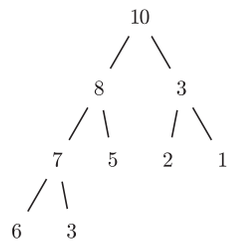


Rys. 1. $a = (10, 8, 3, 7, 5, 2, 1, 6, 3)$

Sortowanie przez kopcowanie

Tomasz KAZANA

W tym artykule zakładam, że Czytelnik choć trochę programował. W szczególności zna podstawy jakiegoś języka programowania, np. Pascala. Jeśli to podstawowe założenie jest spełnione, to – jestem o tym przekonany – mogę śmiało założyć, że jest mu znane również pojęcie tablicy. Tablica to bardzo wygodny sposób reprezentowania w programie ciągu zmiennych. Na potrzeby tego artykułu niektóre takie tablice (czyli reprezentowane przez nie ciągi) będziemy uważać za lepsze od innych. Wyróżnione tablice będziemy nazywać tablicami kopcowymi. Aby na taki tytuł zasłużyć, muszą spełniać następujący warunek: *elementy tablicy zapisane na kartce w kolejnych wierszach pełnego drzewa binarnego* (tytułowego kopca) *muszą zachować porządek starszeństwa*, tzn. zawsze rodzic musi mieć wartość większą bądź równą swoim dzieciom. Przykład takiej tablicy znajduje się na rysunku 1. Nietrudno zauważyć, że powyższy warunek na „kopcowatość" tablicy $a[1...n]$ daje się opisać algebraicznie: zawsze $a[i] \geq a[2i]$ oraz $a[i] \geq a[2i+1]$, o ile tylko indeksy mieszczą się w zakresie $1 \ldots n$.

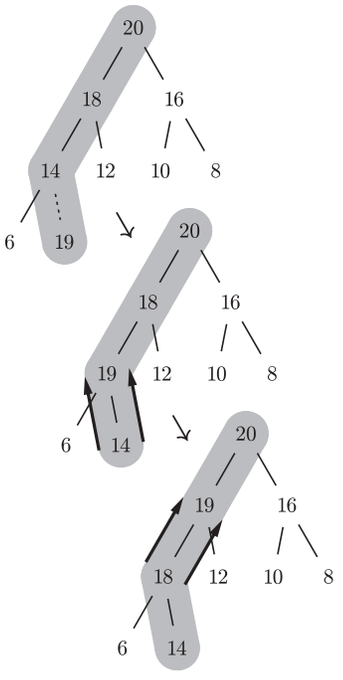
Dziećmi wierzchołka o indeksie i są te o indeksach $2i$ oraz $2i+1$. W drugą stronę: dla wierzchołka o indeksie j jego rodzic ma indeks $\lfloor j/2 \rfloor$ (*pascalowo* j div 2).

Przedstawiliśmy już głównego bohatera, teraz czas na fabułę. Chcemy sortować. To znaczy: poprzestawiać elementy danej (dowolnej, niekoniecznie kopcowej) tablicy tak, aby tworzyły ciąg niemalejący. Zadanie wykonamy w dwóch fazach. Najpierw pomieszamy elementy, aby utworzyły tablicę kopcową. Dopiero wówczas, w fazie drugiej, zajmiemy się sortowaniem właściwym. Pełny kod naszego rozwiązania reprezentują trzy procedury: `sortowanie`, `insert`, `deletemax`, które podajemy zapisane w języku Pascal. Na pierwszy rzut oka mogą one wydawać się dość skomplikowane. Mam jednak nadzieję, że poniższe intuicje pozwolą zrozumieć, jak dokładnie działa nasz program.

Oczywiście, sortujemy tablicę $a[1 \ldots n]$. Używamy też pomocniczej zmiennej globalnej m, której wartość mówi, jak długi prefiks tablicy a jest kopcowy.

Faza pierwsza realizowana jest za pomocą pętli. Po jej i-tym obrocie podtablica $a[1 \ldots i]$ jest zawsze tablicą kopcową! Żeby się o tym przekonać, dokładnie przemyśl działanie procedury `insert`. Idea jest bardzo prosta: powiększamy aktualną tablicę kopcową o jeden element, po czym poprawiamy drzewo binarne, analizując ścieżkę drzewa od nowego elementu $a[i]$ do korzenia $a[1]$. Rysunek 2 powinien być pomocny.

Czytanie i próby dokładnego rozumienia nieswoich programów to żmudne, ale bezwzględnie konieczne wyzwanie stojące przed każdym młodym programistą.

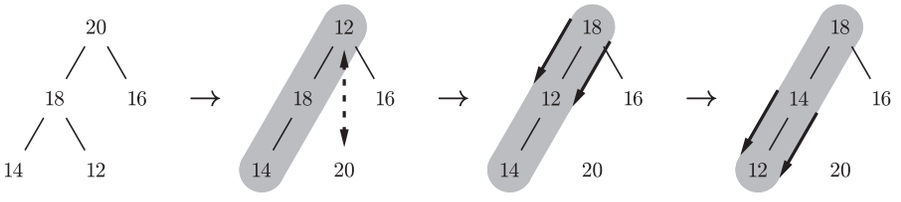


Rys. 2. Schemat działania `insert(19)` dla $a = (20, 18, 16, 14, 12, 10, 8, 6)$

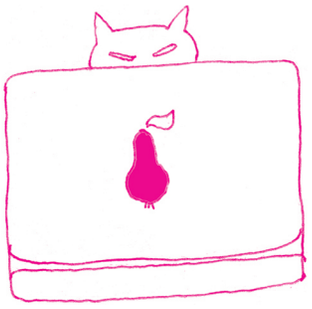
Faza druga jest trochę podobna. Ponownie najważniejsza jest pętla i następujący niezmiennik: gdy indeks pętli jest równy j, to podtablica $a[1 \ldots j]$ jest tablicą kopcową, a elementy $a[j+1 \ldots n]$ są poprawnie posortowane. Aby w to uwierzyć, trzeba oczywiście przeanalizować działanie procedury `deletemax`. Jej zadaniem jest zamienić największy element tablicy kopcowej (czyli zawsze $a[1]$) z ostatnim (czyli $a[m]$) oraz *poprawienie* (rysunek 3) reszty tak, aby wciąż stanowiła tablicę kopcową (choć o jeden element krótszą).

Powyższy algorytm ma dwie bardzo pozytywne cechy. Po pierwsze: jest bardzo szybki. Każde pojedyncze wywołanie procedury `insert` czy `deletemax` zajmuje czas $O(\log n)$ (dlaczego?), a więc łączny czas działania sortowania przez kopcowanie to $O(n \log n)$. Przy pewnych rozsądnych założeniach da się udowodnić, że to optymalny wynik. (Większość narzucających się prostych rozwiązań działa w czasie $O(n^2)$.) Drugą zaletą prezentowanego algorytmu jest złożoność pamięciowa. Poza danymi wejściowymi (tablicą a) potrzebujemy ledwie kilku dodatkowych zmiennych lokalnych. Stąd mówi się czasem, że sortowanie przez kopcowanie działa *w miejscu.*

Co oznacza dziwna wielka litera O tłumaczy Jakub Radoszewski na stronie 10.



Rys. 3. Schemat działania `deletemax` dla $a = (20, 18, 16, 14, 12)$



Poza samym zaprezentowanym (dość szybkim i eleganckim) algorytmem chciałbym zwrócić uwagę Czytelnika na pewną ideę, która za nim stoi. Myślę tu o idei abstrakcyjnych struktur danych. Przykładem takiej struktury jest właśnie kopiec. Z lotu ptaka o kopcu (tablicy kopcowej) można myśleć jak o takim wirtualnym czarnym pudełku, o zawartości którego wcale nie musimy zbyt wiele wiedzieć. Pomyślmy o sytuacji, w której nie chciało nam się analizować działania procedur `insert` oraz `deletemax`. Wiemy tylko, że pierwsza z nich dokłada element do czarnego pudełka, a druga – wyciąga największy. Dodatkowo poinformowano nas, że obie działają szybko (w czasie proporcjonalnym do logarytmu z liczby elementów w pudełku). To już wystarczy do zrealizowania nie tylko sortowania w czasie $O(n \log n)$, ale i napisania np. programu do obsługi harmonogramu zadań, w którym nieustannie pojawiają się nowe pozycje, z różnymi priorytetami, a komputer decyduje, czym się w danej chwili zająć. Takie myślenie (zapominamy o wnętrzu struktury danych i tylko patrzymy na jej funkcjonalność) nazywamy *abstrahowaniem* i ono jest absolutnie solą całej algorytmiki. Nie bez powodu większość kursów i podręczników algorytmiki nosi tytuł „Algorytmy i struktury danych”.

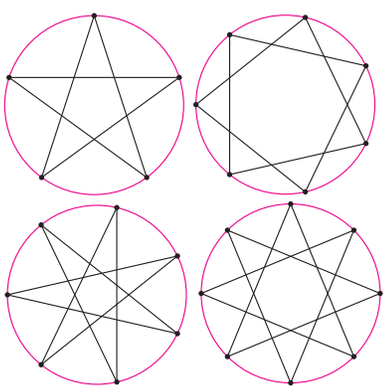
Świat abstrakcyjnych struktur danych to nie tylko kopce, ale i stosy, drzewa czerwono-czarne, kolejki czy miotły. Piękne nie tylko z nazwy, ale i dzięki pomysłowym implementacjom.

```
procedure sort;
var j : integer;
begin
   m := 1;
   for j := 2 to n do insert(a[j]); {faza 1}
   for j := n downto 2 do deletemax; {faza 2}
end;
```

```
procedure insert(v: integer);
var k, p : integer;
begin
   m := m + 1;
   a[m] := v;
   k := m div 2;
   p := m;
   while (k > 0) and (a[k] < v) do
   begin
      a[p] := a[k];
      p := k;
      k := k div 2;
      a[p] := v;
   end;
end;
```

```
procedure deletemax;
var k, p, r : integer;
begin
   r := a[m];
   a[m] := a[1];
   a[1] := r;
   m := m - 1;
   k := 1;
   while (k <= m div 2) do
   begin
      p := 2 * k;
      if (p < m) then
         if a[p] < a[p + 1] then p := p + 1;
      if r >= a[p] then break;
      a[k] := a[p];
      k := p;
   end;
   a[k] := r;
end;
```

Dziwna liczba gwiazdek foremnych



Gwiazdka foremna to łamana zamknięta, wpisana w okrąg i złożona z jednakowej długości cięciw, ale niebędąca wielokątem foremnym. Nie trzeba długo się zastanawiać, by stwierdzić, że odcinki takich łamanych muszą się przecinać.

Każdy łatwo sprawdzi, że obok narysowane są wszystkie gwiazdki mające nie więcej niż 8 wierzchołków.

Jeśli lista wszystkich dzielników pierwszych liczby n to $p_1, p_2, \ldots, p_k$ (uwaga, na liście nie ma powtórzeń, np. 64 ma listę złożoną z jednej liczby), to liczba gwiazdek foremnych mających n wierzchołków dana jest wzorem

$$\frac{n}{2} \cdot \left(\frac{p_1 - 1}{p_1}\right) \cdot \left(\frac{p_2 - 1}{p_2}\right) \cdot \ldots \cdot \left(\frac{p_k - 1}{p_k}\right) - 1,$$

Ale dlaczego akurat tyle?

M.K.