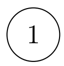
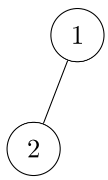
1 Definicje

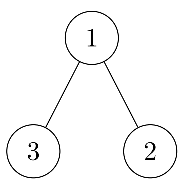
Kopiec to drzewo binarne z porządkiem kopcowym.

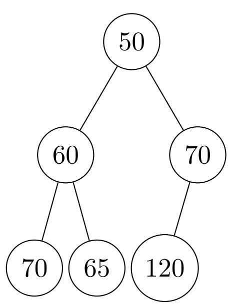
Porządek kopcowy:
Niech $d(v)$ będzie funkcją zwracającą wartość dla wierzchołka v. Wówczas zachodzi $d(v) \leq d(u)$ dla v będącego przodkiem u.

Przykłady poprawnych kopców:





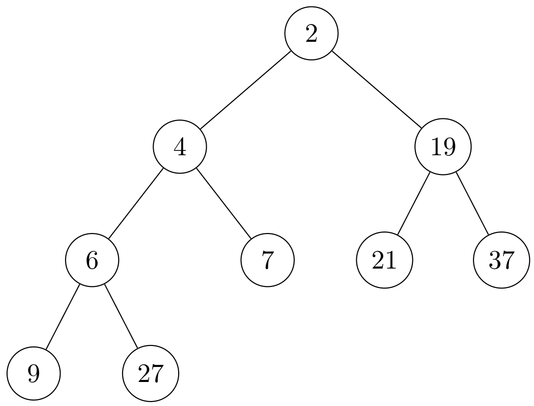




Fakt: wysokość kopca o n elementach wynosi $h = \log n$

2 Reprezentacja kopca

Do pamiętania kopca możemy użyć tablicy.



Powyższe drzewo może być reprezentowane jako:

poziom	korzeń	1	1	2	2	2	2	3	3
element	2	4	19	6	7	21	37	9	27
indeks	1	2	3	4	5	6	7	8	9

Dla takiej reprezentacji synowie elementu v_i to v_{2i} oraz v_{2i+1}.
Natomiast ojcem elementu v_i będzie element $v_{\frac{i}{2}}$.

3 Operacje na kopcu

3.1 *minimum*

return $H[1]$ – zwracamy wartość w korzeniu. Złożoność czasowa takiej operacji to $O(1)$.

3.2 *insert*

Doklejamy nowy liść. Dopóki następuje kolizja z ojcem ($v_{\frac{i}{2}} > v_i$) zamieniamy ich miejscami. Złożoność czasowa takiej operacji to $O(\log n)$.

3.3 *deletemin*

W miejsce korzenia wstawiamy ostatni element kopca. Następnie sprawdzamy czy występuje kolizja z 'mniejszym' z dzieci. Jeśli tak, to dokonujemy zamiany elementów i rekurencyjnie powtarzamy czynność. Złożoność czasowa takiej operacji to $O(\log n)$.

Istnieje również drugie rozwiązanie problemu usuwania wierzchołka. Nadajmy korzeniowi wartość ∞ a następnie spychamy go w dół zamieniając miejscem z synem o mniejszej wartości. Kiedy zepchniemy korzeń do poziomu liści zamieniamy go miejscem z ostatnim liściem (najbardziej na prawo). W takim rozwiązaniu wykonamy mniejszą liczbę porównań (potrzebujemy jednego na zepchnięcie wierzchołka w dół zamiast dwóch).

3.4 *merge*

3.5 *decrease*(k, i, Δ)

Pozwala zmienić wartość i-tego elementu o Δ

3.6 *deletemax*

Do usuwania elementów maksymalnych możemu użyć dwóch kopców, niech nazywają się L, H. W L umieścimy $\lfloor \frac{n}{2} \rfloor$ mniejszych elementów, a w H $\lceil \frac{n}{2} \rceil$ większych elementów, prowadząc jednocześnie krawędzie pomiędzy liśćmi obu kopców, tak aby na **każdej** ścieżce od korzenia L do H był zachowany porządek.

4 tworzenie kopca z tablicy

4.1

Wstawiamy po prostu elementy do kopca.

liczba operacji	liczba elementów
0	1
1	2
2	4
.	.
.	.
.	.
$\log n$	$\frac{n}{2}$

Daje nam to ostatecznie złożoność czasową $O(n \log n)$.

4.2

Druga metoda polega na tworzeniu kopca od dołu.
Wiemy, że liście są poprawnymi kopcami. Dokładamy więc elementy będące ojcami dla kolejnych liści, a jeśli nastąpi kolizja to poprawiamy tak jak w operacjach *insert* i *deletemin*.

liczba operacji	liczba elementów
0	$\frac{n}{2}$
$2 \cdot 1$	$\frac{n}{4}$
$2 \cdot 2$	$\frac{n}{8}$
.	.
.	.
.	.
$2 \cdot \log n$	1

Daje nam to ostatecznie złożoność czasową $O(n)$.

5 Heapsort

Używając kopca możemy posortować dane!
Procedura sortowania wygląda następująco, dla zadanej tablicy A o rozmiarze n:

Algorithm 1 Heapsort

```
make-heap(A)
for iteration = 1, 2, ..., n do
    swap(A[1], A[n − iteration + 1)
    move-down(A, 1)
end for
```

Na koniec działania algorytmu otrzymamy tablicę posortowaną w sposób odwrotny (nie jest to jednak problemem, to my implementujemy kopiec, a zamiana operacji $<$ na $>$ jest prosta).

Powyższy algorytm wykonuje maksymalnie $2 \log n$ porównań dla pojedynczej iteracji, dlatego ostateczna złożoność wynosi $O(n \log n)$.

Istnieje również drugie podejście do rozwiązania tego problemu, które ma jednak znacznie większą szansę na wykonanie mniejszej liczby porównań. Załóżmy, że funkcja *move-down* dodatkowo zwraca indeks elementu po zakończeniu spychania go w dół (jest to dość prosta modyfikacja).

Algorithm 2 Heapsort

make-heap(A)
for $iteration = 1, 2, \ldots, n$ **do**
    $A[1] \leftarrow \infty$
    $index \leftarrow$ move-down(A, 1)
    swap($A[index]$, $A[n - iteration + 1]$)
    move-up(A, $index$)
end for

6 Kolejki priorytetowe

Kolejka priorytetowa jest strukturą danych, która pamięta klucze i ma następujące operacje: *insert*, *min*, *deletemin*. Do ich implementacji możemy wykorzystać strukturę kopców.