ALGORYTMY SORTOWANIA DANYCH

W zagadnieniu sortowania danych rozpatrywać będziemy n liczb całkowitych, będących pierwotnie w losowej kolejności, które należy uporządkować nierosnąco.

Oczywiście sortować możemy też inne typy danych niż liczby całkowite (np. liczby zmiennoprzecinkowe, ciągi znaków alfanumerycznych, itp.), jednak idea algorytmów nie zmienia się.

Podobnie, jeśli chodzi o sortowanie w porządku odwrotnym do założonego (niemalejąco) – zmiany sprowadzają się praktycznie do zamiany znaków mniejszości / większości na przeciwne.

Najprostszym jest algorytm

sortowania bąbelkowego,

który można zaimplementować jako dwie zagnieżdżone pętle typu **for**, z których każda wykonuje się $O(n)$ razy. Zadaniem tych pętli jest umieszczenie każdego z elementów na właściwej pozycji.

```
int B[n];

void bubble(int* B)
{ for(j=1; j<n; j++)
    for(i=n-1; i>0; i--)
      if(B[i-1]>B[i])
      { temp=B[i];
        B[i]=B[i-1];
        B[i-1]=temp;
        };
  };
```

$$\left.\left.\begin{matrix}\text{for(i=...)}\\ \ldots\end{matrix}\right\} O(n) \quad \right\} O(n) \;=\; \boldsymbol{O(n^2)}.$$

Złożoność obliczeniowa sortowania bąbelkowego jest **zawsze** $O(n^2)$ – każda z pętli wykona się zawsze $O(n)$ razy, chociaż np. w sytuacji wstępnego posortowania danych wejściowych liczbę obiegów pętli wewnętrznej można by zmniejszyć.

Uwzględnia to algorytm

sortowania przez wstawianie,

gdzie wewnętrzna pętla „szuka'' właściwej pozycji dla bieżącego elementu, ale tylko wśród elementów wcześniej posortowanych:

```
int B[n];

void insert(int* B)
{ for(i=2;i<n+1;i++)
  { j=i-1;
    while(B[j]<B[j-1])
    { pom=B[j];
      B[j]=B[j-1];
      B[j-1]=pom;
      j--;
      if(j<1) break;
      };
    };
  };
```

Inner block (from `{ j=i-1;` to `if(j<1) break;`): $O(i)$; whole loop: $O(n)$

Czy takie usprawnienie przekłada się na lepszą złożoność obliczeniową?

Liczba iteracji pętli wewnętrznej za **pierwszym** razem (tj. przy pierwszej iteracji pętli zewnętrznej) wynosi maksymalnie **1**, za **drugim** razem – **2**, potem **3**, itd. Najwięcej razy wykona się w ostatniej (($n-1$)**-szej**) iteracji pętli zewnętrznej – $n-1$ razy.

Zatem, aby wyznaczyć **złożoność obliczeniową całej procedury** sortowania przez wstawianie, wystarczy policzyć sumę następującego ciągu:

$$1 + 2 + 3 + \ldots + (n-2) + (n-1) = \sum_{j=1}^{n-1} j,$$

$$\sum_{j=1}^{n-1} j = \frac{1}{2}(n-1)n = \frac{1}{2}\,n^2 - \frac{1}{2}\,n = O(n^2).$$

Uzyskaliśmy więc algorytm tego samego rzędu co algorytm sortowania bąbelkowego.

Jednak rozważmy przypadek, gdy liczby są już posortowane.

(Przypomnijmy, że w takiej sytuacji algorytm sortowania bąbelkowego wykona tyle samo operacji, co w przypadku danych nieposortowanych).

W takim przypadku, w algorytmie sortowania przez wstawianie **pętla wewnętrzna** nie wykona się **ani razu** w każdej z $n-1$ iteracji pętli zewnętrznej. Zatem **czas działania algorytmu** ograniczy się do $\boldsymbol{O(n)}$.

Sortowanie przez kopcowanie

Wykorzystajmy teraz strukturę **kopca** do skonstruowania algorytmu sortowania.

Idea sprowadza się do następujących kroków:

1. Umieść wszystkie **dane wejściowe w kopcu**. Utwórz **pustą tablicę wyjściową** o długości n.

2. Pobierz element **z korzenia na pierwszą wolną** pozycję w tablicy wyjściowej.

3. Umieść **w korzeniu** (w miejsce pobranego elementu) **ostatni element** z kopca (rozmiar kopca zmniejsza się o 1). **Przywróć właściwość kopca** (analogicznie, jak w procedurze usuwania elementu z kopca).

4. Jeśli kopiec **nie jest pusty** to skocz do Kroku 2; w przeciwnym wypadku **STOP** – posortowane dane są w tablicy wyjściowej.

Wyznaczmy teraz **złożoność obliczeniową** algorytmu sortowania przez kopcowanie.

Po pierwsze, należy zauważyć, że w algorytmie kroki 2-4 są powtarzane n razy (w każdej iteracji, kopiec – początkowo n elementowy – zmniejsza się o 1).

Zatem **złożoność całej procedury** można wyliczyć jako:

Złożoność Kroku 1 + $n\cdot$ Złożoność Kroków 2-4.

Wyznaczmy więc **złożoność poszczególnych kroków** algorytmu.

Złożoność Kroku 1: Utworzenie kopca z n losowych danych to, jak już wspomniano, n-krotne wykonanie operacji wstawienia elementu do kopca. Wychodząc z założenia, że wstawienie elementu do kopca n elementowego zajmuje $O(\log n)$ czasu (tyle, ile wynosi wysokość kopca), złożoność Kroku 1 można oszacować jako $\boldsymbol{O(n \log n)}$.

Oczywiście **2-ga część Kroku 1** – utworzenie tabl. wyjśc. – zajmuje $\boldsymbol{O(1)}$.

UWAGA: Utworzenie kopca można też wykonać **szybciej** w sposób *wstępujący*.

W metodzie tej zakładamy, że **od początku kopiec jest wypełniony** losowymi danymi. Rozpoczynając od elementu $\lfloor n/2 \rfloor$ przechodzimy przez kolejne elementy aż do pierwszego i za każdym razem wykonujemy operację **naprawy części kopca** leżącego poniżej (tj. poddrzewa, dla którego bieżący element jest korzeniem), na podobnej zasadzie jak naprawa kopca po usunięciu elementu.

W takiej sytuacji można wykazać, że w kopcu jest najwyżej $\lceil n/2^{k+1} \rceil$ elementów mających poniżej siebie k poziomów. Dla każdego takiego elementu operacja naprawy części kopca leżącego poniżej zajmuje $\boldsymbol{O(k)}$ czasu.

Zatem **złożoność Kroku 1** można wyliczyć jako:

$$\sum_{k=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{k+1}} \right\rceil O(k) = O\left(n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \right).$$

Korzystając teraz z zależności (patrz np. Cormen *i in.*)

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1/2}{(1-1/2)^2} = 2,$$

otrzymamy

$$O\left(n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \right) = O\left(n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \right) = \underline{O(n)}.$$

Złożoność Kroków 2 i 4 to oczywiście $\underline{O(1)}$,

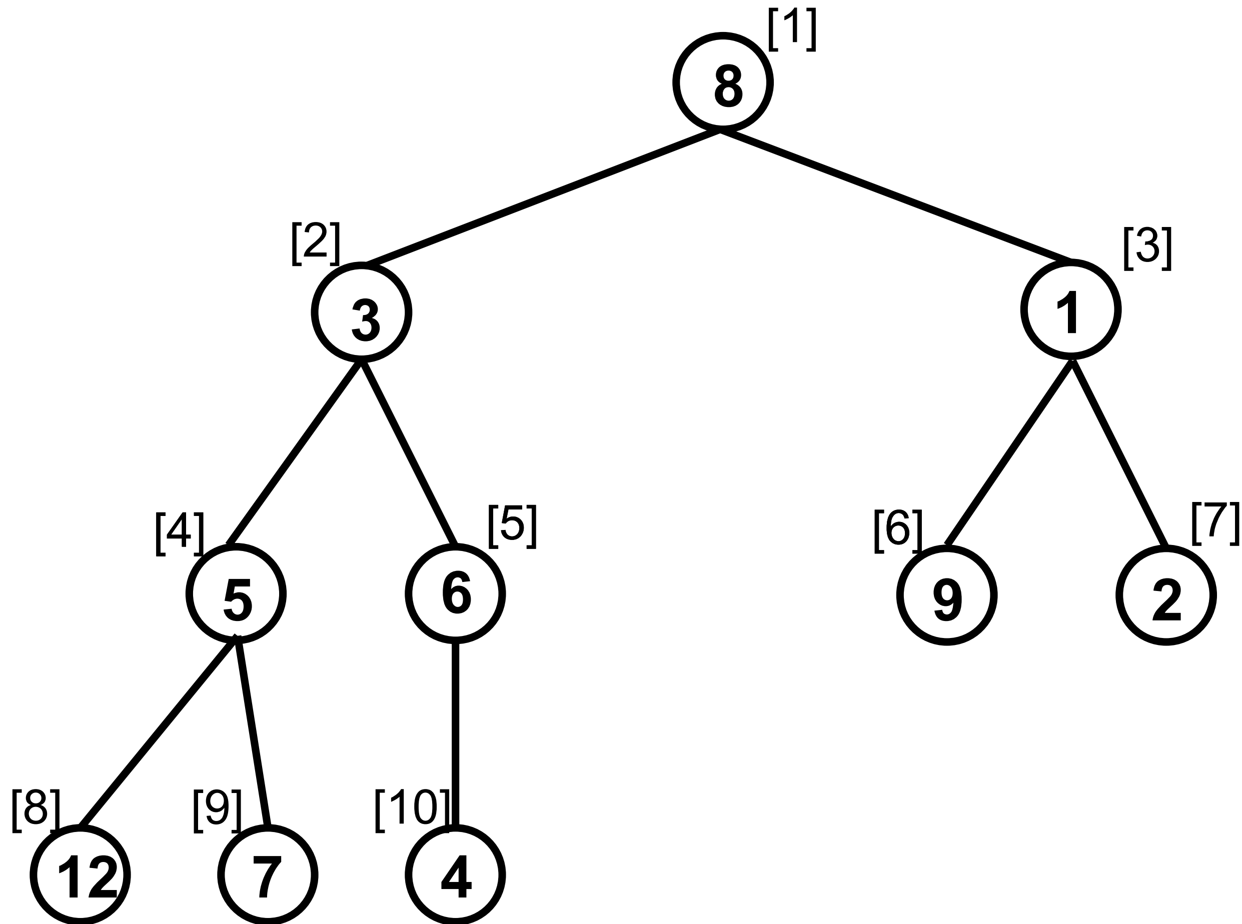
natomiast **Kroku 3** – $\underline{O(\log n)}$.

Zatem **złożoność całego algorytmu** to

$$O(n) + n \cdot (O(1) + O(\log n) + O(1)) = \underline{O(n \log n)}.$$

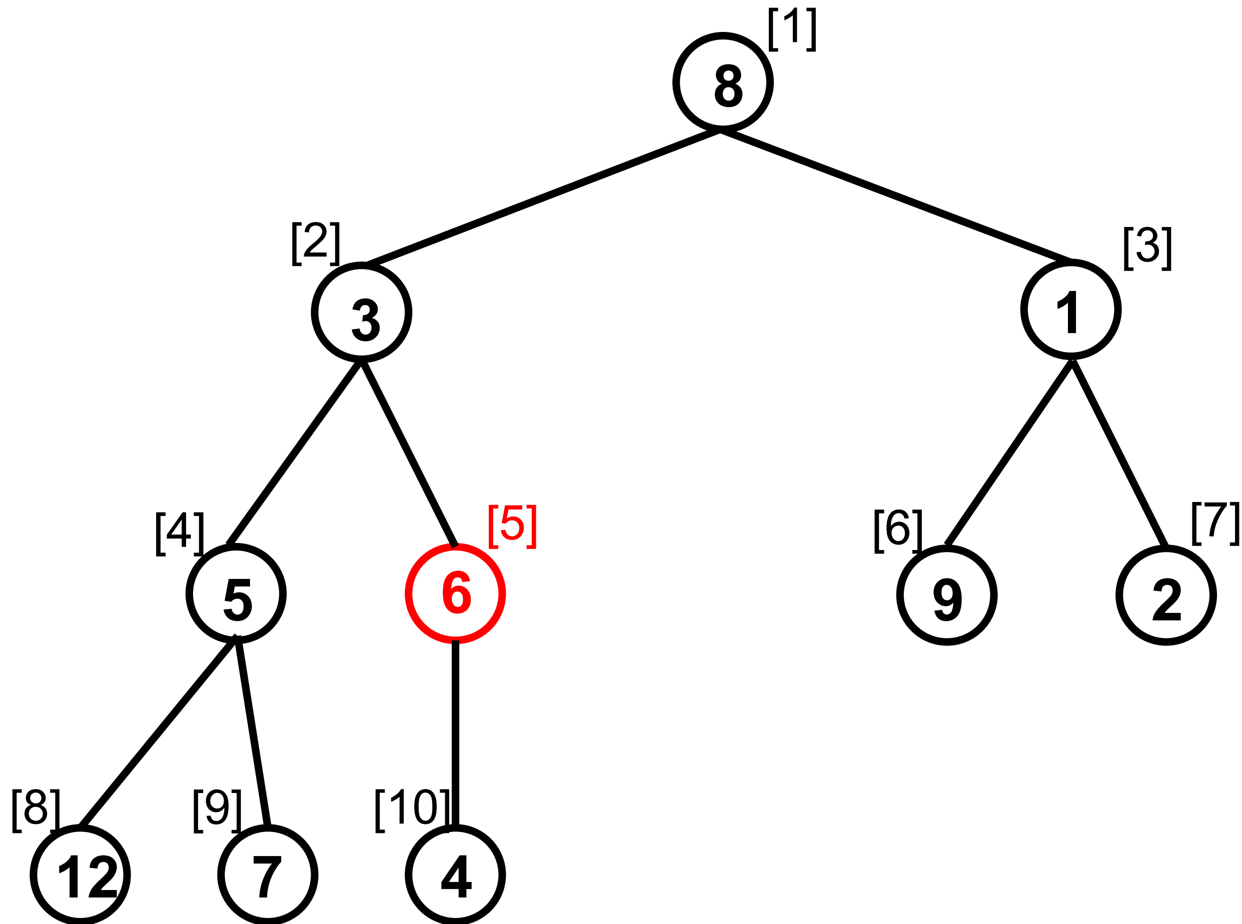
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



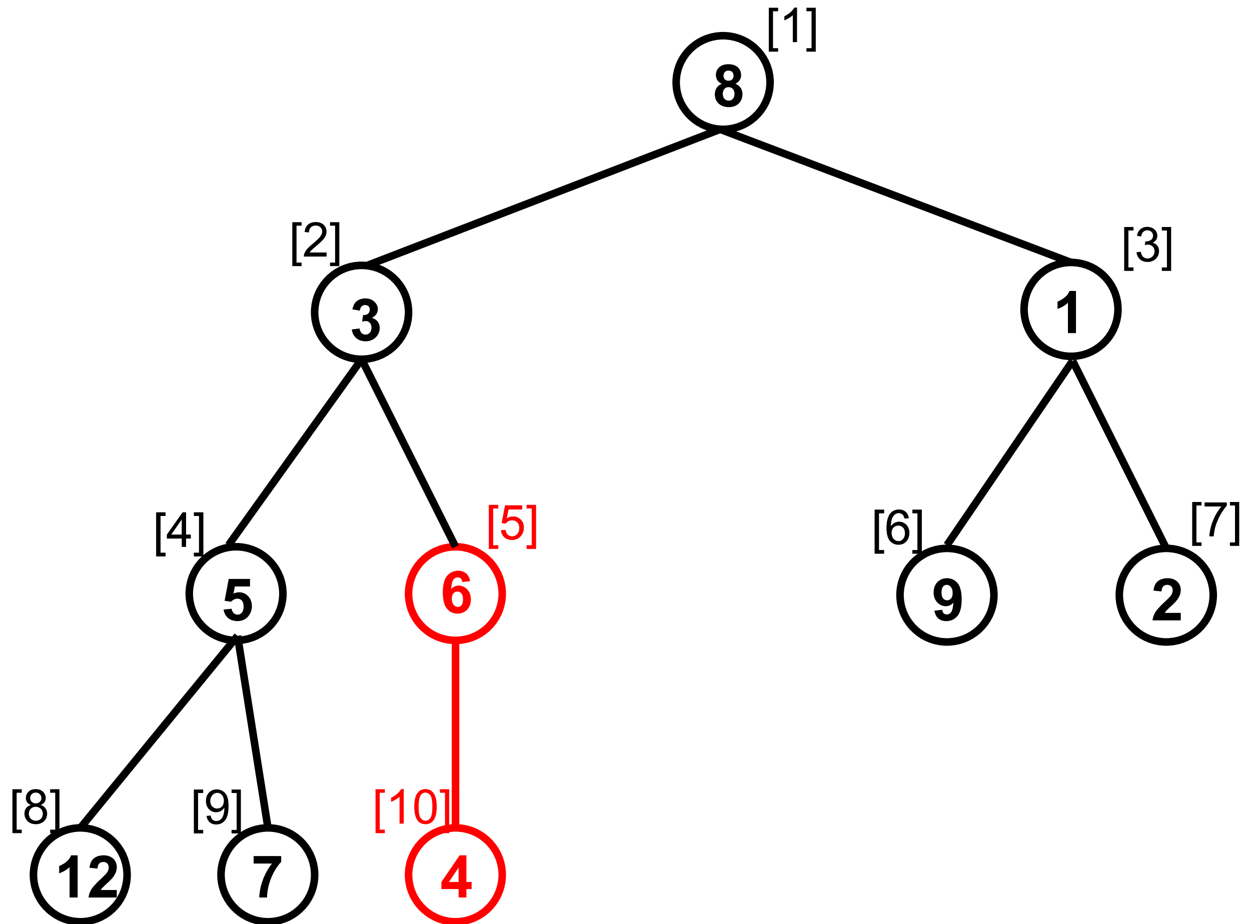
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



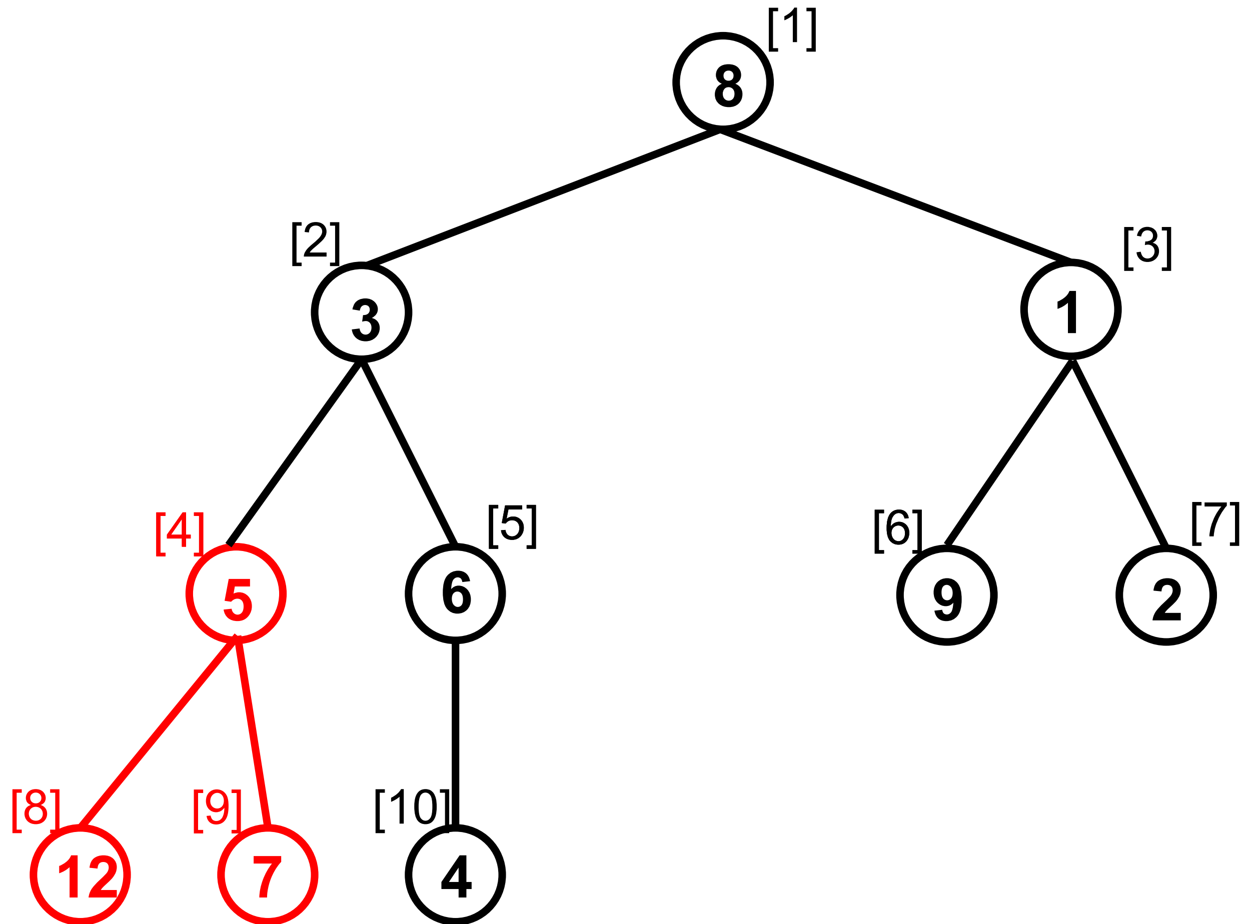
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



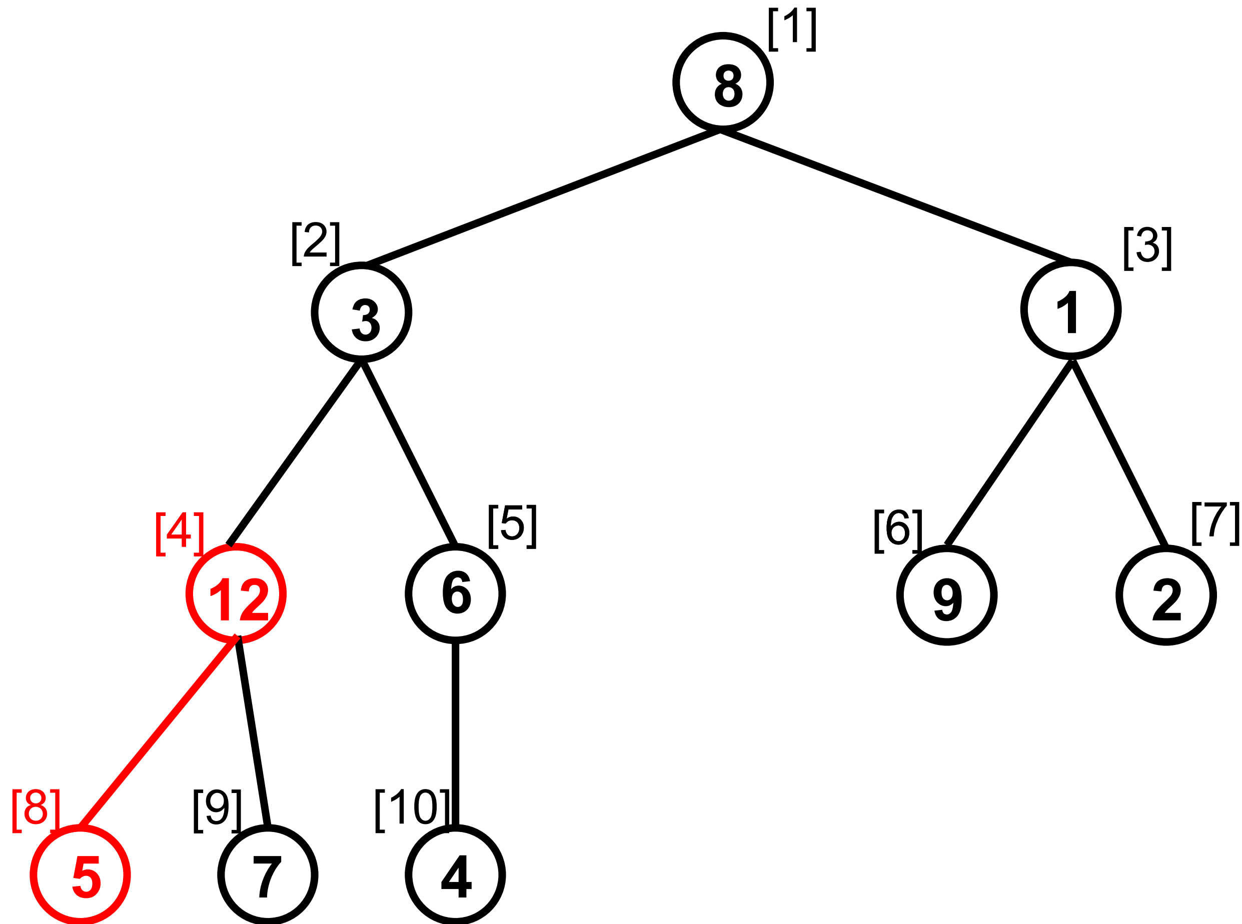
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



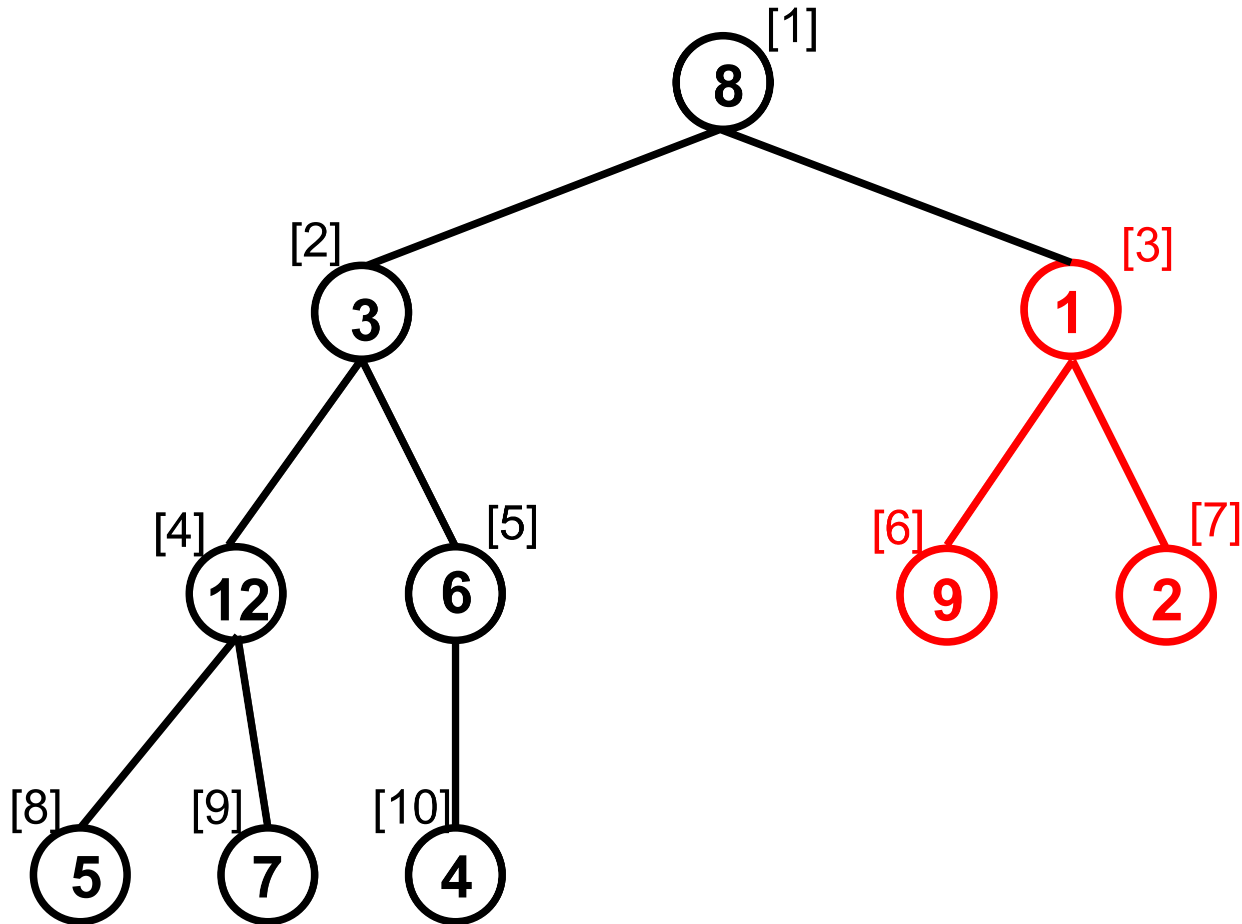
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



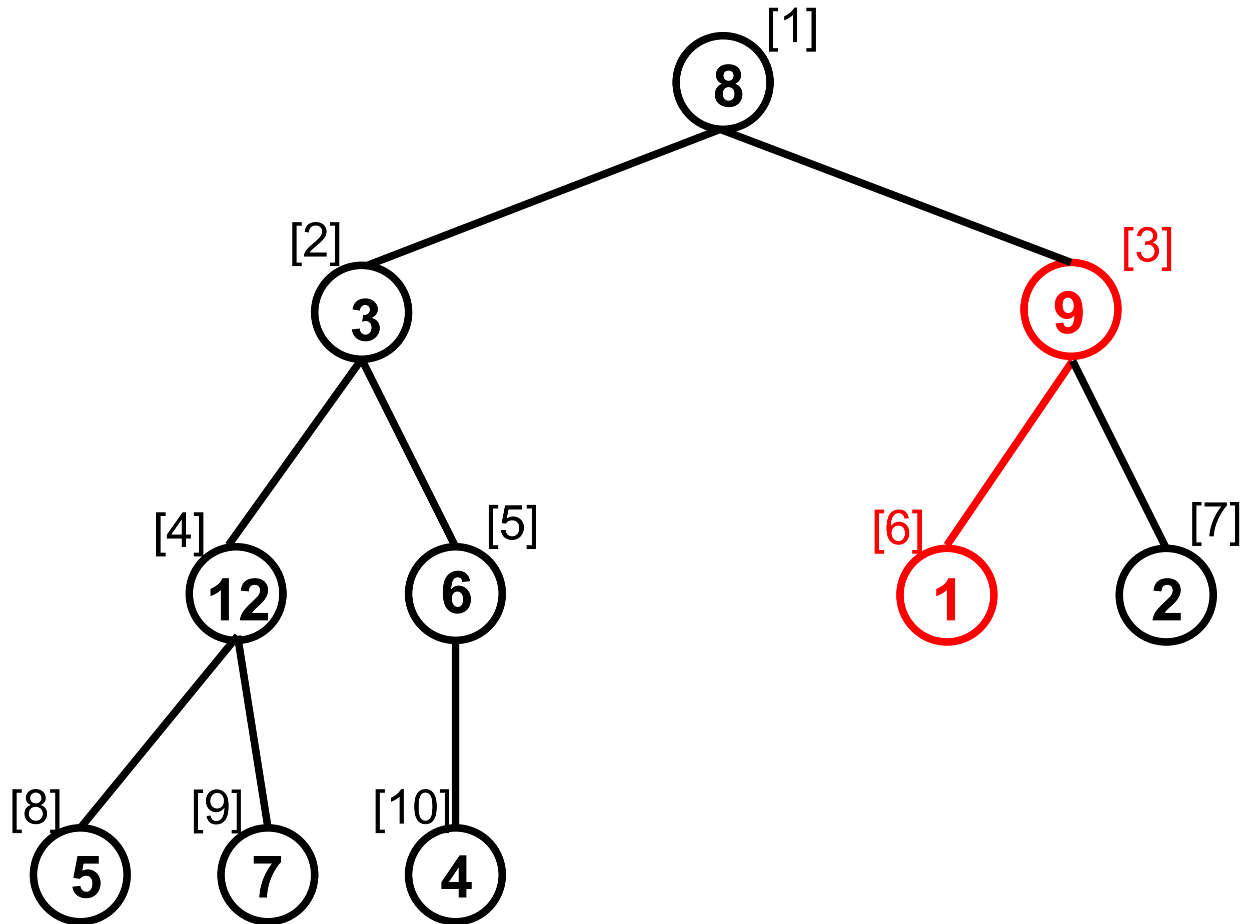
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



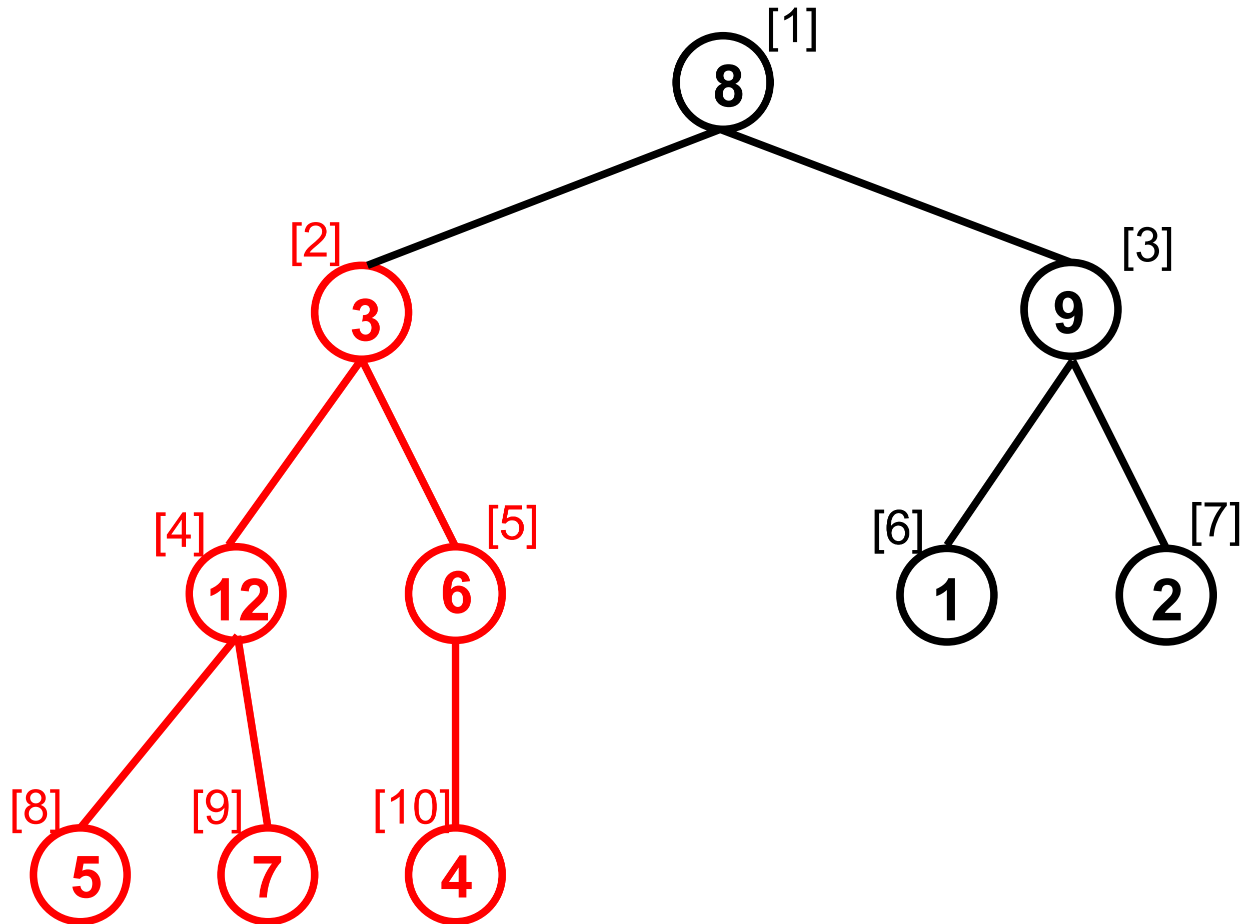
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



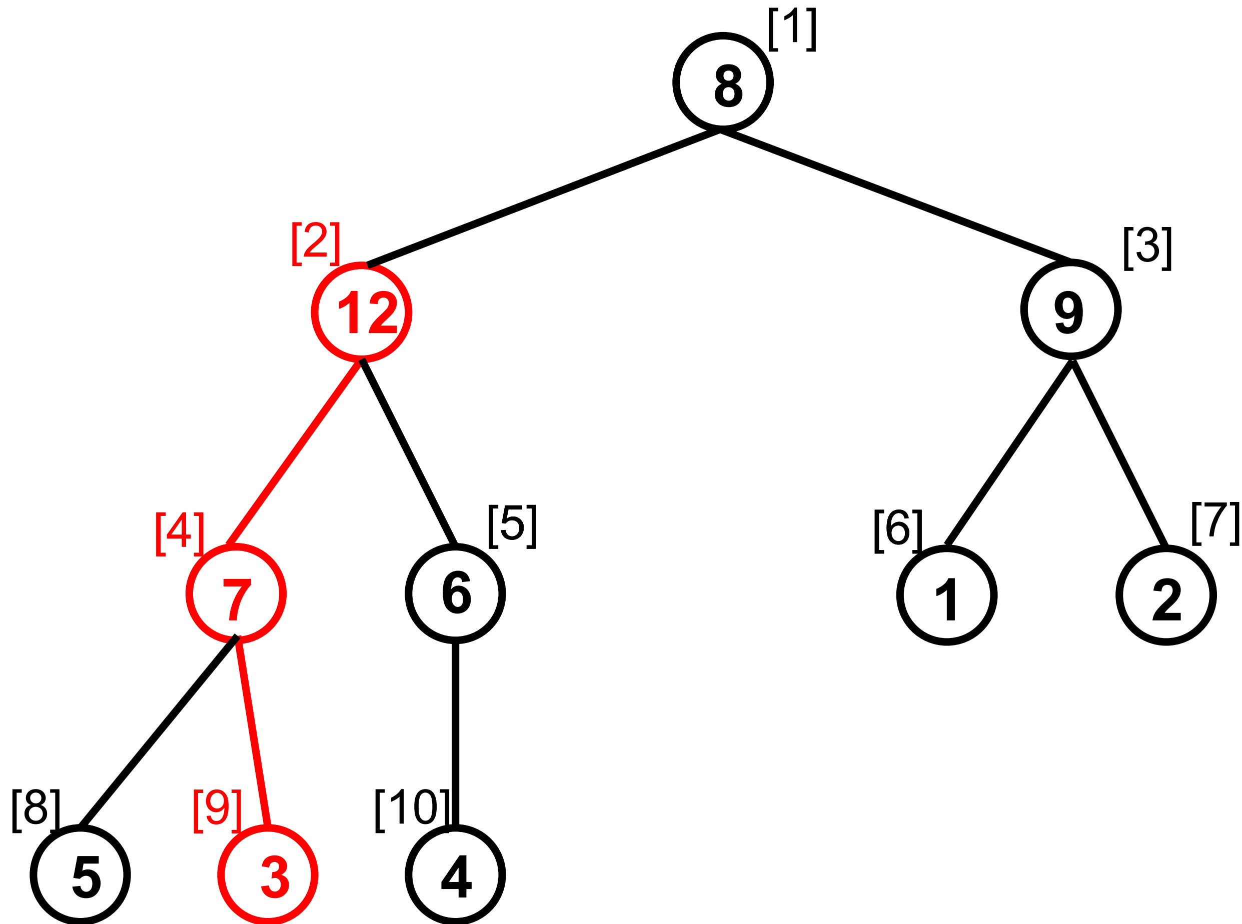
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



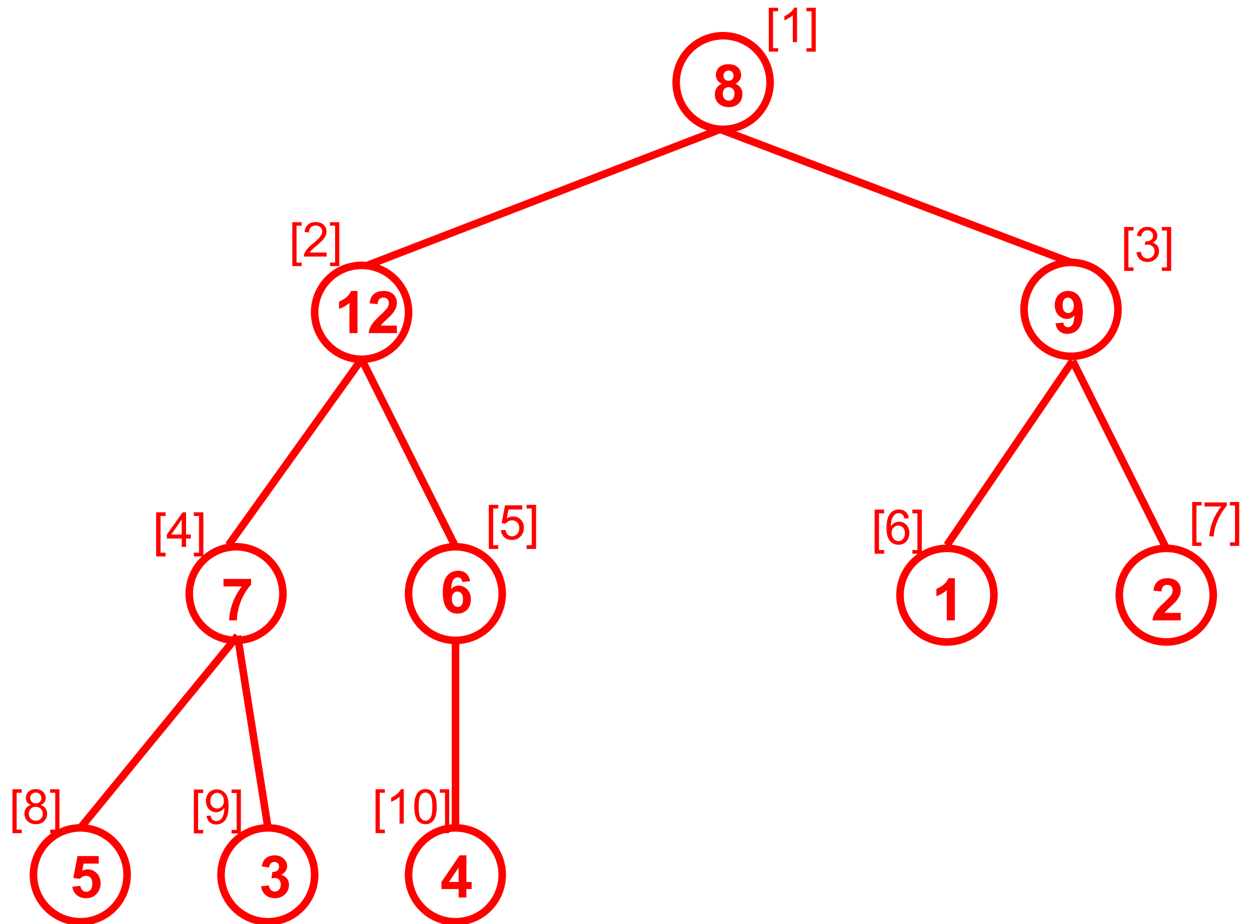
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



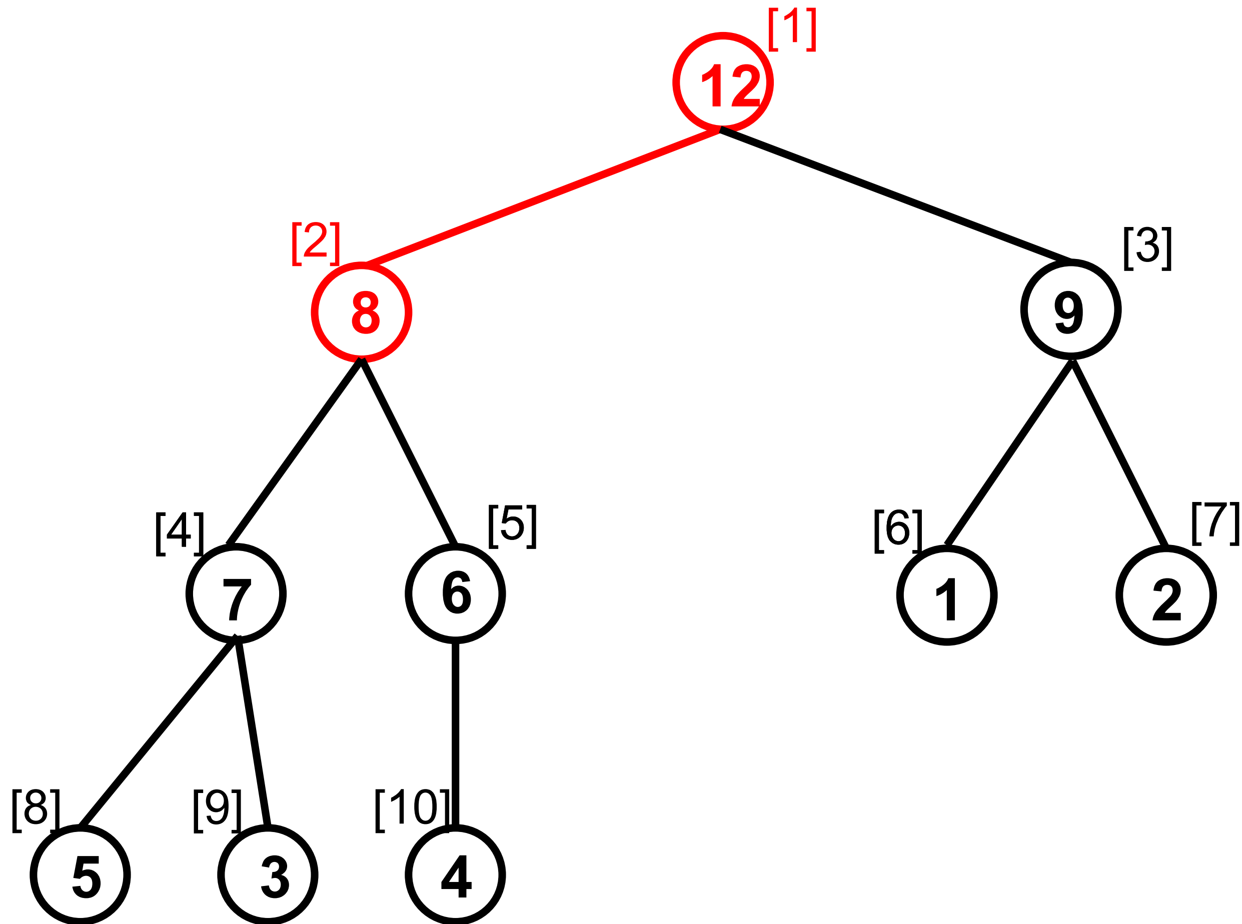
Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

[1] 12

[2] 8 [3] 9

[4] 7 [5] 6 [6] 1 [7] 2

[8] 5 [9] 3 [10] 4

Tablica wyjściowa:

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

[1]

[2] **8**

[3] **9**

[4] **7**

[5] **6**

[6] **1**

[7] **2**

[8] **5**

[9] **3**

[10] **4**

Tablica wyjściowa:

12									

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

Tablica wyjściowa:

12									

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

[1] **4**

[2] **8** [3] **9**

[4] **7** [5] **6** [6] **1** [7] **2**

[8] **5** [9] **3**

Tablica wyjściowa:

12									

Sortowanie przez kopcowanie:

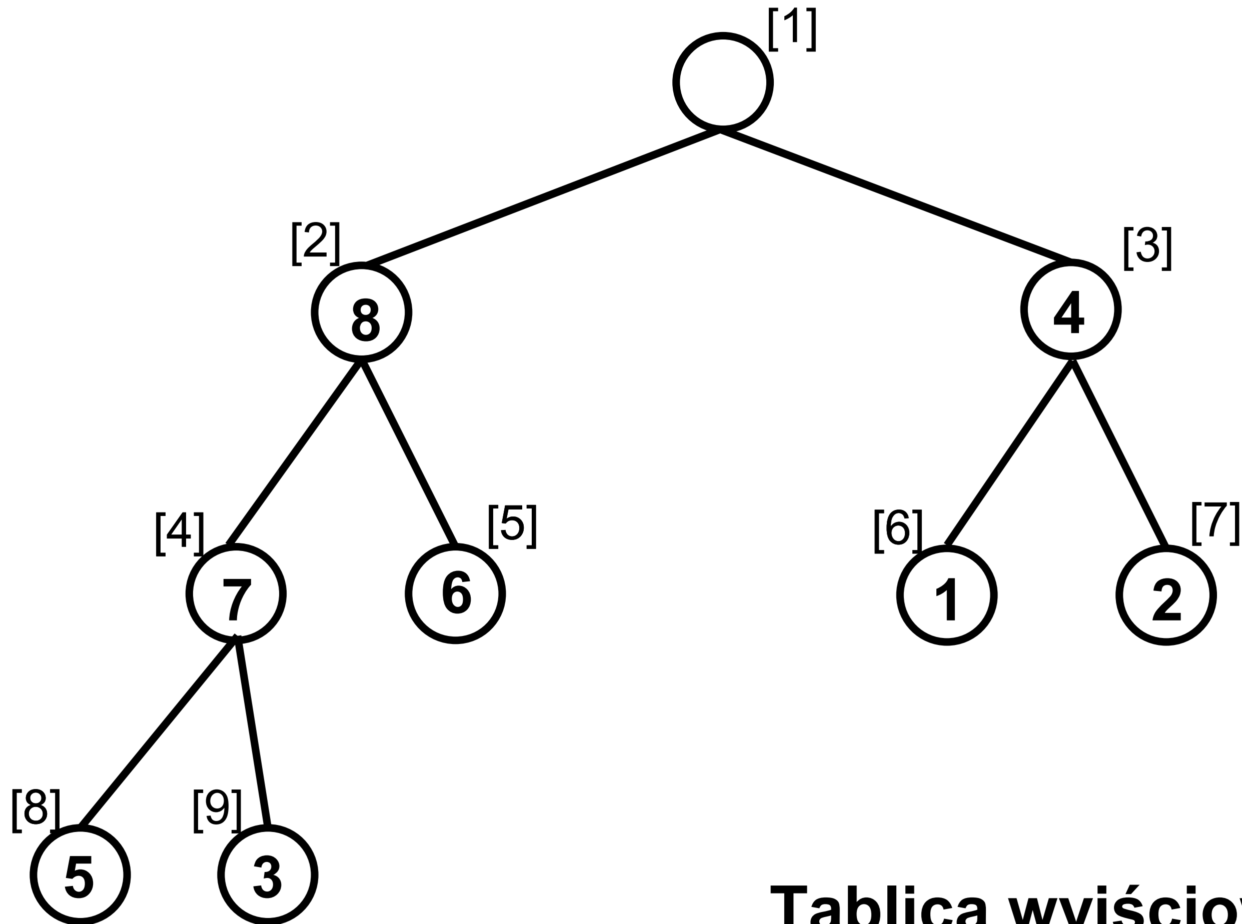
przykład dla n **= 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

Tablica wyjściowa:

12									

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

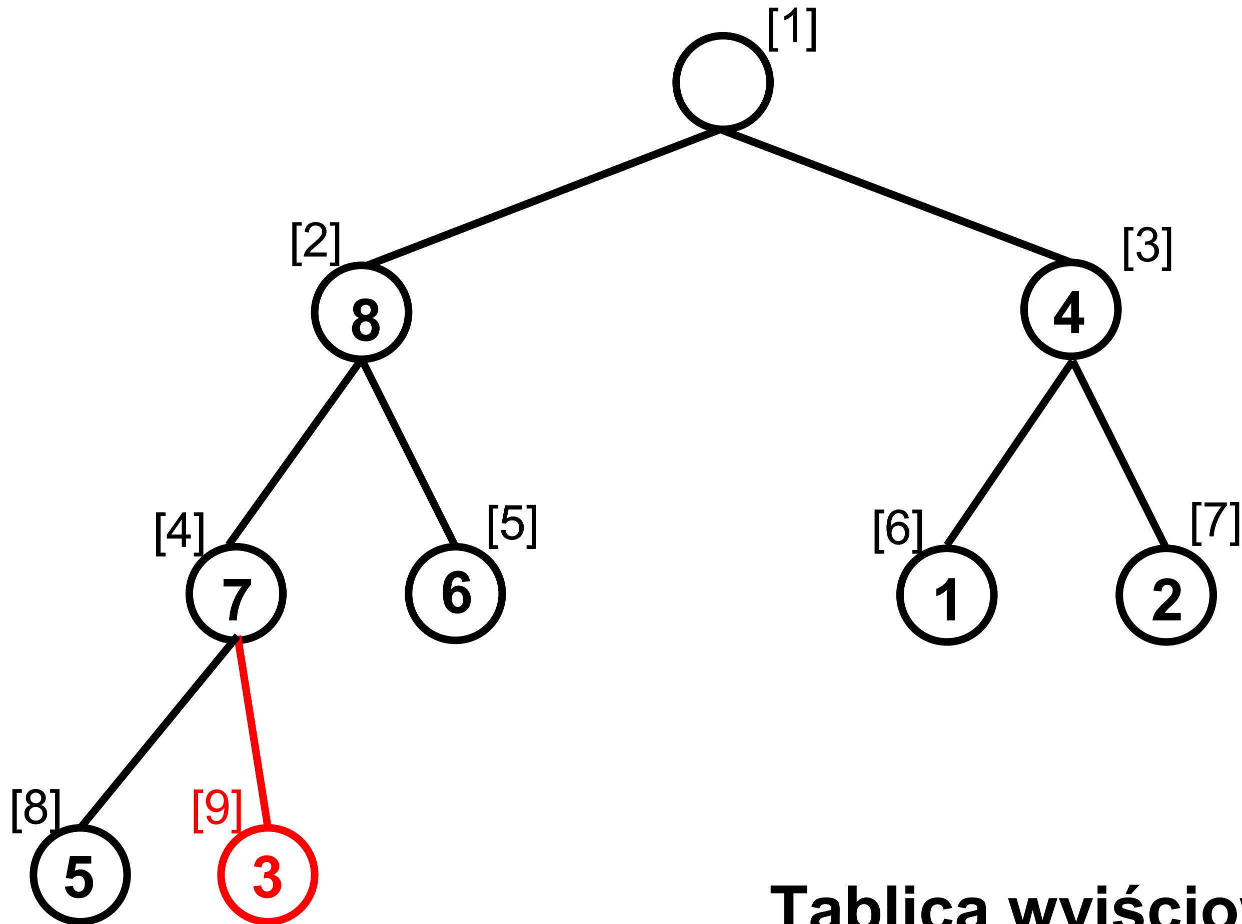


Tablica wyjściowa:

12	9								

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

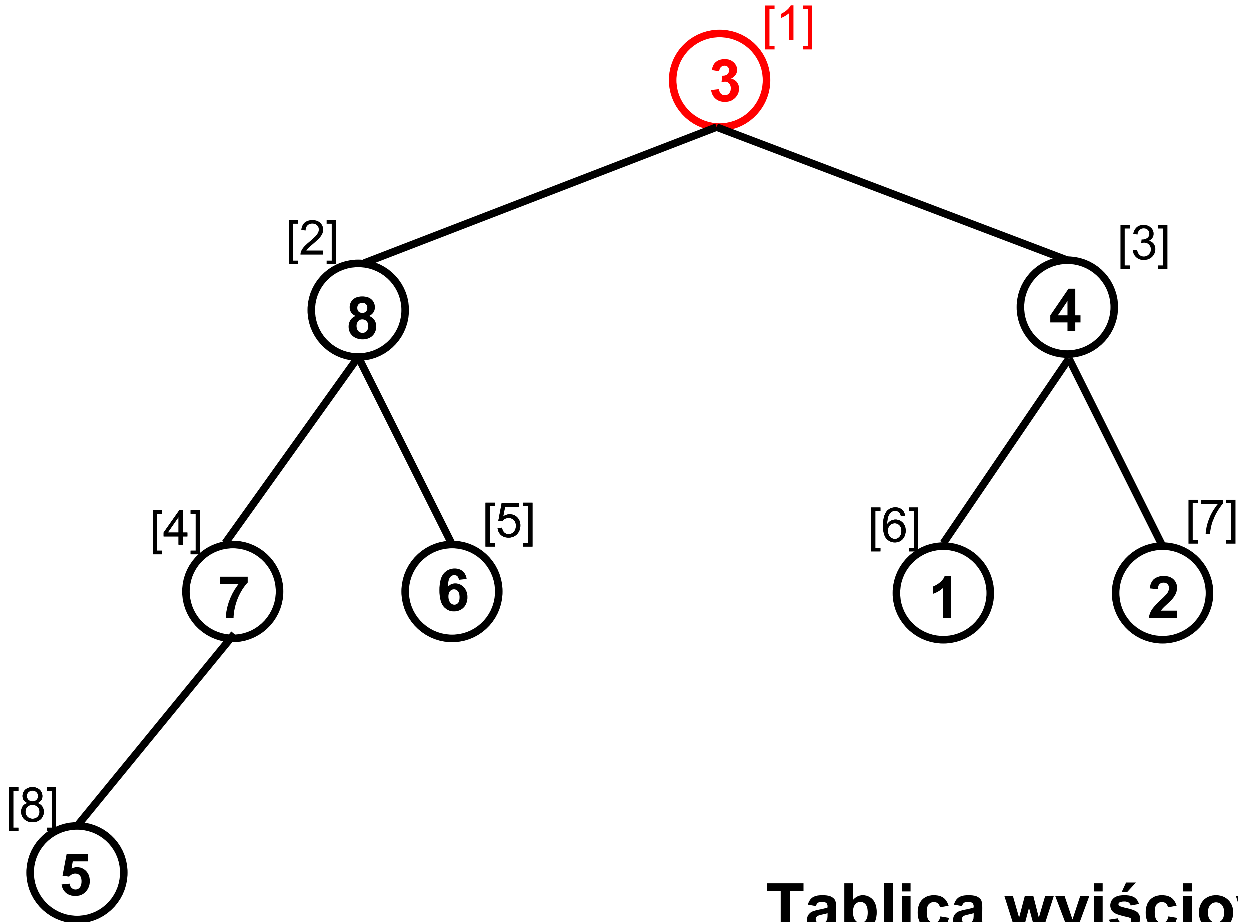


Tablica wyjściowa:

12	9								

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

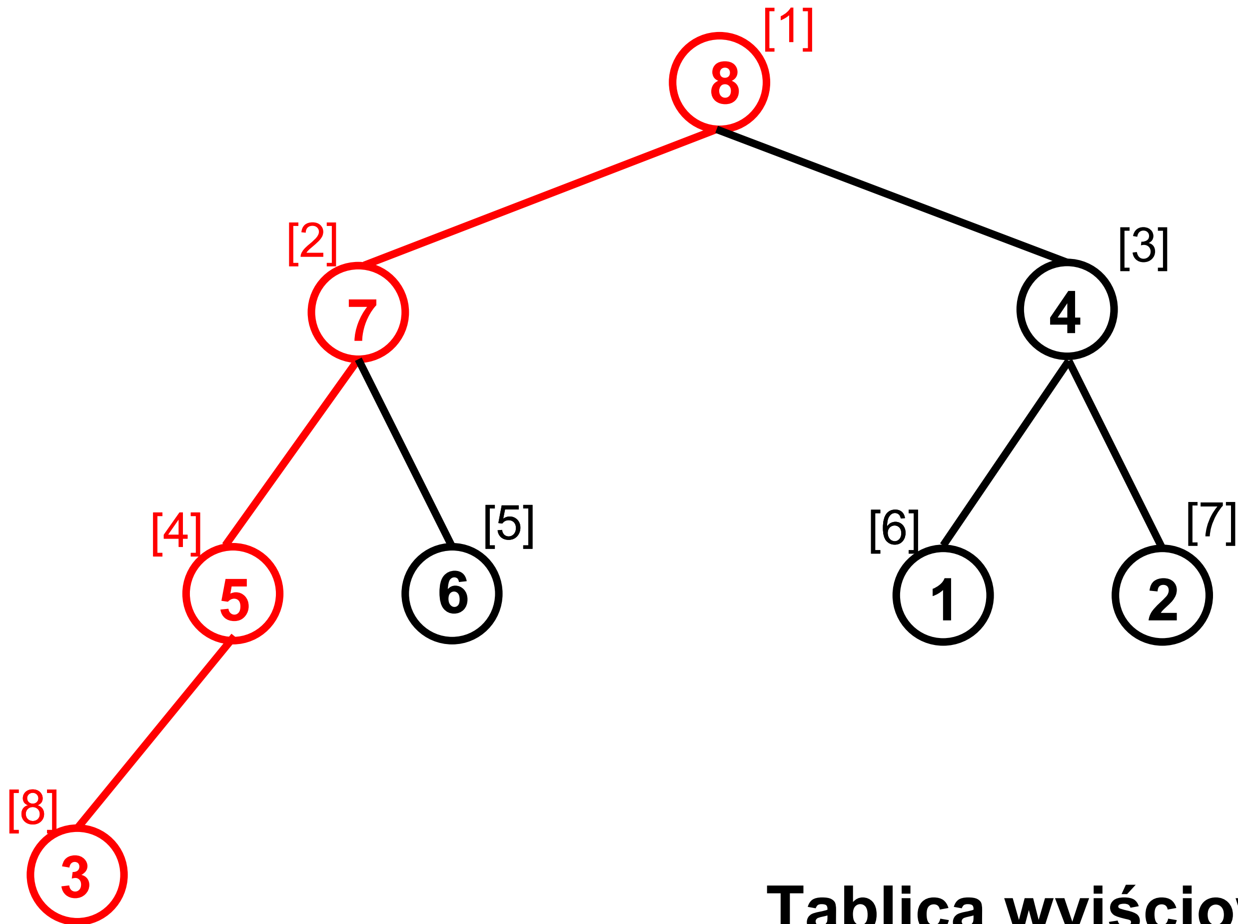


Tablica wyjściowa:

12	9								

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

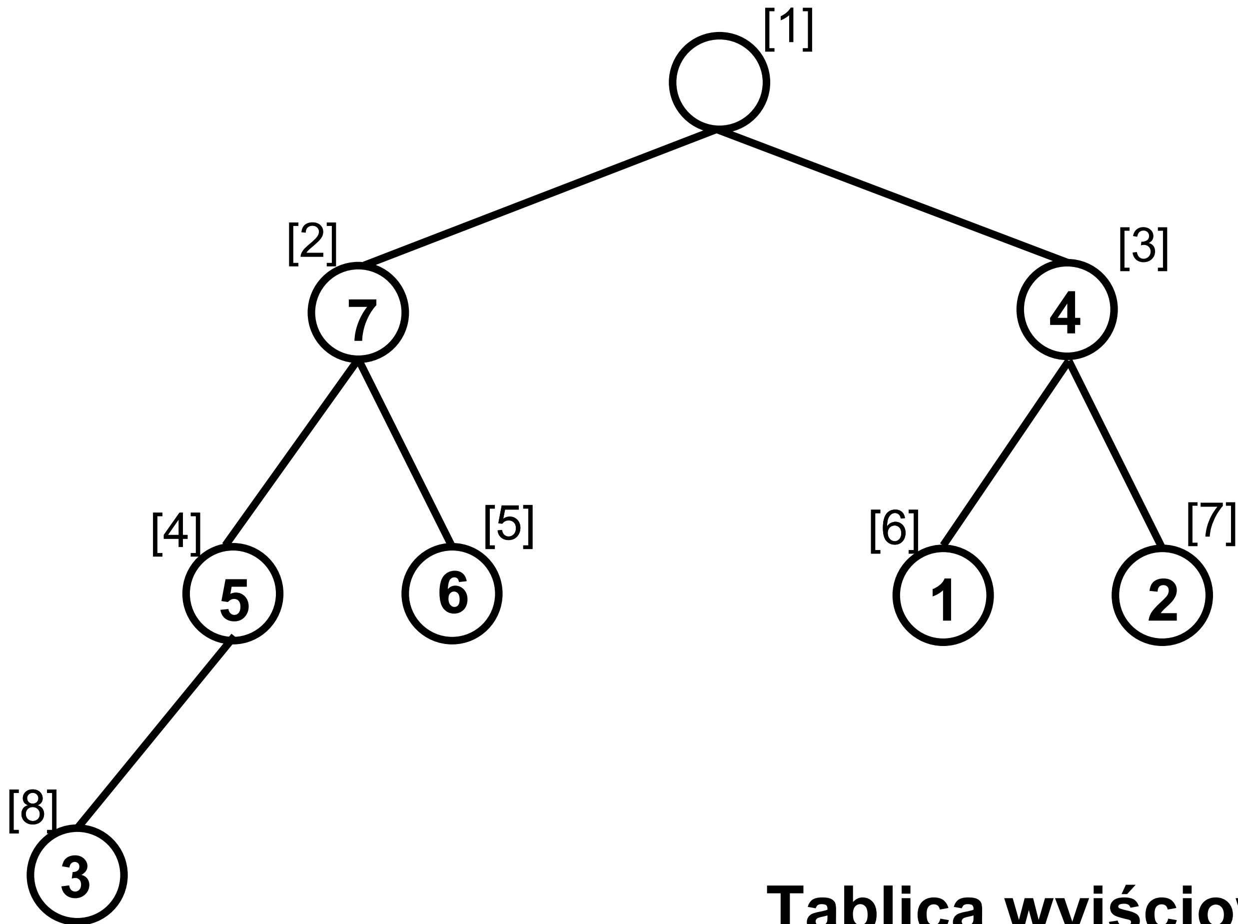


Tablica wyjściowa:

12	9								

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

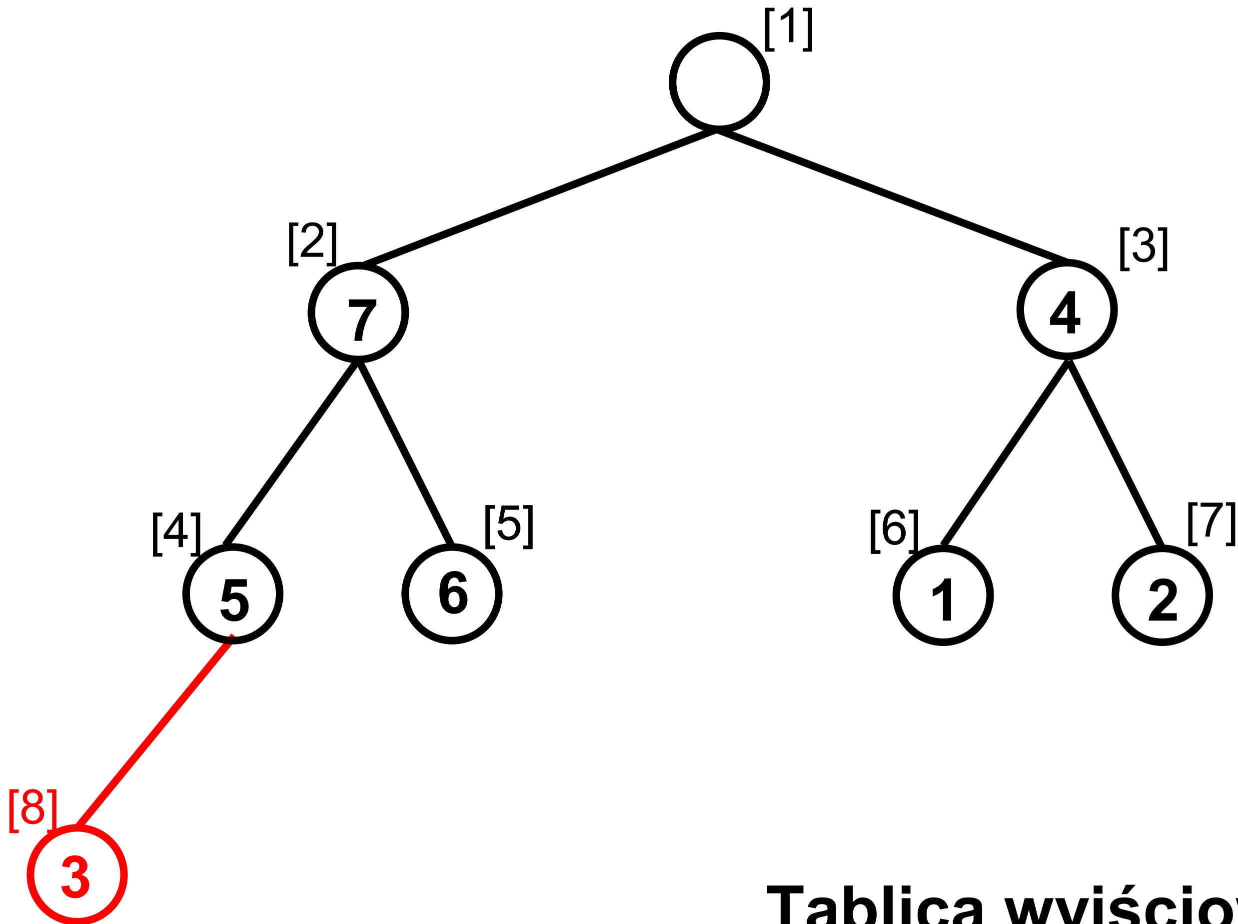


Tablica wyjściowa:

12	9	8							

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

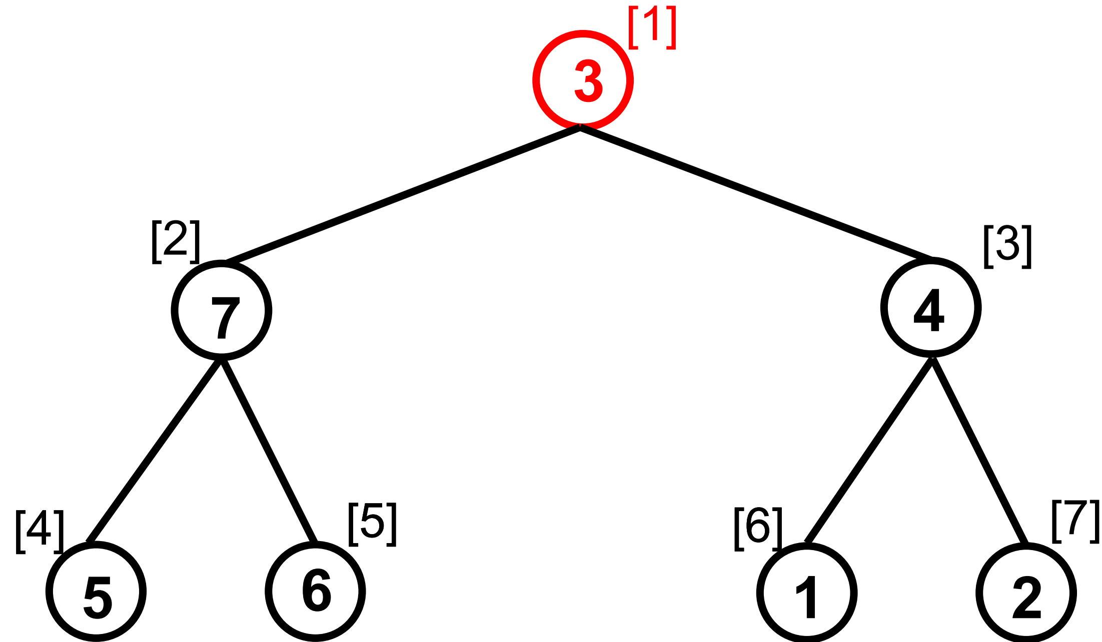


Tablica wyjściowa:

12	9	8							

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

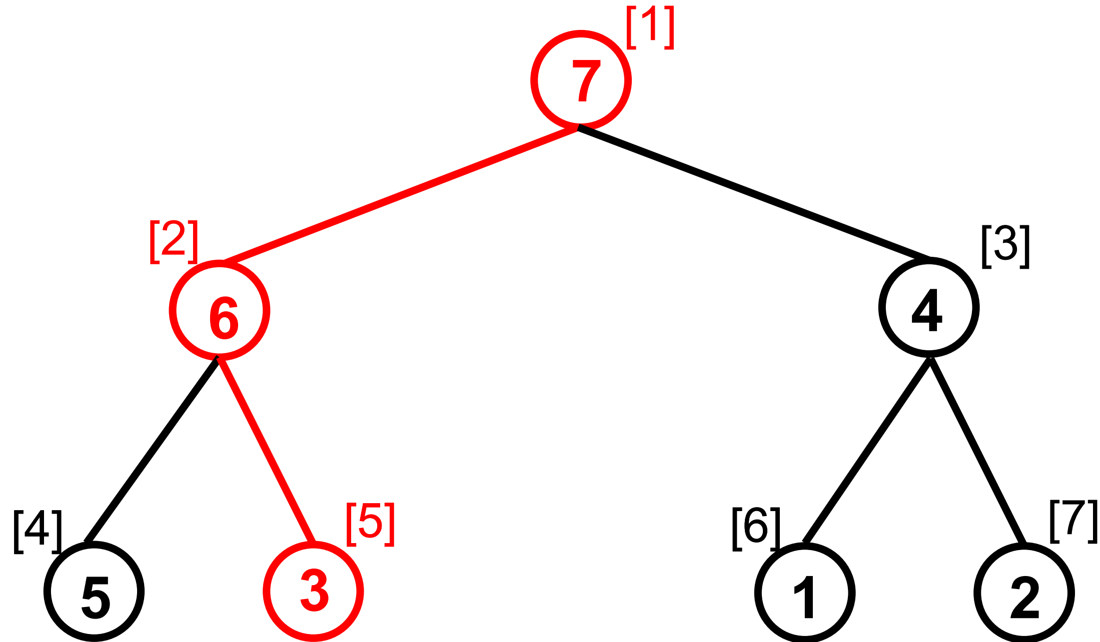


Tablica wyjściowa:

12	9	8							

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

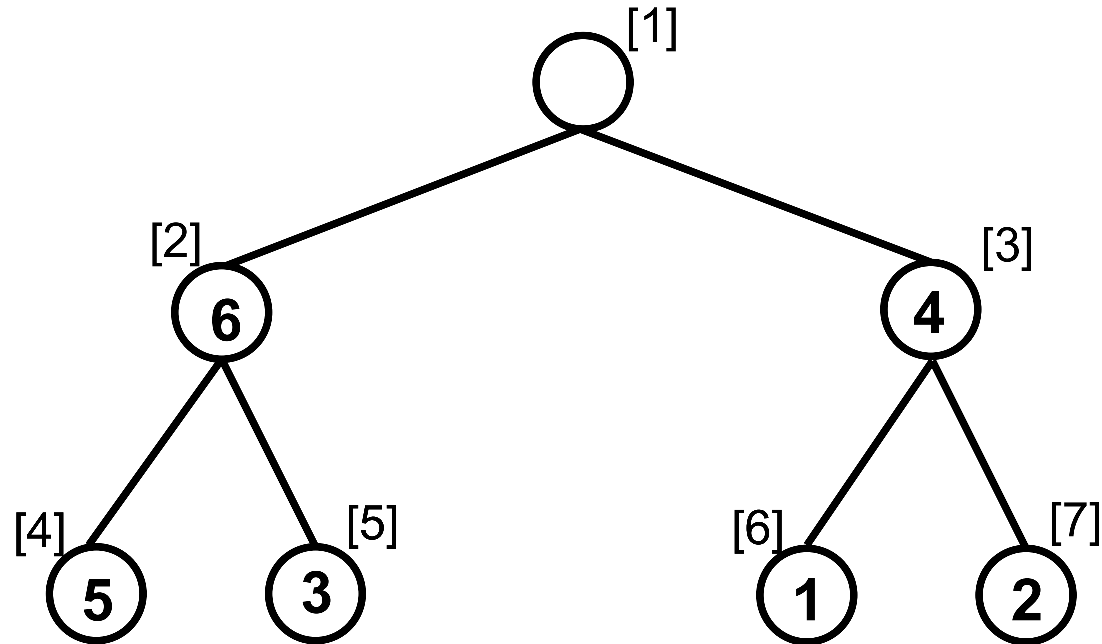


Tablica wyjściowa:

12	9	8							

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

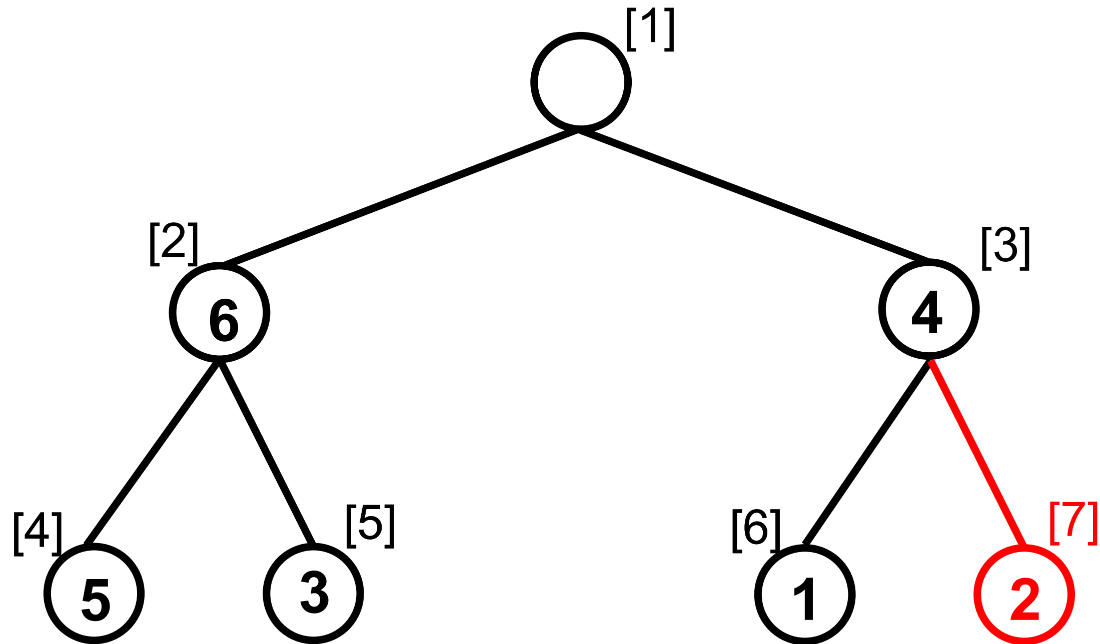


Tablica wyjściowa:

12	9	8	7						

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

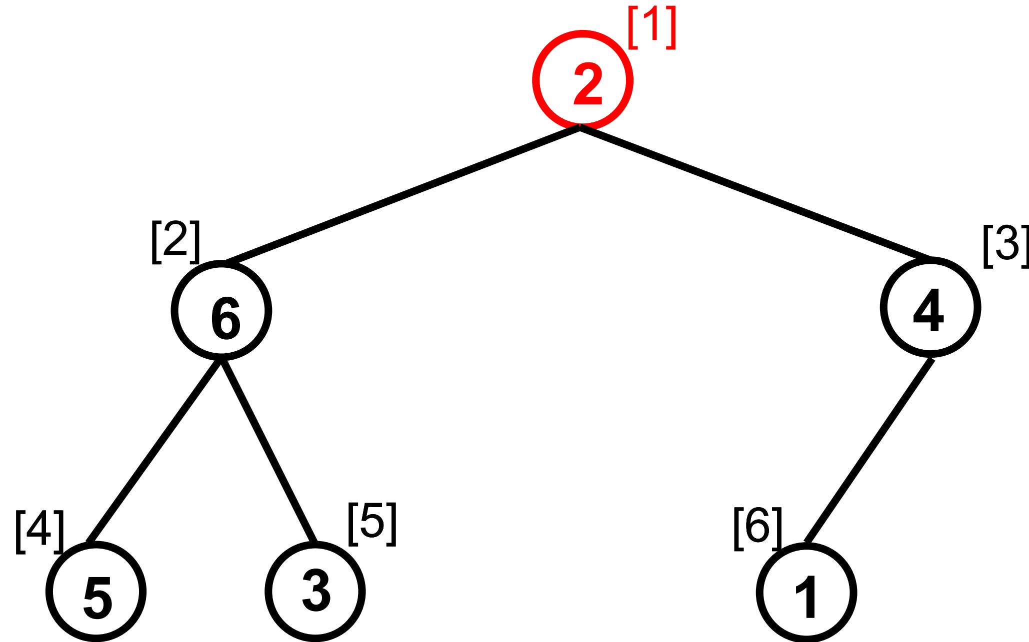


Tablica wyjściowa:

12	9	8	7						

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

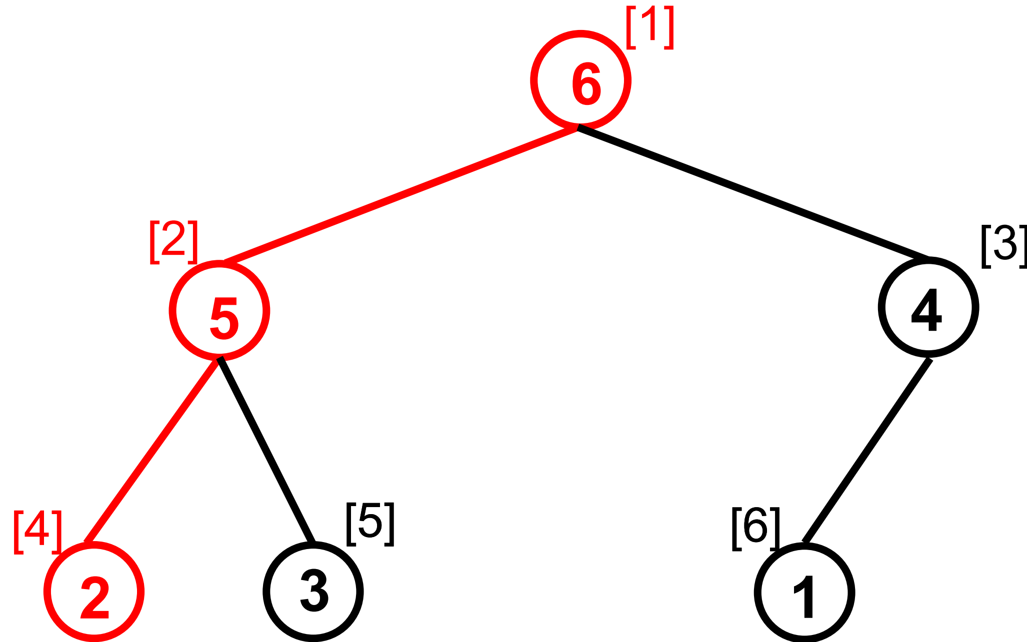


Tablica wyjściowa:

12	9	8	7						

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

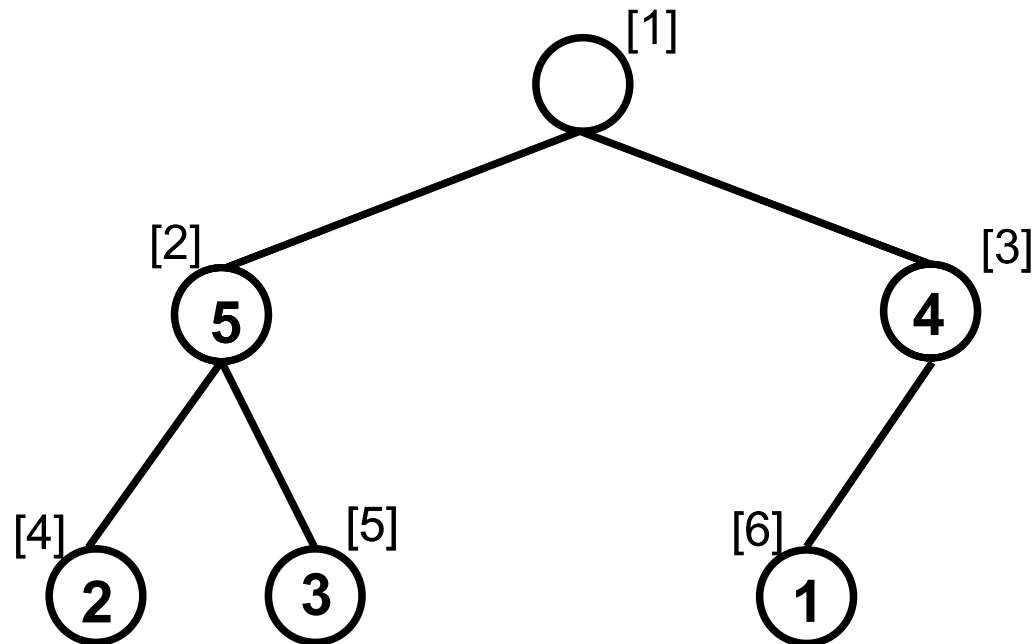


Tablica wyjściowa:

12	9	8	7						

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

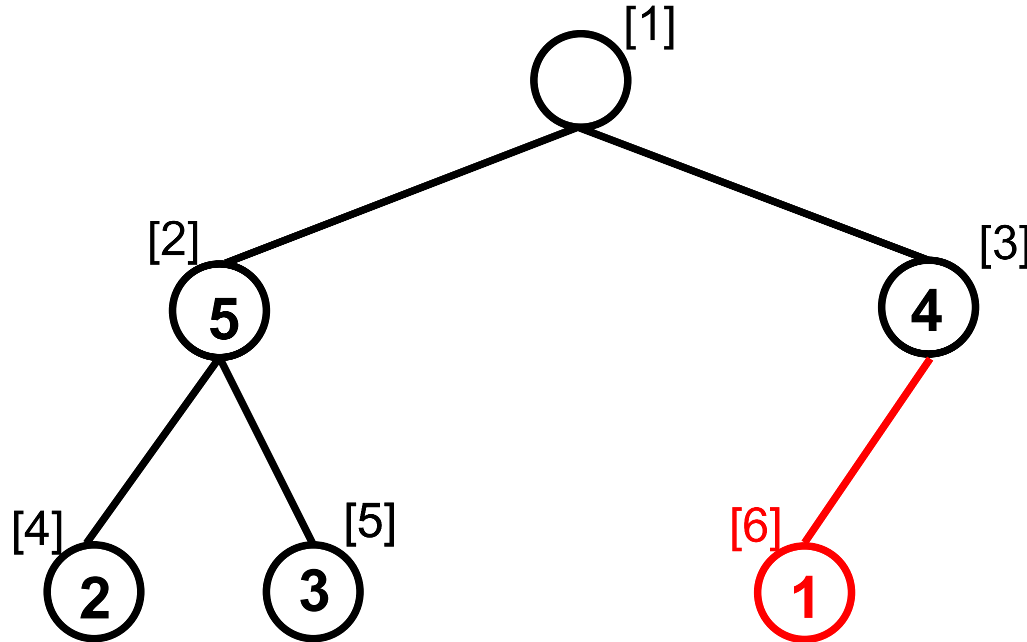


Tablica wyjściowa:

12	9	8	7	6					

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

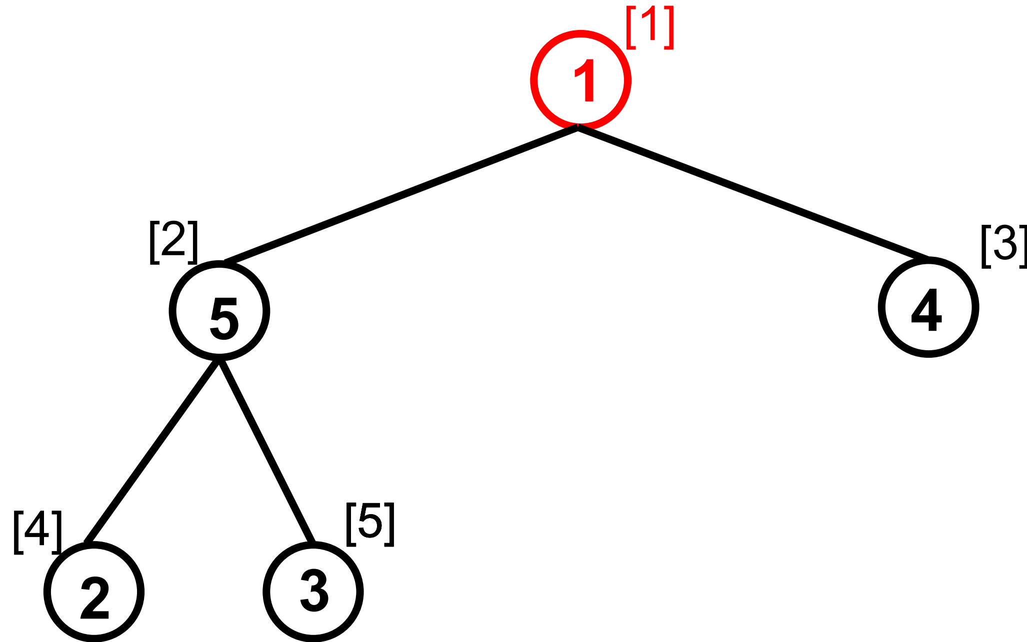


Tablica wyjściowa:

12	9	8	7	6					

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

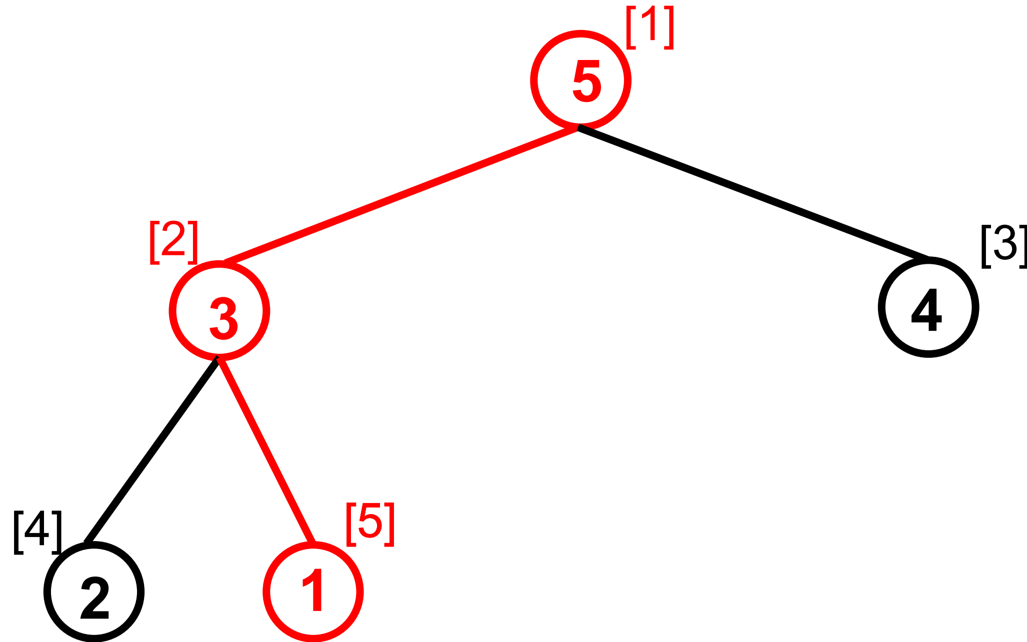


Tablica wyjściowa:

12	9	8	7	6					

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

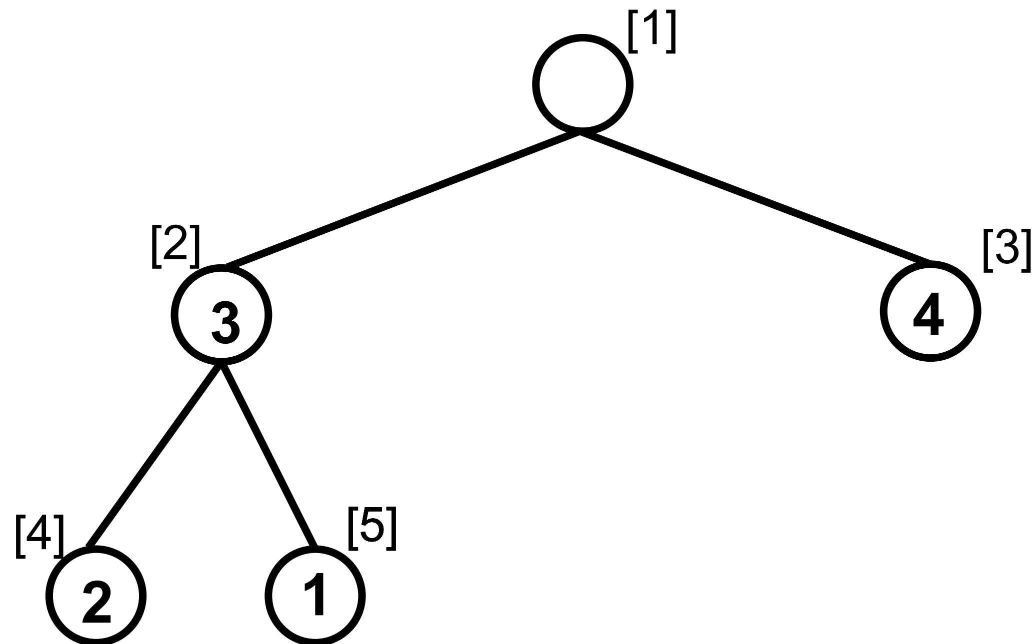


Tablica wyjściowa:

12	9	8	7	6					

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

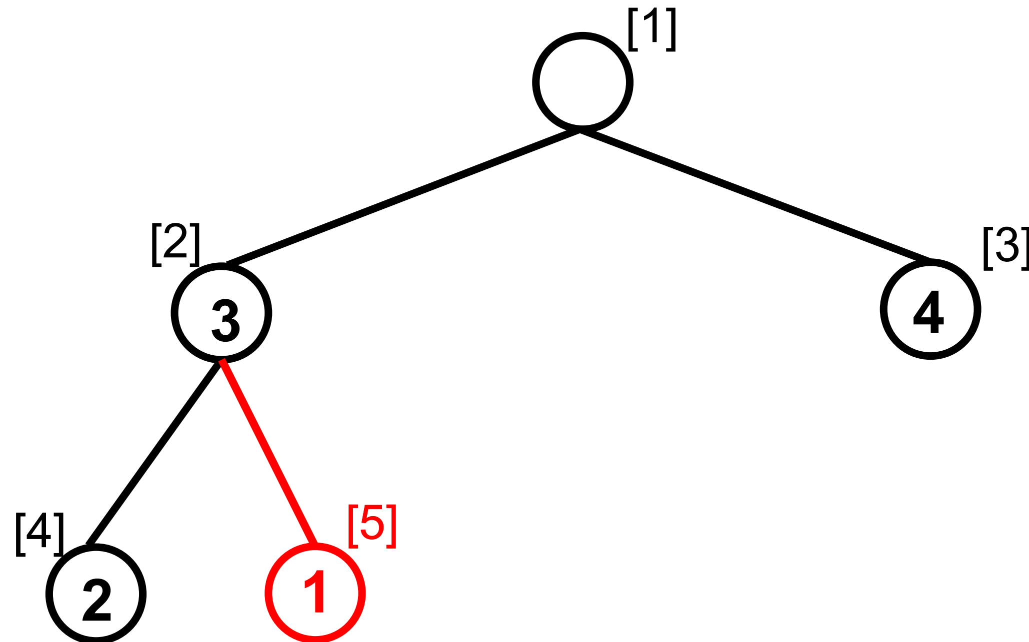


Tablica wyjściowa:

12	9	8	7	6	5				

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

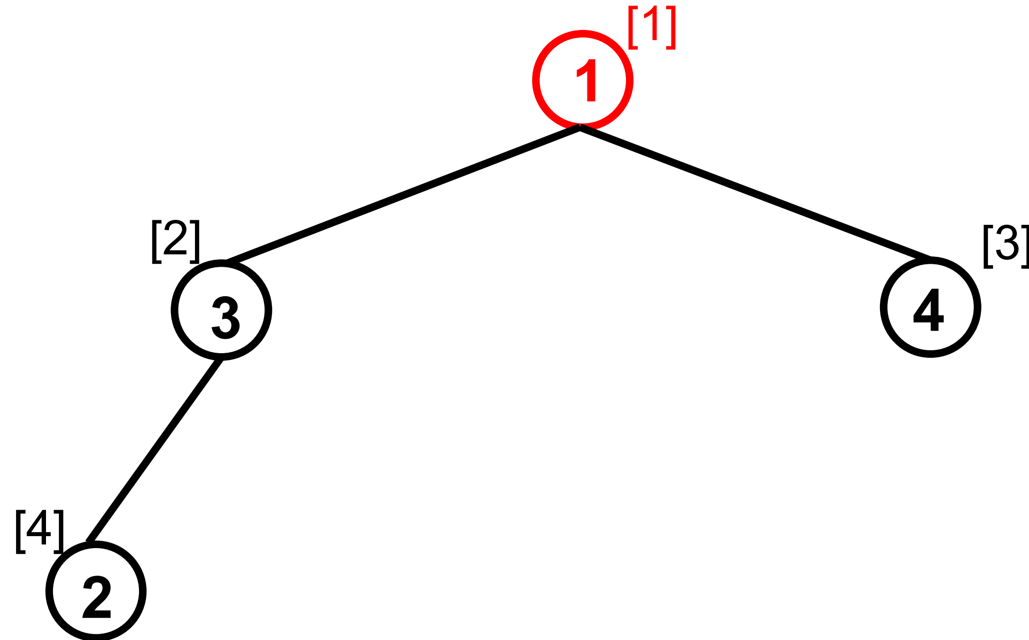


Tablica wyjściowa:

12	9	8	7	6	5				

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

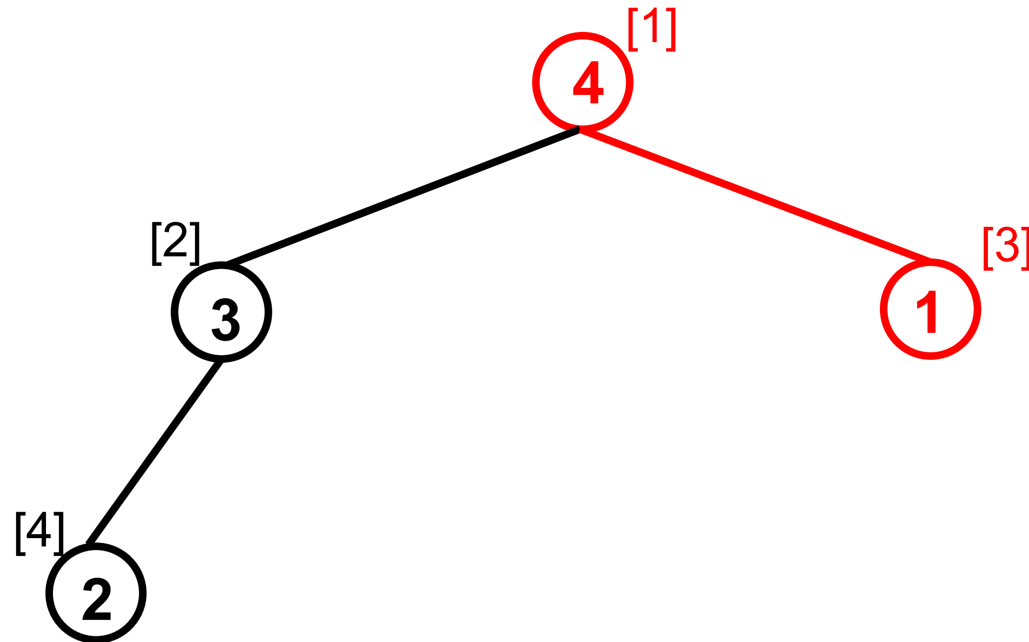


Tablica wyjściowa:

12	9	8	7	6	5				

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

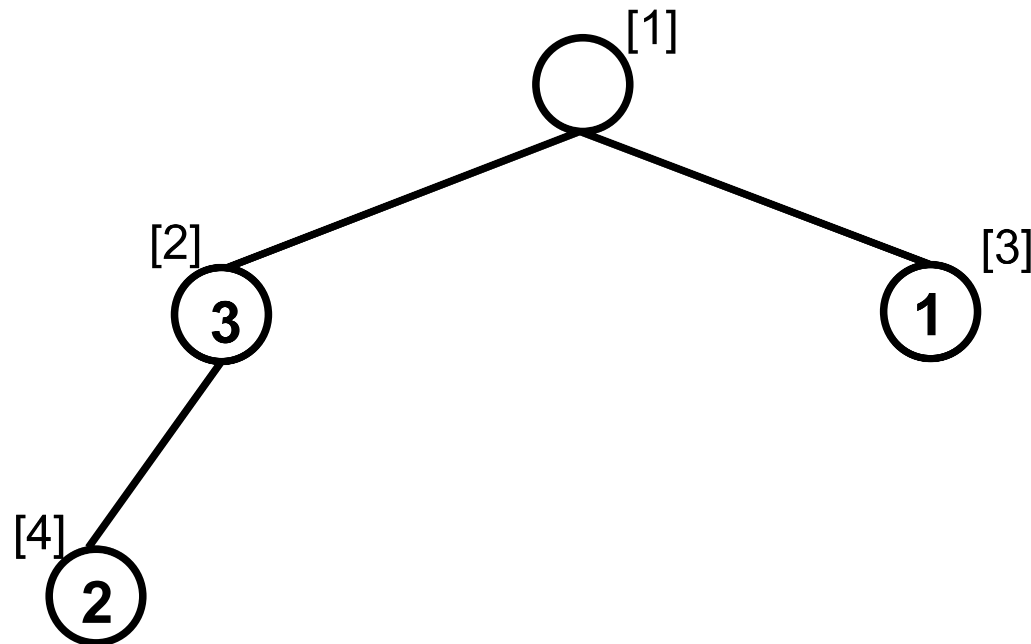


Tablica wyjściowa:

12	9	8	7	6	5				

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

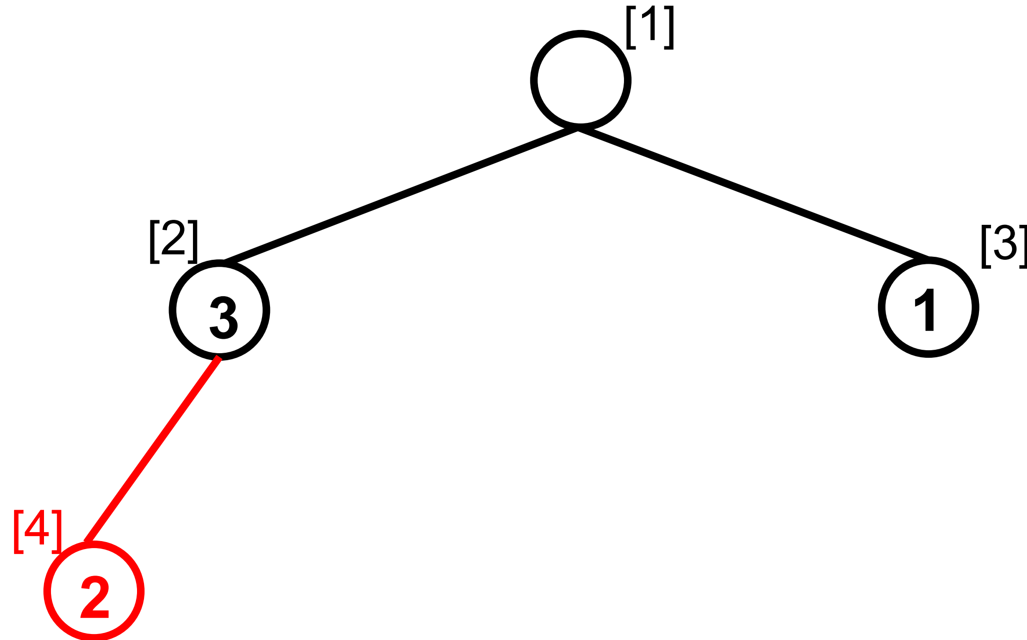


Tablica wyjściowa:

12	9	8	7	6	5	4			

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

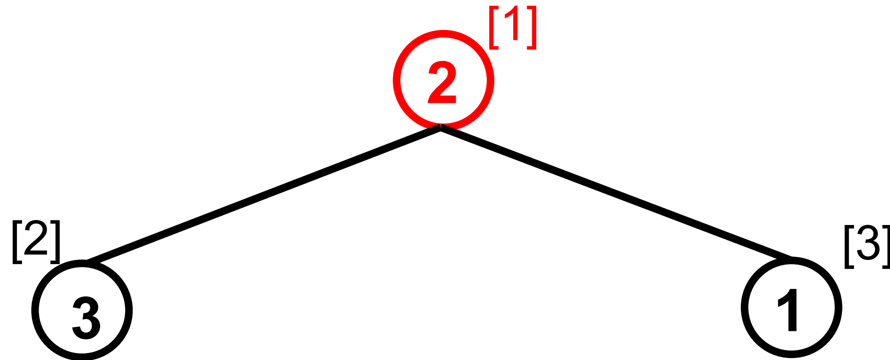


Tablica wyjściowa:

12	9	8	7	6	5	4			

Sortowanie przez kopcowanie:

przykład dla *n* = **10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

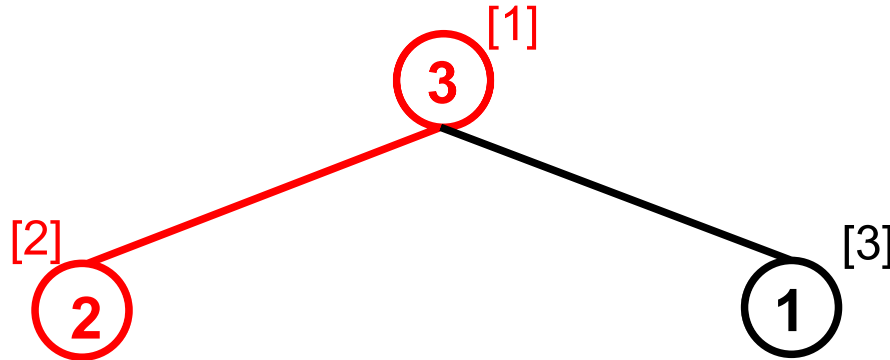


Tablica wyjściowa:

12	9	8	7	6	5	4			

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

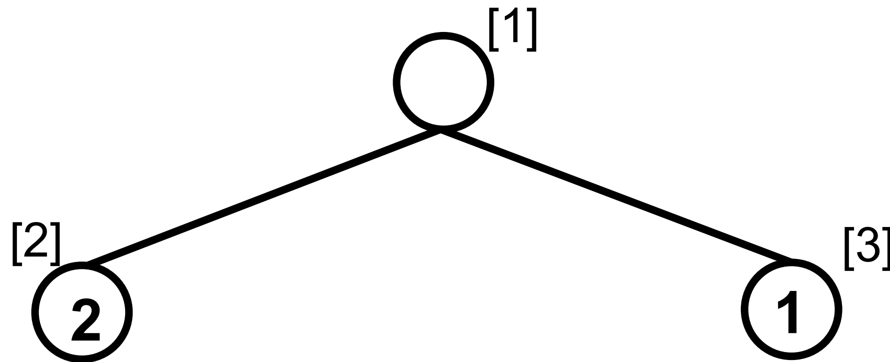


Tablica wyjściowa:

12	9	8	7	6	5	4			

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

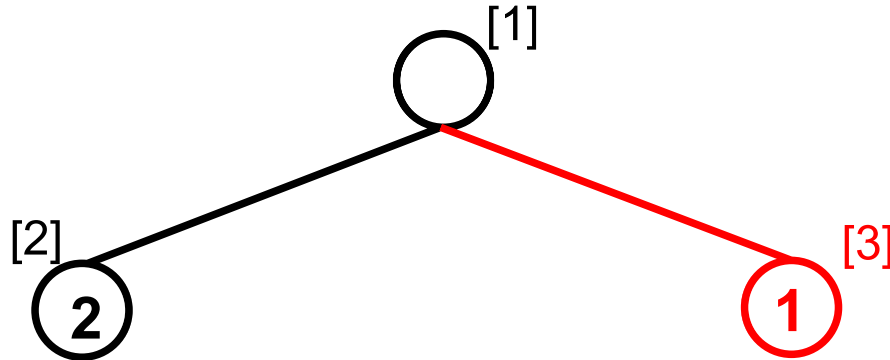


Tablica wyjściowa:

12	9	8	7	6	5	4	3		

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

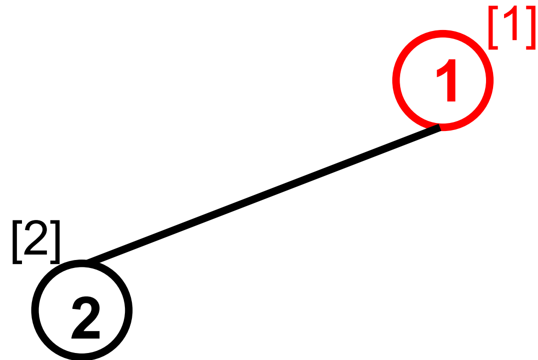


Tablica wyjściowa:

12	9	8	7	6	5	4	3		

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

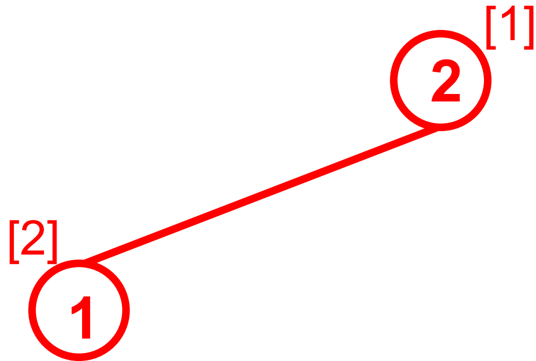


Tablica wyjściowa:

12	9	8	7	6	5	4	3		

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

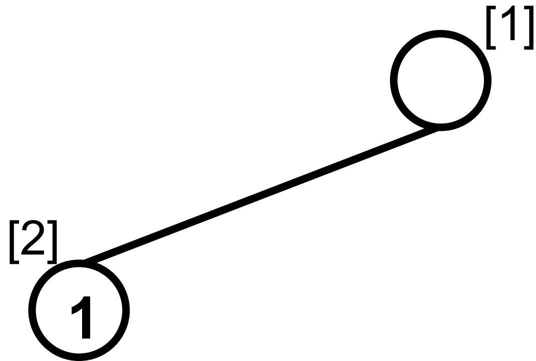


Tablica wyjściowa:

12	9	8	7	6	5	4	3		

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

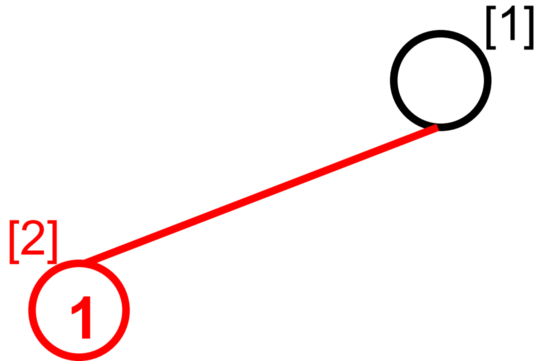


Tablica wyjściowa:

12	9	8	7	6	5	4	3	2	

Sortowanie przez kopcowanie:

przykład dla *n* **= 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



Tablica wyjściowa:

12	9	8	7	6	5	4	3	2	

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

[1]

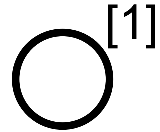
1

Tablica wyjściowa:

12	9	8	7	6	5	4	3	2	

Sortowanie przez kopcowanie:

przykład dla ***n* = 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**



Tablica wyjściowa:

12	9	8	7	6	5	4	3	2	1

Sortowanie przez kopcowanie:

przykład dla *n* **= 10** elementów: **{8, 3, 1, 5, 6, 9, 2, 12, 7, 4}**

KONIEC

Tablica wyjściowa:

12	9	8	7	6	5	4	3	2	1

Algorytm Quicksort

Zasadę działania tego algorytmu najlepiej przedstawić przy pomocy następującej procedury rekurencyjnej:

```
int B[n];

void Quicksort(int* B,int p,int r)
{ if(p<r)
  { q=Partition(B,p,r);
    Quicksort(B,p,q);
    Quicksort(B,q+1,r);
    };
  };
```

Kluczowym elementem algorytmu jest procedura `Partition`.

Procedura ta ma za zadanie taki **podział** obszaru **tablicy** $\boldsymbol{B}$ od indeksu p do indeksu q, aby **przed elementem** $\boldsymbol{q}$ znalazły się elementy **nie większe** od q-tego, a **po** tym elemencie – **nie mniejsze**.

Aby **posortować całą tablicę** B, należy wywołać `Quicksort(B,1,n)`.

Zatem cała tablica dzielona jest **na 2 części**: w pierwszej z nich znajdą się elementy nie większe niż w drugiej. Następnie każda z tych części dzielona jest na dwie mniejsze o takiej samej własności, itd.

Algorytm **kończy działanie**, gdy osiągnie wyłącznie obszary **1- lub 2-elementowe**, i w każdym z 2-elementowych mniejszy element zostanie ustawiony przed większym (oczywiście elementy równe mogą pozostać w dowolnej kolejności).

Jedną z decyzji, jaką należy podjąć przy konstruowaniu procedury `Partition` jest **wybór elementu** $\boldsymbol{q}$, względem którego będzie dokonany podział elementów $B[p] \ldots B[r]$. Jednym z najprostszych sposobów jest wybór elementu **skrajnego**, np. $\boldsymbol{B[r]}$.

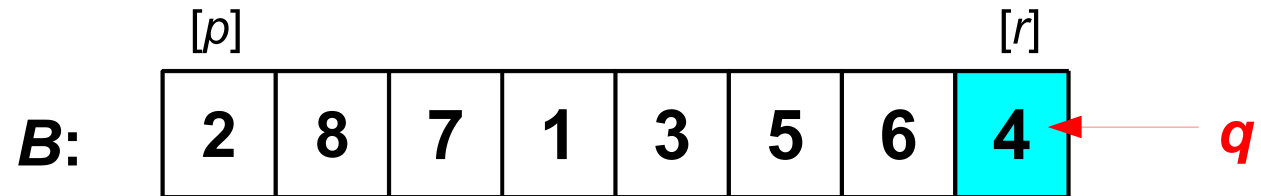
Następnie analizowane są kolejno wszystkie **pozostałe elementy** i umieszczane w pierwszym albo w drugim obszarze (w zależności od tego, czy dany element jest odpowiednio mniejszy czy większy od q-tego). Najlepiej, jeśli powyższą operację wykonamy *w miejscu*.

Zilustrujmy powyższe rozwiązanie na przykładzie.

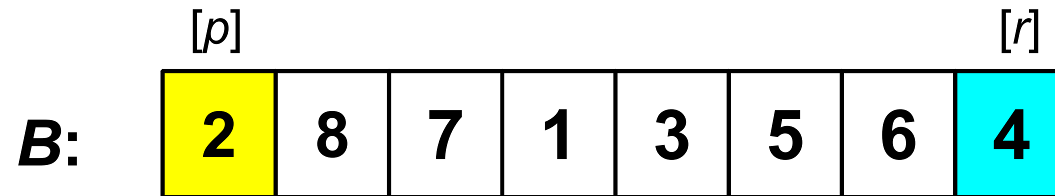
Partition **(*B*, *p*, *r*)**

	[*p*]							[*r*]
***B*:**	**2**	**8**	**7**	**1**	**3**	**5**	**6**	**4**

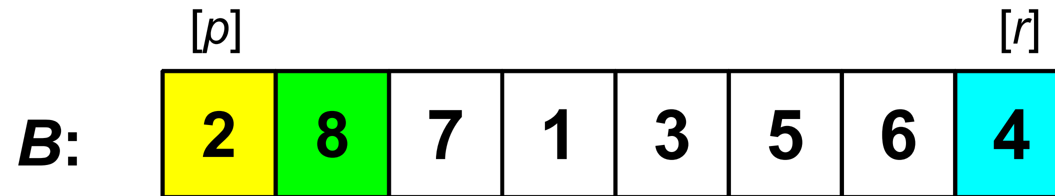
Partition (*B*, *p*, *r*)



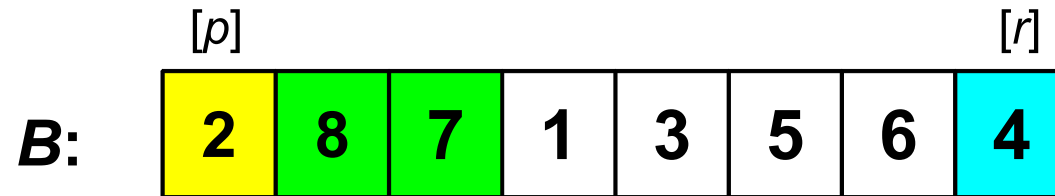
Partition (B, p, r)



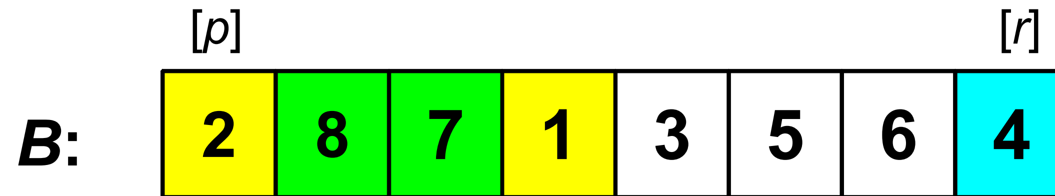
Partition **(*B*, *p*, *r*)**



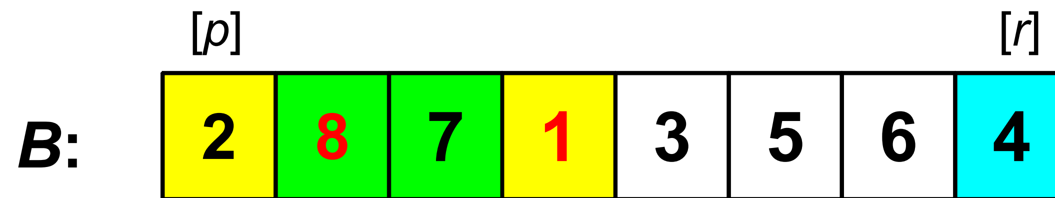
***Partition* (*B*, *p*, *r*)**



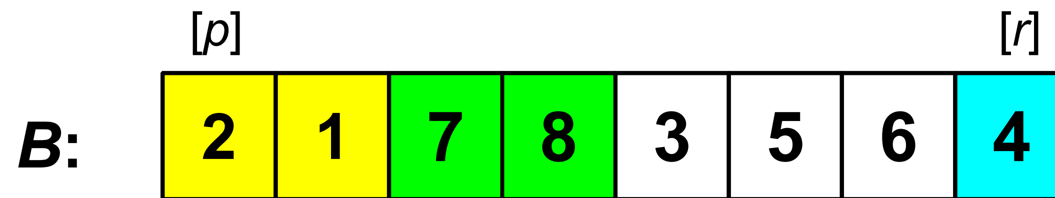
Partition (B, p, r)



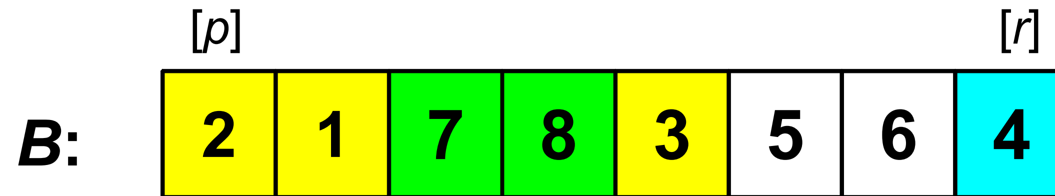
***Partition* (*B*, *p*, *r*)**

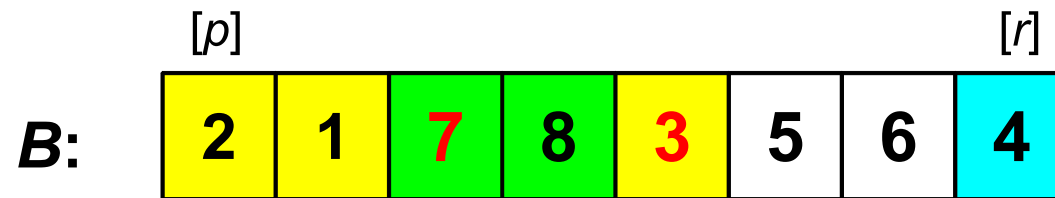


Partition (*B*, *p*, *r*)



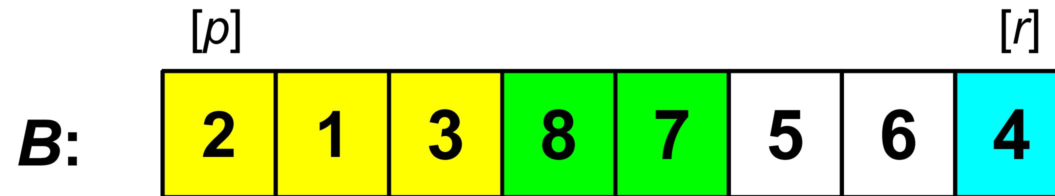
Partition (B, p, r)



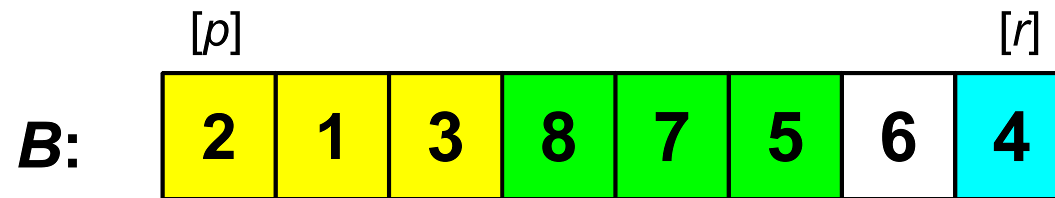


Partition (B, p, r)
[p]
[r]
B:
2
1
7
8
3
5
6
4

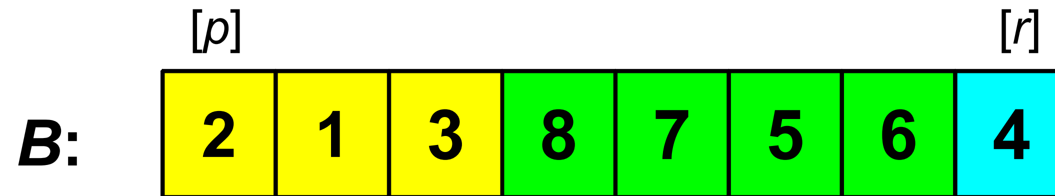
Partition (*B*, *p*, *r*)



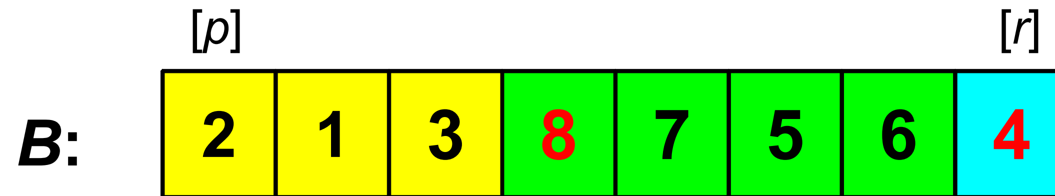
Partition (*B*, *p*, *r*)



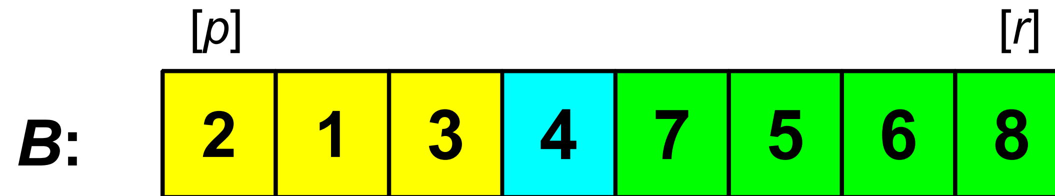
Partition (*B*, *p*, *r*)



Partition (B, p, r)



***Partition* (*B*, *p*, *r*)**

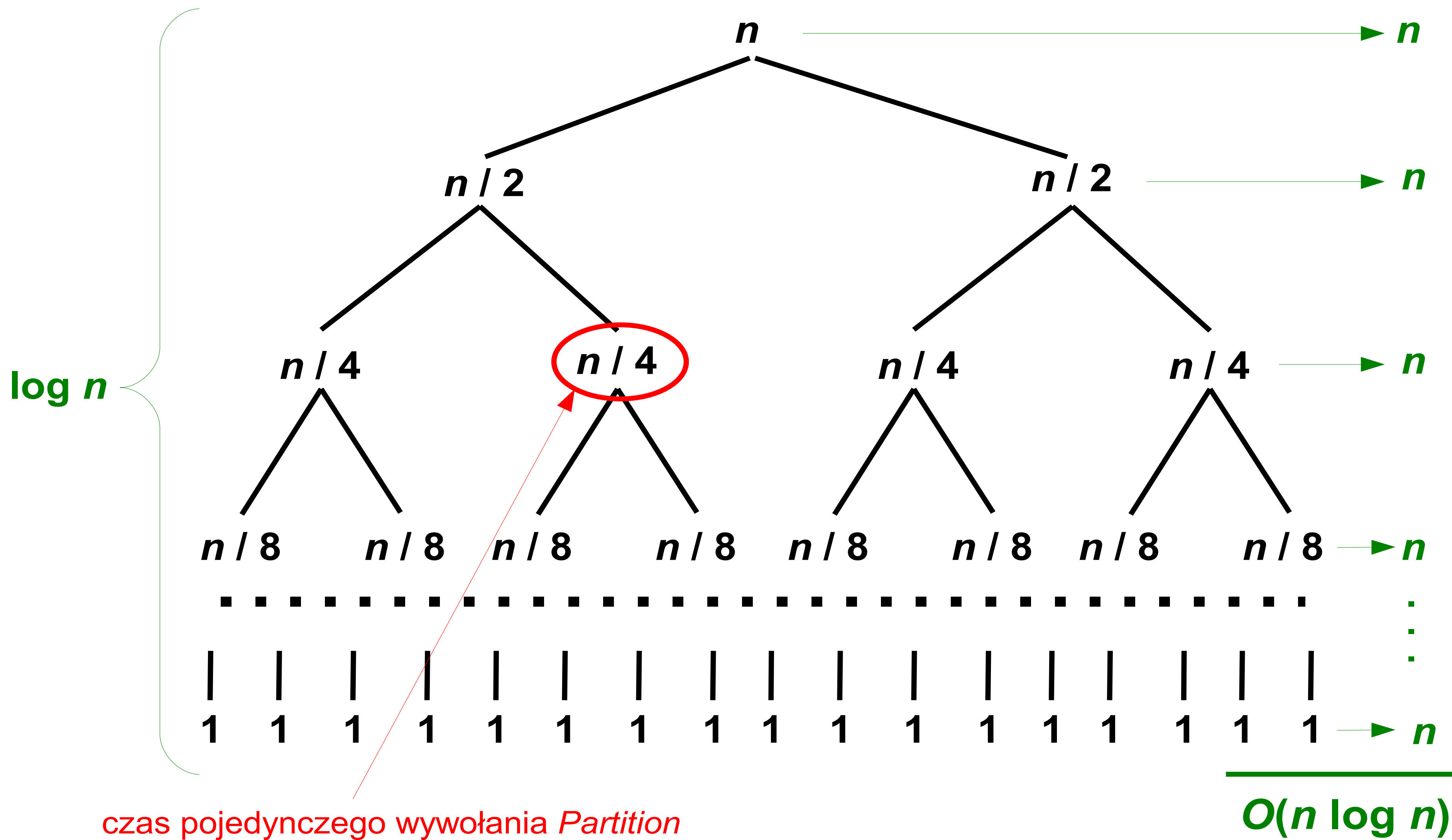


Złożoność obliczeniowa procedury `Partition` wynosi $O(n)$.

A jaka jest złożoność całego algorytmu?

Zależy to ściśle od dokonywanych podziałów.

Jeśli **za każdym razem** wybierany jest do podziału taki element, który dzieli dany obszar na **2 równe części**, to uzyskamy złożoność $O(n \log n)$, co uzasadnia poniższy rysunek.


n
n / 2
n / 2
n / 4
n / 4
n / 4
n / 4
n / 8
n / 8
n / 8
n / 8
n / 8
n / 8
n / 8
n / 8
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
n
n
n
n
n
log n
O(n log n)
czas pojedynczego wywołania Partition

Jednak jeśli do podziału wybierany jest **zawsze** taki element, że **jedna część** dzielonego obszaru **zawiera 1 element**, to uzyskamy wynik $\underline{O(n^2)}$.

Na koniec należy zauważyć, że wystarczy, aby **podział** był dokonywany **w proporcji 9/1**, aby złożoność algorytmu Quicksort wynosiła $O(n \log n)$.

Czy można skonstruować algorytm **sortowania** o złożoności **mniejszej niż** $O(n \log n)$?

Wszystkie dotychczas przedstawione algorytmy są algorytmami działającymi *na zasadzie porównań*. Można wykazać, że algorytmy tego typu **nie mogą mieć złożoności mniejszej niż** $O(n \log n)$. Zatem konstruując algorytm sortowania przez kopcowanie osiągnęliśmy dolną granicę.

Jednak przy pewnych **dodatkowych założeniach** (np. odnośnie wartości danych wejściowych) można uzyskać **lepszy rezultat**.

Sortowanie przez zliczanie

W algorytmie tym mamy **dodatkową tablicę**, C, o rozmiarze równym maksymalnej wartości spośród danych wejściowych, powiedzmy m. Wszystkie komórki tej tablicy są zainicjowane wartością 0.

Działanie algorytmu sprowadza się do **sprawdzenia każdej wartości** tablicy wejściowej i **zwiększenie o 1** tej komórki tablicy C, która odpowiada wartości analizowanej danej (np., jeśli w danej komórce tablicy wejściowej odczytamy 5, to inkrementujemy komórkę $C[5]$).

Po przeanalizowaniu wszystkich n danych wejściowych, w tablicy C mamy informację, ile wśród tych danych było wartości 1, ile wartości 2, itd. Tablicę wyjściową wypełniamy więc tak, że do pierwszych $C[m]$ komórek wpisujemy m, do następnych $C[m-1]$ komórek – $m-1$, itd., a do ostatnich $C[1]$ komórek wpisujemy wartość 1.

Złożoność obliczeniowa algorytmu wynosi $\underline{\boldsymbol{O(n+m)}}$:

- każdą daną wejściową analizujemy 1 raz,
- za każdym razem inkrementacja wybranej komórki tablicy C zajmuje $O(1)$,
- na koniec wypełnienie tablicy wyjściowej wymaga „przejścia" po wszystkich elementach tablicy C.

Dla $m = O(n)$ algorytm ma więc złożoność $\boldsymbol{O(n)}$.

Sortowanie pozycyjne

Sortowanie $\boldsymbol{n}$ **liczb** $\boldsymbol{d}$**-cyfrowych** polega na wykonaniu d sortowań – najpierw wg *najmniej znaczącej* cyfry, później bardziej znaczącej, itd. Na końcu sortujemy wg najbardziej znaczącej cyfry. Trzeba tylko uwzględniać kolejność z poprzedniego sortowania, jeśli cyfry na bieżącej pozycji są jednakowe.

Ponieważ cyfry w systemie np. dziesiętnym mają maksymalnie wartość 9, możemy z powodzeniem wykorzystać do sortowania wg poszczególnych pozycji algorytm sortowania przez zliczanie.

Złożoność obliczeniowa wynosi wtedy $\boldsymbol{O(dn + dm)}$, gdzie m jest maksymalną wartością cyfr (np. $m = 9$ dla systemu dziesiętnego). Jeśli d jest stałą, a $m = O(n)$, to otrzymujemy złożoność $\boldsymbol{O(n)}$.

Sortowanie kubełkowe

Przy założeniu, że **dane wejściowe są z określonego zakresu**, np. [0,1), dzielimy ten przedział na n obszarów o jednakowym rozmiarze (tworzymy n odpowiadających im „kubełków”). Każda dana wpada więc do odpowiedniego kubełka (jeśli dane wejściowe są z rozkładu jednostajnego, w każdym kubełku znajdzie się mniej więcej tyle samo elementów).

Następnie dane **wewnątrz każdego kubełka sortujemy**, np. algorytmem przez wstawianie, i złączamy kubełki razem, otrzymując poprawny ciąg wynikowy.

Można wykazać, że **złożoność obliczeniowa** wynosi $\boldsymbol{O(n)}$.