im przypisywany kolor wierzchołka  $t$ .

Ponieważ faza 2 jest właściwie odwrotnością fazy 1, czas jej działania jest liniowy.

W ten sposób pokazaliśmy, że każdy graf planarny można pokolorować 5 kolorami w czasie liniowym. Zwróćmy jeszcze uwagę, że jeśli chcemy efektywnie manipulować grafem, na przykład usuwać jego wierzchołki i/lub krawędzie, to dowiązania krzyżowe są niezbędne.

## 7.6. Najkrótsze ścieżki i minimalne drzewo rozpinające

Niech  $G = (V, E)$  będzie spójnym grafem nieorientowanym, w którym każdej krawędzi  $e$  jest przypisana dodatnia liczba rzeczywista  $w(e)$ , zwana dalej jej **wagą**. W takim grafie długość ścieżki mierzy się sumą wag krawędzi do niej należących. Podobnie definiuje się koszt drzewa rozpinającego, a mianowicie jako sumę wag jego krawędzi. W tym podrozdziale podamy schemat algorytmu, który zastosujemy dalej do konstruowania algorytmów obliczania

- najkrótszych ścieżek łączących wierzchołki grafu z jednym, wcześniej ustalonym wierzchołkiem;
- drzewa rozpinającego o najmniejszym koszcie.

Algorytm ten służy do tworzenia drzewa rozpinającego danego grafu  $G$ , zakończonego w wybranym wcześniej wierzchołku  $r$  (dany także jako parametr wejściowy). Działa w  $n - 1$  fazach i ma podane tu własności.

- (1) Przed wykonaniem każdej fazy zbiór wierzchołków grafu jest podzielony na dwa zbiory  $L$  i  $R$ . Zbiór  $L$  zawiera zawsze wierzchołek  $r$ .
- (2) Przed wykonaniem  $i$ -tej fazy zbiór  $L$  zawiera  $i$  wierzchołków.
- (3) Dla każdego wierzchołka  $v \in L$ , różnego od  $r$ , jest w konstruowanym drzewie określony poprzednik  $p(v)$  wierzchołka  $v$  oraz liczba  $dist(v)$ , zależna od tego poprzednika. Dla korzenia  $r$  mamy  $p(r) = \text{nil}$ , a  $dist(r) = 0$ . Jeśli dla każdego wierzchołka  $v \in R$  istnieją krawędzie w  $G$  łączące  $v$  z wierzchołkami z  $L$ , to  $p(v)$  jest jednym z tych wierzchołków, a  $dist(v)$  jest liczbą związaną z  $v$ , zależną od  $dist(p(v))$  i  $w(v \rightarrow p(v))$ . Jeżeli  $v$  nie jest połączony z żadnym z wierzchołków z  $L$ , to  $p(v) = \text{nil}$ , a  $dist(v) = \infty$ . W opisie algorytmu symbol  $\infty$  oznacza wartość, co do której zakłada się, że jest większa od każdej liczby rzeczywistej. Jeśli  $v$  jest połączony z wyróżnionym wierzchołkiem  $r$ , to przed pierwszą fazą  $dist(v) = w(v \rightarrow r)$ .
- (4) Każda faza składa się z usunięcia z  $R$  wierzchołka o najmniejszej wartości  $dist$ , wstawienia go do  $L$  oraz (właściwie) modyfikacji  $dist$  i  $p$  dla pewnych wierzchołków z  $R$ .

Oto formalny zapis opisanego schematu tego algorytmu:

```

begin
  L := {x}; R := V \ {x};
  p(x) := nil; dist(x) := 0;
  for v ∈ R do
    if x → v ∈ E then
      begin p(v) := x; dist(v) := w(x → v) end
    else begin p(v) := nil; dist(v) := ∞ end;
  for faza := 1 to n - 1 do

```

```

begin
  u := wierzchołek z R z minimalną wartością dist;
  R := R \ {u}; L := L ∪ {u};
  (*) dla każdego sąsiada u, należącego do R, zmodyfikuj według przyjętej reguły dist oraz p
end;
end;

```

Rodzaj drzewa, jakie otrzymamy, zależy od sprecyzowania reguły (\*). Rozważmy dwa różne warianty.

### WARIANT 1:

Przyjmujemy, że dla każdego wierzchołka  $v \in L$  liczba  $dist(v)$  jest równa długości najkrótszej ścieżki łączącej  $v$  z wyróżnionym wierzchołkiem  $r$ . Zakładamy także, że jeśli  $v \neq r$ , to  $dist(v) = dist(p(v)) + w(v \rightarrow p(v))$ . Jeśli  $v \in R$  i  $dist(v) < \infty$ , to  $dist(v)$  jest długością najkrótszej ścieżki łączącej  $v$  z  $r$  i takiej, że wszystkie wierzchołki na tej ścieżce, z wyjątkiem  $v$ , należą do  $L$ . Krok (\*) ma w tym wypadku następującą postać:

```

for y ∈ {x : x → u ∈ E oraz x ∈ R} do
  if dist(u) + w(u → y) < dist(y) then
    begin p(y) := u; dist(y) := dist(u) + w(u → y) end;

```

Po wykonaniu całego algorytmu dla każdego wierzchołka  $v$  liczba  $dist(v)$  jest długością najkrótszej ścieżki łączącej  $v$  z wyróżnionym wierzchołkiem  $r$ , a ścieżka  $v, p(v), p(p(v)), \dots, r$  jest właśnie taką ścieżką. Algorytm w tej postaci jest znany jako **algorytm Dijkstry**. Jeśli wszystkie wagi są równe 1, to otrzymane drzewo ma następującą własność: w odległości  $k$  w drzewie są wszystkie wierzchołki, których odległość w całym grafie od  $r$  wynosi  $k$ . Takie drzewo nazywa się **drzewem przechodzenia wszerz**. Jak mogliśmy się już przekonać w rozdziale 1, drzewo przechodzenia wszerz można skonstruować za pomocą znacznie prostszego algorytmu.

### WARIANT 2:

W tym wypadku przyjmujemy, że dla każdego wierzchołka  $v \in R$  liczba  $dist(v)$  jest równa minimalnej wadze ze wszystkich wag krawędzi łączących  $v$  z wierzchołkami z  $L$ . Jeśli  $dist(v) < \infty$ , to  $p(v)$  jest tym wierzchołkiem z  $L$ , dla którego  $dist(v) = w(p(v) \rightarrow v)$ . Krok (\*) ma teraz następującą postać:

```

for y ∈ {x : x → u ∈ E oraz x ∈ R} do
  if w(y → u) < dist(y) then
    begin p(y) := u; dist(y) := w(y → u) end;

```

Drzewo, które znajdziemy w tym wypadku jest minimalnym drzewem rozpinającym, tzn. drzewem, w którym suma wag krawędzi jest najmniejsza. Algorytm w powyższej postaci nosi nazwę **algorytmu Prima**.

Realizacja tego algorytmu w czasie  $O(n^3)$  nie powinna Ci przysporzyć kłopotu. W zadaniu 7.15 proponujemy rozwiązanie efektywniejsze.

### Zadania

- 7.1. Niech będą dane liczby naturalne  $n$  i  $m$  oraz ciąg krawędzi  $e_1, e_2, \dots, e_m$  pewnego  $n$ -wierzchołkowego grafu  $G$  (każda krawędź jest parą wierzchołków). Ułóż algorytmy tworzenia
- list sąsiedztwa w czasie  $O(n + m)$ ;
  - macierzy sąsiedztwa w czasie  $O(n^2)$ .
- 7.2. Zaproponuj sposób reprezentacji  $n$ -wierzchołkowych drzew w pamięci rozmiaru  $O(n)$ , umożliwiający sprawdzanie w czasie  $O(1)$ , czy para wierzchołków jest połączona w drzewie krawędzią.
- 7.3. Udowodnij, że wywołanie procedury  $search(u)$  spowoduje „odwiedzenie” wszystkich wierzchołków w grafie, do których prowadzą ścieżki o początku w  $u$ .
- 7.4. Zaproponuj liniowy algorytm sprawdzania, czy graf jest 2-kolorowalny.
- 7.5. Zaproponuj liniowy algorytm kolorowania krawędzi drzewa minimalną liczbą kolorów w taki sposób, żeby krawędzie o wspólnych końcach nie były pokolorowane tym samym kolorem.
- 7.6. Ułóż liniowy algorytm numerowania liczbami  $1, 2, \dots, n$  wierzchołków grafu zorientowanego bez cykli  $G = (V, E)$  ( $|V| = n$ ) w taki sposób, żeby dla każdej krawędzi  $u \rightarrow v \in E$  numer wierzchołka  $u$  był mniejszy niż numer wierzchołka  $v$ .
- 7.7. Niech  $G = (V, E)$  będzie  $n$ -wierzchołkowym grafem dwuspojnym. Funkcję różnicową  $f: V \rightarrow \{1, \dots, n\}$  nazywamy **s-t numeracją grafu**  $G$ , jeżeli wierzchołek  $f^{-1}(1)$  jest połączony w  $G$  krawędzią z wierzchołkiem  $f^{-1}(n)$  oraz dla każdego  $1 < i < n$  wierzchołek  $f^{-1}(i)$  jest połączony krawędziami z wierzchołkami  $f^{-1}(j)$  i  $f^{-1}(k)$  dla pewnych  $j$  i  $k$ , takich że  $j < i < k$ .  
Zaproponuj liniowy algorytm obliczania s-t numeracji danego grafu dwuspojnego  $G$ .
- 7.8. Niech  $R$  będzie relacją na krawędziach grafu  $G$ , zdefiniowaną następująco:  $e_1 R e_2$  wtedy i tylko wtedy, gdy  $e_1 = e_2$  lub istnieje cykl prosty zawierający jednocześnie  $e_1$  i  $e_2$ . Udowodnij, że  $R$  jest relacją równoważności.
- 7.9. Uzasadnij poprawność algorytmu znajdowania dwuspojnych składowych, działającego na podstawie metody przechodzenia w głąb. Wskazówka: dowiedź się o słuszności obserwacji 1-6.

### Zadania

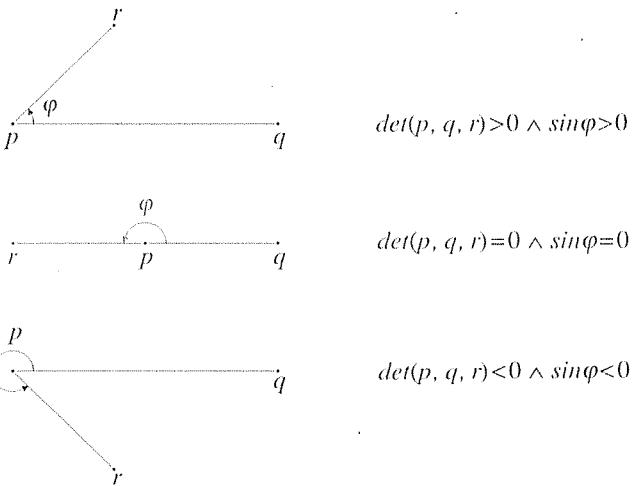
- 7.10. Zaproponuj liniowy algorytm obliczania mostów w grafach.
- 7.11. Udowodnij poprawność algorytmu rozwiązywania problemu 7.3.
- 7.12. Dany jest spójny graf  $G$  i jego drzewo rozpinające  $T$  z wyróżnionym korzeniem  $r$ . Zaproponuj liniowy algorytm sprawdzania, czy można tak uporządkować listy sąsiedztwa  $G$ , żeby  $T$  było dla grafu  $G$  drzewem przechodzenia w głąb.
- 7.13. Zaprojektuj algorytm liniowy, w którym drzewo przechodzenia wszerz jest wykorzystywane do znajdowania mostów w grafie.
- 7.14. Ułóż efektywny algorytm znajdowania spójnych składowych w grafie, ale w którym nie wykorzystuje się ani przechodzenia grafu w głąb, ani przechodzenia grafu wszerz.
- 7.15. Niech  $d \geq 2$  będzie liczbą całkowitą. Przez  **$d$ -kopiec zupełny** rozumiemy kopiec będący zupełnym drzewem  $d$ -arnym, w którym wszystkie poziomy są wypełnione całkowicie, z wyjątkiem co najwyżej ostatniego poziomu, który jest spójnie wypełniony od strony lewej. W rozdziale 2 rozważaliśmy 2-kopce. Zaprojektuj implementację z użyciem  $d$ -kopców algorytmu rozwiązywania problemu najkrótszych ścieżek (minimalnego drzewa rozpinającego) w czasie  $O(d \log_d n + m \log_d n)$ . Zanalizuj koszt algorytmu dla  $d = \lceil 2 + m/n \rceil$ .
- 7.16. Udowodnij, że krawędzie każdego grafu planarnego można zorientować w ten sposób, żeby stopień wyjściowy każdego wierzchołka był  $\leq 5$ . Wykorzystując ten fakt, zaproponuj sposób reprezentacji grafów planarnych w pamięci liniowej, umożliwiający sprawdzanie w czasie stałym, czy para wierzchołków jest połączona krawędzią.
- 7.17. Zaproponuj liniowy algorytm sprawdzania, czy dany graf planarny ma cykl prosty długości 3.
- 7.18. Zaproponuj liniowy algorytm znajdowania minimalnego drzewa rozpinającego w grafach planarnych.
- 7.19. Grafem **zewnętrznie planarnym** nazywamy graf planarny, który można przedstawić na płaszczyźnie w taki sposób, że wierzchołki są wierzchołkami pewnego wielokąta wypukłego, a krawędzie jego bokami (niekoniecznie wszystkimi) lub nie przecinającymi się (parą) przekątnymi. Zaproponuj liniowy algorytm sprawdzania, czy dany graf jest zewnętrznie planarny.
- 7.20. Zaproponuj liniowy algorytm kolorowania wierzchołków grafu zewnętrznie planarnego minimalną liczbą kolorów.

# 8 Algorytmy geometryczne

Jeżeli  $x$  jest liczbą rzeczywistą, to znak liczby  $x$ , oznaczany przez  $sgn(x)$ , definiuje się w następujący sposób:

$$sgn(x) = \begin{cases} +1 & \text{dla } x > 0 \\ 0 & \text{dla } x = 0 \\ -1 & \text{dla } x < 0 \end{cases}$$

Niech  $p, q$  i  $r$  będą różnymi punktami:  $p = (x, y)$ ,  $q = (x', y')$ ,  $r = (x'', y'')$ , a  $det(p, q, r)$  wyznacznikiem macierzy  $\begin{bmatrix} x & y & 1 \\ x' & y' & 1 \\ x'' & y'' & 1 \end{bmatrix}$ . Znak  $det(p, q, r)$  jest równy znakowi sinusa kąta nachylenia wektora  $p \rightarrow r$  do wektora  $p \rightarrow q$ . Powiemy, że punkt  $r$  leży po lewej (prawej) stronie wektora  $p \rightarrow q$ , jeżeli  $det(p, q, r) > 0$  ( $det(p, q, r) < 0$ ). Jeśli  $det(p, q, r) = 0$ , to punkty  $p, q$  i  $r$  są współliniowe (rys. 8.1).



Rys. 8.1. Możliwe położenia punktu  $r$  względem wektora  $p \rightarrow q$

Rozdział ten poświęcimy algorytmom geometrycznym. Dział informatyki zajmujący się tą problematyką jest nazywany geometrią obliczeniową. Przedstawimy najprostsze metody używane w konstruowaniu algorytmów geometrycznych. Elementarnymi operacjami arytmetycznymi, jakie dopuścimy w naszych algorytmach, będą dodawanie, odejmowanie, mnożenie i dzielenie. W szczególności nie będziemy używali funkcji trygonometrycznych. Więcej, założymy, że wszelkie obliczenia są wykonywane w arytmetyce nieskończonej precyzji, a zatem są zawsze dokładne. Nasze rozważania ograniczymy do problemów na płaszczyźnie. Rozpoczniemy od zdefiniowania elementarnych operacji, które potem wykorzystamy w bardziej skomplikowanych algorytmach dla następujących problemów:

- przynależności punktu do wielokąta;
- wypukłej otoczkii;
- najmniej odległej pary punktów;
- przecinających się par odcinków.

Na przykładzie powyższych zagadnień przedstawimy kilka ważnych metod używanych w projektowaniu efektywnych algorytmów geometrycznych. Będą to metody: „dziel i zwyciężaj”, przyrostowa i zamiatania.

Podstawowymi obiektami geometrycznymi, o których będziemy tu mówić, są **punkt**, **odecinek**, **wektor** oraz **prosta**. Każdy punkt  $p$  będziemy reprezentować przez parę jego współrzędnych  $(x(p), y(p))$  w ustalonym wcześniej układzie współrzędnych kartezjańskich. Odcinek o końcach w punktach  $p$  i  $q$  będziemy oznaczać przez  $p \dashv q$ , a wektor o początku w  $p$  i końcu w  $q$  przez  $p \rightarrow q$ . Każdą prostą będziemy reprezentować przez dowolny zawarty w niej odcinek (wektor) o różnych końcach. Mówiąc prostą  $p \dashv q$  ( $p \rightarrow q$ ), będziemy mieć na myśli prostą zawierającą odcinek (wektor)  $p \dashv q$  ( $p \rightarrow q$ ). Zanim przystąpimy do omawiania algorytmów geometrycznych, przypomnijmy kilka prostych faktów z geometrii analitycznej na płaszczyźnie.

## 8.1. Elementarne algorytmy geometryczne

Przedstawianie algorytmów geometrycznych rozpoczniemy od prezentacji algorytmów dla trzech bardzo prostych problemów, które są podstawowe dla naszych dalszych rozważań. Algorytmy te będą dalej wykorzystywane jako podprogramy w algorytmach bardziej skomplikowanych.

**PROBLEM 8.1.**

Naszym zadaniem jest stwierdzić, czy punkty  $p_1$  i  $p_2$  leżą po tej samej stronie prostej  $p - q$ .

**PROBLEM 8.2.**

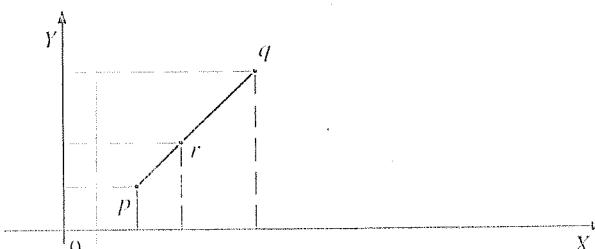
Mamy zbadanie, czy punkt  $r$  należy do odcinka  $p - q$ .

**PROBLEM 8.3.**

Mamy zbadanie, czy dwa odcinki  $p - q$  i  $r - s$  się przecinają.

Żeby rozwiązać problem 8.1, wystarczy sprawdzić, czy  $\text{sgn}(\det(p, q, p_1)) = \text{sgn}(\det(p, q, p_2))$ .

W celu rozwiązywania problemu 8.2 zauważmy, że jeśli punkt  $r$  należy do odcinka  $p - q$ , to rzuty prostokątne  $r$  na osie ( $OX$  i  $OY$ ) układu współrzędnych wpadają do rzutów prostokątnych odcinka  $p - q$  (rys. 8.2). Wynika stąd, że  $r$  należy do odcinka  $p - q$  wtedy i tylko wtedy, gdy  $\min(x(p), x(q)) \leq x(r) \leq \max(x(p), x(q))$ ,  $\min(y(p), y(q)) \leq y(r) \leq \max(y(p), y(q))$  oraz  $p, q$  i  $r$  są współliniowe ( $\text{sgn}(\det(p, q, r)) = 0$ ).



Rys. 8.2. Rozwiązywanie problemu 8.2

Rozwiązywanie problemu 8.3 opiera się na spostrzeżeniu, że dwa odcinki przecinają się wtedy i tylko wtedy, gdy punkty  $p$  i  $q$  leżą po przeciwnych stronach prostej  $r - s$  (wektora  $r \rightarrow s$ ), a punkty  $r$  i  $s$  po przeciwnych stronach prostej  $p - q$  lub któryś z końców jednego z odcinków należy do drugiego odcinka.

Jak łatwo zauważać, każdy z problemów 8.1, 8.2 i 8.3 można rozwiązać w czasie stałym, tylko z użyciem operacji arytmetycznych.

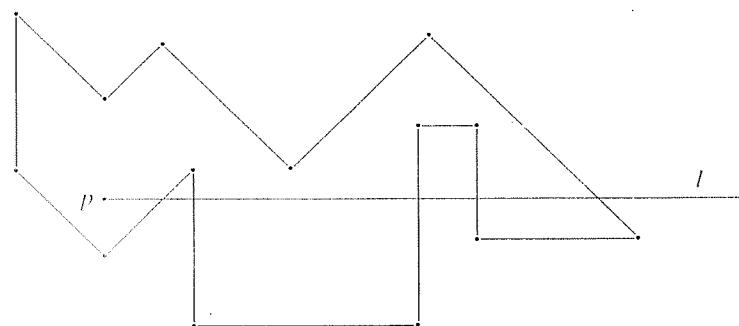
**8.2.****Problem przynależności**

Jednym z najczęściej spotykanych problemów w geometrii obliczeniowej jest problem przynależności.

**PROBLEM 8.4.**

Dany jest obiekt geometryczny  $W$  (zbiór punktów) oraz punkt  $p$ . Mamy zbadanie, czy  $p \in W$ .

Problem ten rozwiążemy dla przypadku, gdy  $W$  jest wielokątem. Odpowiedzmy sobie najpierw na pytanie, co to znaczy, że dany jest wielokąt  $W$ . Każdy  $n$ -wierzchołkowy wielokąt  $W$  będzie reprezentowany przez ciąg punktów  $w_0, \dots, w_{n-1}$  będących jego wierzchołkami i taki, że dla każdego  $i = 0, \dots, n-1$ ,  $w_i - w_{i+1}$  jest bokiem w  $W$  ( $i+1$  jest wyliczane modulo  $n$ ). Jest to przykład reprezentacji nieskończonego obiektu geometrycznego w sposób skończony (podobnie reprezentujemy odcinek i prostą). Przystąpmy teraz do rozwiązywania problemu 8.4. Wiemy, że w czasie liniowym ( $O(n)$ ) można sprawdzić, czy  $p$  należy do któregoś z boków wielokąta. Przyjmijmy zatem, że to nie zachodzi. Idea rozwiązania opiera się na następującym spostrzeżeniu. Niech  $l$  będzie półprostą o początku w  $p$ , taką że żaden z wierzchołków wielokąta  $W$  nie leży na tej półprostej. Punkt  $p$  leży wewnętrznie wielokąta  $W$  wtedy i tylko wtedy, gdy  $l$  przecina brzeg  $W$  nieparzystą liczbę razy (rys. 8.3).



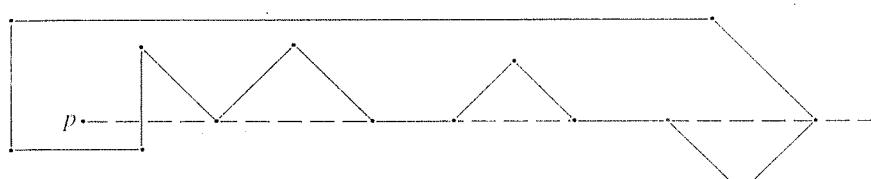
Rys. 8.3. Punkt  $p$  leży wewnętrznie wielokąta. Półprosta  $l$  o początku w  $p$  przecina brzeg wielokąta nieparzystą liczbę razy

Wynika stąd, że w celu rozwiązywania problemu przynależności punktu  $p$  do wielokąta wystarczy wybrać dowolną półprostą  $l$  o początku w  $p$ , na przykład równoległą do osi  $OX$ , i stwierdzić, ile razy przecina ona boki wielokąta. Niestety, może się zdarzyć, że  $l$  przecina brzeg wielokąta w wierzchołkach. Rozważmy dwa przypadki.

- (1) Prosta  $l$  zawiera bok wielokąta  $b$ . Niech  $c$  i  $d$  będą bokami sąsiadującymi z  $b$ , a  $x$  i  $y$  ich końcami nie należącymi do  $b$ . Jeśli punkty  $x$  i  $y$  leżą po przeciwnych stronach półprostej  $l$  (dokładniej prostej zawierającej  $l$ ), to przyjmujemy, że liczba punktów przecięcia  $l$  z bokami  $b$ ,  $c$  i  $d$  wynosi 1. W przeciwnym razie przyjmujemy, że liczba ta wynosi 0.
- (2) Prosta  $l$  przecina brzeg wielokąta w wierzchołku  $v$  i nie zawiera żadnego z boków z nim sąsiadujących. Niech  $b$  i  $c$  będą tymi bokami, a  $x$  i  $y$  ich końcami różnymi od  $v$ .

Jeśli  $x$  i  $y$  leżą po przeciwnych stronach  $l$ , to przyjmujemy, że liczba punktów przecięcia z bokami  $b$  i  $c$  wynosi 1. W przeciwnym razie przyjmujemy, że liczba ta wynosi 0.

Powyższe zasady są oparte na obserwacji, że każdą półprostą o początku w  $p$  można tak obrócić dookoła  $p$  o niewielki kąt  $\epsilon$ , żeby otrzymać półprostą nie przecinającą  $W$  w wierzchołkach (rys. 8.4).



Rys. 8.4. Liczba punktów przecięć półprostej  $l$  z wielokątem, liczona zgodnie z regulami podanymi w przypadkach (1) i (2), wynosi 3. Punkt  $p$  leży wewnątrz wielokąta

Ponieważ koszt obliczeń związanych z każdym bokiem i wierzchołkiem  $n$ -kąta jest stały, złożoność sprawdzenia, czy punkt leży w jego wnętrzu, jest  $O(n)$ . Problem ten możemy rozwiązać dużo szybciej, jeżeli wiemy coś więcej o wielokącie  $W$ . Rozważmy przypadek, gdy  $W$  jest **wypukły**. Przypomnijmy, że wielokąt jest wypukły wtedy, kiedy każdy odcinek o końcach należących do wielokąta jest całkowicie w nim zawarty. W celu sprawdzenia, czy  $p \in W$ , posłużymy się metodą wyszukiwania binarnego. Bez straty ogólności założymy, że wierzchołki  $W$  są dane w kolejności ich występowania na obwodzie, zgodnej z ruchem wskazówek zegara. Jest oczywiste, że w czasie stałym można stwierdzić, czy  $p$  leży wewnątrz trójkąta. Założymy, że  $W$  ma więcej niż trzy wierzchołki. Poprowadźmy przekątną łączącą  $w_0$  z  $w_{\lceil(n-1)/2\rceil}$ . Są trzy możliwości.

- (1) Punkt  $p$  leży na prostej  $w_0 - w_{\lceil(n-1)/2\rceil}$ . W tym wypadku łatwo stwierdzić, czy  $p$  należy do odcinka  $w_0 - w_{\lceil(n-1)/2\rceil}$ , czy leży poza nim.
- (2) Punkt  $p$  leży po lewej stronie wektora  $w_0 \rightarrow w_{\lceil(n-1)/2\rceil}$ . Należy sprawdzić rekurencyjnie, czy  $p$  leży wewnątrz wielokąta o wierzchołkach  $w_0, w_1, \dots, w_{\lceil(n-1)/2\rceil}$ .
- (3) Punkt  $p$  leży po prawej stronie wektora  $w_0 \rightarrow w_{\lceil(n-1)/2\rceil}$ . Należy sprawdzić rekurencyjnie, czy  $p$  leży wewnątrz wielokąta o wierzchołkach  $w_0, w_{\lceil(n-1)/2\rceil}, w_{\lceil(n-1)/2\rceil+1}, \dots, w_{n-1}$ .

Ponieważ na każdym poziomie rekursji wykonujemy stałą liczbę operacji o stałym koszcie, a rozmiar (liczba wierzchołków) badanego wielokąta zmniejsza się blisko o połowę (wynosi co najwyżej  $\lceil(n-1)/2\rceil + 1$ ), koszt sprawdzenia, czy dany punkt leży wewnątrz wielokąta wypukłego, jest  $O(\log n)$ . Podkreślimy na koniec, że chociaż nasz algorytm został opisany z użyciem rekursji, to łatwo go zaprogramować iteracyjnie.

Wielokąty wypukłe odgrywają ważną rolę w geometrii obliczeniowej. Jak mogliśmy zauważyć, obliczenia dla nich są zdecydowanie efektywniejsze. Dlatego kolejnym prob-

lemem którym się zajmiemy, jest problem dotyczący znajdowania wypukłej otoczki skończonego zbioru punktów.

### 8.3. Wypukła otoczka

Wypukłą otoczka dowolnego niepustego zbioru punktów  $S$  nazywamy najmniejszy zbiór wypukły zawierający  $S$ . Można udowodnić, że jeśli  $S$  jest zbiorem skończonym, to jego wypukła otoczka jest wielokątem wypukłym o wierzchołkach ze zbioru  $S$  (czasami zredukowanym do odcinka lub punktu).

#### PROBLEM 8.5.

Dany jest zbiór skończony  $n$  punktów  $S = \{p_1, \dots, p_n\}$ . Mamy znaleźć najmniejszy wielokąt wypukły zawierający  $S$ , a dokładniej – wyznaczyć wierzchołki tego wielokąta w kolejności ich występowania na obwodzie. Dla każdego wierzchołka  $p$  wypukłej otoczki kolejność ta będzie zadana przez dwa wskaźniki  $next(p)$  i  $pred(p)$ , wyznaczające – odpowiednio – wierzchołki następny i poprzedni względem  $p$  przy poruszaniu się po obwodzie otoczki w kierunku przeciwnym do ruchu wskazówek zegara.

Zanim przystąpimy do układania efektywnych algorytmów obliczania wypukłej otoczki, zastanówmy się, jak szybkich algorytmów możemy się spodziewać. Okazuje się, że problemu wypukłej otoczki nie można rozwiązać szybciej niż problemu sortowania. Aby to uzasadnić, pokażemy, że problem sortowania liczb rzeczywistych jest liniowo sprowadzalny do problemu wypukłej otoczki. Niech  $x_1, \dots, x_n$  będą liczbami rzeczywistymi, które chcemy uporządkować. Bez straty ogólności możemy założyć, że liczby te są parami różne i większe od zera. Rozważmy  $n$  punktów na płaszczyźnie  $(x_1, x_1^2), \dots, (x_n, x_n^2)$ . Punkty te leżą na prawym ramieniu paraboli  $y = x^2$  w kolejności wzrostu pierwszej współrzędnej. Kolejność ta wyznacza porządek między elementami ciągu  $x_1, \dots, x_n$ . Wierzchołkami wypukłej otoczki utworzonego zbioru punktów są właśnie te punkty: Ich kolejność (cyklicznie) na obwodzie wielokąta jest taka sama jak na ramieniu paraboli. Znając kolejność występowania punktów  $(x_1, x_1^2), \dots, (x_n, x_n^2)$  na obwodzie wypukłej otoczki, można już w czasie liniowym odtworzyć porządek między liczbami  $x_1, \dots, x_n$ . Wystarczy przejrzeć te punkty kolejno w kierunku przeciwnym do ruchu wskazówek zegara, poczynając od punktu z najmniejszą współrzędną  $x$ . Wynika stąd, że obliczenie wypukłej otoczki dla  $n$  punktów nie może kosztować mniej niż sortowanie  $n$  liczb, czyli  $\Omega(n \log n)$ .

Najprościej wypukłą otoczkę można obliczyć w podany tu sposób:

#### KROK 1:

Znaleźć wszystkie wierzchołki wypukłej otoczki zbioru punktów  $S$ .

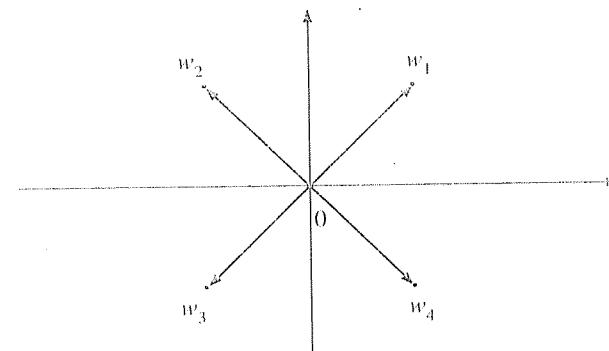
**KROK 2:**

Uporządkować znalezione punkty w kolejności ich występowania na obwodzie wypukłej otoczki.

W celu znalezienia wierzchołków wypukłej otoczki możemy oprzeć się na takim oto spostrzeżeniu: punkt  $p$  nie jest wierzchołkiem wypukłej otoczki wtedy i tylko wtedy, gdy leży wewnątrz pewnego trójkąta o wierzchołkach z  $S$ , różnych od  $p$ , lub należy do odcinka łączącego dwa punkty z  $S$ , różne od  $p$ .

Trójkąt o wierzchołkach  $q_1, q_2$  i  $q_3$  będziemy oznaczać przez  $\Delta(q_1 q_2 q_3)$ . To, czy dany punkt leży wewnątrz  $\Delta(q_1 q_2 q_3)$ , można sprawdzić w czasie stałym. Tak samo w czasie stałym można stwierdzić, czy  $p$  leży na odcinku  $q-r$ . Do powyższych obliczeń stosujemy algorytmy omówione na początku rozdziału. Ponieważ  $n$  punktów daje co najwyżej  $\binom{n}{3}$  różnych trójkątów, to bezpośrednia realizacja kroku 1 zabiera czas  $O(n^3)$ . W kroku 2 musimy rozwiązać następujący problem: danych jest  $n$  wierzchołków  $p_1, \dots, p_n$  pewnego wielokąta wypukłego  $P$ ; trzeba wyznaczyć kolejność ich występowania na obwodzie tego wielokąta.

W celu rozwiązania tego problemu wybierzmy dowolny punkt  $O$  wewnątrz wielokąta  $P$ . Tym punktem może być na przykład centroid, tzn. punkt  $((x(p_1) + \dots + x(p_n))/n, (y(p_1) + \dots + y(p_n))/n)$ .



Rys. 8.5. Wierzchołki wielokąta wypukłego uporządkowane rosnąco względem kątów nachylenia ich wektorów wodzących do osi  $OX$

Bez straty ogólności możemy przyjąć, że środek układu współrzędnych jest w  $O$ . Uporządkujmy wierzchołki rosnąco względem kątów nachylenia ich wektorów wodzących do osi  $X$  (rys. 8.5).

Sortowanie punktów możemy wykonać w czasie  $O(n \log n)$ , ale tylko pod warunkiem, że umiemy porównywać kąty nachylenia wektorów wodzących. Pokażemy, w jaki sposób

tego dokonać bez obliczania tych kątów. Niech  $\alpha(p)$  będzie funkcją określona dla punktów płaszczyzny różnych od  $O$ , zdefiniowaną następująco:

$$\alpha(p) = \begin{cases} y(p)/d(p), & \text{gdy } x(p) \geq 0 \wedge y(p) \geq 0 \\ 2 - y(p)/d(p), & \text{gdy } x(p) \leq 0 \wedge y(p) \geq 0 \\ 2 + |y(p)|/d(p), & \text{gdy } x(p) \leq 0 \wedge y(p) \leq 0 \\ 4 - |y(p)|/d(p), & \text{gdy } x(p) \geq 0 \wedge y(p) < 0 \end{cases}$$

gdzie  $d(p) = |x(p)| + |y(p)|$ . Jako zadanie pozostawiamy Ci udowodnienie, że kąt nachylenia wektora wodzącego punktu  $p_1$  jest mniejszy (równy) niż kąt nachylenia wektora wodzącego punktu  $p_2$  wtedy i tylko wtedy, gdy  $\alpha(p_1) \leq \alpha(p_2)$ . Funkcja  $\alpha(p)$  umożliwia wyznaczenie kolejności wierzchołków na obwodzie wielokąta wypukłego w czasie  $O(n \log n)$ .

Powstaje pytanie, czy żeby wyznaczyć wierzchołki wypukłej otoczki, musimy sprawdzać wszystkie  $\binom{n}{3}$  trójkąty. Bez straty ogólności możemy przyjąć, że znamy pewien punkt  $O$  leżący wewnątrz wypukłej otoczki, na przykład centroid. Latwo zauważycie, że każdy punkt nie będący wierzchołkiem wypukłej otoczki musi wpadać do wnętrza trójkąta wyznaczonego przez punkt  $O$  i pewne dwa kolejne wierzchołki otoczki (lub musi leżeć na jednym z boków takiego trójkąta). Spostrzeżenie to jest kluczowe w algorytmie Grahama służącym do obliczania wypukłej otoczki  $n$  punktów w czasie  $O(n \log n)$ .

## {Algorytm Grahama obliczania wypukłej otoczki}

**KROK 1:**

Wybierz dowolny punkt  $O$  wewnątrz wypukłej otoczki, na przykład centroid. Umieścę w nim środek układu współrzędnych i oblicz współrzędne punktów wejściowych w nowym układzie współrzędnych.

**KROK 2:**

Uporządkuj punkty  $p_1, \dots, p_n$  leksykograficznie względem współrzędnych biegunowych  $(\alpha_i, r_i)$ , gdzie  $\alpha_i$  jest kątem nachylenia wektora wodzącego  $O \rightarrow p_i$  do osi  $OX$ , a  $r_i$  jego długością. (Uwaga: Żeby nie liczyć pierwiastków, w sortowaniu porównujemy  $\alpha(p_i)$  zamiast  $\alpha_i$  oraz  $r_i^2$  zamiast  $r_i$ ).

Z uporządkowanych punktów utwórz dwukierunkową listę cykliczną, w której dla każdego punktu  $p$ ,  $next(p)$  jest następnym (cyklicznie) punktem w wyżej zdefiniowanym porządku, a  $pred(p)$  poprzednim.

Spośród punktów o najmniejszej współrzędnej  $y$  znajdź punkt  $s$  z najmniejszą współrzędną  $x$ .

**KROK 3:**

W kroku 3 przejrzij punkty na liście, usuwając te, które nie są wierzchołkami wypukłej otoczki. Po zakończeniu działania algorytmu lista będzie zawierała tylko wierzchołki wypukłej otoczki w kolejności ich występowania na obwodzie. Listę przeglądaj, zaczynając od punktu  $s$  i kierując się w stronę przeciwną do ruchu wskazówek zegara (zgodnie ze wskaźnikami  $next$ ). W celu wyeliminowania zbędnych punktów zawsze sprawdzaj trzy kolejne punkty  $q_1, q_2$  i  $q_3$  z bieżącej listy. Jeśli  $q_2$  leży wewnętrz  $\Delta(O, q_1, q_3)$ , to usuń z niej  $q_2$  i przejdź do sprawdzania punktów  $q_0, q_2, q_3$ . W przeciwnym razie kolejną badaną trójką punktów są  $q_2, q_3, q_4$ . Przeglądanie  $q_1, q_3$ . W chwilą osiągnięcia wierzchołka startowego  $s$ . Możemy to zapisać tak:

```

 $q := s;$ 
while  $next(q) \neq s$  do
  {*}
    if  $next(q)$  leży wewnętrz  $\Delta(O, q, next(next(q)))$  then
      begin
        {usunięcie punktu  $next(q)$  z listy}
         $next(q) := next(next(q));$ 
         $pred(next(q)) := q;$ 
      {*}
        if  $q \neq s$  then  $q := pred(q)$ 
      end
    {*}
  else  $q := next(q);$ 

```

Na złożoność powyższego algorytmu decydujący wpływ ma krok 2, który można wykonać w czasie  $O(n \log n)$ . Kroki 1 i 3 są wykonywane w czasie liniowym. Uzasadnienia wymaga tylko złożoność kroku 3. W analizie złożoności tego kroku stosujemy zasadę magazynu. W magazynie znajdują się wszystkie punkty na liście między punktem startowym  $s$  a testowanym właśnie punktem  $next(q)$ , włącznie z tym punktem. W każdym wykonaniu instrukcji **while** albo posuwamy się po liście do przodu zgodnie z  $next$  (krok oznaczony {\*}), albo do tyłu zgodnie z  $pred$  (krok oznaczony {\*}). Przejście zgodne z  $next$  odpowiada włożeniu do magazynu nowego punktu  $next(next(q))$ , a przejście zgodne z  $pred$  oznacza usunięcie z magazynu punktu  $next(q)$ . Zwróćmy także uwagę, że jeśli wracamy do punktu startowego  $s$ , to do magazynu jest wklejany bieżący punkt  $next(s)$ . Każdy wierzchołek jest wklejany do magazynu raz i co najwyżej raz może być z niego usunięty. Stąd wynika liniowość kroku 3. W ten sposób udowodniliśmy, że problem wypukłej otoczki można rozwiązać w czasie  $O(n \log n)$ . Na koniec zauważmy jeszcze, że zamiast sprawdzać, czy punkt  $next(q)$  leży wewnętrz  $\Delta(O, q, next(next(q)))$  (warunek oznaczony {\*}), prościej jest testować, czy  $next(q)$  leży po lewej stronie (lub należy do) wektora  $q \rightarrow next(next(q))$ .

Złożoność algorytmu Grahama nie zależy od liczby wierzchołków obliczanej wypukłej otoczki. Przedstawimy teraz algorytm, który umożliwia znalezienie wypukłej otoczki w czasie  $O(kn)$ , gdzie  $k$  jest liczbą jej wierzchołków. Algorytm ten jest szczególnie przydatny wtedy, kiedy wiemy z góry, że liczba wierzchołków wypukłej otoczki jest niewielka, na przykład ograniczona przez stałą niezależną od  $n$ . Pomyśl takiego rozwiązania pochodzącego od R. Jarvisa i jest oparty na podanych tu dwóch spostrzeżeniach.

**8.3. Wypukła otoczka**

- (1) Odcinek  $p \rightarrow q$  o końcach ze zbioru  $S$  jest bokiem wypukłej otoczki wtedy i tylko wtedy, gdy wszystkie punkty z  $S$  należą do tej samej domkniętej półpłaszczyzny wyznaczonej przez prostą  $p \rightarrow q$ , a każdy punkt z  $S$  leżący na tej prostej należy do odcinka  $p \rightarrow q$ .
- (2) Jeśli odcinek  $p \rightarrow q$  jest bokiem wypukłej otoczki, która nie jest zdegenerowana do odcinka lub punktu, to musi istnieć bok różny od  $p \rightarrow q$ , zaczynający się w  $q$  ( $p$ ).

{Algorytm Jarvisa obliczania wypukłej otoczki}

**KROK 1:**

Spośród punktów z najmniejszą współrzędną y znajdź punkt  $d$  położony najbardziej na lewo (czyli z najmniejszą współrzędną x), a spośród punktów z największą współrzędną y punkt  $g$  położony najbardziej na prawo (z największą współrzędną x). Łatwo zauważyc, że oba punkty  $d$  i  $g$  są wierzchołkami wypukłej otoczki.

**KROK 2:**

```

 $p := d;$ 
while  $p \neq g$  do
  begin
    { $p$  jest kolejnym wierzchołkiem wypukłej otoczki}
    umieść środek układu współrzędnych w punkcie  $p$ , a następnie
    spośród punktów o najmniejszym kącie nachylenia wektora
    wodzącego do osi  $p \rightarrow g$  wybierz punkt  $r$  o największej odległości od  $p$ ;
    {wszystkie punkty z  $S$  leżą w jednej półpłaszczyźnie wyznaczonej
    przez odcinek  $p \rightarrow r$ ; odcinek ten jest kolejnym bokiem wypukłej
    otoczki}
     $next(p) := r; pred(r) := p; p := r$ 
  end;

```

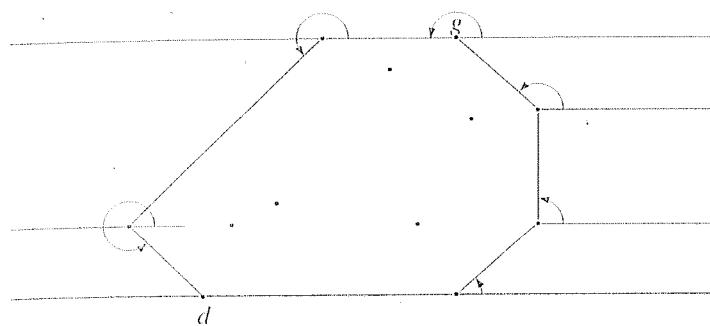
**KROK 3:**

Powtórz krok 2, przyjmując za punkt startowy  $g$ , a za punkt końcowy  $d$ . Rozważaj tylko punkty o kątach nachylenia promieni wodzących  $\geq 180^\circ$ .

Działanie algorytmu Jarvisa przedstawiamy na rysunku 8.6.

Każda iteracja w krokach 2 i 3 jest wykonywana w czasie  $O(n)$ . Ponieważ liczba iteracji jest równa liczbie wierzchołków wypukłej otoczki, złożoność algorytmu Jarvisa wynosi  $O(kn)$ .

Przedstawiliśmy dwa najbardziej znane algorytmy obliczania wypukłej otoczki. Problem wypukłej otoczki jest tak podstawowym problemem w geometrii obliczeniowej jak problem sortowania w całej informatyce. Na jego przykładzie można zademonstrować wiele różnych metod stosowanych w projektowaniu efektywnych algorytmów geometrycznych.



Rys. 8.6. Zasada działania algorytmu Jarvisa

Przedstawimy teraz jeszcze dwa inne algorytmy obliczania wypukłej otoczki. Jeden z nich został zaprojektowany na podstawie metody **przyrostowej**, a drugi na podstawie metody „**dziel i zwyciężaj**” (o której była mowa już wcześniej). Dla prostoty opisu algorytmów zakładamy, że żadne trzy różne punkty zbioru wejściowego  $S$  nie są współliniowe, a żadne dwa nie mają tej samej współrzędnej  $x$ . Jako zadanie pozostawimy Ci uwzględnienie braku powyższego założenia w algorytmach opisanych poniżej. Przyjmijmy też, że liczba punktów w zbiorze  $S$  jest większa niż 3.

W obu algorytmach punkty zbioru wejściowego  $S$  są najpierw porządkowane rosnąco względem pierwszych współrzędnych. Ponieważ sortowanie można wykonać w czasie  $O(n \log n)$ , założymy, że punkty  $p_1, p_2, \dots, p_n$  są dane już w takim porządku.

W algorytmie przyrostowym budujemy ciąg wypukłych otoczek  $WP_i$  dla  $i = 3, 4, \dots, n$ , w którym  $WP_i$  jest wypukłą otoczką punktów  $p_1, p_2, \dots, p_i$ . Otoczka  $WP_n$  jest żądanym wynikiem algorytmu.

{Algorytm przyrostowy obliczania wypukłej otoczki}

**Krok 1:**

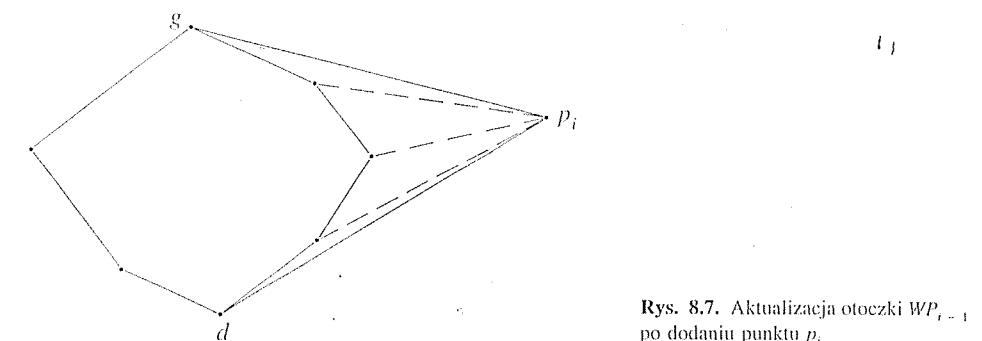
Zbuduj wypukłą otoczkę  $WP_3$ , tzn. dla każdego punktu  $p_i$ , gdzie  $i = 1, 2, 3$ , wyznacz  $next(p_i)$  oraz  $pred(p_i)$ , czyli – odpowiednio – następny i poprzedni punkt dla  $p_i$  w wypukłej otoczce  $WP_3$  w kierunku przeciwnym do ruchu wskazówek zegara.

**Krok 2:**

**for**  $i := 4$  **do**  
**dodaj** punkt  $p_i$  **do** bieżącej wypukłej otoczki  $WP_{i-1}$  **i aktualizuj**  
**dwukierunkową listę cykliczną**  $next$ - $pred$  **tak, żeby reprezentowała**  
**otoczkę**  $WP_i$ ;

Pozostaje jeszcze tylko pokazać, jak należy aktualizować listę  $next$ - $pred$  wierzchołków otoczki  $WP_{i-1}$  po dodaniu punktu  $p_i$ . Niech  $P_j$  będzie zbiorem wierzchołków  $WP_j$  dla

$j = 3, 4, \dots, n$ . W celu aktualizacji listy dla  $WP_{i-1}$  znajdujemy wśród wierzchołków z  $P_{i-1}$  takie punkty  $d$  i  $g$ , że wszystkie punkty z  $P_{i-1} \setminus \{d\}$  leżą na prawo od wektora  $p_i \rightarrow d$ , a wszystkie punkty z  $P_{i-1} \setminus \{g\}$  – na lewo od wektora  $p_i \rightarrow g$ . Następnie w miejscu punktów  $next(d)$ ,  $next(next(d))$ , ...,  $pred(g)$  wstawiamy punkt  $p_i$ . Na rysunku 8.7 jest przedstawiony przebieg tej operacji.

Rys. 8.7. Aktualizacja otoczki  $WP_{i-1}$  po dodaniu punktu  $p_i$ 

Oto formalny opis aktualizacji otoczki  $WP_{i-1}$ .

```
{znajdź punkt d}
d := p_{i-1};
while punkt pred(d) leży na lewo od wektora p_i -> d do
  d := pred(d);
{znajdź punkt g}
g := p_{i-1};
while punkt next(g) leży na prawo od wektora p_i -> g do
  g := next(g);
{usuń punkty między d i g; wstaw w to miejsce p_i}
next(d) := p_i; pred(p_i) := d;
next(p_i) := g; pred(g) := p_i;
```

Stosując zasadę magazynu, łatwo jest pokazać, że jeśli ciąg wejściowy punktów jest uporządkowany rosnąco względem pierwszych współrzędnych, to algorytm przyrostowy działa w czasie liniowym.

Ostatni z prezentowanych algorytmów obliczania wypukłej otoczki działa na podstawie zasady „**dziel i zwyciężaj**”.

```
{Algorytm typu „dziel i zwyciężaj” obliczania wypukłej otoczki}
begin
  if n ≤ 3 then
    zbuduj bezpośrednio dwukierunkową listę cykliczną next-pred
    dla punktów ze zbioru S
  else
```

```

begin
  {dziel}
   $k := \lfloor n/2 \rfloor$ ;  $S_1 := \{p_1, \dots, p_k\}$ ;  $S_2 := \{p_{k+1}, \dots, p_n\}$ ;
  {rekurencja}
  oblicz rekurencyjnie wypukłe otoczki (listy next-pred)
  dla zbiorów  $S_1$  i  $S_2$ ;
  {łącz}
  połącz wypukłe otoczki dla zbiorów  $S_1$  i  $S_2$  w jedną wypukłą
  otoczkę dla zbioru  $S$ 
end
end;

```

W większości algorytmów typu „dziel i zwyciężaj” najtrudniejsze jest obliczenie rezultatu końcowego z rezultatów otrzymanych w wyniku wywołań rekurencyjnych. Podobnie jest w tym wypadku. Pokażemy, jak obliczyć wypukłą otoczkę dwóch wielokątów wypukłych  $W_1$  i  $W_2$  (odpowiednio, wypukłych otoczek dla zbiorów  $S_1$  i  $S_2$ ), takich że wielokąt  $W_1$  leży na lewo od  $W_2$ , tzn. pierwsza współrzędna każdego wierzchołka w  $W_1$  jest mniejsza niż pierwsza współrzędna każdego wierzchołka z  $W_2$ . Zauważmy, że najbardziej wysuniętym na prawo wierzchołkiem wielokąta  $W_1$  jest punkt  $p_k$ , a najbardziej wysuniętym na lewo wierzchołkiem w  $W_2$  jest punkt  $p_{k+1}$ . W celu policzenia wypukłej otoczki obu wielokątów wystarczy znaleźć dwie pary wierzchołków  $g_1, g_2$  oraz  $d_1, d_2$ , takich że  $g_1$  i  $d_1$  są wierzchołkami w  $W_1$ , a  $g_2$  i  $d_2$  wierzchołkami w  $W_2$  oraz wszystkie wierzchołki z obu wielokątów, poza  $g_1$  i  $g_2$ , leżą na prawo od wektora  $g_1 \rightarrow g_2$ , a wszystkie wierzchołki, poza  $d_1$  i  $d_2$ , leżą na lewo od wektora  $d_1 \rightarrow d_2$ . Jeśli z listy *next-pred* dla wielokąta  $W_1$  wytniemy wszystkie punkty leżące między  $d_1$  i  $g_1$  w kierunku przeciwnym do ruchu wskazówek zegara, a z listy *next-pred* dla  $W_2$  wytniemy wszystkie punkty między  $d_2$  i  $g_2$  w kierunku zgodnym z ruchem wskazówek zegara, a następnie połączymy obie listy, to otrzymamy żądaną reprezentację wypukłej otoczki dla zbioru wejściowego  $S$ . Proces ten przedstawiamy na rysunku 8.8.

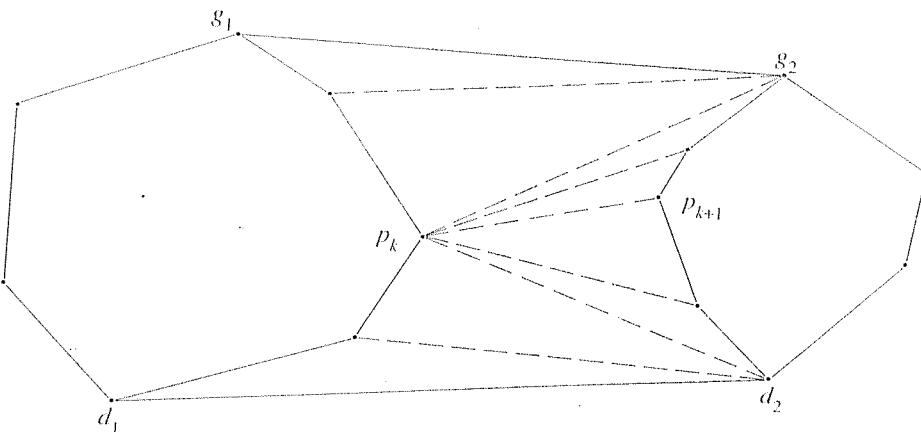
Oto algorytm znajdowania punktów  $g_1$  i  $g_2$ . Podobnie można znaleźć punkty  $d_1$  i  $d_2$ .

{Algorytm znajdowania punktów  $g_1$  i  $g_2$ }

```

begin
   $l := p_k$ ;  $r := p_{k+1}$ ;
  repeat
    while punkt pred( $r$ ) leży na lewo od wektora  $l \rightarrow r$  do
       $r := \text{pred}(r)$ ;
    while punkt next( $l$ ) leży na lewo od wektora  $l \rightarrow r$  do
       $l := \text{next}(l)$ 
    until (pred( $r$ ) leży na prawo od  $l \rightarrow r$ ) and
      (next( $l$ ) leży na prawo od  $l \rightarrow r$ );
     $g_1 := l$ ;  $g_2 := r$ 
  end;

```



Rys. 8.8. Obliczanie wypukłej otoczki dwóch wielokątów wypukłych

Jeśli punkty  $d_1, g_1, d_2$  i  $g_2$  są już obliczone, to wycinanie wierzchołków nie należących do wypukłej otoczki i obliczanie jej reprezentacji wykonuje się następująco:

```

pred( $d_2$ ) :=  $d_1$ ; next( $d_1$ ) :=  $d_2$ ;
pred( $g_1$ ) :=  $g_2$ ; next( $g_2$ ) :=  $g_1$ ;

```

Jaka jest złożoność przedstawionego algorytmu? Zauważmy, że jeśli wierzchołek jest usuwany z listy *next-pred*, to więcej na takiej liście już się nie znajdzie. Czas obliczania wierzchołków  $g_1, d_1, g_2$  i  $d_2$  (instrukcja *repeat*) liniowo zależy od liczby usuwanych wierzchołków. Wynika stąd, że jeśli ciąg wejściowy punktów jest uporządkowany rosnąco względem pierwszych współrzędnych, to powyższy algorytm działa w czasie liniowym.

## 8.4. Metoda zamiatania

Na zakończenie tego rozdziału przedstawimy pewną metodę systematycznego przeszukiwania płaszczyzny, a następnie zastosujemy ją do rozwiązywania dwóch podanych tu problemów.

### PROBLEM 8.6.

Dany jest zbiór  $S$  zawierający  $n$  punktów płaszczyzny. Mamy znaleźć najmniejszą odległą parę punktów w  $S$ , tzn. dwa różne punkty  $p$  i  $q$  w  $S$ , takie że

$$d(p, q) = \min\{d(r, s) : r, s \in S, r \neq s\}$$

gdzie  $d(r, s)$  jest odległością między punktami  $r$  i  $s$ .

**PROBLEM 8.7.**

Danych jest  $n$  odcinków  $l_1, \dots, l_n$ . Mamy wyznaczyć wszystkie pary przecinających się odcinków.

Zauważmy, że każdy z powyższych problemów łatwo rozwiązać w czasie  $O(n^2)$ . W tym celu wystarczy obliczyć odległości między każdą parą punktów z problemu 8.6 i zbadanie każdą parę odcinków z problemu 8.7. Za chwilę pokażemy, jak rozwiązać oba problemy stosując **metodę zamiatania**.

Metoda ta polega na przesuwaniu po płaszczyźnie („zamiataniu”) prostej pionowej („miotły”) od strony lewej do prawej i wykonywaniu obliczeń dla napotykanych obiektów (figur) geometrycznych. W każdym kroku obliczeń każdy obiekt jest albo **przetworzony**, albo aktywny, albo **oczekujący**. Obiekty przetworzone znajdują się zawsze z lewej strony miotły i wszystkie potrzebne obliczenia z nimi związane są już zakończone. Obiekty aktywne są aktualnie przetwarzane, natomiast obiekty oczekujące znajdują się z prawej strony miotły i obliczenia z nimi związane będą dopiero wykonywane. Miotła jest przesuwana po płaszczyźnie w sposób dyskretny. Punkty, między którymi się porusza, są elementami **x-struktury**. Struktura ta jest zazwyczaj kolejką priorytetową, w której punkty są uporządkowane niemalejąco względem ich pierwszych współrzędnych. Miotła zawsze wędruje do punktu z x-struktury z najmniejszą współrzędną  $x$ , który jest później z tej struktury usuwany. Obiekty aktywne natomiast są pamiętane w **y-strukturze**. W każdym położeniu miotły obiekty aktywne są uporządkowane niemalejąco względem drugich współrzędnych pewnych wyróżnionych punktów, należących do tych obiektów. Punkty te, reprezentancje obiektów, mogą być różne w różnych położeniach miotły. Do reprezentacji y-struktury używa się zazwyczaj zrównoważonych drzew poszukiwań binarnych, z pewnymi modyfikacjami zależnymi od rozwiązywanego problemu. Zamiatanie staramy się zawsze tak zorganizować, żeby miotła wędrowała przez możliwie mało punktów i żeby w każdym punkcie była wykonywana tylko stała liczba operacji na y-strukturze.

Przystąpimy teraz do opisu algorytmów dla problemów 8.6 i 8.7. Podobnie jak w wypadku wypukłej otoczki, złożymy, że punkty zbioru  $S$  w problemie 8.6 i końca odcinków wejściowych w problemie 8.7 spełniają następujące warunki: żadne trzy różne punkty zbioru wejściowego  $S$  nie są współliniowe i żadne dwa punkty nie mają tej samej współrzędnej  $x$ . Więcej, złożymy, że w jednym punkcie przecinają się co najwyżej dwa odcinki. Robimy to założenie tylko dla przejrzystości opisu prezentowanych algorytmów. Nie powinieneś mieć żadnych trudności z taką zmianą przedstawionych tu algorytmów, żeby działały one poprawnie dla każdych danych wejściowych.

**8.4.1.****Najmniej odległa para punktów**

Poprawność algorytmu, który za chwilę opiszymy, wynika z następującej obserwacji: jeśli  $S$  jest zbiorem  $n$  punktów, a  $\delta$  odlegością między najmniej odległą parą punktów z  $S$ , to każdy kwadrat o bokach o długości  $\delta$  zawiera co najwyżej 4 punkty z  $S$ .

Załóżmy przeciwnie. Niech  $K$  będzie kwadratem o bokach o długości  $\delta$ , zawierającym więcej niż 4 punkty z  $S$ . Jeśli podzielimy  $K$  na cztery mniejsze kwadraty o długości boków  $\delta/2$  każdy, to jeden z nich musi zawierać co najmniej 2 punkty. Największa odległość między punktami takiego kwadratu wynosi  $(\sqrt{2}\delta)/2$ . Jest to sprzeczne z założeniem, że odległość  $\delta$  jest najmniejsza.

Algorytm obliczania najmniej odległej pary punktów metodą zamiatania jest następujący. W x-strukturze są przechowywane punkty zbioru wejściowego  $S$ , uporządkowane niemalejąco względem pierwszych współrzędnych. W tym algorytmie x-struktura jest listą. Wskaźnik do punktu z listy, do którego ma być przesunięta miotła, znajduje się na zmiennej *current*. Do reprezentacji y-struktury używamy zrównoważonego drzewa poszukiwań (na przykład drzewa AVL), w liściach którego od strony lewej do prawej znajdują się punkty aktywne, uporządkowane niemalejąco względem współrzędnej  $y$ . Liście drzewa AVL są połączone w listę dwukierunkową. Jeśli  $q$  jest punktem w y-strukturze, to *pred*( $q$ ) jest w tej strukturze punktem bezpośrednio poprzedzającym  $q$ , a *succ*( $q$ ) jest punktem bezpośrednio następującym po  $q$ . Dla lepszej organizacji obliczeń zakładamy, że w liściach znajdują się dwa sztuczne punkty (strażnicy)  $(-\infty, -\infty)$ ,  $(-\infty, +\infty)$ . Jak już wielokrotnie przyjmowaliśmy w tej książce,  $-\infty$  i  $+\infty$  oznaczają liczby, odpowiednio, mniejszą i większą od każdej liczby pojawiającej się w trakcie obliczeń. Punkt  $(-\infty, -\infty)$  jest pierwszym punktem w y-strukturze, a punkt  $(-\infty, +\infty)$  ostatnim. Dodatkowo zakładamy, że każdy węzeł  $v$  w drzewie jest wyposażony w atrybut *max*( $v$ ), którego wartością jest największa współrzędna  $y$  spośród drugich współrzędnych punktów znajdujących się w liściach poddrzewa o korzeniu w  $v$ . Umożliwi to szybkie wyszukiwanie punktów w liściach drzewa. Na y-strukturze definiujemy operacje *find*( $p$ ), *insert*( $p$ ) i *delete*( $p$ ). W wyniku operacji *find*( $p$ ) zostaje zwrócony taki punkt  $q$  z y-struktury, że  $y(p) \leq y(q)$  i druga współrzędna każdego punktu w y-strukturze poprzedzającego  $q$  nie przekracza  $y(p)$ . Operacja *insert*( $p$ ) służy do wstawienia punktu  $p$  do y-struktury. W wyniku operacji *delete*( $p$ ) zostaje usunięty punkt  $p$  z y-struktury. Jako zadanie pozostawiamy Ci opracowanie algorytmu tych operacji w takim sposób, żeby przebiegały w czasie  $O(\log k)$ , gdzie  $k$  jest liczbą punktów w y-strukturze. W każdym położeniu miotły jest pamiętana najmniejsza odległość  $\delta$  między punktami na lewo od miotły. Punktami aktywnymi są wszystkie punkty na lewo od miotły, których odległość od niej wynosi co najwyżej  $\delta$ . Punkty aktywne znajdują się w x-strukturze (która jest listą uporządkowaną) między punktami wskazywanymi przez *first-active* i *current*, bez tego ostatniego. Zakładamy też, że każdy punkt aktywny w x-strukturze ma wskaźnik do swojego wystąpienia w y-strukturze. Rozważmy sytuację, kiedy miotła przesuwa się do punktu  $p = (x(p), y(p))$  (punkt wskazywany przez *current*). Wśród punktów aktywnych znajdujemy takie dwa kolejne punkty  $l$  i  $r$ , że  $y(l) \leq y(p) \leq y(r)$ . W tym celu wykorzystujemy operację *find* i listę, na której znajdują się liście. W wyniku dolożenia nowego punktu  $p$  dotycząca minimalna odległość  $\delta$  między punktami może zmaleć. Które punkty mogą leżeć w odległości od  $p$  mniejszej niż  $\delta$ ? Łatwo zauważyc, że tymi punktami mogą być tylko punkty aktywne o drugiej współrzędnej z przedziału  $[y(p) - \delta, y(p) + \delta]$ . Z obserwacji wynika, że takich punktów jest co najwyżej 8 – cztery poprzedzające

na liście  $l$  (włącznie z tym punktem) i cztery występujące po  $r$  (także włącznie z tym punktem). Oto formalny opis algorytmu obliczania najmniej odległej pary punktów.

```

begin
  {inicjowanie y-struktury i x-struktury}
  y-struktura := {(-∞, -∞); (-∞, +∞)};
  x-struktura := lista uporządkowana  $n$  punktów ze zbioru  $S$ ;
  {pierwszym punktem „postoju” miotły jest pierwszy punkt
  w x-strukturze}
  current := wskaźnik do pierwszego punktu w x-strukturze;
  first-active := current;
  δ := +∞;
  while current ≠ nil {nie wyczerpaliśmy punktów z x-struktury} do
  begin
    p := punkt na liście wskazywany przez current;
    {„usuń” p z x-struktury}
    current := wskaźnik do następnego elementu w x-strukturze;
    r := find(p); l := pred(r); i := 0;
    {obliczanie odległości między p i co najwyżej 4 punktami
    występującymi po r (włącznie z tym punktem) w y-strukturze;
    aktualizacja δ}
    while (r ≠ (-∞, +∞)) and (i < 4) do
    begin
      if  $d(p, r) < \delta$  then
        begin {aktualizacja δ}
          δ :=  $d(p, r)$ ;
          { $P, Q$  jest parą punktów w odległości  $\delta$ }
          P := p; Q := r
        end;
      r := succ(r); i := i + 1
    end;
    {obliczanie odległości między p i co najwyżej 4 punktami
    występującymi przed l (włącznie z tym punktem)
    w y-strukturze; aktualizacja δ}
    i := 0;
    while (l ≠ (-∞, -∞)) and (i < 4) do
    begin
      if  $d(p, l) < \delta$  then
        begin {aktualizacja δ}
          δ :=  $d(p, l)$ ;
          { $P, Q$  jest parą punktów o odległości  $\delta$ }
          P := p; Q := l
        end;
      l := pred(l); i := i + 1
    end;
  end;

```

```

  {punkty aktywne, które znalazły się w odległości większej
  niż  $\delta$ , są usuwane z y-struktury}
  q := punkt w x-strukturze wskazywany przez first-active;
  while ( $x(p) - x(q) > \delta$ ) do
  begin
    first-active := wskaźnik do następnego punktu
    w x-strukturze po q;
    delete(q); {usunięcie q z y-struktury}
    q := punkt wskazywany przez first-active
  end;
  {wstawienie punktu p do y-struktury}
  insert(p)
  end {while current ≠ ...}
end; { $P, Q$  jest najmniej odległą parą punktów}

```

Jaka jest złożoność przedstawionego algorytmu? Zainicjowanie x-struktury może być wykonane w czasie  $O(n \log n)$ . Każdy punkt trafia do y-struktury tylko raz (operacja  $insert$ ) i co najwyżej raz jest z niej usuwany (operacja  $delete$ ). Zarówno wstawienie punktu, jak i usunięcie zabierają czas  $O(\log n)$ . W każdym wykonaniu instrukcji `while current ≠ ...` jest sprawdzanych tylko co najwyżej 8 punktów z y-struktury. Wynika stąd, że znalezienie za pomocą tego algorytmu najmniej odległej pary punktów zajmuje  $O(n \log n)$  czasu. Można udowodnić, że rozwiązanie to jest optymalne. Dalej na przykładzie przedstawiamy zasady działania omówionego właśnie algorytmu (rys. 8.9). Rozwiązanie zostało po raz pierwszy podane w pracy [HNS].

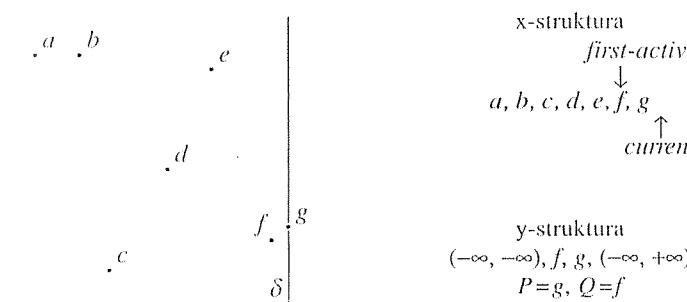
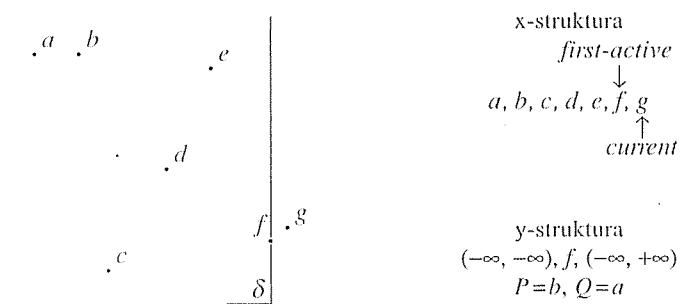
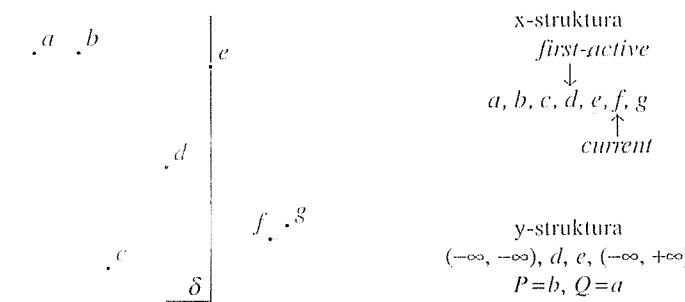
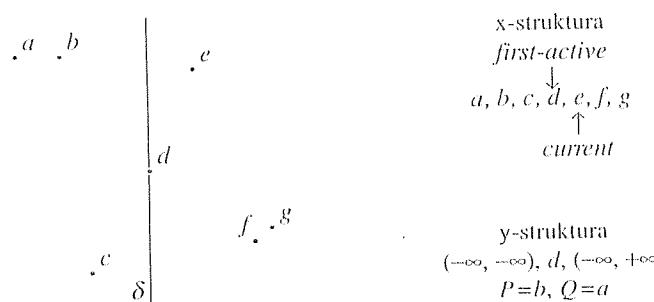
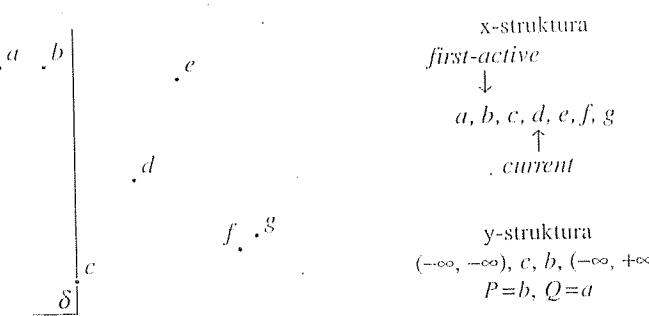
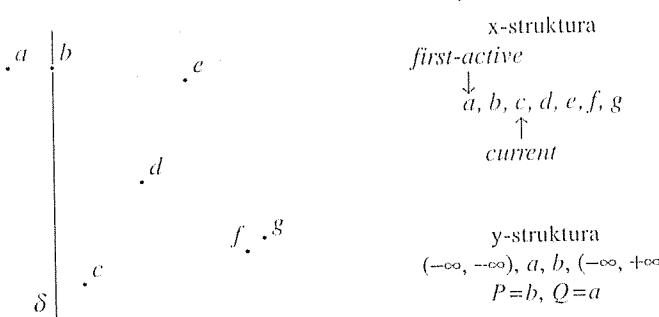
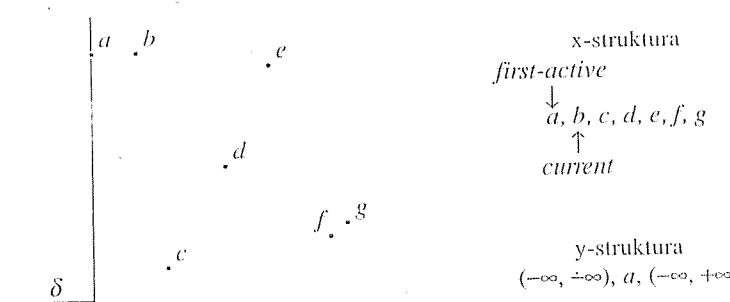
#### 8.4.2. Pary przecinających się odcinków

Jak już wspominaliśmy, problem przecinających się odcinków można łatwo rozwiązać w czasie  $O(n^2)$ . W tym celu wystarczy sprawdzić każdą parę odcinków. Rozwiązanie to jest optymalne, gdy liczba par przecinających się odcinków jest rzędu  $n^2$ . W algorytmie, który za chwilę przedstawimy, jest użyta metoda zamiatańia. Jego złożoność zależy od liczby przecinających się par odcinków. Jeśli  $s$  jest liczbą takich par, to za pomocą naszego algorytmu można je znaleźć w czasie  $O((n + s) \log n)$ . W każdym położeniu miotły odcinkami przetworzonymi są wszystkie odcinki, których oba końce znajdują się na lewo od miotły. Odcinkami aktywnymi są te, które przecinają miotłę. Do odcinków oczekujących zaliczają się odcinki o obu końcach położonych na prawo od miotły. Będziemy żądać, żeby był spełniony taki oto niezmiennik.

##### NIEZMIENNIK 1.

Pary odcinków przecinających się na lewo od miotły są już wyznaczone.

W y-strukturze są przechowywane odcinki aktywne, uporządkowane rosnąco względem współrzędnych y ich punktów przecięć z miotłą. Punkty te dzielą miotłę na przedziały.



Rys. 8.9. Znajdowanie najmniej odległej pary punktów metodą zamiatania

Do reprezentacji y-struktury używamy zrównoważonego drzewa poszukiwań (na przykład drzewa AVL) w sposób umożliwiający wykonywanie następujących operacji w czasie  $O(\log k)$ , gdzie  $k$  jest liczbą odcinków w y-strukturze:

- (a) *find( $p$ )*: szukanie odcinków aktywnych, których punkty przecięcia z miotłą wyznaczającą przedział zawierający punkt  $p$ ;
- (b) *insert( $l$ )*: dodanie odcinka aktywnego  $l$  do y-struktury;
- (c) *delete( $l$ )*: usunięcie odcinka  $l$  z y-struktury;
- (d) *pred( $l$ )*, *succ( $l$ )*: szukanie poprzednika (następnika) odcinka  $l$  w y-strukturze;

(e) *interchange*( $l, l'$ ): zmiana kolejności występowania w y-strukturze sąsiednich odcinków  $l$  i  $l'$ , tzn. jeśli przed wykonaniem *interchange*  $l$  poprzedza  $l'$ , to po jej wykonaniu  $l'$  poprzedza  $l$ .

W każdym położeniu miotły x-struktury zawiera tylko punkty leżące z jej prawej strony, a w szczególności punkty końcowe odcinków aktywnych oraz oczekujących. Dodatkowo żądamy, żeby był spełniony tak oto niezmiennik.

#### NIEZMIENNIK 2.

W x-strukturze znajdują się punkty przecięć, leżące na prawo od miotły, wszystkich par odcinków aktywnych, które kiedykolwiek były sąsiednimi w y-strukturze.

Niezmiennik 2 gwarantuje, że jeśli punkt przecięcia  $p$  pary odcinków leży najbliżej miotły ze wszystkich punktów (punktów przecięć bądź końców) położonych na prawo od niej odcinków, to  $p$  występuje w x-strukturze. Niezmiennik ten jest kluczowy dla poprawności przedstawianego algorytmu. Punkty w x-strukturze są uporządkowane rosnąco względem współrzędnej  $x$ . Do reprezentacji x-struktury używamy zrównoważonego drzewa poszukiwań, zaimplementowanego w taki sposób, żeby można było w czasie logarytmicznym wykonywać następujące operacje:

- (a) *findmin*:: wyznaczenie punktu z minimalną współrzędną  $x$ ;
- (b) *deletemin*:: usunięcie punktu z minimalną współrzędną;
- (c) *add*( $p$ ) :: sprawdzenie, czy punkt  $p$  jest w x-strukturze i dodanie  $p$  do x-struktury, gdy go tam nie ma.

Oto bardziej formalny opis algorytmu obliczania par przecinających się odcinków:

```

begin
  y-struktura := Ø; {początkowo y-struktura nie zawiera żadnych
  odcinków}
  x-struktura := 2n punktów końcowych odcinków  $l_1, \dots, l_n$ ,
  uporządkowanych rosnąco względem współrzędnej x;
  while x-struktura ≠ Ø do
    begin
      {*}   p := findmin; {punkt z minimalną współrzędną x
      w x-strukturze};
      deletemin; {usuń p z x-struktury}
      if p jest lewym końcem pewnego odcinka l then
      begin
        {znajdź (sąsiednie) odcinki  $l'$  i  $l''$  w y-strukturze,
        które wyznaczają przedział zawierający p, i wstaw l do
        y-struktury}
        ( $l', l''$ ) := Find(p); { $l'$  poprzedza  $l''$  w y-strukturze}
        insert(l);
        if odcinki  $l$  i  $l'$  przecinają się then
        begin
          {dodaj punkt przecięcia odcinków l i  $l'$  do
          y-struktury}
        end;
      end;
    end;
  end;
end;

```

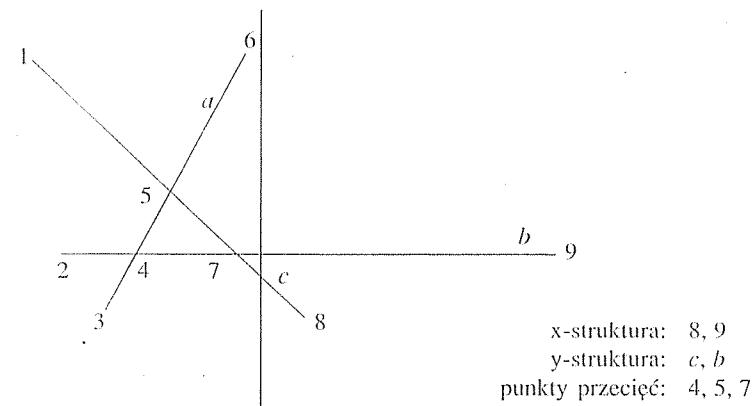
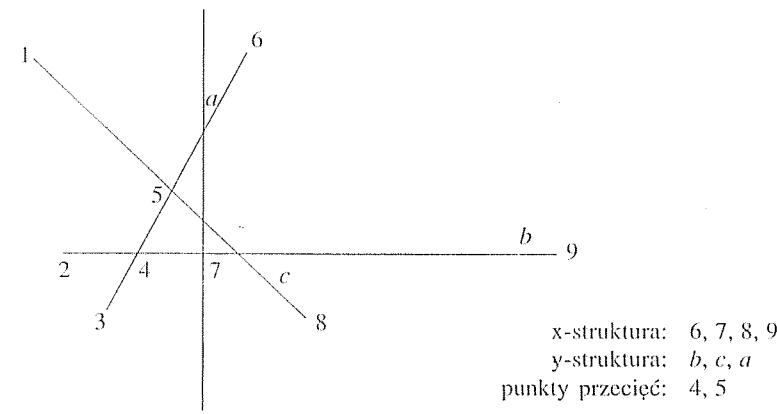
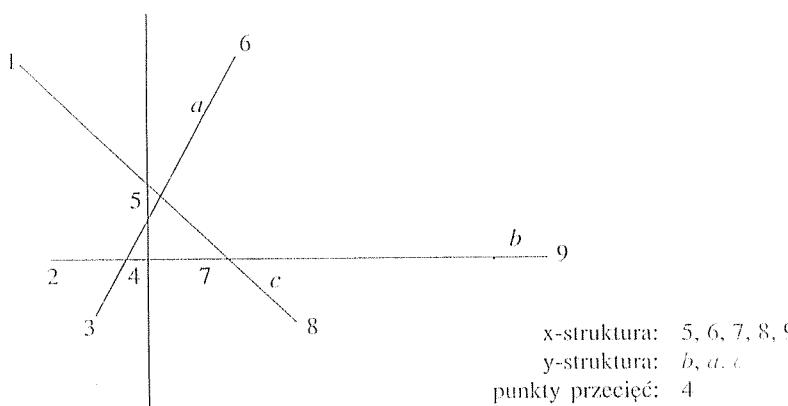
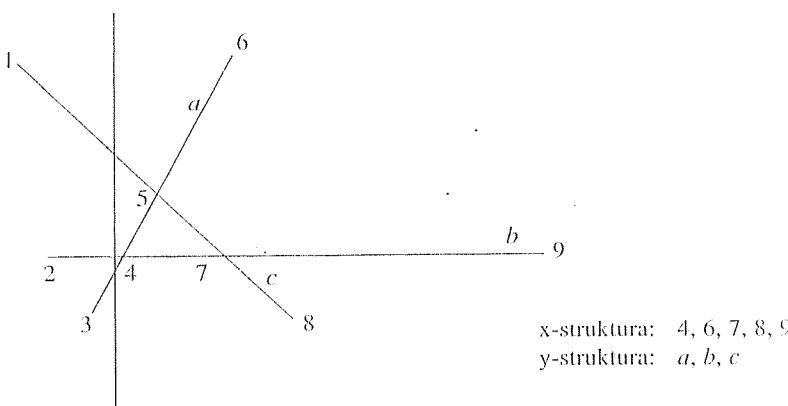
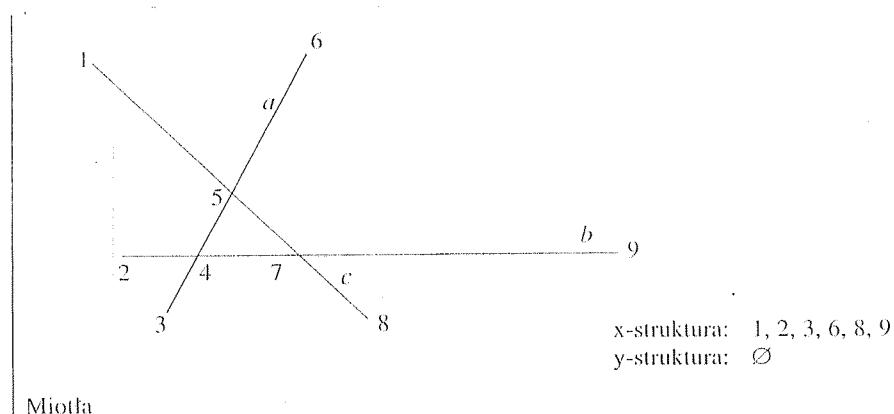
#### 8.4. Metoda zamiatania

```

x-struktury}
q := punkt przecięcia l i  $l'$ ; add(q)
end;
if odcinki l i  $l''$  przecinają się then
begin
  {dodaj punkt przecięcia odcinków l i  $l''$  do
  x-struktury}
  q := punkt przecięcia l i  $l''$ ; add(q)
end
end;
if p jest prawym końcem pewnego odcinka l then
begin
  l' := pred(l); l'' := succ(l);
  {l' i l'' są sąsiadami l w y-strukturze}
  {usuń l z y-struktury}
  delete(l);
  if odcinki  $l'$  i  $l''$  przecinają się na prawo od miotły then
  begin
    {dodaj punkt przecięcia  $l'$  i  $l''$  do x-struktury}
    q := punkt przecięcia  $l'$  i  $l''$ ; add(q)
  end
end;
if q jest punktem przecięcia odcinków  $l'$  i  $l''$  then
begin
  wypisz parę  $l'$  i  $l''$  jako parę przecinających się
  odcinków;
  {zamień  $l'$  z  $l''$  w y-strukturze}
  interchange( $l'$ ,  $l''$ );
  k := pred( $l''$ ); {k jest sąsiadem  $l''$  w y-strukturze
  różnym od  $l'$ }
  if odcinki k i  $l''$  przecinają się na prawo od miotły then
  begin
    {dodaj punkt przecięcia odcinków k i  $l''$  do
    x-struktury}
    q := punkt przecięcia k i  $l''$ ; add(q)
  end;
  m := succ( $l'$ ); {m jest sąsiadem  $l'$  w y-strukturze,
  różnym od  $l''$ }
  if odcinki m i  $l'$  przecinają się na prawo od miotły then
  begin
    {dodaj punkt przecięcia odcinków m i  $l'$  do
    x-struktury}
    q := punkt przecięcia m i  $l'$ ; add(q)
  end
end;
end {instrukcji while}
end;

```

Na rysunku 8.10 jest przedstawiony przykład działania algorytmu.



Rys. 8.10. Przykład działania algorytmu znajdowania par przecinających się odcinków metodą zamiatania

W celu uzasadnienia poprawności powyższego algorytmu wystarczy pokazać, że są zachowane niezmienniki. Mówimy, że punkt jest **krytyczny**, jeżeli jest końcem odcinka lub punktem przecięcia dwóch odcinków. Niezmiennik 2 gwarantuje, że punkt  $p$  wybierany w wierszu  $\{*\}$  jest punktem krytycznym położonym najbliżej miotły z prawej strony. Każdy taki punkt jest badany tylko raz, a następnie usuwany z x-struktury. Wynika stąd, że wszystkie punkty przecięć zostają znalezione. Wiersze oznaczone  $\{**\}$  gwarantują, że y-struktura zawiera tylko odcinki aktywne i to w dobrym porządku. Gwarancja zachowania niezmiennika 2 są wiersze oznaczone  $\{***\}$ . Liczba iteracji instrukcji **while** wynosi  $2n + s$ . W każdej iteracji jest wykonywana pewna stała liczba operacji na y-strukturze i x-strukturze. Każda z tych operacji kosztuje co najwyżej  $O(\log(n + s))$ . Ponieważ  $s \leq n^2$ , łączny koszt wykonania algorytmu jest  $O((n + s)\log n)$ . (Powyższe rozwiązanie zostało po raz pierwszy zaproponowane w pracy [BO]. Algorytm przedstawiony w tym opracowaniu pochodzi z [M]).

## Zadania

- 8.1. Ułóż nierekurencyjną wersję algorytmu sprawdzania przynależności punktu do wielokąta wypukłego.
- 8.2. Niech  $\alpha$  będzie funkcją zdefiniowaną w tym rozdziale. Udowodnij, że dla punktów  $p$  i  $q$  różnych od środka układu współrzędnych  $O \alpha(p) \leq \alpha(q)$  wtedy i tylko wtedy, gdy kąt nachylenia wektora wodzącego punktu  $p$  do osi  $OX$  jest nie większy od kąta nachylenia wektora wodzącego punktu  $q$ .
- 8.3. Zmodyfikuj algorytmy „dziel i zwyciężaj” oraz przyrostowy obliczania wypukłej otoczki. Zrób to tak, żeby działały dla każdego zbioru wejściowego punktów.
- 8.4. Przeprowadź dokładną analizę algorytmów „dziel i zwyciężaj” oraz przyrostowego do obliczania wypukłej otoczki.
- 8.5. Algorytm „dziel i zwyciężaj” dla problemu wypukłej otoczki jest podobny do algorytmu sortowania przez scalanie. Zaproponuj algorytm obliczania wypukłej otoczki analogiczny do algorytmu quicksort.
- 8.6. Zaproponuj strukturę danych umożliwiającą wykonywanie na dynamicznym zbiorze punktów  $S$  następujących operacji:
  - (a)  $insert(S, p):: S := S \cup \{p\}$  (czas wykonania  $O(\max(1, \log n))$ );
  - (b)  $CH(S)::$  podanie wypukłej otoczki dla  $S$  (czas wykonania  $O(\max(1, n))$ );

gdzie  $n$  jest liczbą elementów w zbiorze  $S$ .

- 8.7. Zaproponuj algorytm liniowy obliczania wypukłej otoczki dla wierzchołków wielokąta danych w kolejności ich występowania na jego obwodzie.
- 8.8. Zaproponuj algorytm liniowy obliczania wielokąta  $W$  będącego sumą danych wielokątów wypukłych  $W_1$  i  $W_2$ .
- 8.9. Zaproponuj algorytm liniowy obliczania wielokąta  $W$  będącego iloczynem danych wielokątów wypukłych  $W_1$  i  $W_2$ .
- 8.10. Ułóż algorytm sprawdzania w czasie liniowym, czy dane dwa wielokąty  $W_1$  i  $W_2$  są podobne. (Wskazówka: Sprawdź problem do równości słów cyklicznych, rozważając kąty wielokątów).
- 8.11. Zaproponuj algorytm typu „dziel i zwyciężaj” znajdowania najmniej odległej pary punktów wśród danych  $n$  punktów  $p_1, \dots, p_n$ .

## Zadania

- 8.12. Ułóż algorytm znajdowania w czasie  $O(n \log n)$  najbardziej odległej pary punktów wśród danych  $n$  punktów  $p_1, \dots, p_n$ .
- 8.13. Podaj dokładną implementację x- i y-struktur z algorytmu obliczania najmniej odległej pary punktów.
- 8.14. Podaj dokładną implementację x- i y-struktur z algorytmu obliczania par przecinających się odcinków.
- 8.15. Złożoność pamięciowa algorytmu obliczania przecinających się par odcinków jest  $O(s + n)$ . Zmodyfikuj algorytm w taki sposób, żeby złożoność pamięciowa była  $O(n)$ . (Zauważmy, że takie rozwiązanie jest znacznie oszczędniejsze pamięciowo wtedy, kiedy  $s$  jest rzędu  $n^2$ ).
- 8.16. Triangulację wielokąta  $W$  nazywamy podział  $W$  przekątnymi na trójkąty. Zaproponuj algorytm z wykorzystaniem metody zamiatania do znajdowania triangulacji danego wielokąta  $W$ .
- 8.17. Niech  $W$  będzie wielokątem o bokach równoległych do osi układu współrzędnych. Zastosuj metodę zamiatania do wyznaczania długości najkrótszej liniowej zwyczajnej, łączącej dwa punkty z  $W$  i całkowicie w nim zawartej.