Politechnika Wrocławska

Struktury danych i złożoność obliczeniowa Wykład 1.

Prof. dr hab. inż. Jan Magott

Formy zajęć:

- Wykład 1 godz.,
- Ćwiczenia 2 godz.,
- Projekt 1 godz. .

Strona kursu:

http://www.zio.iiar.pwr.wroc.pl/sdizo.html

Struktury danych wchodzące w zakres kursu:

- Tablice,
- Listy,
- Kolejki,
- Stosy,
- Kopce,
- Grafy,
- Drzewa binarne,
- Tablice haszujące.

Pięć następujących slajdów pochodzi
z Polskich Ram Kwalifikacyjnych kursu

Struktury danych i złożoność obliczeniowa

Forma zajęć - wykład		
Wy1	Polskie Ramy Kwalifikacyjne przedmiotu. Literatura. Podstawowe zasady analizy algorytmów: poprawność i skończoność (asercje i niezmienniki pętli). Struktury danych: stosy, kolejki, listy, kopce (sortowanie przez kopcowanie) w implementacji tablicowej.	2
Wy2	Złożoność obliczeniowa (klasy złożoności czasowej i pamięciowej), koszt zamortyzowany.	2
Wy3	Podstawowe techniki budowy algorytmów: metoda „dziel i zwyciężaj”, metoda zachłanna, transformacyjna konstrukcja algorytmu.	2
Wy4	Kodowanie dziesiętne, dwójkowe i jedynkowe danych wejściowych problemu. „Rozsądna” reguła kodowania.	1
Wy5	Algorytmy grafowe: reprezentacja grafów, metody przeszukiwania, minimalne drzewa rozpinające, problemy ścieżkowe.	2
Wy6	Problemy „ łatwe” i „trudne”. Problemy optymalizacyjne i decyzyjne.	1
Wy7	Model obliczeń RAM. Deterministyczne jednotaśmowe i k-taśmowe maszyny Turinga. Przykładowe programy dla tych maszyn.	1
	Niedeterministyczna maszyna Turinga. Twierdzenie o relacji między Niedeterministyczną a Deterministyczną Maszyną	

Wy4	Kodowanie dziesiętne, dwójkowe i jedynkowe danych wejściowych problemu. „Rozsądna” reguła kodowania.	1
Wy5	Problemy „ łatwe” i „trudne”. Problemy optymalizacyjne i decyzyjne. Algorytmy grafowe: reprezentacja grafów, metody przeszukiwania, minimalne drzewa rozpinające, problemy ścieżkowe.	2
Wy6	Problemy „ łatwe” i „trudne”. Problemy optymalizacyjne i decyzyjne.	1
Wy7	Model obliczeń RAM. Deterministyczne jednotaśmowe i k-taśmowe maszyny Turinga. Przykładowe programy dla tych maszyn.	1
Wy8	Niedeterministyczna maszyna Turinga. Twierdzenie o relacji między Niedeterministyczną a Deterministyczną Maszyną Turinga. Klasy P i NP problemów decyzyjnych. Transformacja wielomianowa. Problem NP-zupełny. Dowodzenie NP-zupełności problemów decyzyjnych.	1
Wy9	Dowody NP-zupełności wybranych problemów.	1
Wy10	Kolokwium	2
	Suma godzin	15

Forma zajęć - ćwiczenia		Liczba godzin
Ćw1	Zajęcia wprowadzające. Omówienie programu, podanie wymagań.	1
Ćw2	Podstawowe zasady analizy algorytmów	2
Ćw3	Podstawowe struktury danych: kolejki, listy, stosy, kopce	3
Ćw4	Struktury drzewiaste: BST, AVL, B-R, B-drzewo	5
Ćw5	Algorytmy sortowania np. Insertion-, Quick-, Merge-, Heap-, Radix-	3
Ćw6	Tablice haszujące	2
Ćw7	Algorytmy wyszukiwania wzorców	1
Ćw8	Algorytmy grafowe: reprezentacja grafów, metody przeszukiwania, minimalne drzewa rozpinające, problemy ścieżkowe	6
Ćw9	Wybrane problemy złożoności obliczeniowej: model maszyny Turinga (DTM, NDTM), redukcja wielomianowa	5
Ćw10	Kolokwium	2
	Suma godzin	30

Forma zajęć – projekt		Liczba godzin
Pr1	Sprawy organizacyjne, omówienie zadań projektowych, wymagań oraz warunków zaliczenia.	2
Pr2	Badanie efektywności operacji na danych w podstawowych strukturach danych.	5
Pr3	Badanie efektywności wybranych algorytmów grafowych np. w zależności od rozmiaru, struktury czy sposobu reprezentacji grafu.	8
	Suma godzin	15

Oceny (F – formująca (w trakcie semestru), P – podsumowująca (na koniec semestru)		Sposób oceny osiągnięcia efektu kształcenia
F1		Odpowiedzi ustne, Wyniki kolokwiów cząstkowych.
F2		Wyniki realizacji zadań projektowych
F3		Kolokwium pisemne
$P = 0,5 \cdot F1 + 0,25 \cdot F2 + 0,25 \cdot F3$ jeśli $(3 \leqslant F1$ and $3 \leqslant F2$ and $3 \leqslant F3)$		

Literatura

[1] T. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, „Wprowadzenie do algorytmów”, WNT 2007.

[2] J. Błażewicz, „Problemy optymalizacji kombinatorycznej”, PWN, Warszawa 1996.

[3] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., New York, 1979.

Plan 1. wykładu:

1. Podstawowe zasady analizy algorytmów: poprawność, skończoność,
2. Podstawowe struktury danych: stosy, kolejki, kopce zbudowane z użyciem tablic,

Podstawowe zasady analizy algorytmów: poprawność, skończoność

Algorytm to procedura do rozwiązywania problemu.

Algorytm może być wyrażony w:

- Języku naturalnym,
- Języku formalnym,
- Języku zawierającym konstrukcje języka naturalnego i formalnego,
- Schematem blokowym,
- Diagramem aktywności (czynności) języka UML,
- Pseudokodzie,
- Języku programowania.

Podstawowe zasady analizy algorytmów: poprawność, skończoność

Algorytm rozwiązywania danego problemu obliczeniowego jest **poprawny**, jeśli dla każdej instancji (egzemplarza) tego problemu zatrzymuje się i daje dobry wynik.

Podstawowe zasady analizy algorytmów: poprawność, skończoność

Asercja 0

Instrukcja 1

Asercja 1

Instrukcja 2

Asercja 2

. . .

Asercja n-1

Instrukcja n

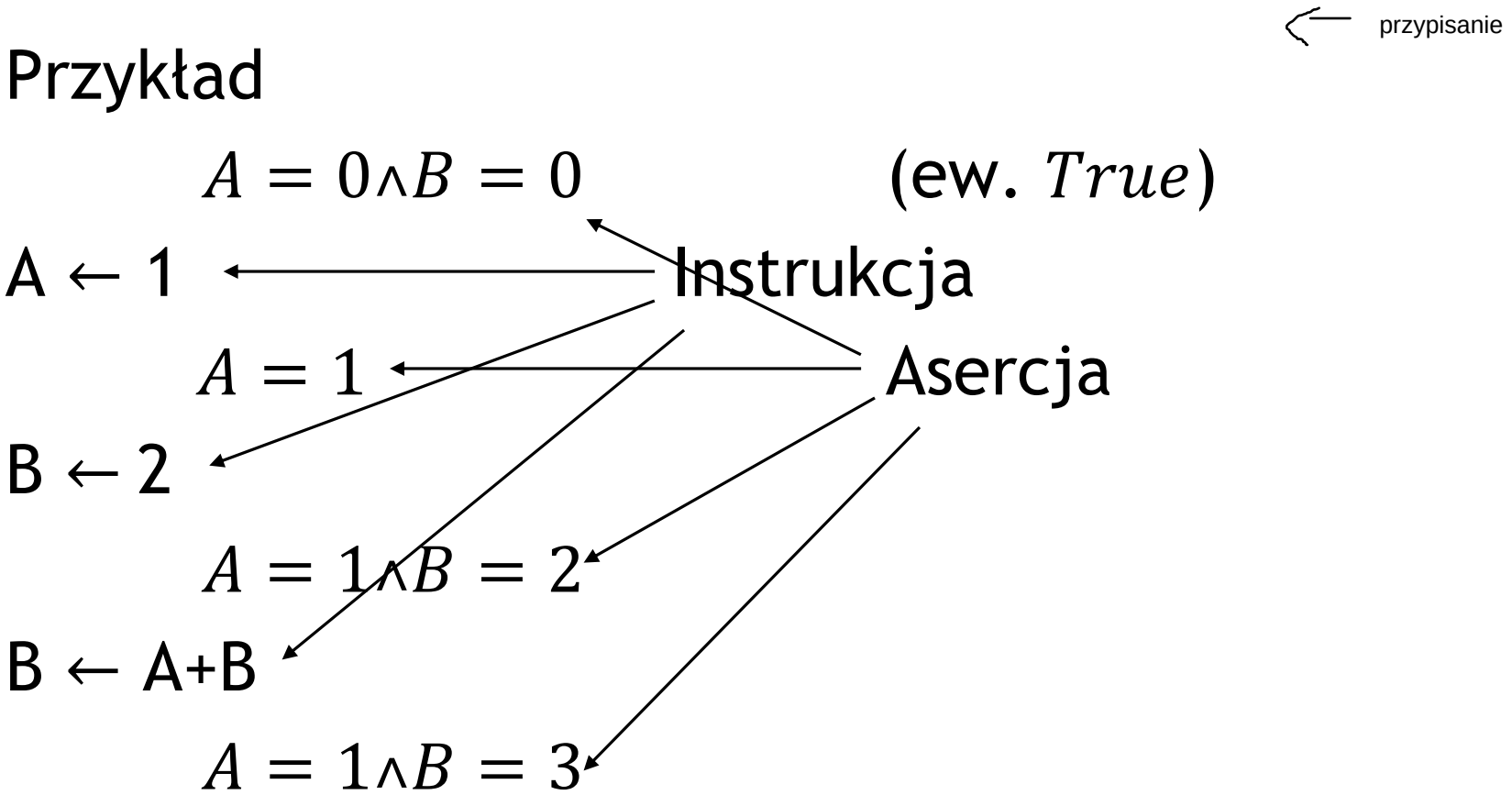
Asercja n

Asercja – warunek charakteryzujący stan zmiennych programu w pewnym punkcie wykonania

$$(\forall i \epsilon \overline{\{0, n-1\}})(Asercja\ i \overset{Instr\ i+1}{\Longrightarrow} Asercja\ i+1)$$

przed kad instrukcj i po kadej jest asercja

Podstawowe zasady analizy algorytmów: poprawność, skończoność



Podstawowe zasady analizy algorytmów: poprawność, skończoność

Poprawność algorytmów

Niezmiennik (ang. invariant) pętli jest warunkiem, który:

Inicjowanie: Jest prawdziwy przed pierwszą iteracją pętli,

Niezmienniczość: Jeśli jest prawdziwy przed pewną iteracją pętli, to jest prawdziwy po niej,

Kończenie: Po zakończeniu pętli, z niezmiennika można udowodnić poprawność algorytmu pętli.

Własność stopu: dla poprawnych danych wejściowych algorytm zatrzymuje się w skończonym czasie.

Podstawowe zasady analizy algorytmów: poprawność, skończoność

Niezmiennik sumowania n liczb zawartych w tablicy A

$SUM(A, n, S)$

$S \leftarrow 0$

for $i \leftarrow 1$ **to** n

do $S \leftarrow S + A[i]$

Niezmiennik $S = \sum_{j=1}^{i-1} A[j]$ (przed $i-$tym wykonaniem),

Inicjowanie: $S = \sum_{j=1}^{1-1} A[j] = \sum_{j=1}^{0} A[j] = 0,$

Niezmienniczość: $\sum_{j=1}^{i-1} A[j]$ przed $i-$tym wykonaniem $\Rightarrow$

$\sum_{j=1}^{i} A[j]$ przed $(i+1)-$tym wykonaniem,

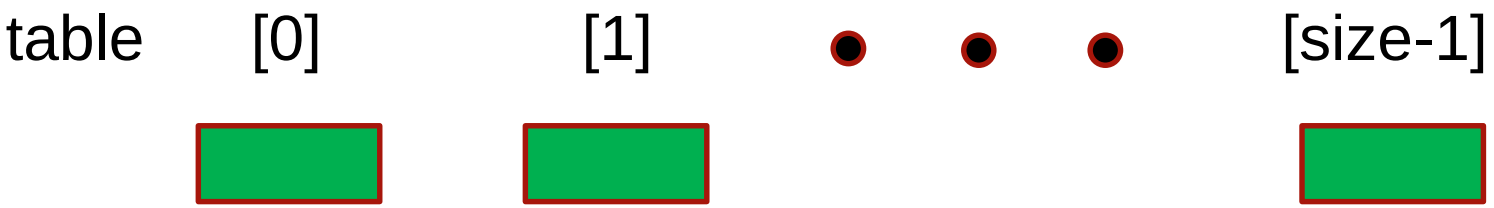
Kończenie: $S = \sum_{j=1}^{n} A[j]$ po zakończeniu pętli.

Podstawowe zasady analizy algorytmów: poprawność, skończoność

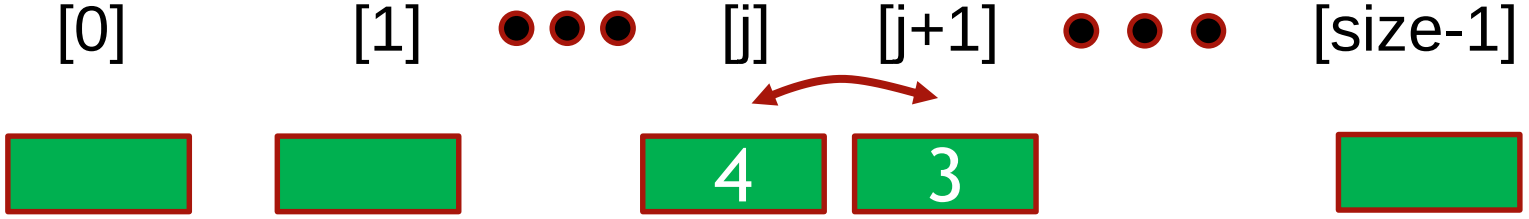
Algorytm sortowania bąbelkowego niemalejąco w kodzie C

naturalnie jest zrobi z uyciem 2 niezmienników

```
void bubblesort(int table[], int size)
{
        int i, j, temp;
        for (i = 0; i<size; i++)
    {
                for (j=0; j<size-1-i; j++)
                {
                        if (table[j] > table[j+1])
                        {
                                temp = table[j+1];
                                table[j+1] = table[j];
                                table[j] = temp;
                        }
                }
    }
}
```



W języku C: i++ jest wykonywane po wykonaniu treści pętli.



Podstawowe zasady analizy algorytmów: poprawność, skończoność

Niezmiennik pętli wewnętrznej: Przed wykonaniem pętli wewnętrznej dla zmiennej i w pętli zewnętrznej i j w wewnętrznej, gdzie $j < size - 1 - i$, liczba na pozycji j jest nie mniejsza niż poprzedzające ją.

Inicjowanie:Przed wykonaniem pętli wewnętrznej dla zmiennej i w pętli zewnętrznej i zmiennej $j = 0$ w wewnętrznej, gdzie $j < size - 1 - i$, liczba na pozycji j jest nie mniejsza niż poprzedzające ją

Niezmienniczość:

Przed wykonaniem pętli wewnętrznej dla zmiennej i w pętli zewnętrznej i zmiennej j w wewnętrznej, gdzie $j < size - 1 - i$, liczba na pozycji j jest nie mniejsza niż poprzedzające ją $\Rightarrow$

Po wykonaniu pętli wewnętrznej dla zmiennej i w pętli zewnętrznej i zmiennej j w wewnętrznej, gdzie $j < size - 1 - i$, liczba na pozycji $j + 1$ jest nie mniejsza niż poprzedzające ją.

Podstawowe zasady analizy algorytmów: poprawność, skończoność

1 12 3 9 8 2 … 11 9 15 17 23 29

nieposortowane posortowane

Niezmiennik pętli zewnętrznej:

Przed wykonaniem pętli zewnętrznej dla i, liczby od pozycji $size - i$ są posortowane.

$\Rightarrow$

Po wykonaniu pętli zewnętrznej dla i, liczby od pozycji $size - i - 1$ są posortowane.

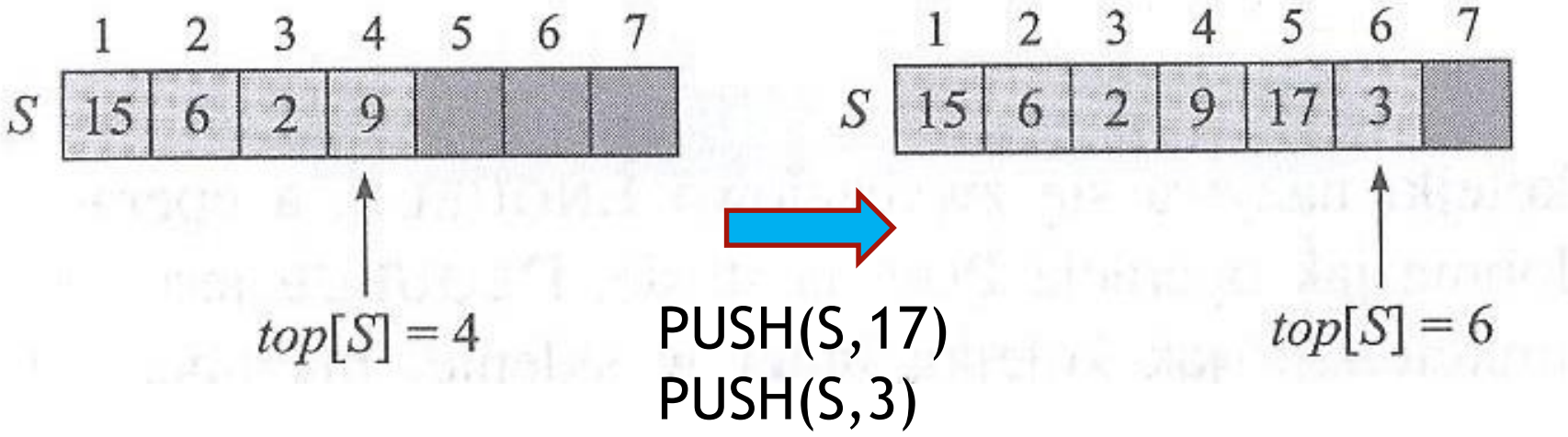
Podstawowe struktury danych

Struktury danych:

- stosy,
- kolejki,
- kopce

zbudowane z użyciem tablic.

Podstawowe struktury danych (stosy)



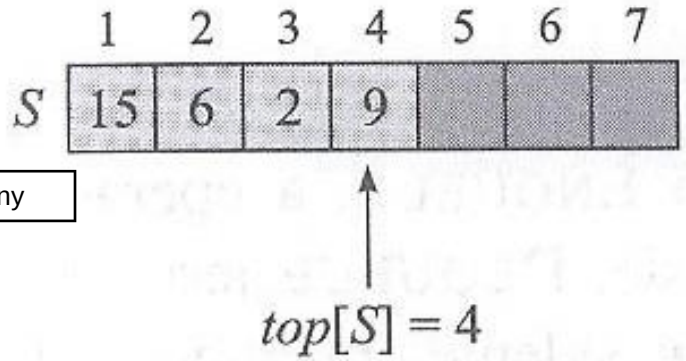
[Źródło CLRS, Wprowadzenie do algorytmów]

Stos z co najwyżej n elementami implementowany jako tablica S[1..n].
top[S] – numer elementu na szczycie stosu,
S[1.. *top*[S]] - elementy na stosie,
top[S]=0 - stos jest pusty

Podstawowe struktury danych (stosy)

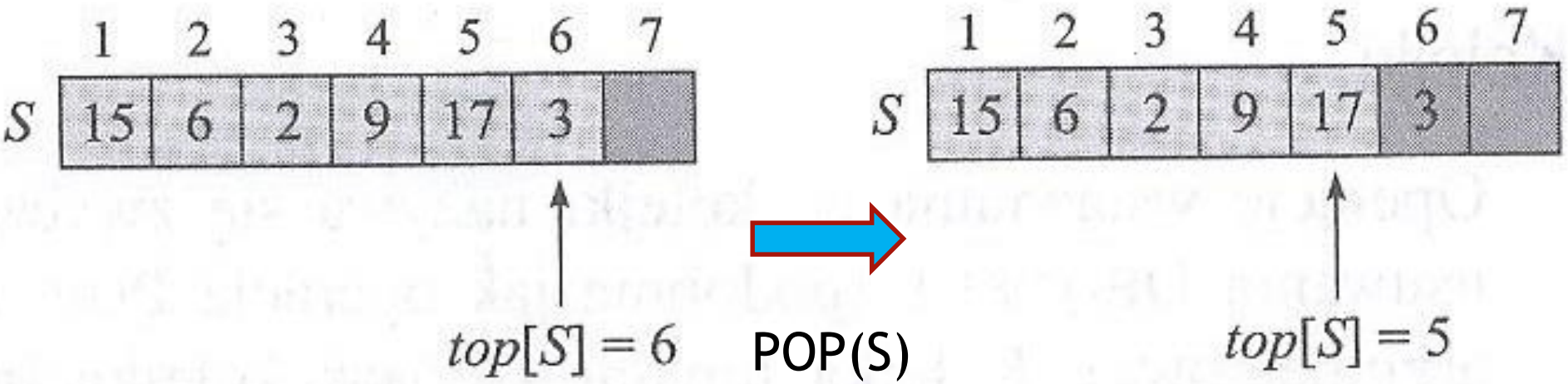
```
STACK-FULL(S)
    if top[S]=n
        then return True
        else return False
```

jeeli góra ma pozycje caoci to jest peny

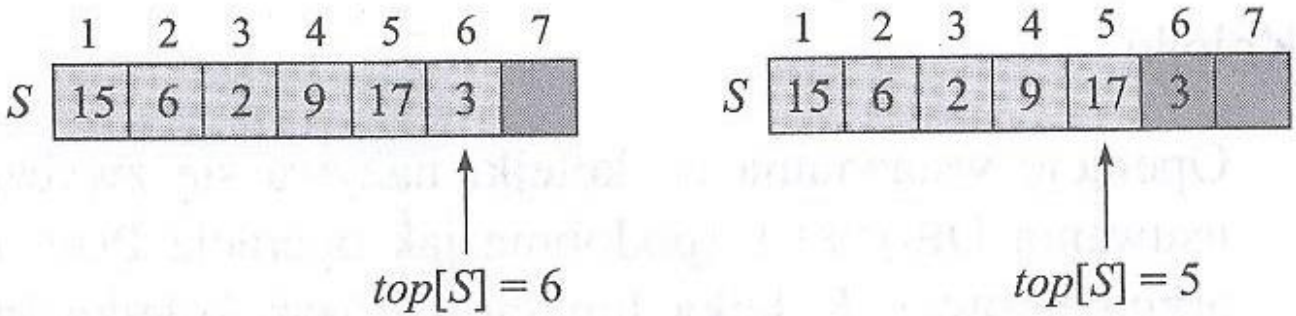



```
PUSH(S,x)
    if STACK-FULL(S)
        then error „STACK-FULL(S)”
        else top[S] ← top[S]+1
             S[top[S]] ← x
```

Podstawowe struktury danych (stosy)



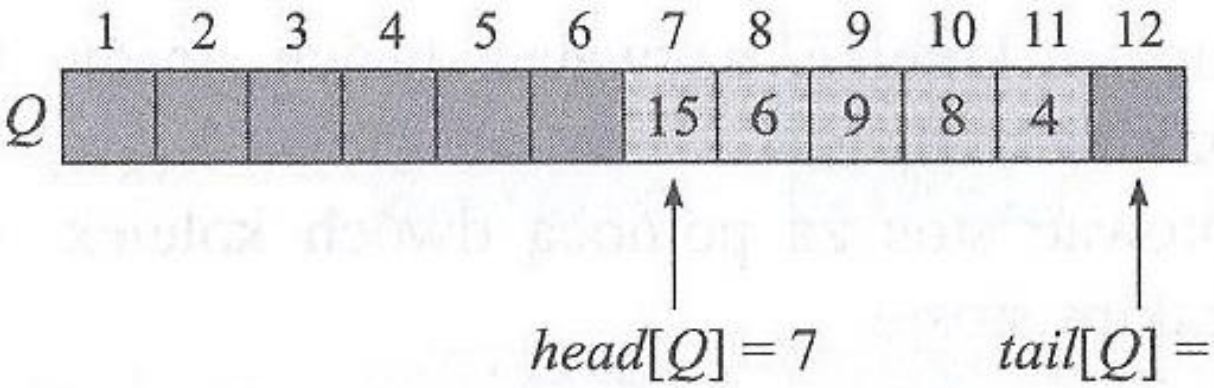
Podstawowe struktury danych (stosy)



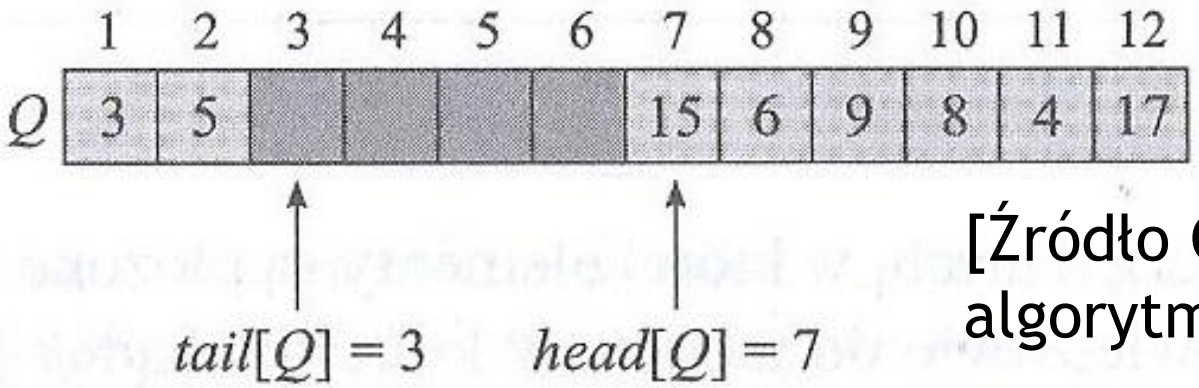

```
STACK-EMPTY(S)
    if top[S]=0
        then return True
        else return False

POP(S)
    if  STACK-EMPTY(S)
        then  error  „STACK-EMPTY(S)”
        else  top[S] ← top[S]-1
              return  S[top[S]+1]
```

Podstawowe struktury danych (kolejki)



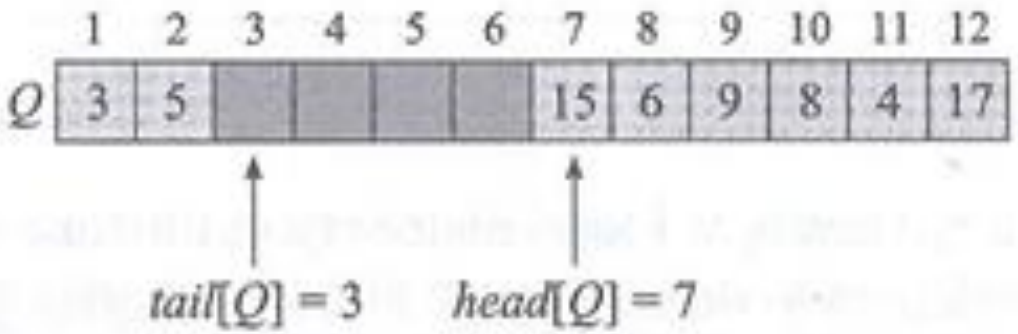
ENQUEUE(Q,17)
ENQUEUE(Q,3)
ENQUEUE(Q,5)



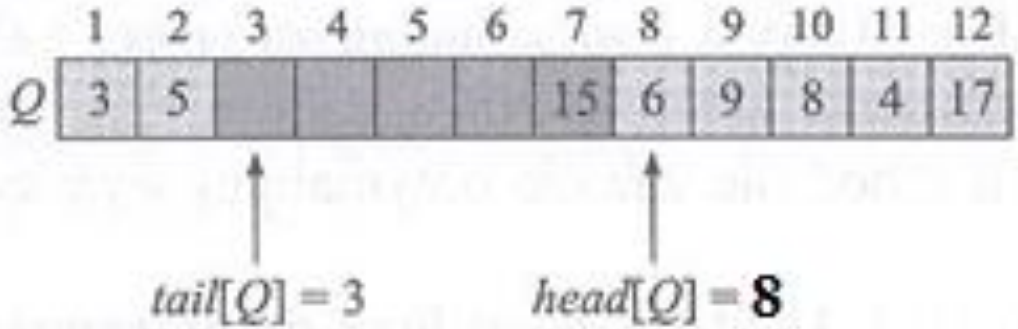
[Źródło CLRS, Wprowadzenie do algorytmów]

Kolejka cykliczna (bufor cykliczny) z co najwyżej $n-1$ elementami implementowana jako tablica $Q[1..n]$.

Podstawowe struktury danych (kolejki)



DEQUEUE(Q)



Podstawowe struktury danych (kolejki)

$Q[1..n]$,

QUEUE-FULL(Q)
    if $head[Q] = (tail[Q] + 1) \bmod n$
        then return True
        else return False

$O(1)$

ENQUEUE (Q, x)
    if QUEUE-FULL(Q)
        then error „QUEUE-FULL(Q)”
        else $Q[tail[Q]] \leftarrow x$
            if $tail[Q] = n$
                then $tail[Q] \leftarrow 1$
                else $tail[Q] \leftarrow tail[Q] + 1$

$O(1)$

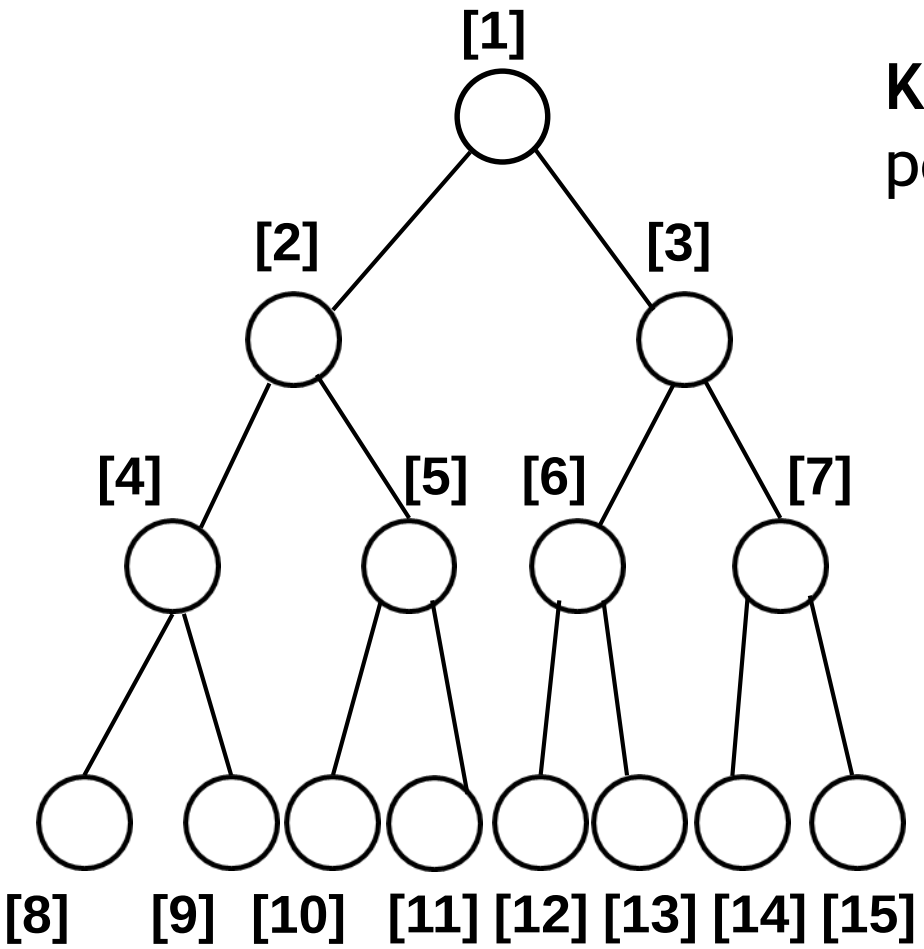
Podstawowe struktury danych (kolejki)

$Q[1..n]$, Początek: $head[Q] = tail[Q] = 1$

QUEUE-EMPTY(Q)
    if $head[Q] = tail[Q]$ $O(1)$
        then return True
        else return False

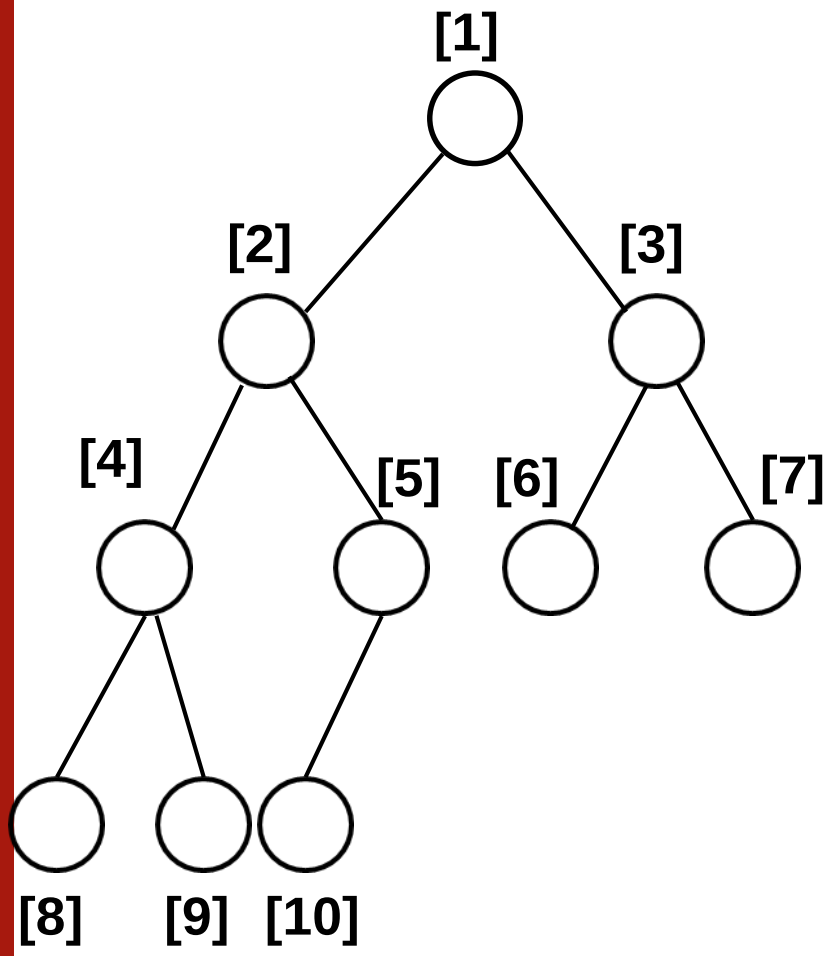
DEQUEUE (Q)
    if QUEUE-EMPTY(Q)
        then error „QUEUE-EMPTY(Q)” $O(1)$
        else $x \leftarrow Q[head[Q]]$
            if $head[Q] = n$
                then $head[Q] \leftarrow 1$ **else** $head[Q] \leftarrow head[Q] + 1$
            return x

Kopce



Kopiec binarny jest prawie pełnym drzewem binarnym.

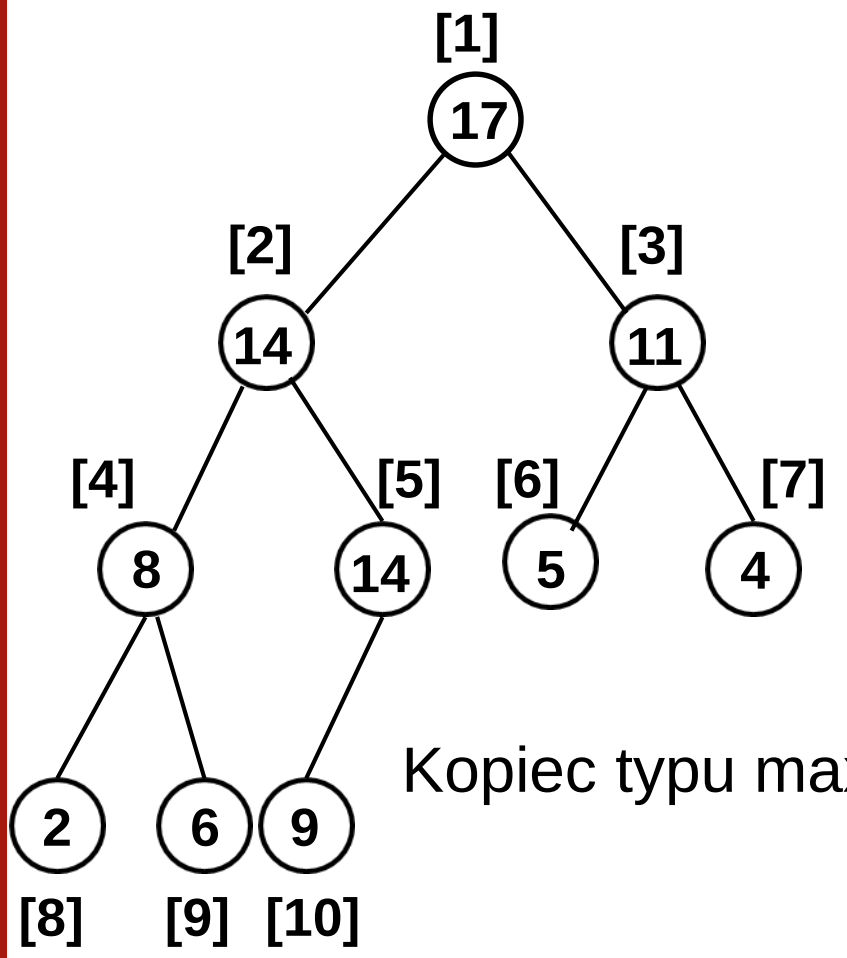
Pełne drzewo binarne



Kopiec binarny jest prawie pełnym drzewem binarnym.

Prawie pełne drzewo binarne

Kopce



Kopiec typu max

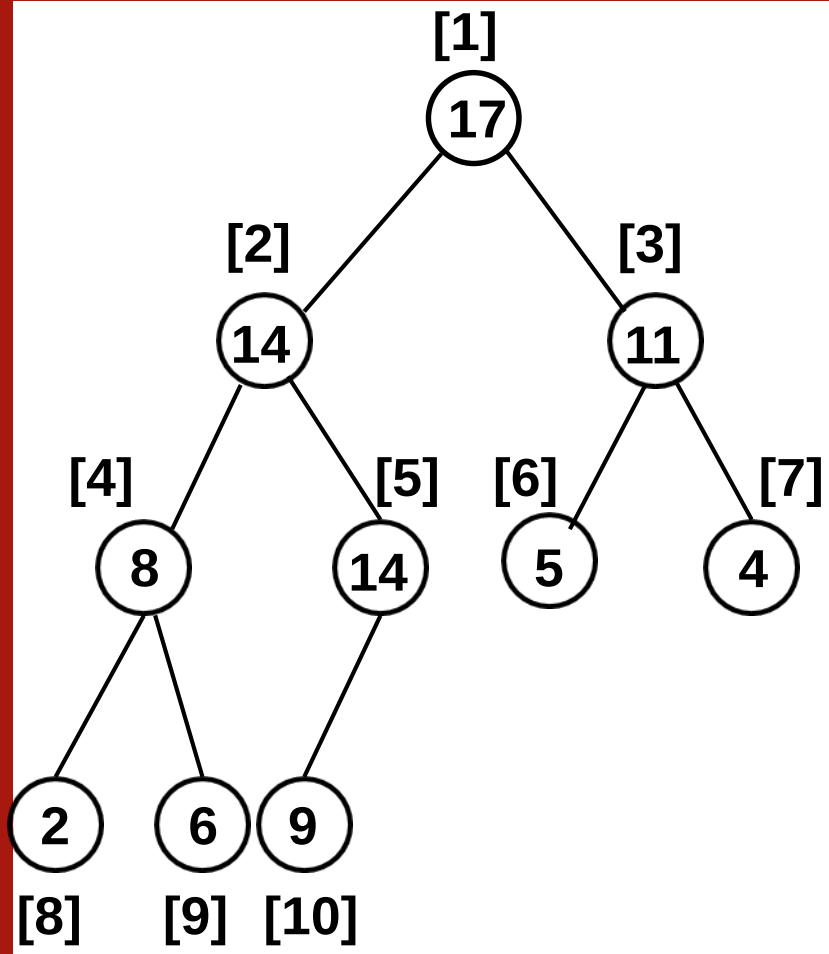
Kopiec binarny jest to prawie pełne drzewo binarne.

Rodzaje kopców:

typu max (wartość rodzica jest nie mniejsza niż dziecka),

typu min (wartość rodzica jest nie większa niż dziecka).

Kopce



$length[A]$ – liczba elementów tablicy A,

$heap - size[A]$ - liczba elementów kopca w tablicy A,

LEFT(i)

return $2 \cdot i$

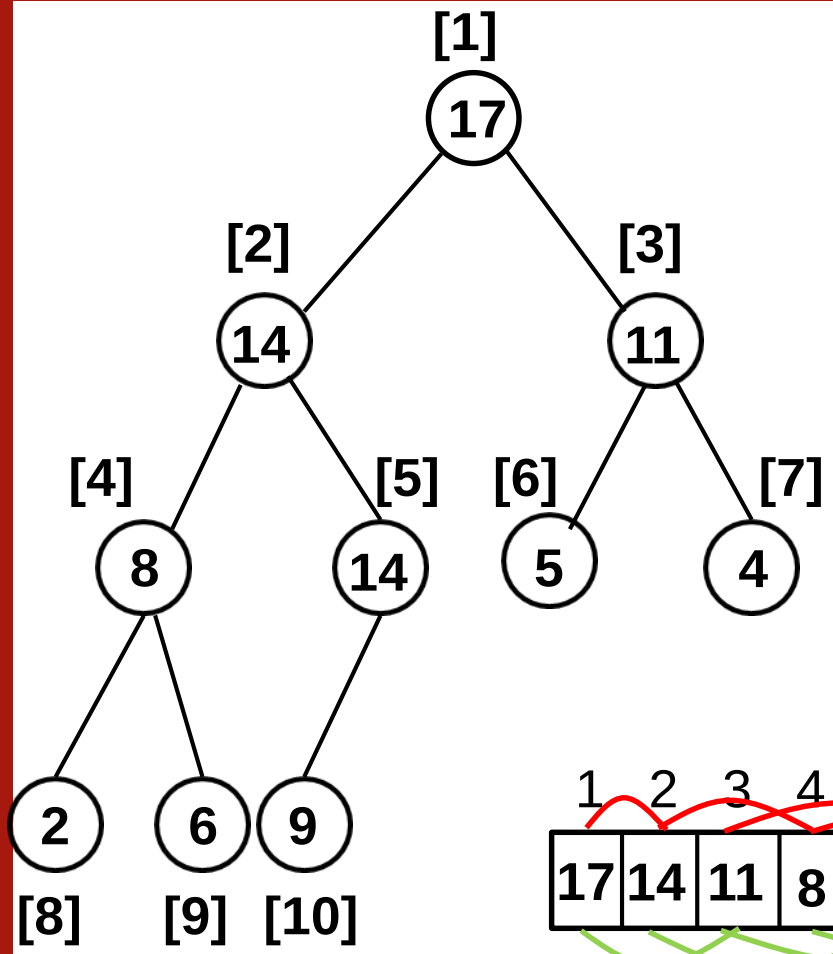
RIGHT(i)

return $(2 \cdot i+1)$

PARENT(i)

return $\lfloor i/2 \rfloor$

Kopce



LEFT(i)

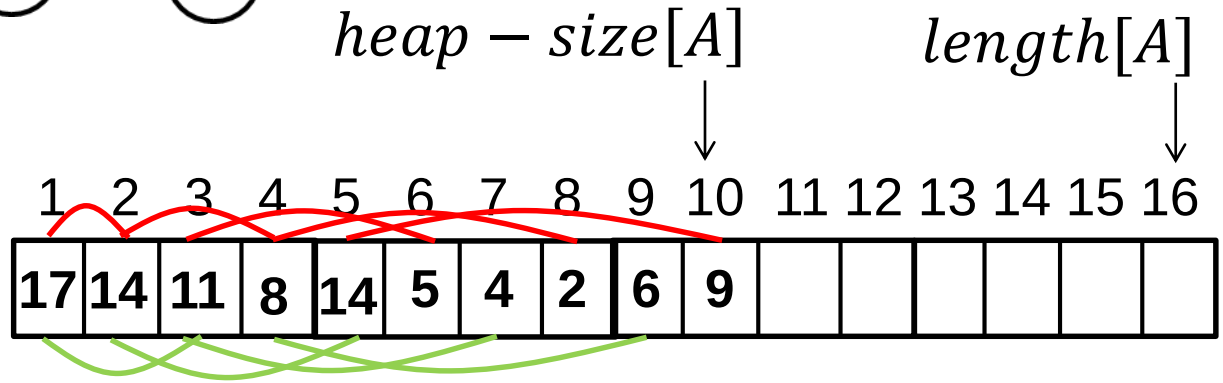
return $2 \cdot i$

RIGHT(i)

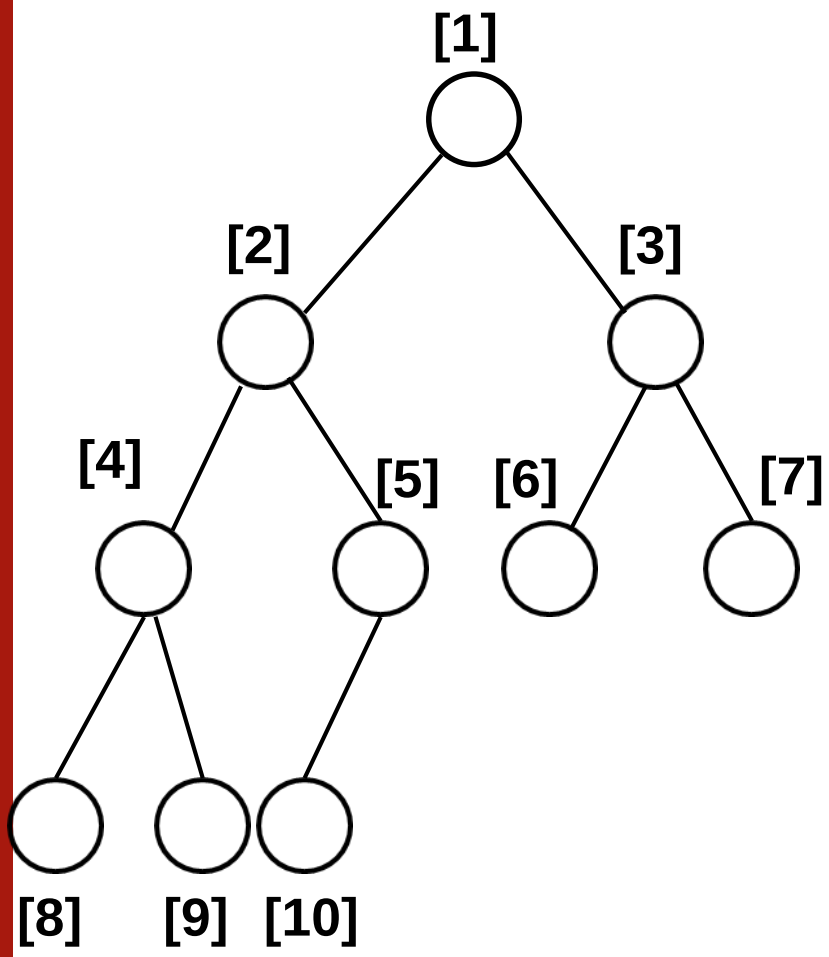
return $(2 \cdot i+1)$

PARENT(i)

return $\lfloor i/2 \rfloor$



Kopce



Własność

Kopiec o $2 \leq n$ wierzchołkach. Elementy o indeksach $\lfloor n/2 \rfloor+1$ do n są liśćmi, a o mniejszych indeksach nie są nimi.

Przykład

$\lfloor n/2 \rfloor= \lfloor 10/2 \rfloor=5$

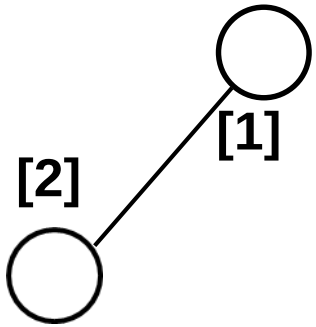
$\lfloor n/2 \rfloor+1= \lfloor 10/2 \rfloor+1=6$

Kopce

Własność
Kopiec o $2 \leq n$ wierzchołkach.
Elementy o indeksach $\lfloor n/2 \rfloor+1$ do n są liśćmi, a o mniejszych indeksach nie są nimi.

Dowód indukcyjny

1. $n = 2$,
   element o indeksie $\lfloor 2/2 \rfloor+1=2$ jest liściem, o indeksie $\lfloor 2/2 \rfloor=1$ nim nie jest.



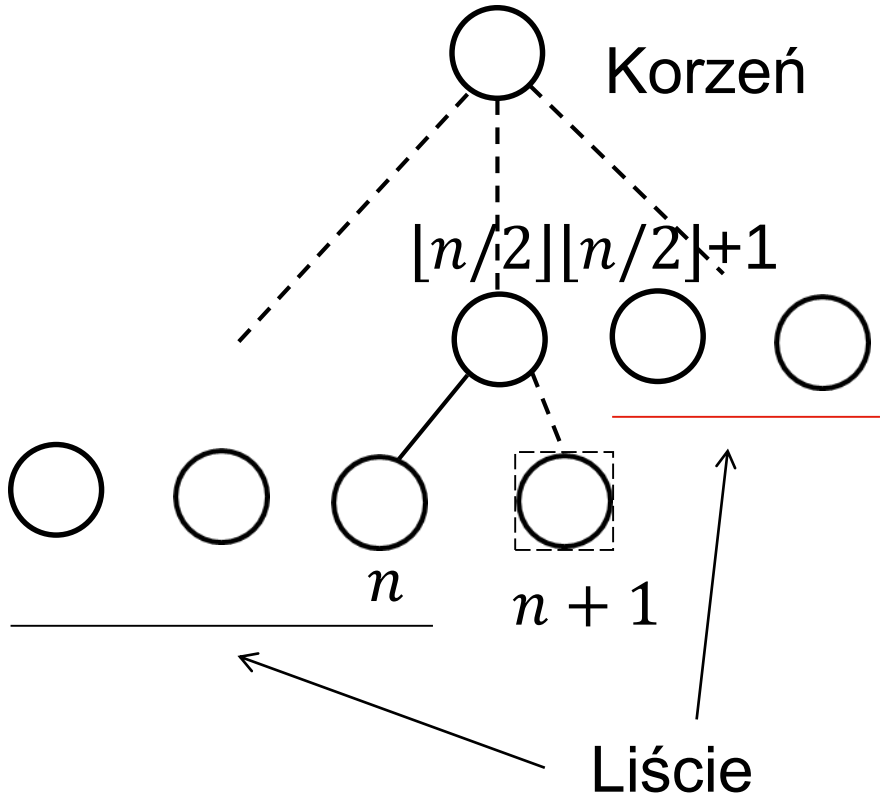
Kopce

Dowód indukcyjny

Założenie indukcyjne:
Elementy o indeksach $\lfloor n/2 \rfloor+1$ do n są liśćmi, a o mniejszych indeksach nie są nimi.

Teza indukcyjna:
Elementy o indeksach $\lfloor (n+1)/2 \rfloor+1$ do $n+1$ są liśćmi, a o mniejszych indeksach nie są nimi.

2.1 n parzyste



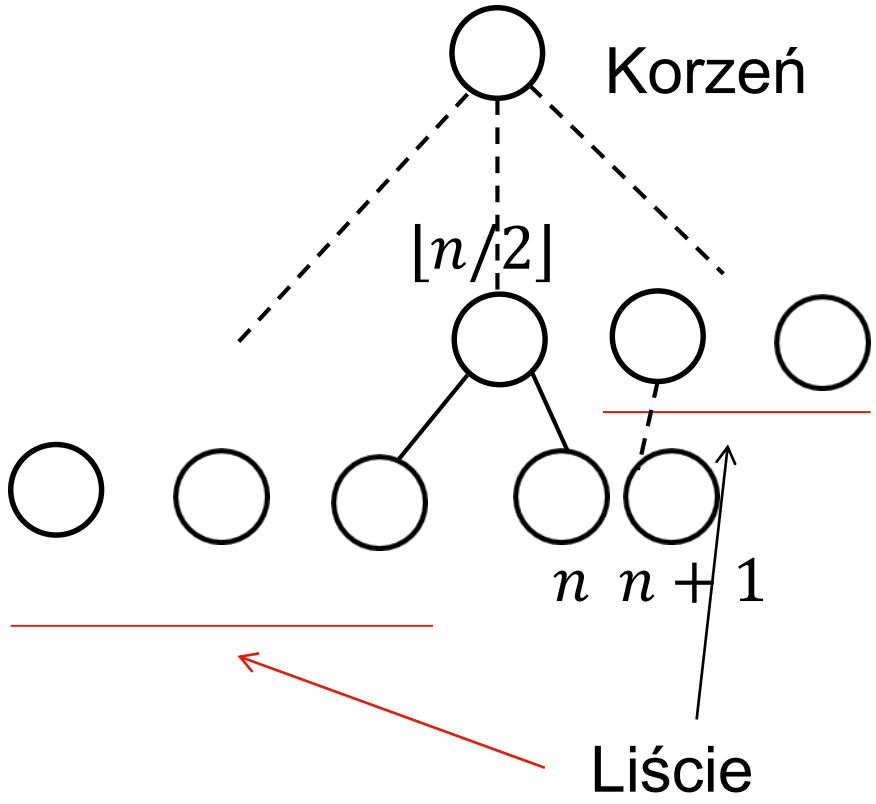
Kopce

Dowód indukcyjny

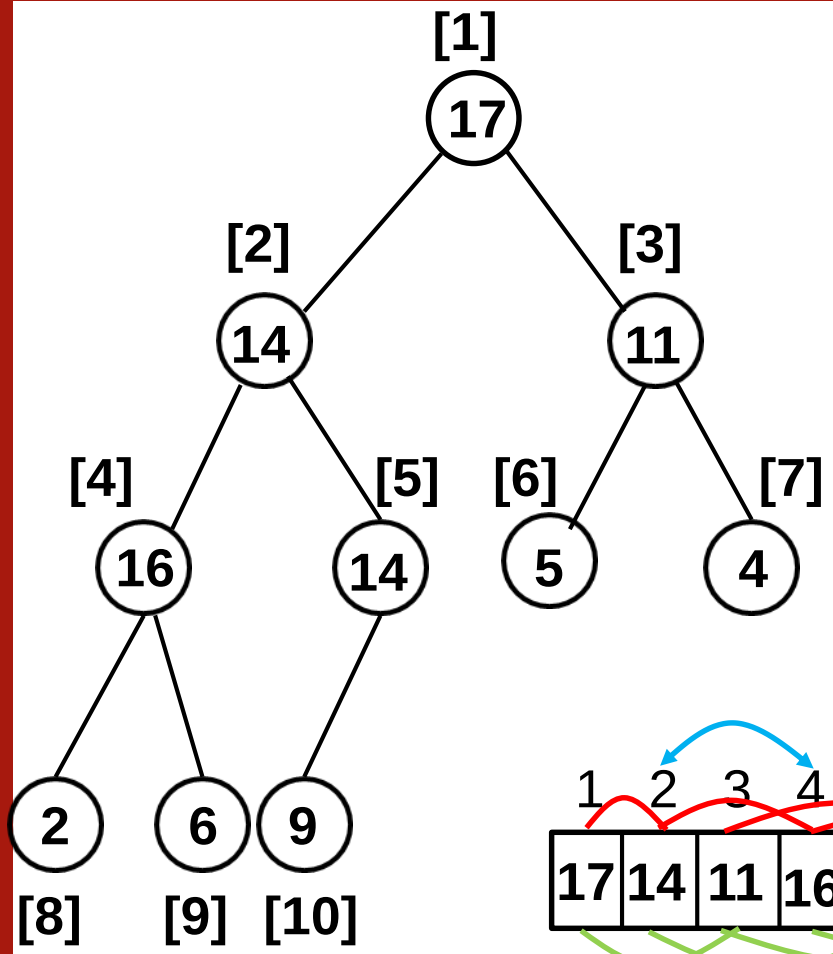
2.2 n nieparzyste

Założenie indukcyjne:
Elementy o indeksach $\lfloor n/2 \rfloor+1$ do n są liśćmi, a o mniejszych indeksach nie są nimi.

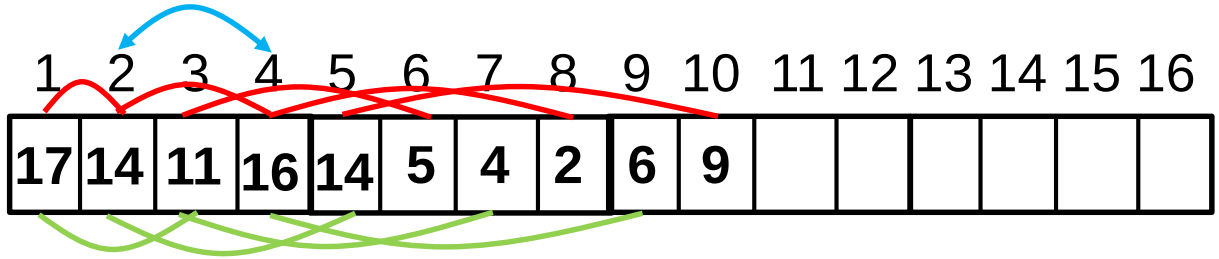
Teza indukcyjna:
Elementy o indeksach $\lfloor (n+1)/2 \rfloor+1$ do $n+1$ są liśćmi, a o mniejszych indeksach nie są nimi.



Kopce



Własność
Operacje zamiany elementów rodzica i dziecka w tablicy nie zmieniają relacji incydencji LEFT, RIGHT, PARENT między pozycjami w tablicy.

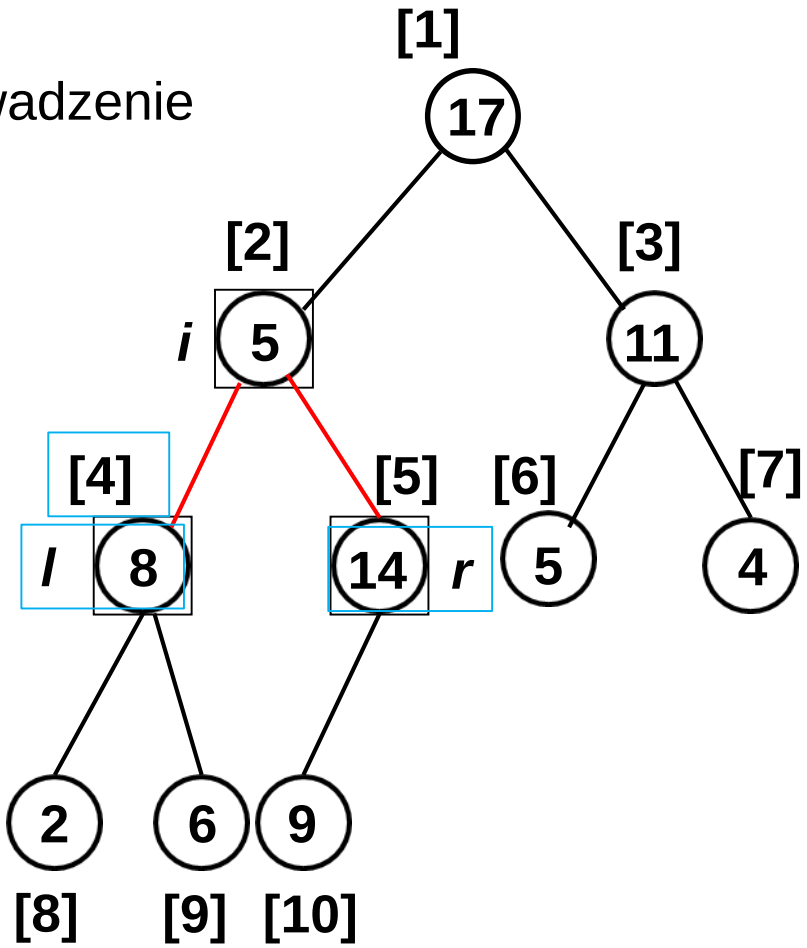


Kopce

Przywracanie własności kopca

```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] i A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] i A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then zamień A[i] ↔ A[largest]
10       MAX-HEAPIFY(A, largest)
```

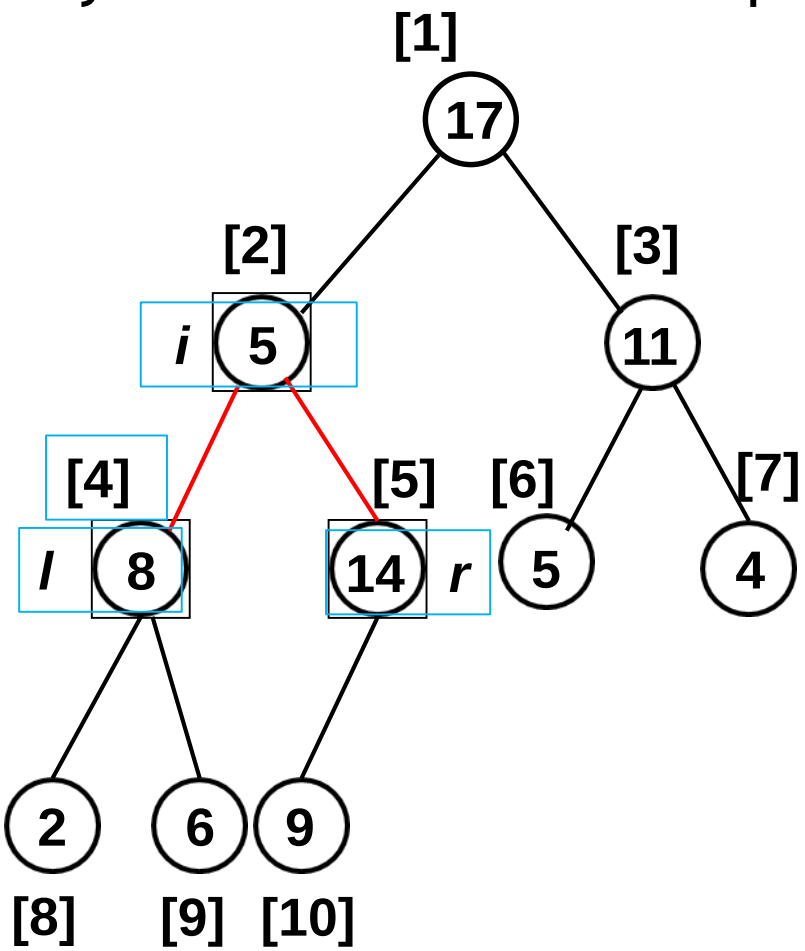
[Źródło: CLRS, Wprowadzenie do algorytmów]

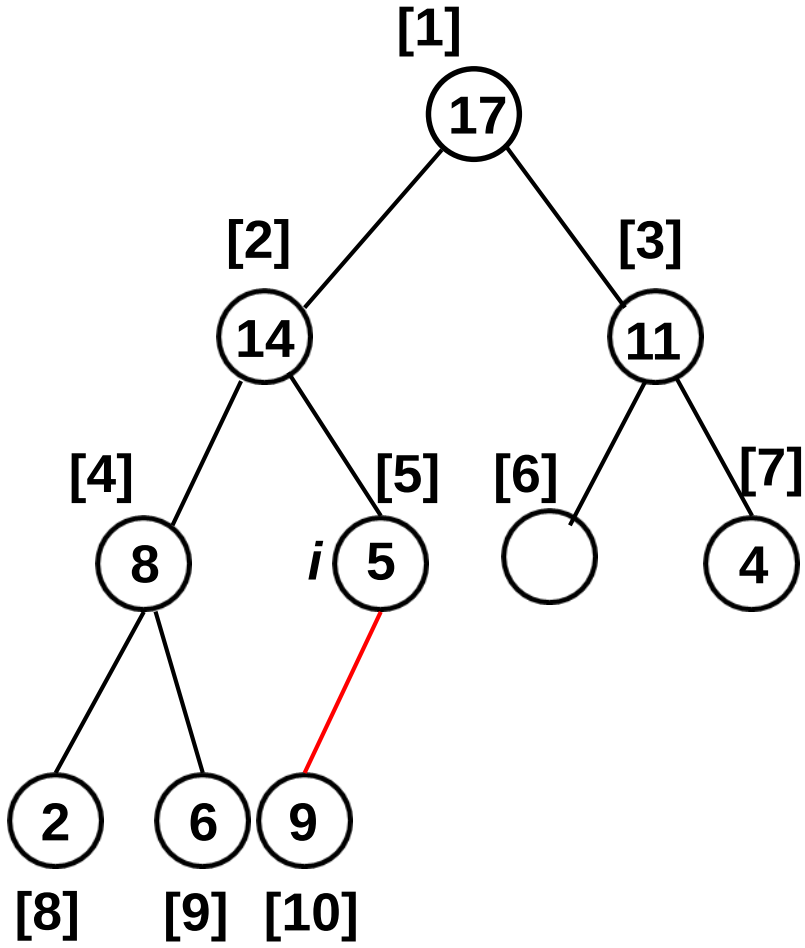


Założenie: Drzewa binarne o korzeniach w LEFT(i), RIGHT(i) są kopcami typu max.

Kopce

Przywracanie własności kopca

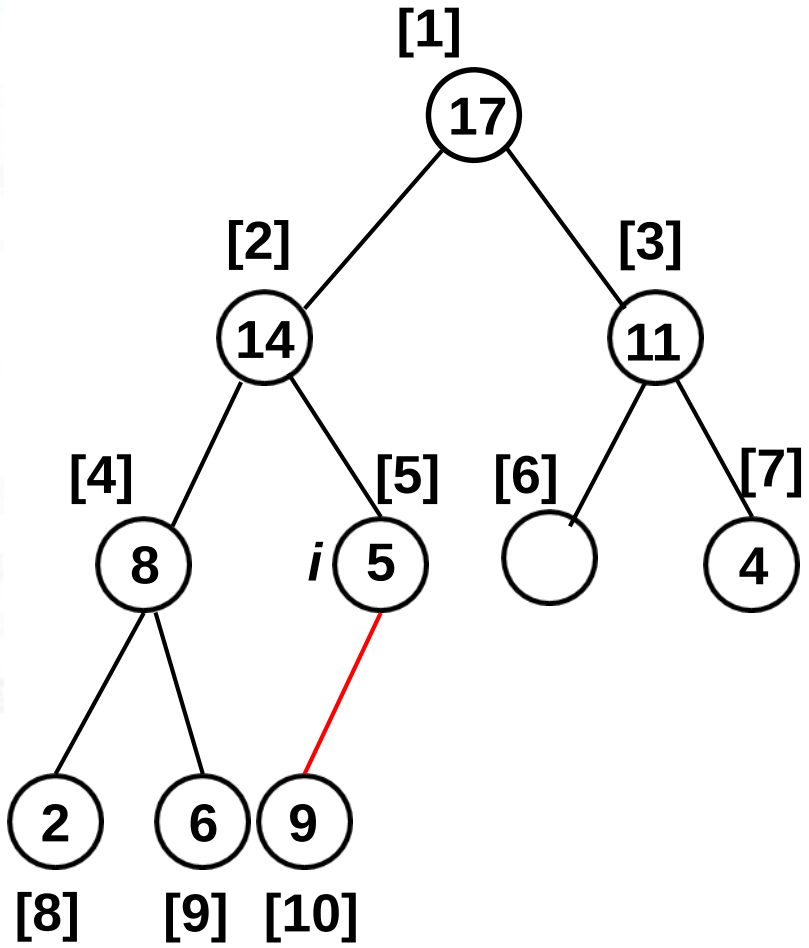




Kopce

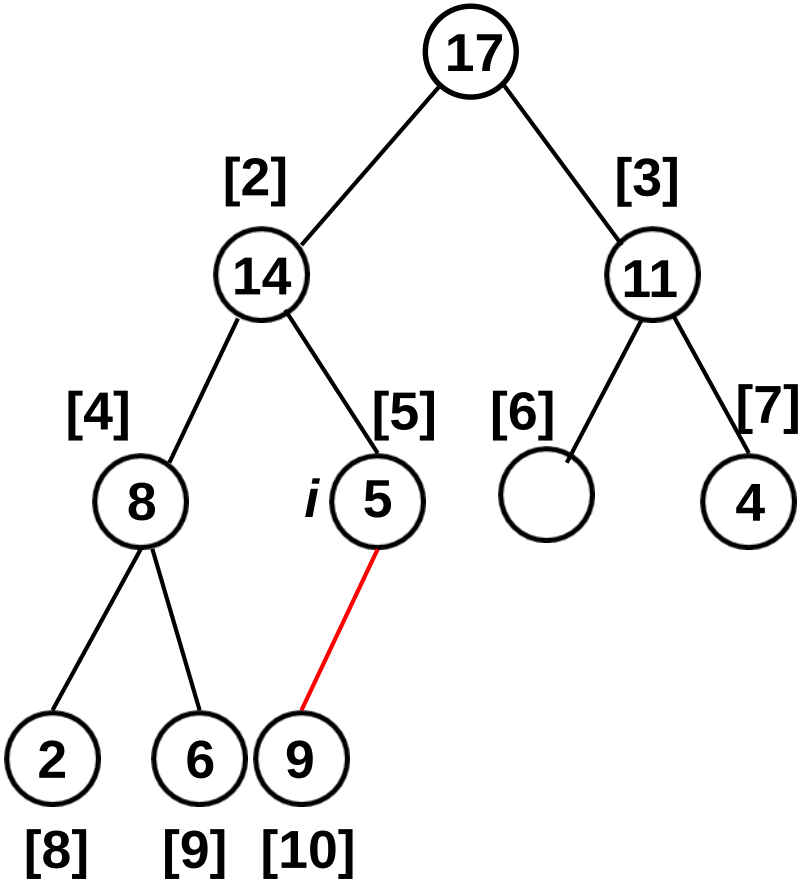
Przywracanie własności kopca

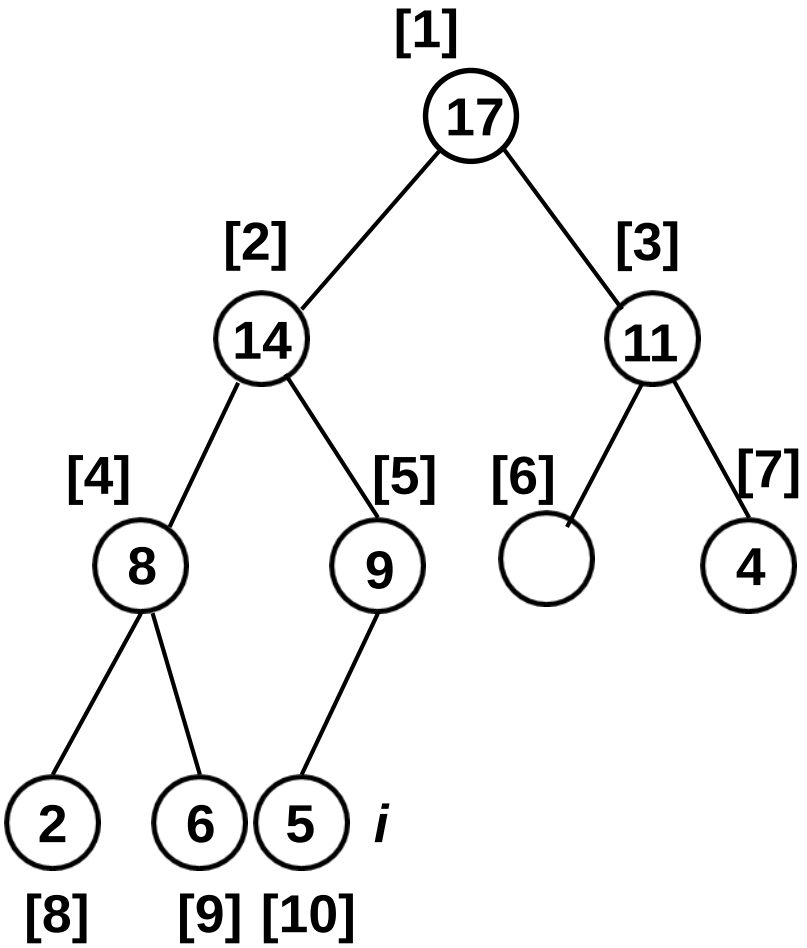
```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] i A[l] > A[i]
 4      then largest ← l
 5      else largest ← i
 6  if r ≤ heap-size[A] i A[r] > A[largest]
 7      then largest ← r
 8  if largest ≠ i
 9      then zamień A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```



Założenie: Drzewa binarne o korzeniach w LEFT(i), RIGHT(i) są kopcami typu max.

Kopce



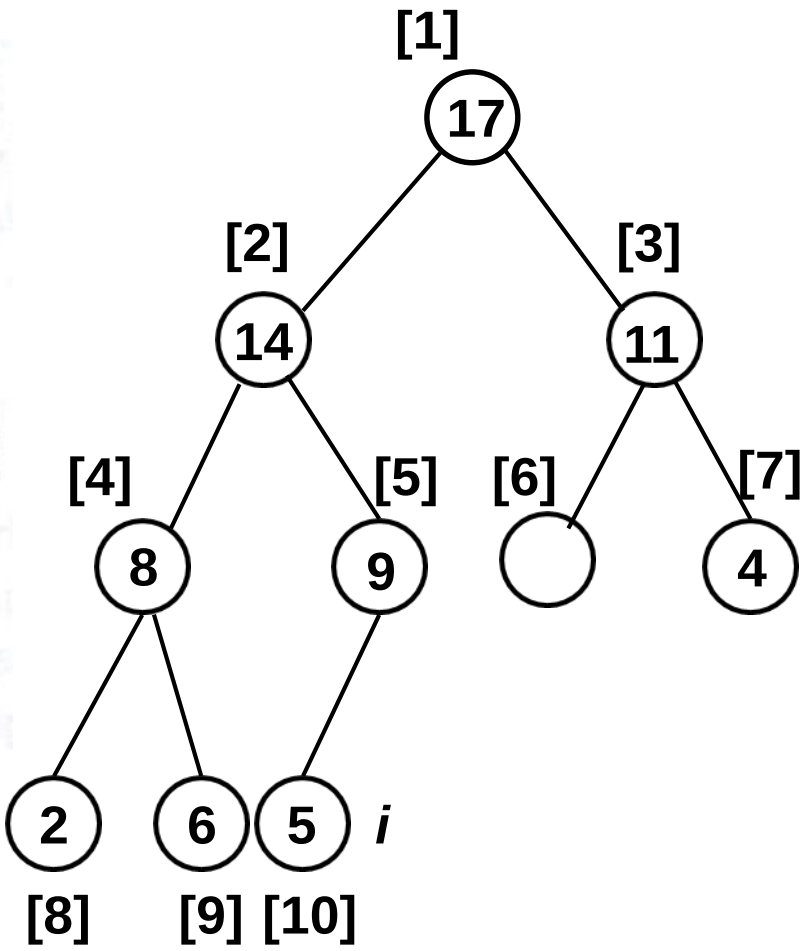


Kopce

Przywracanie własności kopca

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] i A[l] > A[i]
 4      then largest ← l
 5      else largest ← i
 6  if r ≤ heap-size[A] i A[r] > A[largest]
 7      then largest ← r
 8  if largest ≠ i
 9      then zamień A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

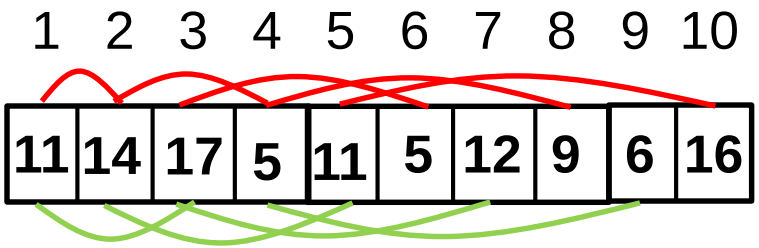
$O(\ln n)$



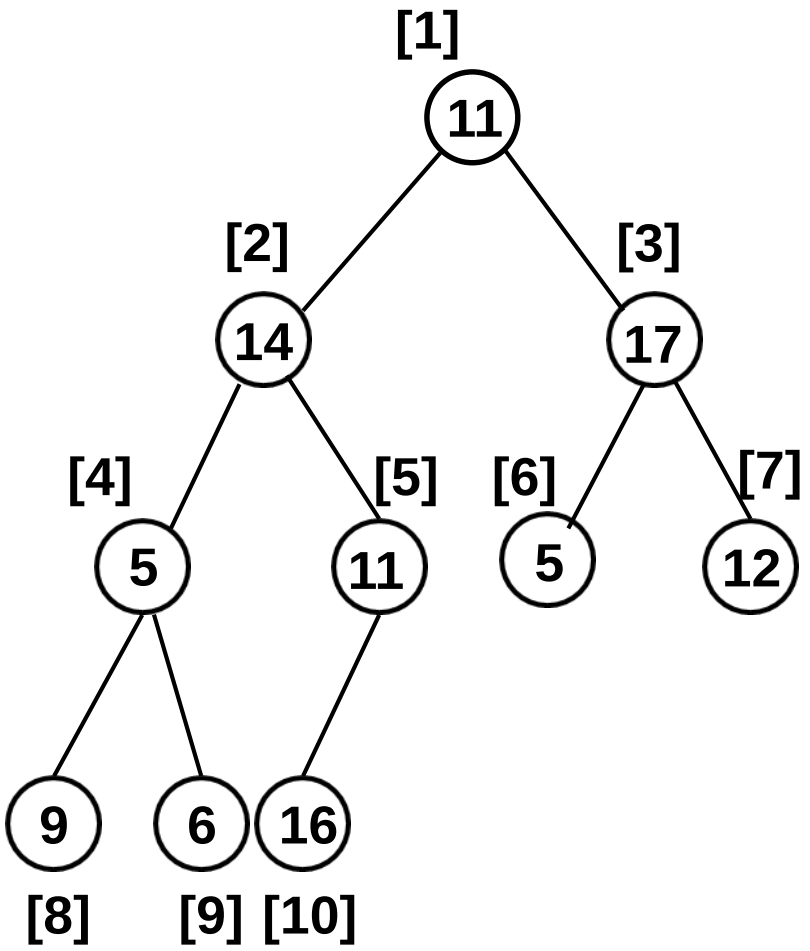
Kopce

Budowanie kopca typu max

Punktem wyjścia tablica $A[1..n]$
gdzie $n = length[A]$



To nie jest kopiec typu max

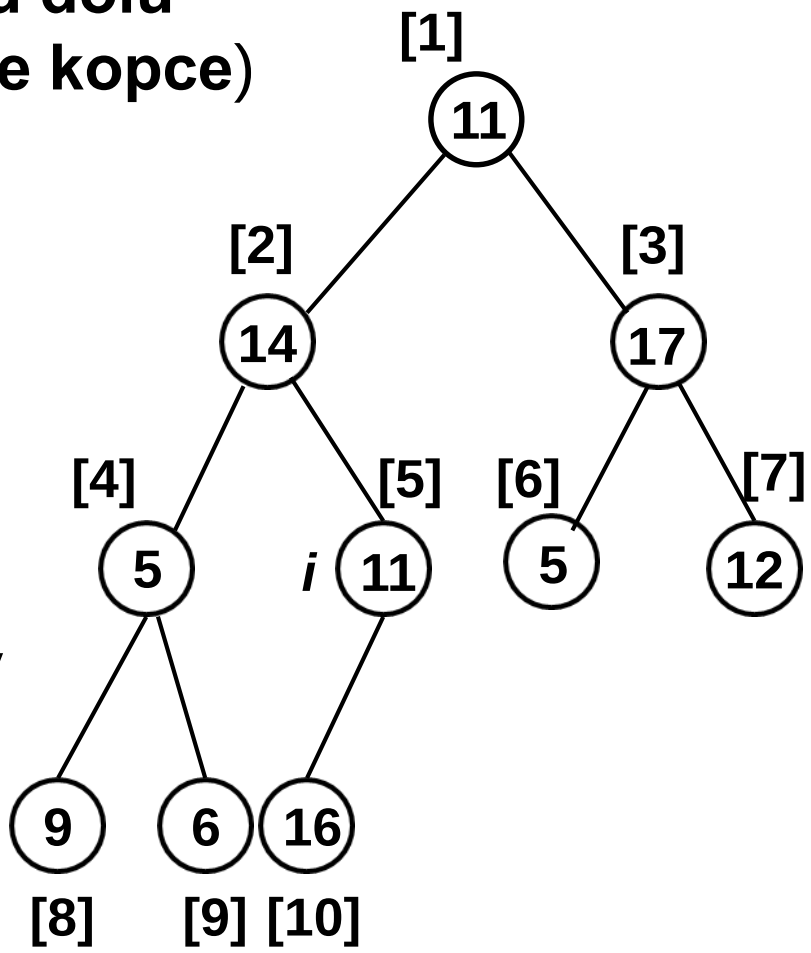


Kopce

Budowanie kopca typu max (**od dołu budowane są coraz to większe kopce**)

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3     do MAX-HEAPIFY(A, i)
```

[Źródło: CLRS, Wprowadzenie do algorytmów]

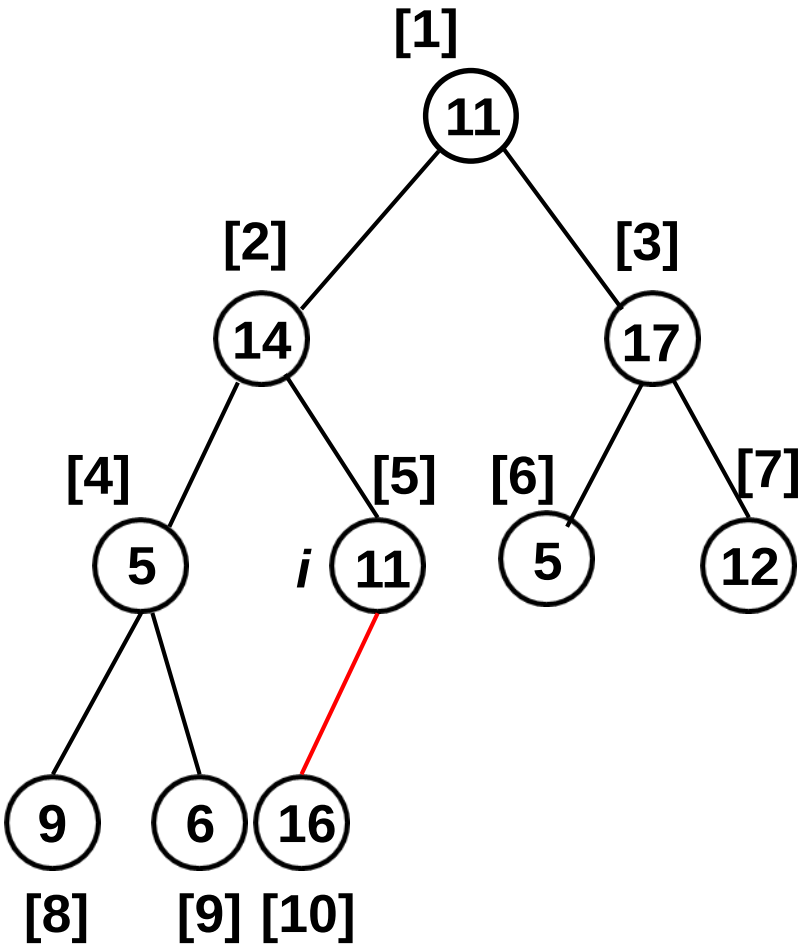


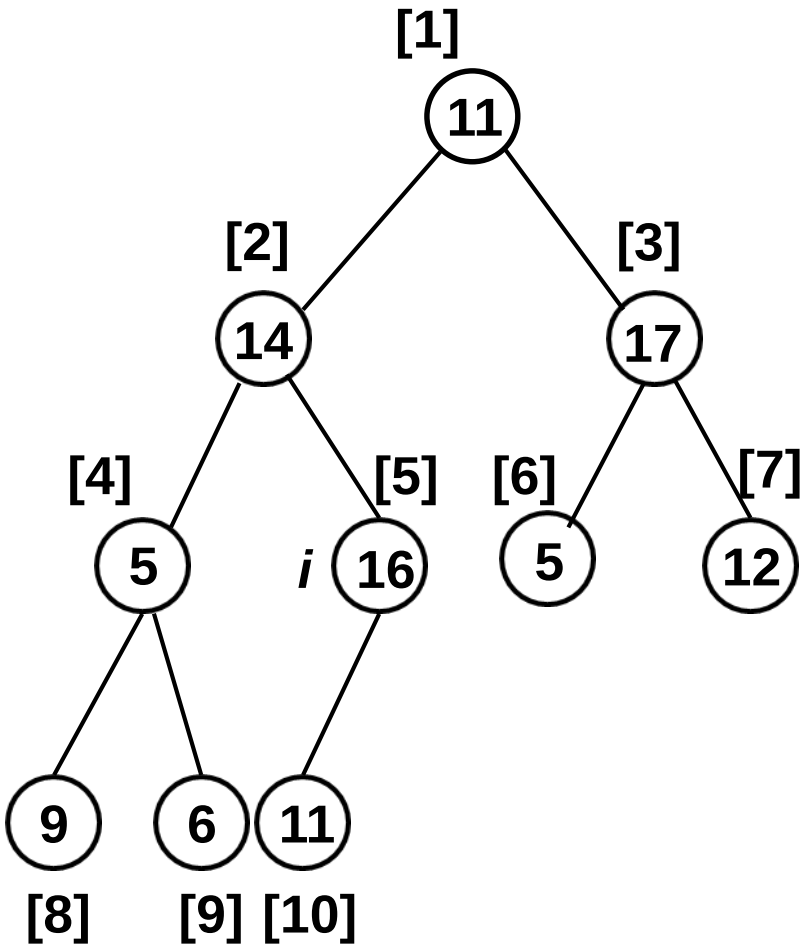
Własność

Kopiec o $2 \leq n$ liściach. Elementy o indeksach $\lfloor n/2 \rfloor+1$ do n są liśćmi (są **kopcami jednoelementowymi**), a o mniejszych indeksach nie są liśćmi.

Kopce

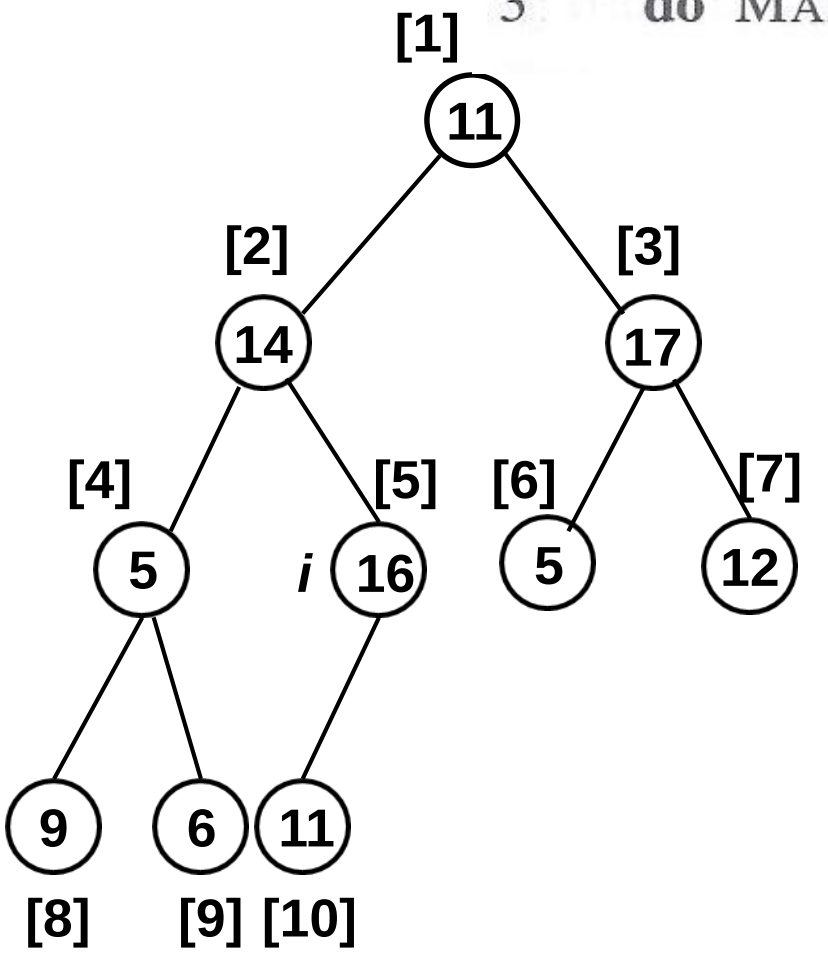
Budowanie kopca typu max

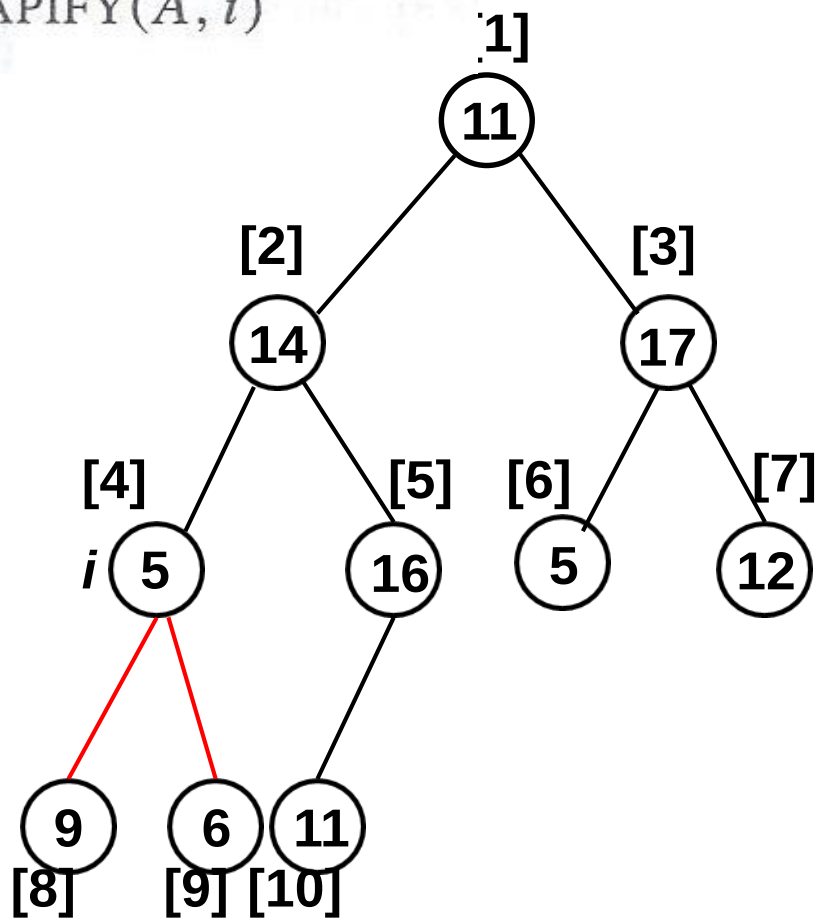


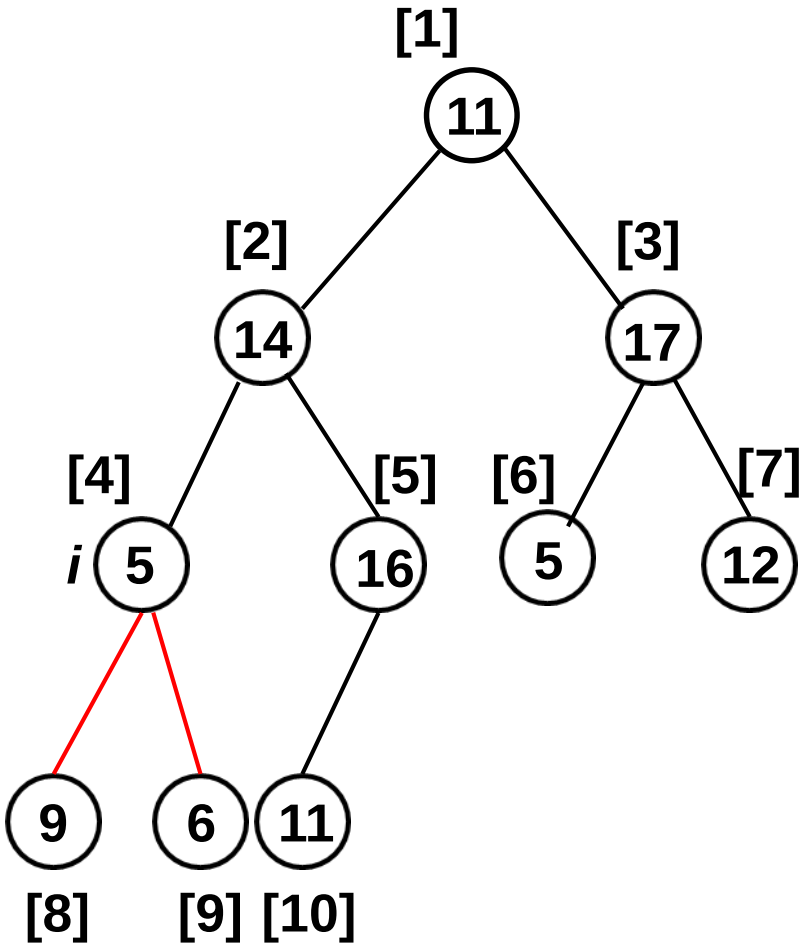


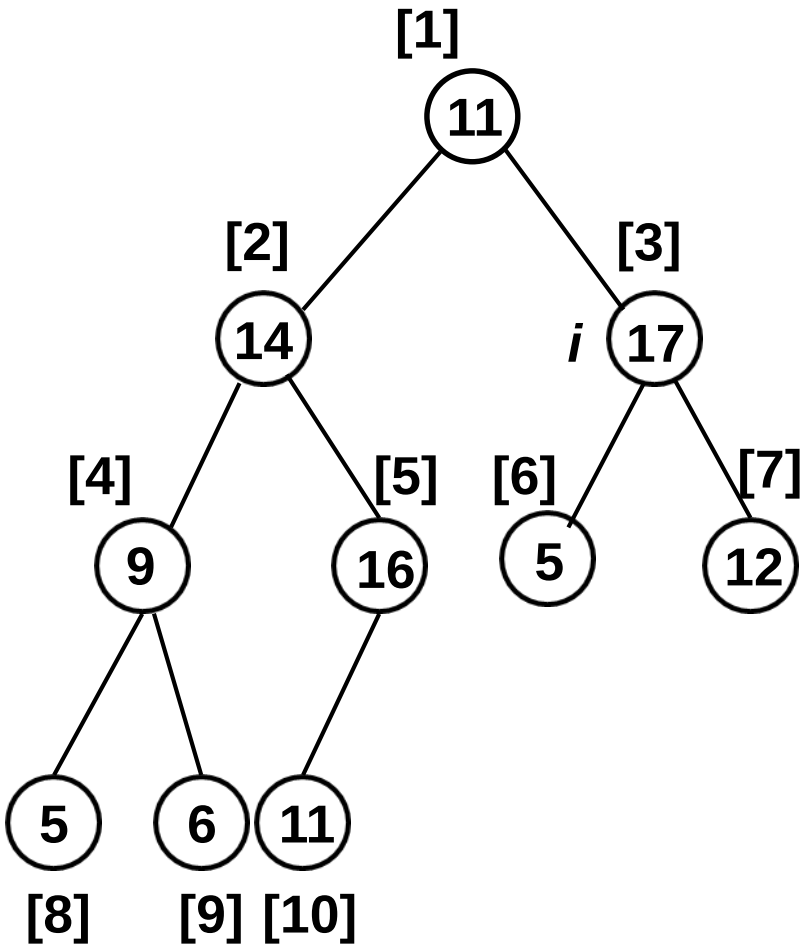
Kopce

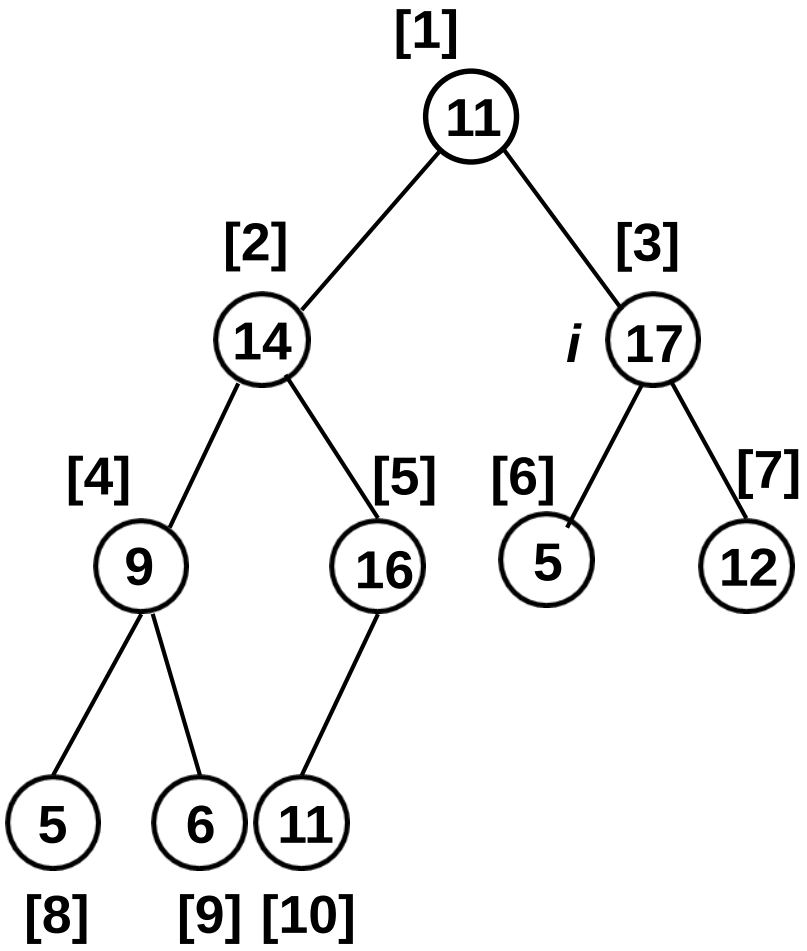
```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3      do MAX-HEAPIFY(A, i)
```

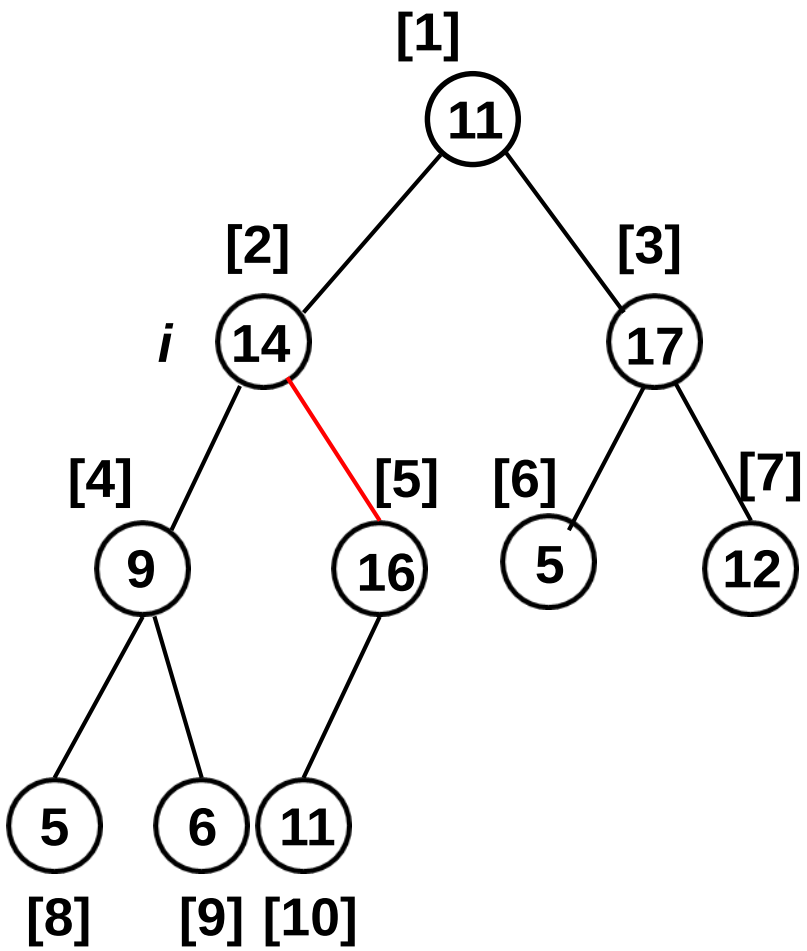


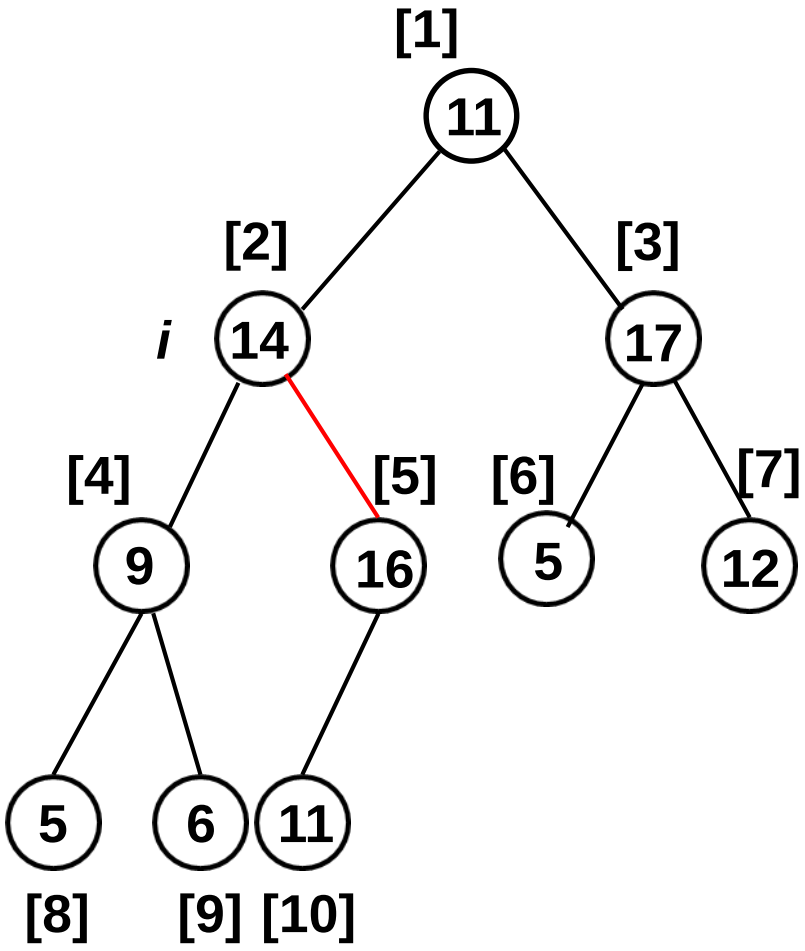


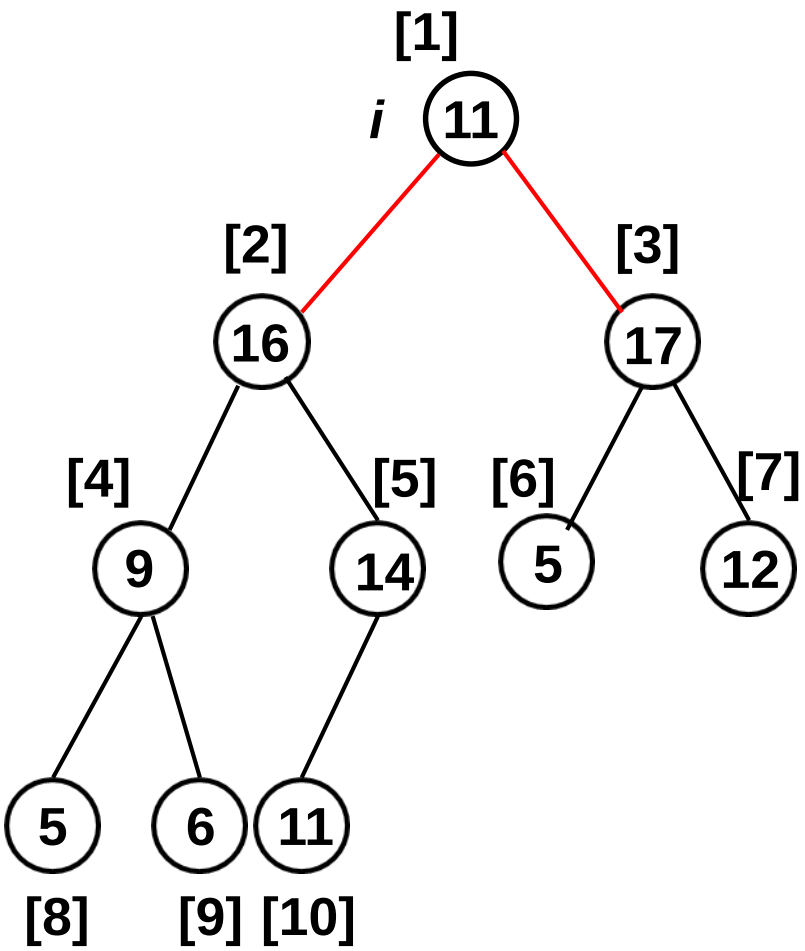

[1]
11
[2]
14
[3]
17
[4]
i
5
[5]
16
[6]
5
[7]
12
9
6
11
[8]
[9]
[10]

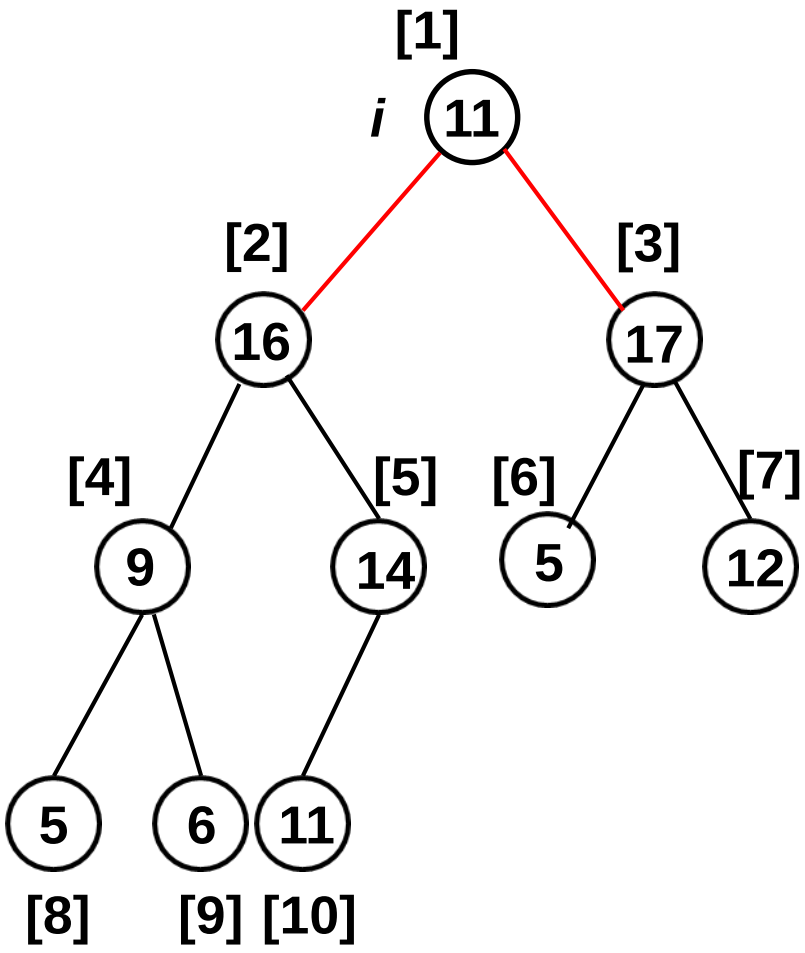

[1]
11
[2]
14
[3]
i
17
[4]
9
[5]
16
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]

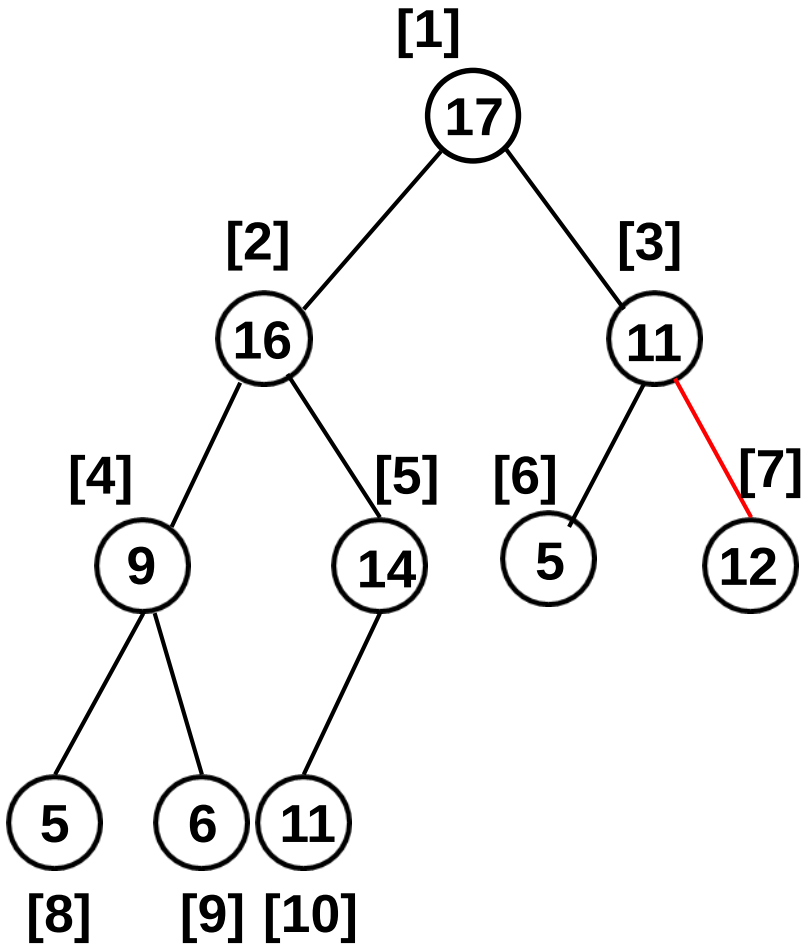

[1]
11
[2]
14
i
[3]
17
[4]
9
[5]
16
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]

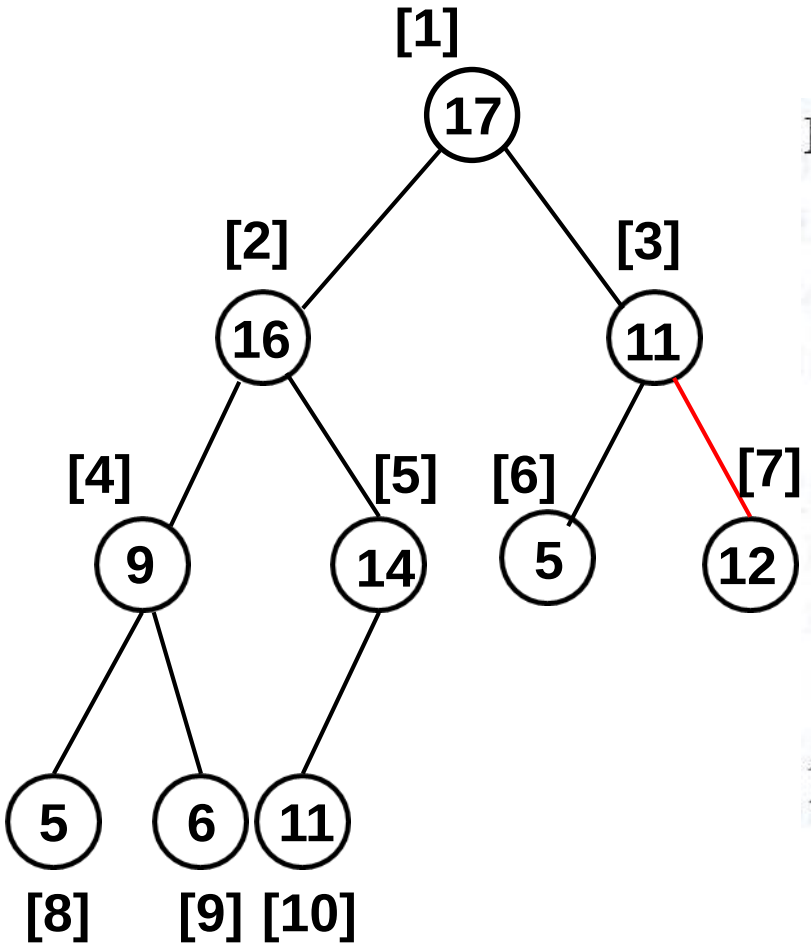

[1]
11
[2]
i
14
[3]
17
[4]
9
[5]
16
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]


[1]
11
[2]
i
14
[3]
17
[4]
9
[5]
16
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]

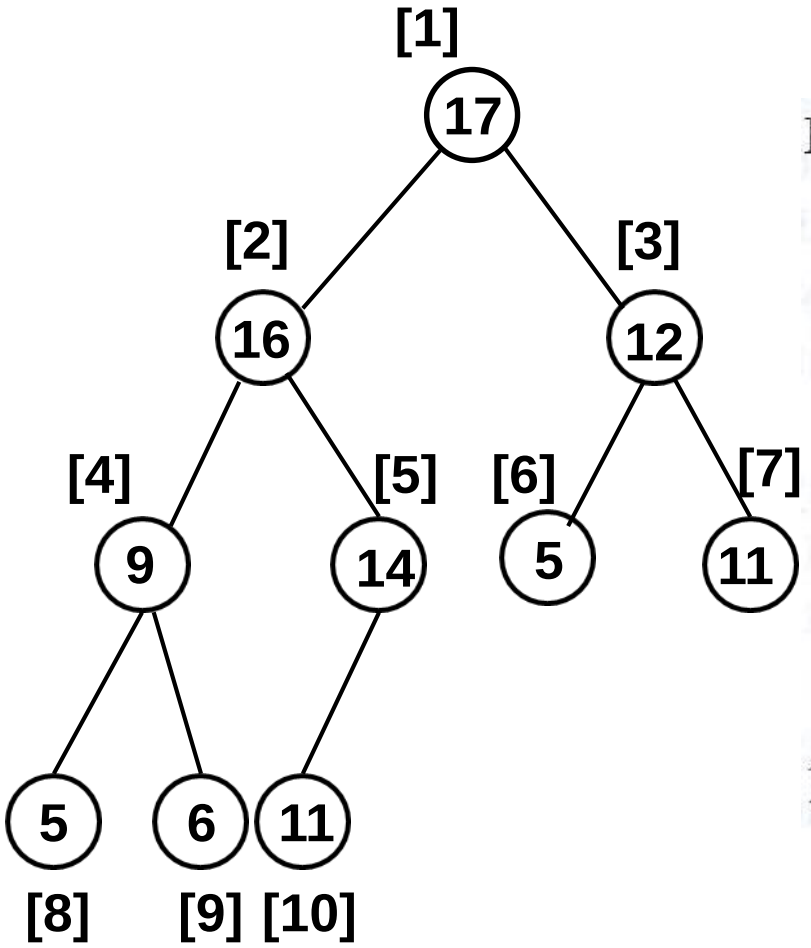

[1]
i
11
[2]
16
[3]
17
[4]
9
[5]
14
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]


[1]
i
11
[2]
16
[3]
17
[4]
9
[5]
14
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]


[1]
17
[2]
16
[3]
11
[4]
9
[5]
14
[6]
5
[7]
12
5
6
11
[8]
[9]
[10]




```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] i A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] i A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then zamień A[i] ↔ A[largest]
10        MAX-HEAPIFY(A, largest)
```




```
MAX-HEAPIFY(A, i)
 1   l ← LEFT(i)
 2   r ← RIGHT(i)
 3   if l ≤ heap-size[A] i A[l] > A[i]
 4       then largest ← l
 5       else largest ← i
 6   if r ≤ heap-size[A] i A[r] > A[largest]
 7       then largest ← r
 8   if largest ≠ i
 9       then zamień A[i] ↔ A[largest]
10           MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3     do MAX-HEAPIFY(A, i)
```

$O(\ln n)$

$O(n \ln n)$

$O(n)$

[Źródło: CLRS, Wprowadzenie do algorytmów]

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3     do MAX-HEAPIFY(A, i)
```

Niezmiennik pętli:
Przed wykonaniem pętli **for** dla i,
każdy węzeł $i+1, i+2, \ldots, length[A]$
jest korzeniem kopca typu max.

Sortowanie przez kopcowanie

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3     do zamień A[1] ↔ A[i]
4        heap-size[A] ← heap-size[A] − 1
5        MAX-HEAPIFY(A, 1)
```

Niezmiennik pętli:
Na początku pętli **for** dla i:
fragment tablicy $A[1..i]$ jest kopcem typu max zawierającym i najmniejszych elementów z $A[1..n]$ $(n = length[A])$,
a fragment $A[i+1..n]$ zawiera $n - i$ posortowanych największych elementów z $A[1..n]$.

Sortowanie przez kopcowanie

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3     do zamień A[1] ↔ A[i]
4        heap-size[A] ← heap-size[A] − 1
5        MAX-HEAPIFY(A, 1)
```

Nie gorzej niż $O(n \ln n)$

$O(n \ln n)$

$O(\ln n)$

$O(n \ln n)$