

Kolejki

Kolejka priorytetowa

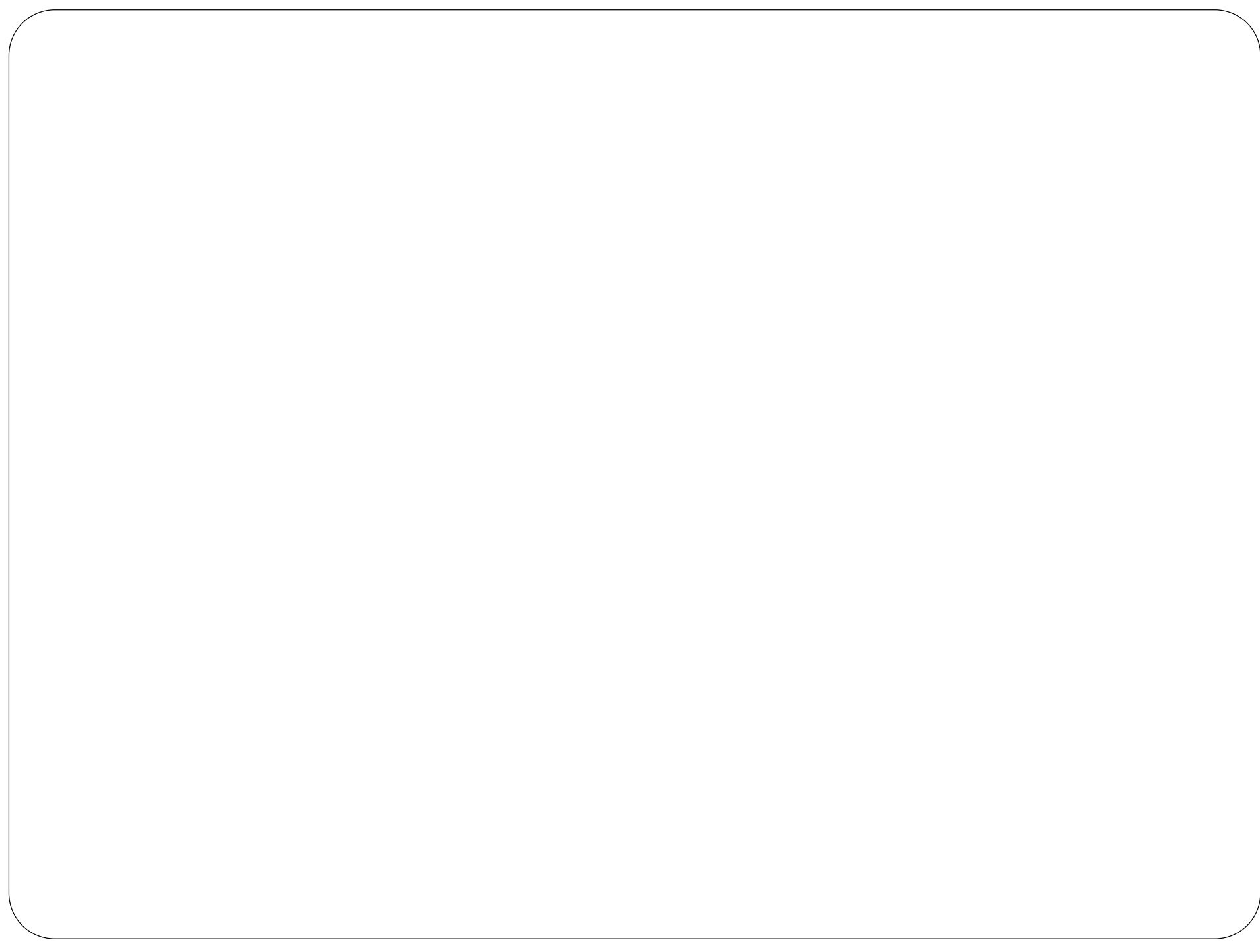
- **Kolejka priorytetowa** (ang. priority queue) to struktura danych pozwalająca efektywnie realizować następujące operacje na zbiorze dynamicznym, którego elementy pochodzą z określonego uniwersum z porządkiem liniowym:
 - `insert(x)` – dodanie nowego elementu `x` do zbioru,
 - `top()` – odczytanie wartości największego elementu w zbiorze,
 - `extract-max(x)` – usunięcie ze zbioru największego elementu.
 - `init()` – utworzenie kolejki priorytetowej z zadanego zbioru danych
- Często rozważa się kolejki priorytetowe, w których poszukuje się elementu minimalnego zamiast maksymalnego.

Implementacja listowa kolejki priorytetowej

- Lista uporządkowana od elementu największego w głowie do najmniejszego w ogonie może być implementacją kolejki priorytetowej.
- Złożoność pamięciowa: $O(n)$
- Złożoność czasowa operacji kolejkowych:
 - $\text{insert}(x) - O(n)$
 - $\text{top}() - O(1)$
 - $\text{extract-max}() - O(1)$
 - $\text{init}() - O(n \log(n))$

Implementacja listowa kolejki priorytetowej

- Lista nieuporządkowana może być implementacją kolejki priorytetowej (ewentualnie z elementem maksymalnym w głowie listy).
- Złożoność pamięciowa: $O(n)$
- Złożoność czasowa operacji kolejkowych:
 - $\text{insert}(x) - O(1)$
 - $\text{top}() - O(n)$ (ewentualnie $O(1)$ gdy na początku znajduje się element maksymalny)
 - $\text{extract-max}() - O(n)$
 - $\text{init}() - O(n)$



Porządek kopcowy w drzewie

- W drzewie jest zachowany **porządek kopcowy**, gdy w każdym węźle synowie przechowują wartości większe (mniejsze) od wartości w węźle.
- Element minimalny (maksymalny) w drzewie z porządkiem kopcowym znajduje się w korzeniu (element minimalny jest w którymś z liści).
- Gdy poruszamy się po drzewie od korzenia do liścia to odwiedzamy coraz to większe (mniejsze) wartości.

Drzewa lewicowe

- **Drzewa lewicowe** to drzewa binarne (czyli każdy węzeł może mieć 0, 1 lub 2 potomków) spełniające, oprócz warunku kopca, tzw. warunek lewicowości.
- **Warunek lewicowości** mówi, że dla każdego węzła skrajnie prawa ścieżka zaczynająca się w danym węźle jest najkrótszą ścieżką od tego węzła do liścia.
- Dzięki temu w każdym drzewie lewicowym **prawa wysokość**, czyli długość skrajnej prawej ścieżki od korzenia do liścia, jest co najwyżej logarytmicznej wielkości, w porównaniu z liczbą elementów drzewa.
- Dodatkowo, aby umożliwić efektywne wykonywanie operacji na drzewie, w każdym węźle przechowywana jest prawa wysokość poddrzewa zaczepionego w tym węźle.

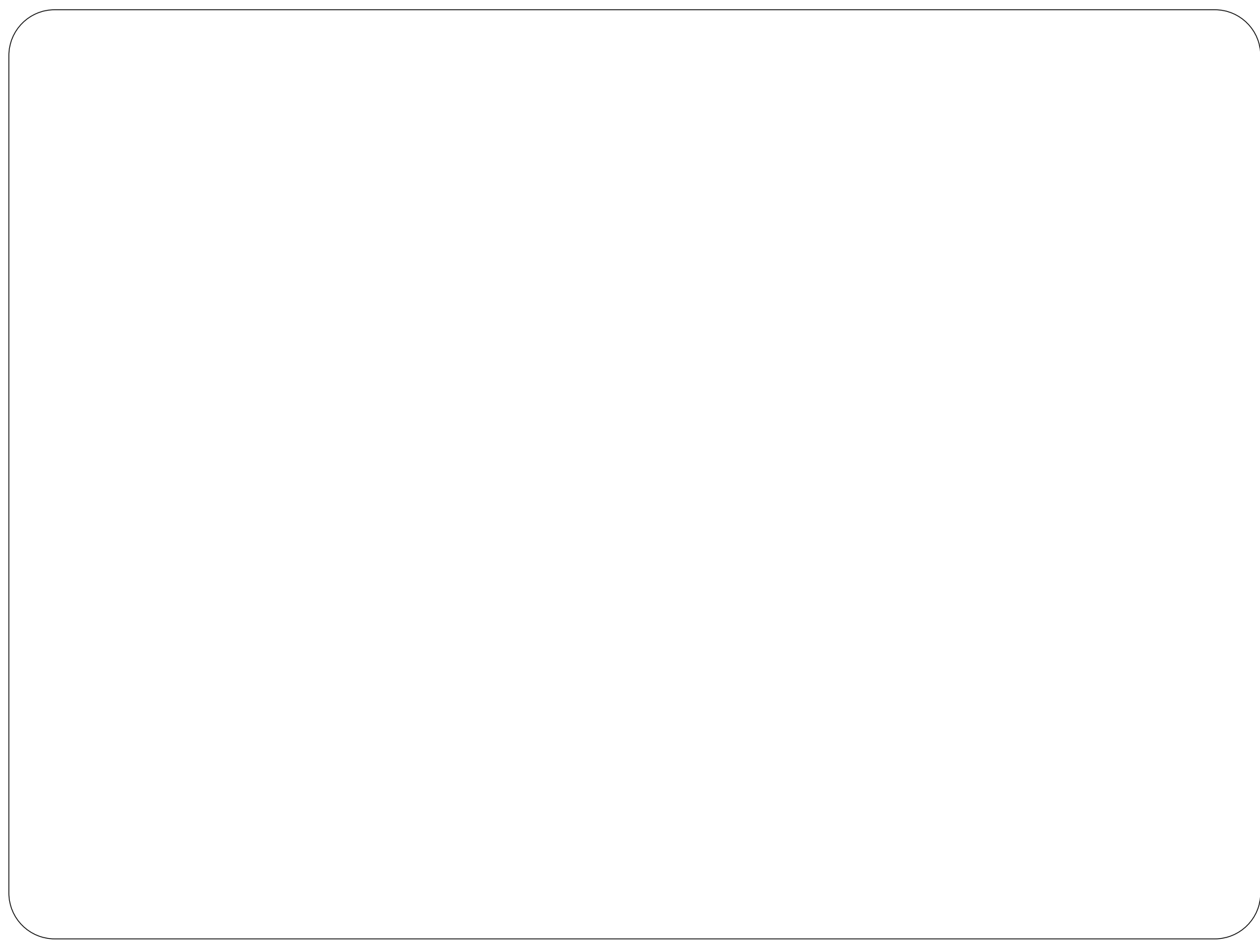
Kopce lewicowe

- **Kopiec lewicowy**, to drzewo lewicowe , w którym zachowany jest porządek kopcowy.
- Najważniejszą operacją na kopcach lewicowych jest ich **łącznie** – pozostałe operacje wykonuje się bardzo prosto:
 - wstawianie elementu do istniejącego drzewa polega na utworzeniu jednoelementowego drzewa i połączeniu go z danym drzewem;
 - usuwanie najmniejszego to usunięcie korzenia drzewa i połączenie poddrzew.



Łączenie kopców lewicowych

- Dane są dwa kopce lewicowe A i B.
- Zakładamy, że element szczytowy w kopcu A jest mniejszy od elementu szczytowego w kopcu B.
- Rekurencyjnie łączymy A_r prawe poddrzewo kopca A i kopiec B w kopiec lewicowy C.
- Wybieramy kopiec o mniejszej prawej wysokości spośród A_l lewego poddrzewa kopca A i kopca C i mniejszy z nich podłączamy jako prawe poddrzewo korzenia kopca A a większe jako lewe.
- Czas, pamięć: $O(\log n)$



Kopiec binarny

- Kopiec binarny to binarne drzewo zupełne z porządkiem kopcowym.
- W n -elementowym kopcu binarnym jest $\lceil n/2 \rceil$ liści.
- N -elementowy kopic binarny jest drzewem regularnym gdy n jest nieparzyste albo posiada jeden węzeł tylko z lewym synem (liściem) gdy n jest parzyste.
- Wysokość (krawędziowa) n -elementowego kopca binarnego wynosi $\lfloor \log n \rfloor$.

Tablicowa implementacja kopca

- Drzewo zupełne wstawiamy do tablicy poziomami, zaczynając od komórki nr 1 (komórka nr 0 nie jest wykorzystywana).
- Numer lewego syna dla komórki i -tej wynosi:
$$\text{left}(i) = i * 2 = i \ll 1$$

a dla prawego syna:
$$\text{right}(i) = i * 2 + 1 = (i \ll 1) \& 1$$
- Numer ojca komórki i -tej wynosi:
$$\text{parent}(i) = \lfloor i / 2 \rfloor = i \gg 1$$

Przesiewanie

- Rozważmy zmianę wartości jednego elementu w kopcu: gdy go zwiększymy element ten trzeba przesiać w górę, gdy go zmniejszymy trzeba go przesiać w dół.
- Przesiewanie to przesuwanie elementu w górę kopca lub dół poprzez zamiany wartości w węzłach tak daleko, aż nie zostanie przywrócony porządek kopcowy.

Przesiewanie

Przesiewanie w górę kopca $\text{heap}[1..n]$

```
sieve-up(int i)
{
    if i=1 then return;
    h := parent(i);
    if heap[i] <= heap[h] then return;
    heap[i] := heap[h];
    sieve-up(h);
}
```

Czas: $O(\log n)$ gdzie n to ilość elementów w kopcu

Pamięć: $O(\log n)$ – głębokość rekurencji (w procedurze iteracyjnej $O(1)$)

Przesiewanie

Przesiewanie w dół kopca $\text{heap}[1\dots n]$

```
sieve-down(int i)
{
    if i > n/2 then return;
    m := left(i); // lewy syn;
    if m+1 <= n then
        if heap[m+1] > heap[m] then m++;
    if heap[m] <= heap[i] then return;
    heap[i] := heap[m];
    sieve-down(m);
}
```

Czas: $O(\log n)$ gdzie n to ilość elementów w kopcu

Pamięć: $O(\log n)$ – głębokość rekurencji (w procedurze iteracyjnej $O(1)$)

Implementacja operacji kopcowych

- Wstawienie nowego elementu do kopca $\text{heap}[1..n]$ dopisujemy element na końcu tablicy i przesiewamy go w górę:

```
insert(comparable x)
{
    n++;
    heap[n] := x;
    sieve-up(n);
}
```

- Wyciągnięcie elementu maksymalnego z kopca

```
heap[1..n]:
extract-max()
{
    heap[1] := heap[n];
    sieve-down(1);
    return heap[n--];
}
```

Implementacja operacji kopcowych

- Odczytanie elementu maksymalnego z kopca
`heap[1..n]:`
`top() -> comparable`
`{`
 `return heap[1];`
`}`
- Czas: $O(\log n)$ dla operacji `insert()` i `extract_max()` oraz $O(1)$ dla `top()`, gdzie n to ilość elementów w kopcu.
- Pamięć: $O(\log n)$ dla operacji modyfikujących i $O(1)$ dla `top()` – głębokość rekurencji (implementując przesiewanie w sposób iteracyjny zapotrzebowanie na pamięć dla wszystkich operacji wyniesie $O(1)$).

Budowanie kopca

- Dana jest tablica n-elementowa z danymi.
- Naszym zadaniem jest zbudowanie kopca.
- Idea rozwiązania: budowa kopca od dołu
 - wszystkie liście są kopcami;
 - jeśli do korzenia są połączone dwa kopce, to wystarczy przesiać drzewo w dół od korzenia aby otrzymać kopiec.

```
init-heap()  
{  
    for i =  $\lfloor n/2 \rfloor$  ... 1 do  
        sieve-down(i);  
}
```

- Czas: $O(n)$

Sortowanie kopcowe

- Idea: budujemy kopic poprzez wkładanie kolejnych elementów do kopca a potem usuwamy kolejno elementu od największego do najmniejszego i umieszczamy je w tablicy wynikowej od końca.
- Czas: $O(n \cdot \log n)$
- Pamięć: $O(1)$ gdy nie używamy rekurencji w przesiewaniach albo $O(\log n)$ w przeciwnym razie
- Sortowanie kopcowe nie jest stabilne.
- Sortowanie kopcowe może działać w miejscu, gdy wyeliminujemy rekurencję z przesiewania.