

# Heapsort – sortowanie przez kopcowanie

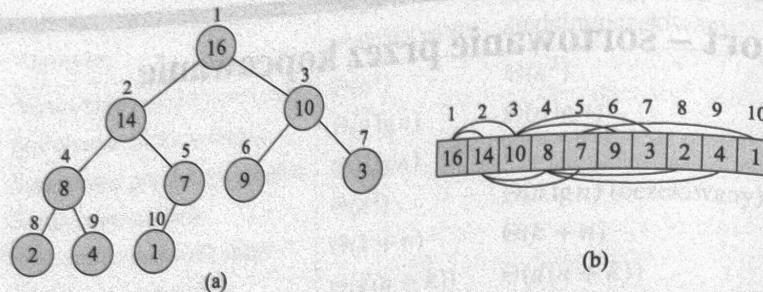
W tym rozdziale wprowadzimy nowy algorytm sortowania – sortowanie przez kopcowanie (heapsort). Podobnie jak dla algorytmu sortowania przez scalanie, ale nie jak dla sortowania przez wstawianie, czas działania algorytmu sortowania przez kopcowanie wynosi  $O(n \lg n)$ . Tak jak sortowanie przez wstawianie, ale inaczej niż sortowanie przez scalanie, algorytm heapsort sortuje w miejscu: tylko stała liczba elementów tablicy jest w czasie działania algorytmu przechowywana poza tablicą wejściową. A zatem algorytm ten łączy dobre cechy opisanych wcześniej algorytmów.

W algorytmie heapsort wprowadza się także nowy sposób konstrukcji algorytmów: używa się tu struktury danych, zwanej „kopcem”, do przetwarzania danych w czasie działania algorytmu. Kopiec jest nie tylko użyteczny dla algorytmu heapsort, ale działa również jako wydajna kolejka priorytetowa. Kopce pojawią się znowu w algorytmach opisanych w dalszych rozdziałach.

## 6.1 Kopce

**Kopiec (binarny)** jest to tablicowa struktura danych, którą można rozpatrywać jako prawie pełne drzewo binarne (patrz dodatek B.5.3), jak to widać na rys. 6.1. Każdy węzeł drzewa odpowiada pewnemu elementowi tablicy. Drzewo jest pełne na wszystkich poziomach z wyjątkiem, być może, najniższego, który jest wypełniony od strony lewej do pewnego miejsca. Tablica  $A[1 : n]$  reprezentująca kopiec to obiekt z atrybutem  $A.\text{heap-size}$ , określającym liczbę elementów kopca przechowywanych w tablicy. Oznacza to, że chociaż cała tablica  $A[1 : n]$  może zawierać jakieś wartości, to elementami kopca są jedynie te mieszczące się w  $A[1 : A.\text{heap-size}]$ , gdzie  $0 \leq A.\text{heap-size} \leq n$ . Jeśli  $A.\text{heap-size} = 0$ , to kopiec jest pusty. Korzeniem drzewa jest  $A[1]$ , a mając dany indeks  $i$  węzła, można łatwo obliczyć indeksy jego ojca  $\text{PARENT}(i)$ , lewego syna  $\text{LEFT}(i)$  i prawego syna  $\text{RIGHT}(i)$ .

Na większości komputerów procedura  $\text{LEFT}$  może policzyć  $2i$  za pomocą jednej instrukcji, po prostu przesuwając binarną reprezentację  $i$  o jeden bit w lewo. Podobnie, procedura  $\text{RIGHT}$  może szybko policzyć  $2i + 1$  przez przesunięcie binarnej reprezentacji  $i$  w lewo o jeden bit i dodanie jedynki. Procedura  $\text{PARENT}$  może policzyć  $\lfloor i/2 \rfloor$  przez przesunięcie  $i$  w prawo o jeden



Rysunek 6.1 Kopiec typu max rozpatrywany jako (a) drzewo binarne i (b) tablica. Liczba w kółku w każdym węźle drzewa jest wartością przechowywaną w tym węźle. Liczba nad węzłem jest odpowiadającym mu indeksem w tablicy. Linie łączące elementy tablicy przedstawiają relację ojciec-syn; ojcowie są zawsze na lewo od synów. Drzewo ma wysokość 3; poddrzewo węzła o indeksie 4 (zawierającego wartość 8) ma wysokość 1.

bit. W dobrej implementacji algorytmu heapsort te trzy procedury są często implementowane jako „makra” albo procedury „in-line”.

```
PARENT( $i$ )
1 return  $\lfloor i/2 \rfloor$ 

LEFT( $i$ )
1 return  $2i$ 

RIGHT( $i$ )
1 return  $2i + 1$ 
```

Kopce występują w dwóch odmianach: kopce typu max i kopce typu min. W obu przypadkach wartości w węzłach spełniają **własność kopca**, zależną od typu kopca. W **kopcach typu max** spełniona jest **własność kopca typu max**: dla każdego węzła  $i$ , który nie jest korzeniem, zachodzi  $A[\text{PARENT}(i)] \geq A[i]$ ,

tzn. że wartość w węźle jest nie większa niż wartość przechowywana w jego ojcu. Stąd wynika, że największy element kopca typu max jest umieszczony w korzeniu, a poddrzewa każdego węzła zawierają wartości mniejsze niż wartość przechowywana w tym węźle. Budowa kopca typu min jest odwrotna; **własność kopca typu min** polega na tym, że dla każdego węzła  $i$ , który nie jest korzeniem, zachodzi

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

Najmniejszy element kopca typu min znajduje się w korzeniu.

W algorytmie heapsort stosujemy kopiec typu max. Kopce typu min są najczęściej wykorzystywane w kolejkach priorytetowych, które będziemy omawiać w podrozdz. 6.5. Będziemy precyzyjnie określać, czy w konkretnym zastosowaniu potrzebujemy kopca typu max czy typu min, a w przypadku własności odnoszących się do obu typów będziemy mówili po prostu „kopiec”.

Traktując kopiec jako drzewo, definiujemy **wysokość węzła** w kopcu jako liczbę krawędzi na najdłuższej prostej ścieżce prowadzącej od tego węzła do liścia, a wysokość kopca jako wysokość jego korzenia. Ponieważ kopiec mający  $n$  elementów ma kształt pełnego drzewa binarnego, jego wysokość wynosi  $\Theta(\lg n)$  (patrz zad. 6.1-2). Zobaczmy, że podstawowe operacje na kopcach działają w czasie co najwyżej proporcjonalnym do wysokości drzewa, czyli  $O(\lg n)$ . Resztę tego rozdziału poświęcamy kilku podstawowym procedurom i pokazujemy, jak można ich użyć w algorytmie sortowania i strukturze kolejki priorytetowej.

- Procedura **MAX-HEAPIFY**, działająca w czasie  $O(\lg n)$ , służy do przywracania własności kopca typu max.
- Procedura **BUILD-MAX-HEAP**, działająca w czasie liniowym, tworzy kopiec typu max z nieuporządkowanej tablicy danych wejściowych.
- Procedura **HEAPSORT**, działająca w czasie  $O(n \lg n)$ , sortuje tablicę w miejscu.
- Procedury **MAX-HEAP-INSERT**, **MAX-HEAP-EXTRACT-MAX**, **MAX-HEAP-INCREASE-KEY** i **MAX-HEAP-MAXIMUM**, działające w czasie  $O(\lg n)$ , pozwalają na użycie kopca jako kolejki priorytetowej. Działają one w czasie  $O(\lg n)$ , plus czas potrzebny na powiązanie obiektów wstawianych do kolejki priorytetowej z indeksami w kopcu.

### Zadania

#### 6.1-1

Podaj największą i najmniejszą możliwą liczbę elementów w kopcu o wysokości  $h$ .

#### 6.1-2

Pokaż, że  $n$ -elementowy kopiec ma wysokość  $\lfloor \lg n \rfloor$ .

#### 6.1-3

Pokaż, że największy element w dowolnym poddrzewie kopca typu max znajduje się w korzeniu tego poddrzewa.

#### 6.1-4

Gdzie w kopcu typu max można znaleźć element najmniejszy, przy założeniu że wszystkie elementy są różne?

#### 6.1-5

Na którym poziomie w kopcu typu max może się znajdować  $k$ -ty najmniejszy element, dla  $2 \leq k \leq \lfloor n/2 \rfloor$ , przy założeniu że wszystkie elementy są różne?

#### 6.1-6

Czy tablica posortowana jest kopcem typu min?

#### 6.1-7

Czy tablica o zawartości  $(33, 19, 20, 15, 13, 10, 2, 13, 16, 12)$  jest kopcem typu max?

## 6.1-8

Pokaż, że w tablicowej reprezentacji kopca  $n$ -elementowego liście to węzły o indeksach  $[n/2] + 1, [n/2] + 2, \dots, n$ .

## 6.2 Przywracanie własności kopca

**MAX-HEAPIFY** jest procedurą służącą do utrzymywania własności kopca typu max. Jej danymi wejściowymi są: tablica  $A$  oraz indeks  $i$  w tej tablicy. Przy wywołaniu **MAX-HEAPIFY** zakłada się, że drzewa binarne o korzeniach w  $\text{LEFT}(i)$  i  $\text{RIGHT}(i)$  są kopczami typu max, ale że element  $A[i]$  może być mniejszy od swoich synów, przez co narusza własność kopca typu max. Zadaniem procedury **MAX-HEAPIFY** jest spowodowanie, żeby wartość  $A[i]$  „spłynęła” w dół drzewa tak, by poddrzewo zaczepione w węźle  $i$  stało się kopcem typu max.

**MAX-HEAPIFY( $A, i$ )**

```

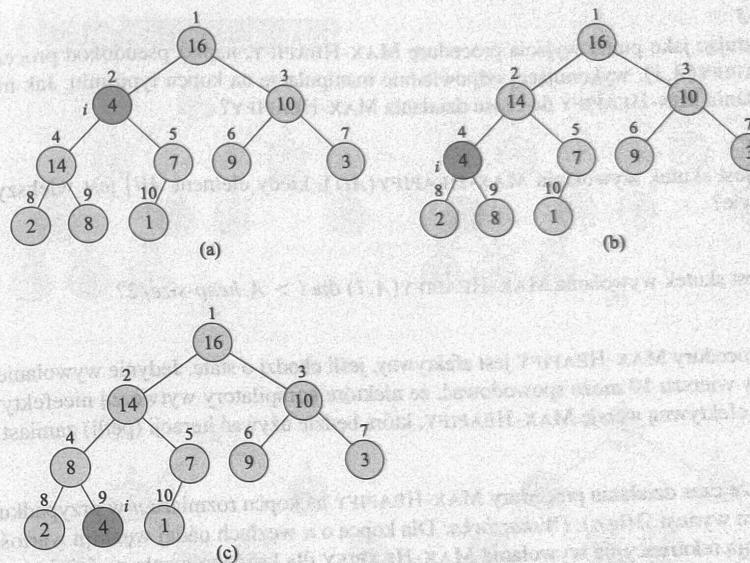
1    $l = \text{LEFT}(i)$ 
2    $r = \text{RIGHT}(i)$ 
3   if  $l \leq A.\text{heap-size}$  i  $A[l] > A[i]$ 
4      $largest = l$ 
5   else  $largest = i$ 
6   if  $r \leq A.\text{heap-size}$  i  $A[r] > A[largest]$ 
7      $largest = r$ 
8   if  $largest \neq i$ 
9     zamień  $A[i] \leftrightarrow A[largest]$ 
10    MAX-HEAPIFY( $A, largest$ )

```

Rysunek 6.2 ilustruje działanie procedury **MAX-HEAPIFY**. W każdym kroku jest wybierany największy z elementów  $A[i], A[\text{LEFT}(i)]$  i  $A[\text{RIGHT}(i)]$ , a jego indeks jest zachowywany w zmiennej  $largest$ . Jeśli największy jest  $A[i]$ , to poddrzewo zaczepione w  $i$  jest kopcem typu max i procedura kończy działanie. W przeciwnym razie jeden z synów jest największym elementem i następuje zamiana elementów na pozycjach  $i$  oraz  $largest$ , co powoduje, że węzeł  $i$  oraz jego synowie spełniają własność kopca typu max. Węzeł  $largest$  zmniejszył jednak właśnie swoją wartość i dlatego poddrzewo zaczepione w  $largest$  może nie spełniać własności kopca typu max. Z tego powodu procedurę **MAX-HEAPIFY** trzeba wywołać rekurencyjnie na tym poddrzewie.

W celu przeprowadzenia analizy **MAX-HEAPIFY**, przez  $T(n)$  oznaczmy pesymistyczny czas działania tej procedury na poddrzewie rozmiaru co najwyżej  $n$ . Dla poddrzewa zaczepionego w danym węźle  $i$  ten czas to  $\Theta(1)$  na poprawienie zależności między  $A[i], A[\text{LEFT}(i)]$  i  $A[\text{RIGHT}(i)]$  plus czas potrzebny na rekurencyjne wywołanie **MAX-HEAPIFY** na poddrzewie zaczepionym w jednym z synów węzła  $i$  (o ile dochodzi do wywołania rekurencyjnego). Każde z poddrzew synów węzła ma rozmiar najwyżej  $2n/3$  (patrz zad. 6.2-2), czas działania **MAX-HEAPIFY** można zatem opisać rekurencją

$$T(n) \leq T(2n/3) + \Theta(1).$$



Rysunek 6.2 Działanie procedury **MAX-HEAPIFY( $A, 2$ )**, gdzie  $A.\text{heap-size} = 10$ . Węzeł, który potencjalnie narusza własność kopca typu max, ma kolor ciemnoszary. (a) Początkowa konfiguracja kopca, w której wartość  $A[2]$  w węźle  $i = 2$  narusza własność kopca typu max, gdyż nie jest on większy od obu swoich synów. Właściwość kopca typu max jest przywracana węźłowi 2 w (b) przez zamianę  $A[2]$  z  $A[4]$ , co narusza własność kopca typu max w węźle 4. Rekurencyjne wywołanie **MAX-HEAPIFY( $A, 4$ )** ustawia  $i = 4$ . Po zamianie  $A[4]$  z  $A[9]$ , co jest pokazane w (c), węzeł 4 jest poprawiony, a rekurencyjne wywołanie **MAX-HEAPIFY( $A, 9$ )** nie zmienia więcej struktury danych.

Rozwiązaniem powyższej zależności jest, zgodnie z przypadkiem 2 twierdzenia o rekurencji uniwersalnej (twierdzenie 4.1 na stronie 96),  $T(n) = O(\lg n)$ . Inaczej mówiąc, czas działania **MAX-HEAPIFY** na węźle o wysokości  $h$  wynosi  $O(h)$ .

## Zadania

## 6.2-1

Zilustruj (podobnie jak na rys. 6.2) działanie procedury **MAX-HEAPIFY( $A, 3$ )** dla tablicy  $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$ .

## 6.2-2

Pokaż, że w poddrzewie każdego z synów korzenia kopca o  $n$  węzłach jest co najwyżej  $2n/3$  węzłów. Jaka jest najmniejsza wartość stałej  $\alpha$ , takiej że każde poddrzewo zawiera co najwyżej  $\alpha n$  węzłów? Jak to wpływa na rekurencję (6.1) i jej rozwiązanie?

## 6.2-3

Traktując jako punkt wyjścia procedurę MAX-HEAPIFY, napisz pseudokod procedury MIN-HEAPIFY( $A, i$ ), wykonującej odpowiednie manipulacje na kopcu typu min. Jak ma się czas działania MIN-HEAPIFY do czasu działania MAX-HEAPIFY?

## 6.2-4

Jaki jest skutek wywołania MAX-HEAPIFY( $A, i$ ), kiedy element  $A[i]$  jest większy niż jego synowie?

## 6.2-5

Jaki jest skutek wywołania MAX-HEAPIFY( $A, i$ ) dla  $i > A.\text{heap-size}/2$ ?

## 6.2-6

Kod procedury MAX-HEAPIFY jest efektywny, jeśli chodzi o stałe. Jedynie wywołanie rekurencyjne w wierszu 10 może spowodować, że niektóre kompilatory wytworzą nieefektywny kod. Napisz efektywną wersję MAX-HEAPIFY, która będzie używać iteracji (pętli) zamiast rekursji.

## 6.2-7

Wykaż, że czas działania procedury MAX-HEAPIFY na kopcu rozmiaru  $n$  w przypadku pesymistycznym wynosi  $\Omega(\lg n)$ . (Wskazówka: Dla kopca o  $n$  węzłach nadaj węzłom wartości, które spowodują rekurencyjne wywołania MAX-HEAPIFY dla każdego węzła na ścieżce prostej od korzenia do liścia).

## 6.3 Budowanie kopca

Procedura BUILD-MAX-HEAP przekształca tablicę  $A[1 : n]$  w kopiec typu max, wywołując wielokrotnie MAX-HEAPIFY w sposób wstępujący (ang. bottom-up). W myśl zadania 6.1-8 wszystkie elementy podtablicy  $A[(\lfloor n/2 \rfloor + 1) : n]$  są liścimi drzewa, zatem każdy z nich jest już 1-elementowym kopcem. Procedura BUILD-MAX-HEAP przechodzi przez pozostałe węzły drzewa i wywołuje w każdym z nich MAX-HEAPIFY. Na rysunku 6.3 pokazany jest przykład działania procedury BUILD-MAX-HEAP.

**BUILD-MAX-HEAP( $A, n$ )**

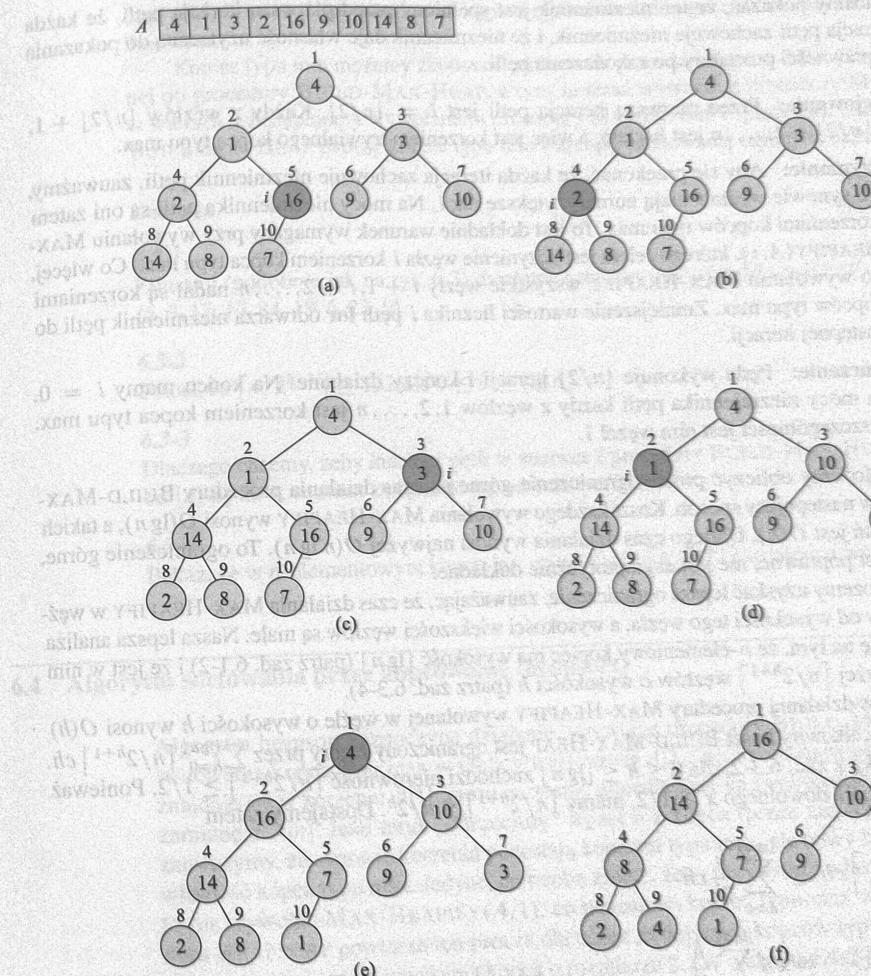
```

1    $A.\text{heap-size} = n$ 
2   for  $i = \lfloor n/2 \rfloor$  downto 1
3     MAX-HEAPIFY( $A, i$ )

```

Aby wykazać, że procedura BUILD-MAX-HEAP działa poprawnie, skorzystamy z następującego niezmiennika pętli:

Na początku każdej iteracji pętli **for** w wierszach 2–3 każdy węzeł  $i + 1, i + 2, \dots, n$  jest korzeniem kopca typu max.



**Rysunek 6.3** Działanie procedury BUILD-MAX-HEAP, pokazujące strukturę danych przed wywołaniem MAX-HEAPIFY w wierszu 3. Węzeł odpowiadający indeksowi  $i$  ma kolor ciemnoszary. (a) 10-elementowa tablica wejściowa A i reprezentowane przez nią drzewo binarne. Indeks pętli  $i$  wskazuje na węzeł 5 przed wywołaniem MAX-HEAPIFY( $A, i$ ). (b) Struktura danych, która jest wynikiem wywołania. Indeks pętli  $i$  dla następnego wywołania MAX-HEAPIFY( $A, i$ ). (c)–(e) Następne iteracje pętli for w BUILD-MAX-HEAP. Zauważmy, że kiedy MAX-HEAPIFY wskazuje na węzeł 4. (f) Kopia typu max po zakończeniu działania BUILD-MAX-HEAP.

Musimy pokazać, że ten niezmiennik jest spełniony przed pierwszą iteracją pętli, iż każda iteracja pętli zachowuje niezmiennik, iż niezmiennik daje własność użyteczną do pokazania poprawności procedury po zakończeniu pętli.

**Inicjowanie:** Przed pierwszą iteracją pętli jest  $i = \lfloor n/2 \rfloor$ . Każdy z węzłów  $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$  jest liściem, a więc jest korzeniem trywialnego kopca typu max.

**Utrzymanie:** Aby się przekonać, że każda iteracja zachowuje niezmiennik pętli, zauważmy, że synowie węzła  $i$  mają numery większe niż  $i$ . Na mocy niezmiennika pętli są oni zatem korzeniami kopców typu max. To jest dokładnie warunek wymagany przy wywołaniu  $\text{MAX-HEAPIFY}(A, i)$ , którego celem jest uczynienie węzła  $i$  korzeniem kopca typu max. Co więcej, po wywołaniu  $\text{MAX-HEAPIFY}$  wszystkie węzły  $i + 1, i + 2, \dots, n$  nadal są korzeniami kopców typu max. Zmniejszenie wartości licznika  $i$  pętli **for** odtwarza niezmiennik pętli do następnej iteracji.

**Zakończenie:** Pętla wykonuje  $\lfloor n/2 \rfloor$  iteracji i kończy działanie. Na końcu mamy  $i = 0$ . Na mocy niezmiennika pętli każdy z węzłów  $1, 2, \dots, n$  jest korzeniem kopca typu max. W szczególności jest nim węzeł 1.

Możemy obliczyć proste ograniczenie górne na czas działania procedury  $\text{BUILD-MAX-HEAP}$  w następujący sposób. Koszt każdego wywołania  $\text{MAX-HEAPIFY}$  wynosi  $O(\lg n)$ , a takich wywołań jest  $O(n)$ . Dlatego czas działania wynosi najwyżej  $O(n \lg n)$ . To ograniczenie górne, choć jest poprawne, nie jest asymptotycznie dokładne.

Możemy uzyskać lepsze ograniczenie, zauważając, że czas działania  $\text{MAX-HEAPIFY}$  węzła zależy od wysokości tego węzła, a wysokości większości węzłów są małe. Nasza lepsza analiza opiera się na tym, że  $n$ -elementowy kopiec ma wysokość  $\lfloor \lg n \rfloor$  (patrz zad. 6.1-2) i że jest w nim co najwyżej  $\lceil n/2^{h+1} \rceil$  węzłów o wysokości  $h$  (patrz zad. 6.3-4).

Czas działania procedury  $\text{MAX-HEAPIFY}$  wywołanej w węźle o wysokości  $h$  wynosi  $O(h)$  i dlatego całkowity koszt  $\text{BUILD-MAX-HEAP}$  jest ograniczony z góry przez  $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil ch$ . Jak wynika z zad. 6.3-2, dla  $0 \leq h \leq \lfloor \lg n \rfloor$  zachodzi nierówność  $\lceil n/2^{h+1} \rceil \geq 1/2$ . Ponieważ  $\lceil x \rceil \leq 2x$  dla dowolnego  $x \geq 1/2$ , mamy  $\lceil n/2^{h+1} \rceil \leq n/2^h$ . Dostajemy zatem

$$\begin{aligned} \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil ch &\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\ &= cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\ &\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &\leq cn \cdot \frac{1/2}{(1 - 1/2)^2} \quad (\text{z równania (A.11) na str. 1072 dla } x = 1/2) \\ &= O(n). \end{aligned}$$

Stąd widać, że możemy zbudować kopiec typu max z nieuporządkowanej tablicy w czasie liniowym.

Kopiec typu min możemy zbudować za pomocą procedury  $\text{BUILD-MIN-HEAP}$ , analogicznej do procedury  $\text{BUILD-MAX-HEAP}$ , z tym że teraz wywołanie procedury  $\text{MAX-HEAPIFY}$  w wierszu 3 zastępujemy wywołaniem procedury  $\text{MIN-HEAPIFY}$  (patrz zad. 6.2-3). Procedura  $\text{BUILD-MIN-HEAP}$  tworzy kopiec typu min z nieuporządkowanej tablicy w czasie liniowym.

## Zadania

### 6.3-1

Zilustruj (podobnie jak na rys. 6.3) działanie procedury  $\text{BUILD-MAX-HEAP}$  dla tablicy  $A = (5, 3, 17, 10, 84, 19, 6, 22, 9)$ .

### 6.3-2

Pokaż, że  $\lceil n/2^{h+1} \rceil \geq 1/2$  dla  $0 \leq h \leq \lfloor \lg n \rfloor$ .

### 6.3-3

Dlaczego chcemy, żeby indeks  $i$  pętli w wierszu 2 procedury  $\text{BUILD-MAX-HEAP}$  zmniejszał się od  $\lfloor n/2 \rfloor$  do 1, zamiast zwiększać się od 1 do  $\lfloor n/2 \rfloor$ ?

### 6.3-4

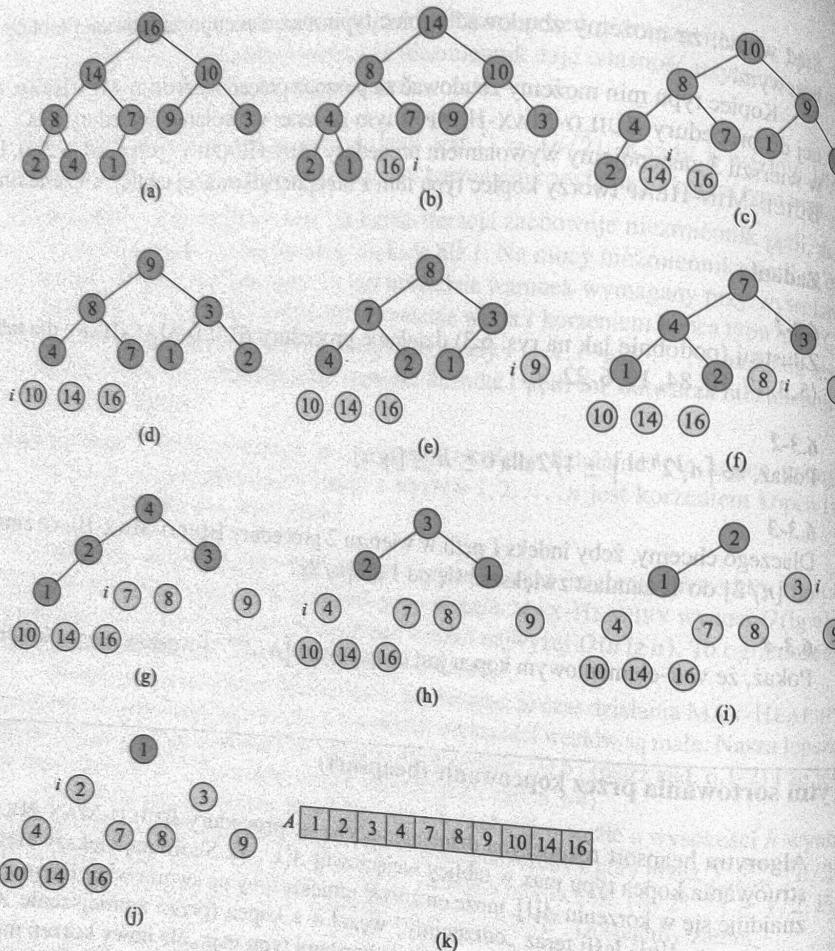
Pokaż, że w  $n$ -elementowym kopcu jest co najwyżej  $\lceil n/2^{h+1} \rceil$  węzłów o wysokości  $h$ .

## 6.4 Algorytm sortowania przez kopcowanie (heapsort)

Algorytm heapsort rozpoczętu działania, używając procedury  $\text{BUILD-MAX-HEAP}$  do skonstruowania kopca typu max w tablicy wejściowej  $A[1 : n]$ . Skoro największy element tablicy znajduje się w korzeniu  $A[1]$ , może on zostać umieszczony na swoim właściwym miejscu przez zamianę z  $A[n]$ . Jeśli teraz „odrzucimy” węzeł  $n$  z kopca (przez zmniejszenie  $A.\text{heap-size}$ ), zauważymy, że synowie korzenia pozostają kopczami typu max, ale nowy korzeń może naruszać własność kopca typu max. Jedyne, co trzeba zrobić, żeby przywrócić własność kopca typu max, to raz wywołać  $\text{MAX-HEAPIFY}(A, 1)$ , co pozostawi kopiec typu max w  $A[1 : n - 1]$ . Procedura  $\text{HEAPSORT}$  powtarza ten proces dla coraz mniejszych kopków typu max, począwszy od rozmiaru  $n - 1$ , aż do uzyskania kopca o rozmiarze 2. (W zadaniu 6.4-2 jest podane precyzyjne sformułowanie niezmiennika pętli).

### HEAPSORT( $A, n$ )

- 1  $\text{BUILD-MAX-HEAP}(A, n)$
- 2 **for**  $i = n$  **downto** 2
- 3     zamień  $A[1] \leftrightarrow A[i]$
- 4      $A.\text{heap-size} = A.\text{heap-size} - 1$
- 5      $\text{MAX-HEAPIFY}(A, 1)$



Na rysunku 6.4 widać przykład działania algorytmu heapsort po początkowym zbudowaniu kopca typu max w wierszu 1. Pokazany jest kopiec typu max przed pierwszą iteracją pętli for wierszach 2-5 i po każdej iteracji.

Czas działania procedury HEAPSORT wynosi  $O(n \lg n)$ , ponieważ wywołanie BUILD-MAX-HEAP zajmuje czas  $O(n)$ , a każde z  $n - 1$  wywołań MAX-HEAPIFY zajmuje czas  $O(\lg n)$ .

**Zadania****6.4-1**

Zilustruj (podobnie jak na rys. 6.4) działanie procedury HEAPSORT dla tablicy  $A = \{5, 13, 2, 25, 7, 17, 20, 8, 4\}$ .

**6.4-2**

Uzasadnij poprawność procedury HEAPSORT, korzystając z następującego niezmiennika pętli:

Na początku każdej iteracji pętli **for** w wierszach 2–5 podtablica  $A[1 : i]$  jest kopcem typu max zawierającym  $i$  najmniejszych elementów z  $A[1 : n]$ , a podtablica  $A[i + 1 : n]$  zawiera posortowanych  $n - i$  największych elementów z  $A[1 : n]$ .

**6.4-3**

Jaki jest czas działania procedury HEAPSORT dla tablicy  $A$  o długości  $n$ , która jest już posortowana rosnąco (malejaco)?

**6.4-4**

Pokaż, że czas działania procedury HEAPSORT w przypadku pesymistycznym wynosi  $\Omega(n \lg n)$ .

**★ 6.4-5**

Pokaż, że jeśli wszystkie elementy są różne, to czas działania procedury HEAPSORT w przypadku optymistycznym wynosi  $\Omega(n \lg n)$ .

**6.5 Kolejki priorytetowe**

W rozdziale 8 zobaczymy, że dowolny oparty na porównaniach algorytm sortowania wymaga użycia  $\Omega(n \lg n)$  porównań, a zatem działa w czasie  $\Omega(n \lg n)$ . Heapsort jest więc asymptycznie optymalny w klasie algorytmów sortowania opartych na porównaniach. Jednak dobra implementacja algorytmu quicksort, taka jak opisana w rozdz. 7, jest na ogół lepsza w praktyce. Pomimo to kopiec jako struktura danych ma wiele zastosowań. W tym podrozdziale opisujemy jedno z najbardziej popularnych zastosowań kopca: jako efektywnej kolejki priorytetowej. Tak jak kopce, kolejki priorytetowe wywiążą dwóch typów: min i max. My skoncentrujemy się na implementacji kolejek priorytetowych typu max, opartej z kolei na kopcach typu max. Napisanie procedur obsługi kolejek priorytetowych typu min pozostawiamy Czytelnikom (patrz zad. 6.5-3).

**Kolejka priorytetowa** to struktura danych służąca do reprezentowania zbioru  $S$  elementów, z których każdy ma przyporządkowaną wartość zwaną **kluczem**. Na **kolejce priorytetowej typu max** można wykonać następujące operacje:

- **INSERT( $S, x$ )** wstawia element  $x$  z kluczem  $k$  do zbioru  $S$ . Tę operację można zapisać jako  $S = S \cup \{x\}$ .
- **MAXIMUM( $S$ )** daje w wyniku element zbioru  $S$  o największym kluczum.

- EXTRACT-MAX( $S$ ) usuwa i daje w wyniku element zbioru  $S$  o największym kluczu.
- INCREASE-KEY( $S, x, k$ ) zmienia wartość klucza elementu  $x$  na nową wartość  $k$ , o której zakładając, że jest nie mniejsza niż aktualna wartość klucza  $x$ .

Jednym z zastosowań kolejki priorytetowej typu max jest szeregowanie zadań na wspólnym komputerze. W kolejce priorytetowej typu max są przechowywane zadania, które mają być wykonane, i ich priorytety względem siebie. Kiedy zadanie się kończy lub zostaje przerwane, zadanie o największym priorytecie jest wybierane spośród zadań czekających za pomocą operacji EXTRACT-MAX. Nowe zadanie może zostać dodane do kolejki w dowolnej chwili za pomocą operacji INSERT.

Na kolejce priorytetowej typu min można wykonywać operacje INSERT, MINIMUM, EXTRACT-MIN i DECREASE-KEY. Takiej kolejki można użyć jako symulatora zdarzeń. Elementami kolejki są zdarzenia, które należy symulować, każde z przyporządkowanym czasem wystąpienia, który służy jako klucz. Zdarzenia muszą być symulowane w kolejności ich zajścia, ponieważ symulacja zdarzenia może spowodować, że inne zdarzenia będą symulowane w przyszłości. Program symulacyjny korzysta w każdym kroku z operacji EXTRACT-MIN do wybierania kolejnego zdarzenia, które będzie symulował. Kiedy generowane są nowe zdarzenia, zostają one dodane do kolejki za pomocą operacji INSERT. Inne zastosowania kolejek priorytetowych typu min, ze szczególnym podkreśleniem operacji DECREASE-KEY, przedstawimy w rozdz. 21 i 22.

Kiedy w konkretnym zastosowaniu implementujemy kolejkę priorytetową za pomocą kopca, elementy kolejki priorytetowej odpowiadają pewnym obiektom. Każdy obiekt zawiera klucz. Implementując kolejkę priorytetową za pomocą kopca, musimy umieć określić, który obiekt odpowiada danemu elementowi kolejki priorytetowej, lub odwrotnie. Ponieważ elementy kopca są przechowywane w tablicy, potrzebujemy jakieś metody odwzorowania między obiektami programu a indeksami w tablicy.

Sposobem powiązania obiektów programu z elementami kopca są *uchwyty*, czyli dodatkowe informacje przechowywane w obiektach i w elementach kopca, które umożliwiają to powiązanie. Uchwyty są często implementowane tak, by otaczający je kod nie mógł w nie ingerować, co pozwala zachować barierę abstrakcji między programem a kolejką priorytetową. Uchwyt w obiekcie programu mógłby na przykład zawierać odpowiedni indeks w tablicy reprezentującej kopiec. Ponieważ dostęp do tego indeksu ma wyłącznie kod implementujący operacje kolejki priorytetowej, jest on całkowicie ukryty przed kodem programu. Elementy kopca zmieniają swoje położenie w tablicy podczas operacji kopcowych, zatem w implementacji kolejki priorytetowej przy zmianie położenia elementów kopca trzeba również uaktualniać indeksy z tablicy w odpowiednich uchwytech. W drugą stronę, każdy element w kopcu może zawierać wskaźnik do odpowiedniego obiektu programu, ale kopiec traktuje ten wskaźnik tylko jako nieprzejrzysty dla siebie uchwyt, a dopiero sam program odwzorowuje go na konkretny obiekt. Zazwyczaj pesymistyczny narzut czasowy związany z obsługą uchwytów to  $O(1)$  na każdy dostęp.

Podejście stanowiące alternatywę dla umieszczania uchwytów w obiektach programu polega na przechowywaniu wraz z kolejką priorytetową odwzorowania obiektów programu na indeksy w tablicy reprezentującej kopiec. Zaletą takiego postępowania jest to, że powiązanie jest w całości zawarte w kolejce priorytetowej, dzięki czemu nie trzeba niczego dodawać do obiektów programu.

Wadą jest dodatkowy koszt związany ze stworzeniem i utrzymywaniem takiego odwzorowania. Jedną z możliwości realizacji odwzorowania jest tablica z haszowaniem (patrz rozdz. 11)<sup>1</sup>. W tablicy z haszowaniem dodatkowy oczekiwany czas odwzorowania obiektu na indeks w tablicy to tylko  $O(1)$ , chociaż w przypadku pesymistycznym może to być nawet  $\Theta(n)$ .

Zobaczmy, jak można zaimplementować operacje kolejki priorytetowej typu max za pomocą kopca typu max. W poprzednich podrozdziałach traktowaliśmy elementy tablicy jako klucze do posortowania, niejawnie zakładając, że wszelkie dane dodatkowe są przemieszczane wraz ze stwarzyszonymi z nimi kluczami. Kiedy kopiec stanowi implementację kolejki priorytetowej, zamiast tego traktujemy każdy element tablicy jako wskaźnik do obiektu w kolejce priorytetowej, czyli obiekt jest odpowiednikiem danych dodatkowych podczas sortowania. Dalej, zakładamy, że każdy taki obiekt ma atrybut *key*, który określa położenie obiektu w kopcu. Jeśli kopiec jest reprezentowany przez tablicę  $A$ , odwołanie do tego atrybutu jest postaci  $A[i].key$ . Procedura MAX-HEAP-MAXIMUM implementuje operację MAXIMUM w czasie  $\Theta(1)$ , a procedura MAX-HEAP-EXTRACT-MAX stanowi implementację operacji EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX jest podobna do treści pętli for (wiersze 3–5) procedury HEAPSORT. Niejawnie zakładamy, że MAX-HEAPIFY porównuje obiekty w kolejce priorytetowej na podstawie ich atrybutów *key*. Zakładamy również, że kiedy w MAX-HEAPIFY są zamieniane elementy tablicy, następuje zamiana wskaźników, a także aktualnione zostaje odwzorowanie między obiektami a indeksami w tablicy. Czas działania procedury MAX-HEAP-EXTRACT-MAX wynosi  $O(\lg n)$ , ponieważ wykonuje ona tylko stałą ilość pracy oprócz czasu  $O(\lg n)$  zużywanego przez procedurę MAX-HEAPIFY oraz narzutu związanego z koniecznością odwzorowania obiektów kolejki priorytetowej na indeksy w tablicy.

#### MAX-HEAP-MAXIMUM( $A$ )

```

1 if  $A.\text{heap-size} < 1$ 
2   error „kopiec pusty”
3 return  $A[1]$ 
```

#### MAX-HEAP-EXTRACT-MAX( $A$ )

```

1 max = MAX-HEAP-MAXIMUM( $A$ )
2  $A[1] = A[A.\text{heap-size}]$ 
3  $A.\text{heap-size} = A.\text{heap-size} - 1$ 
4 MAX-HEAPIFY( $A, 1$ )
5 return max
```

Procedura MAX-HEAP-INCREASE-KEY stanowi implementację operacji INCREASE-KEY. Sprawdza ona, czy nowa wartość  $k$  nie spowoduje zmniejszenia klucza w obiekcie  $x$ , i jeśli tak nie jest, to nadaje obiektowi  $x$  nową wartość klucza  $k$ . Następnie znajdowany jest indeks  $i$  w tablicy odpowiadający obiektowi  $x$ , tzn. taki że  $A[i] = x$ . Ponieważ zwiększenie klucza w  $A[i]$

<sup>1</sup> W Pythonie słowniki są zaimplementowane przy pomocy tablic z haszowaniem.

mogło naruszyć własność kopca typu max, zatem w sposób przypominający pętlę wstawiania (wiersze 5–7) w procedurze **INSERTION-SORT** na str. 48 ścieżkę prostą od tego węzła w stronę korzenia w celu znalezienia właściwego miejsca dla nowego, zwiększonego klucza. Podczas tego przechodzenia ścieżki za każdym razem porównujemy element z jego ojcem, zamieniając klucz i kontynuując marsz ku korzeniowi, jeśli klucz w elemencie jest większy, zatrzymując się zaś, kiedy klucz w elemencie jest mniejszy niż w ojcu, ponieważ wtedy spełniona jest już własność kopca typu max. (Dokładne sformułowanie niezmiennika petli jest podane w zad. 6.5–7). Podobnie jak w przypadku **MAX-HEAPIFY** użytego w kolejce priorytetowej, **MAX-HEAP-INCREASE-KEY** przy zamianie elementów tablicy aktualnia informacje pozwalające odwzorowywać obiekty na indeksy w tablicy. Na rysunku 6.5 jest przedstawiony przykład działania procedury **MAX-HEAP-INCREASE-KEY**. Oprócz narzutu związanego z odwzorowaniem obiektów z kolejki priorytetowej na indeksy w tablicy, czas działania **MAX-HEAP-INCREASE-KEY** na  $n$ -elementowym kopcu wynosi  $O(\lg n)$ , ponieważ ścieżka od węzła uaktualnianego w wierszu 3 do korzenia ma długość  $O(\lg n)$ .

**MAX-HEAP-INCREASE-KEY( $A, x, k$ )**

```

1 if  $k < x.key$ 
2   error „nowy klucz jest mniejszy niż klucz aktualny”
3  $x.key = k$ 
4 znajdź indeks  $i$  obiektu  $x$  w tablicy  $A$ 
5 while  $i > 1$  i  $A[\text{PARENT}(i)].key < A[i].key$ 
6   zamień  $A[i] \leftrightarrow A[\text{PARENT}(i)]$ , uaktualniając informację
      odwzorowującą obiekty kolejki priorytetowej na indeksy w tablicy
7    $i = \text{PARENT}(i)$ 

```

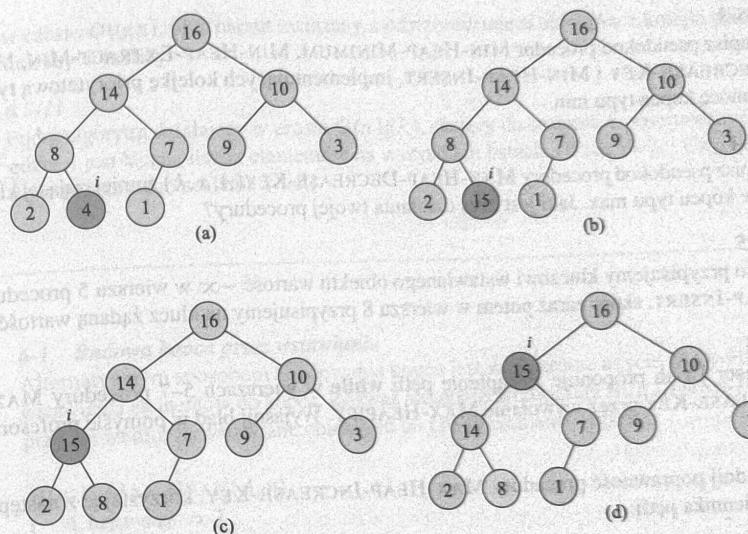
**MAX-HEAP-INSERT( $A, x, n$ )**

```

1 if  $A.\text{heap-size} == n$ 
2   error „przepelenie kopca”
3  $A.\text{heap-size} = A.\text{heap-size} + 1$ 
4  $k = x.key$ 
5  $x.key = -\infty$ 
6  $A[A.\text{heap-size}] = x$ 
7 odwzoruj  $x$  na indeks  $\text{heap-size}$  w tablicy
8 MAX-HEAP-INCREASE-KEY( $A, x, k$ )

```

Procedura **MAX-HEAP-INSERT** stanowi implementację operacji **INSERT**. Jej argumentami są: tablica  $A$  reprezentująca kopiec typu max, nowy obiekt  $x$ , który ma być wstawiony do tego kopca oraz rozmiar  $n$  tablicy  $A$ . Na początku procedura sprawdza, czy w tablicy jest miejsce na nowy element. Następnie powiększa ona kopiec przez dodanie do drzewa nowego liścia o kluczu  $-\infty$ , po czym wywołuje **HEAP-INCREASE-KEY** w celu wpisania do tego nowego węzła właściwej wartości klucza i odtworzenia własności kopca typu max. Czas działania **MAX-HEAP-INSERT** na  $n$ -elementowym kopcu to  $O(\lg n)$ , plus narzut związany z odwzorowaniem obiektów z kolejki priorytetowej na indeksy.



Rysunek 6.5 Działanie procedury **MAX-HEAP-INCREASE-KEY**. Na rysunku pokazane są tylko klucze obiektów z kolejki priorytetowej. Węzeł indeksowany przez  $i$  w każdej iteracji ma kolor ciemnoszary. (a) Kopiec typu max z rys. 6.4(a);  $i$  indeksuje węzeł, którego klucz ma wzrosnąć. (b) Klucz w tym węźle został zwiększy do wartości 15. (c) Po pierwszej iteracji petli while w wierszach 5–7 klucz w węźle  $i$  i jego ojcu zostały zmienione, a indeks  $i$  przesunięty w górę do ojca. (d) Kopiec typu max po kolejnej iteracji petli while. W tym momencie  $A[\text{PARENT}(i)] \geq A[i]$ . Własność kopca typu max jest teraz spełniona i procedura kończy działanie

Podsumowując, kopiec umożliwia wykonywanie wszystkich operacji kolejki priorytetowej na zbiorze  $n$  elementów w czasie  $O(\lg n)$ , plus narzut związany z odwzorowaniem obiektów z kolejki priorytetowej na indeksy w tablicy.

**Zadania**

**6.5-1** Założmy, że obiekty w kolejce priorytetowej to same klucze. Zilustruj działanie procedury **MAX-HEAP-EXTRACT-MAX** na kopcu  $A = \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\}$ .

**6.5-2** Założmy, że obiekty w kolejce priorytetowej to same klucze. Zilustruj działanie procedury **MAX-HEAP-INSERT( $A, 10, 15$ )** na kopcu  $A = \{15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1\}$ .

**6.5-3**

Napisz pseudokod procedur MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY i MIN-HEAP-INSERT, implementujących kolejkę priorytetową typu min za pomocą kopca typu min.

**6.5-4**

Napisz pseudokod procedury MAX-HEAP-DECREASE-KEY( $A, x, k$ ) zmniejszającej klucz obiektu w kopcu typu max. Jaki jest czas działania twojej procedury?

**6.5-5**

Po co przypisujemy kluczowi wstawianego obiektu wartość  $-\infty$  w wierszu 5 procedury MAX-HEAP-INSERT, skoro zaraz potem w wierszu 8 przypisujemy na klucz żądaną wartość?

**6.5-6**

Profesor Uriah proponuje zastąpienie pętli **while** w wierszach 5–7 procedury MAX-HEAP-INCREASE-KEY przez wywołanie MAX-HEAPIFY. Wyjaśnij błąd w pomyśle profesora.

**6.5-7**

Uzasadnij poprawność procedury MAX-HEAP-INCREASE-KEY, korzystając z następującego niezmiennika pętli:

Na początku każdej iteracji pętli **while** w wierszach 5–7:

1. Jeśli obydwa węzły PARENT( $i$ ) i LEFT( $i$ ) istnieją, to  $A[\text{PARENT}(i)].key \geq A[\text{LEFT}(i)].key$
2. Jeśli obydwa węzły PARENT( $i$ ) i RIGHT( $i$ ) istnieją, to  $A[\text{PARENT}(i)].key \geq A[\text{RIGHT}(i)].key$ .
3. Podtablica  $A[1 : A.\text{heap-size}]$  spełnia własność kopca typu max z co najwyżej jednym wyjątkiem:  $A[i].key$  może być większe niż  $A[\text{PARENT}(i)].key$ .

Możesz założyć, że w chwili wywołania MAX-HEAP-INCREASE-KEY podtablica  $A[1 : A.\text{heap-size}]$  spełnia własność kopca typu max.

**6.5-8**

Każda operacja zamiany w wierszu 6 procedury MAX-HEAP-INCREASE-KEY wymaga zazwyczaj trzech przypisań, nie licząc aktualniania odwzorowania obiektów na indeksy w tablicy. Pokaż, jak wykorzystać pomysł z wewnętrznej pętli w procedurze INSERTION-SORT do zmniejszenia liczby przypisań z trzech do zaledwie jednego.

**6.5-9**

Pokaż, jak za pomocą kolejki priorytetowej zaimplementować zwykłą kolejkę (FIFO) oraz stos. (Kolejki FIFO i stosy są zdefiniowane w podrozdz. 10.1.3).

**6.5-10**

Operacja MAX-HEAP-DELETE( $A, x$ ) polega na usunięciu obiektu  $x$  z kopca typu max. Podaj implementację MAX-HEAP-DELETE, która działa na  $n$ -elementowym kopcu typu max.

w czasie  $O(\lg n)$ , plus narzut związany z odwzorowaniem obiektów z kolejki priorytetowej na indeksy w tablicy.

**6.5-11**

Podaj algorytm działający w czasie  $O(n \lg k)$ , służący do scalania  $k$  posortowanych list w jedną, gdzie  $n$  jest łączną liczbą elementów na wszystkich listach. (Wskazówka: Użyj kopca typu min do jednoczesnego scalania  $k$  list).

**Problemy****6-1 Budowa kopca przez wstawianie**

Alternatywnym sposobem zbudowania kopca jest wielokrotne użycie MAX-HEAP-INSERT do wstawiania elementów do kopca. Rozważ następującą procedurę BUILD-MAX-HEAP', działającą przy założeniu, że wstawiane obiekty to po prostu elementy kopca:

BUILD-MAX-HEAP'( $A, n$ )

- 1  $A.\text{heap-size} = 1$
- 2 **for**  $i = 2$  **to**  $n$
- 3     MAX-HEAP-INSERT( $A, A[i], n$ )

(a) Czy procedury BUILD-MAX-HEAP i BUILD-MAX-HEAP' zawsze tworzą taki sam kopiec, gdy zostaną uruchomione dla tej samej tablicy? Udowodnij, że tak jest, albo podaj kontrprzykład.

(b) Pokaż, że w przypadku pesymistycznym procedura BUILD-MAX-HEAP' wymaga czasu  $\Theta(n \lg n)$ , żeby zbudować  $n$ -elementowy kopiec.

**6-2 Analiza kopków rzędu  $d$** 

**Kopiec rzędu  $d$  ( $d$ -kopiec)** jest podobny do kopca binarnego, ale (z jednym możliwym wyjątkiem) węzły niebędące liśćmi mają po  $d$  synów zamiast po dwóch. We wszystkich punktach tego problemu należy założyć, że czas potrzebny do utrzymania odwzorowania między obiektami a elementami kopca to  $O(1)$  na każdą operację.

(a) Opisz, jak można reprezentować kopiec rzędu  $d$  w tablicy.

(b) Stosując notację  $\Theta$ , wyraź wysokość kopca rzędu  $d$  o  $n$  elementach za pomocą  $d$  i  $n$ .

(c) Podaj efektywną implementację operacji EXTRACT-MAX dla  $d$ -kopca typu max. Wyraź jej czas działania za pomocą  $d$  i  $n$ .

(d) Podaj efektywną implementację operacji INCREASE-KEY( $A, i, k$ ) dla  $d$ -kopca typu max. Wyraź jej czas działania za pomocą  $d$  i  $n$ .

- (e) Podaj efektywną implementację operacji **INSERT** dla  $d$ -kopca typu max. Wyraź jej czas działania za pomocą  $d$  i  $n$ .

### 6-3 Tablice Younga

**Tablica Younga** o wymiarach  $m \times n$  to macierz  $m \times n$ , której elementy w każdym wierszu są posortowane od strony lewej do prawej, a elementy w każdej kolumnie są posortowane od góry do dołu. Niektóre elementy w tablicy Younga mogą być równe  $\infty$ , co należy traktować jako brak elementu. Tak więc w tablicy Younga można przechowywać  $r \leq mn$  skończonych wartości.

- (a) Narysuj tablicę Younga  $4 \times 4$  zawierającą elementy  $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$ .
- (b) Uzasadnij, że tablica Younga  $Y$  o wymiarach  $m \times n$  jest pusta, jeśli  $Y[1, 1] = \infty$ . Uzasadnij, że tablica  $Y$  jest pełna (zawiera  $mn$  elementów), jeśli  $Y[m, n] < \infty$ .
- (c) Podaj działający w czasie  $O(m + n)$  algorytm implementujący operację **EXTRACT-MIN** na niepustej tablicy Younga  $m \times n$ . Twój algorytm powinien wykorzystywać rekurencyjną procedurę rozwiązującą problem  $m \times n$  przez rekurencyjne rozwiązywanie jednego z podproblemów:  $(m - 1) \times n$  albo  $m \times (n - 1)$ . (Wskazówka: Rozważ **MAX-HEAPIFY**). Wyjaśnij, dlaczego Twoja implementacja **EXTRACT-MIN** działa w czasie  $O(m + n)$ .
- (d) Pokaż, jak wstawić nowy element do niewypełnionej całkowicie tablicy Younga  $m \times n$  w czasie  $O(m + n)$ .
- (e) Pokaż, w jaki sposób – bez użycia żadnych innych metod sortowania – posortować  $n^2$  liczb przy użyciu tablicy Younga  $n \times n$  w czasie  $O(n^3)$ .
- (f) Podaj działający w czasie  $O(m + n)$  algorytm sprawdzania, czy dana liczba znajduje się w danej tablicy Younga  $m \times n$ .

### Uwagi do rozdziału

Algorytm heapsort został wynaleziony przez Williamsa [456], który również opisał implementację kolejki priorytetowej za pomocą kopca. Procedura **BUILD-MAX-HEAP** została zaproponowana przez Floyda [145]. Schaffer i Sedgewick [395] pokazali, że w przypadku optymistycznym heapsortu liczba przemieszczeń elementów w kopcu to w przybliżeniu  $(n/2) \lg n$  i że średnia liczba przemieszczeń to w przybliżeniu  $n \lg n$ .

Kopców typu min używamy do implementacji kolejek priorytetowych typu min w rozdz. 15, 21 i 22. Inne, bardziej skomplikowane struktury danych pozwalają uzyskać lepsze oszacowania kosztu niektórych operacji kolejki priorytetowej typu min. Fredman i Tarjan [156] zaprojektowali kopce Fibonacciego, które umożliwiają wykonywanie operacji **INSERT** i **DECREASE-KEY** w zamortyzowanym czasie  $O(1)$  (patrz rozdz. 16). Oznacza to, że średni pesymistyczny czas przypadający na jedną z tych operacji wynosi  $O(1)$ . Brodal, Lagogiannis i Tarjan

następnie „ścisłe” kopce Fibonacciego (ang. *strict Fibonacci heaps*), które osiągają takie oszacowania czasów operacji w sensie pesymistycznym. Przy założeniu, że klucze są różne i wybierane ze zbioru  $\{0, 1, \dots, n\}$  nieujemnych liczb całkowitych, drzewa van Emde Boasa [440, 441] umożliwiają wykonywanie operacji **INSERT**, **SEARCH**, **MINIMUM**, **MAXIMUM**, **PREDECESSOR** i **SUCCESSOR** w czasie  $O(\lg \lg n)$ . Przy założeniu, że dane są  $b$ -bitowymi liczbami całkowitymi, a pamięć komputera składa się z adresowalnych słów  $b$ -bitowych, Fredman i Willard [157] pokazali, jak zaimplementować operację **MINIMUM** działającą w czasie  $O(1)$  oraz operacje **INSERT** i **EXTRACT-MIN** działające w czasie  $O(\sqrt{\lg n})$ . Thorup [436] poprawił oszacowanie  $O(\sqrt{\lg n})$  do  $O(\lg \lg n)$ , stosując randomizowane haszowanie, ale z wykorzystaniem tylko liniowej pamięci.

Z ważnym szczególnym przypadkiem kolejek priorytetowych mamy do czynienia, gdy ciąg operacji **EXTRACT-MIN** jest **monotoniczny**, tzn. kiedy wartości zwarcane przez kolejne wywołania **EXTRACT-MIN** monotonicznie rosną. Taka sytuacja ma miejsce w wielu ważnych zastosowaniach, takich jak algorytm Dijkstry znajdowania najkrótszych ścieżek z jednym źródłem, omówiony w rozdz. 22, czy symulacja zdarzeń dyskretnych. W algorytmie Dijkstry szczególnie istotne jest efektywne zaimplementowanie operacji **DECREASE-KEY**. Dla przypadku monotonicznego, jeśli dane są liczbami całkowitymi z zakresu  $1, 2, \dots, C$ , Ahuja, Melhorn, Orlin i Tarjan [8] opisują implementację operacji **EXTRACT-MIN** i **INSERT** działającą w czasie zamortyzowanym  $O(\lg C)$  (więcej o analizie kosztu zamortyzowanego można przeczytać w rozdz. 16) oraz **DECREASE-KEY** działającą w czasie  $O(1)$  z wykorzystaniem struktury danych zwanej kopcem pozycyjnym (ang. *radix heap*). Oszacowanie  $O(\lg C)$  można polepszyć do  $O(\sqrt{\lg C})$  za pomocą kopców Fibonacciego w połączeniu z kopcami pozycyjnymi. To oszacowanie wzmacnili do  $O(\lg^{1/3+\epsilon} C)$  Cherkassky, Goldberg i Silverstein [90], którzy połączyli wielopoziomową strukturę kubelkową Denarda i Foxa [112] ze wspomnianym wyżej kopcem Thorupa. Raman [375] jeszcze poprawił ten wynik, uzyskując oszacowanie  $O(\min(\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n))$  dla dowolnego ustalonego  $\epsilon > 0$ .

Zaproponowano wiele innych wariantów kopków. Brodal [72] opracował przegląd tych wyników.