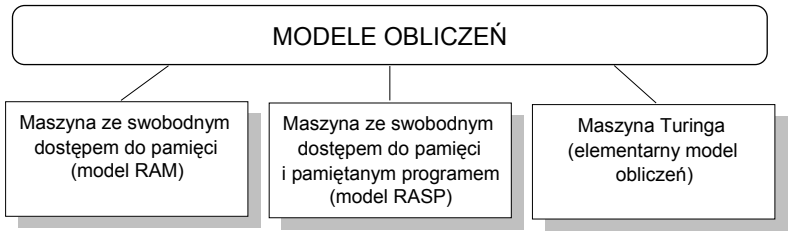
SPIS TREŚCI

1 Algorytmy i modele obliczeń

1.1 Pojęcia podstawowe

Pojęciem *algorytm* określa się najczęściej pewien przepis (sposób postępowania), który ma prowadzić do rozwiązania określonego zadania. Przyjmuje się, że przepis ten jest na tyle precyzyjny, że posługiwanie się nim polega wyłącznie na automatycznym wykonywaniu zawartych tam poleceń. Zakłada się dalej, że polecenia występujące w algorytmie są *wykonywalne*, tzn. dostępne, a pisząc algorytm wystarczy się nimi jedynie posłużyć. Zestaw dostępnych poleceń zależy od przyjętego modelu, w którym mają być realizowane obliczenia, określanego jako *model obliczeń*. Na rys.1.1 pokazano znane z literatury przykłady modeli obliczeń [1].

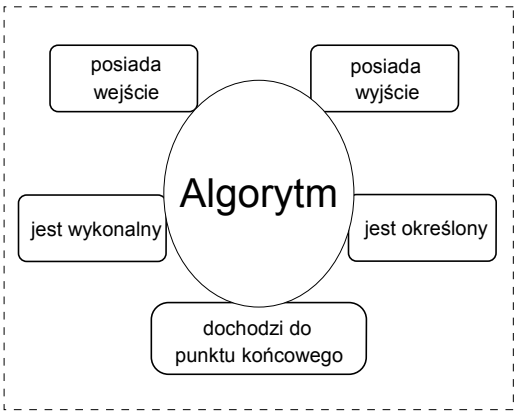


Rys.1.1 Przykłady modeli obliczeń.

Wykonalność można rozpatrywać także w odniesieniu do pewnego zbioru obiektów i dostępnych w nim pierwotnych operacji. Zbiór taki nazywa się *dziedziną algorytmiczną* [4].

Obok wykonalności, jako istotne dla algorytmów rozważa się także pojęcia *wejścia* i *wyjścia*. Wejście oznacza zwykle pewne dane pobierane przez algorytm w celu ich przetworzenia, podczas gdy wyjście odnosi się do wyniku działania algorytmu. Ważną właściwością wymaganą od algorytmu jest jego *skończoność*, co oznacza, że algorytm powinien zakończyć swoje działanie po wykonaniu skończonej liczby operacji. Dla algorytmów obowiązuje także

określoność, tzn. wymóg, aby każda operacja w algorytmie była sformułowana w sposób zapewniający jej jednoznaczną interpretację. Zestawienie wymienionych cech algorytmów pokazuje rys.1.2, a bardziej pełne ich omówienie zawiera praca [7].

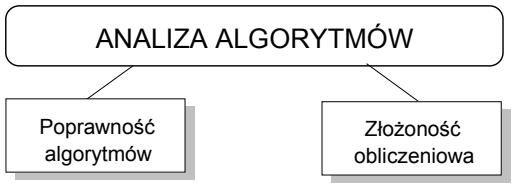


Rys.1.2 Cechy algorytmów.

Do podstawowych pytań pojawiających się w trakcie teoretycznych badań nad obliczeniami należą m.in.:

- Jak dla określonego zadania znaleźć efektywny algorytm?
- Jeśli algorytm już znaleziono, to jak porównać go z innymi, które rozwiązują podobne zadania?
- Jak udowodnić poprawność algorytmu?
- W jakim sensie można wykazać, że pewne algorytmy są „najlepszymi z możliwych”?

Spośród dziedzin zajmujących się algorytmami istotne znaczenie *ma analiza algorytmów*, której zasadnicze kierunki dotyczą *poprawności* i *złożoności obliczeniowej* (rys.1.3).



Rys.1.3 Kierunki badań analizy algorytmów.

Poprawność dotyczy przede wszystkim pojęć i metod związanych z dowodzeniem poprawności algorytmów, takich jak: własność stopu, częściowa poprawność, metoda niezmienników itp. Zagadnienia te są na tyle ważne, aby poświęcić im odrębny wykład i nie będą tutaj omawiane.

Dziedziną, której poświęcimy znacznie więcej miejsca będzie natomiast złożoność obliczeniowa algorytmów i związane z nią pojęcia, do których należą m.in.: złożoność czasowa, pamięciowa, średnia i pesymistyczna. Ten obszar analizy algorytmów koncentruje się na określaniu zasobów (czas obliczeń i pamięć), jakie są potrzebne do wykonania badanego algorytmu.

1.2 Złożoność algorytmu

Oceny złożoności algorytmu można dokonać według różnych kryteriów. Często stosowana jest metoda polegająca na badaniu, jak zwiększa się czas obliczeń lub wielkość pamięci potrzebnej do wykonania algorytmu, w miarę wzrostu rozmiaru danych wejściowych. Rozmiar tych danych nazywać będziemy *rozmiarem zadania*, albo *rozmiarem wejścia*, przyjmując, że jest to konkretna liczba charakteryzująca objętość danych wejściowych. Jako przykłady rozmiarów wejścia można wymienić:

- liczbę elementów tablicy wejściowej w algorytmach sortowania,
- liczbę krawędzi grafu w algorytmach operujących na grafach,
- wymiary macierzy w zadaniu mnożenia macierzy.

Czas potrzebny na wykonanie algorytmu jako funkcja rozmiaru zadania jest nazywany *złożonością czasową* tego algorytmu. Charakter tej złożoności przy dążeniu do wartości granicznej wraz ze wzrostem rozmiaru zadania jest określany jako *asymptotyczna złożoność czasowa*. W analogiczny sposób można zdefiniować *pojęcia złożoności pamięciowej* i *asymptotycznej złożoności pamięciowej*.

Wprowadzenie asymptotycznej złożoności algorytmu jest szczególnie ważne, gdyż przy jej pomocy można określić rozmiar zadań, które mogą być rozwiązane za pomocą tego algorytmu. Jeśli np. algorytm przetwarza wejście o rozmiarze n, w czasie cn^2, gdzie c jest pewną stałą, wówczas mówimy, że złożoność czasowa tego algorytmu jest rzędu n^2, co zapisujemy jako $O(n^2)$ i

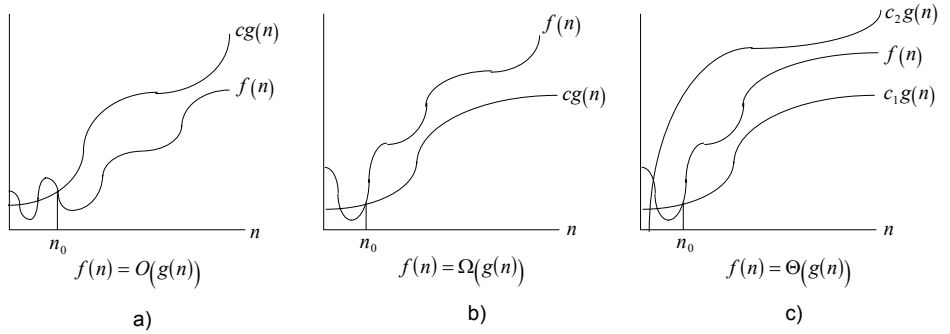
czytamy "duże o od n kwadrat". Ogólnie biorąc, pewna nieujemna funkcja $f(n)$ jest $O(g(n))$, jeśli istnieje taka stała c, że

$$0 \le f(n) \le cg(n)$$

dla wszystkich n oprócz pewnego, być może pustego, zbioru nieujemnych wartości n.

Symbol $O(g(n))$ jest zaliczany do tzw. *notacji asymptotycznych*, używanych do opisu asymptotycznego czasu działania algorytmów. Na rysunku 1.4a podano interpretację graficzną notacji O. Wynika z niej, że przy pomocy zapisu $f(n) = O(g(n))$ podaje się górne ograniczenie funkcji z dokładnością do stałego współczynnika. Bardziej precyzyjnie zapis ten oznacza, że istnieją dodatnie stałe c i n_0 takie, że na prawo od n_0 wartość $f(n)$ jest zawsze nie większa od $cg(n)$.

Ponadto na rys.1.4b i 1.4c przedstawiono w podobny sposób notacje Θ i Ω, będące także przykładami notacji asymptotycznych.



Rys.1.4 Graficzna interpretacja notacji asymptotycznych.

W odróżnieniu od notacji O, która ogranicza funkcję z góry, notacja Ω (rys.1.4b) ogranicza funkcję z dołu. Zapis $f(n) = \Omega(g(n))$ oznacza, że istnieją dodatnie stałe c i n_0 takie, że na prawo od n_0 wartość $f(n)$ jest zawsze większa od $cg(n)$. Przedstawiona na rys.1.4c notacja pozwala szacować funkcję z dołu i z góry. Pisząc zatem $f(n) = \Theta(g(n))$ wskazujemy, że istnieją

dodatnie stałe c_1, c_2 i n_0 takie, że na prawo od n_0 wartość $f(n)$ jest zawsze większa od $c_1 g(n)$ i nie większa od $c_2 g(n)$.

Przykład 1.1

Przyjmując

$$f(n) = \frac{1}{2}n^2 - 3n \text{ oraz } g(n) = n^2$$

wykażemy, że $f(n) = \Theta\big(g(n)\big)$.

Aby tego dokonać, należy wyznaczyć dodatnie stałe c_1, c_2 i n_0 takie, że

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \qquad \text{dla } n \ge n_0. \tag{1.1}$$

Dzieląc nierówność (1.1) przez n^2 otrzymujemy zależność

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

z której wynika, że nierówność $\frac{1}{2} - \frac{3}{n} \le c_2$ jest spełniona dla każdego $n \ge 1$, jeśli przyjąć $c_2 = \frac{1}{2}$, a nierówność $c_1 \le \frac{1}{2} - \frac{3}{n}$ jest spełniona dla każdego $n \ge 7$, jeśli przyjąć $c_1 = \frac{1}{14}$. Stąd dla $c_1 = \frac{1}{14}$, $c_2 = \frac{1}{2}$ i $n_0 = 7$ zachodzi nierówność (1.1), co oznacza, że $f(n) = \Theta\big(g(n)\big)$.

Mogłoby się wydawać, że imponujący wzrost szybkości współczesnych komputerów powoduje, że prace nad poszukiwaniem efektywnych algorytmów nie są uzasadnione. Celem kolejnych dwóch przykładów będzie wykazanie, że tak nie jest, gdyż efektywne algorytmy mogą w sposób nie mniej istotny jak rozwój sprzętu wpływać na wydajność komputerów (mierzoną np. rozmiarem zadań, które mogą być rozwiązywane).

Przykład 1.2 [1]

W dwóch pierwszych kolumnach tab.1.1 zamieszczono pięć przykładowych algorytmów wraz z ich złożonością czasową, rozumianą tutaj jako liczba jednostek czasu potrzebnych do obróbki (przetworzenia) wejścia o rozmiarze n. Jeśli za jednostkę czasu przyjąć jedną milisekundę, to algorytm A_1 może w ciągu sekundy przetworzyć wejście o rozmiarze 1000, natomiast algorytm A_5 wejście o rozmiarze zaledwie 9.

Tab.1.1 Złożoności czasowe i maksymalne rozmiary zadań dla wybranych algorytmów.

Algorytm	Złożoność czasowa	Maksymalny rozmiar zadania		
		1 sekunda	1 minuta	1 godzina
A_1	n	1000	$6 \cdot 10^4$	$3{,}6 \cdot 10^6$
A_2	$n \lg n$	140	4893	$2{,}0 \cdot 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

W kolejnych kolumnach tab.1.1 zamieszczono maksymalne rozmiary zadań które mogą być przetworzone przez rozważane algorytmy, jeśli za całkowity czas obliczeń przyjąć odpowiednio 1 sekundę, 1 minutę i 1 godzinę.

Załóżmy dalej, że w pewnym okresie szybkość komputerów wzrasta 10-krotnie. W tab.1.2 pokazano, jak wzrosną rozmiary zadań, które mogą być przetworzone przez algorytmy $A_1 \div A_5$, w wyniku takiego wzrostu szybkości komputerów.

Tab.1.2 Wpływ zwiększenia szybkości komputera na maksymalny rozmiar zadania.

Algorytm	Złożoność czasowa	Maksymalny rozmiar zadania	
		przed zwiększeniem	po zwiększeniu
A_1	n	s_1	$10s_1$
A_2	$n \lg n$	s_2	$\sim 10s_2$ dla dużych s_2
A_3	n^2	s_3	$3{,}16\, s_3$
A_4	n^3	s_4	$2{,}15 s_4$
A_5	2^n	s_5	$s_5 + 3{,}3$

Zauważmy, że dla algorytmu A_5 10-krotne zwiększenie prędkości komputera pozwala zwiększyć rozmiar zadania, które może być rozwiązane zaledwie o 3, podczas gdy w przypadku algorytmu A_3 rozmiar ten się potraja.

Porównajmy teraz efekty zwiększenia szybkości komputerów z efektami zastosowania bardziej efektywnego algorytmu. W tym celu powróćmy do tab.1.1 i przyjmijmy za podstawę porównania jedną minutę. Widać przy tym, że zamiana algorytmu A_4 na algorytm A_3 pozwala rozwiązać zadanie 6-krotnie większe. Natomiast zamiana A_4 na A_2 pozwala rozwiązać zadanie o rozmiarze 125-krotnie większym. Wyniki te oznaczają, że polepszenie osiągów spowodowane zastosowaniem bardziej efektywnego algorytmu może wyraźnie przewyższyć zyski wynikające ze zwiększenia szybkości komputerów.

Przykład 1.3 [5]

Zakłada się, że równolegle na dwóch komputerach o różnej mocy obliczeniowej jest sortowana tablica zawierająca milion elementów ($n = 1000000$).

Przyjmujemy dalej, że na superkomputerze wykonującym 100 milionów operacji na sekundę zastosowano algorytm sortowania o złożoności $2n^2$, a na mikrokomputerze wykonującym 1 milion operacji na sekundę zastosowano inny algorytm sortowania o złożoności $50n \lg n$. Przedstawimy teraz szacunkowe obliczenia czasów jakie potrzebują oba komputery na posortowanie tablicy wejściowej.

Dla superkomputera:

$$\frac{2 \cdot \left(10^6\right)^2 instrukcji}{10^8 instrukcji / sek.} \approx 20000 sek. \approx 5{,}56 godz.$$

Dla mikrokomputera:

$$\frac{50 \cdot 10^6 \lg 10^6 instrukcji}{10^6 instrukcji / sek.} \approx 1000 sek. \approx 16{,}67 \min.$$

Używając zatem algorytmu, którego czas działania jest opisany funkcją niższego rzędu (nawet jeśli został on napisany przez słabszego programistę) mikrokomputer działa około 20-krotnie szybciej niż superkomputer.

W dotychczasowych rozważaniach zwracaliśmy uwagę przede wszystkim na to, jakiego rzędu jest wzrost złożoności. Należy jednak pamiętać o tym, że w niektórych przypadkach istotna może okazać się także wartość współczynnika stałego. W szczególności, przy mniejszym rozmiarze zadań, algorytmy o dużym stopniu wzrastania złożoności lecz mniejszym współczynniku stałym mogą okazać się bardziej efektywne od algorytmów o mniejszym stopniu wzrastania złożoności, ale dużym współczynniku stałym. Dla ilustracji przyjmijmy liczbę sortowanych elementów w ostatnim przykładzie jako n=10. W przypadku tak małego rozmiaru zadania algorytm użyty przez superkomputer wymaga jedynie 200 instrukcji, podczas gdy algorytm użyty przez mikrokomputer ok. 1500. Oznacza to, że ze względu na większy współczynnik stały w wyrażeniu na złożoność algorytmu korzyści z zastosowania dla mikrokomputera bardziej wydajnego algorytmu uwidocznią się dopiero dla zadań o większym rozmiarze.

1.3 Model obliczeniowy RAM

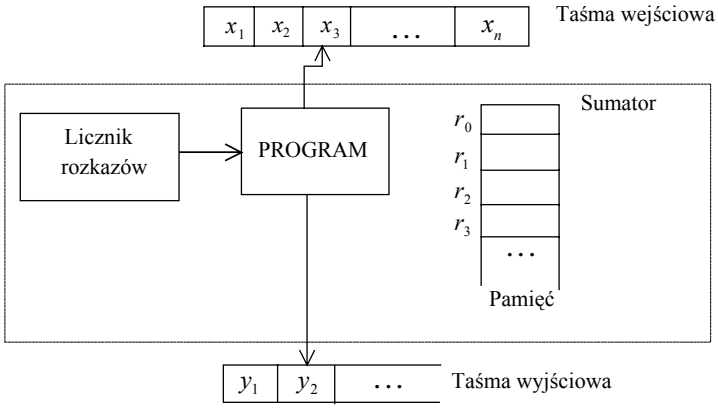
1.3.1 Opis maszyny RAM

Dalsze rozważania na temat algorytmów i ich złożoności poprzedzimy wprowadzeniem hipotetycznego modelu maszyny (urządzenia liczącego), za pomocą którego będą realizowane nasze algorytmy. Jednocześnie przyjmiemy pewne ustalenia odnośnie tego, co będzie określane pojęciem „elementarny krok obliczeń". Z literatury znane są różne modele obliczeń (por.rys.1.1), należy jednak stwierdzić, że nie istnieje model uniwersalny, który byłby najlepszy dla wszystkich przypadków.

Tutaj przedstawiony zostanie rozważany w [1] model maszyny ze swobodnym dostępem do pamięci (model RAM – *Random Access Machine*). Model ten reprezentuje maszynę z jednym sumatorem, w której program nie jest przechowywany w pamięci, a zatem nie może on modyfikować sam siebie. Schemat modelu RAM (maszyny RAM) pokazano na rys.1.5. Maszyna składa się z następujących trzech elementów:

- taśmy wejściowej, z której dane mogą być tylko czytane,
- taśmy wyjściowej, na którą dane mogą być tylko zapisywane,
- pamięci.

Taśma wejściowa stanowi ciąg klatek, z których każda może zawierać liczbę całkowitą. Każdorazowo, kiedy z taśmy wejściowej zostanie odczytana liczba, głowica czytająca jest przesuwana o jedną pozycję w prawo. *Analogicznie taśma wyjściowa* jest także ciągiem klatek (na początku pustych), do których maszyna może tylko zapisywać. Po zapisie do kolejnej klatki głowica zapisująca jest przesuwana o jedną klatkę w prawo. Pamięć składa się z rejestrów $r_0, r_1, r_2, r_3, \ldots$, z których każdy może przechowywać (pamiętać) dowolną liczbę całkowitą. Rejestr r_0 pełni w obliczeniach specjalną rolę i jest nazywany *sumatorem*. Na liczbę rejestrów nie nakładamy górnego ograniczenia, zakładając, że rozmiar zadania jest zawsze na tyle mały, że mieści się ono w pamięci, a liczby całkowite biorące udział w obliczeniach mieszczą się w pojedynczych komórkach (rejestrach).



Rys.1.5 Schemat maszyny RAM [1].

Program maszyny RAM to ciąg poleceń (inaczej: instrukcji, rozkazów lub komend), które mogą być poprzedzone etykietą. Licznik rozkazów zawiera numer kolejnego rozkazu do wykonania. Dokładny typ komend nie jest istotny, o ile przypominają one te, które występują w realnych komputerach. Przyjmiemy, że są dostępne komendy arytmetyczne, wejścia/wyjścia oraz komendy skoku bezwarunkowego i warunkowego, a także występuje możliwość adresowania pośredniego. Wszystkie obliczenia są prowadzone na sumatorze, którym jest rejestr r_0. Przykładowy zestaw poleceń maszyny RAM zamieszczono w tab.1.3.

Tab.1.3 Polecenia maszyny RAM.

L.p.	Kod operacji	Adres	L.p.	Kod operacji	Adres
1.	LOAD	operand	7.	READ	operand
2.	STORE	operand	8.	WRITE	operand
3.	ADD	operand	9.	JUMP	etykieta
4.	SUB	operand	10.	JGTZ	etykieta
5.	MULT	operand	11.	JZERO	etykieta
6.	DIV	operand	12.	HALT	

W ogólnym przypadku komenda składa się z kodu operacji i adresu, który może być operandem lub etykietą (wyjątkiem jest komenda HALT, gdzie adres nie występuje). Etykietą jest nazwa symboliczna, a operand może należeć do jednego z trzech następujących typów:

$=i$ - liczba całkowita i; operand tego typu nazywamy *literałem*,

i - zawartość rejestru o numerze i (i powinno być nieujemne),

$*i$ - wartością operandu jest zawartość rejestru j, gdzie j jest liczbą nieujemną przechowywaną w rejestrze i (adresacja pośrednia).

Rozważymy dalej pewne odwzorowanie c zbioru liczb całkowitych nieujemnych na zbiór liczb całkowitych takie, że $c(i)$ oznacza zawartość rejestru o numerze i. Odwzorowanie c pozwala zatem przedstawić mapę pamięci (rejestrów) maszyny RAM i wraz z zawartością licznika rozkazów definiuje tzw. *znaczenie programu* w danej chwili. Na początku przyjmuje się $c(i) = 0,\ i \geq 0$, oraz zakłada, że licznik rozkazów wskazuje na pierwszą komendę do wykonania, a taśma wyjściowa jest pusta. Po wykonaniu pewnego rozkazu k licznik rozkazów automatycznie wskazuje na rozkaz $(k+1)$ za wyjątkiem przypadków, kiedy rozkazem k-tym był któryś z następujących rozkazów: JUMP, JGTZ, JZERO lub HALT.

Aby w pewien formalny sposób opisać działanie komend z tab.1.3 wprowadzimy jeszcze funkcję $v(a)$ wyznaczającą wartość operandu a w następujący sposób:

$$v(=i) = i \qquad v(i) = c(i) \qquad v(*i) = c(c(i)).$$

Opis komend maszyny RAM, z wykorzystaniem wprowadzonych notacji zestawiono w tab.1.4.

Tab.1.4 Opis poleceń maszyny RAM.

Komenda	Opis działania
LOAD a	$c(0) \leftarrow v(a)$, co oznacza zapis wartości operandu a do rejestru 0 (sumatora).
STORE i	$c(i) \leftarrow c(0)$
STORE $*i$	$c(c(i)) \leftarrow c(0)$
ADD a	$c(0) \leftarrow c(0) + v(a)$
SUB a	$c(0) \leftarrow c(0) - v(a)$
MULT a	$c(0) \leftarrow c(0) \times v(a)$
DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
READ i	$c(i) \leftarrow$ kolejny symbol wejściowy
READ $*i$	$c(c(i)) \leftarrow$ kolejny symbol wejściowy
WRITE a	$v(a)$ jest zapisywane do tej klatki taśmy wyjściowej, przy której aktualnie znajduje się głowica
JUMP b	Licznik rozkazów jest ustawiany na komendę z etykietą b
JGTZ b	Jeśli $c(0) > 0$, to licznik rozkazów jest ustawiany na komendę z etykietą b; w przeciwnym razie na komendę następną
JZERO b	Jeśli $c(0) = 0$, to licznik rozkazów jest ustawiany na komendę z etykietą b; w przeciwnym razie na komendę następną
HALT	Zatrzymanie programu

Przykład 1.4

Wyznaczyć wartość funkcji:

$$f(n) = \begin{cases} 0 & \text{dla } n = 0 \\ n^n & \text{dla } n \geq 1 \end{cases}.$$

Odpowiedni program dla maszyny RAM ma postać:

Etykieta	Komenda	Adres	Komentarz
	READ	1	*Wczytaj liczbę n do rejestru 1*

```
            LOAD    1               Umieść n  w sumatorze
            JGTZ    dod             Wykonuj obliczenia tylko wtedy, gdy
                                    n>0
            WRITE   =0              W przypadku gdy n=0 wypisz 0
            JUMP    Koniecjesli     Skocz na koniec programu
dod:        LOAD    1               Umieść n w sumatorze
            STORE   2               Zapisz n do rejestru 2
            LOAD    1               Umieść n w sumatorze
            SUB     =1              Oblicz n-1
            STORE   3               Zapisz n-1 do rejestru 3
dopoki:     LOAD    3               Umieść zawartość rejestru 3( licznik
                                    pętli) w sumatorze
            JGTZ    Kontyn          Kontynuuj, jeśli w rejestrze 3 jest
                                    liczba większa od 0
            JUMP    Koniecdopoki    Skocz do wypisania wyniku
Kontyn:     LOAD    2               Zapisz iloczyn cząstkowy do sumatora
            MULT    1               Pomnóż iloczyn cząstkowy przez n
            STORE   2               Zapisz nowy iloczyn cząstkowy do
                                    rejestru 2
            LOAD    3               Umieść licznik pętli w sumatorze
            SUB     =1              Odejmij 1 od licznika pętli
            STORE   3               Zapisz nową wartość licznika pętli w
                                    sumatorze
            JUMP    Dopoki          Skocz do sprawdzenia warunku pętli
Koniecdopoki: WRITE 2               Wypisz zawartość rejestru 2 (tj.
                                    obliczone n^n )
Koniecjesli: HALT                   Koniec programu
```

1.3.2 Uproszczony język programowania

Celem uzyskania bardziej przejrzystego zapisu algorytmów wykonywalnych w ramach modelu RAM dość często używa się uproszczonego języka wyższego poziomu, określanego jako *pseudojęzyk* lub *pseudokod* [1, 5]. W prezentowanych dalej przykładach najczęściej używany będzie język wzorowany na pseudokod zaproponowany w [5]. Poniżej przedstawiony zostanie pewien nieformalny opis tego języka.

1. Pseudokod tym różni się od typowego języka programowania, że dopuszcza dowolny typ opisu matematycznego, a nawet słownego, jeśli tylko jest on w pełni zrozumiały i może zostać przetłumaczony na komendy prostego modelu, np. modelu RAM.
2. Nie ustala się z góry dopuszczalnych typów danych, a zmiennymi są zwykle liczby i tablice. Dodatkowe typy, jak: listy, kolejki, stosy, grafy itp. są wprowadzane w razie potrzeby. Tam, gdzie to możliwe, unika się ich formalnego opisu.
3. Stosuje się tradycyjny zapis matematyczny oraz konstrukcje znane z języków programowania typu Pascal i C. Do podstawowych operatorów pseudokodu należą:

a) nadawanie wartości

zmienna ← *wyrażenie*

b) operator warunkowy

jeśli *warunek*
  to *operator(y)*
  inaczej *operator(y)*

c) operator pętli ***dopóki***

dopóki *warunek*
  wykonuj *operator(y)*

d) operator pętli ***powtarzaj***

powtarzaj *operator(y)*
aż_do *warunek*

e) operator pętli ***dla***

dla *zmienna* ← *wartość_początkowa* [***w_dół_***]***do*** *wartość_końcowa*
  wykonuj *operator(y)*

4. Blok poleceń pseudojęzyka (instrukcję złożoną) wydzielamy za pomocą nawisów klamrowych { i }. Można także stosować „wcięcia” celem zwiększenia czytelności programu.

5. Symbol „▷” oznacza, że reszta wiersza jest komentarzem.
6. Wielokrotne przypisywanie w postaci $i \leftarrow j \leftarrow e$ oznacza przypisanie zmiennym i oraz j wartości wyrażenia e. Jest ono równoważne przypisaniu $j \leftarrow e$ łącznie z przypisaniem $i \leftarrow j$.
7. Zmienne są z założenia lokalne w danej procedurze, chyba, że wyraźnie zaznaczono inaczej.
8. Dostęp do elementu tablicy odbywa się przez podanie jej nazwy oraz indeksu żądanego elementu w nawiasie kwadratowym, np. $A[i]$ jest i-tym elementem tablicy A. Zapis $A[1..j]$ oznacza natomiast podtablicę tablicy A złożoną z elementów $A[1], A[2],...A[j]$.
9. Dane złożone z kilku części są zorganizowane jako obiekty złożone z *atrybutów* (pól). Odwołanie do konkretnego atrybutu pewnego obiektu następuje przez podanie nazwy tego obiektu w nawiasach kwadratowych poprzedzonej nazwą atrybutu, np. $f[x]$ jest odwołaniem do atrybutu f obiektu x.
10. Zmienna odpowiadająca tablicy lub obiektowi jest traktowana jako wskaźnik do danych reprezentujących tę tablicę lub ten obiekt. Dla wszystkich pól f obiektu x przypisanie $y \leftarrow x$ powoduje $f[y] = f[x]$. Ponadto jeśli teraz wykonamy $f[x] \leftarrow 3$, to w następstwie nie tylko $f[x] = 3$, lecz także $f[y] = 3$. Wskaźnik, który na nic nie wskazuje posiada specjalną wartość *nil*.
11. Parametry są przekazywane do procedury przez wartość, tzn., że wywoływana procedura otrzymuje swoją kopię parametrów, a zmiany wewnętrzne tych kopii nie są widoczne przez program wywołujący. Obiekty są przekazywane przez wskaźniki do nich. Jeśli na przykład obiekt x jest parametrem procedury, to przypisanie $x \leftarrow y$ wewnątrz procedury nie jest widoczne na zewnątrz, ale przypisanie $f[x] \leftarrow 3$ jest widoczne.

Przykład 1.5

Zapisany w pseudokodzie algorytm K.1.1 oblicza wartość funkcji z przykładu 1.4.

K.1.1

```
POTĘGA-N-DO-N
1 czytaj r1
2 jeśli r1≤0 to pisz 0
3  inaczej { r2←r1
4    r3←r1-1
5    dopóki r3>0
6      wykonuj { r2←r2*r1
7        r3←r3-1 }
8    pisz r2 }
```

Numerowanie wierszy programu w pseudokodzie pozwala łatwiej odwoływać się do tych wierszy w opisach prezentowanych algorytmów.

Programy w pseudokodzie są równoważne programom RAM w tym sensie, że jest możliwe ich tłumaczenie na program w języku maszyny RAM. Problem takiej translacji stanowi jednak odrębne zagadnienie i nie będzie tutaj omawiany.

W literaturze z dziedziny algorytmów obok pseudokodów zbliżonych do zdefiniowanego, są używane także popularne języki programowania, np. Pascal [3, 4] lub C [6].

1.3.3 Złożoność pesymistyczna i oczekiwana

Jak już wspomniano, dwa podstawowe wskaźniki efektywności algorytmów stanowią złożoność pamięciowa i złożoność czasowa będące funkcjami rozmiaru wejścia. Za jednostkę złożoności pamięciowej przyjmuje się zwykle *słowo pamięci maszyny*. Natomiast w przypadku złożoności czasowej, która powinna być niezależna od komputera, języka programowania i sposobu zakodowania programu, rozważa się tzw. *operacje dominujące*. Są to operacje charakterystyczne dla danego algorytmu, a ich liczba jest proporcjonalna do liczby wykonań wszystkich operacji jednostkowych w dowolnej realizacji komputerowej danego algorytmu [3]. Na przykład w algorytmach wyznaczania wartości wielomianów mogą to być podstawowe operacje arytmetyczne, a w algorytmach sortowania – operacje porównania dwóch elementów sortowanej

tablicy. Za jednostkę złożoności czasowej przyjmuje się wtedy *wykonanie jednej operacji dominującej*.

Jeśli dla ustalonego rozmiaru wejścia w charakterze miary złożoności przyjąć największą ze złożoności dla wszystkich możliwych wejść danego rozmiaru, to taka złożoność jest określana jako *pesymistyczna*. Jeśli natomiast jako miarę złożoności przyjąć pewną „średnią" złożoność dla wszystkich wejść danego rozmiaru to taką złożoność określamy jako *oczekiwaną*. Zazwyczaj złożoność oczekiwana jest trudniejsza do określenia niż złożoność pesymistyczna.

Pojęcia złożoności pesymistycznej i złożoności oczekiwanej można także wprowadzić w sposób formalny. Uczynimy to w stosunku do złożoności czasowej używając następujących oznaczeń:

$n = |d|$ - rozmiar zestawu danych d,

D_n - zbiór zestawów danych wejściowych rozmiaru n,

$r(d)$ - Liczba operacji dominujących dla zestawu danych wejściowych d,

X_n - Zmienna losowa, której wartością jest $r(d)$ 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. W pewnych przypadkach można założyć, że jest to rozkład równomierny, tzn. każdy zestaw danych rozmiaru n może pojawić się na wejściu algorytmu z jednakowym prawdopodobieństwem.

Definicja 1.1

Pesymistyczna złożoność czasowa algorytmu to funkcja rozmiaru wejścia określona następująco:

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

gdzie sup oznacza górny kres zbioru.

Definicja 1.2

Oczekiwana złożoność czasowa to funkcja rozmiaru wejścia określona następująco

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

Jest to zatem wartość oczekiwana zmiennej losowej X_n.

1.3.4 Złożoność programów w języku RAM

W celu otrzymania złożoności czasowej i pamięciowej programów RAM należy określić czas niezbędny do wykonania każdej komendy RAM oraz objętość pamięci używanej przez każdy z rejestrów, przyjmując przy tym pewne kryteria wagowe. Rozważa się następujące kryteria [1]:

- równomierne kryterium wagowe oraz
- logarytmiczne kryterium wagowe.

W przypadku równomiernego kryterium wagowego zakłada się, że każda komenda RAM potrzebuje na jej wykonanie jednej jednostki czasu, a każdy rejestr wykorzystuje jedną jednostkę pamięci.

Logarytmiczne kryterium wagowe jest oparte na założeniu, że koszt wykonania każdej komendy (jej waga) jest proporcjonalna do długości operandów. Rozważymy następującą funkcję logarytmiczną określoną na zbiorze liczb całkowitych

$$l(i) = \begin{cases} \lfloor \lg |i| \rfloor + 1, & i \neq 0, \\ 1, & i = 0. \end{cases}$$

Postać tej funkcji uwzględnia fakt, że dwójkowy zapis liczby całkowitej i w rejestrze wymaga $\lfloor \lg |i| \rfloor + 1$ bitów (zapis lg oznacza $\log_2$).

W celu wyznaczenia *logarytmicznej złożoności czasowej* programów w języku RAM, określa się wagi logarytmiczne dla poszczególnych typów operandów oraz komend maszyny RAM. Wagi logarytmiczne $t(a)$ dla wszystkich trzech typów operandów podano w tab.1.5, a wagi dla komend w tab.1.6.

Tab.1.5 Wagi logarytmiczne operandów.

Operand a	**Waga logarytmiczna** $t(a)$
$=i$	$l(i)$
i	$l(i)+l(c(i))$
$*i$	$l(i)+l(c(i))+l(c(c(i)))$

Tab.1.6 Wagi logarytmiczne komend.

Komenda	**Waga logarytmiczna**
LOAD a	$t(a)$
STORE i	$l(c(0))+l(i)$
STORE $*i$	$l(c(0))+l(i)+l(c(i))$
ADD a	$l(c(0))+t(a)$
SUB a	$l(c(0))+t(a)$
MULT a	$l(c(0))+t(a)$
DIV a	$l(c(0))+t(a)$
READ i	$l(wejscie)+l(i)$
READ $*i$	$l(wejscie)+l(i)+l(c(i))$
WRITE a	$t(a)$
JUMP b	1
JGTZ b	$l(c(0))$
JZERO b	$l(c(0))$
HALT	1

Dla przykładu wyznaczymy wagę logarytmiczną komendy ADD *i. Najpierw jest określany stopień trudności w dekodowaniu adresu operandu *i. Przeglądanie liczby całkowitej i zajmuje czas $l(i)$. Następnie należy odczytać zawartość $c(i)$ rejestru i pozwalającą ustalić rejestr gdzie umieszczono dodawaną liczbę, co wymaga czasu $l(c(i))$. Wreszcie odczyt zawartości $c(c(i))$ rejestru $c(i)$, tj. samej adresowanej liczby wymaga czasu $l(c(c(i)))$. W sumie odczyt dodawanej liczby zajmuje $l(i)+l(c(i))+l(c(c(i)))$ jednostek czasu. Ponieważ komenda ADD*i dodaje liczbę całkowitą $c(c(i))$ do liczby całkowitej

$c(0)$ w sumatorze, zatem waga dla całej komendy jest równa sumie $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$.

Rozważymy teraz problem określania *logarytmicznej złożoności pamięciowej* programu RAM. Przyjmuje się, że jest to suma

$$\sum_i l(x_i),$$

w której uwzględniono wszystkie rejestry *i* (łącznie z sumatorem) biorące udział w obliczeniach, a przez x_i oznaczono największą wartość bezwzględną liczby całkowitej, która w trakcie obliczeń pojawiła się w *i*-tym rejestrze.

Przykład 1.6

Dokonać oceny złożoności obliczeniowej programu obliczającego wartość funkcji z przykł.1.4.

Nietrudno zauważyć, że ze względu na złożoność czasową dominującym fragmentem jest pętla zawierająca komendę MULT. Gdy komenda ta wykonuje się po raz *i*-ty, to sumator zawiera n^i, a rejestr 1 zawiera n. W sumie komenda MULT wykonuje się $n-1$ razy. Przyjmując równomierne kryterium wagowe, każda z komend MULT wymaga czasu równego jednej jednostce, a zatem na wykonanie wszystkich tych komend potrzebny jest czas $O(n)$. W przypadku kryterium logarytmicznego *i*-ta komenda MULT zajmuje czas

$$l(n^i) + l(n) \approx (i+1)\lg n$$

a zatem czas wykonania wszystkich komend MULT jest równy

$$\sum_{i=1}^{n-1} (i+1)\lg n,$$

tzn. $O(n^2 \lg n)$.

Złożoność pamięciowa jest wyznaczana wielkością liczb przechowywanych w rejestrach od 0 do 3. Dla równomiernego kryterium wagowego złożoność ta jest pewną stałą, a jej asymptotyczną granicę górną oznaczać będziemy jako $O(1)$. Ponieważ największą z przechowywanych w rejestrach liczb jest n^n, a $l(n^n) \approx n \lg n$, stąd w przypadku kryterium logarytmicznego otrzymujemy złożoność pamięciową $O(n \lg n)$.

Należy zauważyć, że dla tego programu równomierne kryterium wagowe jest uzasadnione tylko w przypadku, jeśli liczba n^n może być zapisana w postaci jednego słowa maszynowego. Jeśli tak nie jest, wówczas nawet logarytmiczna złożoność czasowa jest do pewnego stopnia nierealistyczna, gdyż zakłada ona, że dwie liczby całkowite *i* i *j* mogą być pomnożone w czasie $O(l(i)+l(j))$, co nie jest bynajmniej oczywiste [1].

1.3.5 Złożoność programów w pseudokodzie

Wprawdzie podstawowe miary złożoności zostały przez nas określone w terminach modelu obliczeniowego RAM, lecz w wielu przypadkach ocena złożoności konkretnego algorytmu może być dokonana na podstawie jego zapisu w pseudokodzie.

Podobnie jak dla języka RAM, także w tym przypadku istotny będzie czas i pamięć potrzebne do wykonania komend pseudokodu. Sposób oceny złożoności czasowej algorytmu zapisanego w pseudokodzie ilustruje rozważany w [5] przykład sortowania przez wstawianie. Algorytm ten przedstawiono w pierwszej kolumnie tab.1.7, a koszt wykonania poszczególnych instrukcji i liczbę ich wykonań odpowiednio w kolumnach drugiej i trzeciej.

Przyjęto, że *n* oznacza liczbę elementów tablicy A, tzn. jest równe wartości atrybutu $length[A]$. Przez t_j oznaczono liczbę sprawdzeń warunku wejścia do pętli **dopóki** w wierszu 5 dla pewnej wartości *j*. W celu oceny złożoności obliczeniowej oblicza się całkowity czas działania algorytmu, który jest sumą czasów działania poszczególnych instrukcji. Jeśli koszt (czas wykonania) pewnej instrukcji powtarzanej n razy wynosi c_i, to całkowity koszt wykonania tej instrukcji wynosi $c_i n$.

Tab.1.7 Wpływ instrukcji algorytmu sortowania przez wstawianie na jego złożoność.

Instrukcje	**Koszt**	**Liczba wykonań**
SORT-WSTAW *(A)*		
1 **dla** *j*←2 **do** *length*[*A*]	c_1	n
2 **wykonuj** { *key*←*A*[*j*]	c_2	$n-1$
3 ▷ Wstaw *A*[*j*] do posortowanego ciągu *A*[1..*j*-1].	0	$n-1$
4 *i*←*j*-1	c_4	$n-1$
5 **dopóki** *i*>0 i *A*[*i*]>*key*	c_5	$\sum_{j=2}^{n} t_j$
6 **wykonuj** { *A*[*i*+1] ←*A*[*i*]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 *i*←*i-1* }	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 *A*[*i*+1]←*key* }	c_8	$n-1$

Sumując koszty wykonania poszczególnych instrukcji z tab.1.7 otrzymujemy

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

Zauważmy, że w przypadku, gdy tablica wejściowa jest już posortowana (tzw. przypadek *optymistyczny*), mamy

$$t_j = 1 \quad \text{dla} \quad j = 2,3,...,n,$$

co daje minimalny czas działania algorytmu równy

$$\begin{aligned} M(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Jak widać, wielkość ta jest *funkcją liniową* względem n.

W przypadku *pesymistycznym* (tablica jest posortowana w porządku odwrotnym) mamy

$$t_j = j \quad \text{dla} \quad j = 2,3,...,n.$$

Z uwględnieniem zależności

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ oraz } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

otrzymujemy

$$W(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) +$$

$$+ c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) =$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 - \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8),$$

co oznacza, że pesymistyczna złożoność czasowa tego algorytmu jest *funkcją kwadratową* względem n. Stosując poznaną notację asymptotyczną zapiszemy, że pesymistyczny czas działania algorytmu sortowania przez wstawianie wynosi $O(n^2)$. W analizowanych dalej algorytmach zwykle będziemy rozważać przypadek pesymistyczny.

Pytania kontrolne

1. Podać definicję oraz charakterystyczne cechy algorytmu.
2. Określić następujące pojęcia:
   a) analiza algorytmów;
   b) złożoność obliczeniowa algorytmu;
   c) poprawność algorytmu;
   d) asymptotyczna złożoność czasowa;
   e) rozmiar zadania.
3. Podać definicję i ilustrację graficzną asymptotycznej granicy górnej.
4. Podać definicję i ilustrację graficzną asymptotycznej granicy dolnej.
5. Wyjaśnić sens zapisu $f(n) = \Theta(g(n))$.
6. Scharakteryzować model obliczeniowy RAM.
7. Omówić komendy maszyny RAM.

8. Wymienić podstawowe konwencje zapisu algorytmów w pseudojęzyku.

9. Określić następujące pojęcia:
   a) złożoność pamięciowa;
   b) operacja dominująca;
   c) złożoność czasowa;
   d) złożoność pesymistyczna;
   e) złożoność oczekiwana (średnia).

10. Podać formalną definicję złożoności pesymistycznej i złożoności oczekiwanej.

11. Na czym polegają: równomierne i logarytmiczne kryterium wagowe?

12. Podać wagi logarytmiczne operandów w poleceniach maszyny RAM.

13. Podać i uzasadnić wagi logarytmiczne komend LOAD, ADD, READ, JGTZ, HALT.

14. Podać i uzasadnić wagi logarytmiczne komend STORE, MULT, WRITE, JUMP, JZERO.

Zadania

1. Wykazać, że:

   a) $6n^2 = O(n^3)$; b) $6n^3 \neq \Theta(n^2)$; c) $\frac{1}{2}n^2 - 3n = \Omega(n^2)$;

   d) dla funkcji kwadratowej $f(n) = an^2 + bn + c$, gdzie a, b i c są stałymi oraz $a > 0$, zachodzi $f(n) = \Theta(n^2)$.

2. Zapisać w języku maszyny RAM algorytm czytający liczby wejściowe, aż do napotkania zera. Obliczyć sumę tych liczb i wypisać wynik.

3. Zapisać w pseudojęzyku następujące algorytmy:
   a) Wyznaczanie największego wspólnego podzielnika dwóch liczb (algorytm Euklidesa)
   b) Poszukiwanie maksymalnej wartości w tablicy jednowymiarowej
   c) Sortowanie *przez selekcję* tablicy A złożonej z n elementów. Znajdujemy najmniejszy element w tablicy i wstawiamy go na

pierwsze miejsce. Następnie znajdujemy drugi najmniejszy element i wstawiamy go na drugie miejsce. W ten sposób postępujemy dla wszystkich n elementów tablicy A.

4. Dany jest następujący algorytm:

```
MAX
1  czytaj x
2  max←x
3  dopóki x≠0
4        wykonuj { jeśli x>max
5                  to max←x
6           czytaj x }
7  pisz max
```

   a) Napisać program w języku maszyny RAM realizujący ten algorytm
   b) Określić złożoność czasową i pamięciową programu RAM.

5. Zapisać w pseudojęzyku algorytm, którego zadaniem jest sprawdzenie, czy na wejściu pojawiła się jednakowa liczba jedynek i dwójek. Po przeczytaniu 0, algorytm wypisuje 1, jeśli liczba jedynek i dwójek była jednakowa i zatrzymuje się. Zakładamy, że oprócz 0, 1 i 2 na wejściu nie mogą pojawić się żadne inne symbole. Dla danego algorytmu wykonać także następujące zadania:

   a) Napisać program w języku maszyny RAM realizujący ten algorytm
   b) Określić złożoność czasową i pamięciową programu RAM.

6. Napisać w pseudojęzyku algorytm wyznaczania wartości $n!$ dla podanej liczby nieujemnej n, a następnie:

   a) Napisać program w języku maszyny RAM realizujący ten algorytm
   b) Określić złożoność czasową i pamięciową programu RAM.

Zadania 4,5,6 wykonać przyjmując zarówno równomierne, jak i logarytmiczne kryterium wagowe.

2 Struktury dynamiczne i rekursja

2.1 Wprowadzenie

Istnieje obszerna klasa elementarnych struktur danych, które wymagają *stałej*, określanej w chwili ich utworzenia, wielkości pamięci. Przykładami są znane z Pascala typy liczbowe, logiczne, tablice i rekordy. Tymczasem w rozwiązywaniu wielu problemów przydatne okazują się takie struktury, których chwilowa postać i rozmiar zmieniają się w trakcie przetwarzania, a pamięć dla nich jest wydzielana i zwalniana w miarę potrzeby. Tego typu obiekty tworzą klasę tzw. *struktur dynamicznych*, do których zaliczamy m.in.: listy, stosy, kolejki i drzewa. Wiele struktur dynamicznych posiada charakter *rekursywny*, stąd algorytmy operujące na takich strukturach wygodnie jest konstruować stosując *rekursję*.

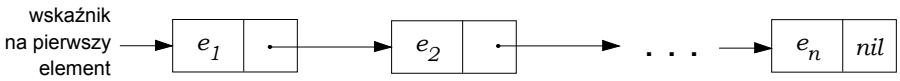
2.2 Listy

2.2.1 Definicje i reprezentacja pamięciowa

Podstawową strukturą dynamiczną jest lista. Najprościej listę można określić jako *skończony ciąg elementów należących do pewnego zbioru*. Jeżeli jest to ciąg: $e_1, e_2, ..., e_n$, to listę często zapisuje się w postaci:

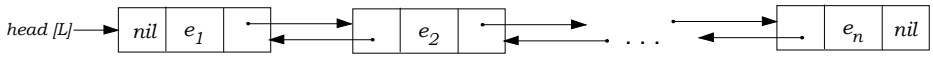
$$(e_1\ e_2\ ...\ e_n).$$

Najprostszą realizacją listy jest dynamiczna struktura pamięci pokazana na rys.2.1.



Rys.2.1 Lista jednokierunkowa.

Elementarny składnik tej struktury składa się z dwóch pól, które można traktować jako odrębne komórki pamięci. Pierwsze z pól zawiera sam pamiętany element (jego zawartość informacyjną), a drugie tzw. *wskaźnik* informujący o położeniu następnego elementu w liście. Wyjątkiem jest ostatni składnik, gdzie w polu wskaźnika umieszczono symbol *nil*, informujący, że w tym miejscu nie występuje wskaźnik na żaden element. Wskaźnik na pierwszy element jest także pamiętany w specjalnie do tego celu przeznaczonej zmiennej. Struktura przedstawiona na rys.2.1 jest przykładem tzw. *listy jednokierunkowej*, w której pojedynczy składnik zawiera wskaźnik tylko na element następny. Najczęściej listę przedstawia się w postaci nieco bardziej rozbudowanej struktury określanej jako *lista dwukierunkowa* pokazana na rys.2.2.

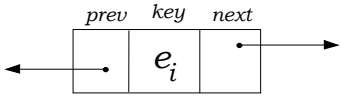


Rys.2.2 Lista dwukierunkowa.

Pojedynczy składnik listy dwukierunkowej zawiera pole informacyjne i dwa pola wskaźników: lewy wskazujący na poprzedni element oraz prawy wskazujący na element następny. Zauważmy, że lewy wskaźnik pierwszego elementu i prawy wskaźnik ostatniego elementu listy dwukierunkowej zawierają symbol *nil* (nie wskazują na żadne elementy). Taka organizacja list sprawia, że w odróżnieniu od tablic, gdzie używa się indeksów, tutaj porządek elementów jest określony za pomocą wskaźników związanych z każdym elementem listy.

Przetwarzając listy w pseudokodzie wprowadzimy specjalną zmienną typu lista repezentującą strukturę z rys. 2.2. Przyjmuje się, że wskaźnik na pierwszy element listy jest pamiętany jako wartość jej atrybutu o nazwie *head*. Zakładając zatem, że *L* jest zmienną oznaczającą listę z rys.2.2, odwołanie do wskaźnika na początek tej listy zapisujemy jako *head[L]*. Operacje na listach często wymagają użycia atrybutów także dla pojedynczego elementu listy. Jeśli przyjąć, że *x* jest elementem listy dwukierunkowej, to odwołania do jego pól są możliwe za pomocą następujących atrybutów (rys.2.3):

prev[*x*] - wskaźnik na poprzedni element,
key[*x*] - zawartość informacyjna elementu,
next[*x*] - wskaźnik na następny element.



Rys.2.3 Pola odpowiadające atrybutom elementu listy.

Wprowadzony wcześniej symbol *nil* można także traktować jako „wskaźnik pusty", stąd *head[L]=nil* oznacza, że lista L jest pusta (nie zawiera żadnych elementów). Z kolei spełnienie równości *prev[x]=nil* oznacza, że element *x* nie ma poprzednika, jeśli natomiast *next[x]=nil*, to element *x* nie ma następnika.

2.2.2 Podstawowe operacje na listach

Poszukiwanie elementu w liście

Rozważymy funkcję LISTA-POSZ z parametrami *L* i *k*, wyznaczającą pierwszy napotkany element listy *L*, którego pole *key*[*x*] (pole klucza) ma wartość *k*. W wyniku funkcja zwraca wskaźnik na ten element, a jeśli elementu nie znaleziono, jest zwracana wartość *nil*. Przykładowy zapis funkcji LISTA-POSZ przedstawiono w postaci kodu K.2.1.

K.2.1

```
LISTA-POSZ(L,k)
1  x←head[L]
2  dopóki x≠nil i key[x] ≠k
3     wykonuj x←next[x]
4  zwróć x
```

Poszukiwanie w tym algorytmie odbywa się metodą przeglądania kolejnych elementów listy *L*. Na początku zmiennej *x* jest nadawana wartość wskaźnika na początek listy. Następnie w pętli **dopóki** *x* przyjmuje wartości wskaźników na kolejne elementy listy, aż do znalezienia poszukiwanego elementu lub wyczerpania elementów listy. Jeśli liczbę elementów listy oznaczyć przez *n*, to pesymistyczny czas działania tego algorytmu wynosi *O(n)*, ponieważ w najgorszym przypadku okaże się konieczne sprawdzenie wszystkich elementów listy.

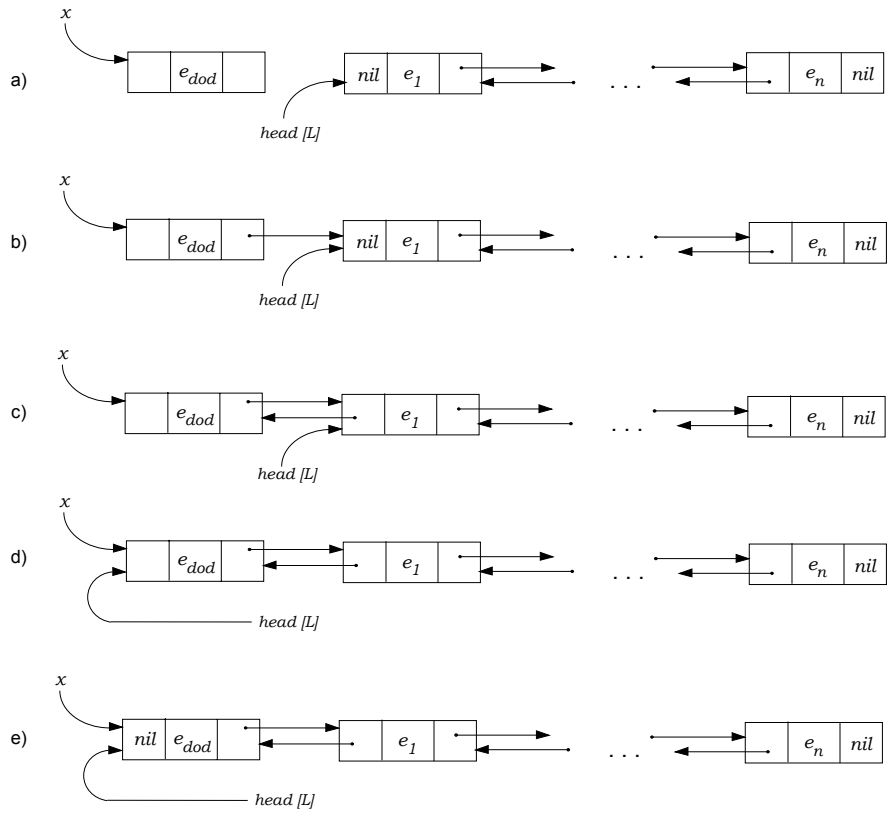
Wstawianie elementu do listy

Kod K.2.2 zawiera procedurę LISTA-WSTAW z parametrami L i x, która dołącza nowy element x na początek listy L.

K.2.2

```
LISTA-WSTAW(L,x)
1  next[x]← head[L]
2  jeśli head[L]≠nil
3      to prev[head[L]]← x
4  head[L]← x
5  prev[x]← nil
```

Wstawianie nowego elementu x na początek listy niepustej listy L zilustrowano na rys.2.4.



Rys.2.4 Wstawianie nowego elementu do listy.

Przed wykonaniem procedury wstawiania *head*[*L*] jest wskaźnikiem na początek listy, a *x* wskazuje na nowy element (rys.2.4a). Pierwszą operacją wykonywaną w linii 1 kodu K.2.2 jest nadanie wskaźnikowi *next*[*x*] wartości *head*[*L*], co prowadzi do sytuacji pokazanej na rys.2.4b. Kolejne operacje odpowiadające liniom 3, 4 i 5 algorytmu przedstawiono odpowiednio na rys.2.4c, 2.4d i 2.4e.

Czas działania procedury LISTA-WSTAW nie zależy od liczby elementów w liście *L*, a zatem może być zapisany jako *O(1)*.

Usuwanie elementu z listy

Wprowadzimy procedurę LISTA-USUŃ(*L*,*x*) (kod K.2.3), która usuwa z listy *L* element *x*.

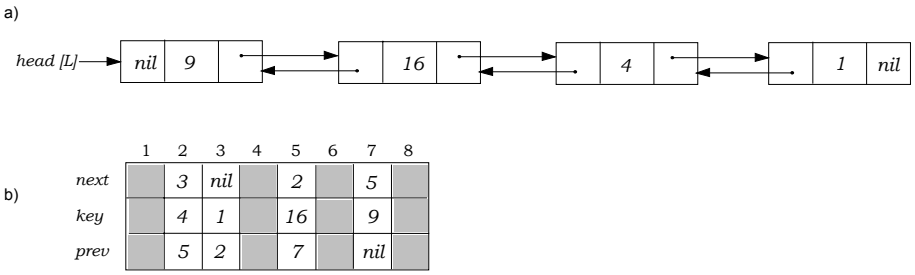
K.2.3

```
LISTA-USUŃ(L,x)
1  jeśli prev[x]≠nil
2     to next[prev[x]]← next[x]
3     inaczej head[L]← next[x]
4  jeśli next[x]≠nil
5     to prev[next[x]]← prev[x]
```

Wprawdzie procedura LISTA-USUŃ działa w czasie *O*(*1*), jednak pesymistyczny koszt usunięcia elementu *x*, dla którego zamiast wskaźnika znana byłaby tylko zawartość informacyjna (*key*[*x*]) wynosi *O*(*n*). Wynika to z konieczności uprzedniego znalezienia elementu *x*, co wymagałoby użycia funkcji LISTA-POSZ (kod K.2.1).

Przedstawienie list za pomocą tablic

Reprezentacja list w pamięci jest możliwa także bez użycia wskaźników. Jeden ze sposobów polegający na użyciu tablic jednowymiarowych pokazano na rys.2.5.



Rys.2.5 Reprezentowanie listy przy pomocy tablic jednowymiarowych.

Rysunek 2.5a przedstawia listę (9 16 4 1) jako strukturę zbudowaną przy pomocy wskaźników. Ta sama lista może być przedstawiona za pomocą trzech tablic: *next*, *key* i *prev* w sposób pokazany na rys.2.5b. Tablica *key* zawiera wartości kluczy (część informacyjną) elementów listy, a wskaźniki są reprezentowane przez tablice *next* i *prev*. Zatem dla pewnego indeksu *i* wartości *prev*[*i*], *key*[*i*] oraz *next*[*i*] leżące w kolumnie *i* określają pojedynczy element listy. Na przykład element listy o kluczu 16 posiada poprzednika, którym jest element przechowywany w tablicy *key* z indeksem 7. Jest to element o wartości klucza 9.

W podanym przykładzie do reprezentacji list użyto trzech tablic. Innym możliwym sposobem jest użycie do tego celu tylko jednej tablicy.

2.3 Stosy i kolejki

Stosy i kolejki reprezentują grupę obiektów, określanych jako struktury z ograniczonym dostępem. Ograniczenie polega na tym, że tryb dodawania i usuwania elementów jest z góry ustalony i charakterystyczny dla danej struktury.

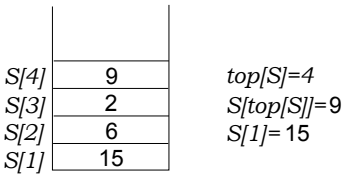
2.3.1 Stosy

Struktura nazywana *stosem* posiada ustalony element nazywany wierzchołkiem, który jest jedynym dostępnym w danej chwili jej składnikiem. Sytuacja taka oznacza, że dowolna operacja usuwania ze stosu może dotyczyć tylko jego aktualnego wierzchołka. Ponadto dodanie nowego elementu do stosu jest możliwe tylko przez umieszczenie tego elementu na „szczycie” stosu, tak aby

stał się on nowym wierzchołkiem. Mówimy, że elementy stosu są obsługiwane według porządku *LIFO* (*Last In First Out*), co oznacza „ostatni przyszedł, pierwszy wyszedł".

Sugestywną ilustracją stosu może być pewna liczba talerzy poukładanych jeden na drugim. Przy dodawaniu kolejnego talerza najbardziej rozsądnym postępowaniem jest umieszczenie go na samej górze. W przypadku pobierania talerza, zdejmujemy ten, który znajduje się na szczycie, tj. został położony jako ostatni.

Jako struktury reprezentującej stos w pamięci można użyć tablicy jednowymiarowej $S[1..n]$. Dla tej tablicy określimy atrybut *top*[*S*] podający numer (indeks) elementu ostatnio umieszczonego na stosie. Stos składa się zatem z elementów $S[1], S[2], ..., S[top[S]]$, gdzie $S[1]$ oznacza element na dnie stosu, a $S[top[S]]$ jest elementem na wierzchołku stosu (rys.2.6). Przyjmuje się, że jeśli $top[S] = 0$, to stos jest pusty i próba pobrania elementu z takiego stosu spowoduje komunikat o błędzie „niedomiaru".



Rys.2.6 Stos.

Tradycyjnie operację umieszczania na stosie nazywa się PUSH, a operację zdejmowania ze stosu – POP. Posługując się zdefiniowaną tutaj „tablicową" reprezentacją stosu operację umieszczania elementu *x* na stosie *S* można napisać w postaci prostej procedury (kod K.2.4).

K.2.4

```
PUSH(S,x)
1  top[S]← top[S]+1
2  S[top[S]]← x
```

Kod K.2.5 zawiera procedurę zdejmowania ze stosu. Procedura ta wykorzystuje funkcję STOS-PUSTY(*S*), sprawdzającą czy stos jest pusty (kod K.2.6)

K.2.5

```
POP(S)
1  jeśli STOS-PUSTY(S)
2     to błąd „niedomiar”
3     inaczej {top[S]← top[S]-1
4       { zwróć S[top[S]+1]}
```

K.2.6

```
STOS-PUSTY(S)
1  jeśli top[S]=0
2     to TRUE
3     inaczej FALSE
```

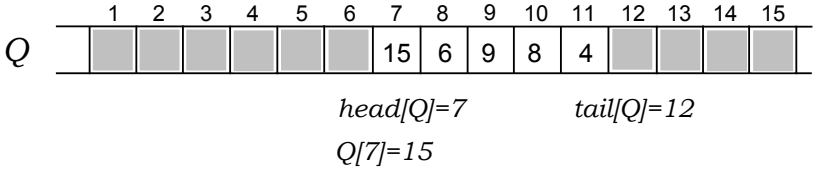
Należy zauważyć, że użycie procedury PUSH wymaga podania dwóch argumentów, podczas gdy w przypadku procedury POP wystarczy tylko jeden (identyfikator stosu). Wynika to stąd, że pobieranym elementem stosu może być w danej chwili tylko jego wierzchołek, tzn. element $S[top[S]]$.

2.3.2 Kolejki

Pojęciem kolejka określa się strukturę z ograniczonym dostępem, dla której określono początek i koniec. Prostą ilustracją może być kolejka osób do kasy, w której nowe osoby mają prawo ustawić się na końcu, a obsługiwana jest zawsze osoba z początku. Tak jak w przypadku stosu, procedura obsługi kolejki ma ogólnie przyjętą nazwę. Mówimy, że elementy kolejki są obsługiwane według porządku *FIFO* (*First In First Out*), co oznacza „pierwszy przyszedł, pierwszy wyszedł”

Podobnie jak dla stosu przyjmiemy tablicową implementację kolejki zakładając, że elementy kolejki są pamiętane w tablicy jednowymiarowej $Q[1..n]$. Posłużymy się następującymi atrybutami tablicy Q (rys.2.7):

head[*Q*] - indeks elementu znajdującego się na początku kolejki (głowa),
tail[*Q*] - indeks pierwszego wolnego elementu na końcu kolejki (ogon).



Rys.2.7 Kolejka.

W procedurach wstawiania i usuwania elementu z kolejki będziemy zakładać, że tablica $Q[1..n]$ jest cykliczna, tzn. po elemencie $Q[n]$ następuje element

$Q[1]$. Na początku przyjmuje się $head[Q] = tail[Q] = 1$. Spełnienie warunku $head[Q] = tail[Q]$ oznacza, że kolejka jest pusta. Jeśli natomiast zachodzi równość $head[Q] = tail[Q] + 1$, to kolejka jest pełna. Przyjmując tablicową reprezentacją kolejki operację umieszczania nowego elementu x w kolejce Q można napisać w postaci procedury ENQUEUE(Q,x) (kod K.2.7), a operację usuwania z kolejki jako funkcję DEQUEUE(Q) (kod K.2.8).

K.2.7

```
ENQUEUE(Q,x)
1 Q[tail[Q]]← x
2 jeśli tail[Q]= length[Q]
3     to tail[Q]← 1
4     inaczej tail[Q]← tail[Q]+1
```

K.2.8

```
DEQUEUE(Q)
1 x← Q[head[Q]]
2 jeśli head[Q]= length[Q]
3     to head[Q]← 1
4     inaczej head[Q]← head[Q]+1
5 zwróć x
```

Obie procedury działają w czasie $O(1)$. Przyjęto także, że w przypadku dodawania nowego elementu kolejka nie jest pełna, a w przypadku, usuwania, kolejka nie jest pusta.

2.4 Obliczanie wartości wyrażeń z wykorzystaniem stosu

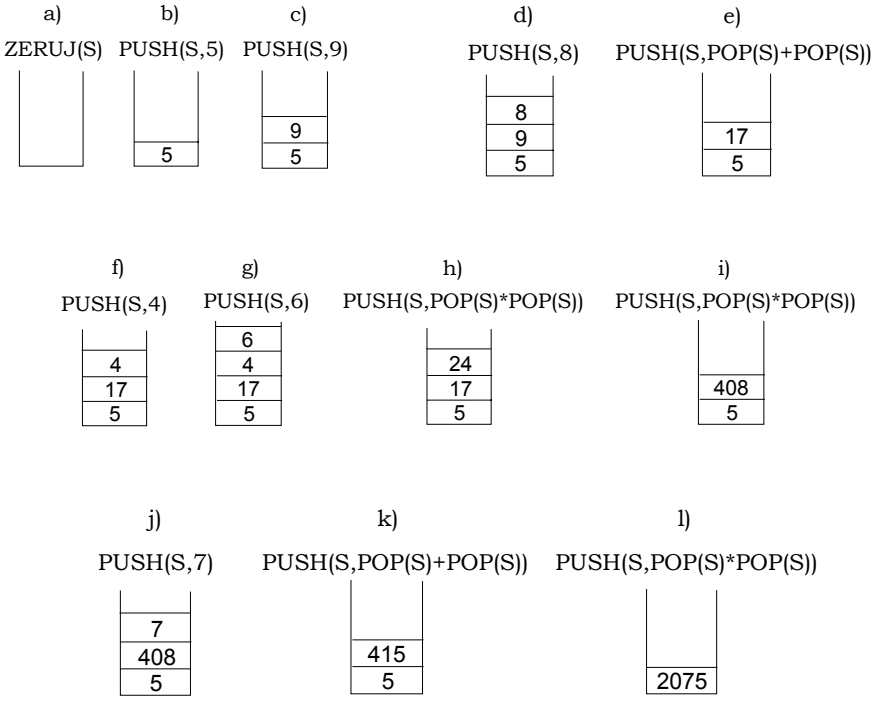
Zastosowanie stosu można pokazać na przykładzie obliczania wartości prostych wyrażeń arytmetycznych, w których występują tylko liczby całkowite 0...9, nawiasy ‘(‘ i ‘)' oraz operacje dodawania i mnożenia. Do klasy takich wyrażeń należą np.:

$(2+1)$,
$((2+1)*7)$,
$5*(((9+8)*(4*6))+7)$, itp.

Stos okazuje się być wygodnym miejscem do pamiętania wyników pośrednich w takich obliczeniach. Na przykład wartość wyrażenia

$$5*(((9+8)*(4*6))+7)$$

może być wyznaczona w wyniku ciągu operacji na stosie pokazanych na rys.2.8.



Rys.2.8 Użycie stosu do wyznaczania wartości wyrażenia.

Podczas wykonywania dowolnej operacji odpowiednie operandy są pobierane ze stosu, a wynik jest umieszczany także na stosie. Zauważmy, że taki sposób wykonywania obliczeń wymaga, aby operandy były umieszczone na stosie, zanim zostanie napotkany znak operacji. Pożądane byłoby zatem, aby wyrażenie zostało wcześniej zapisane w pewnej postaci pośredniej, w której operandy występują zawsze przed znakiem operacji. Dla naszego wyrażenia jest to zatem ciąg znaków:

5 9 8 + 4 6 * * 7 + *.

Taki zapis wyrażeń jest nazywany zapisem *postfiksowym albo odwrotnym zapisem polskim*. W odróżnieniu od zapisu postfiksowego, tradycyjny zapis matematyczny, gdzie operandy są oddzielone znakiem operacji będziemy nazywać zapisem *infiksowym*. Istotną właściwością zapisu postfiksowego jest także to, że nie wymaga on nawiasów. Pomimo braku nawiasów wyrażenia są obliczane od lewej strony do prawej we właściwej kolejności. Notacja

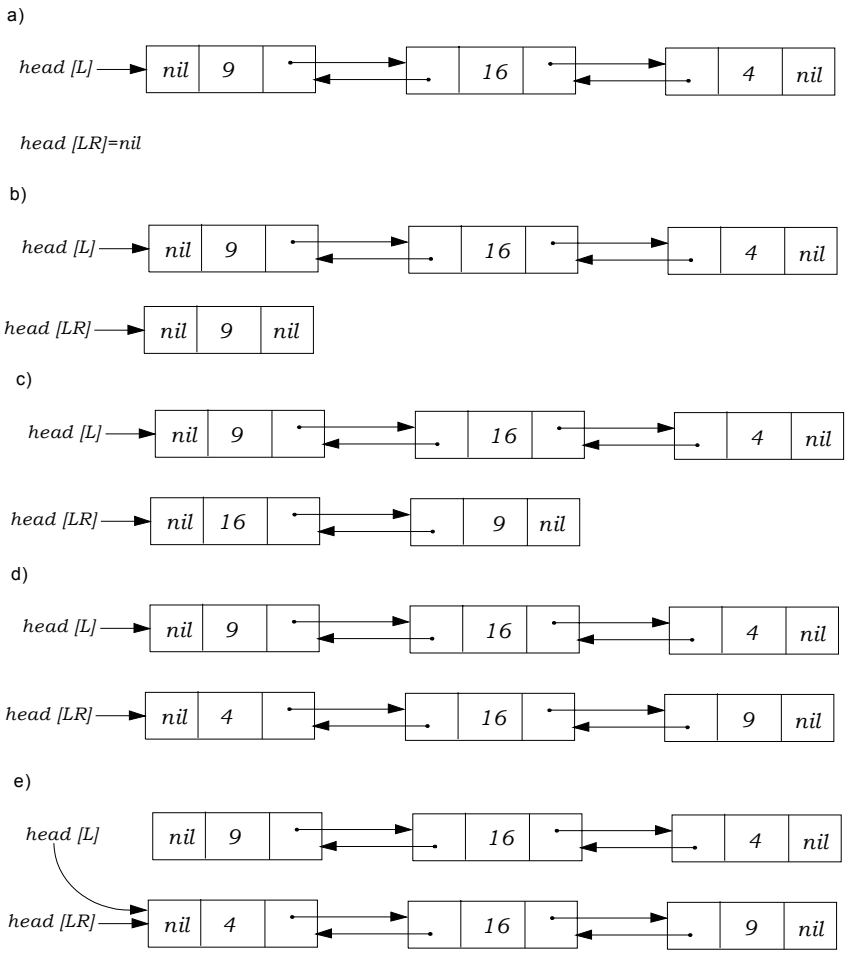
postfiksowa okazuje się być idealnym zapisem pośrednim dla wyrażeń arytmetycznych, które mają być obliczane za pomocą stosu. Mechanizm ten można zastosować w kalkulatorach i w komputerowej realizacji języków programowania.

Naszym zadaniem jest opracowanie algorytmu, który przekształca tradycyjny zapis infiksowy w odwrotny zapis polski. Wcześniej jednak przedstawimy pomocniczą procedurę LISTA-ODWR(*L*) (kod K.2.9), która realizuje prostą, ale przydatną operację odwracania kolejności elementów w liście *L*.

K.2.9

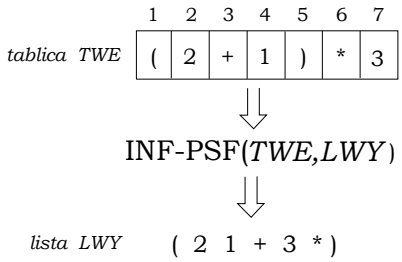
```
LISTA-ODWR(L)
1  x ← head[L]
2  head[LR] ← nil
3  dopóki x≠nil
4    wykonuj { LISTA-WSTAW(LR,x)
5              x ← next[x] }
6  head[L] ← head[LR]
```

Procedura LISTA-ODWR(*L*) korzysta ze zmiennej roboczej *LR*, która jest typu lista i służy do budowania listy odwróconej. Wykorzystuje się przy tym przedstawioną wcześniej procedurę wstawiania nowego elementu do listy. Po zbudowaniu listy odwróconej wskaźnik na jej początek staje się wartością wskaźnika na początek listy *L* (linia 6 w kodzie K.2.9). Kolejne etapy odwracania przykładowej listy (9 16 4) za pomocą procedury LISTA-ODWR(*L*) pokazano na rysunku 2.9.



Rys.2.9 Odwracanie listy.

Procedura odwracania listy zostanie wykorzystana w opisanym dalej algorytmie INF-PSF przejścia od zapisu infiksowego do zapisu postfiksowego. Przykładowe wyrażenie infiksowe jest przekształcane na wyrażenie postfiksowe w sposób pokazany na rys.2.10.



Rys.2.10 Przejście z zapisu infiksowego na postfiksowy.

Wejście algorytmu stanowi tablica *TWE* zawierająca pojedyncze znaki zadanego wyrażenia infiksowego, a wynik (odwrotny zapis polski) jest otrzymywany w postaci listy wyjściowej *LWY*. Algorytm w postaci procedury INF-PSF z parametrami *TWE* i *LWY* zawiera kod K.2.10.

K.2.10

```
INF-PSF(TWE, LWY)
1       top[STOS] ← 0
2       head[LWY] ← nil
3       i ← 1
4       powtarzaj
5         jeśli TWE[i] jest liczbą
6         to { key[x]← TWE[i]
7                LISTA-WSTAW(LWY, x) }
8         inaczej  jeśli TWE[i]="+" lub TWE[i]="*"
9                  to PUSH(STOS, TWE[i])
10                 inaczej jeśli TWE[i]=")"
11                        to { key[x]← POP(STOS)
12                                LISTA-WSTAW(LWY, x) }
13       i← i+1
14      aż_do i>length[TWE]
15      jeśli STOS-PUSTY(STOS)
16        to LISTA-ODWR(LWY)
17        inaczej { key[x]← POP(STOS)
18                  LISTA-WSTAW(LWY,x)
19            LISTA-ODWR(LWY) }
```

Algorytm analizuje kolejne elementy tablicy wejściowej *TWE*. Jeśli kolejny element tej tablicy zawiera operand (liczbę), wówczas jest ona niezwłocznie umieszczana w liście wyjściowej. Operacja ta jest realizowana w liniach 6 i 7 i

wymaga użycia zmiennej pomocniczej *x*, która staje się wskaźnikiem do dołączanego elementu listy. Znaki operacji są tymczasowo umieszczane na stosie (linia 9). Lewe nawiasy są ignorowane, a napotkanie prawego nawiasu oznacza, że oba operandy dla operacji wykonywanej na danym poziomie zagnieżdżenia obliczeń już wystąpiły i należy pobrać ze stosu odpowiadający im znak operacji oraz dołączyć go do listy wyjściowej (linie 11 i 12). Dla uproszczenia zakłada się, że na danym poziomie zagnieżdżenia występują dokładnie dwa operandy połączone znakiem operacji. Należy także zaznaczyć, że algorytm nie zajmuje się ewentualnymi błędami w wyrażeniach.

Linie od 15 do 19 zawierają polecenia generujące końcową postać wyniku. Warto zauważyć, że lista wynikowa jest odwracana za pomocą procedury LISTA-ODWR(*LWY*), dzięki czemu operandy i znaki operacji w wyrażeniu postfiksowym pojawiają się w naturalnym porządku „od lewej do prawej".

Jak już wspomniano, głównym powodem, dla którego stosujemy zapis postfiksowy jest to, że wartość tak zapisanego wyrażenia może być łatwo obliczona przy pomocy stosu. Algorytm wykonujący takie obliczenie nie jest zbyt skomplikowany, a przy jego opracowaniu pomocną może okazać się analiza przykładu obliczania wyrażenia 5 9 8 + 4 6 * * 7 + * z rys.2.8.

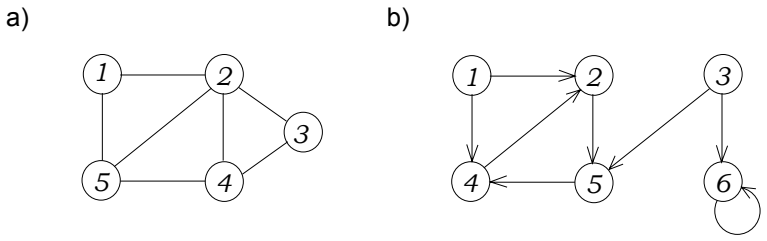
2.5 Grafy

2.5.1 Grafy skierowane i nieskierowane

Zdefiniujemy podstawowe pojęcia oraz sposób reprezentowania w pamięci dla popularnej klasy struktur nazywanych grafami.

Definicja 2.1

Grafem nazywamy parę zbiorów $G = (V, E)$, gdzie V jest skończonym zbiorem wierzchołków, a E jest zbiorem krawędzi. Przykłady grafów pokazano na rys.2.11.



Rys.2.11 Przykłady grafów: a) graf skierowany, b) graf nieskierowany.

Na rys.2.11a przedstawiono graf określany jako *nieskierowany*, dla którego

$V = \{1,2,3,4,5\}$, a

$E = \{(1,2),(1,5),(2,3),(2,4),(2,5),(3,4),(4,5)\}$.

Zbiór krawędzi E takiego grafu jest zatem zbiorem par (u,v), gdzie $u,v \in V$. Zauważmy, że pary (u,v) i (u,v) oznaczają tę samą krawędź, a ponadto $u \neq v$ (nie występują pętle, tj. krawędzie prowadzące do tego samego wierzchołka). Zbiór krawędzi E w grafie nieskierowanym jest zbiorem nieuporządkowanych par wierzchołków (u,v). Krawędź (u,v) jest określana jako *incydentna* z wierzchołkami *u* i *v*.

Na rys.2.11b pokazano graf *skierowany*, w którym

$V = \{1,2,3,4,5,6\}$, a

$E = \{(1,2),(1,4),(2,5),(3,5),(3,6),(4,2),(5,4),(6,6)\}$.

Zauważmy, że w grafie skierowanym mogą wystąpić pętle do tego samego wierzchołka, a krawędź (u,v) jest tutaj określana, jako wychodząca z wierzchołka *u* i wchodząca do wierzchołka *v*.

Jeśli (u,v) jest krawędzią grafu $G = (V,E)$, to mówimy, że wierzchołek *u* jest *sąsiedni* do wierzchołka *v*. Fakt ten dla grafu skierowanego można zapisać jako $u \to v$.

Liczbę wszystkich wierzchołków grafu oznaczymy symbolem $|V|$, a liczbę krawędzi symbolem $|E|$.

Definicja 2.2

Stopniem wierzchołka w grafie nieskierowanym nazywamy liczbę incydentnych z nim krawędzi. W przypadku wierzchołków grafu skierowanego operuje się pojęciami *stopnia wyjściowego* (liczba krawędzi wychodzących) i *stopnia wejściowego* (liczba krawędzi wchodzących).

Definicja 2.3

Ścieżką (drogą) długości k z wierzchołka u do wierzchołka u' w grafie $G = (V, E)$ nazywamy ciąg wierzchołków $\langle v_0, v_1, ..., v_k \rangle$, takich, że $u = v_0$ i $u' = v_k$ oraz $(v_{i-1}, v_i) \in E$ dla $i = 1, ..., k$.

Definicja 2.4

Cyklem w grafie skierowanym nazywamy ścieżkę $\langle v_0, v_1, ..., v_k \rangle$ zawierającą co najmniej jedną krawędź, dla której $v_0 = v_k$.

Definicja 2.5

Cyklem w grafie nieskierowanym nazywamy ścieżkę $\langle v_0, v_1, ..., v_k \rangle$, dla której $v_0 = v_k$, wierzchołki $v_1, v_2, ..., v_k$ są różne i $k \geq 2$.

Graf nie zawierający cykli nazywamy acyklicznym.

Definicja 2.6

Graf nieskierowany jest *spójny* jeśli każda para wierzchołków jest połączona ścieżką. W przypadku grafu skierowanego istnieje pojęcie *silnej spójności*, które oznacza, że każde dwa wierzchołki są osiągalne jeden z drugiego.

2.5.2 Sposoby reprezentowania grafów

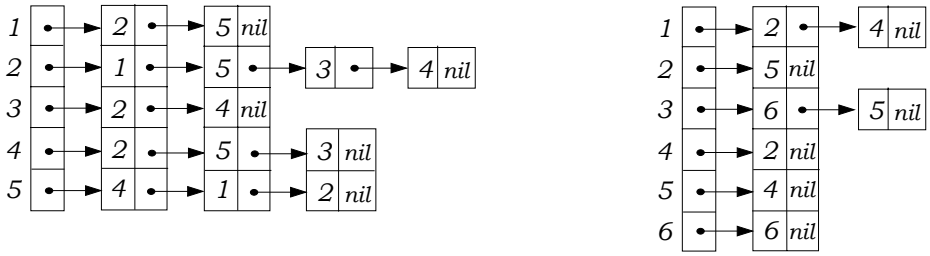
Istnieją dwa podstawowe sposoby reprezentowania grafów. Są to:

1. Listy sąsiedztwa
2. Macierze sąsiedztwa.

Przedstawienie grafów za pomocą list sąsiedztwa

Jako strukturę reprezentującą graf $G=(V,E)$ rozważamy tablicę *Adj* rozmiaru $|V|$, której elementami są listy odpowiadające wierzchołkom z *V*. Dla każdego wierzchołka $u \in V$ listę $Adj[u]$ tworzą wszystkie wierzchołki *v* takie, że $(u,v) \in E$. Listy sąsiedztwa dla grafów z rys.2.11a i 2.11b przedstawiono odpowiednio na rys.2.12a i 2.12b.

a) b)



Rys.2.12 Listy sąsiedztwa.

Zaletą reprezentacji listowej jest to, że umożliwia ona przedstawienie w zwarty sposób grafów rzadkich, dla których $|E| << |V|^2$. Przy takiej reprezentacji rozmiar wymaganej pamięci wynosi $O(\max(|V|,|E|)) = O(|V|+|E|)$.

Przedstawienie grafów za pomocą macierzy sąsiedztwa

Graf $G=(V,E)$ może być reprezentowany także przez tablicę dwuwymiarową, składającą się wyłącznie z zer i jedynek, nazywaną *macierzą sąsiedztwa*. Zakładamy wtedy, że wierzchołki grafu są ponumerowane od 1 do $|V|$. Macierz sąsiedztwa jest tablicą dwuwymiarową $A=(a_{ij})$ $(1 \le i \le |V|, 1 \le j \le |V|)$, gdzie

$$a_{ij} = \begin{cases} 1 & \text{jesli } (i,j) \in \text{E}, \\ 0 & \text{w przeciwnym razie.} \end{cases}$$

Macierze reprezentujące grafy z rys.2.11a i 2.11b przedstawiono odpowiednio na rys.2.13a i 2.13b.

a)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Rys.2.13 Macierze sąsiedztwa.

Taka reprezentacja jest korzystna wówczas, gdy graf jest „gęsty" tzn, $|E|$ jest bliskie $|V|^2$ lub gdy występuje potrzeba szybkiego stwierdzania, czy istnieje krawędź łącząca dwa zadane wierzchołki. Rozmiar wymaganej pamięci wynosi $O(|V|^2)$. Dla grafu nieskierowanego pamięć można zaoszczędzić, wykorzystując symetryczność macierzy sąsiedztwa.

2.6 Drzewa

2.6.1 Definicje

Zdefiniujemy podstawowe pojęcia odnoszące się obszernej klasy struktur danych określanych jako *drzewa*.

Definicja 2.7

Drzewem zorientowanym (skierowanym) nazywamy skierowany graf acykliczny spełniający następujące trzy warunki:

1. Istnieje dokładnie jeden wierzchołek, do którego nie dochodzi żadna krawędź. Wierzchołek ten nazywamy *korzeniem* drzewa.
2. Dla dowolnego wierzchołka *v* w drzewie istnieje droga prowadząca od korzenia do tego wierzchołka i jest to droga jedyna.
3. Każdy wierzchołek nie będący korzeniem ma dokładnie jedną krawędź wchodzącą do niego.

Dla drzewa zorientowanego z rys.2.14 zbiorem wierzchołków jest $V = \{1,2,3,4,5,6,7\}$, a zbiorem krawędzi $E = \{(1,2),(1,3),(2,4),(2,5),(2,6)(3,7)\}$. Korzeniem jest wierzchołek 1.

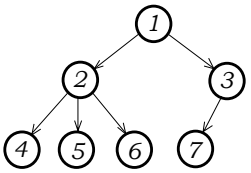
Często rozważa się drzewa *niezorientowane*, tj. takie, w których istnieje droga nie tylko od korzenia do pewnego węzła lecz droga w obu kierunkach.

Definicja 2.8

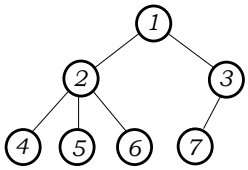
Drzewem niezorientowanym nazywamy graf nieskierowany, który jest:

1. Spójny tj. istnieje droga pomiędzy dwoma dowolnymi wierzchołkami.
2. Acykliczny.
3. Posiada wyróżniony wierzchołek nazywany korzeniem

Drzewa niezorientowane będziemy przedstawiać graficznie w sposób pokazany na rys. 2.15.



Rys.2.14 Drzewo zorientowane.



Rys.2.15 Drzewo niezorientowane.

W przypadku drzew wierzchołki często nazywa się *węzłami*.

Definicja 2.9

Jeśli rozważyć krawędź $(v,w) \in E$ w drzewie zorientowanym, to węzeł *v* jest nazywany *rodzicem* wierzchołka *w*, a węzeł *w* jest nazywany *synem* wierzchołka *v*. Pojęcie rodzica i syna ma zastosowanie także dla drzew niezorientowanych. W tym przypadku także rozważa się krawędź $(v,w) \in E$, z tym, że węzeł *v* jest węzłem położonym o jeden poziom wyżej w strukturze drzewa niż węzeł *w*.

Definicja 2.10

Jeżeli w drzewie zorientowanym istnieje droga prowadząca od węzła *v* do węzła *w*, to węzeł *w* nazywamy *potomkiem* węzła *v*, a węzeł *v* nazywamy *przodkiem* węzła *w*. Dla drzewa niezorientowanego należy zaznaczyć, że węzeł *v* jest położony wyżej w strukturze drzewa niż węzeł *w*.

Definicja 2.11

Pewien węzeł *v* wraz ze wszystkimi jego potomkami nazywamy *poddrzewem*. Węzeł *v* jest wtedy określany jako korzeń tego poddrzewa.

Definicja 2.12

Węzeł nie posiadający potomków nazywamy *liściem*.

Definicja 2.13

Głębokością węzła nazywamy długość drogi od korzenia do tego węzła.

Definicja 2.14

Wysokość węzła oznacza maksymalną długość drogi od tego węzła do liścia. Wysokością drzewa jest zatem wysokość jego korzenia.

W wielu zastosowaniach przydatne są tzw. drzewa *uporządkowane*.

Definicja 2.15

Drzewem uporządkowanym nazywamy drzewo, w którym zbiór synów każdego węzła jest uporządkowany. W graficznej reprezentacji drzew uporządkowanych będziemy zakładać, że synowie każdego węzła są uporządkowani od lewej strony do prawej.

2.6.2 Drzewa binarne

Ważną klasę drzew uporządkowanych stanowią drzewa binarne.

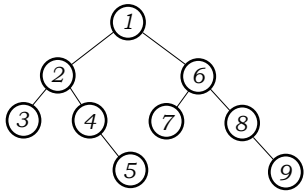
Definicja 2.16

Drzewem binarnym nazywamy drzewo spełniające dwa warunki:

i. Każdy węzeł posiada co najwyżej dwóch synów
ii. Wśród synów każdego węzła da się wyróżnić *lewy* i *prawy*.

Przykład drzewa binarnego przedstawia rys. 2.16a, a prosty sposób reprezentowania tego drzewa za pomocą tablic pokazano na rys. 2.16b.

a)



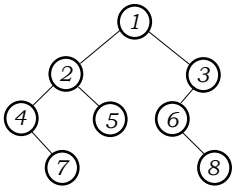
b)

	LS	*RS*
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	0

Rys.2.16 Reprezentowanie drzewa binarnego za pomocą tablic.

W proponowanym tutaj sposobie reprezentowania drzewa binarnego wykorzystuje się dwie tablice jednowymiarowe *LS* i *RS*. Dla pewnego indeksu *i* elementy tablic *LS*[*i*] i *RS*[*i*] służą do pamiętania indeksów odpowiednio lewego i prawego syna węzła *i*.

Ze względów praktycznych związanych z zastosowaniem drzew binarnych w algorytmach istotny jest sposób (kolejność) przechodzenia węzłów takiego drzewa. Drzewo pokazane na rys.2.17, które posłuży jako przykładowe podczas prezentowania różnych metod przechodzenia drzewa binarnego.



Rys.2.17 Przykład drzewa binarnego.

Zdefiniowane zostaną trzy metody przechodzenia (odwiedzania) węzłów drzewa binarnego.

Przechodzenie drzewa metodą *preorder*

Odwiedzanie kolejnych węzłów drzewa binarnego metodą *preorder* odbywa się według algorytmu:

1. Odwiedź pewien węzeł.
2. Przejdź *metodą preorder* kolejno poddrzewa: lewe i prawe, których korzeniami są synowie rozważanego węzła.

Ogólny schemat metody preorder przedstawiono na rys.2.18a, a kolejność przechodzenia węzłów drzewa binarnego z rys.2.17 zaznaczono na rys.2.18b. Przechodzenie rozpoczęto od korzenia.

Rys.2.18 Ilustracja metody preorder.

Do określenia *metody preorder* użyto samego definiowanego pojęcia (*metoda preorder*). Taki sposób definiowania nazywa się rekurencyjnym i będzie zastosowany także przy definiowaniu dwóch pozostałych metod.

Przechodzenie drzewa metodą *postorder*

Odwiedzanie kolejnych węzłów drzewa binarnego metodą *postorder* odbywa się według algorytmu:

1. Przejdź *metodą postorder* kolejne poddrzewa: lewe i prawe, których korzeniami są synowie rozważanego węzła.
2. Odwiedź rozważany węzeł.

Ogólny schemat metody postorder przedstawiono na rys.2.19a, a kolejność przechodzenia węzłów drzewa binarnego z rys.2.17 zaznaczono na rys.2.19b.

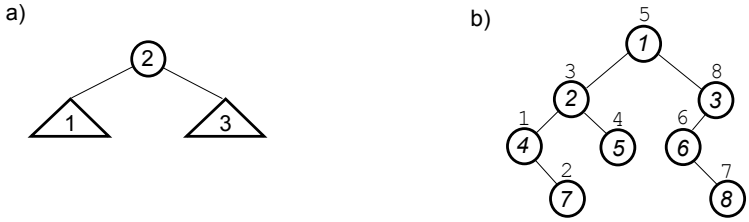
Rys.2.19 Ilustracja metody postorder.

Przechodzenie drzewa metodą *inorder*

Odwiedzanie kolejnych węzłów drzewa binarnego metodą *inorder* odbywa się według algorytmu:

1. Przejdź *metodą inorder* lewe poddrzewo potomków rozważanego węzła.
2. Odwiedź rozważany węzeł.
3. Przejdź *metodą inorder* prawe poddrzewo potomków rozważanego węzła.

Ogólny schemat metody inorder przedstawiono na rys.2.20a, a kolejność przechodzenia węzłów drzewa binarnego z rys.2.17 zaznaczono na rys.2.20b.



Rys.2.20 Ilustracja metody inorder.

Drzewa regularne i pełne

Zdefiniujemy jeszcze dwie podklasy drzew binarnych tj. drzewa *regularne* i drzewa *pełne*.

Definicja 2.17

Regularnym drzewem binarnym nazywamy drzewo, w którym każdy z węzłów jest albo liściem, albo posiada jednocześnie lewego i prawego syna.

Rys.2.21 Drzewa binarne: regularne (a) i nieregularne (b).

Zgodnie z definicją 2.17 drzewo na rys.2.21a jest regularne, podczas gdy drzewo na rys.2.21b nie jest.

Definicja 2.18

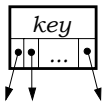
Drzewo binarne nazywamy *pełnym* gdy istnieje liczba *k* taka, że dla każdego węzła o głębokości mniejszej niż *k* występują oba węzły będące jego synami, a każdy węzeł o głębokości *k* jest liściem.

Rys.2.22 Drzewa binarne: pełne (a) i niepełne (b).

Zatem pełne drzewo binarne to takie drzewo, w którym dla wszystkich węzłów wewnętrznych (tj. nie będących liśćmi) występują jednocześnie obaj synowie, a głębokość wszystkich liści jest jednakowa. Zgodnie z definicją 2.18 drzewo na rys.2.22a jest pełne, podczas gdy drzewo na rys.2.22b nie jest.

2.6.3 Użycie struktur wskaźnikowych

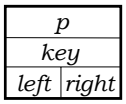
Podobnie jak listy, drzewa często są reprezentowane przy użyciu struktur ze wskaźnikami. Analogicznie do elementów listy, węzły drzewa są wtedy strukturami o kilku polach, wsród których występuje pole klucza oraz dodatkowe pola przeznaczone do pamiętania wskaźników do innych węzłów (rys.2.23). Liczba i sposób interpretacji tych wskaźników na ogół zależy od rodzaju drzewa.



Rys.2.23 Węzeł drzewa reprezentowanego za pomocą struktur ze wskaźnikami.

Drzewa binarne

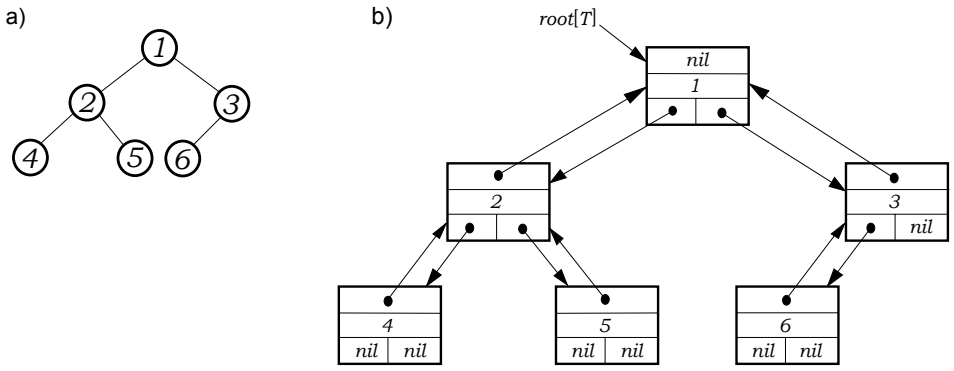
Struktura pojedynczego węzła drzewa binarnego jest pokazana na rys.2.24.



Rys.2.24 Węzeł drzewa binarnego jako struktura ze wskaźnikami.

W każdym węźle drzewa binarnego jest pamiętana część informacyjna węzła (pole *key*) oraz wskaźniki *p* - do węzła rodzica, *left* – do lewego syna i *right* – do prawego syna tego węzła. W pseudokodzie dla pewnego węzła *x* operować

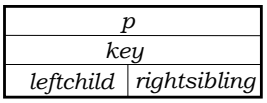
będziemy odpowiednio atrybutami *key*[*x*], *p*[*x*], *left*[*x*] oraz *right*[*x*]. Jeśli *p*[*x*]=*nil*, to węzeł *x* jest korzeniem. Jeśli natomiast *left*[*x*]=*nil* lub *right*[*x*]=*nil* to węzeł *x* nie ma odpowiednio lewego lub prawego syna. Atrybut *root*[*T*] drzewa *T* oznacza wskaźnik na korzeń drzewa. Jeśli *root*[*T*]=*nil* to drzewo jest puste. Przykład reprezentowania drzewa binarnego z rys.2.25a przy użyciu wskaźników pokazano na rys.2.25b.



Rys.2.25 Proste drzewo binarne jako struktura ze wskaźnikami.

Drzewa o dowolnej strukturze

W przypadku jeśli na strukturę drzewa nie nakłada się wstępnych ograniczeń, tzn. liczba synów każdego węzła nie jest z góry ustalona, bardziej odpowiednia dla pojedynczego węzła jest struktura pokazana na rys.2.26. Umożliwia ona reprezentację dowolnego drzewa o *n* węzłach w pamięci $O(n)$.

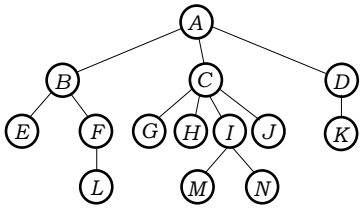


Rys.2.26 Węzeł drzewa dowolnego jako struktura ze wskaźnikami.

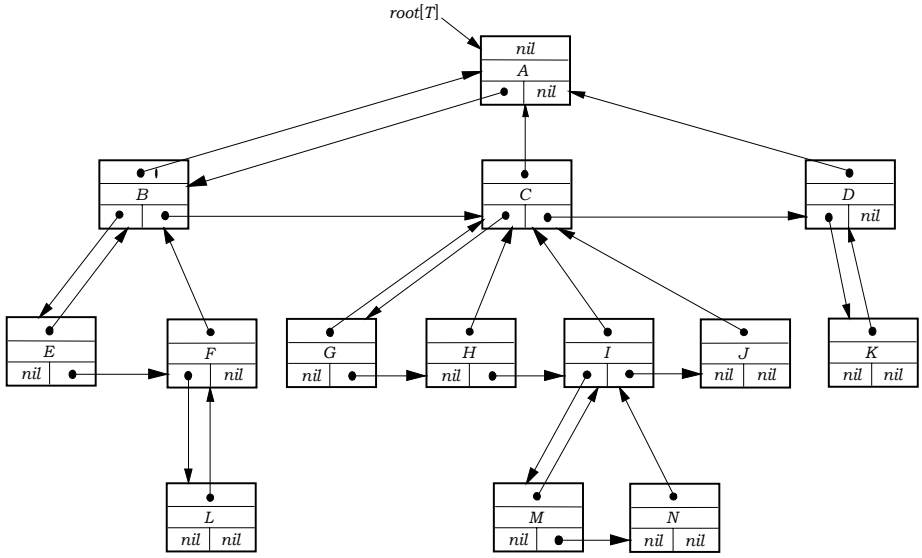
W każdym węźle drzewa występują teraz następujące pola:

p[*x*]	- wskaźnik na węzeł-rodzic,
key[*x*]	- zawartość informacyjna węzła,
leftchild[*x*]	- wskaźnik na węzeł-syn położony najbardziej z lewej,
rightsibling[*x*]	- wskaźnik na prawego „brata” tj. prawy sąsiedni węzeł będący synem tego samego węzła-rodzica.

W pseudokodzie dla pewnego węzła x operować będziemy odpowiednio atrybutami $p[x]$, $key[x]$, $leftchild[x]$ oraz $rightsibling[x]$. Jeśli węzeł x nie ma synów, to $leftchild[x]=nil$. Jeśli natomiast jest on najbardziej na prawo położonym synem pewnego węzła to $rightsibling[x]=nil$. Przykład reprezentowania drzewa z rys.2.27 przy użyciu wskaźników pokazano na rys.2.28.



Rys.2.27 Drzewo o strukturze dowolnej.



Rys.2.28 Drzewo z rys.2.27 jako struktura ze wskaźnikami.

2.7 Rekursja

Rekursję nazywaną także *rekurencją* zastosowaliśmy już, definiując metody przechodzenia węzłów drzewa binarnego.

Zastosowanie rekurencji często prowadzi do bardziej przejrzystych i zrozumiałych definicji i algorytmów. Przykładami są m.in. także rekurencyjne definicje listy i drzewa binarnego.

Definicja 2.19 (Rekurencyjna definicja listy)

Listą nazywamy strukturę zdefiniowaną na skończonym zbiorze elementów, która:

- albo nie zawiera żadnych elementów i jest nazywana listą *pustą*,
- albo jest konkatenacją (połączeniem) elementu i *listy*.

Definicja 2.20 (Rekurencyjna definicja drzewa binarnego)

Drzewo binarne jest strukturą zdefiniowaną na skończonym zbiorze węzłów w ten sposób, że:

- albo nie zawiera żadnych węzłów i jest nazywane drzewem *pustym*,
- albo składa się z trzech rozłącznych zbiorów węzłów: korzenia*, drzewa binarnego* nazywanego lewym poddrzewem oraz *drzewa binarnego* nazywanego prawym poddrzewem.

Rekurencję stosuje się powszechnie także jako metodę organizacji obliczeń.

Definicja 2.21

Procedurę, która bezpośrednio lub pośrednio wywołuje samą siebie nazywamy *rekurencyjną*.

Przykłady algorytmów rekurencyjnych

Przykład 2.1

Typowym zadaniem, gdzie można zastosować rekurencję jest obliczanie potęgi dodatniej liczby *m* według zależności:

$$m^n = \begin{cases} m \times m^{n-1} & \text{dla } n \geq 1 \\ 1 & \text{dla } n = 0 \end{cases}. \qquad (2.1)$$

Zakładamy, że wykładnik potęgi *n* przyjmuje wartości całkowite nieujemne. Jak wynika z zależności (2.1) w celu obliczenia *n*-tej potęgi liczby *m* należy obliczyć (*n*-1)-szą potęgę *m* i pomnożyć otrzymaną wartość przez *m*.

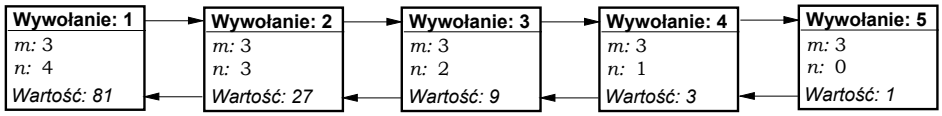
Przypadkiem szczególnym jest n=0, kiedy wartość potęgi wynosi 1. Algorytm taki realizuje procedura rekurencyjna K.2.12.

K.2.12

```
POTĘGA-M-DO-N(m,n)
1    jeśli n=0
2       to zwróć 1
3       inaczej zwróć (m * POTĘGA-M-DO-N(m,n-1))
```

W kolejnym wywołaniu procedura POTĘGA-M-DO-N sprawdza wartość wykładnika potęgi n. Jeśli wartość ta wynosi 0, wówczas jako wynikiem tego wywołania procedury jest wartość 1. W przeciwnym razie procedura wywołuje samą siebie w celu obliczenia m^{n-1}, obliczona wartość jest mnożona przez m, po czym obliczony iloczyn jest zwracany jako efekt bieżącego wywołania procedury.

Działanie procedury rekurencyjnej łatwiej jest śledzić przy pomocy prostej symulacji graficznej, przedstawiającej kolejne wywołania oraz zwracane wartości [8]. Na rys.2.29 pokazano w ten sposób porządek wywołań rekrencyjnych oraz zwracane wartości podczas obliczania wartości 3^4.



Rys.2.29 Ilustracja wywołań w procedurze rekurencyjnej obliczania potegi.

Pierwsze wywołanie procedury ma postać, POTĘGA-M-DO-N(3,4), zatem najbardziej położony na lewo prostokąt pokazuje wartości parametrów m i n jako równe odpowiednio 3 i 4. W kolejnych wywołaniach wartość n zmniejsza się każdorazowo o 1, aż do osiągnięcia wartości 0, co kończy dalsze wywołania. Wówczas są obliczane wartości kolejnych potęg i zwracane na coraz wyższe poziomy wywołań, przy czym na najwyższy poziom jest zwracany wynik końcowy wynoszący 81.

Przykład 2.2

Jako drugi przykład rozważymy przedstawiony w [1] algorytm przechodzenia metodą inorder drzewa binarnego reprezentowanego za pomocą tablic *LS* i *RS* (zob.rys.2.16). Procedura przechodzenia jest przedstawiona jako kod K.2.13.

K.2.13

```
DRZB-INORDER(węzeł)
1   jeśli LS[węzeł]≠0
2     to DRZB-INORDER(LS[węzeł])
3   NUMER[węzeł]← licznik
4   licznik ← licznik+1
5   jeśli RS[węzeł]≠0
6     to DRZB-INORDER(RS[węzeł])
```

W algorytmie przyjęto, że istnieje pewna zmienna globalna *licznik*, na początku równa 1. Przy pierwszym wywołaniu procedury parametr wejściowy *węzeł* powinien mieć także wartość 1. Wynikiem końcowym jest wypełnienie tablicy *NUMER* określającej kolejność przechodzenia węzłów drzewa w ten sposób, że element *NUMER*[*i*] zawiera liczbę naturalną pokazującą jako który z kolei jest odwiedzany węzeł *i*.

Pytania kontrolne

1. Omówić listę dwukierunkową i jej reprezentację przy użyciu wskaźników.
2. Przedstawić podstawowe operacje na listach:
   a) przeszukiwanie listy;
   b) włączanie nowego elementu;
   c) usuwanie z listy;
   d) odwracanie kolejności elementów.
3. Omówić stos i jego reprezentację tablicową. Przedstawić podstawowe operacje na stosie.
4. Omówić kolejkę i jej reprezentację tablicową. Przedstawić podstawowe operacje dla kolejki.
5. Przedstawić algorytm przejścia od zapisu infiksowego na zapis postfiksowy dla prostych wyrażeń.
6. Omówić sposoby przedstawienia listy za pomocą:
   a) kilku tablic;

b) jednej tablicy.

7. Podać definicję i sposoby reprezentacji grafów.
8. Podać definicję i sposoby reprezentacji drzew.
9. Podać rekurencyjną definicję listy i drzewa.

Zadania

1. Zaproponować sposób przedstawienia stosu za pomocą listy. Napisać procedury POP i PUSH dla reprezentacji listowej.
2. Zaproponować sposób przedstawienia kolejki za pomocą listy. Napisać procedury ENQUEUE i DEQUEUE dla reprezentacji listowej.
3. Napisać w pseudokodzie algorytm wyznaczania wartości prostych wyrażeń zapisanych w notacji postfiksowej. Zakłada się, że wyrażenia zawierają tylko pojedyncze cyfry oraz znaki operacji + i *. Należy wykorzystać stos.
4. Napisać algorytm zliczania elementów w podanej liście:
   a) bez wykorzystania rekursji;
   b) z wykorzystaniem rekursji.
5. Rozważyć listy zagnieżdżone, tj. takie, których elementy mogą same być listami. Stosując rekursję napisać algorytm wyznaczający ogólną liczbę elementów w listach o dowolnym stopniu zagnieżdżenia.
6. *Wieże Hanoi*. Stosując rekursję należy zaprogramować przeniesienie *n* dysków z pręta A na pręt B, wykorzystując przy tym pręt C jako pomocniczy. Funkcja powinna zwracać ilość ruchów potrzebnych do wykonania takiej operacji. Wejściem jest liczba dysków *n*.

Powinny przy tym być spełnione następujące warunki:
- tylko jeden dysk może być przenoszony równocześnie,
- wszystkie dyski mają różne średnice i nigdy dysk większy nie może być położony na mniejszym,
- wyjściowym położeniem jest sytuacja na rysunku.

3 Zastosowanie drzew i algorytmy przeszukiwania

3.1 Drzewa przeszukiwań binarnych

Drzewa przeszukiwań binarnych określane także jako drzewa BST *(binary search trees*) stanowią specjalną klasę drzew binarnych, dla których można stosunkowo łatwo skonstruować podstawowe algorytmy jak: wyszukiwanie, minimum, maksimum, dołączanie itd. Podstawowe operacje na drzewach BST wymagają czasu proporcjonalnego do wysokości drzewa.

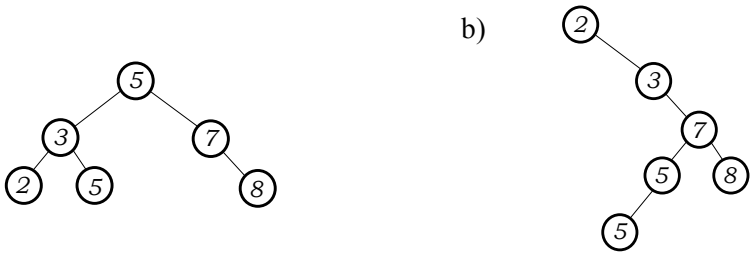
Definicja 3.1

Drzewem przeszukiwań binarnych (BST) nazywamy drzewo binarne, w którym wartości kluczy są przechowywane w taki sposób, aby spełniały tzw. *własność drzewa BST*, zgodnie z którą dla dowolnego węzła *x* zachodzi:

- jeśli *y* jest węzłem lewego poddrzewa węzła *x*, to $key[y] \le key[x]$ oraz
- jeśli *y* jest węzłem prawego poddrzewa węzła *x*, to $key[y] \ge key[x]$.

Przykłady drzew BST przedstawia rys. 3.1.

a) b)



Rys.3.1 Przykłady drzew BST.

W drzewach BST z rys.3.1a i 3.1b występują dokładnie te same wartości kluczy, lecz ze względu na większą wysokość drzewo z rys.3.1b jest mniej efektywne podczas wykonywania typowych operacji.

Podana własność drzewa BST pozwala w prosty sposób posortować rosnąco (wypisać w porządku rosnącym) klucze wszystkich węzłów drzewa. W tym celu wystarczy wykorzystać metodę przechodzenia *inorder*, jak uczyniono to w algorytmie przedstawionym jako kod K.3.1.

K.3.1

```
DRZB-INORDER1(x)
1     jeśli x≠nil
2        to { DRZB-INORDER1(left[x])
3             wypisz key[x]
4               DRZB-INORDER1(right[x]) }
```

W algorytmie założono, że drzewo BST jest reprezentowane za pomocą struktury ze wskaźnikami. Parametrem procedury przy pierwszym jej wywołaniu jest wskaźnik na węzeł-korzeń. Procedura sprawdza, czy rozważany węzeł istnieje (linia 1), po czym realizuje kolejno następujące czynności: wywołuje przechodzenie lewego poddrzewa, wypisuje wartość klucza w węźle oraz wywołuje przechodzenie prawego poddrzewa. W ten sposób dla drzewa z rys.3.1 zostanie wypisany ciąg kluczy: 2 3 5 5 7 8.

3.1.1 Przeszukiwanie w drzewach BST

Przeszukiwanie w drzewie BST jest realizowane przy pomocy rekursywnej funkcji BST-POSZ(x,k) (kod K.3.2).

K.3.2

```
BST-POSZ(x,k)
1     jeśli x=nil lub k= key[x]
2        to zwróć x
3      jeśli k<key[x]
4        to zwróć BST-POSZ(left[x],k)
5        inaczej zwróć BST-POSZ(right[x],k)
```

Parametr x jest wskaźnikiem do pewnego węzła (w szczególności może być wskaźnikiem do korzenia drzewa), a k wartością poszukiwanego klucza. Przeszukiwanie odbywa się w poddrzewie, którego korzeniem jest x. Funkcja zwraca wskaźnik do znalezionego węzła lub *nil* jeśli węzeł o podanym kluczu nie istnieje. Przeszukiwanie rozpoczyna się od węzła początkowego i jest kontynuowane w dół drzewa. Dla każdego węzła x, który nie jest pusty,

algorytm porównuje wartość klucza dla tego węzła z wartością poszukiwaną *k*. Jeśli wartości te są równe, to przeszukiwanie zostaje przerwane (węzeł został znaleziony). Jeśli *x* ma wartość *nil* to przeszukiwanie także zostaje przerwane (węzeł nie został znaleziony). Jeśli natomiast sprawdzany węzeł ma wartość klucza inną od poszukiwanej, wówczas korzystamy z własności drzewa BST w następujący sposób:

- poszukujemy dalej tylko w lewym poddrzewie, jeśli wartość poszukiwanego klucza była mniejsza od napotkanego,
- poszukujemy dalej tylko w prawym poddrzewie, jeśli wartość poszukiwanego klucza była niemniejsza od napotkanego.

Algorytm przeszukiwania w drzewie BST można także napisać nie stosując rekursji. Wersję bez rekursji określaną także jako *wersja iteracyjna* przedstawia kod K.3.3.

K.3.3

```
BST-POSZ-ITER(x,k)
1     dopóki x≠nil  i  k≠key[x]
2         wykonuj jeśli k<key[x]
3                   to x←left[x]
4                   inaczej x←right[x]
5     zwróć x
```

3.1.2 Inne operacje na drzewach BST

Poszukiwanie węzła o minimalnym kluczu

Algorytm poszukiwania węzła o minimalnym kluczu w drzewie BST realizuje funkcja BST-MIN(*x*) (kod K.3.4).

K.3.4

```
BST-MIN(x)
1     dopóki left[x]≠nil
2         wykonuj x←left[x]
3     zwróć x
```

Przeszukiwanie rozpoczyna się od dowolnego węzła *x* i odbywa się w poddrzewie, którego korzeniem jest węzeł *x*. Węzeł o minimalnym kluczu jest poszukiwany według prostej strategii „podążania" za wskaźnikiem *left*

kolejnych węzłów, aż do napotkania węzła, dla którego wskaźnik *left*[*x*] jest pusty. Węzeł o minimalnym kluczu jest zatem lewym skrajnym węzłem w przeszukiwanym poddrzewie.

W analogiczny sposób można napisać funkcję wyszukiwania węzła o maksymalnym kluczu. Łatwo zauważyć, że obie funkcje działają w czasie O(*h*), gdzie *h* jest wysokością drzewa BST.

Następniki i poprzedniki

Problem poszukiwania następnika pewnego węzła *x* w drzewie BST polega na udzieleniu odpowiedzi na pytanie: który węzeł będzie odwiedzany jako następny po *x* podczas przechodzenia drzewa w porządku *inorder*. Jeśli wszystkie klucze są różne, to następnikiem węzła *x* jest węzeł o najmniejszym kluczu, większym niż *key*[*x*]. Drzewo BST zapewnia wyznaczenie następnika bez wykonywania operacji porównania. Skonstruujemy funkcję BST-NAST(*x*) (kod K.3.5), gdzie parametr *x* jest wskaźnikiem na rozważany węzeł, a w wyniku jest zwracany wskaźnik na następnik węzła *x*. Jeśli *x* nie posiada następnika, wówczas jest zwracany wskaźnik pusty *nil*.

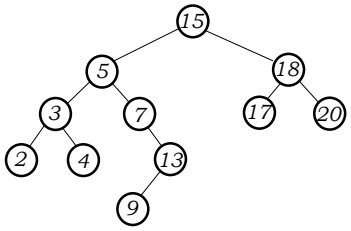
K.3.5

```
BST-NAST(x)
1     jeśli right[x]≠nil
2        to zwróć DRZB-MIN(right[x])
3     y←p[x]
4     dopóki x≠nil  i x=right[y]
5        wykonuj { x←y
6                  y←p[y] }
7     zwróć y
```

W algorytmie K.3.5 rozważono dwa przypadki. Jeśli węzeł *x* posiada prawe poddrzewo, to następnik wyznacza się łatwo, gdyż jest to najmniejszy węzeł prawego poddrzewa. W przypadku, gdy węzeł *x* nie ma prawego poddrzewa (ale ma następnik *y*), to *y* jest najniższym przodkiem węzła *x*, takim, że jego lewy syn jest także przodkiem *x*.

W drzewie BST na rys.3.2 następnikiem węzła 13 jest węzeł 15. Aby go wyznaczyć, wystarczy przejść w górę drzewa do napotkania węzła, który jest lewym węzłem-synem swojego rodzica. Przeszukiwanie takiego węzła odbywa

się w wierszach 3-6 algorytmu K.3.5. Znaleziona wartość *y* jest zwracana w wierszu 7.



Rys.3.2 Drzewo BST, w którym następnikiem węzła 13 jest węzeł 15.

Dla drzewa o wysokości *h* algorytm wyznaczania następnika, podobnie jak analogiczny algorytm wyznaczania poprzednika, działają w czasie O(*h*).

Operacje wstawiania i usuwania

Operacje wstawiania i usuwania w drzewie BST tym różnią się od dotychczas omówionych operacji, że modyfikują (zmieniają) one dynamiczną strukturę drzewa. W ogólnym przypadku dynamiczna struktura drzewa musi być wtedy przeorganizowana tak, aby zachować własność drzewa BST. Omówimy procedurę wstawiania nowego węzła *z* do drzewa przeszukiwań binarnych *T* (kod K.3.6).

K.3.6

```
BST-WSTAW(T,z)
1       y←nil
2       x←root[T]
3       dopóki x≠nil
4          wykonuj { y←x
5             jeśli key[z]<key[x]
6                to x←left[x]
7                inaczej x←right[x] }
8       p[z]←y
9       jeśli y=nil
10         to root[T]←z
11         inaczej jeśli key[z]<key[y]
12                   to left[y]←z
13                   inaczej right[y]←z
```

Zakłada się, że wstawiany węzeł przygotowano w ten sposób że *key*[*z*]=*v*, *left*[*z*]=*nil* oraz *right*[*z*]=*nil*. W wyniku wykonania procedury wstawiania, drzewo wejściowe *T* oraz pole *p*[*z*] są modyfikowane tak, że *z* staje się węzłem drzewa *T*.

Podobnie jak w przypadku procedury wyszukiwania, przeglądanie, drzewa rozpoczyna się w korzeniu i przebiega w dół drzewa. Wskaźnik *x* postępuje po pewnej ścieżce w drzewie, a zmienna *y* zawiera wskazanie na rodzica węzła *x*. W wierszach od 3 do 7 wskaźniki *x* i *y* są przesuwane w dół drzewa w kierunku prawym lub lewym, w zależności od wyniku porównania kluczy elementów wstawianego i napotkanego. Proces ten trwa aż do chwili, gdy *x* przyjmie wartość *nil* i w tym właśnie miejscu jest wstawiany nowy węzeł. Wstawianie odbywa się w wierszach od 8 do 13.

3.2 Kopce

3.2.1 Definicja i własność kopca

Definicja 3.2

Kopiec (heap) to struktura, która może być rozpatrywana jako rodzaj „prawie pełnego" drzewa binarnego (rys.3.3a). Drzewo to jest pełne w tym sensie, że jest wypełnione na wszystkich poziomach z wyjątkiem być może najniższego, który jest wypełniony od lewej do prawej do pewnego miejsca. Jednocześnie rozważa się tablicę A, której elementy odpowiadają węzłom drzewa w sposób wynikający z porównania rysunków 3.3a i 3.3b.

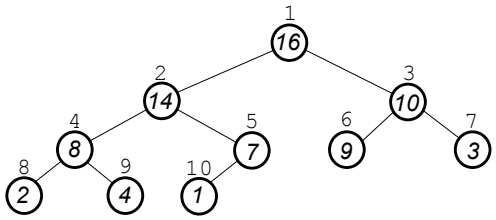
Tablica A reprezentująca kopiec posiada dwa następujące atrybuty:

length[*A*] - liczba elementów w *A*,
heap-size[*A*] - liczba elementów kopca,

dla których zachodzi nierówność:

$$heap\text{-}size[A] \leq length[A].$$

a)



b)

	1	2	3	4	5	6	7	8	9	10	11
A	16	14	10	8	7	9	3	2	4	1	

Rys.3.3 Reprezentowanie kopca przy pomocy drzewa binarnego i tablicy.

Zakłada się, że korzeniem drzewa jest $A[1]$, a wszystkie elementy z indeksem większym od *heap-size*[A] nie zawierają elementów kopca. Przy takim sposobie reprezentowania kopca, można na podstawie danego indeksu i pewnego węzła obliczyć indeks jego rodzica, a także indeksy lewego i prawego syna. Realizują to proste funkcje PARENT(i) (kod.3.7), LEFT(i) (kod.3.8) oraz RIGHT(i) (kod.3.9).

K.3.7

```
PARENT(i)
1    zwróć ⌊i/2⌋
```

K.3.8

```
LEFT(i)
1    zwróć 2i
```

K.3.9

```
RIGHT(i)
1    zwróć 2i+1
```

Własność kopca

Kopiec posiada *specjalną własność*, która mówi, że dla każdego węzła i, który nie jest korzeniem zachodzi

$$A[\text{PARENT}(i)] \geq A[i].$$

Z własności kopca wynika, że największy element kopca znajduje się w korzeniu, a poddrzewa każdego węzła zawierają wartości mniejsze niż wartość umieszczona w tym węźle. Ponieważ kopiec jest budowany jako pełne drzewo binarne, stąd jego wysokość dla n węzłów wynosi $\Theta(\lg n)$. Jest to także istotna cecha kopca, z której wynika, że podstawowe algorytmy na kopcach działają w czasie $O(\lg n)$.

3.2.2 Algorytmy na kopcach

Przywracanie własności kopca

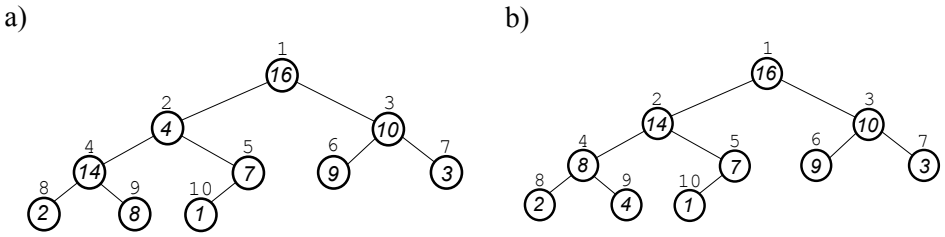
Działania na kopcu mogą prowadzić do zaburzenia jego własności, stąd podstawową operacją jest procedura przywracania własności kopca. W algorytmie K.3.10 założono, że kopiec jest reprezentowany przy pomocy tablicy A, dla której w węźle $A[i]$ należy przywrócić własność kopca.

K.3.10

```
KOPIEC-PRZYWR(A,i)
1    l← LEFT(i)
2    r← RIGHT(i)
3    jeśli l≤heap-size[A] i A[l]>A[i]
4       to największy←l
5       inaczej największy←i
6    jeśli r≤heap-size[A] i A[r]>A[największy]
7       to największy←r
8    jeśli największy≠i
9       to { zamień A[i]↔A[największy]
10           KOPIEC-PRZYWR(A,największy)}
```

W każdym kroku procedury jest wybierany największy spośród elementów $A[i]$, A[LEFT(i)] i A[RIGHT(i)], a jego indeks zostaje zapamiętany w zmiennej *największy*. Jeśli największym okazał się $A[i]$, to własność kopca w węźle $A[i]$ jest spełniona i procedura kończy pracę. W przeciwnym razie, tj. gdy największym okazał się jeden z synów węzła $A[i]$, następuje zamiana miejscami węzłów $A[i]$ oraz A[*największy*]. W wyniku węzeł $A[i]$ oraz jego synowie spełniają własność kopca. Węzeł A[*największy*] ma teraz poprzednią wartość węzła $A[i]$, dlatego poddrzewo zaczepione w węźle A[*największy*] może nie spełniać już własności kopca. W takim przypadku procedurę przywracania należy wywołać rekurencyjnie dla poddrzewa z korzeniem A[*największy*].

Przykładem zastosowania procedury K.3.10 jest przywrócenie własności kopca strukturze z rys.3.4a. W wyniku otrzymujemy kopiec pokazany na rys.3.4b.



Rys. 3.4 Efekt działania procedury przywracania własności kopca.

Jak wynika z rys.3.4a własność kopca nie jest spełniona dla węzła o wartości klucza 4, tj. węzła $A[2]$, zatem pierwsze wywołanie procedury przywracania własności kopca ma postać KOPIEC-PRZYWR(A,2). Prowadzi ono do zamiany miejscami węzłów $A[2]$ i $A[4]$, z kluczami odpowiednio 4 i 14. Taka zamiana powoduje jednak zaburzenie własności kopca w węźle $A[4]$, który ma teraz wartość klucza 4, podczas gdy jego synami są węzły $A[8]$ i $A[9]$, z kluczami odpowiednio 2 i 8. Niezbędne jest zatem rekursywne wywołanie KOPIEC-PRZYWR(A,4), które prowadzi do zamiany miejscami węzłów $A[4]$ i $A[9]$, z kluczami odpowiednio 4 i 8 oraz przywrócenia własności kopca w całym poddrzewie, którego korzeniem jest węzeł $A[2]$ (rys.3.4b).

Budowanie kopca

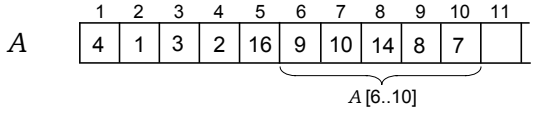
Za pomocą procedury przywracania własności kopca można z elementów zadanej tablicy $A[1..n]$ zbudować kopiec taki, że *heap-size*$[A]=n$. Należy przy tym zauważyć, że każdy element podtablicy $A[\lfloor n/2 \rfloor+1..n]$ jest liściem w drzewie binarnym reprezentującym kopiec. Elementy te zatem można traktować jako jednoelementowe kopce, w których własność kopca na pewno jest spełniona. Na przykład dla tablicy wyjściowej $A[1..10]$, pokazanej na rys. 3.5a podtablicą liści jest $A[6..10]$ (por.rys.3.5b).

Procedurę budowania kopca przedstawiono jako kod K.3.11.

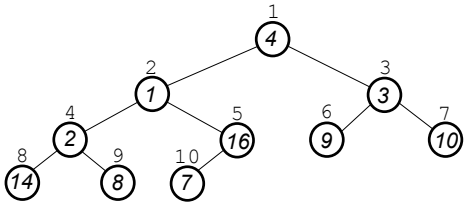
K.3.11

```
KOPIEC-BUDUJ(A)
1     heap-size[A] ← length[A]
2     dla i ← ⌊length[A]/2⌋ w_dół_do 1
3        wykonuj KOPIEC-PRZYWR(A,i)
```

a)



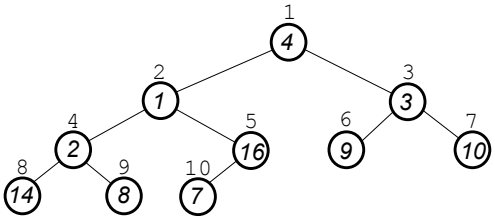
b)



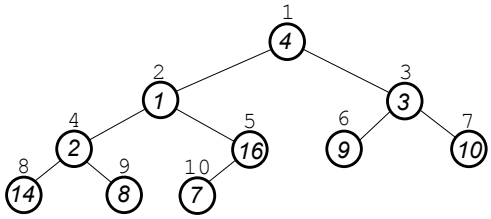
Rys.3.5 Zakres podtablicy liści (a) dla drzewa (b).

W algorytmie wykorzystano fakt, że w węzłach podtablicy $A[\lfloor n/2 \rfloor+1..n]$ własność kopca jest już spełniona. Następnie procedura przechodzi przez pozostałe węzły, tzn. węzły podtablicy $A[1..\lfloor n/2 \rfloor]$ i wywołuje w każdym z nich procedurę przywracania własności kopca. Węzły są przy tym odwiedzane w kolejności malejących numerów od $\lfloor n/2 \rfloor$ do 1. W każdym z węzłów zostaje wywołana procedura KOPIEC-PRZYWR, w efekcie czego poddrzewa zaczepione w tych węzłach stają się prawidłowymi kopcami.

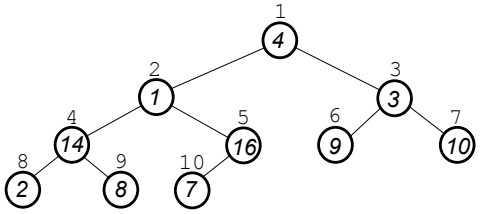
Przykładem użycia procedury KOPIEC-BUDUJ są pokazane na rys.3.7-3.11 kolejne kroki budowania kopca na podstawie tablicy z rys.3.5a. Drzewo wyjściowe przedstawia rys.3.6.



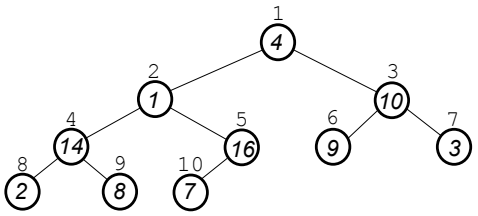
Rys.3.6 Stan wyjściowy do budowania kopca z elementów tablicy z rys.3.5a.



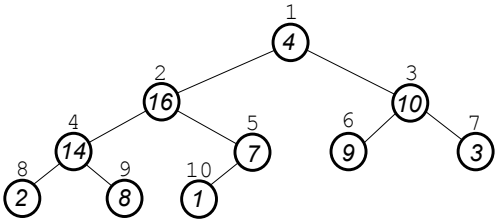
Rys.3.7 Stan po wywołaniu KOPIEC-PRZYWR(*A*,*5*).



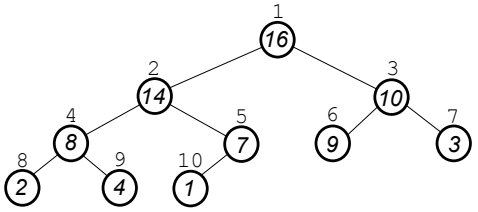
Rys.3.8 Stan po wywołaniu KOPIEC-PRZYWR(*A*,*4*).



Rys.3.9 Stan po wywołaniu KOPIEC-PRZYWR(*A*,*3*).



Rys.3.10 Stan po wywołaniu KOPIEC-PRZYWR(*A*,*2*).



Rys.3.11 Stan po wywołaniu KOPIEC-PRZYWR(*A*,*1*).

3.2.3 Kolejki priorytetowe

Kolejka priorytetowa to struktura przechowująca S elementów, z których każdy ma przyporządkowaną wartość klucza. Typowymi operacjami są wtedy: wstawianie nowego, a także wyszukiwanie oraz usuwanie elementu o maksymalnym kluczu. Przykładem zastosowania takiej kolejki jest przydzielanie zasobów w komputerze pracującym współbieżnie, tzn. wykonującym więcej niż jedno zadanie. Zadania do wykonania posiadają priorytety i są ustawiane w kolejce. Kiedy pewne zadanie kończy się lub zostaje przerwane, spośród oczekujących jest wybierane zadanie o najwyższym priorytecie, co oznacza usunięcie go z kolejki. Okazuje się, że kolejkę priorytetową oraz podstawowe operacje w takiej kolejce wygodnie jest zrealizować przy pomocy kopca.

Usuwanie maksymalnego elementu

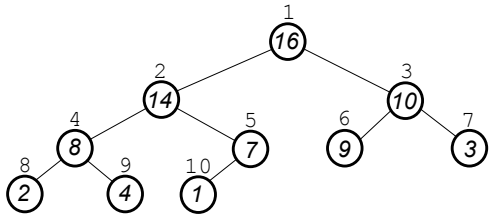
Usuwanie maksymalnego elementu z kolejki priorytetowej jest równoznaczne z usunięciem maksymalnego elementu z kopca reprezentującego tę kolejkę. Realizuje to algorytm K.3.12.

K.3.12

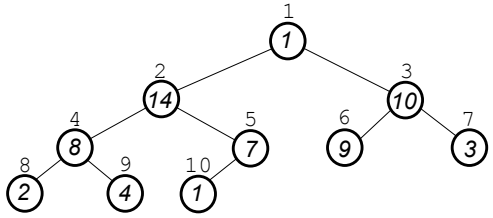
```
KOPIEC-USUŃ-MAX(A)
1     jeśli heap-size[A] < 1
2        to błąd „kopiec pusty”
3     max ← A[1]
4     A[1] ← A[heap-size[A]]
5     heap-size[A] ← heap-size[A]-1
6     KOPIEC-PRZYWR(A,1)
7     zwróć max
```

Linie 1 i 2 w funkcji K.3.12 zabezpieczają przed usuwaniem z pustej kolejki. Elementem usuwanym jest zawsze korzeń kopca reprezentującego kolejkę priorytetową, tj. element $A[1]$. Wartość tego elementu jest zwracana w linii 7 jako wynik działania funkcji. Linie od 4 do 6 służą do zmodyfikowania kopca (w tym przywrócenia własności kopca) po usunięciu z niego maksymalnego elementu.

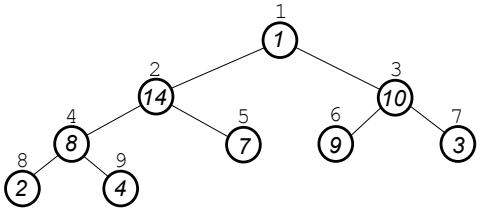
Na rys.3.13-3.15 zilustrowano kolejne etapy usuwania maksymalnego elementu z kolejki reprezentowanej przez kopiec z rys.3.12.



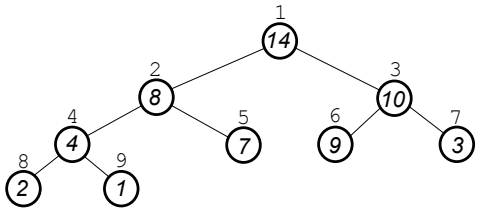
Rys.3.12 Kopiec przed usunięciem maksymalnego elementu.



Rys.3.13 Kopiec po wykonaniu polecenia z linii 4 w procedurze K.3.12.



Rys.3.14 Kopiec po wykonaniu polecenia z linii 5 w procedurze K.3.12.



Rys.3.15 Kopiec po wywołaniu procedury KOPIEC-PRZYWR(*A*,1).

Dodawanie nowego elementu

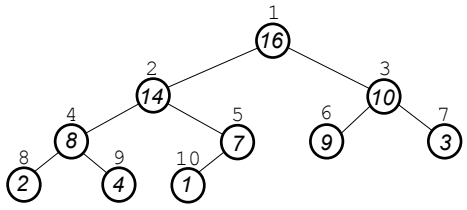
Dodanie nowego elementu do kolejki priorytetowej polega na powiększeniu kopca o nowy węzeł tak, aby został on wstawiony na właściwe miejsce. Operację taką realizuje algorytm K.3.13.

K.3.13

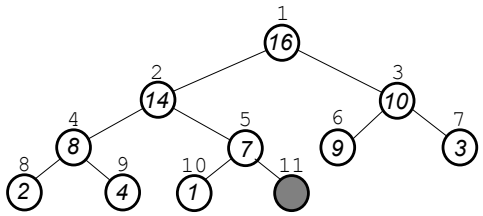
```
KOPIEC-WSTAW(A,key)
1       heap-size[A] ← heap-size[A]+1
2       i ← heap-size[A]
3       dopóki i > 1  i  A[PARENT(i)] < key
4          wykonuj  { A[i] ← A[PARENT(i)]
5                     i ← PARENT(i) }
6       A[i] ← key
```

Parametrami procedury wstawiania są: kopiec reprezentujący kolejkę priorytetową oraz klucz dodawanego elementu. Algorytm działa w ten sposób, że powiększa najpierw kopiec dodając nowy liść (linia 1). Następnie w sposób podobny jak procedura sortowania przez wstawianie, przechodzi ścieżkę od nowego liścia do korzenia celem określenia właściwego miejsca dla wstawianego elementu.

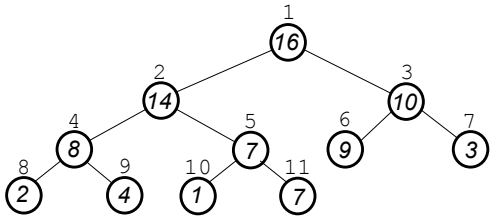
Na rys.3.17-3.20 zilustrowano kolejne etapy dodawania elementu z kluczem 15 do kolejki reprezentowanej przez kopiec z rys.3.16.



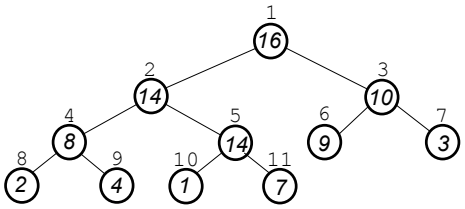
Rys.3.16 Kopiec przed dodaniem elementu z kluczem 15.



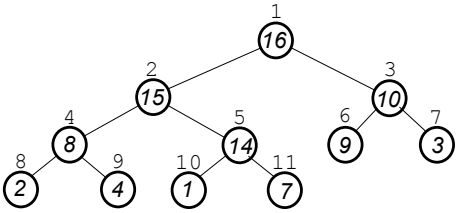
Rys.3.17 Kopiec po wykonaniu polecenia z linii 1 w procedurze K.3.13.



Rys.3.18 Kopiec po 1-szym wykonaniu pętli w liniach 3÷5 procedury K.3.13.



Rys.3.19 Kopiec po 2-gim wykonaniu pętli w liniach 3÷5 procedury K.3.13.



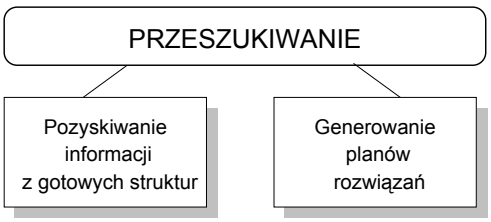
Rys.3.20 Kopiec po wykonaniu polecenia z linii 6 w procedurze K.3.13.

Czas działania procedury KOPIEC-WSTAW(A,*key*) na *n* elementowym kopcu wynosi $O(\lg n)$, ponieważ jest równy długości ścieżki poprowadzonej od nowego liścia do korzenia.

3.3 Przeszukiwanie w drzewach

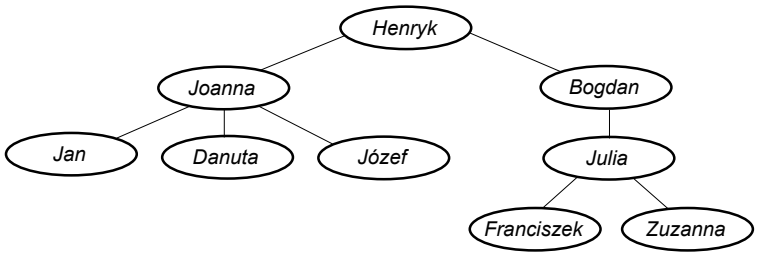
3.3.1 Typy zadań przeszukiwania

Opracowanie efektywnych procedur przeszukiwania jest kluczowym zagadnieniem w dziedzinie algorytmów. Na rys.3.21 wyodrębniono dwie podstawowe klasy problemów, które mogą być rozwiązywane przy pomocy przeszukiwania.



Rys.3.21 Klasy zadań przeszukiwania w drzewach.

Pierwsza grupa zadań dotyczy przypadków, kiedy z istniejącej struktury należy uzyskać określone informacje, np. wyszukać konkretny element w zadanym



Rys.3.22 Drzewo zależności przodek-potomek.

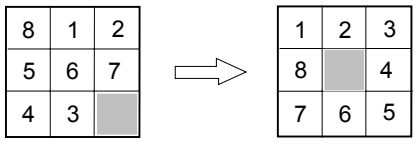
drzewie binarnym. Inne przykłady można podać rozważając drzewo na rys.3.22.

Jest to struktura pokazująca zależności przodek-potomek dla członków pewnej rodziny. Przeszukiwanie w takim drzewie jest podstawą do udzielania odpowiedzi m.in. na następujące pytania:

- czy Zofia należy do rodziny?
- czy Zuzanna jest potomkiem Bogdana?

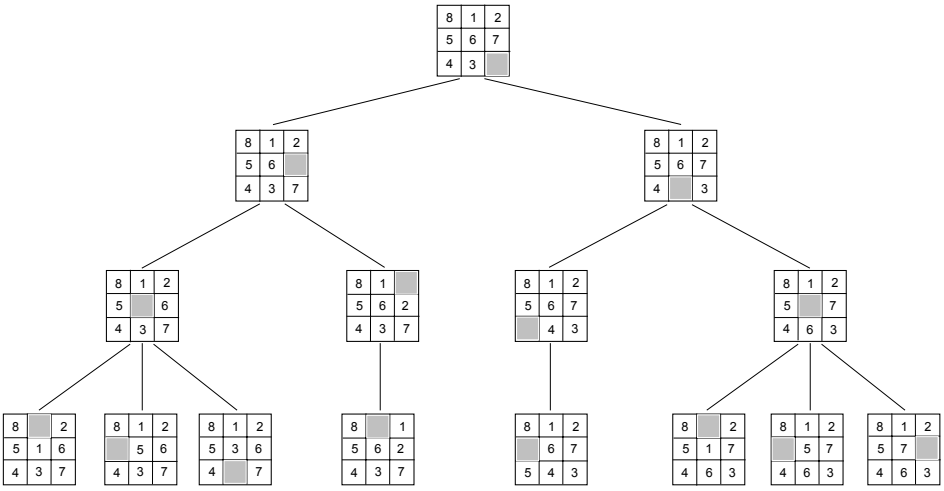
itp.

Druga grupa zadań przeszukiwania dotyczy generowania planów rozwiązań i może być zilustrowana przy pomocy prostej zabawki-układanki przedstawionej na rys.3.23 [2].



Rys.3.23 Układanka.

Układanka składa się z 8-miu segmentów, które mogą być przemieszczane w obrębie 9-ciu pól. Przyjmując za punkt wyjścia pewien stan początkowy, należy tak przesuwać segmenty, aby otrzymać nowy układ przyjęty jako stan docelowy. Zadanie przeszukiwania w tym przypadku polega na automatycznym wygenerowaniu takiego ciągu przesunięć segmentów, który prowadzi do stanu docelowego. Powstaje w ten sposób drzewo przeszukiwań, którego węzły reprezentują bieżące stany układanki tj. chwilowe układy segmentów. Dla pewnego węzła w drzewie przeszukiwań jego synami są stany osiągalne z tego węzła po wykonaniu pojedynczego przesunięcia (rys.3.24).



Rys.3.24 Drzewo stanów układanki.

Należy zwrócić uwagę na fakt, że w odróżnieniu od zadań pozyskiwania informacji z gotowych struktur, w zadaniach generowania planów rozwiązań

przeszukiwana struktura istnieje tylko konceptualnie, tzn. nie musi być wcześniej zbudowana w całości. Budowane są tylko jej niezbędne części, w miarę jak postępuje proces przeszukiwania.

Dla obu wymienionych klas zadań przeszukiwania konieczne jest rozwiązanie następujących problemów:

1. Dostęp do węzłów
2. Systematyczne sięganie do węzłów
3. Sprawdzanie węzłów.

Dostęp do węzłów

W przypadku przeszukiwania w gotowej strukturze dostęp do węzłów jest rozumiany jako możliwość odczytania wszystkich synów dowolnego węzła. Drzewa są zazwyczaj reprezentowane tak, że operacja ta jest bardzo prosta (np. wskaźniki do synów węzła *x* drzewa binarnego reprezentowanego za pomocą struktury ze wskaźnikami, odczytujemy za pomocą atrybutów *left*[*x*] i *right*[*x*], a dla drzew dowolnych należy wykorzystać atrybuty *leftchild*[*x*] oraz *rightsibling*[*x*]).

W zadaniach generowania planów rozwiązań dostęp do węzłów polega na otrzymaniu (wygenerowaniu) węzłów-synów dowolnego węzła, przy czym sposób otrzymania tych węzłów jest, ogólnie biorąc, zależny od typu zadania. W przypadku układanki konieczna jest procedura, która dla dowolnego układu segmentów generuje zbiór stanów „sąsiednich" otrzymywanych z bieżącego przez wykonanie pojedynczego przesunięcia.

Systematyczne sięganie do węzłów

Problem dostępu do węzłów jest ważnym, ale nie jedynym w zadaniach przeszukiwania. Załóżmy, że dla drzewa zależności przodek-potomek (rys.3.22) mamy odpowiedzieć na pytanie: czy węzeł Zuzanna występuje w tym drzewie? Potrzebna jest wtedy określona strategia, według której należy sięgać do kolejnych węzłów drzewa w trakcie przeszukiwania. Na ogół stosuje się dwie podstawowe strategie przeszukiwania:

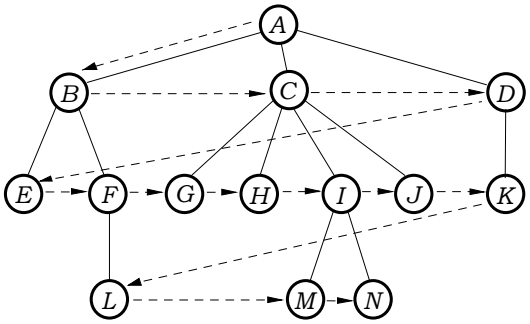
- strategia *wszerz* (*BFS – Breadth First Search*) oraz
- strategia *w głąb* – (*DFS - Depth First Search*).

Sprawdzanie węzłów

Operacja sprawdzania węzłów ma na celu ustalenie, czy węzeł osiągnięty w wyniku systematycznego sięgania jest węzłem poszukiwanym.

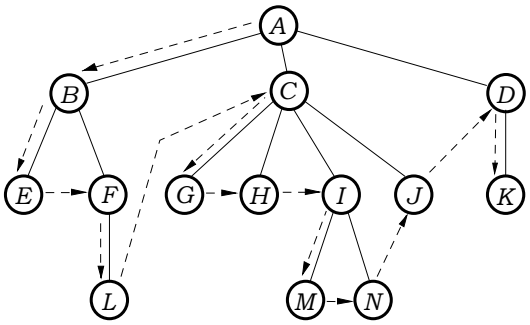
3.3.2 Strategie wszerz i w głąb

Jak już wspomniano systematyczne sięganie do węzłów odbywa się zwykle według strategii *wszerz* lub według strategii *w głąb*. Strategię sięgania *wszerz* pokazano schematycznie na rys.3.25. Zakłada ona przeglądanie kolejno wszystkich węzłów danego poziomu, a dopiero po ich wyczerpaniu przejście do następnego poziomu.



Rys.3.25 Przeszukiwanie wszerz.

Inaczej odbywa się przeglądanie węzłów według strategii w głąb (rys.3.26). W tym przypadku „odwiedzając" węzły należy poruszać się maksymalnie w dół, a dopiero gdy przestaje to być możliwe, powrócić do węzła wyższego poziomu.



Rys.3.26 Przeszukiwanie w głąb.

Inaczej odbywa się przeglądanie węzłów według strategii w głąb (rys.3.26). W tym przypadku „odwiedzając" węzły należy poruszać się maksymalnie w dół, a dopiero gdy przestaje to być możliwe, powrócić do węzła wyższego poziomu.

Postawimy teraz zadanie opracowania algorytmu przeglądającego węzły dowolnego drzewa według strategii wszerz. Zakłada się przy tym, że drzewo jest reprezentowane przy pomocy struktury ze wskaźnikami. Algorytm przeglądania realizuje funkcja DRZEWO-WSZERZ(*W1*,*W2*) (kod K.3.13). Funkcja ta sprawdza, czy węzeł *W2* jest węzłem potomnym węzła *W1*, tzn. czy *W2* występuje w poddrzewie, którego korzeniem jest *W1*.

K.3.13

```
DRZEWO-WSZERZ(W1, W2)
1  do_sprawdzenia ← nil
2  SYNOWIE(W1, do_sprawdzenia)
3  dopóki do_sprawdzenia ≠ nil
4      wykonuj  {  pierwszy ← PIERWSZY(do_sprawdzenia)
5      jeśli key[pierwszy] = key[W2] to zwróć W2
6      SYNOWIE(pierwszy, nowe_węzły)
7      RESZTA(do_sprawdzenia)
8      NOWE-NA-KONIEC(do_sprawdzenia, nowe_węzły) }
9   wypisz „węzła nie znaleziono"
```

W algorytmie wykorzystano dwie struktury listowe: *do_sprawdzenia* i *nowe_węzły,* zawierające odpowiednio węzły oczekujące aktualnie na sprawdzenie oraz węzły, które są synami bieżącego węzła. Użyto także następujących pomocniczych procedur i funkcji:

SYNOWIE(*w*, *lista_s*)	dla węzła *w* generuje listę *lista_s* węzłów-synów tego węzła,
PIERWSZY(*lista_w*)	dla niepustej listy węzłów *lista_w* zwraca wskaźnik na pierwszy węzeł,
RESZTA(*lista_w*)	w liście węzłów *lista_w* usuwa pierwszy węzeł,
NOWE-NA-KONIEC(*lista_w1*, *lista_w2*)	na koniec listy węzłów *lista_w1* dodaje listę węzłów *lista_w2*.

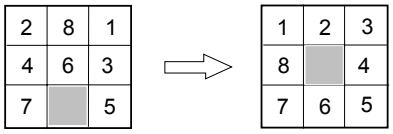
Ich opracowanie pozostawia się Czytelnikowi.

Algorytm przeszukiwania w głąb można otrzymać dokonując niewielkiej modyfikacji funkcji DRZEWO-WSZERZ. Podstawowa zmiana dotyczy linii 8, gdzie wywołanie procedury NOWE-NA-KONIEC należy zastąpić wywołaniem analogicznej procedury NOWE-NA-POCZĄTEK, która dołączy nowe węzły, będące synami bieżącego węzła nie na koniec, lecz na początek listy *do_sprawdzenia*.

3.3.3 Przeszukiwanie sterowane i niesterowane

Strategie wszerz i w głąb są przykładami przeszukiwania *niesterowanego* w tym sensie, że porządek według którego są sprawdzane węzły drzewa jest z góry określony, a aktualna zawartość informacyjna drzewa nie ma na niego wpływu. W niektórych zastosowaniach zachodzi potrzeba sterowania przeszukiwaniem w celu osiągnięcia lepszego rozwiązania. W takim przypadku metoda nie ogranicza się wyborem pierwszego węzła z listy *do_sprawdzenia*, lecz wybierany jest taki węzeł, który według pewnego kryterium jest najbardziej odpowiedni, aby przeszukiwanie było kontynuowane w kierunku tego właśnie węzła. Można np. zastosować kryterium polegające na wyborze węzła „najbardziej podobnego“ do poszukiwanego. Taki sposób postępowania jest nazywany *przeszukiwaniem sterowanym*, a zastosowana strategia nosi nazwę *najpierw najlepszy* (*BF - best first*).

Wybór „najbardziej podobnego“ węzła w drzewie przeszukiwań można przedstawić na przykładzie układanki z rys.3.23. Rozważymy dwa stany układanki pokazane na rys.3.27.



Rys.3.27 Aktualny i docelowy stan układanki.

Aby ocenić podobieństwo stanów aktualnego i docelowego wprowadzimy miarę podobieństwa (odległości) pomiędzy stanami. Przyjmuje się następujące ustalenia:

1. Dla każdego segmentu układanki wprowadza się wagę równą liczbie pozycji o jaką jest on odległy od swojej pozycji docelowej.

2. Do wagi segmentu z pkt.1 dodaje się 3, jeśli segment jest w pozycji środkowej oraz 6, jeśli jest on na obrzeżach i nie poprzedza segmentu, który docelowo po nim następuje.

Stosując te ustalenia można określić odległość stanów aktualnego od docelowego z rys.3.27. Wyniki zamieszczono w tab.3.1.

Tab.3.1 Tabela naliczania odległości stanów układanki z rys.3.27.

Segment	**Naliczona waga**
1	2+6=8
2	1+6=7
3	1+6=7
4	2+6=8
5	0+6=6
6	1+3=4
7	0+6=6
8	2+0=2
Odległość stanów (suma)	48

Podobne oceny odległości stanu aktualnego od docelowego mogą posłużyć jako kryterium wyboru drogi w drzewie przeszukiwań i tym samym stanowić podstawę przeszukiwania sterowanego.

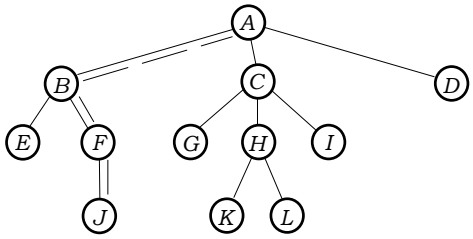
3.3.4 Przeszukiwanie pełne

Omawiane dotychczas algorytmy były tak skonstruowane, że przeszukiwanie kończyło się z chwilą napotkania pierwszego węzła, który spełniał zadane kryterium. Można jednak postawić wymóg wyszukania wszystkich węzłów spełniających to kryterium, a zatem kontynuować przeszukiwanie w całej strukturze. Taki sposób postępowania określimy jako *przeszukiwanie pełne*, a jego wynikiem jest w przypadku ogólnym lista wyszukanych węzłów.

Realizacja przeszukiwania pełnego wymagałaby niewielkiej modyfikacji algorytmu DRZEWO-WSZERZ, polegającej na wprowadzeniu listy *wynik,* do której po kolei byłyby dodawane węzły spełniające kryterium przeszukiwania. Przeszukiwanie kończyłoby się dopiero z chwilą wyczerpania listy węzłów *do_sprawdzenia*.

3.3.5 Generowanie dróg rozwiązań

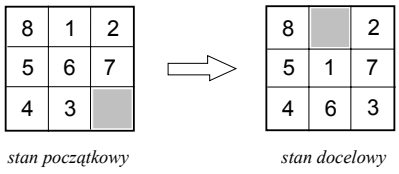
Rozważymy teraz problem poszukiwania drogi pomiędzy dwoma zadanymi węzłami, tzn. otrzymania pełnej listy węzłów występujących podczas przechodzenia od jednego węzła do drugiego. Na przykład drogę pomiędzy węzłami *A* i *J* w drzewie na rys.3.28 można zapisać w postaci listy węzłów (*A B F J*).



Rys.3.28 Drzewo z zaznaczoną drogą od węzła A do węzła J.

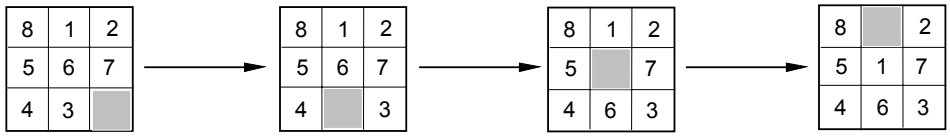
Szczególne znaczenie ma poszukiwanie drogi w drzewach, w których korzeń reprezentuje pewien stan początkowy a synowie dowolnego węzła możliwe stany osiągane z węzła-rodzica w wyniku wykonania elementarnej operacji. Załóżmy, że poszukujemy w drzewie węzła reprezentującego pewien stan docelowy określany jako rozwiązanie. W takim przypadku nie wystarcza tylko stwierdzenie, czy poszukiwane rozwiązanie istnieje, lecz należy podać drogę w drzewie przeszukiwań od stanu początkowego (korzenia) do rozwiązania. Droga ta wyznacza listę elementarnych operacji potrzebnych do przejścia od stanu początkowego do rozwiązania. Ten rodzaj przeszukiwania określimy jako *generowanie dróg rozwiązań*.

W przypadku układanki elementarną operacją jest dozwolone, pojedyncze przesunięcie segmentu. Algorytm, który automatycznie generuje rozwiązanie dla układanki powinien zwracać w wyniku listę kolejnych konfiguracji segmentów, z których pierwsza jest stanem początkowym, ostatnia stanem docelowym, a każde dwie sąsiednie powinny być osiągalne jedna z drugiej przy pomocy elementarnej operacji. Załóżmy, że należy znaleźć rozwiązanie dla układanki w sytuacji pokazanej na rys.3.29.



Rys.3.29 Aktualny i docelowy stan układanki w zadaniu poszukiwania drogi rozwiązania.

Przykładem drogi rozwiązania otrzymanej na podstawie drzewa z rys.3.24. jest sekwencja stanów układanki pokazana na rys.3.30.



Rys.3.30 Droga rozwiązania dla sytuacji z rys.3.29.

Pytania kontrolne

1. Podać definicję:
   a) drzewa skierowanego;
   b) drzewa nieskierowanego.
2. Wyjaśnić następujące pojęcia:
   a) rodzic i syn węzła;
   b) przodek i potomek węzła;
   c) korzeń, poddrzewo, liść;
   d) głębokość i wysokość węzła oraz wysokość drzewa;
   e) drzewo uporządkowane i drzewo binarne;
   f) regularne drzewo binarne;
   g) pełne drzewo binarne.
3. Omówić tablicową reprezentację drzew binarnych.
4. Omówić sposoby odwiedzania wierzchołków drzewa binarnego.

5. Przedstawić sposoby reprezentacji drzew za pomocą struktur wskaźnikowych.
6. Podać definicję drzewa przeszukiwań binarnych oraz omówić podstawowe operacje w drzewach BST.
7. Podać sposób reprezentowania oraz podstawową własność kopca.
8. Omówić procedury:
   a) przywracania własności kopca;
   b) budowania kopca.
9. Omówić kolejkę priorytetową oraz operacje:
   a) usuwania maksymalnego elementu;
   b) wstawiania nowego elementu.
10. Scharakteryzować dwa podstawowe typy zadań przeszukiwania w drzewach.
11. Omówić strategie przeszukiwania:
   a) wszerz (*breadth-first*),
   b) w głąb (*depth-first*),
   c) najpierw najlepszy (*best-first*).

Zadania

1. Napisać rekursywne procedury przechodzenia drzewa binarnego w porządkach:
   a) preorder; b) postorder; c) inorder.
   Dla każdego z punktów a), b) i c) rozważyć dwa warianty: i) drzewo jest reprezentowane tablicowo; ii) drzewo jest reprezentowane za pomocą struktur wskaźnikowych.
2. Wprowadzimy następujące pojęcia:
   Węzłem wewnętrznym nazywamy dowolny węzeł drzewa nie będący liściem. Stopień węzła to liczba jego synów. Rząd drzewa wynosi k, jeśli stopień dowolnego węzła nie przekracza k. Pełne drzewo rzędu k, to

drzewo rzędu k, w którym wszystkie liście mają tę samą głębokość, a wszystkie węzły wewnętrzne mają stopień k.
Wyprowadzić wzór na liczbę węzłów pełnego drzewa rzędu k, jeśli jego wysokość wynosi h.

3. Napisać procedury wykonania następujących operacji na drzewach BST:
   a) wyszukiwanie maksymalnego elementu;
   b) wyszukiwanie poprzednika zadanego węzła;
   c) usuwanie zadanego węzła.
4. Napisać procedurę przeszukiwania drzewa w głąb.
5. Napisać procedurę budowania drzewa przeszukiwań binarnych z elementów uporządkowanej tablicy jednowymiarowej.

4 Sortowanie

4.1 Wprowadzenie

Sortowanie polega na zmianie kolejności elementów pewnego ciągu tak, aby zostały one ustawione w porządku niemalejącym lub nierosnącym. W zbiorze, do którego należą elementy ciągu, powinien być określony *porządek*.

Definicja 4.1

Częściowym porządkiem na zbiorze *S* nazywamy relację *R*, która posiada następujące właściwości:

- *Zwrotność*. Dla dowolnego $a \in S$, zachodzi aRa.
- *Przechodniość*. Dla dowolnych $a,b,c \in S$, zachodzi: jeśli aRb i bRc to aRc.
- *Antysymetryczność*. Jeśli aRb i bRa to $a = b$.

Przykładami częściowych porządków są: relacja ≤ w zbiorze liczb całkowitych oraz relacja $\subseteq$ dla zbiorów.

Definicja 4.2

Całkowitym (*liniowym*) *porządkiem* na zbiorze *S* nazywamy częściowy porządek *R* na *S*, taki, że dla dowolnych $a,b \in S$, zachodzi aRb albo bRa.

Relacja ≤ jest przykładem liniowego porządku w zbiorze liczb całkowitych, natomiast relacja $\subseteq$ dla zbiorów nie jest liniowym porządkiem.

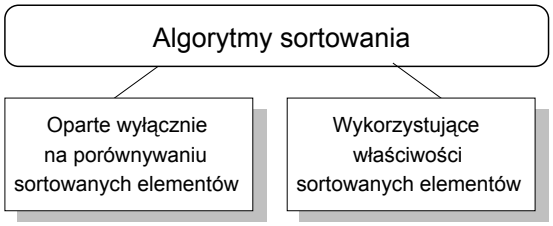
Definicja 4.3

Niech *S* będzie zbiorem, w którym jest określony liniowy porządek ≤, a $\langle a_1, a_2, ..., a_n \rangle$ ciągiem elementów z *S*. Zadaniem *sortowania* jest wyznaczenie permutacji, tj. pewnej kolejności $\langle a_1', a_2', ..., a_n' \rangle$, zadanych *n* elementów takiej, że

$$a_1' \le a_2' \le ... \le a_n' .$$

Rozważymy dwie następujące klasy algorytmów sortowania (rys.4.1):

1. Algorytmy oparte wyłącznie na porównywaniu sortowanych elementów. Dla takich algorytmów wykażemy, że do posortowania ciągu złożonego z n elementów potrzeba co najmniej $n\lg n$ porównań.
2. Algorytmy korzystające z pewnych własności sortowanych elementów (np. z faktu, że sortowane liczby są jednostajnie rozłożone w przedziale $[0,1)$. Dla takich algorytmów nie obowiązuje dolna granica $\Omega(n \lg n)$.



Rys.4.1 Klasy algorytmów sortowania.

Inny podział algorytmów sortowania może być dokonany ze względu na użycie pamięci. Wyróżniamy wtedy:

- algorytmy *wewnętrzne*, gdy dane są przechowywane wyłącznie w pamięci wewnętrznej,
- algorytmy *zewnętrzne*, gdy w przeważającej części dane znajdują się poza pamięcią wewnętrzną.

Jeśli sortowanie jest częścią innego algorytmu, wówczas liczba sortowanych elementów nie jest zwykle duża, co pozwala je zmieścić w pamięci wewnętrznej. Z kolei sortowanie zewnętrzne dotyczy głównie takich zastosowań, jak operacje w systemach księgowych, gdzie danych jest zwykle znacznie więcej, niż może pomieścić pamięć wewnętrzna.

4.2 Algorytmy oparte na porównaniach

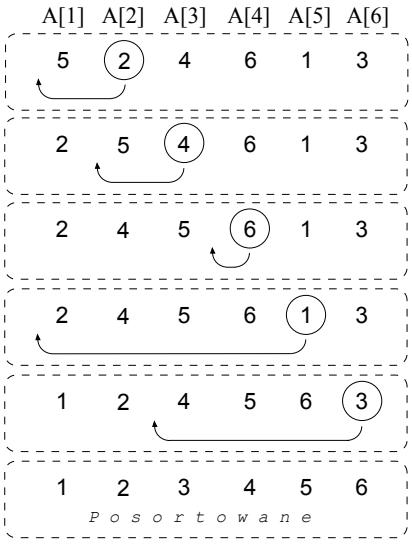
4.2.1 Drzewa decyzyjne

Rozważymy ponownie algorytm sortowania przez wstawianie (kod K.4.1).

K.4.1

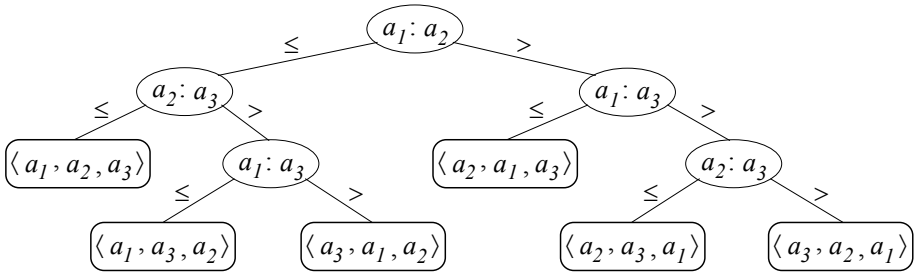
```
SORT-WSTAW(A)
1   dla j←2 do length[A]
2     wykonuj { key ← A[j]
3           ▷ Wstaw A[j] do posortowanego ciągu A[1..j-1]
4           i ← j-1
5           dopóki i>0 i A[i]>key
6             wykonuj { A[i+1] ← A[i]
7                       i ← i-1 }
8           A[i+1]← key }
```

Jest to efektywny algorytm w przypadku sortowania niewielkiej liczby elementów, a jego działanie można porównać do układania talii kart w ręku. Zaczynając od „pustej" ręki dokładamy kolejne karty do już uporządkowanej talii w ręku tak, aby po dodaniu karty były nadal uporządkowane. Elementy tablicy $A[1..j\text{-}1]$ reprezentują (uporządkowane) karty trzymane w ręce, a elementy $A[j+1..n]$ reprezentują nieuporządkowany stos kart na stole. Indeks j wskazuje na kartę aktualnie wstawianą do talii. Indeks ten, począwszy od wartości 2, przesuwa się w tablicy w stronę rosnących indeksów. W każdej iteracji pętli zewnętrznej (linie 1-8) pobierany jest z tablicy element $A[j]$. Następnie w pętli wewnętrznej (linie 5-7) jest poszukiwane właściwe miejsce dla elementu $A[j]$ w już uporządkowanej części tablicy. W tym celu, począwszy od pozycji j-1, elementy tablicy są przemieszczane w stronę rosnących indeksów dopóty, aż nie „zwolni" się miejsce, w którym powinien być umieszczony element $A[j]$. Sortowanie tablicy $A = \langle 5,2,4,6,1,3 \rangle$ za pomocą opisywanego algorytmu przedstawiono na rys.4.2.



Rys.4.2 Sortowanie przez wstawianie.

Sortowanie przez wstawianie jest przykładem sortowania wewnętrznego, a algorytm jest oparty wyłącznie na porównaniach sortowanych elementów. W takich algorytmach ustalenie względnego położenia elementów a_i oraz a_j jest możliwe za pośrednictwem porównania $a_i \le a_j$, a do ilustracji wykonywanych porównań wygodnie jest użyć struktury nazywanej drzewem decyzyjnym. Na rysunku 4.3 przedstawiono drzewo decyzyjne odpowiadające algorytmowi sortowania ciągu $\langle a_1, a_2, a_3 \rangle$.



Rys.4.3 Drzewo decyzyjne dla sortowanie 3-ch elementów.

Drzewo decyzyjne pokazuje zatem porównania wykonywane przez algorytm sortujący dla ustalonego rozmiaru danych wejściowych. Inne aspekty algorytmu, jak sterowanie i przepływ danych nie są tutaj uwzględniane.

Każdy węzeł wewnętrzny odpowiada porównaniu elementów a_i oraz a_j $(1 \le i, j \le 3)$. Z każdym liściem drzewa jest natomiast związana pojedyncza permutacja ciągu wejściowego. Wykonanie algorytmu polega na przejściu od korzenia do jednego z liści, a dojście do liścia oznacza, że algorytm znalazł już uporządkowanie (permutację) elementów. Sortowanie na podstawie drzewa decyzyjnego z rys.4.3 można prześledzić zakładając

$$a_1 = -1, \quad a_2 = 2, \quad a_3 = 0.$$

W tym przypadku algorytm wykona się „wzdłuż" ścieżki:

$$a_1 : a_2 \rightarrow a_2 : a_3 \rightarrow a_1 : a_3 \rightarrow \langle a_1, a_3, a_2 \rangle .$$

4.2.2 Dolne ograniczenie na czas sortowania

Jak wynika z rys.4.3, długość najdłuższej ścieżki od korzenia drzewa decyzyjnego do liścia odpowiada liczbie porównań dokonywanej przez algorytm sortujący w przypadku pesymistycznym. Jeśli zatem uda się określić dolne ograniczenie na wysokość drzew decyzyjnych, to będzie ono jednocześnie dolnym ograniczeniem na czas działania algorytmów sortujących wyłącznie za pomocą porównań. Oceny dolnego ograniczenia można dokonać korzystając ze sformułowanego i udowodnionego dalej twierdzenia [5].

Twierdzenie 4.1

Każde drzewo decyzyjne odpowiadające algorytmowi poprawnie sortującemu *n* elementów ma wysokość $\Omega(n \lg n)$.

Dowód

Rozważymy drzewo decyzyjne o wysokości *h* odpowiadające algorytmowi sortującemu *n* elementów. Liczba permutacji dla ciągu złożonego z *n* elementów wynosi *n*!, drzewo to ma zatem *n*! liści. Ponieważ drzewo binarne o wysokości *h* nie może mieć więcej niż 2^h liści stąd

$$n! \le 2^h .$$

Logarytmując obustronnie ostatnią nierówność otrzymujemy

$$h \ge \lg(n!) \qquad (4.1)$$

Dalej skorzystamy z następującego wzoru Stirlinga

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right),$$

z którego wynika, że

$$n! > \left(\frac{n}{e}\right)^n \tag{4.2}$$

gdzie $e = 2{,}71828\ldots$ jest podstawą logarytmu naturalnego. Uwzględniając nierówność (4.2) otrzymujemy na podstawie (4.1):

$$h \geq \lg\left(\frac{n}{e}\right)^n = n\lg n - n\lg e = \Omega(n\lg n).$$

Uwaga

W wyniku dokonanej wcześniej analizy algorytmu sortowania przez wstawianie otrzymano złożoność pesymistyczną $O(n^2)$. Oznacza to, że górne ograniczenie na czas działania tego algorytmu nie jest równe dolnej granicy $\Omega(n\lg n)$ z naszego twierdzenia. Mówimy, że nie jest to asymptotycznie optymalny algorytm sortujący wyłącznie za pomocą porównań.

4.2.3 Algorytmy asymptotycznie optymalne

Rozważymy dwa przykłady algorytmów asymptotycznie optymalnych: sortowanie przez kopcowanie i sortowanie przez scalanie.

Sortowanie przez kopcowanie (*heapsort*)

Algorytm sortowania przez kopcowanie (kod K.4.2) używa kopca zbudowanego z elementów sortowanej tablicy *A*.

K.4.2

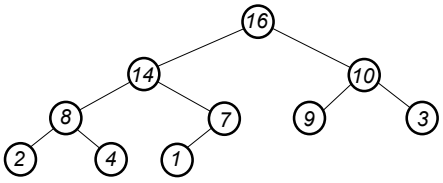
```
SORT-KOPCOW(A)
1     KOPIEC-BUDUJ(A)
2     dla i← length[A] w_dół_do 2
3       wykonuj { zamień A[1]↔A[i]
```

4 *heap-size*[*A*] ← *heap-size*[*A*]-1
5 KOPIEC-PRZYWR(*A*,1) }

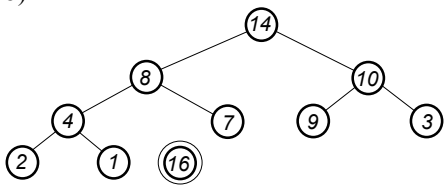
	1	2	3	4	5	6	7	8	9	10	11
A	16	14	10	8	7	9	3	2	4	1	

Rys.4.4 Tablica elementów do posortowania.

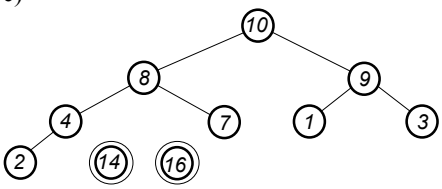
a)



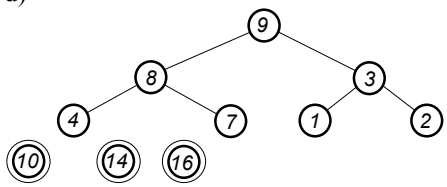
b)



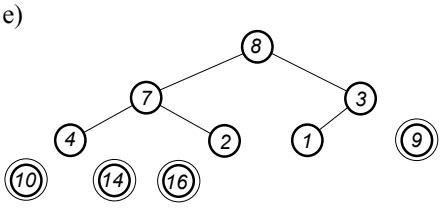
c)



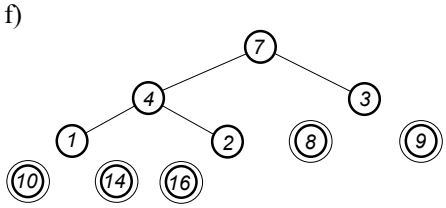
d)



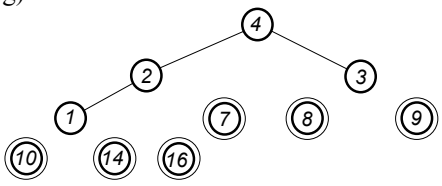
e)



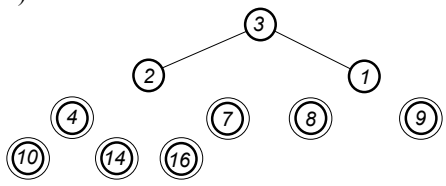
f)



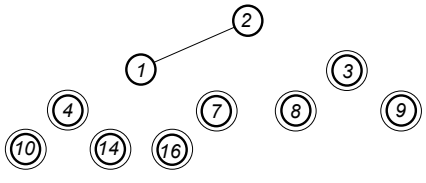
g)



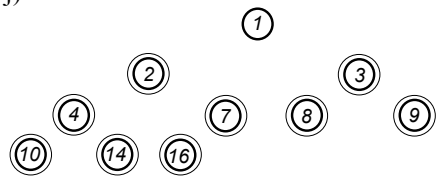
h)



i)



j)



Rys.4.5 Sortowanie przez kopcowanie tablicy z rys.4.4.

Algorytm ten rozpoczyna działanie używając podanej wcześniej procedury budowania kopca z *n*-elementowej tablicy *A*. Ponieważ największy element znajduje się w korzeniu *A*[1], stąd może on zostać umieszczony na swoim miejscu, tj. na końcu tablicy A przez zamianę miejscami z *A*[*n*]. Po tej zamianie odrzucamy element *A*[*n*] zmniejszając rozmiar kopca o 1 i przywracamy własność kopca dla pozostałych elementów. Taka operacja jest następnie powtarzana dla kopca o rozmiarze *n*-1, *n*-2, itd., aż otrzymamy kopiec o rozmiarze 2.

Działanie algorytmu sortowania przez kopcowanie zilustrowano na przykładzie sortowania tablicy *A* zawierającej 10 liczb (rys.4.4). Rysunki 4.5a÷4.5j pokazują postać kopca oraz fragment posortowanej tablicy na początku kolejnych iteracji pętli **dla** w linii 2.

Algorytm ten wykonuje funkcję budowania kopca działającą w czasie $O(n)$, a następnie dokonuje *n*-1 wywołań funkcji przywracania właściwości kopca, z których każde zajmuje czas $O(\lg n)$. Stąd ogólny czas działania całej procedury sortowania przez kopcowanie wynosi $O(n \lg n)$.

Metoda „dziel i zwyciężaj" i sortowanie przez scalanie

Stosowanie rekurencji sprzyja zwięzłemu i eleganckiemu zapisowi algorytmów, lecz samo w sobie nie zwiększa ich efektywności (np. w sensie skrócenia czasu obliczeń). Jeśli jednak rekurencję dopełnić pewnymi dodatkowymi zabiegami, jak równoważenie oraz zastosowanie metody określanej jako „dziel i zwyciężaj" można uzyskać bardziej efektywne algorytmy.

W podejściu „dziel i zwyciężaj" występuje rekursja, przy czym każdy poziom rekursji ma następującą strukturę:

dziel	-	dzielimy problem na podproblemy;
zwyciężaj	-	na tym etapie rozwiązujemy podproblemy rekurencyjnie wykorzystując metodę „dziel i zwyciężaj", chyba, że są one tak małych rozmiarów, że nie wymagają zastosowania

rekursji;

połącz - łączymy rozwiązania podproblemów, aby otrzymać rozwiązanie całego problemu.

Metodę „dziel i zwyciężaj” zaprezentujemy na przykładzie algorytmu sortowania przez scalanie działającego w czasie $\Theta(n \lg n)$. Dla uproszczenia będziemy zakładać, że sortowanym ciągiem jest

$$\langle x_1, x_2, ..., x_n \rangle, \qquad n = 2^k, \ k \in N$$

a poszukuje się permutacji

$$\langle x'_1, x'_2, ..., x'_n \rangle$$

elementów ciągu początkowego takich, że

$$x'_1 \le x'_2 \le ... \le x'_n .$$

Etapy algorytmu są następujące:

1. Dziel Dzielimy *n*-elementowy ciąg na dwa podciągi po *n*/2 elementów każdy.

2. Zwyciężaj Sortujemy otrzymane podciągi, używając rekurencyjnie sortowania przez scalanie. Jeśli na którymś poziomie rekursji otrzymujemy do posortowania ciąg jednoelementowy, wówczas nie następuje dalsze zagłębianie wywołań rekursywnych.

3. Połącz Łączymy ciągi posortowane na niższych poziomach rekursji w jeden posortowany ciąg.

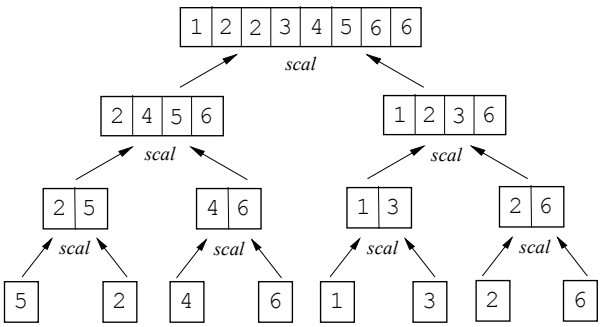
Główna procedura sortowania została przedstawiona jako kod K.4.3.

K.4.3

```
SORT-SCALANIE(A, p, r)
1  jeśli p<r
2    to { q ← ⌊(p+r)/2⌋
3         SORT-SCALANIE(A, p, q)
4         SORT-SCALANIE(A, q+1, r)
5         SCALAJ(A, p, q, r) }
```

Procedura SORT-SCALANIE(A,p,r) sortuje podtablicę $A[p..r]$ złożoną z m elementów (m=r-p+1). Jeśli $p \geq r$, to tablica jest już posortowana i nie następuje dalsze zagłębianie wywołań rekursywnych. W przeciwnym razie dzieli się tablicę $A[p..r]$ na dwie podtablice: $A[p..q]$ zawierającą $\lceil m/2 \rceil$ elementów oraz $A[q..r]$ zawierającą $\lfloor m/2 \rfloor$ elementów. Dla tak otrzymanych tablic wywołuje się procedurę sortowania przez scalanie (linie 3 i 4). Następnie posortowane „połówki" powinny zostać połączone (scalone) tak, aby w wyniku otrzymać także posortowaną tablicę (linia 5). Opracowanie procedury scalającej pozostawia się czytelnikowi (zob zad.2 na końcu rozdziału). Aby zatem posortować tablicę $A = \langle A[1], A[2], A[3],..., A[length[A]] \rangle$, procedurę sortowania wywołuje się następująco: SORT-SCALANIE(A,1,$length$[A]).

Schemat sortowania przez scalanie dla $A = \langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$ pokazano na rys.4.6.



Rys.4.6 Sortowanie przez scalanie.

Ocenimy teraz złożoność czasową algorytmu sortowania przez scalanie. Jest to algorytm rekurencyjny, który wykorzystuje metodę „dziel i zwyciężaj". Czas działania $T(n)$ takich algorytmów można opisać zależnością rekurencyjną postaci:

$$T(n) = \begin{cases} \Theta(1), & n \leq s \\ aT(n/b) + D(n) + C(n), & n > s \end{cases} \tag{4.3}$$

Zapis ten oznacza, że jeśli problem jest dostatecznie mały (rozmiar zadania nie przekracza pewnej stałej s), wówczas jego rozwiązanie zajmuje stały czas $\Theta(1)$. Jeśli natomiast dzielimy problem na a podproblemów i każdy z nich jest

rozmiaru *n/b*, to czas rozwiązania tych podproblemów wyniesie $aT(n/b)$. Do tej wielkości należy dodać czas $D(n)$ przeznaczony na dzielenie problemu na podproblemy oraz czas scalania $C(n)$.

W przypadku sortowania przez scalanie zależność rekurencyjna (4.3) ma postać:

$$T(n)=\begin{cases}\Theta(1), & n=1\\ 2T(n/2)+\Theta(n), & n>1\end{cases} \tag{4.4}$$

gdzie składnik $\Theta(n)$ oznacza czas scalania, a czas dzielenia, jako stały pominięto. Złożoność obliczeniową algorytmu sortowania przez scalanie można otrzymać rozwiązując równanie rekurencyjne (4.4). Rozwiązanie to można otrzymać wprost, jeśli skorzystać z podanego niżej twierdzenia [5].

Twierdzenie 4.2

Niech *a*,*b*,*c* będą nieujemnymi stałymi. Rozwiązanie równania rekurencyjnego

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

dla *n* będących potęgą liczby *c* ma postać

$$T(n)=\begin{cases}O(n), & a<c\\ O(n\lg n), & a=c\\ O\left(n^{\log_c a}\right), & a>c\end{cases}.$$

Z powyższego twierdzenia, którego dowód pominiemy, wynika, że złożoność obliczeniowa algorytmu sortowania przez scalanie wynosi $O(n\lg n)$. Uwzględniając wcześniej udowodnione twierdzenie 4.1 otrzymujemy, że złożoność ta wynosi $\Theta(n\lg n)$.

4.3 Sortowanie w czasie liniowym

Przedstawimy dwa algorytmy, dla których nie obowiązuje dolna granica na czas sortowania $\Omega(n\lg n)$, jak to ma miejsce w przypadku algorytmów opartych wyłącznie na porównaniach. Prezentowane tutaj algorytmy oprócz porównań

wykorzystują pewne specyficzne właściwości sortowanych elementów, przy czym może się zdarzyć, że porównania nie są w ogóle konieczne.

4.3.1 Sortowanie przez zliczanie

W algorytmie sortowania przez zliczanie zakładamy, że każdy z n sortowanych elementów jest liczbą całkowitą z przedziału $1..k$. Jeśli $k = O(n)$, to wtedy sortowanie działa w czasie $O(n)$. Idea algorytmu polega na określeniu dla każdego elementu x, ile elementów jest mniejszych lub równych x. Mając tę liczbę, możemy określić dokładną pozycję x w posortowanym ciągu. Procedurę sortowania przez zliczanie przedstawia kod K.4.4.

K.4.4

```
SORT-ZLICZANIE(A, B, k)
1    dla i←1 do k
2      wykonuj C[i] ← 0
3    dla j←1 do length[A]
4      wykonuj C[A[j]] ← C[A[j]] + 1
5    ▷ C[i] zawiera teraz liczbę elementów równych i
6    dla i←2 do k
7      wykonuj C[i] ← C[i] + C[i-1]
8    ▷ C[i] zawiera teraz liczbę elementów mniejszych lub równych i
9    dla j← length[A] w_dół_do 1
10     wykonuj { B[C[A[j]]] ← A[j]
11               C[A[j]] ← C[A[j]] – 1 }
```

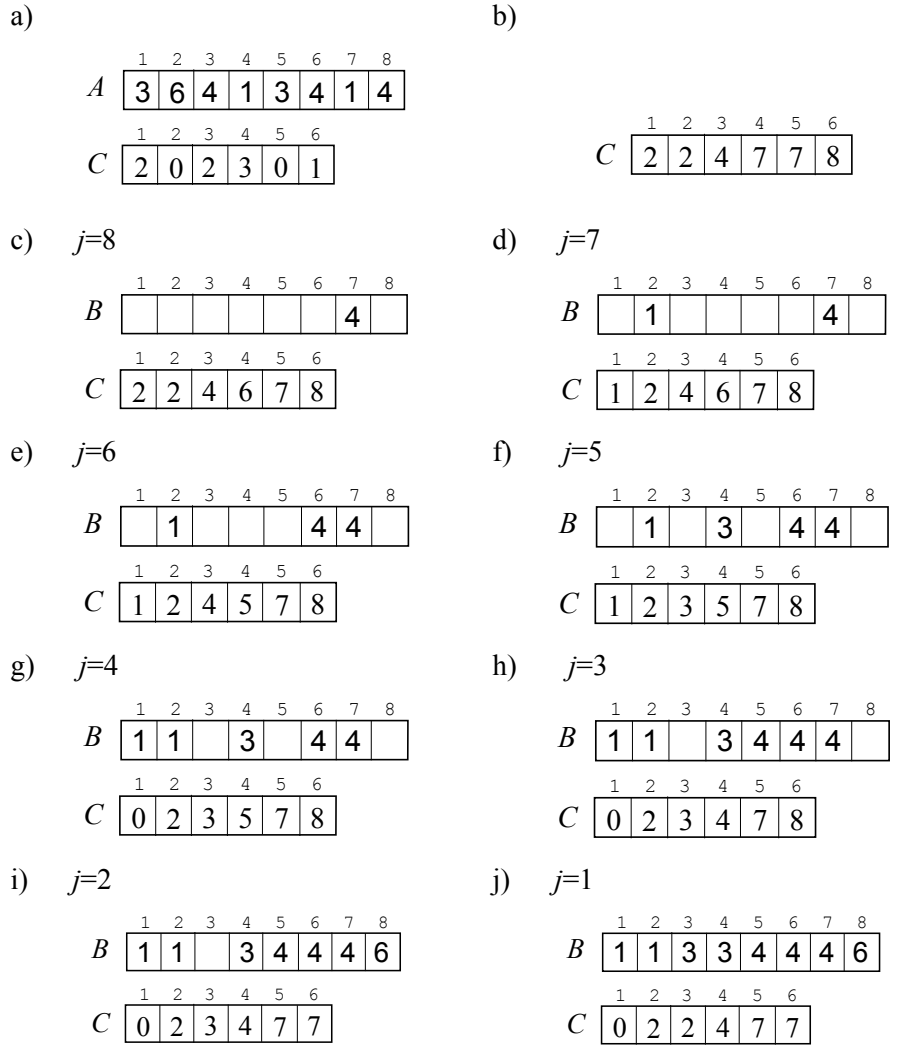
W procedurze SORT-ZLICZANIE przyjęto, że dane wejściowe są elementami tablicy $A[1..n]$, $(n = length[A])$. Tablica $B[1..n]$ jest tablicą wyjściową, gdzie zostaną umieszczone posortowane elementy, a tablica $C[1..k]$ jest tablicą pomocniczą. W liniach 1-2 tablica C jest zerowana, natomiast operacje w liniach 3-4 prowadzą do zapisu w każdym elemencie $C[i]$ liczby całkowitej odpowiadającej ilości elementów tablicy wejściowej równych i. Następnie w liniach 6-7 jest wyznaczana i zapisywana w $C[i]$ liczba elementów tablicy wejściowej mniejszych lub równych i. Ostatecznie w liniach 9-11 wszystkie elementy zostają rozmieszczone na właściwych miejscach w tablicy B.

Działanie algorytmu K.4.4 zilustrujemy na przykładzie sortowania tablicy

$$A = \langle 3, 6, 4, 1, 3, 4, 1, 4 \rangle,$$

dla której $n = 8$ oraz $k = 6$.

Po wykonaniu linii 4 tablice A i C mają postać pokazaną na rys.4.7a, natomiast po wykonaniu linii 7 tablica C zostaje wypełniona w sposób przedstawiony na rys.4.7b. Rysunki 4.7c-4.7j pokazują postać tablic B i C po kolejnych iteracjach wykonywanych w liniach 9-11.



Rys.4.7 Ilustracja algorytmu sortowania przez zliczanie dla tablicy ⟨3,6,4,1,3,4,1,4⟩.

Ocenimy czas działania algorytmu sortowania przez zliczanie. Pętle w liniach 1-2 oraz 3-4 wymagają odpowiednio czasów $O(k)$ oraz $O(n)$. Ponieważ analogiczne wymogi mają pętle w liniach 6-7 oraz 9-11, stąd całkowity czas działania procedury K.4.4 można zapisać jako $O(k+n)$. Z założenia $k = O(n)$ wynika dalej, że czas ten wynosi $O(n)$, a zatem jest ograniczony funkcją liniową względem liczby sortowanych elementów *n*. Ocena ta świadczy o dużej efektywności algorytmu sortowania przez zliczanie.

4.3.2 Sortowanie kubełkowe

Algorytm sortowania kubełkowego również należy do algorytmów szybkich, tj. sortujących w czasie liniowym. Podobnie jak w przypadku sortowania przez zliczanie, tutaj także nakłada się ograniczenie na dane wejściowe. Polega ono na założeniu, że sortowane są liczby wybrane losowo (z rozkładem jednostajnym) z przedziału $[0..1)$. Procedurę sortowania kubełkowego przedstawiono w postaci kodu K.4.5.

K.4.5

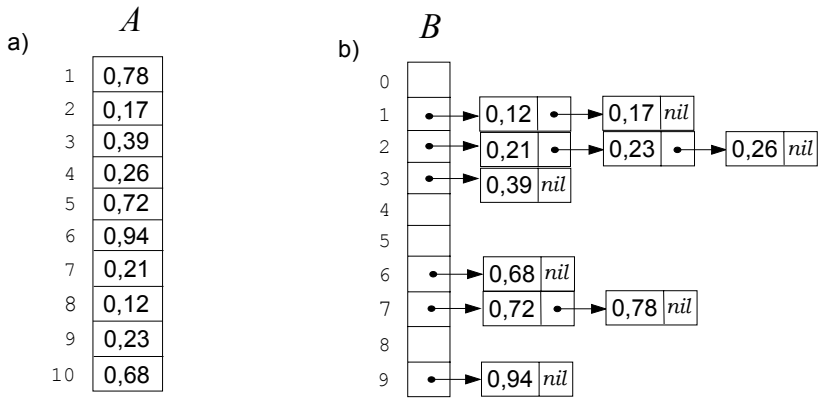
```
SORT-KUBEŁKOWE(A)
1    n ← length[A]
2    dla i←1 do n
3      wykonuj wstaw A[i] do listy B[⌊n A[i]⌋]
4    dla i←0 do n-1
5      wykonuj posortuj listę B[i] przez wstawianie
6    połącz listy B[0], B[1],..., B[n-1] w tej kolejności
```

Parametrem wejściowym jest *n*-elementowa tablica *A*, której elementy spełniają zależność $0 \le A[i] < 1$. Algorytm sortowania zakłada podział przedziału $[0..1)$ na *n* podprzedziałów jednakowych rozmiarów nazywanych „kubełkami”, a następnie rozrzucenie *n* liczb do tych podprzedziałów. W tym celu jest wykorzystana tablica list $B[0..n-1]$ reprezentujących kubełki. Liczby w podprzedziałach są następnie sortowane przez wstawianie, a posortowane listy z każdego podprzedziału łączy się ze sobą.

Działanie algorytmu zilustrujemy na przykładzie 10-elementowej tablicy

$$A = \langle 0{,}78;\ 0{,}17;\ 0{,}39;\ 0{,}26;\ 0{,}72;\ 0{,}94;\ 0{,}21;\ 0{,}12;\ 0{,}23;\ 0{,}68 \rangle .$$

Rysunek 4.8a przedstawia nieposortowaną tablicę A, natomiast rys.4.8b tablicę B po wykonaniu pętli w liniach 4-5 procedury.



Rys.4.8 Sortowanie kubełkowe.

Poprawność sortowania kubełkowego

W celu wykazania, że algorytm sortowania kubełkowego działa poprawnie, rozważa się ([5]) dwa dowolne elementy $A[i]$ i $A[j]$. Jeśli podczas sortowania trafią one do tego samego kubełka, to w tablicy wynikowej na pewno wystąpią w poprawnej kolejności, ponieważ liczby w każdym kubełku są sortowane przez wstawianie, a poprawność tej metody zakładamy. Załóżmy zatem, że elementy $A[i]$ i $A[j]$ trafią do różnych kubełków o numerach odpowiednio i' i j', a ponadto $i' < j'$. Po połączeniu kubełków elementy listy $B[i']$ znajdą się przed elementami listy $B[j']$, a zatem element $A[i]$ poprzedzi $A[j]$ w ciągu wynikowym. Musi wtedy zachodzić $A[i] \le A[j]$, co wykażemy metodą „nie wprost", zakładając, że jest przeciwnie. Otrzymujemy wówczas

$$i' = \lfloor nA[i] \rfloor \ge \lfloor nA[j] \rfloor = j'$$

co jest sprzeczne z wcześniejszym założeniem, że $i' < j'$.

Złożoność sortowania kubełkowego

Można również wykazać ([5]), że czas działania algorytmu sortowania kubełkowego jest liniowy względem liczby sortowanych elementów. Ponieważ pesymistyczny czas wykonania operacji we wszystkich liniach oprócz linii 5

wynosi $O(n)$, pozostaje dokonać analizy sortowania wszystkich list $B[i]$, $i = 0,1,...n-1$ zawierających elementy w kubełkach. Oznaczymy przez n_i zmienną losową odpowiadającą liczbie elementów w kubełku $B[i]$. Pesymistyczny czas sortowania przez wstawianie wynosi $O(n^2)$, a zatem oczekiwany czas posortowania elementów listy $B[i]$ wynosi

$$\mathrm{E}\left[O(n_i^2)\right] = O(\mathrm{E}[n_i^2]).$$

Ocenimy teraz całkowity oczekiwany czas posortowania elementów we wszystkich kubełkach, tzn.

$$\sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) = O\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right). \tag{4.5}$$

Mamy *n* elementów i *n* list. Ponieważ do każdej listy trafiają elementy z $1/n$ części całego przedziału $[0,1)$, to prawdopodobieństwo tego, że dany element trafi do listy $B[i]$, jest jednakowe dla wszystkich list i wynosi $p = 1/n$. Zmienna losowa n_i ma zatem rozkład dwumianowy z wartością oczekiwaną

$$\mathrm{E}[n_i] = np = 1$$

i wariancją

$$\mathrm{Var}[n_i] = np(1-p) = 1 - \frac{1}{n}.$$

Na podstawie znanej zależności dla wartości oczekiwanej i wariancji zmiennej losowej otrzymujemy

$$\mathrm{E}[n_i^2] = \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n} = \Theta(1).$$

Wynik ten, w połączeniu z (4.5), pozwala stwierdzić, że czas posortowania elementów we wszystkich kubełkach jest liniowy, a zatem oczekiwany czas sortowania kubełkowego jest także liniowy.

Pytania kontrolne

1. Podać definicję oraz przykłady porządku częściowego i porządku liniowego.

2. Sformułować ogólnie zadanie sortowania oraz scharakteryzować dwie klasy algorytmów sortowania.
3. Wyjaśnić pojęcia sortowania wewnętrznego i zewnętrznego.
4. Omówić na przykładzie użycie drzewa decyzyjnego do prezentacji algorytmu sortowania za pomocą porównań.
5. Podać dolne ograniczenie na pesymistyczny czas sortowania w algorytmach wykorzystujących tylko porównania i udowodnić jego poprawność.
6. Przedstawić algorytm sortowania przez wstawianie i oszacować jego złożoność czasową.
7. Przedstawić algorytm sortowania przez kopcowanie i oszacować jego złożoność czasową.
8. Na przykładzie algorytmu sortowania przez scalanie przedstawić istotę metody „dziel i zwyciężaj".
9. Dla algorytmu sortowania przez scalanie pokazać zastosowanie zależności rekurencyjnej do oszacowania czasu działania algorytmu.
10. Omówić algorytm sortowania przez zliczanie działający w czasie liniowym i wyjaśnić fakt, że dla tego algorytmu nie obowiązuje granica $\Omega(n \lg n)$.
11. Omówić algorytm sortowania kubełkowego i wykazać jego poprawność. Jaka cecha łączy algorytmy sortowania przez zliczanie i sortowania kubełkowego?

Zadania

1. Skonstruować drzewo decyzyjne odpowiadające algorytmowi sortowania przez wstawianie dla ciągu a_1, a_2, a_3, a_4.
2. Napisać procedurę SCALAJ(*A,p,q,r*) realizującą scalanie posortowanych tablic $A[p..q]$ i $A[q+1..r]$ w jedną posortowaną tablicę $A[p..r]$, tak, aby mogła ona być wykorzystana w algorytmie sortowania przez scalanie.
3. Zilustrować działanie procedury sortowania przez zliczanie dla tablicy $A = \langle 7,1,3,1,2,4,5,7,2,4,3 \rangle$.
4. Zilustrować działanie procedury sortowania kubełkowego dla tablicy $A = \langle 0{,}79, 0{,}13, 0{,}16, 0{,}64, 0{,}39, 0{,}20, 0{,}89, 0{,}53, 0{,}71, 0{,}42 \rangle$.

5 Bibliografia

[1] **Aho A.V., Hopcroft J.E., Ullman J.D.**: *Projektowanie i analiza algorytmów komputerowych*. PWN, 1983.

[2] **Anderson J.R., Corbett A.T., Reiser B.J.**: *Essential LISP*. Addison-Wesley Publishing Company, 1987.

[3] **Banachowski L., Diks K., Rytter W.**: *Algorytmy i struktury danych.* Wydawnictwa Naukowo-Techniczne, 1996.

[4] **Banachowski L., Kreczmar A.**: *Elementy analizy algorytmów*. Wydawnictwa Naukowo-Techniczne, 1982.

[5] **Cormen T.H., Leiserson C.E,, Rivest R.L.**: *Wprowadzenie do algorytmów.* Wydawnictwa Naukowo-Techniczne, 1997.

[6] **Drozdek A., Simon D.L.**: *Struktury danych w języku C*. Wydawnictwa Naukowo-Techniczne, 1996.

[7] **Knuth D.E.**: *Sztuka programowania. Tom 1. Algorytmy podstawowe*. Wydawnictwa Naukowo-Techniczne, 2002.

[8] **Winston P.H., Horn B.K.P.**: *LISP. Third Edition*. Addison-Wesley Publishing Company, 1989.

DODATEK

Zadania na egzaminy i kolokwia

W uzupełnieniu zostaną podane przykładowe zestawy zadań, jakie pojawiały się w ostatnich latach na egzaminach i kolokwiach. Przy każdym zadaniu w nawiasach podano maksymalną liczbę punktów, na którą może być oceniona prawidłowa odpowiedź. Czas rowiązywania w przeliczeniu na 1 punkt wynosił średnio 2 minuty. Możliwe jest wystąpienie w różnych zestawach podobnych, a nawet identycznych zadań. Pozostawiono je celowo, aby Czytelnik mógł zmierzyć się z „oryginalnymi" zestawami obejmującymi z reguły odpowiednio dobrany zakres materiału.

Zestaw 1

1. Zdefiniować jednym zdaniem następujące pojęcia:
   a) głębokość wierzchołka drzewa; (1)
   b) operand typu *i* w języku maszyny RAM; (1)
   c) silnie spójny graf skierowany; (1)
   d) procedura rekurencyjna; (1)
   e) algorytm sortujący w czasie liniowym; (1)
   f) asymptotycznie optymalny algorytm sortujący wyłącznie za pomocą porównań. (1)
2. Napisać rekurencyjną procedurę obliczającą *n*-tą potęgę liczby *m* (*m>0*). (2)
3. Rozważyć następujący program w języku maszyny RAM:

```
      LOAD =0
      STORE 1
      LOAD =1
      STORE 2
ET1:  LOAD =5
      SUB 2
      JGTZ Et2
      JUMP Et3
 Et2: LOAD 1
      ADD 2
      STORE 1
      LOAD 2
      ADD  =1
      STORE 2
      JUMP ET1
Et3:  WRITE 1
      HALT
```

a) Jaka jest waga logarytmiczna operandu w pierwszej komendzie LOAD? (1)
b) Jaka jest wartość operandu w komendzie z etykietą Et1? (1)
c) Jakiego typu adres występuje w komendzie JGTZ? (1)
d) Ile razy wykona się komenda ADD 2 ? (1)
e) Podać wynik działania programu (1)
f) Ile rejestrów wykorzystano w obliczeniach ? (1)
g) Jakie zadanie pełni tutaj rejestr 0 ? (1)

4. Rozważyć wyrażenie: (((2-1)*3)+1).
   a) Podać odwrotny zapis polski tego wyrażenia. (1)
   b) Przedstawić ciąg operacji na stosie wykonywanych w trakcie otrzymania zapisu postfiksowego. (2)
   c) Przedstawić ciąg operacji na stosie wykonywanych w trakcie obliczania wartości otrzymanego wyrażenia postfiksowego. (2)

5. Zakładamy że liczby: 1, 3, 5, 7, 9, 0 są odpowiednio elementami A[1],..., A[6] tablicy A.
   a) Sporządzić rysunek pokazujący rozmieszczenie tych liczb w węzłach odpowiedniego drzewa binarnego reprezentującego kopiec zbudowany w wyniku wywołania procedury KOPIEC-BUDUJ(A). (2)
   b) Przedstawić rysunek pokazujący stan po usunięciu maksymalnego elementu z kolejki priorytetowej reprezentowanej przez kopiec z p. a). (2)
   c) Jak będzie wyglądała kolejka priorytetowa reprezentowana przez kopiec z p. a) po wykonaniu operacji dodawania elementu o wartości 8? (2)

6. Zakłada się, że tablica A=⟨3,4,1,4⟩ jest sortowana przez zliczanie według następującego algorytmu:

```
SORT-ZLICZANIE(A, B, k)
1  dla i←1 do k
2    wykonuj C[i] ← 0
3  dla j←1 do length[A]
4    wykonuj C[A[j]] ← C[A[j]] + 1
5  dla i←2 do k
6    wykonuj C[i] ← C[i] + C[i-1]
7  dla j← length[A] w_dół_do 1
8    wykonuj { B[C[A[j]]] ← A[j]
9  C[A[j]] ← C[A[j]] – 1 }
```

   a) Jaka jest zawartość tablicy C przed wejściem do pętli w liniach 7-9? (1)
   b) Jaka jest zawartość tablicy C po wykonaniu procedury? (1)
   c) Jaka jest zawartość tablicy B po wykonaniu procedury? (1)
   d) Jaka jest zawartość tablicy A po wykonaniu procedury? (1)

Zestaw 2

1. Zdefiniować jednym zdaniem następujące pojęcia:
   a) złożoność pamięciowa; (1)
   b) lista (definicja rekurencyjna); (1)
   c) stopień węzła w drzewie; (1)
   d) rząd drzewa; (1)
   e) procedura rekurencyjna; (1)
   f) algorytm sortujący w czasie liniowym. (1)

2. Napisać w pseudokodzie procedurę , która dla *i*-tego elementu kopca wypisuje po kolei elementy tworzące drogę od tego elementu do korzenia kopca. (2)

3. Rozważyć program:

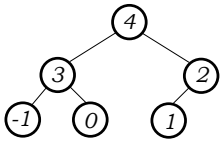
```
        LOAD =1
        STORE 1
        LOAD =1
        STORE 2
ET1:    LOAD =4
        SUB 2
        JGTZ Et2
        JUMP Et3
 Et2:   LOAD 1
        MULT 2
        STORE 1
        LOAD 2
        ADD  =1
        STORE 2
        JUMP ET1
Et3:    WRITE 1
        HALT
```

a) Jaka jest waga logarytmiczna drugiej komendy STORE? (1)
b) Jaka jest wartość operandu w komendzie z etykietą Et1? (1)
c) Jakiego typu adres występuje w komendzie JGTZ? (1)
d) Ile razy wykona się komenda MULT 2 ? (1)
e) Podać wynik działania programu. (1)
f) Ile rejestrów wykorzystano w obliczeniach? (1)
g) Podać wagę logarytmiczną operandu komendy MULT podczas jej ostatniego wykonania. (1)

4. Zakładamy że liczby: 1, 3, 5, 7, 9, 0 są odpowiednio elementami A[1],..., A[6] tablicy A.
   a) Sporządzić rysunek pokazujący rozmieszczenie tych liczb w węzłach odpowiedniego drzewa binarnego reprezentującego kopiec zbudowany w wyniku wywołania procedury KOPIEC-BUDUJ(A). (2)

b) Przedstawić rysunek pokazujący stan po usunięciu maksymalnego elementu z kolejki priorytetowej reprezentowanej przez kopiec z p. a). (2)
c) Jak będzie wyglądała kolejka priorytetowa reprezentowana przez kopiec z p. a) po wykonaniu operacji dodawania elementu o wartości 8 ? (2)

5. Rozważyć następujące drzewo binarne:



a) Podać kolejność odwiedzania wierzchołków tego drzewa metodą preorder. (1)
b) Podać kolejność odwiedzania wierzchołków tego drzewa metodą postorder. (1)
c) Podać kolejność odwiedzania wierzchołków tego drzewa metodą inorder. (1)

6. Podać definicję porządku częściowego. (2)

7. Zakłada się, że tablica A=⟨3,4,1,4⟩ jest sortowana przez zliczanie według następującego algorytmu:

```
SORT-ZLICZANIE(A, B, k)
1     dla i←1 do k
2       wykonuj C[i] ← 0
3     dla j←1 do length[A]
4       wykonuj C[A[j]] ← C[A[j]] + 1
5     dla i←2 do k
6       wykonuj C[i] ← C[i] + C[i-1]
7     dla j← length[A] w_dół_do 1
8       wykonuj { B[C[A[j]]] ← A[j]
9                 C[A[j]] ← C[A[j]] – 1 }
```

Przedstawić, jak zmienia się zawartość tablic B i C po wykonaniu kolejnych iteracji w wierszach 7-9. Przed wejściem do pętli w liniach 7-9 C=⟨1,1,2,4⟩. (4)

Zestaw 3

1. Zdefiniować jednym zdaniem następujące pojęcia:
   a) rozmiar zadania; (1)
   b) notacja $f(n)=\Omega(g(n))$; (1)
   c) operand typu *i w języku maszyny RAM; (1)
   d) złożoność pesymistyczna; (1)
   e) lista dwukierunkowa; (1)
   f) struktury z ograniczonym dostępem; (1)
   g) cykl w grafie nieskierowanym. (1)

2. Zapisać w pseudojęzyku algorytm wypisujący elementy listy *L* reprezentowanej przy pomocy wskaźników. (3)

3. Rozważyć program:

```
        LOAD =10
        STORE 2
Et1:    READ 1
        LOAD 1
        JZERO Et3
        LOAD 1
        SUB 2
        JGTZ Et2
        JUMP Et1
Et2:    LOAD 1
        STORE 2
        JUMP ET1
Et3:    WRITE 2
        HALT
```

   a) Jaka jest wartość operandu w pierwszej komendzie LOAD? (1)
   b) Jaka jest wartość operandu w komendzie z etykietą Et1? (1)
   c) Jakiego typu adres występuje w komendzie JZERO? (1)
   d) (1) Podać wynik działania programu po wprowadzeniu liczb: 1 2 3 4 0.
   e) (1) Podać wynik działania programu po wprowadzeniu liczb: 10 20 30 40 0.

4. Rozważyć graf nieskierowany G={V, E}, gdzie:
   V={a, b, c, d},
   E={(a,b), (a,c), (b,c), (b,d),(c,d)}.
   a) Narysować ten graf. (1)
   b) Naszkicować reprezentację grafu G z użyciem list sąsiedztwa. (2)
   c) Naszkicować reprezentację grafu G z użyciem macierzy sąsiedztwa. (2)

5. Rozważyć wyrażenie: (2-(1*(3+1))).
   a) Podać odwrotny zapis polski tego wyrażenia. (1)
   b) Przedstawić ciąg operacji na stosie wykonywanych w trakcie otrzymywania zapisu postfiksowego. (2)
   c) Przedstawić ciąg operacji na stosie wykonywanych w trakcie obliczania wartości otrzymanego w p. b) wyrażenia postfiksowego. (2)

6. Zakładamy że liczby: 1, 3, 6, –1, 0, 4, 2 są odpowiednio elementami A[1],..., A[7] tablicy A.
   a) Przedstawić rysunki pokazujące układ tych liczb w tablicy A oraz w węzłach odpowiedniego drzewa binarnego po zbudowaniu kopca. (2)
   b) Jak będzie wyglądała kolejka priorytetowa reprezentowana przez kopiec z rys a) po wykonaniu operacji usuwania maksymalnego elementu? (2)
   c) Rozmieścić liczby (pierwotnej) tablicy A w węzłach drzewa binarnego, tak, aby powstało drzewo BST o minimalnej głębokości. (1)

Zestaw 4

1. Zdefiniować jednym zdaniem następujące pojęcia:
   a) głębokość wierzchołka drzewa; (1)
   b) drzewo niezorientowane; (1)
   c) operand typu *i* w języku maszyny RAM; (1)
   d) silnie spójny graf skierowany; (1)
   e) lista dwukierunkowa; (1)
   f) sortowanie wewnętrzne; (1)
   g) cykl w grafie skierowanym. (1)

2. Zapisać w pseudojęzyku procedurę MAX wyznaczającą maksymalną spośród czytanych kolejno liczb. Procedura ta kończy pracę z chwilą napotkania zera, wypisując maksymalną liczbę. (3)

3. Rozważyć program:

```
        LOAD =10
        STORE 2
Et1:    READ 1
        LOAD 1
        JZERO Et3
        LOAD 1
        SUB 2
        JGTZ Et2
        JUMP Et1
Et2:    LOAD 1
        STORE 2
        JUMP ET1
Et3:    WRITE 2
        HALT
```

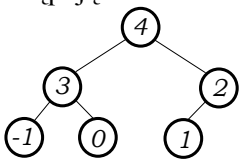
   a) Jaka jest wartość operandu w pierwszej komendzie LOAD? (1)
   b) Jaka jest wartość operandu w komendzie z etykietą Et1? (1)
   c) Jakiego typu adres występuje w komendzie JZERO? (1)
   d) (1) Podać wynik działania programu po wprowadzeniu liczb: 1 2 3 4 0.
   e) (1) Podać wynik działania programu po wprowadzeniu liczb: 10 20 30 40 0.

4. Rozważyć graf skierowany G={V, E}, gdzie:
   V={a, b, c, d},
   E={(a,b), (a,c), (b,d), (c,b),(d,c), (d,d)}.
   a) Narysować ten graf. (1)
   b) Naszkicować reprezentację grafu G z użyciem list sąsiedztwa. (2)
   c) Naszkicować reprezentację grafu G z użyciem macierzy sąsiedztwa. (2)

5. Rozważyć wyrażenie: (((2-1)*3)+1).
   a) Podać odwrotny zapis polski tego wyrażenia. (1)
   b) Przedstawić ciąg operacji na stosie wykonywanych w trakcie otrzymywania zapisu postfiksowego. (2)

c) Przedstawić ciąg operacji na stosie wykonywanych w trakcie obliczania wartości otrzymanego w p. b) wyrażenia postfiksowego. (2)

6. Rozważyć następujące drzewo binarne:



a) Podać kolejność odwiedzania wierzchołków tego drzewa metodą preorder. (1)
b) Podać kolejność odwiedzania wierzchołków tego drzewa metodą postorder. (1)
c) Podać kolejność odwiedzania wierzchołków tego drzewa metodą inorder. (1)
d) Czy drzewo to ma własność drzewa BST? (1)
e) Czy drzewo to ma własnooość kopca? (1)

Zestaw 5

1. Zdefiniować jednym zdaniem następujące pojęcia:
   a) rozmiar zadania; (1)
   b) równomierne kryterium wagowe; (1)
   c) operand typu *i* w języku maszyny RAM; (1)
   d) spójny graf nieskierowany; (1)
   e) procedura rekurencyjna. (1)

2. Zapisać w pseudojęzyku procedurę, która wypisuje minimalny element tablicy jednowymiarowej *A* zadanej jako parametr. (3)

3. Rozważyć program:

```
        LOAD =0
        STORE 1
        LOAD =1
        STORE 2
ET1:    LOAD =5
        SUB 2
        JGTZ Et2
        JUMP Et3
Et2:    LOAD 1
        ADD 2
        STORE 1
        LOAD 2
        ADD  =1
        STORE 2
        JUMP ET1
Et3:    WRITE 1
        HALT
```

   a) Jaka jest waga logarytmiczna operandu w pierwszej komendzie STORE? (1)

b) Jaka jest wartość operandu w komendzie z etykietą Et1? (1)
c) Jakiego typu adres występuje w komendzie JGTZ? (1)
d) Ile razy wykona się komenda ADD 2 ? (1)
e) Podać wynik działania programu. (1)
f) Ile rejestrów wykorzystano w obliczeniach? (1)
g) Jakie zadanie pełni tutaj rejestr 0 ? (1)

4. Zdefiniować jednym zdaniem następujące pojęcia:
   a) regularne drzewo binarne; (1)
   b) podstawowa własność kopca; (1)
   c) węzeł wewnętrzny drzewa; (1)
   d) drzewo poszukiwań binarnych; (1)
   e) asymptotycznie optymalny algorytm sortujący wyłącznie za pomocą porównań. (1)

5. Napisać iteracyjną wersję algorytmu poszukiwania w drzewie BST elementu o podanym kluczu. (3)

6. Podać istotę metody „dziel i zwyciężaj”. (2)

7. Zakładamy że liczby: 1, 2, 3, 4, 5, 6 są odpowiednio elementami A[1],..., A[6] tablicy A.
   a) Sporządzić rysunki pokazujące rozmieszczenie tych liczb w tablicy A oraz w węzłach odpowiedniego drzewa binarnego tak, aby był to stan wyjściowy dla procedury budowania kopca. (1)
   b) Przedstawić rysunki podobne jak w p. a), ale po zbudowaniu kopca. (2)
   c) Jak będzie wyglądała kolejka priorytetowa reprezentowana przez kopiec z rys. b) po wykonaniu operacji usuwania maksymalnego elementu? (2)