Algorytmy i struktury danych

Drzewa

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Problem

- przychodzi do nas strumień obiektów (np. napisów, dat, krotek itp.)
- chcemy:
  - wiedzieć, czy obiekty się powtarzają
  - policzyć powtarzające się obiekty
  - powiązać dodatkowe dane z obiektami
  - itp.
- gdyby obiekty były niedużymi liczbami:
  - `tablica[obiekt]++;`
  - `if(tablica[obiekt]) > 0 ...`
  - `tablica[obiekt]=dodatkowe_dane ...`

Problem

- możemy też napisać:

```
obiekt = wczytaj();
for(i = 0; i < o; i++)
      if(tab2[i] == obiekt) break;
if(i == o) tab2[o++] = obiekt;
tablica[i]++;
```

- co jeżeli różnych obiektów będzie dużo?
- co jeżeli nie będziemy znali liczby różnych obiektów?

Pamięci asocjacyjne

- rozwiązaniem jest wykorzystanie pamięci asocjacyjnej (skojarzeniowej)
- jest to pamięć przechowująca powiązane pary (*klucz*, *wartość*)
- *klucz* jest "indeksem" w tablicy, *wartość* jest elementem w niej przechowywanym
- zarówno klucz jak i wartość może być dowolnego typu
- na pytanie o klucz, pamięć taka odpowiada skojarzoną z nią wartością (chyba, że z danym kluczem nie jest skojarzona żadna wartość)
- np. słownik polsko-angielski
  - kluczem jest słowo polskie
  - wartością jest słowo angielskie

Pamięci asocjacyjne

- skojarzenie jest jednokierunkowe: możemy pytać tylko o klucz
  - w słowniku polsko-angielskim trudno znaleźć tłumaczenie angielskiego wyrazu
- jeżeli potrzebujemy również skojarzenia odwrotnego — używamy drugiego słownika
- tablica jest szczególnym typem pamięci asocjacyjnej, w której kluczem są niewielkie liczby naturalne

Rozwiązanie

- możemy teraz napisać:

```
PamiecAsocjacyjna T;
...
obiekt = wczytaj();
if(!ZNAJDZ(T, obiekt)) T[obiekt] = 1;
else T[obiekt] = T[obiekt] + 1;
```

Drzewa poszukiwań binarnych (BST)

- szybki algorytm wyszukiwania — wyszukiwanie binarne
- jeżeli *klucz* jest mniejszy od wybranego elementu, ograniczamy się do lewej strony, jeżeli większy, ograniczamy się do prawej strony
- "rozwidlona" lista:

```
struct Node {
      Klucz key;
      Dana value;
      Node *parent;
      Node *left;
      Node *right;
      };
```

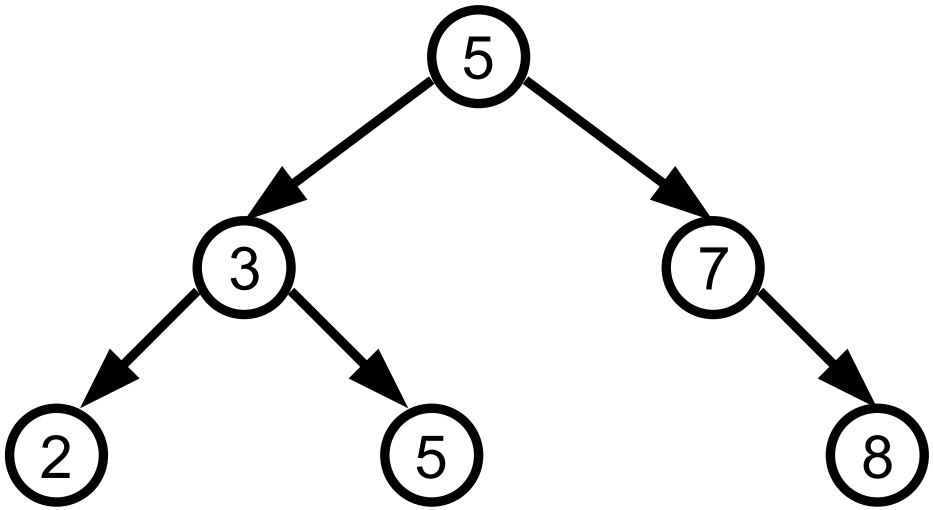
Drzewa poszukiwań binarnych (BST)

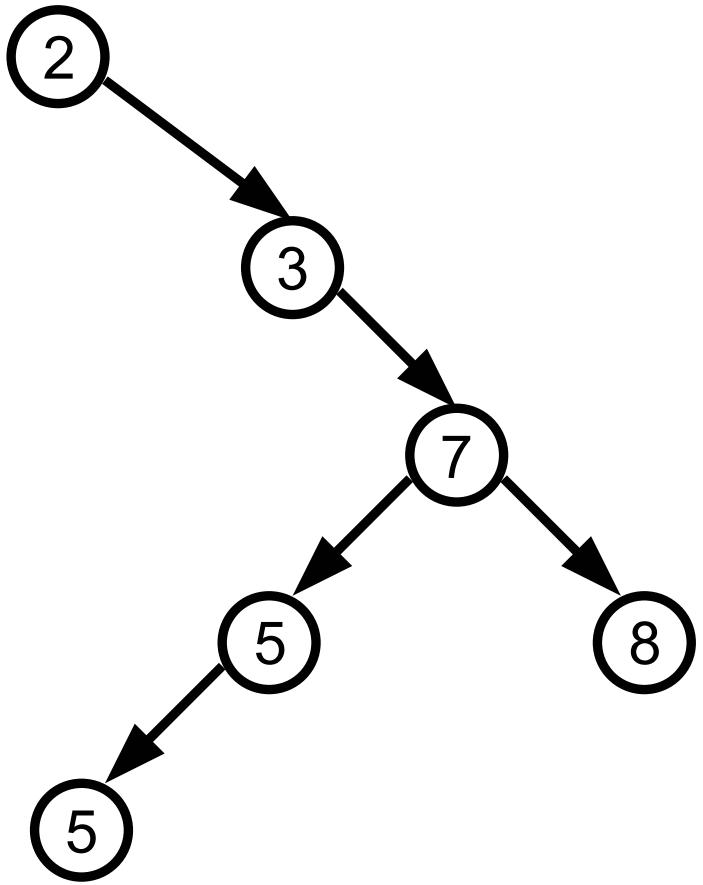
- w lewym poddrzewie znajdują się wszystkie klucze mniejsze (mniejsze bądź równe, jeżeli dopuszczamy obecność wielu takich samych kluczy w drzewie) od klucza węzła
- w prawym poddrzewie znajdują się wszystkie klucze większe od klucza węzła

Drzewa poszukiwań binarnych (BST)

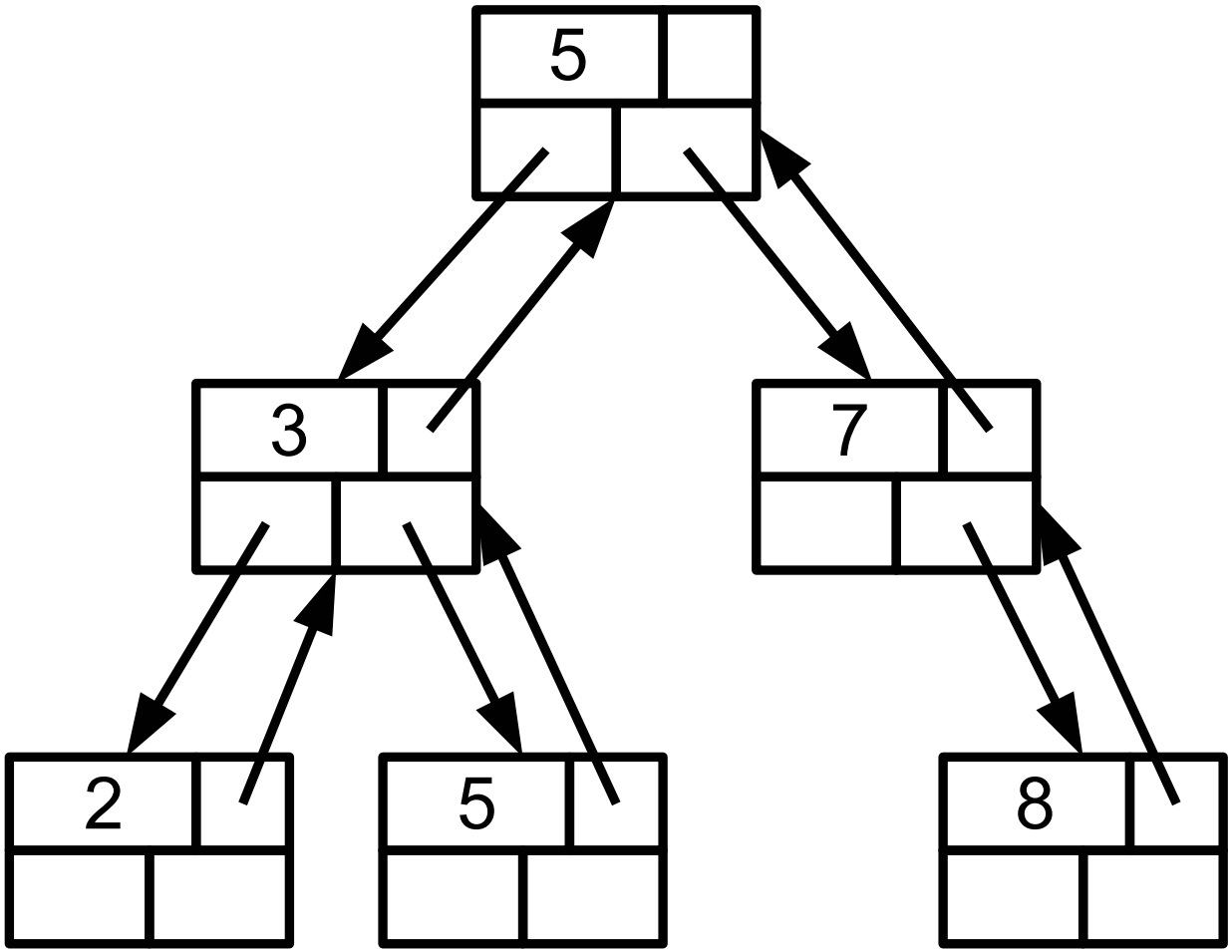
- pozwalają wykonywać operacje właściwe dla pamięci asocjacyjnych (wstawianie, usuwanie, wyszukiwanie elementu wg. klucza)
- dodatkowo pozwalają znaleźć minimum, maksimum oraz następnika i poprzednika danego elementu (następny/poprzedni element w kolejności rosnących kluczy)
- złożoność tych operacji jest proporcjonalna do wysokości drzewa
- w losowo zbudowanym drzewie binarnym będzie to $O(\log n)$
- w pesymistycznym przypadku jest to niestety $O(n)$

Drzewa poszukiwań binarnych (BST)





Drzewa poszukiwań binarnych (BST)



Przeglądanie drzewa

- wypisanie wszystkich elementów drzewa
- preorder
  - węzeł, lewe poddrzewo, prawe poddrzewo
- inorder
  - lewe poddrzewo, węzeł, prawe poddrzewo
- postorder
  - lewe poddrzewo, prawe poddrzewo, węzeł

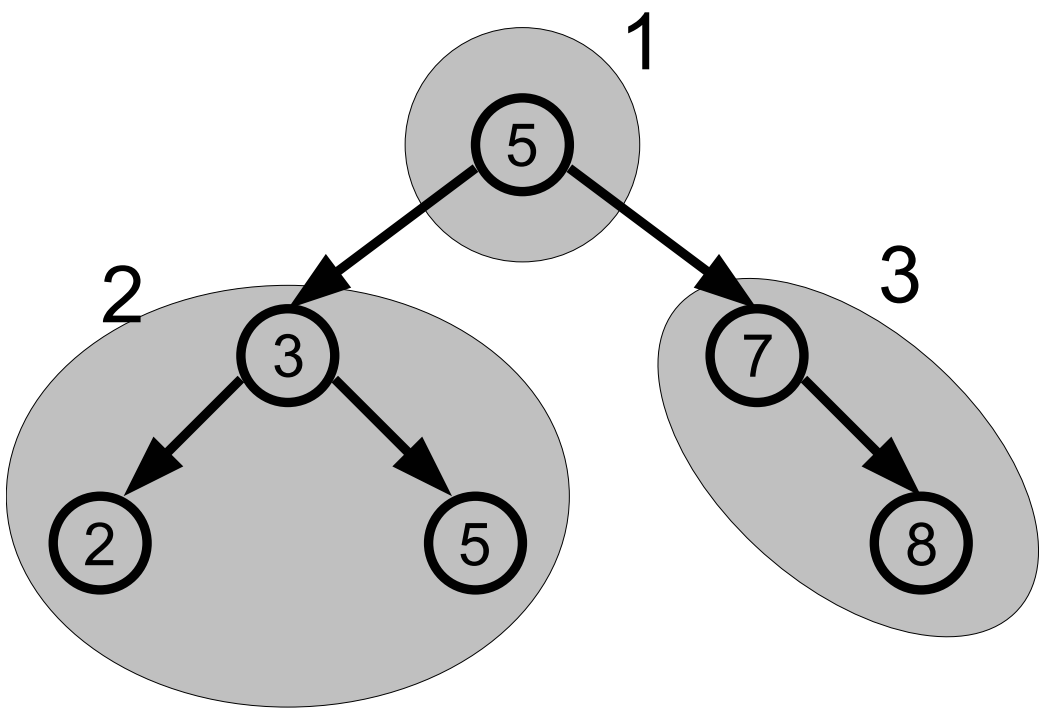
Przeglądanie preorder

- pozwala na łatwe zapisywanie/odczytywanie drzewa z pamięci zewnętrznej

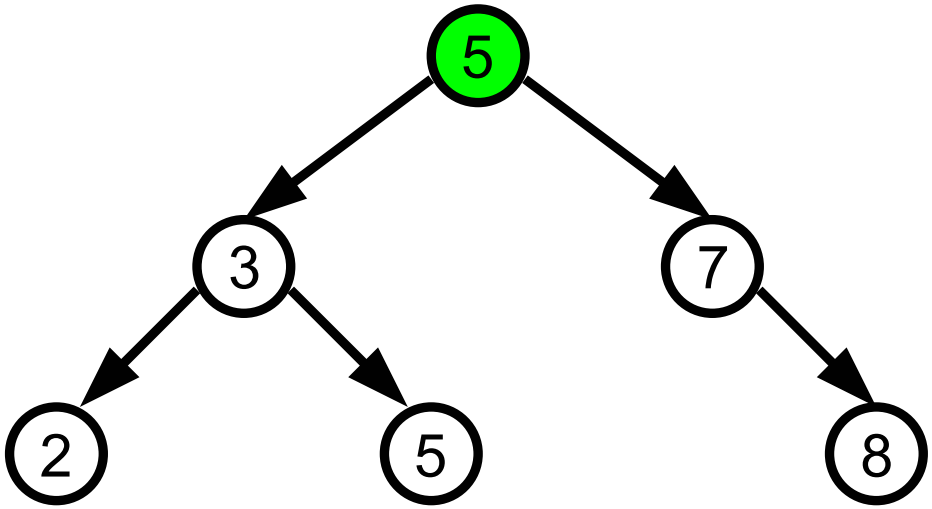
PREORDER(*root*)

1: **if** *root* $\neq$ *nullptr* **then**
2: **print** *root*
3: PREORDER(*root.left*)
4: PREORDER(*root.right*)
5: **else** {tylko jeżeli np. zapisujemy drzewo na dysk}
6: **print** *nullptr*
7: **end if**

Przeglądanie preorder

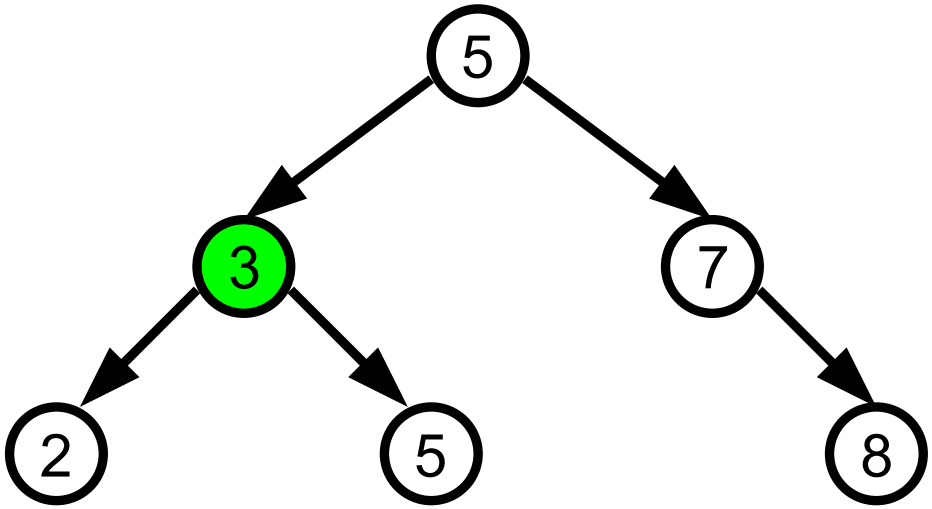


Przeglądanie preorder



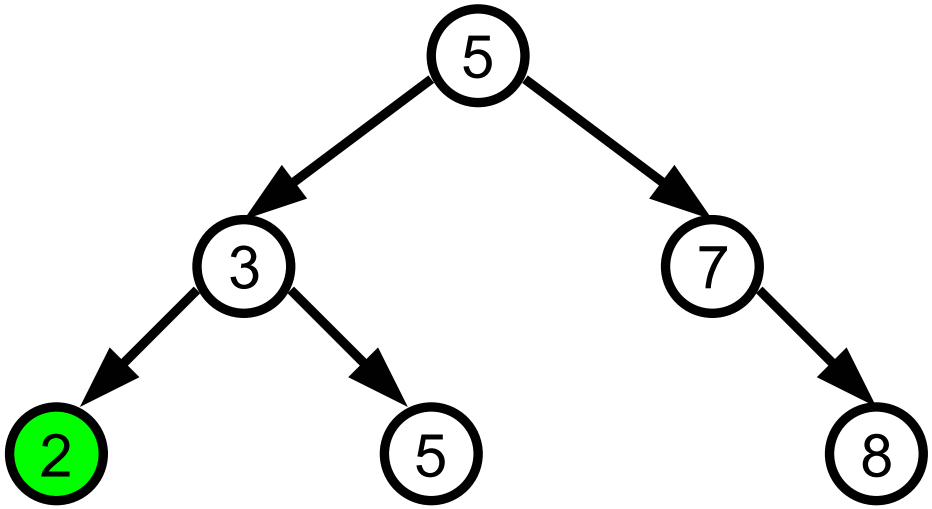
5

Przeglądanie preorder



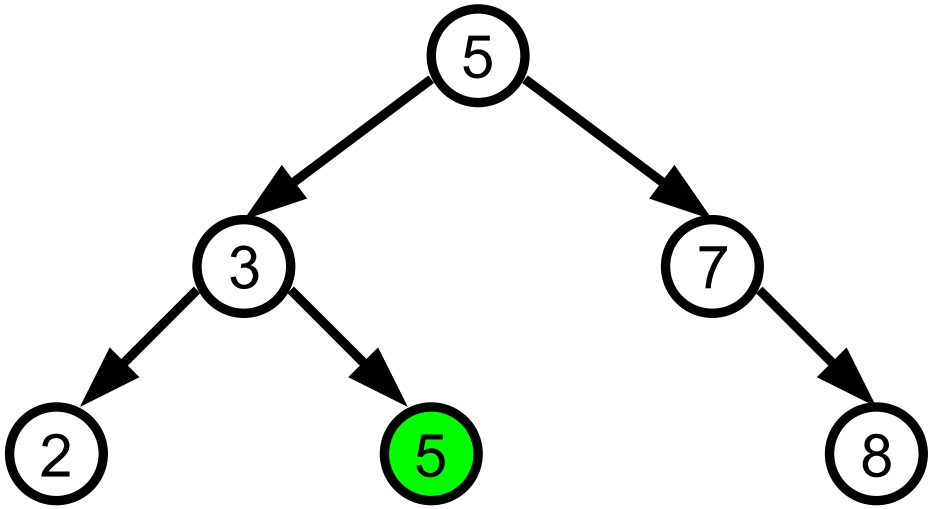
5, 3

Przeglądanie preorder



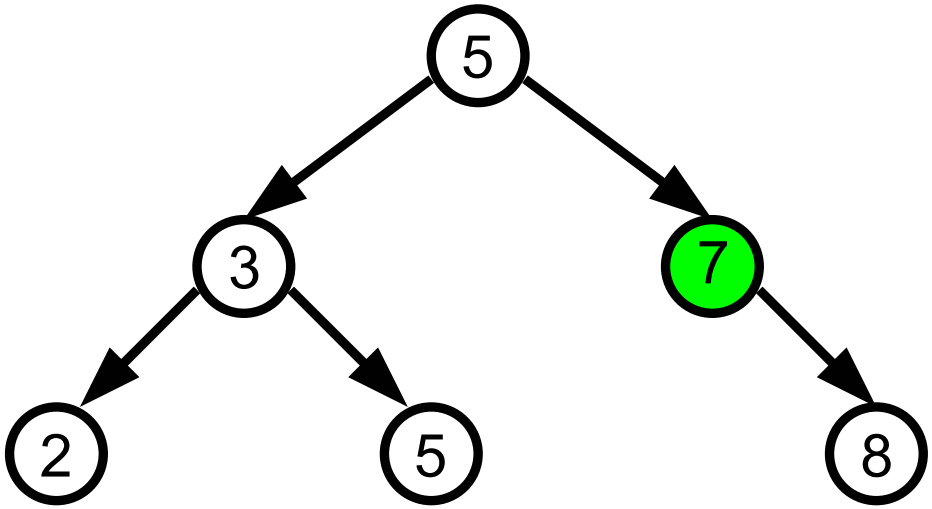
5, 3, 2

Przeglądanie preorder



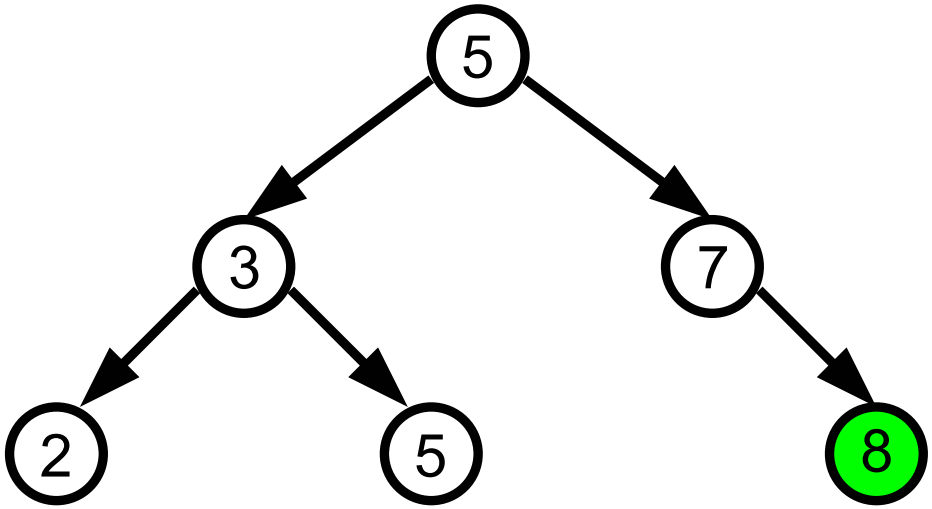
5, 3, 2, 5

Przeglądanie preorder



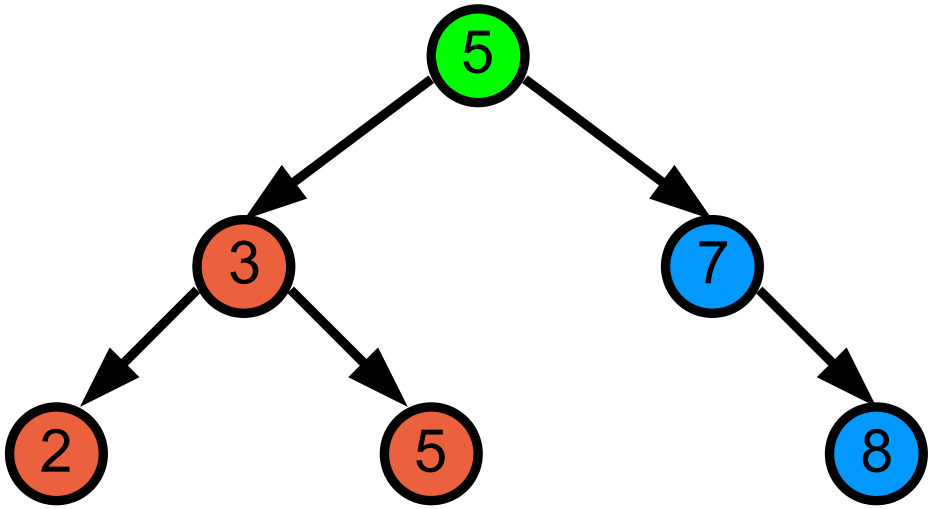
5, 3, 2, 5, 7

Przeglądanie preorder



5, 3, 2, 5, 7, 8

Przeglądanie preorder



5, 3, 2, 5, 7, 8

Przeglądanie preorder – iteracyjnie

PREORDER(*root*)

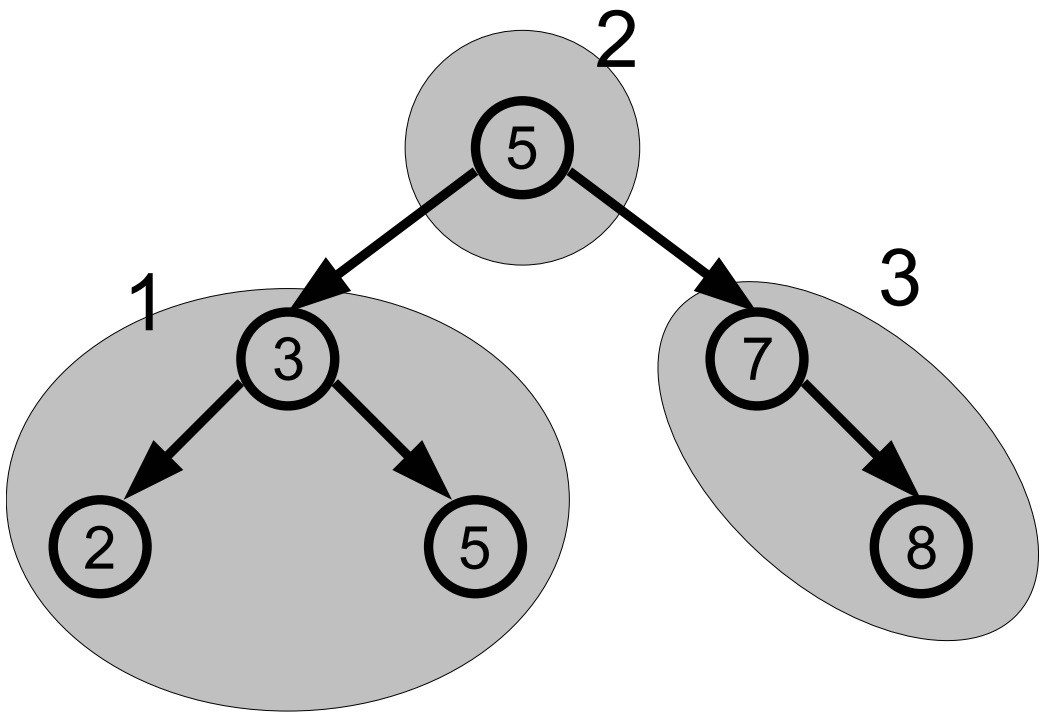
```
1: stos = pusty stos
2: stos.push(root)
3: while not stos.empty() do
4:         w = stos.top()
5:         stos.pop()
6:         print w.value
7:         if w.right ≠ nullptr then stos.push(w.right)
8:         if w.left ≠ nullptr then stos.push(w.left)
9: end while
```

Przeglądanie inorder

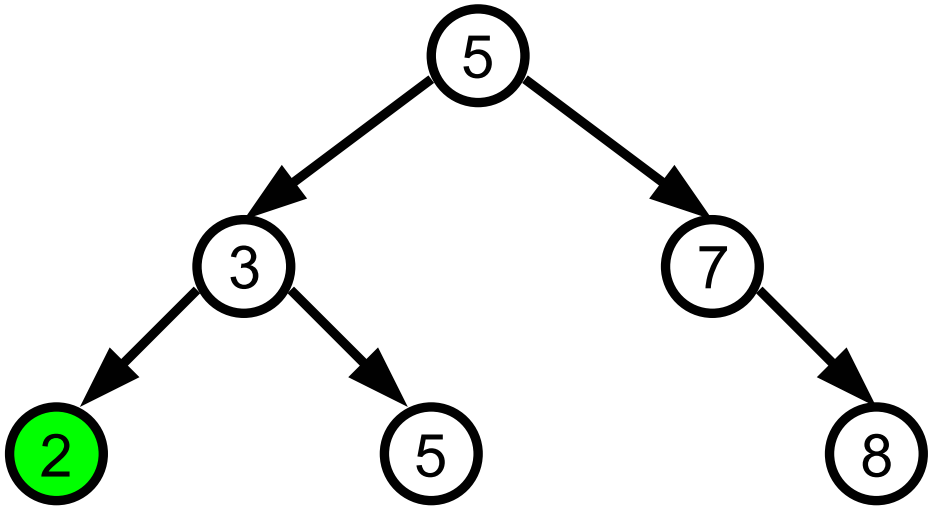
- wypisuje elementy w kolejności niemalejących kluczy

```
INORDER(root)
1: if root ≠ nullptr then
2:         INORDER(root.left)
3:         print root
4:         INORDER(root.right)
5: end if
```

Przeglądanie inorder

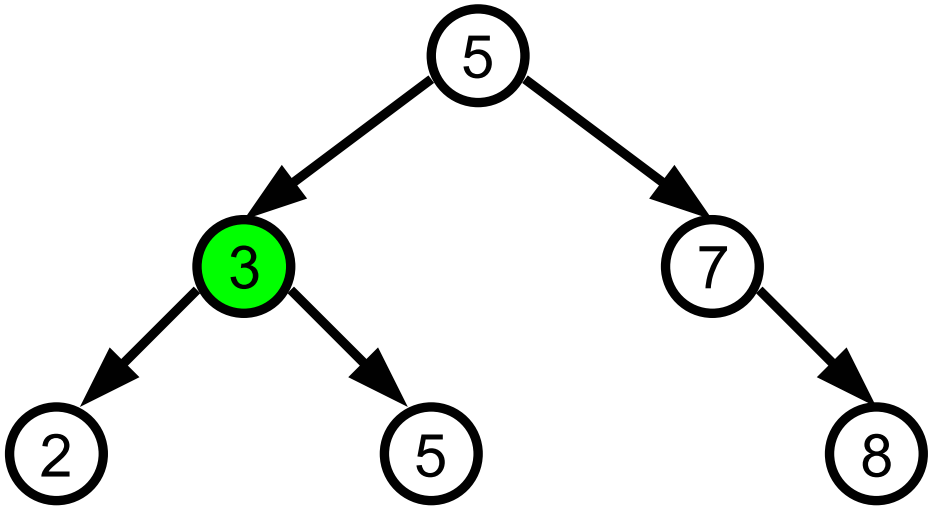


Przeglądanie inorder



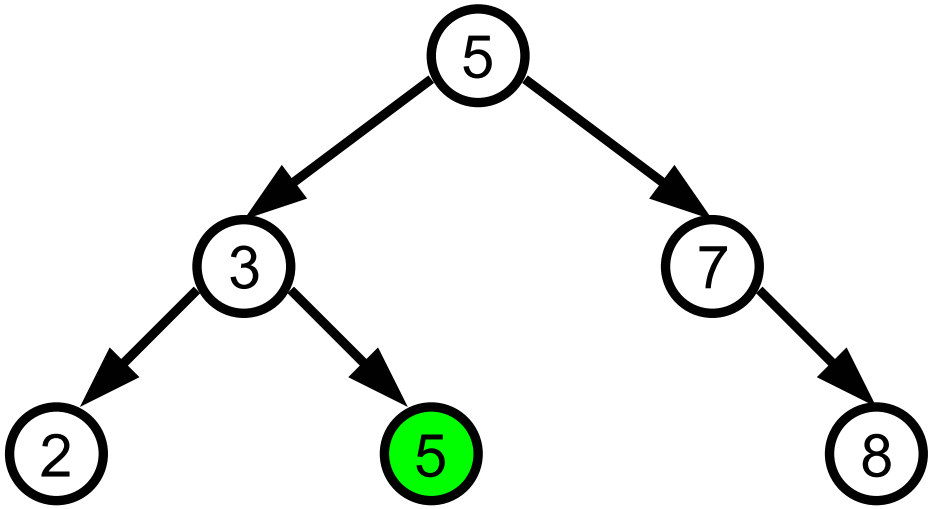
2

Przeglądanie inorder



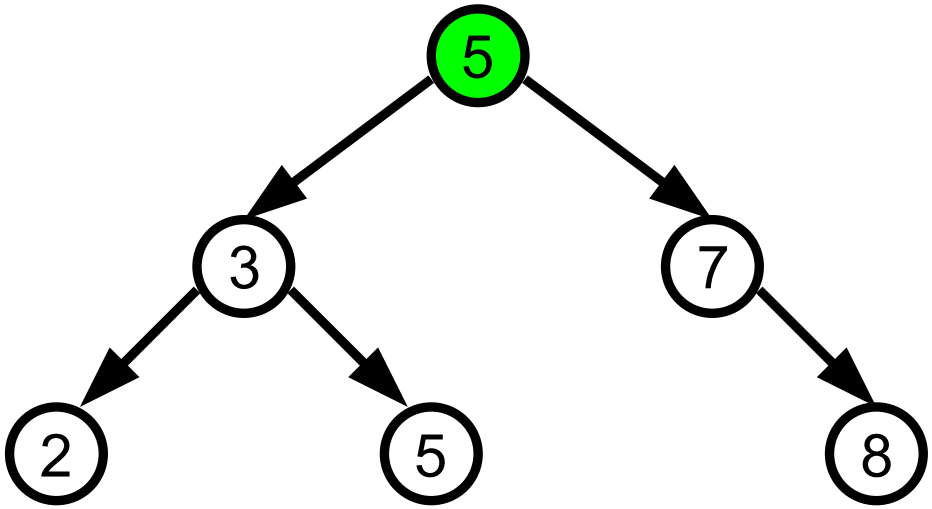
2, 3

Przeglądanie inorder



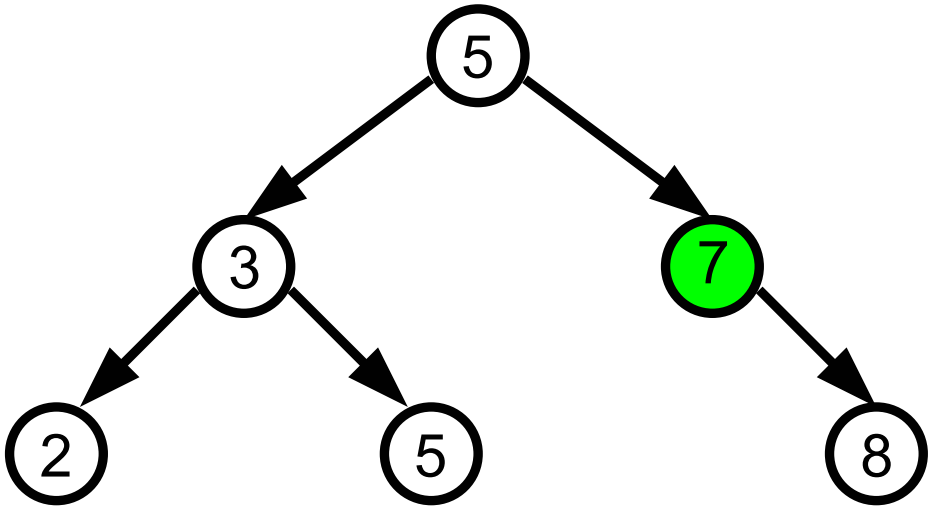
2, 3, 5

Przeglądanie inorder



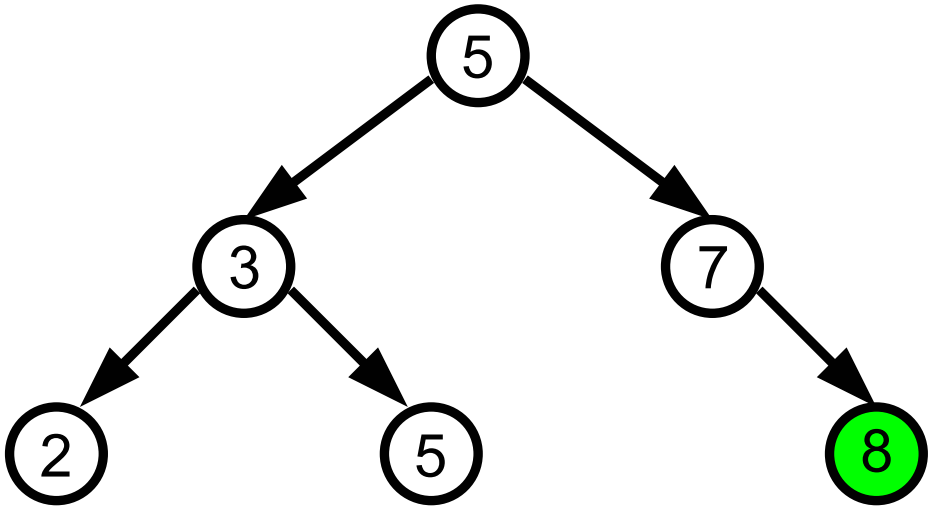
2, 3, 5, 5

Przeglądanie inorder



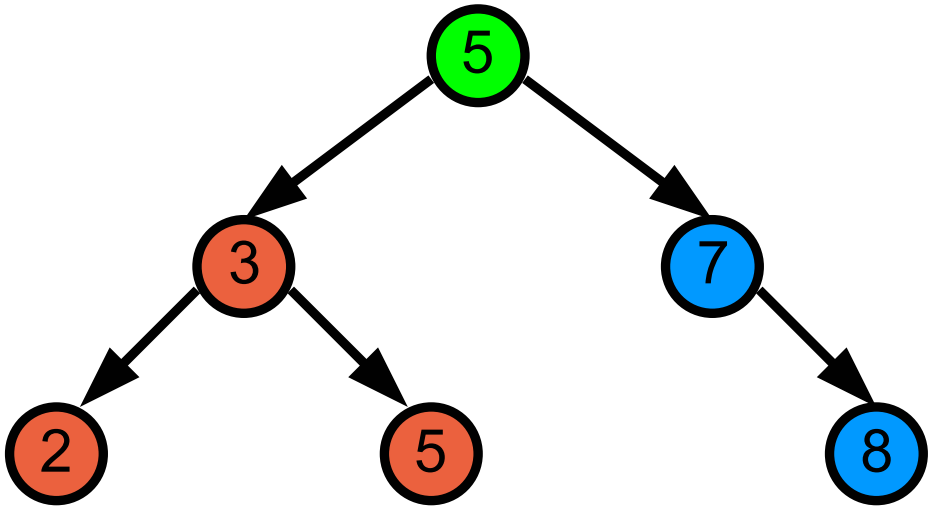
2, 3, 5, 5, 7

Przeglądanie inorder



2, 3, 5, 5, 7, 8

Przeglądanie inorder



2, 3, 5, 5, 7, 8

Przeglądanie postorder

```
POSTORDER(root)
1: if root ≠ nullptr then
2:         POSTORDER(root.left)
3:         POSTORDER(root.right)
4:         print root
5: end if
```

Wysokość drzewa

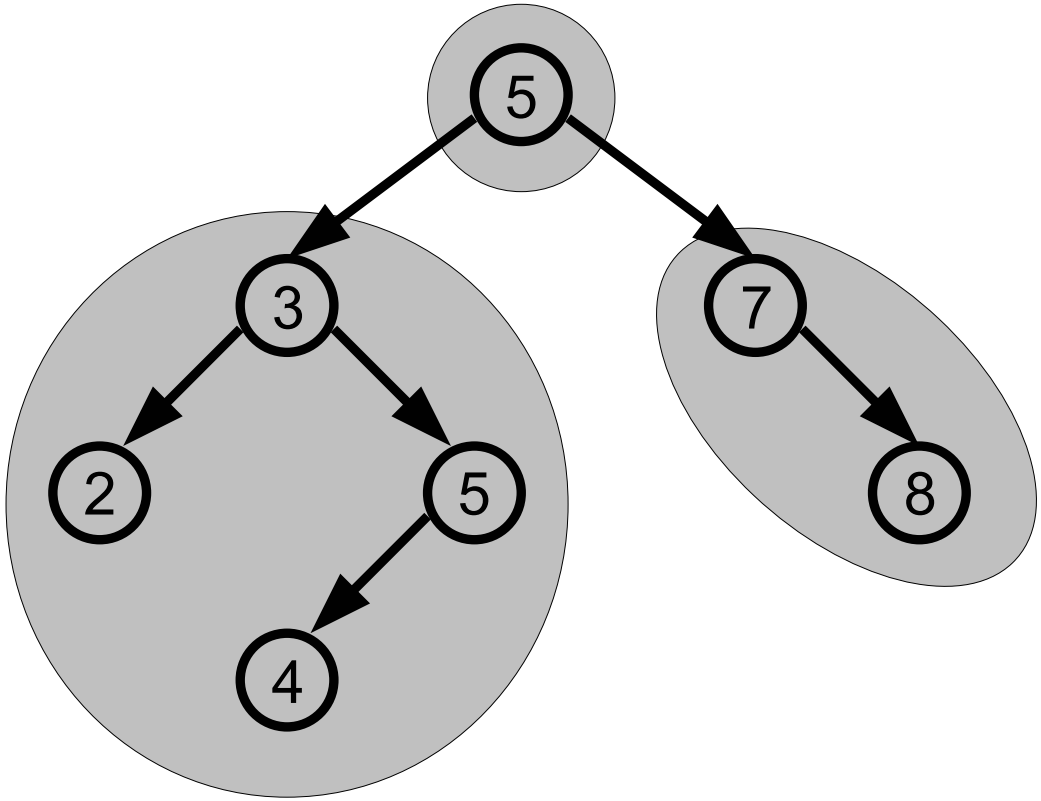
- wysokość drzewa o korzeniu *root* jest wysokością większego z jego poddrzew powiększoną o 1 (drzewo, tak jak listy jest strukturą rekurencyjną)

`WYSOKOSC(`*root*`)`

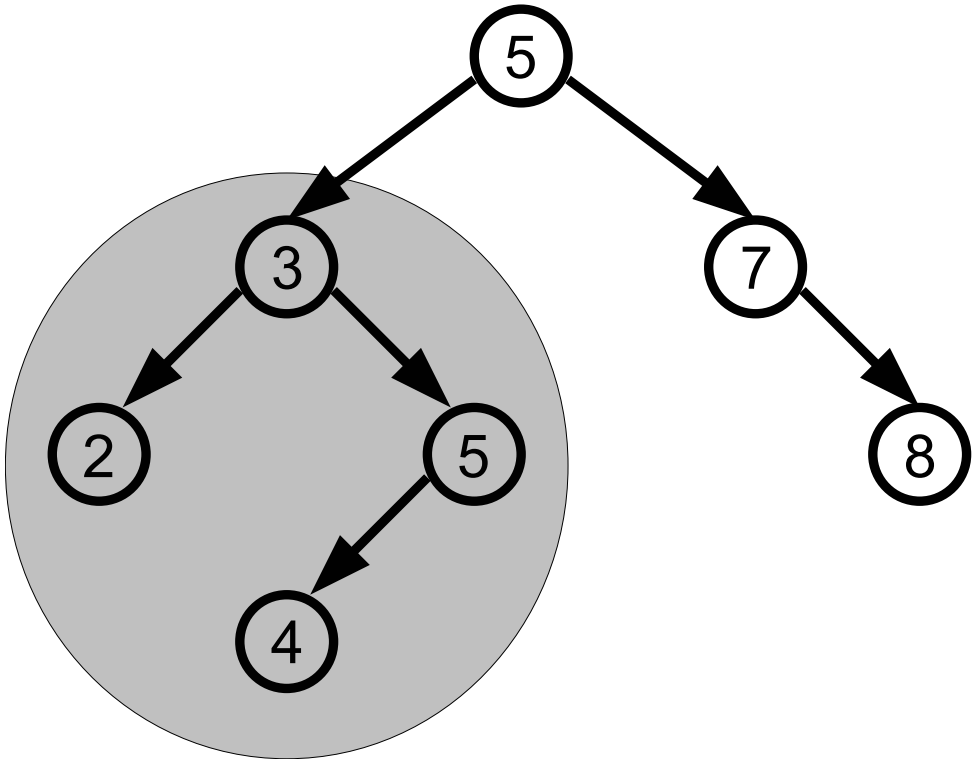
1: **if** $root = nullptr$ **then**
2: **return** 0
3: **end if**
4: $hl = WYSOKOSC(root.left)$
5: $hr = WYSOKOSC(root.right)$
6: **return** $\max(hl, hr) + 1$

Wysokość drzewa

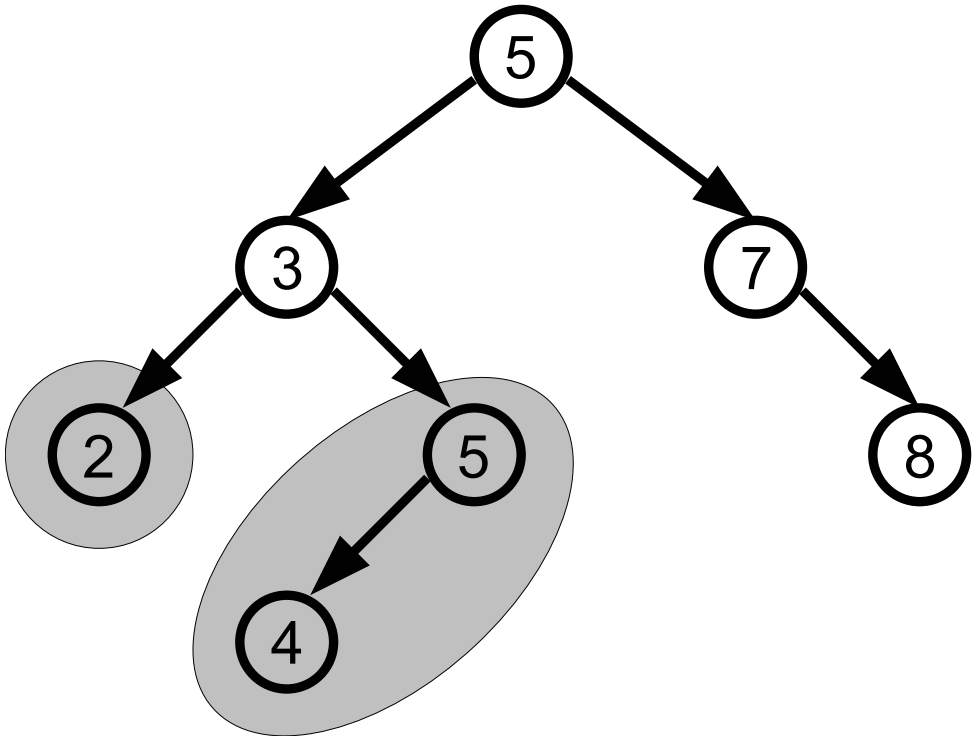
WYS = max(WYS(root.left), WYS(root.right)) + 1



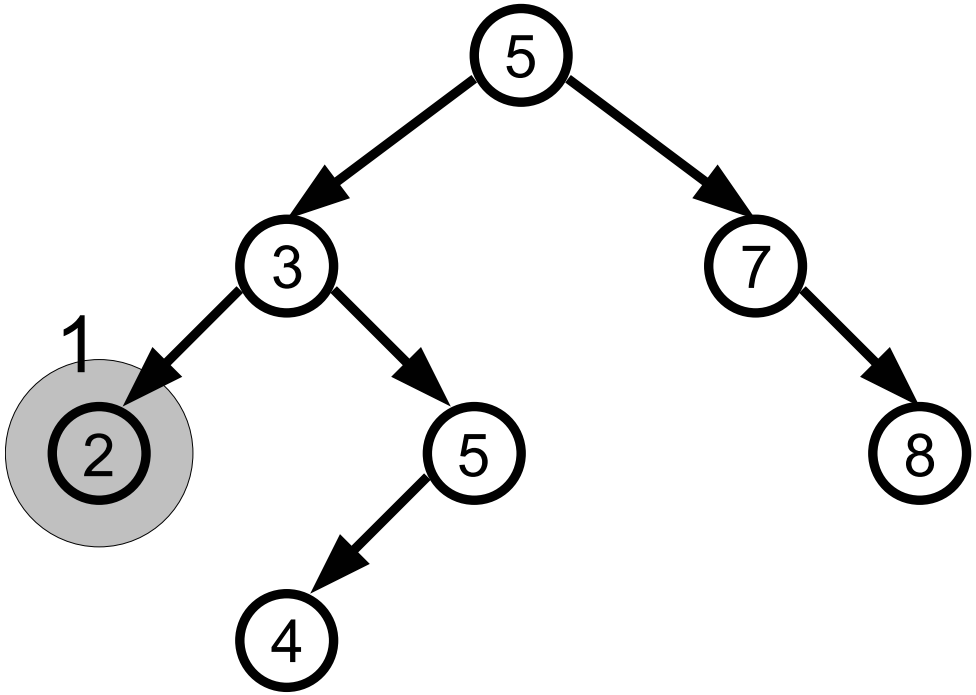
Wysokość drzewa



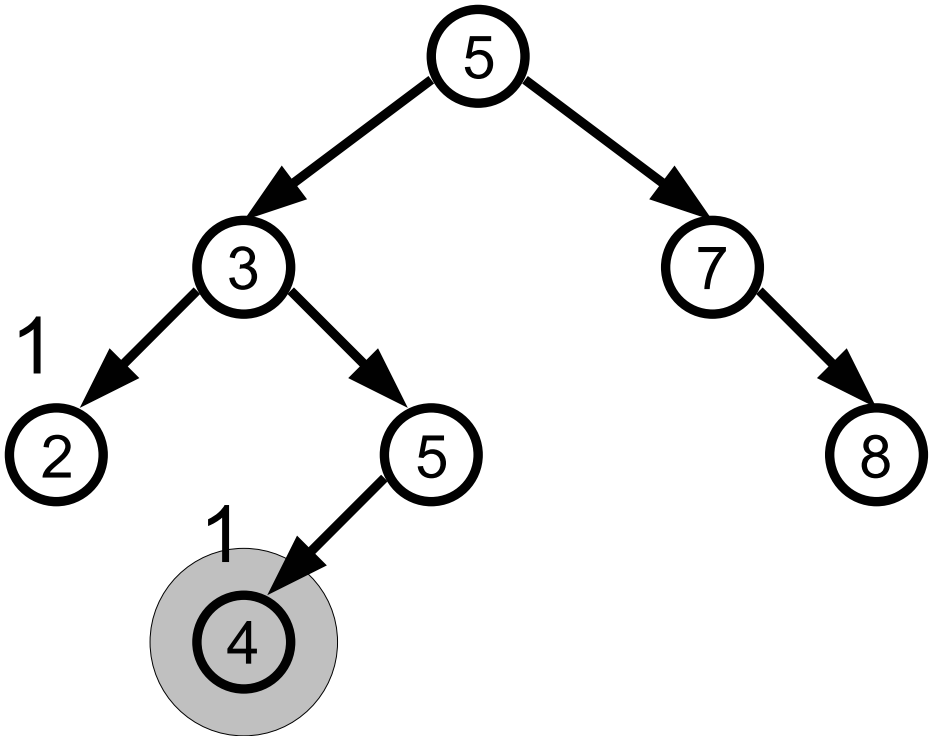
Wysokość drzewa



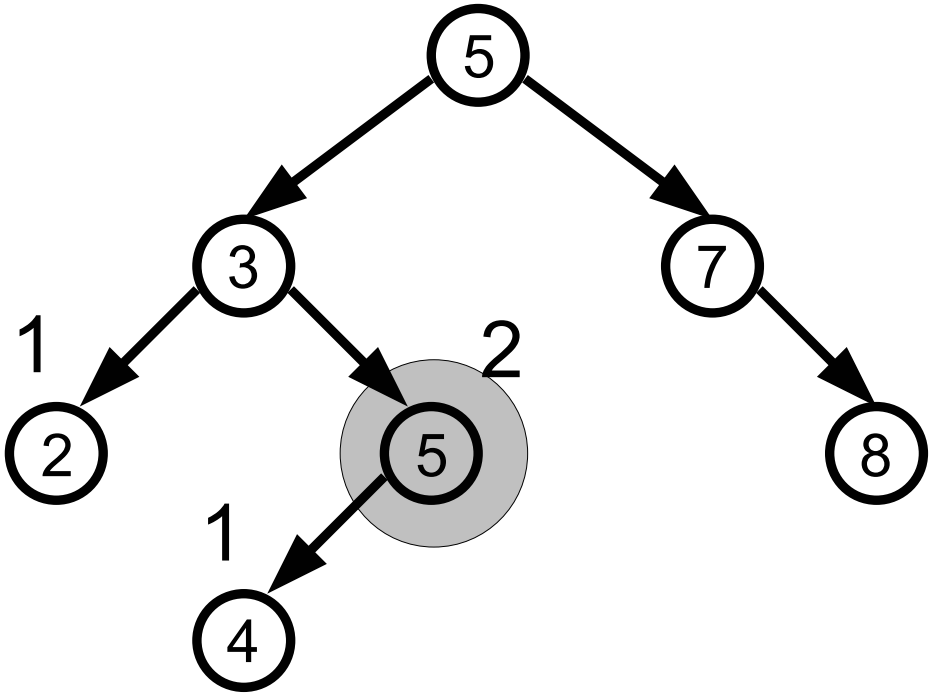
Wysokość drzewa



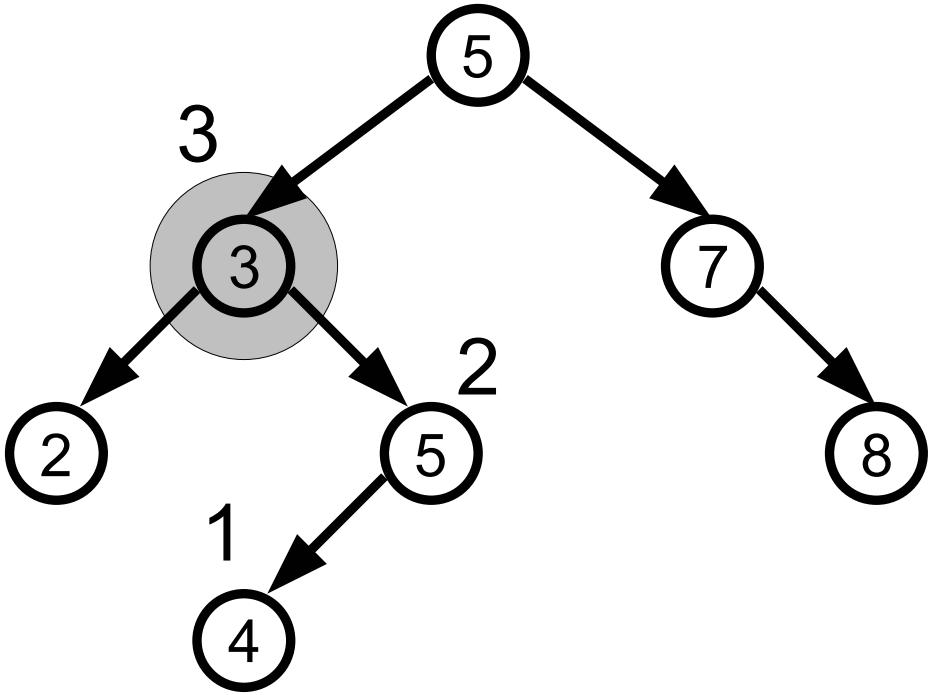
Wysokość drzewa



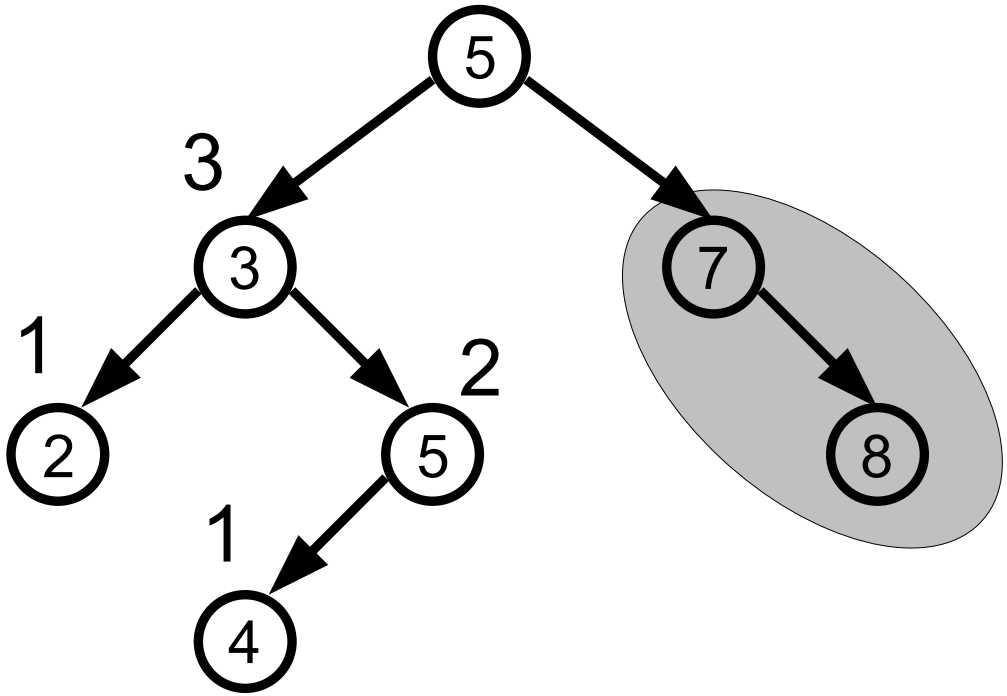
Wysokość drzewa



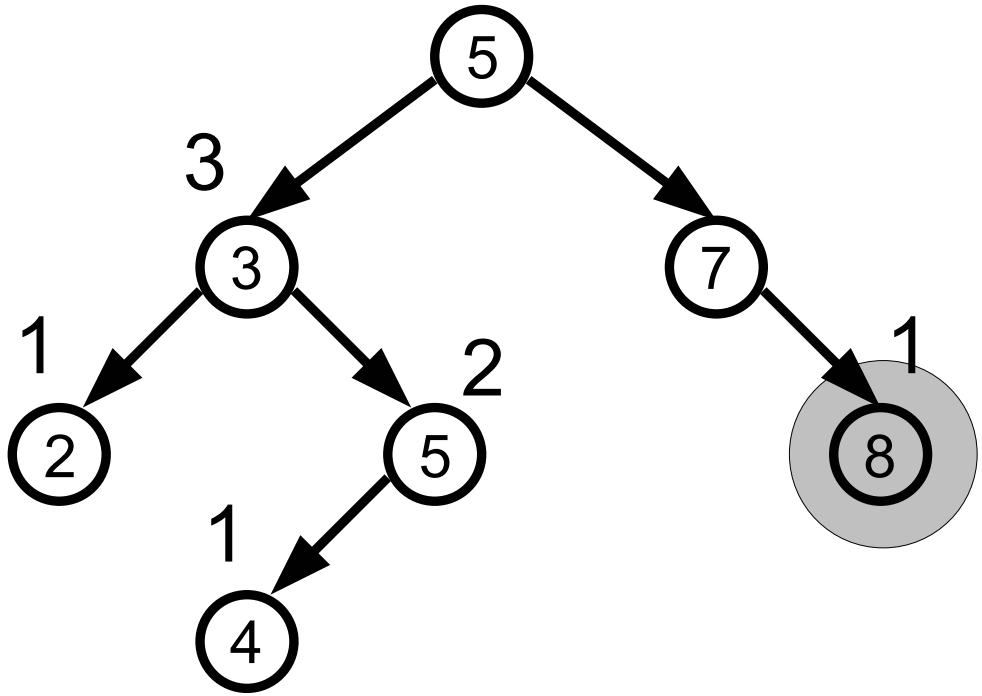
Wysokość drzewa



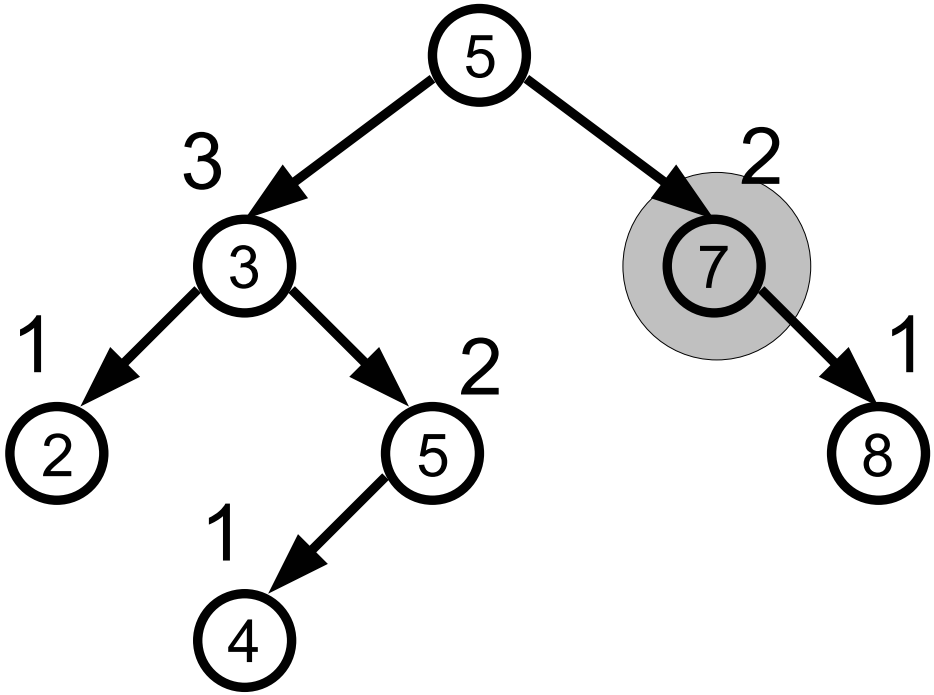
Wysokość drzewa



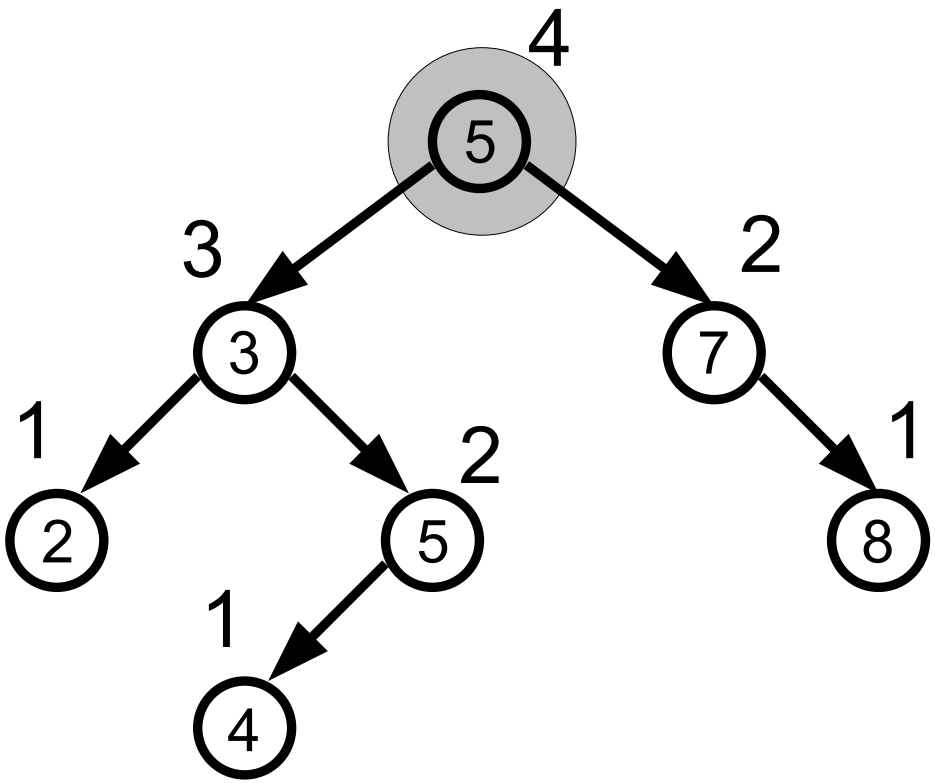
Wysokość drzewa



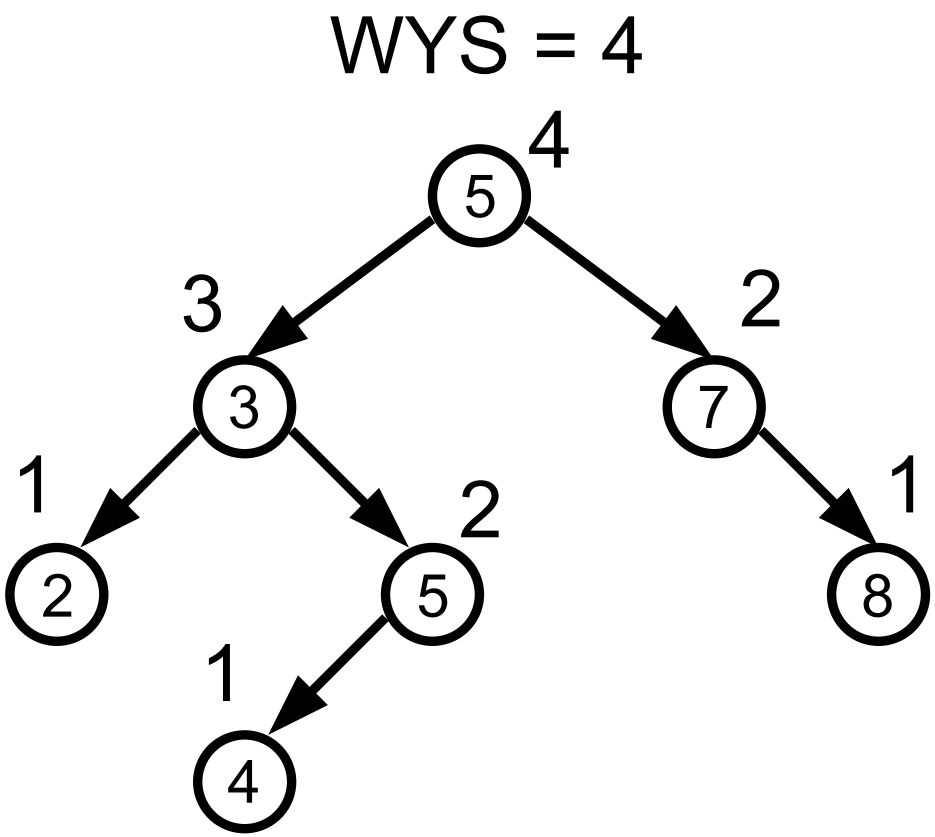
Wysokość drzewa



Wysokość drzewa



Wysokość drzewa



Wyszukiwanie elementu

- chcemy znaleźć element o podanym kluczu

```
ZNAJDZ(root, klucz)
1: if root = nullptr or root.key = klucz then
2:         return root
3: end if
4: if klucz < root.key then
5:         return ZNAJDZ(root.left, klucz)
6: else
7:         return ZNAJDZ(root.right, klucz)
8: end if
```

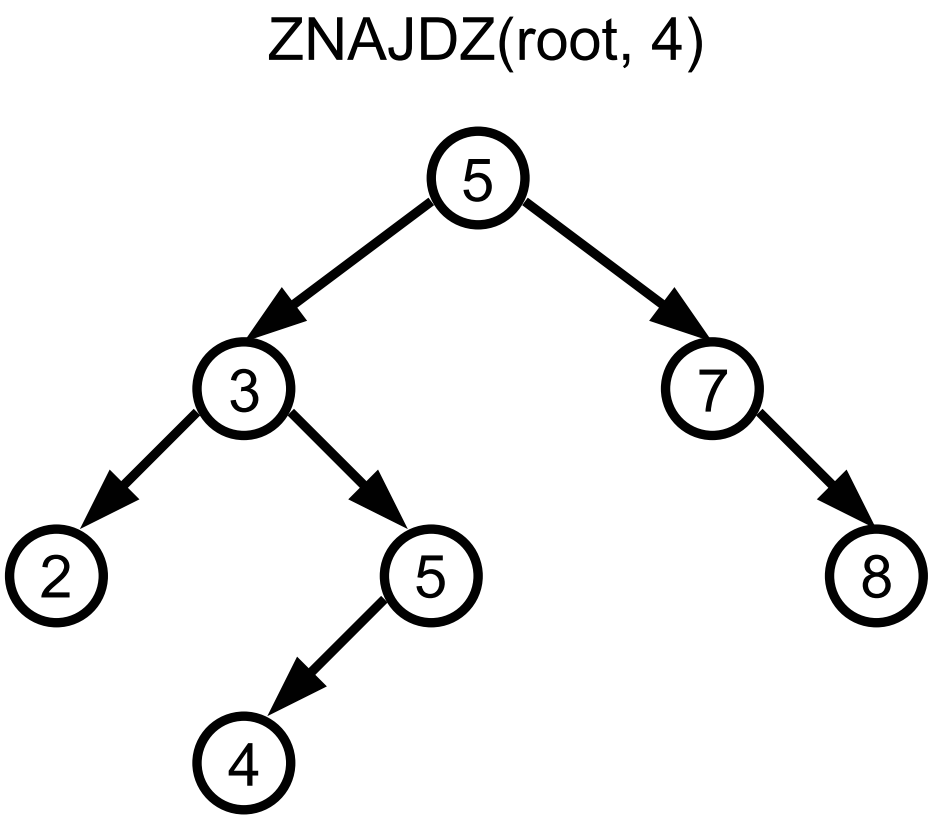
Wyszukiwanie elementu

- wersja iteracyjna

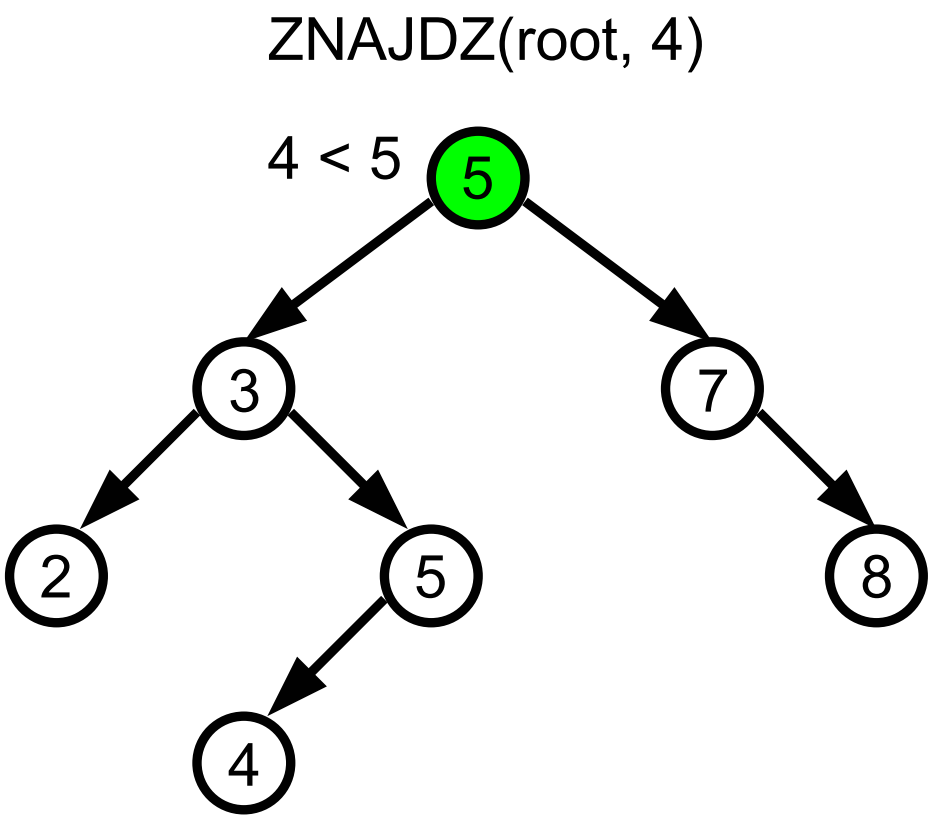
`ZNAJDZ(`*root*`,` *klucz*`)`

```
1: curr = root
2: while curr ≠ nullptr and curr.key ≠ klucz do
3:         if klucz < curr.key then
4:                 curr = curr.left
5:         else
6:                 curr = curr.right
7:         end if
8: end while
9: return curr
```

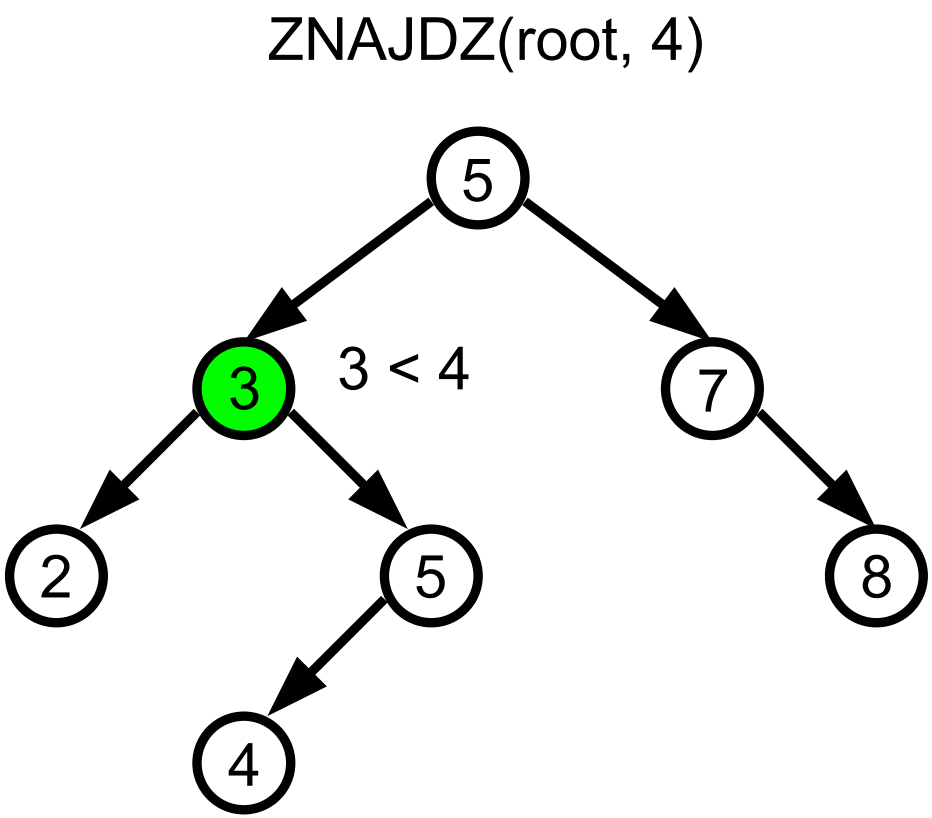
Wyszukiwanie



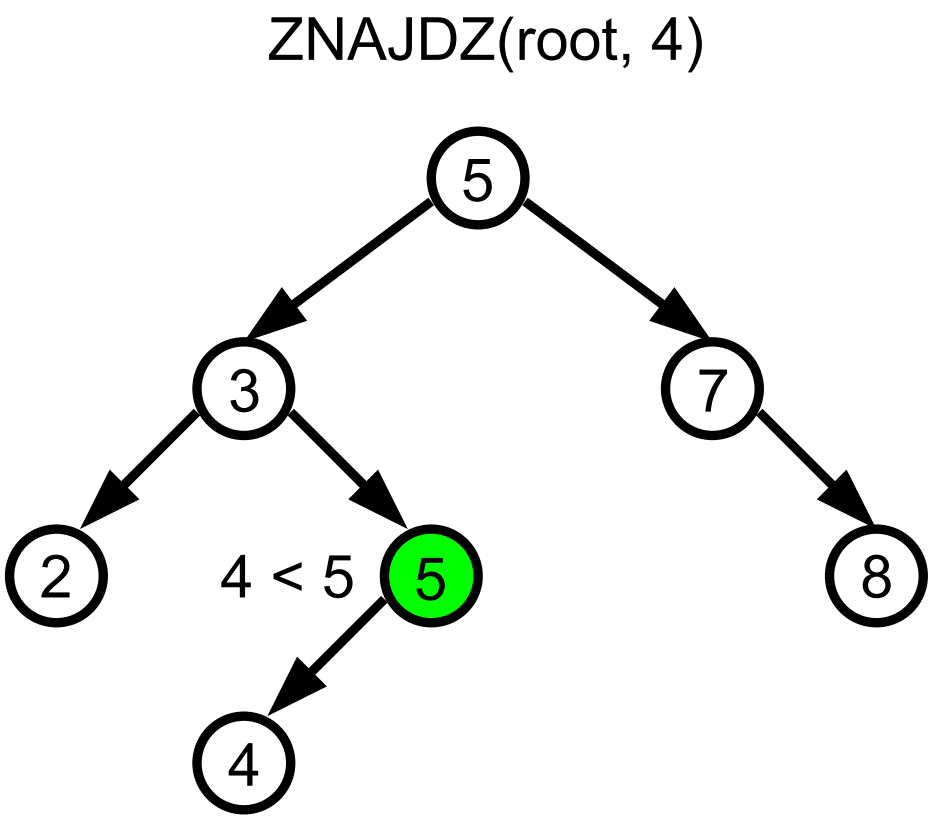
Wyszukiwanie



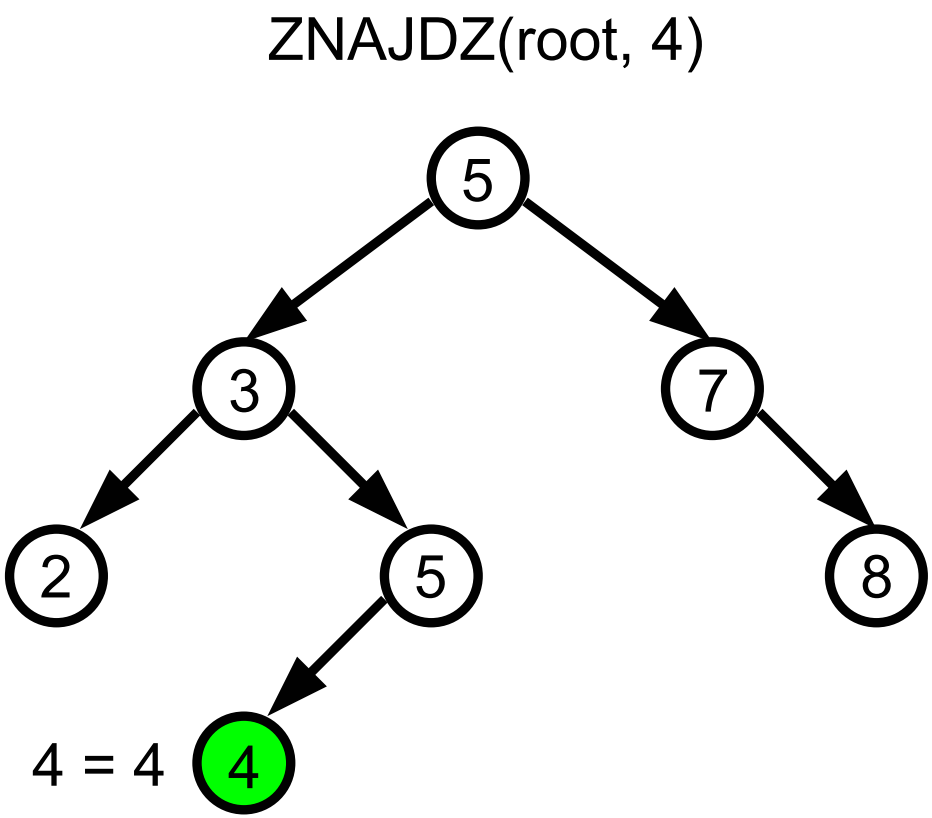
Wyszukiwanie



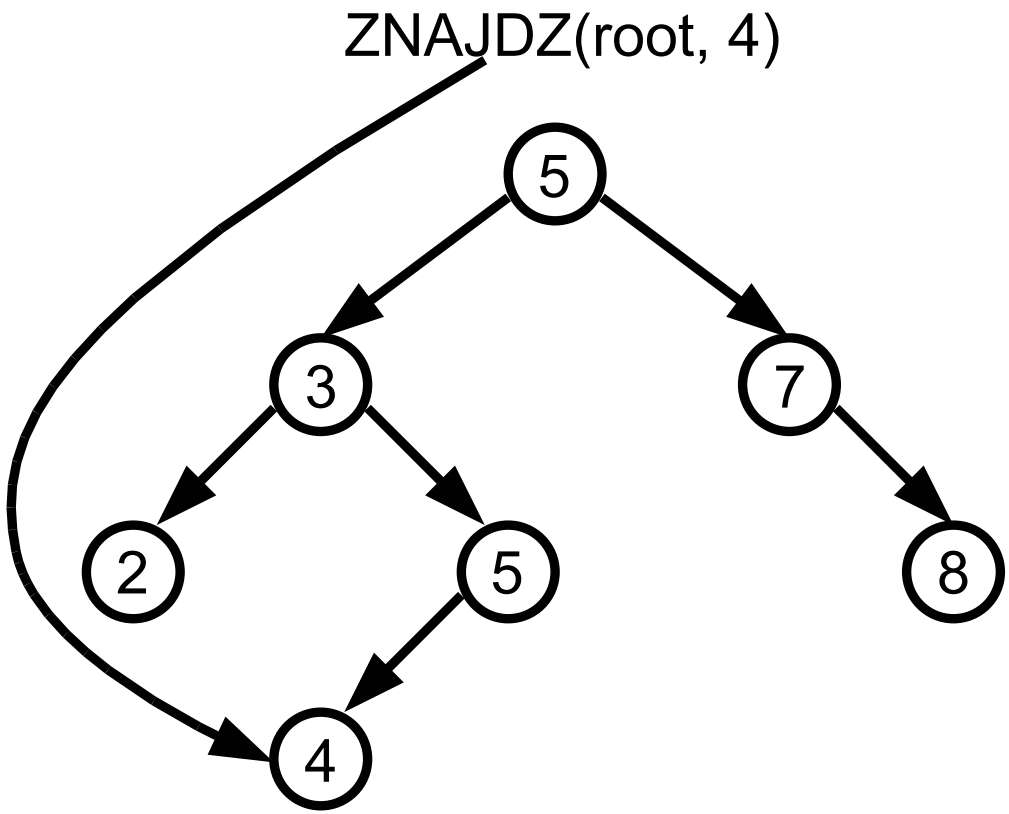
Wyszukiwanie



Wyszukiwanie



Wyszukiwanie



Minimum

- szukamy węzła o najmniejszym kluczu w drzewie (rekurencyjnie)

`MINIMUM(`*root*`)`

1: **if** *root* = *nullptr* **or** *root.left* = *nullptr* **then**
2: **return** *root*
3: **else**
4: **return** `MINIMUM(`*root*`)`
5: **end if**

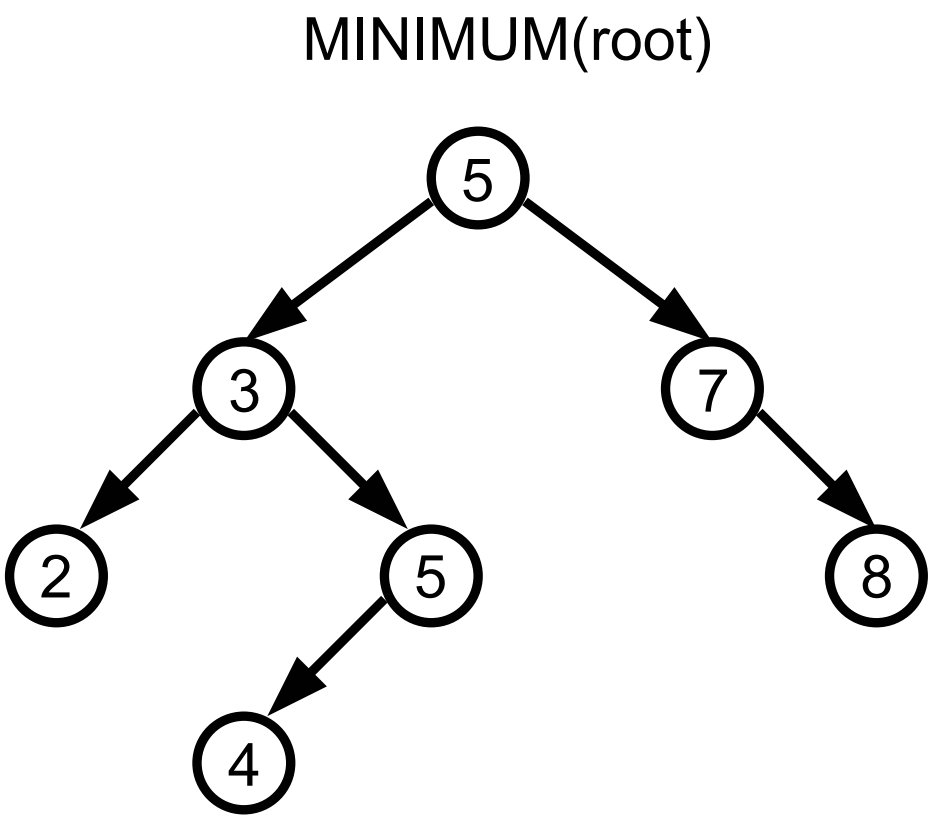
Minimum

- szukamy węzła o najmniejszym kluczu w drzewie (iteracyjnie)

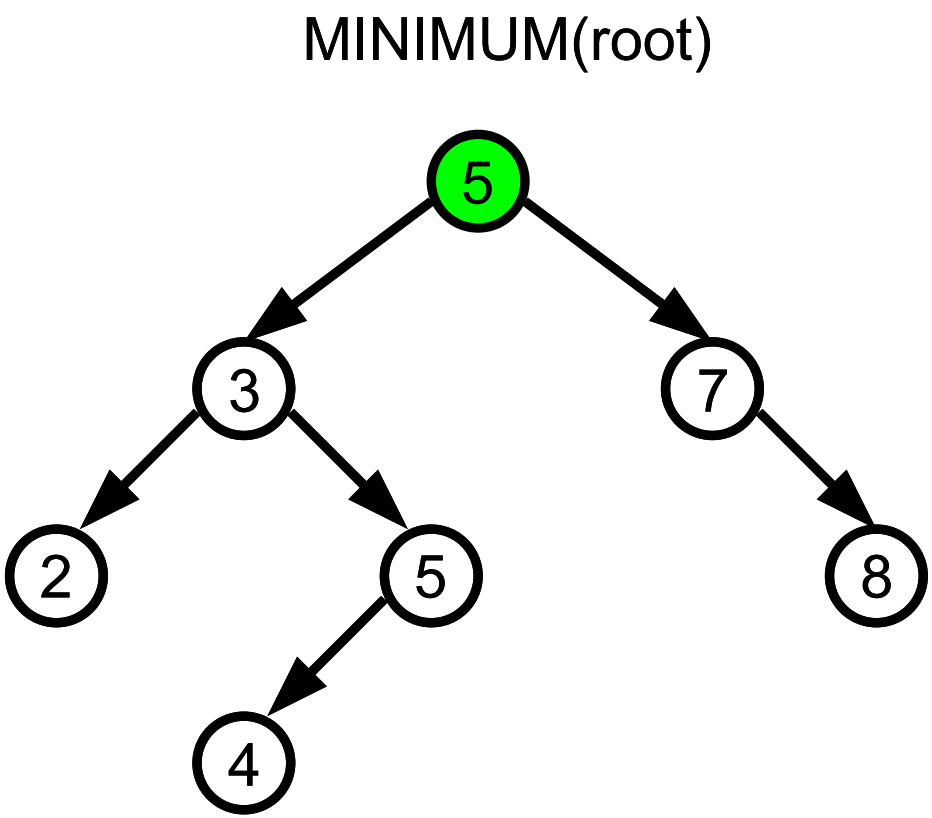
`MINIMUM(`*root*`)`

1: **if** $root = nullptr$ **then**
2: **return** *nullptr*
3: **end if**
4: $curr = root$
5: **while** $curr.left \neq nullptr$ **do**
6: $curr = curr.left$
7: **end while**
8: **return** *curr*

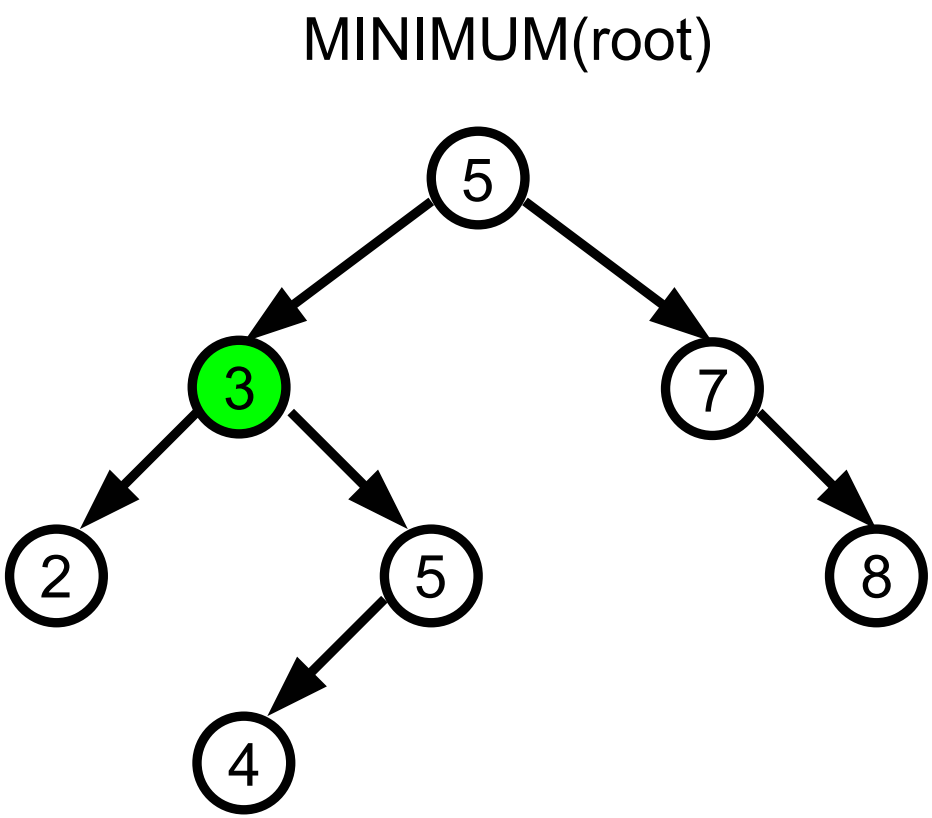
Minimum



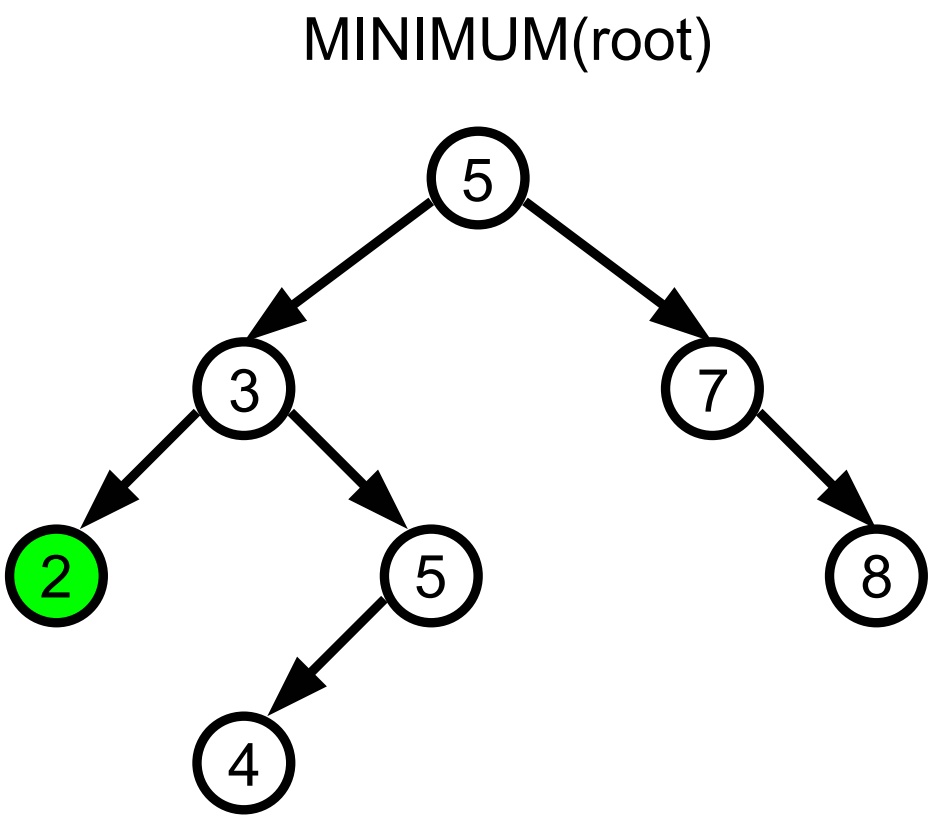
Minimum



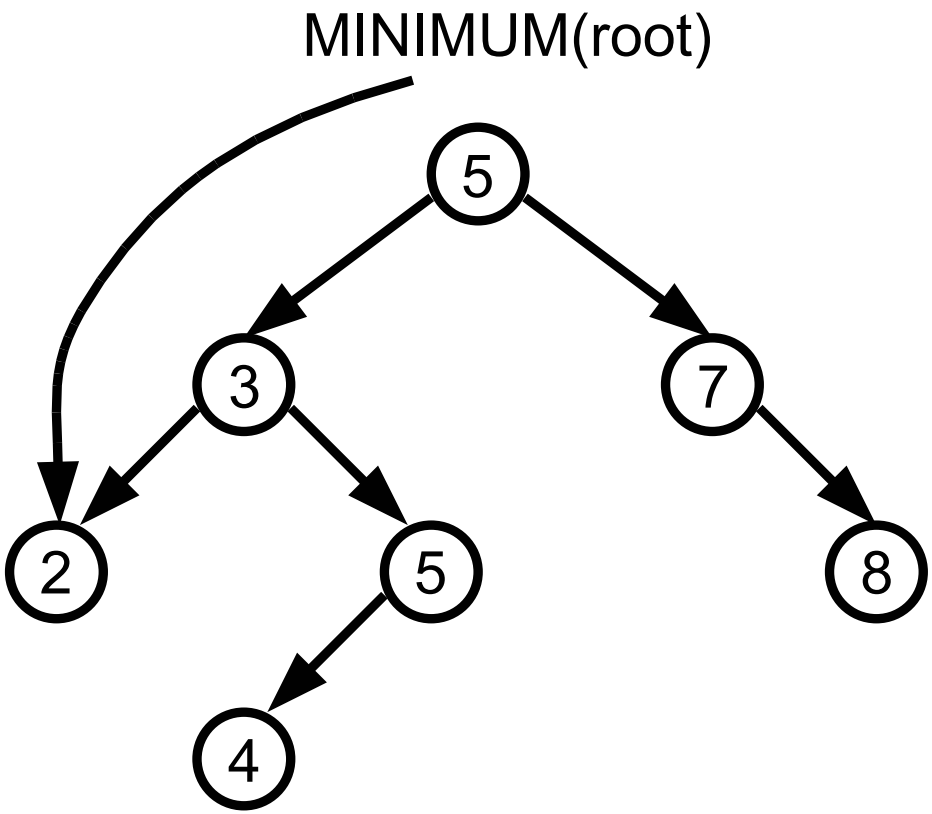
Minimum



Minimum



Minimum



Maksimum

- odnalezienie elementu o maksymalnym kluczu wygląda analogicznie:

`MAXIMUM(`*root*`)`

1: **if** $root = nullptr$ **then**
2: **return** *nullptr*
3: **end if**
4: $curr = root$
5: **while** $curr.right \neq nullptr$ **do**
6: $curr = curr.right$
7: **end while**
8: **return** *curr*

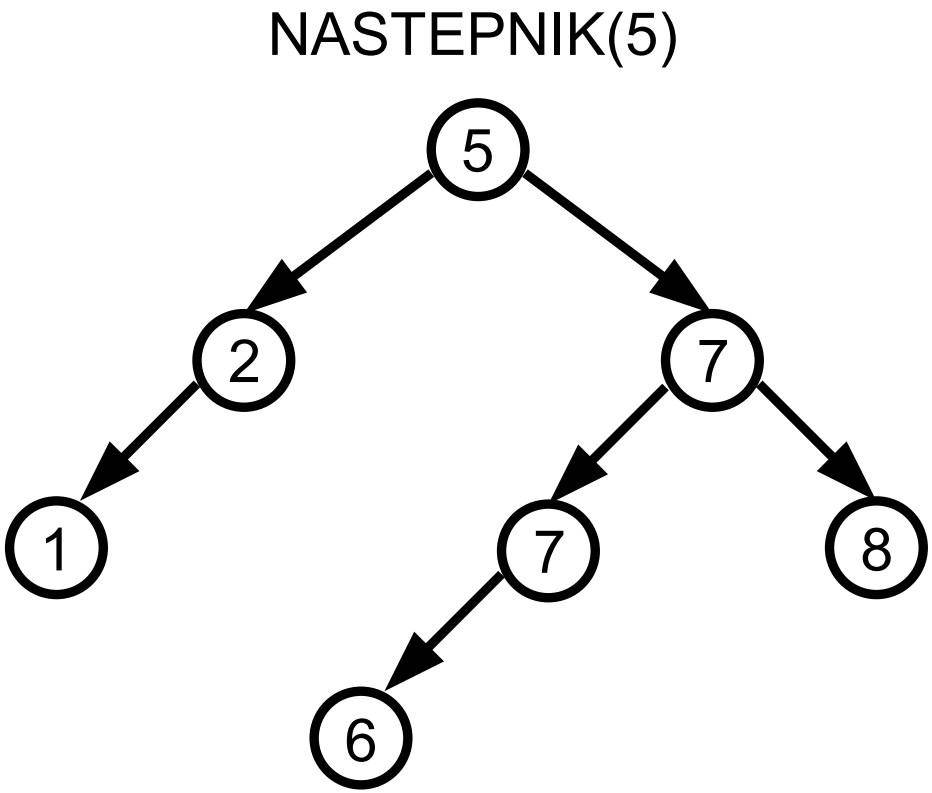
Następnik

- należy znaleźć węzeł o kolejnym kluczu (najmniejszym spośród większych od bieżącego)

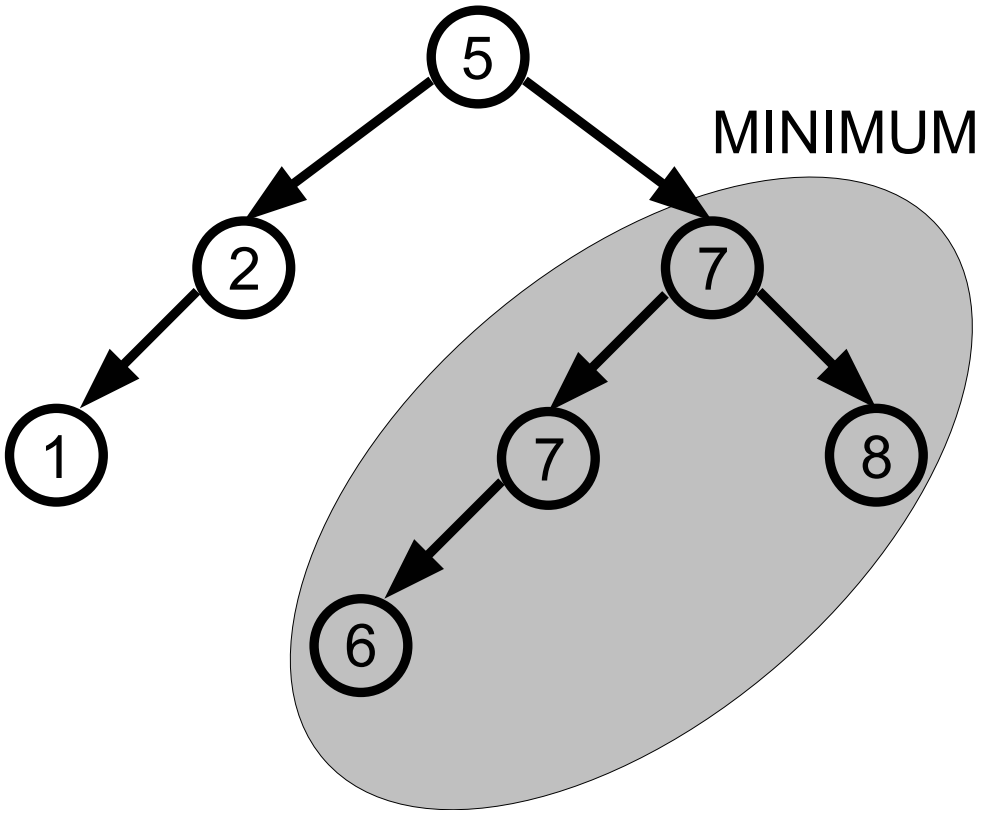
`NASTEPNIK(`*wezel*`)`

```
1:  curr = wezel
2:  if curr.right ≠ nullptr then
3:          return MINIMUM(curr.right)
4:  end if
5:  parent = curr.parent
6:  while parent ≠ nullptr and curr = parent.right do
7:          curr = parent
8:          parent = parent.parent
9:  end while
10: return parent
```

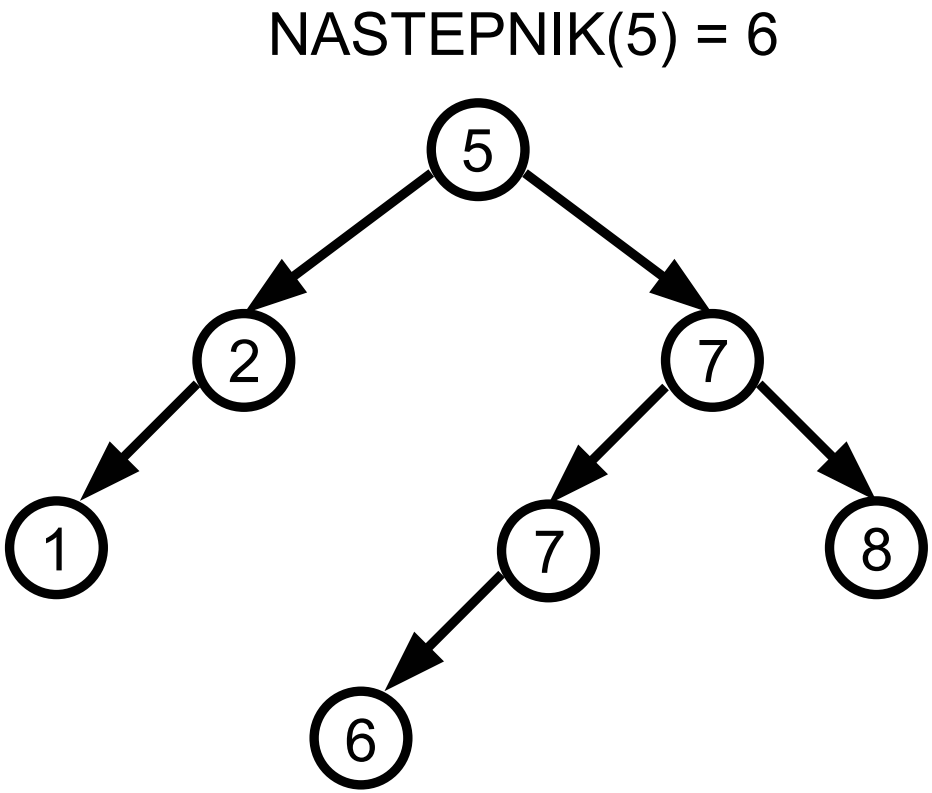
Następnik



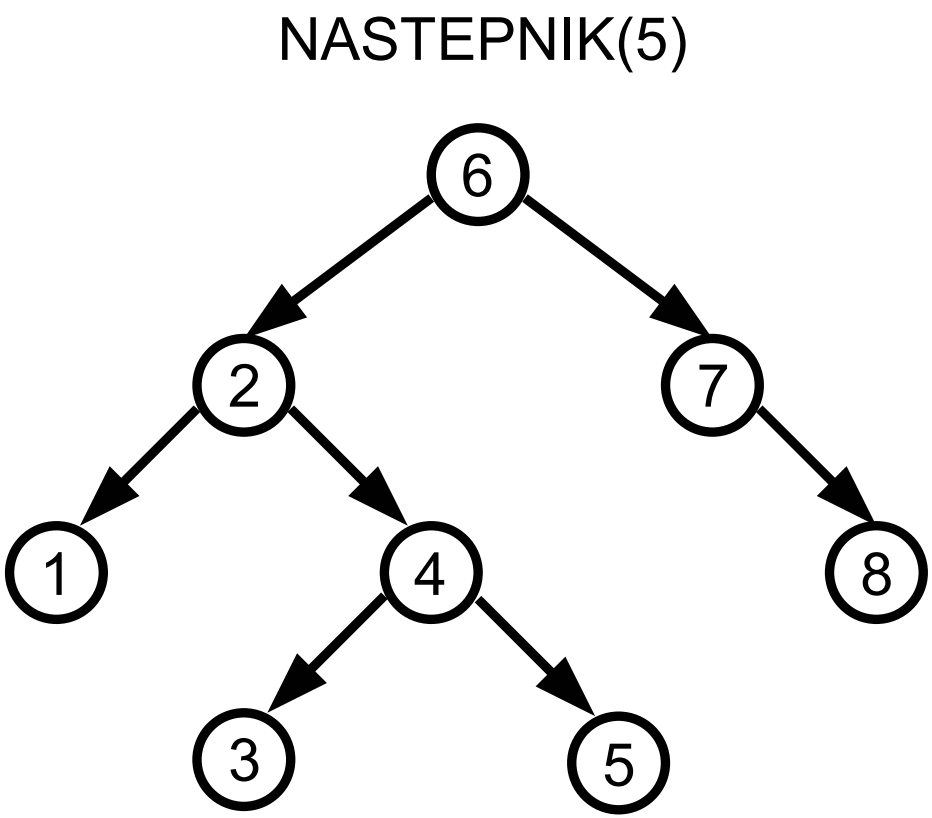
Następnik



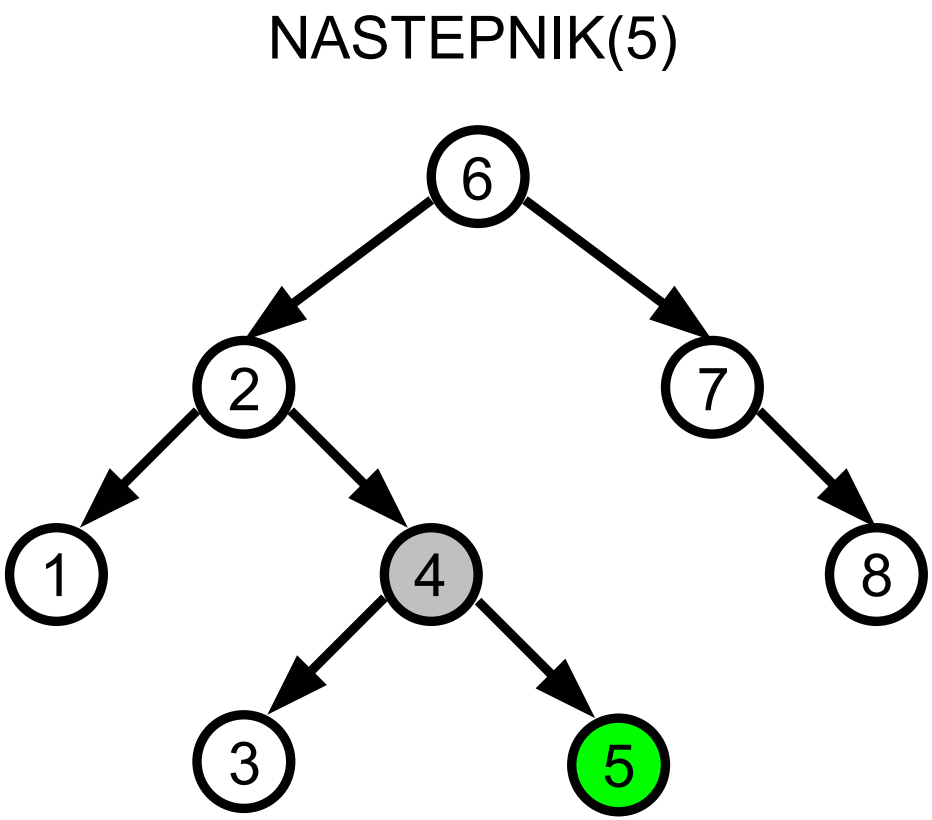
Następnik



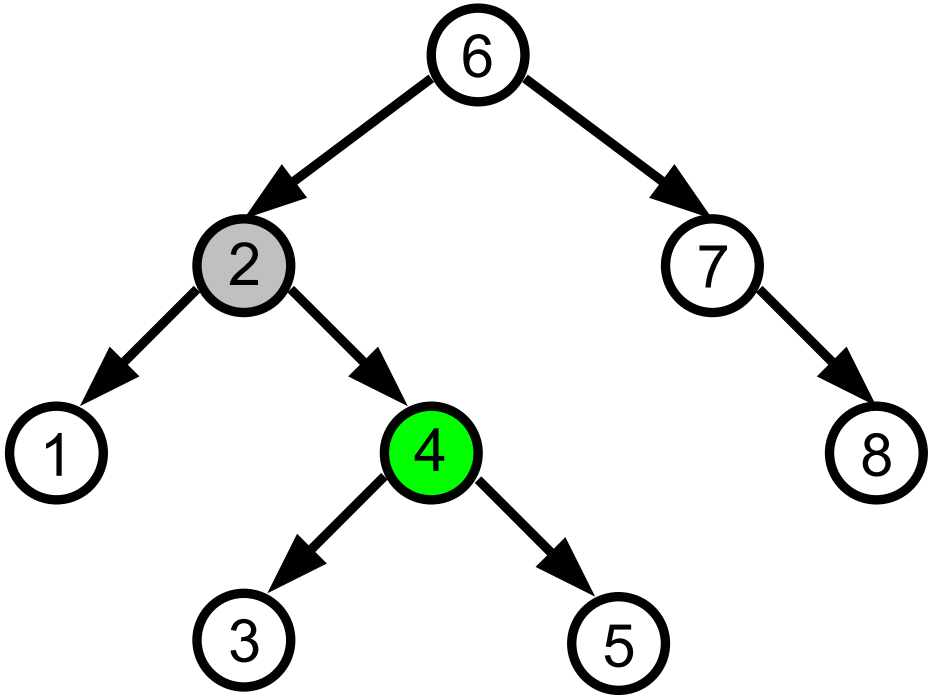
Następnik



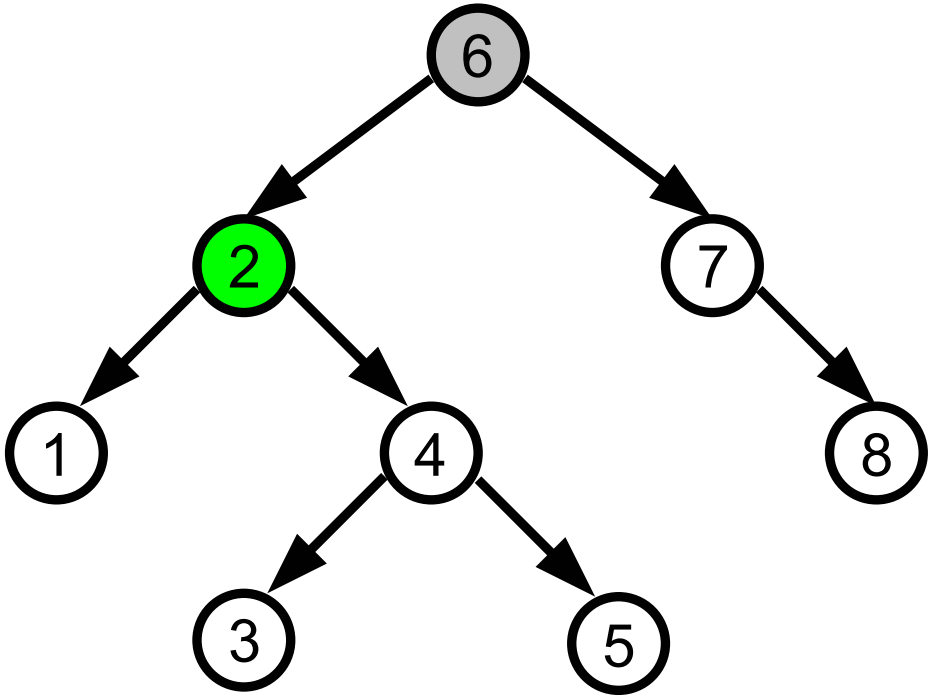
Następnik



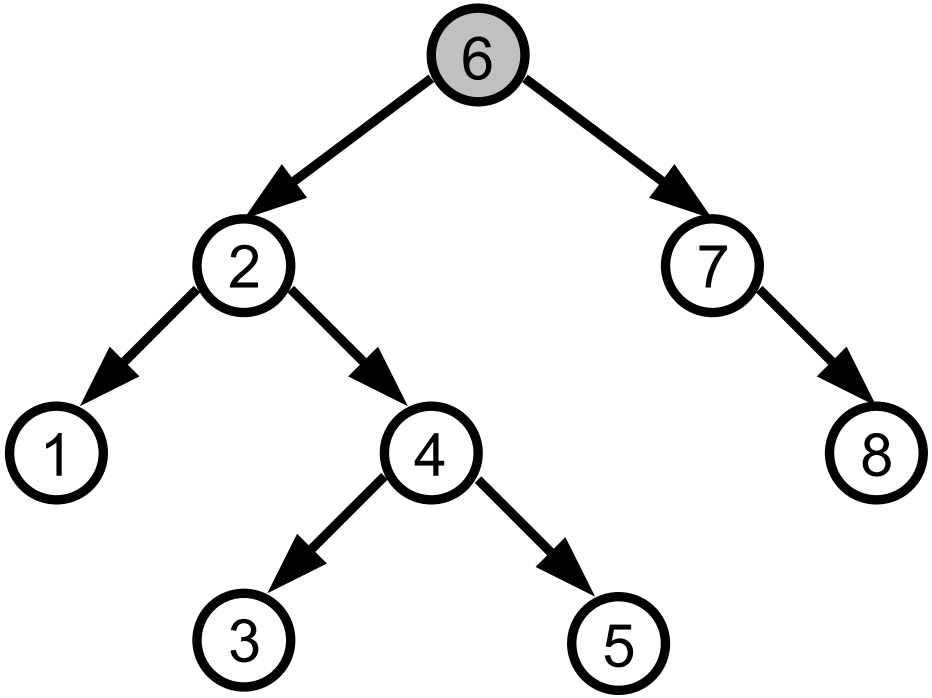
Następnik



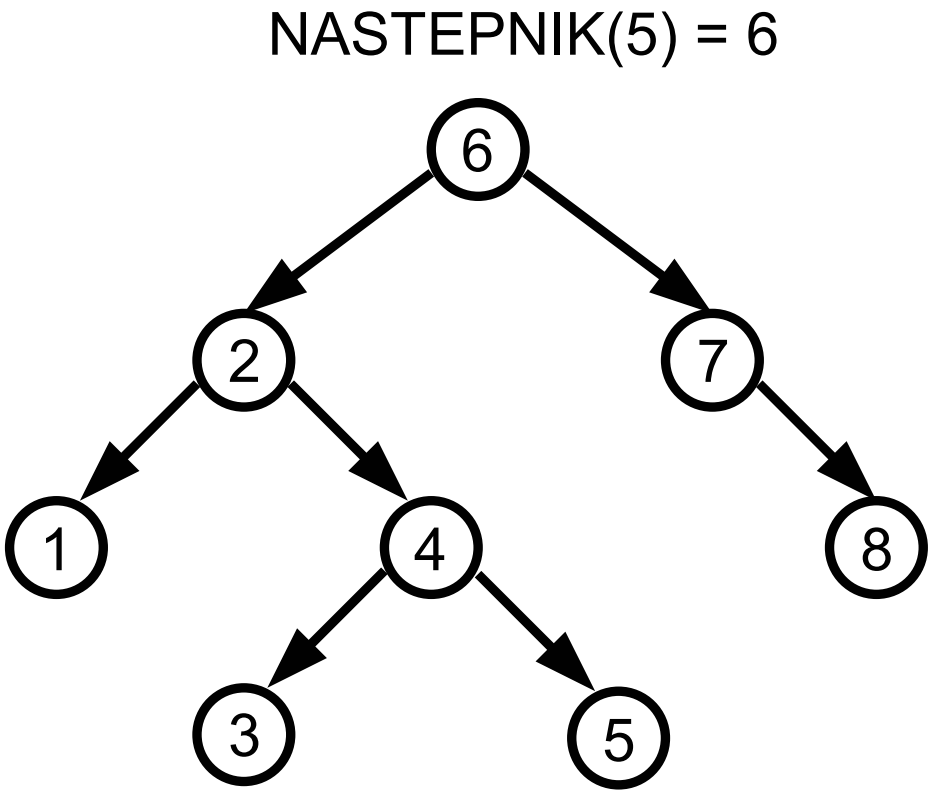
Następnik



Następnik



Następnik



Wstawianie węzła

- wstawiany węzeł będzie zawsze liściem
- postępujemy tak jak podczas wyszukiwania
- w miejscu gdy dojdziemy do *nullptr* wstawiamy nasz element
- zakładamy, że węzeł *nowy* ma już wypełnione pola *key* i *value*

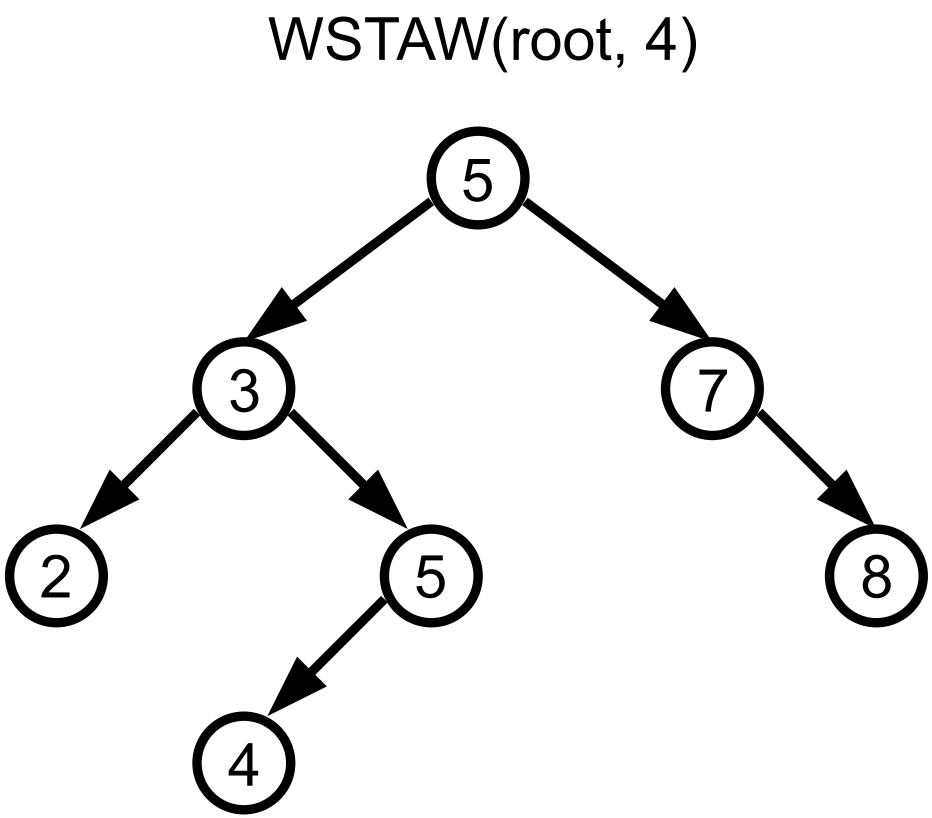
Wstawianie węzła

```
WSTAW(inout root, nowy)
 1: parent = nullptr
 2: curr = root
 3: while curr ≠ nullptr do
 4:         parent = curr
 5:         if nowy.key ≤ curr.key then curr = curr.left
 6:         else curr = curr.right
 7: end while
 8: nowy.parent = parent
 9: if parent = nullptr then
10:         root = parent
11: else if nowy.key ≤ parent.key then
12:         parent.left = nowy
13: else
14:         parent.right = nowy
15: end if
```

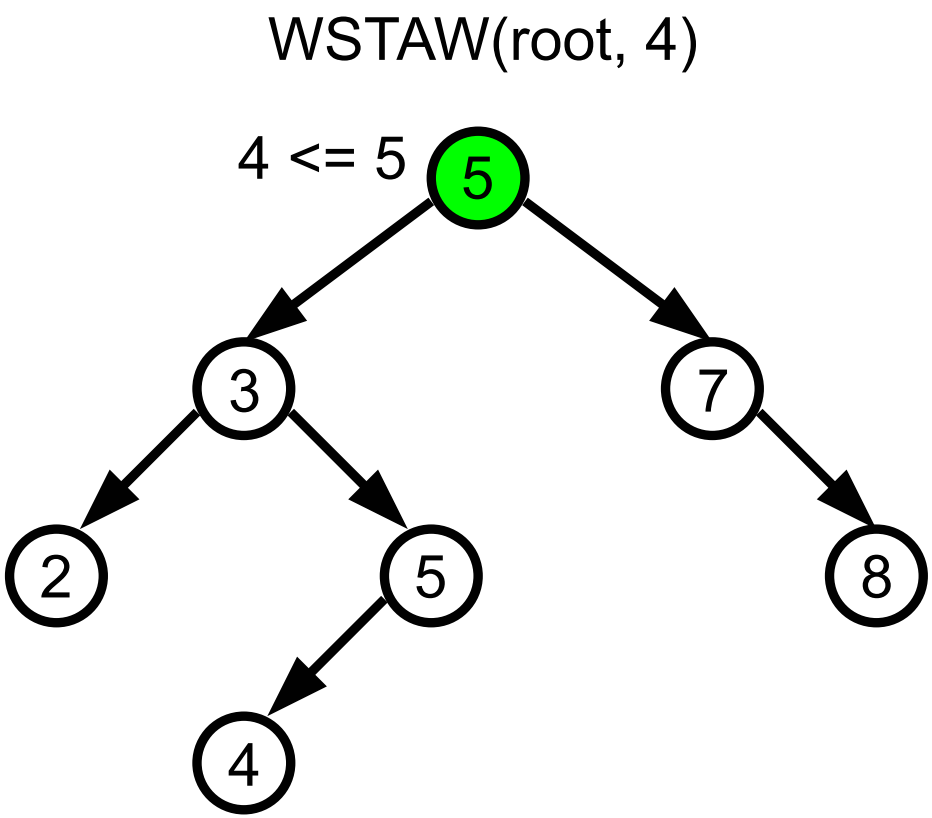
Wstawianie węzła (rekurencyjnie)

```
WSTAW(root, nowy)
1: if root == nullptr then return nowy
2: if nowy.key ≤ root.key then
3:         parent.left = WSTAW(root.left, nowy)
4:         root.left.parent = root
5: else
6:         parent.right = WSTAW(root.right, nowy)
7:         root.right.parent = root
8: end if
9: return root
```

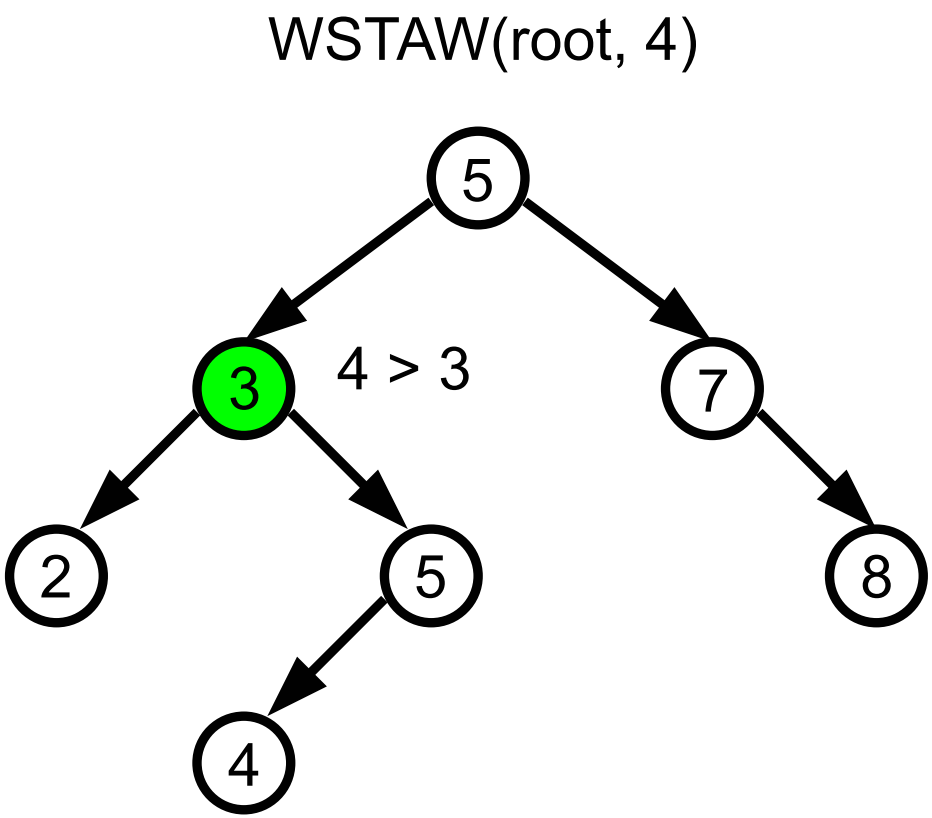
Wstawianie



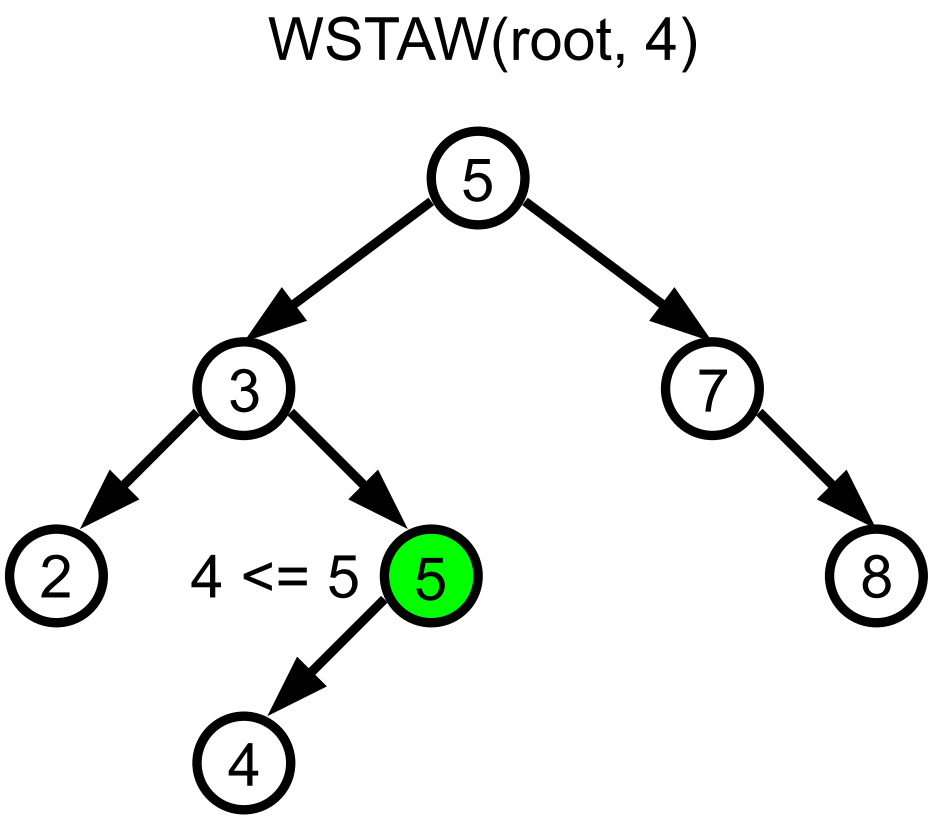
Wstawianie



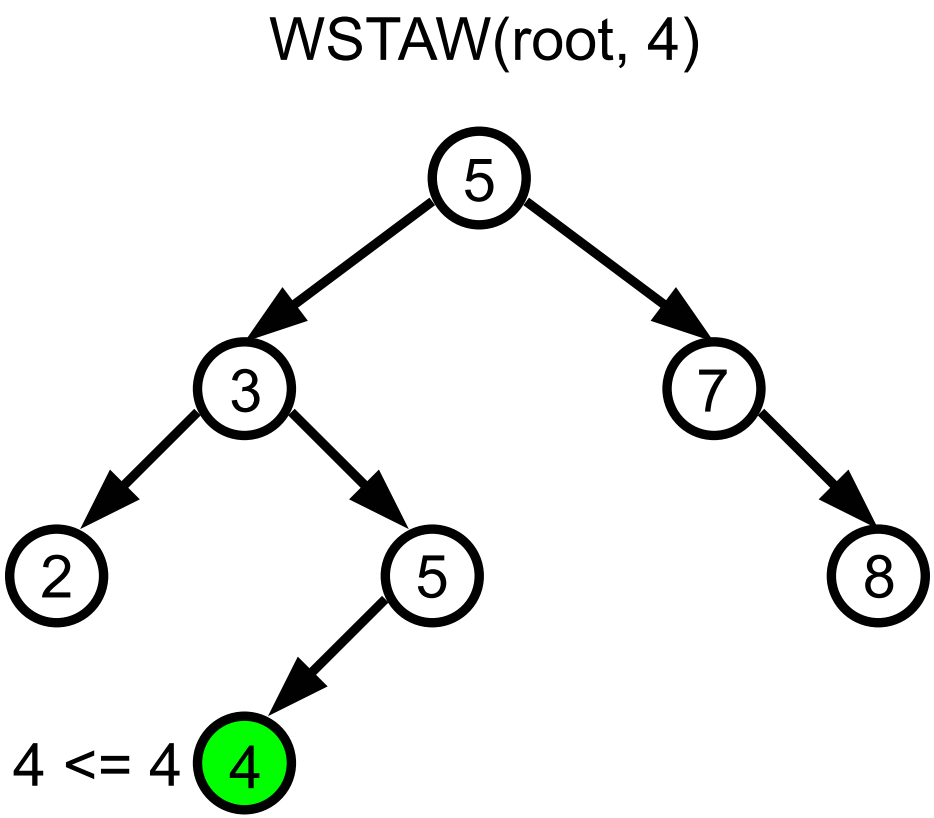
Wstawianie



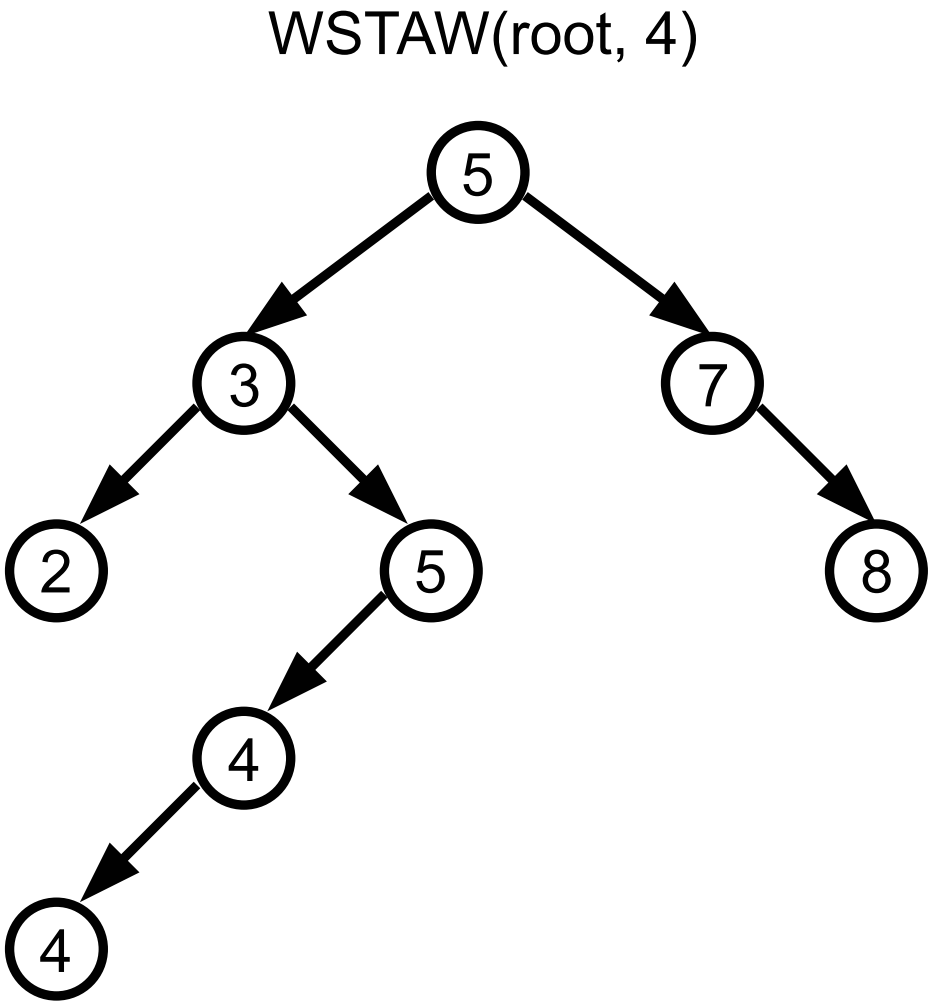
Wstawianie



Wstawianie



Wstawianie



Usuwanie węzła

- jeżeli usuwany węzeł jest liściem, możemy go łatwo usunąć
- jeżeli usuwany węzeł ma jednego potomka, możemy "skrócić" gałąź, której jest elementem
- usunięcie węzła z dwoma potomkami wymaga więcej pracy, należy:
  - znaleźć jego następnika (ten na pewno będzie miał co najwyżej jednego potomka)
  - przepisać zawartość następnika w miejsce usuwanego węzła
  - usunąć następnika

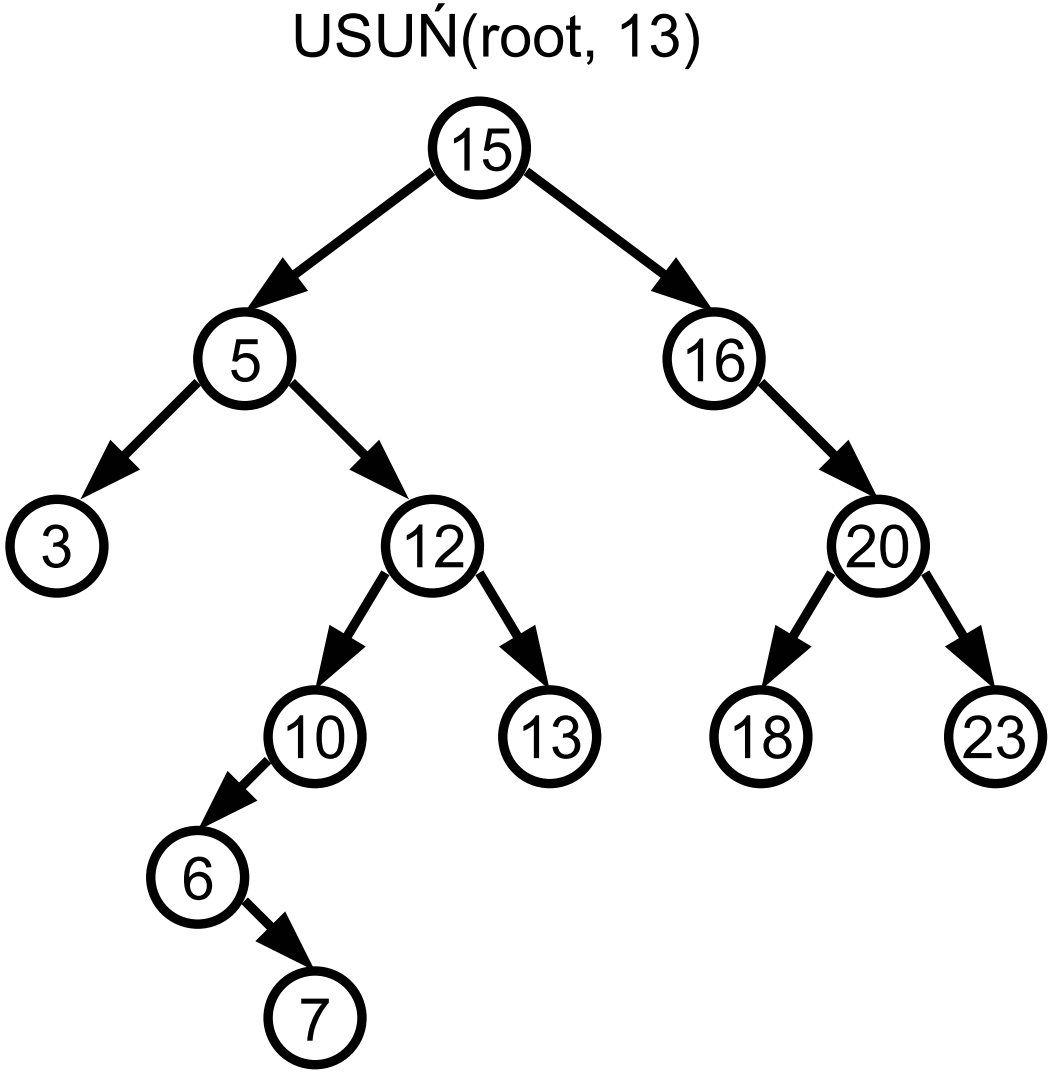
Usuwanie węzła

```
USUN(inout root, wezel)
  if wezel.left = nullptr or wezel.right = nullptr then
        usuwany = wezel
  else
        usuwany = NASTEPNIK(wezel)
  end if
  if usuwany.left ≠ nullptr then
        potomek = usuwany.left
  else
        potomek = usuwany.right
  end if
  if potomek ≠ nullptr then
        potomek.parent = usuwany.parent
  end if
```

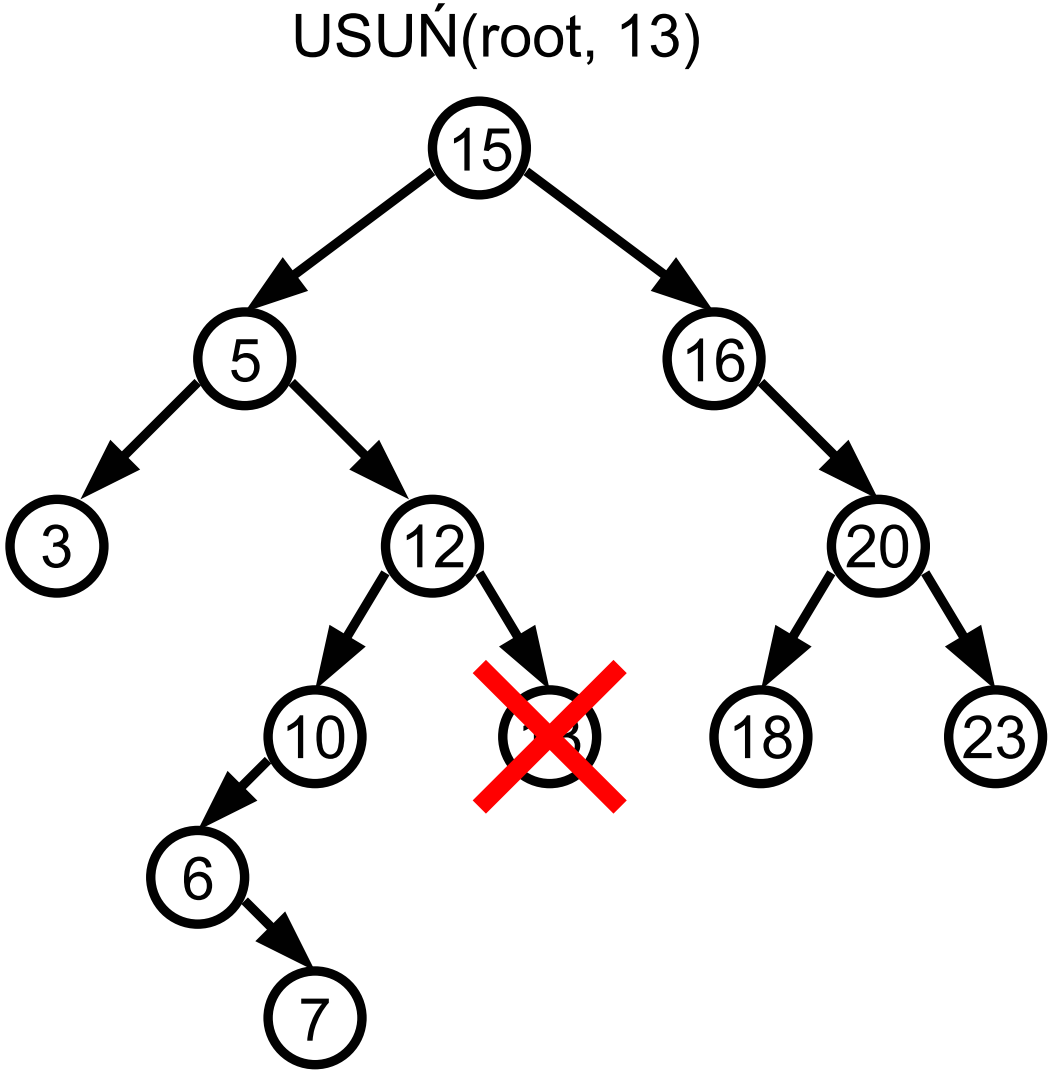
Usuwanie węzła

if *usuwany*.*parent* = *nullptr* **then**
    root = *potomek*
else if *usuwany* = *usuwany*.*parent*.*left* **then**
    usuwany.*parent*.*left* = *potomek*
else
    usuwany.*parent*.*right* = *potomek*
end if
if *usuwany* ≠ *wezel* **then**
    wezel.*key* = *usuwany*.*key*
    wezel.*value* = *usuwany*.*value*
end if
zwolnij pamięć zajmowaną przez *usuwany*

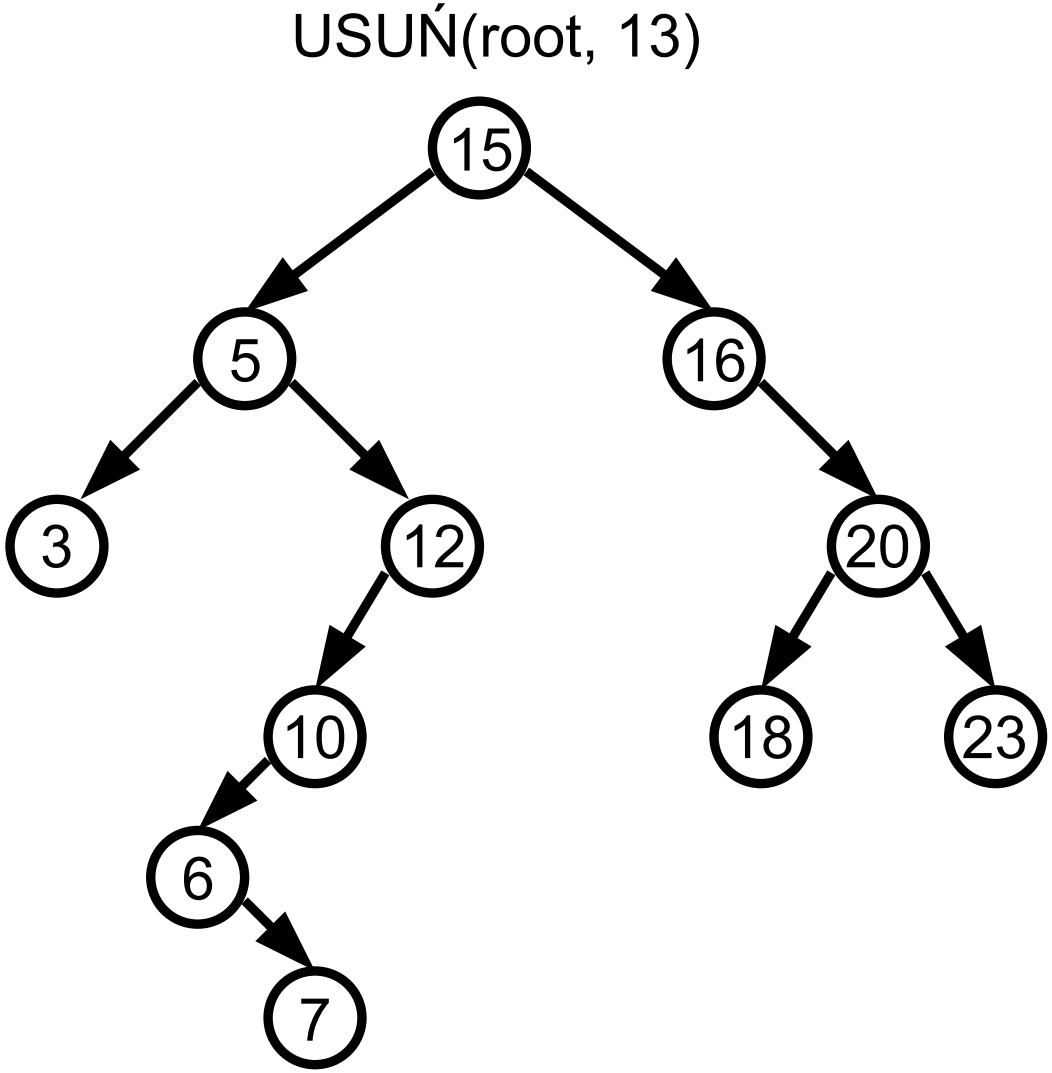
Usuwanie liścia



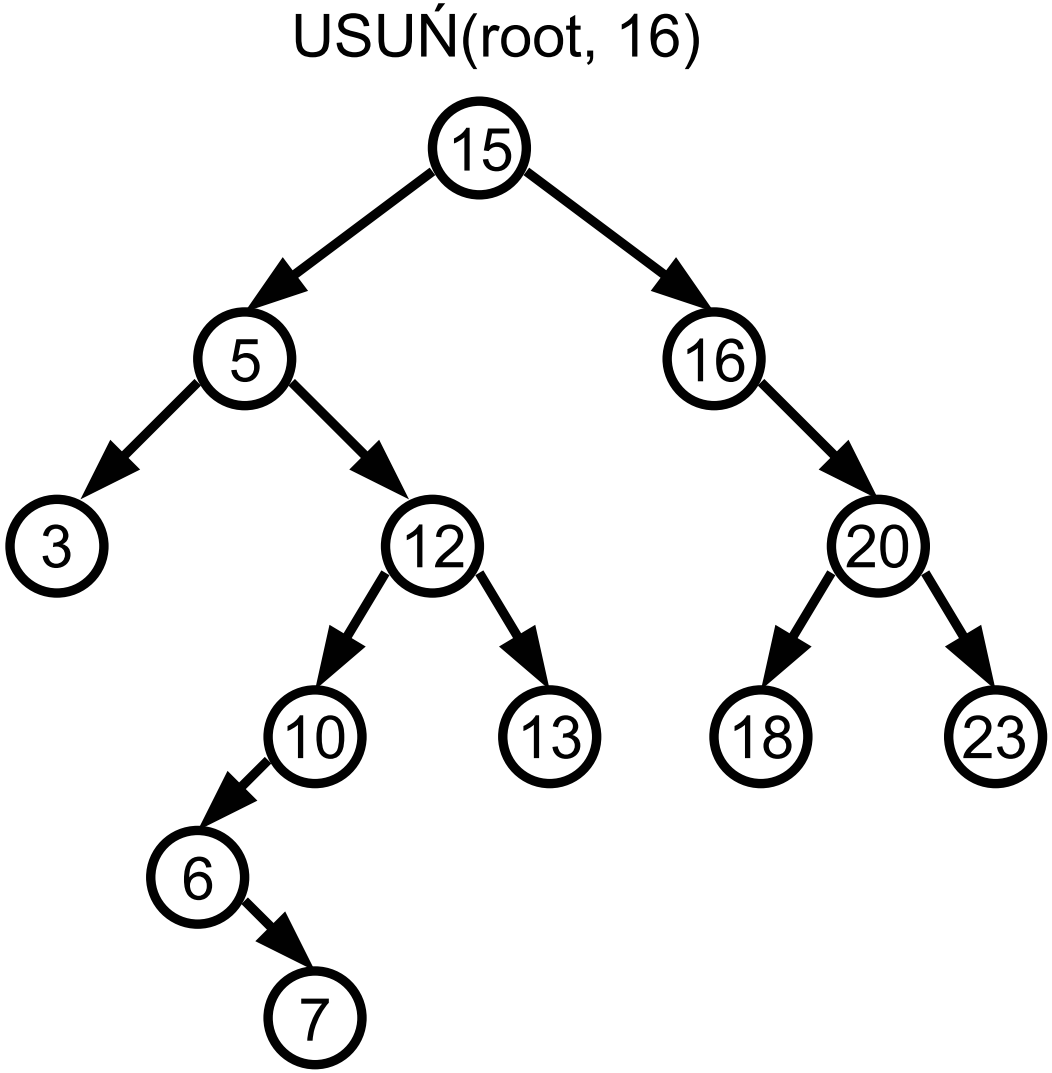
Usuwanie liścia



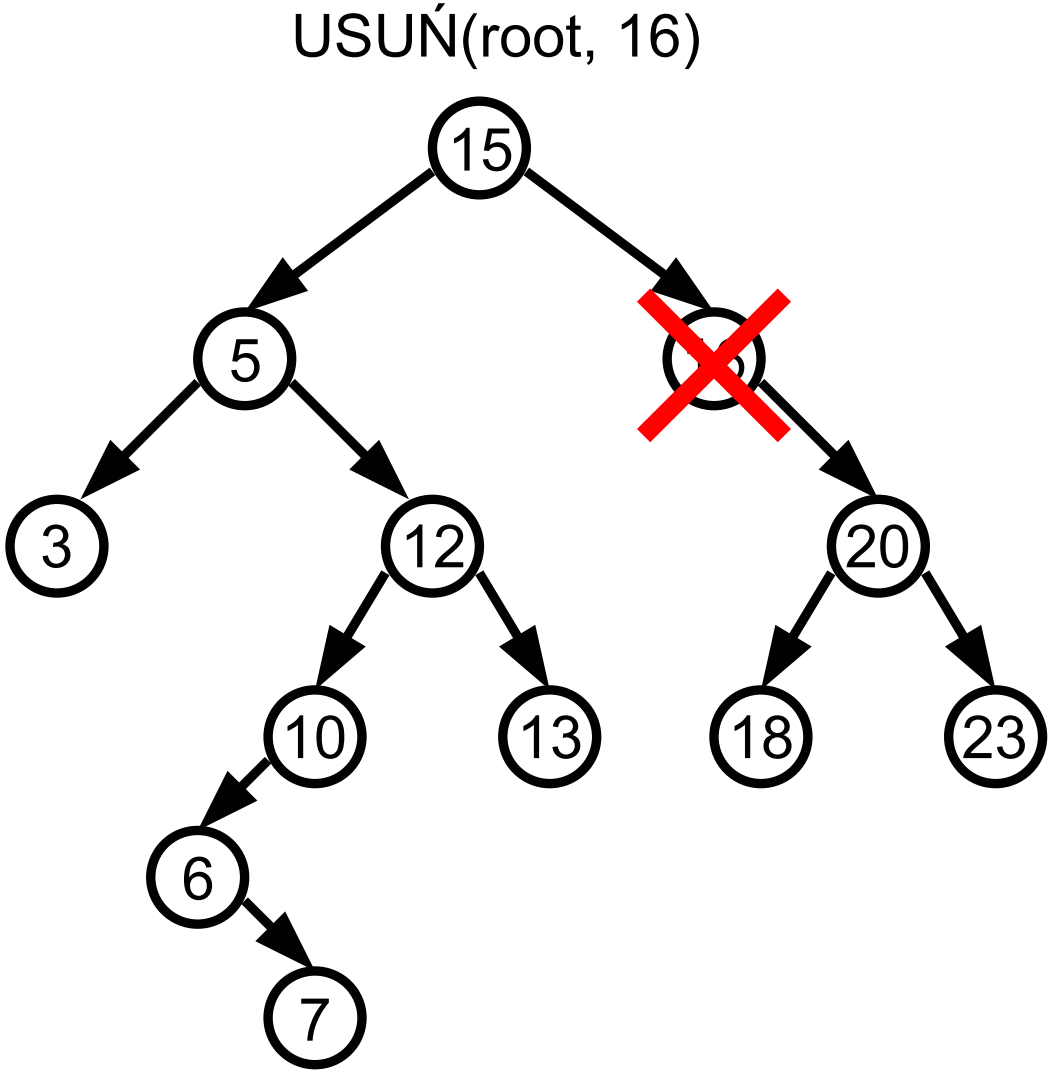
Usuwanie liścia



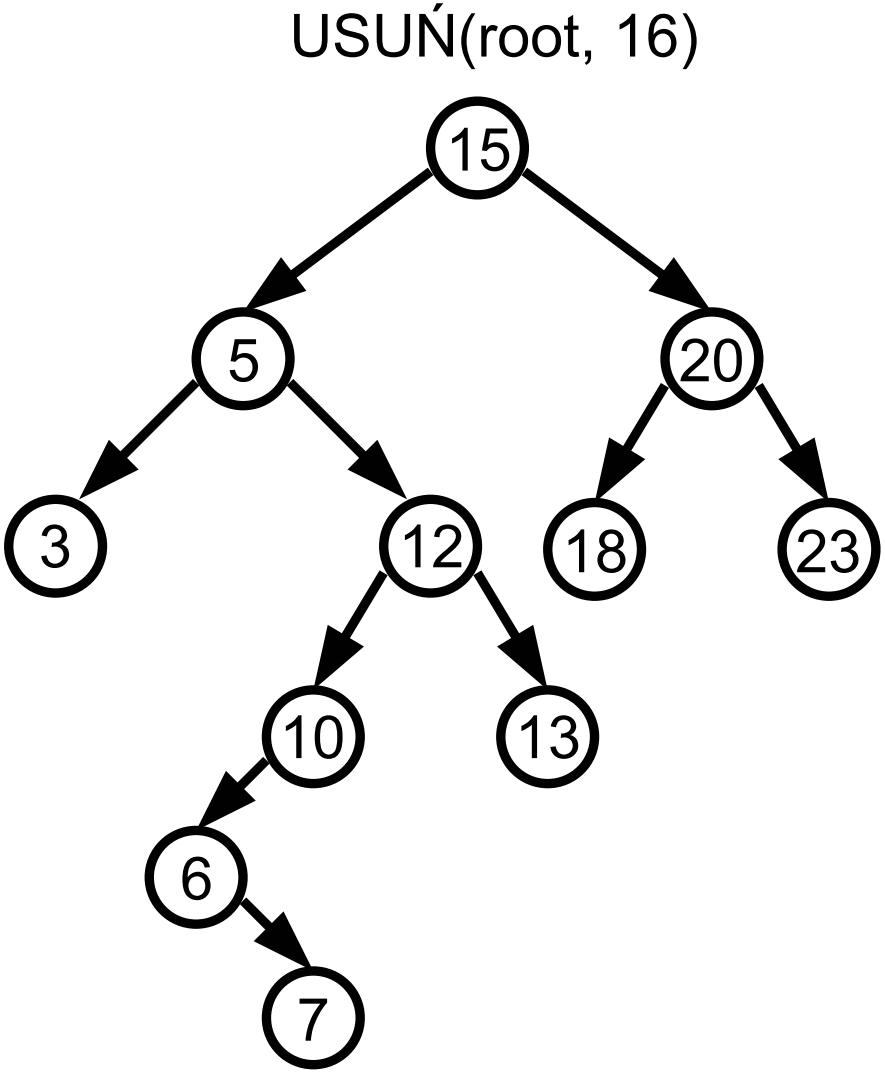
Usuwanie węzła z 1 potomkiem



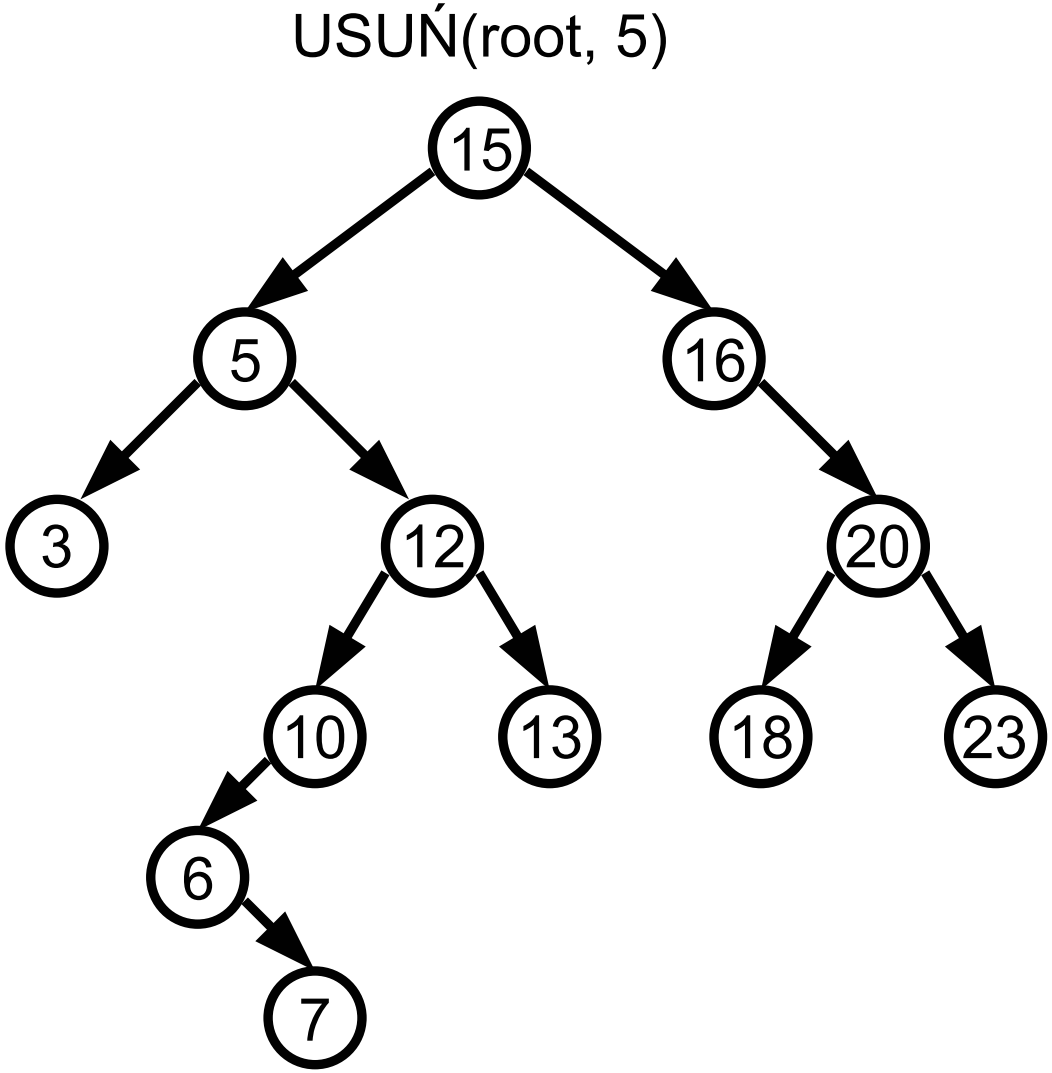
Usuwanie węzła z 1 potomkiem



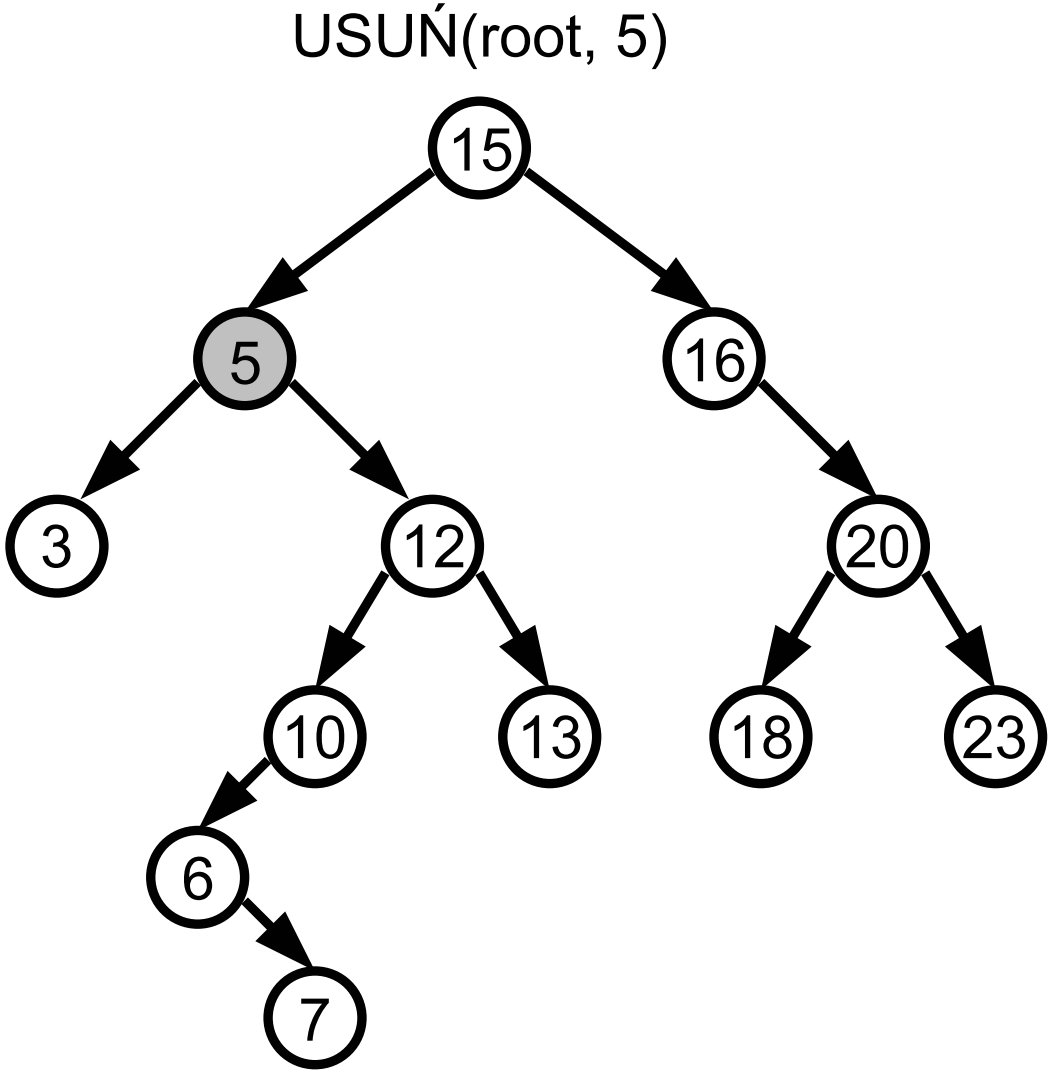
Usuwanie węzła z 1 potomkiem



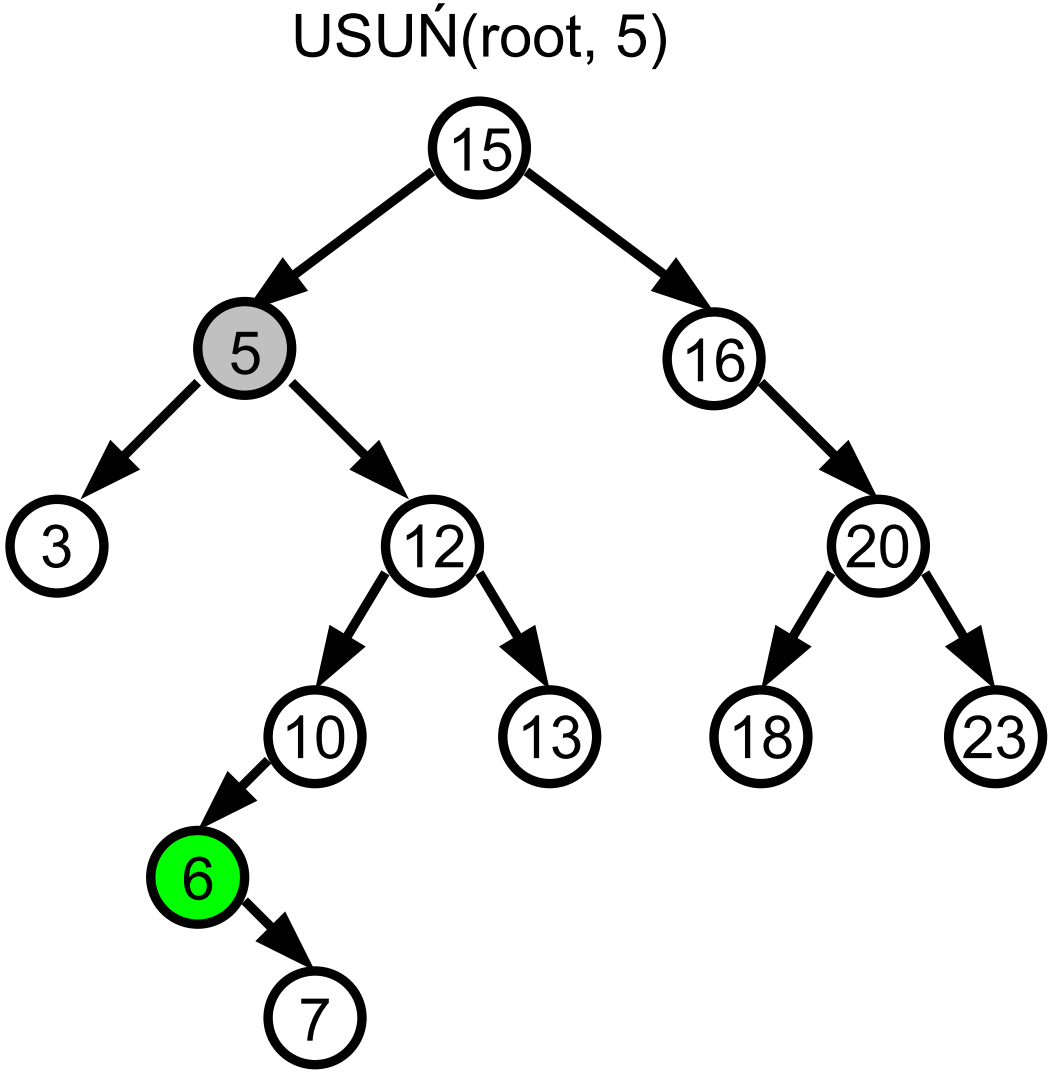
Usuwanie węzła z 2 potomkami



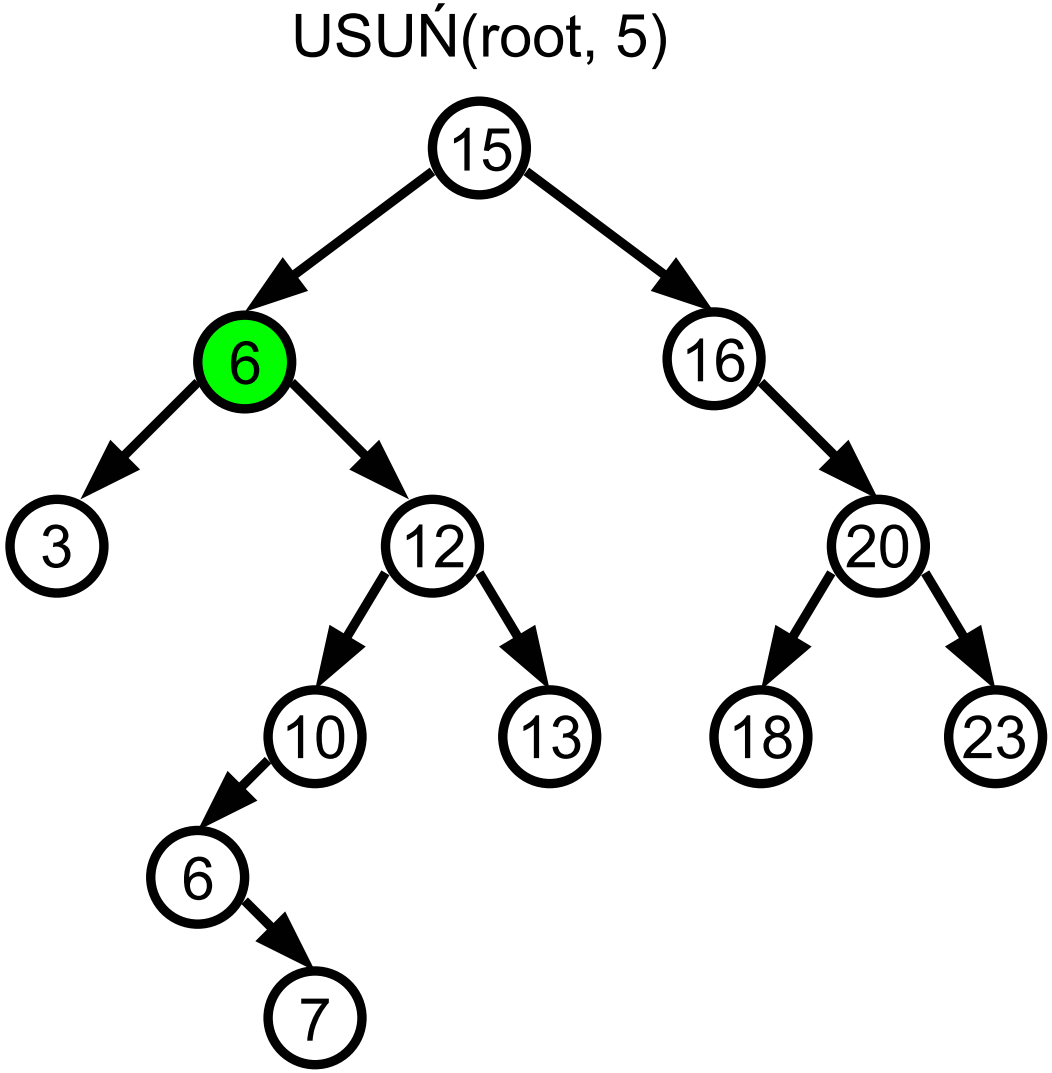
Usuwanie węzła z 2 potomkami



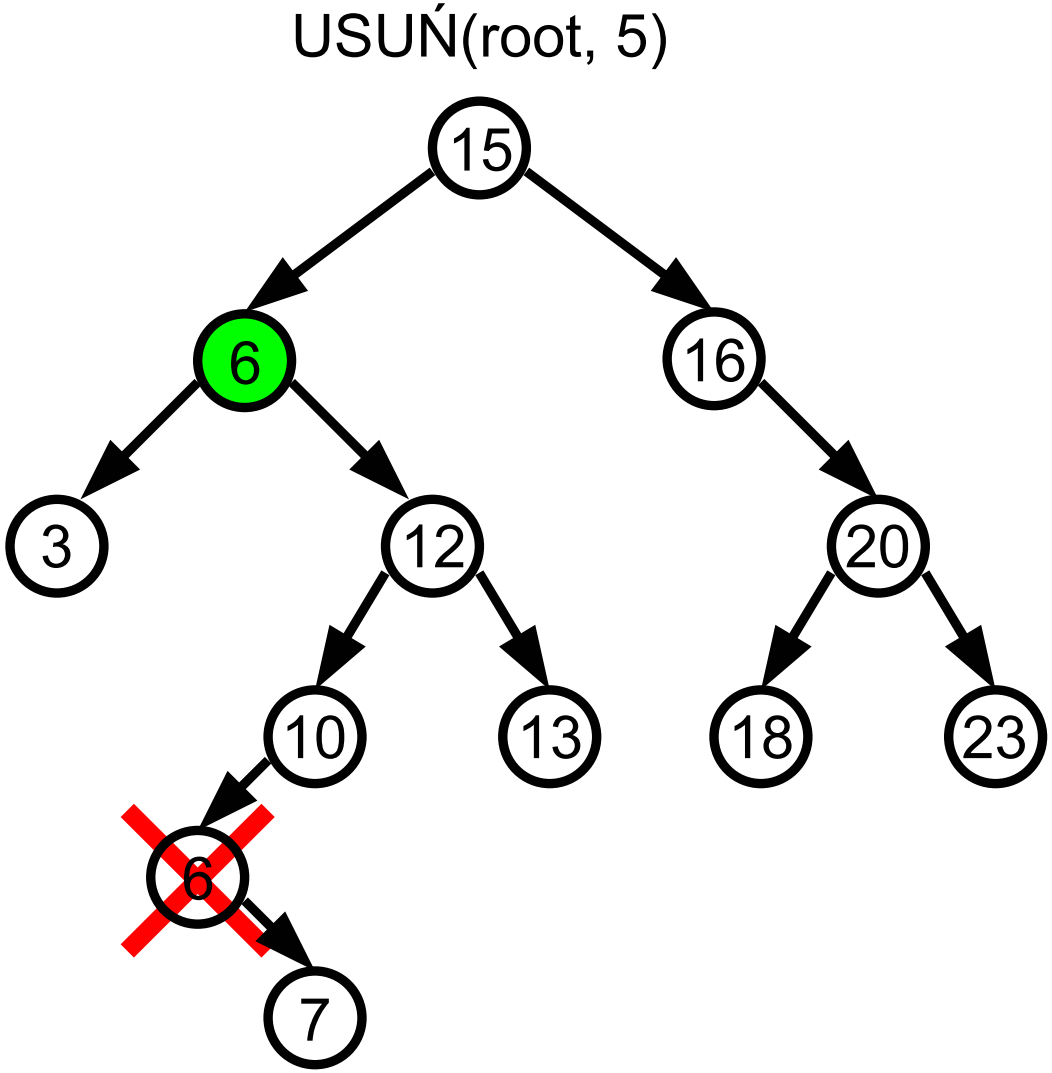
Usuwanie węzła z 2 potomkami



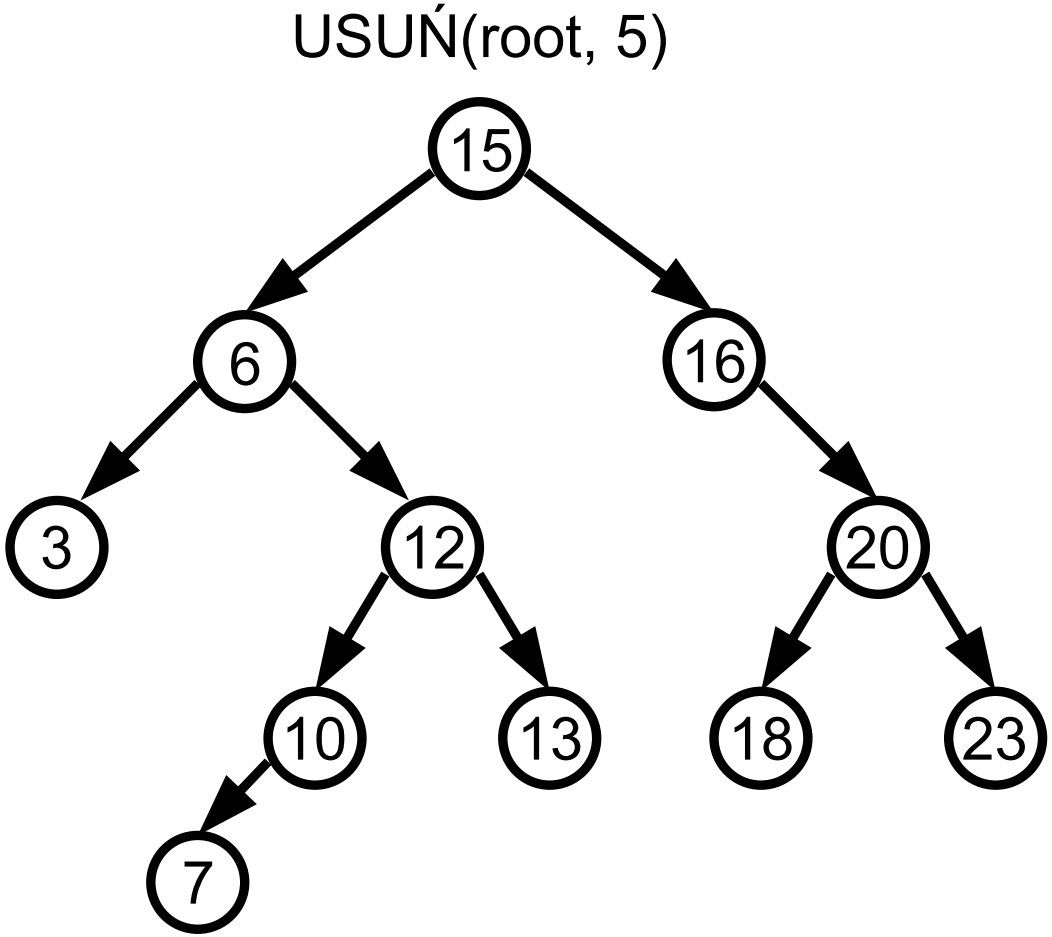
Usuwanie węzła z 2 potomkami



Usuwanie węzła z 2 potomkami



Usuwanie węzła z 2 potomkami



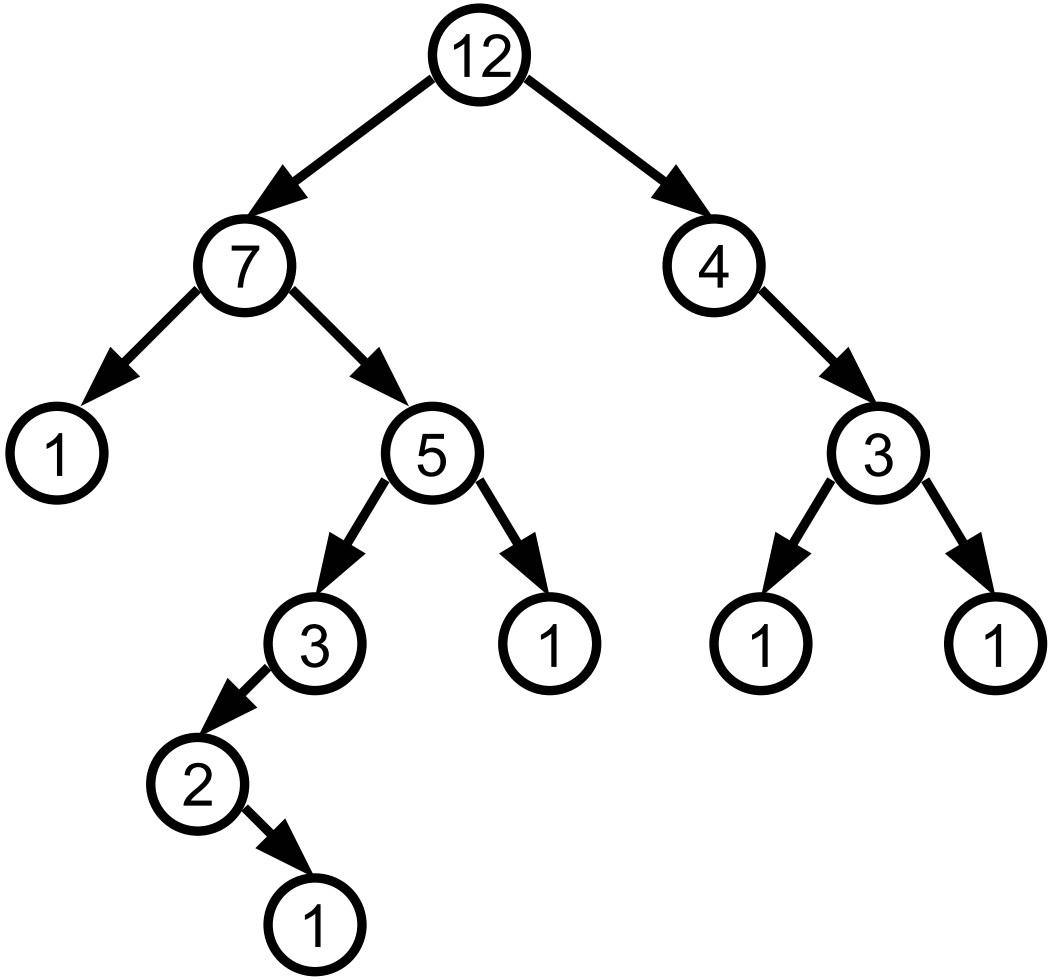
Drzewa poszukiwań binarnych

- pesymistyczna złożoność przedstawionych operacji wynosi $O(n)$
- np. ciąg wstawień do drzewa kluczy: $1, 2, 3, \ldots$
- istnieją rozszerzenia drzew poszukiwań binarnych gwarantujące logarytmiczną wysokość drzewa (więcej na następnych wykładach)

Drzewa statystyk pozycyjnych

- możemy rozbudować drzewo binarne o możliwość znajdowania i-tej statystyki pozycyjnej w czasie $O(\log n)$ (pesymistycznie $O(n)$)
- dodajemy do węzła informację o rozmiarze drzewa, którego jest korzeniem
- wtedy $root.size = root.left.size + root.right.size + 1$
- należy pamiętać, aby pole to aktualizować podczas każdego wstawiania i usuwania elementu (wymaga to czasu proporcjonalnego do wysokości drzewa)

Drzewa statystyk pozycyjnych



i-ta statystyka pozycyjna

```
STATYSTYKA(root, i)
1: if i = root.left.size + 1 then
2:         return root.key
3: else if i ≤ root.left.size then
4:         return STATYSTYKA(root.left, i)
5: else
6:         return STATYSTYKA(root.right, i − root.left.size − 1)
7: end if
```

Drzewa słownikowe (TRIE)

- przeszukując drzewo binarne, w każdym kroku z porównania kluczy wykorzystujemy tylko jeden bit informacji (przejście w lewo/ przejście w prawo)
- w drzewach słownikowych wykorzystujemy tej informacji więcej
- drzewa są bardziej rozgałęzione (już nie tylko binarne)
- asymptotyczna złożoność operacji pozostaje taka sama ($O(h)$), ale zmniejsza się wysokość — tę samą ilość węzłów możemy umieścić w drzewie o mniejszej wysokości

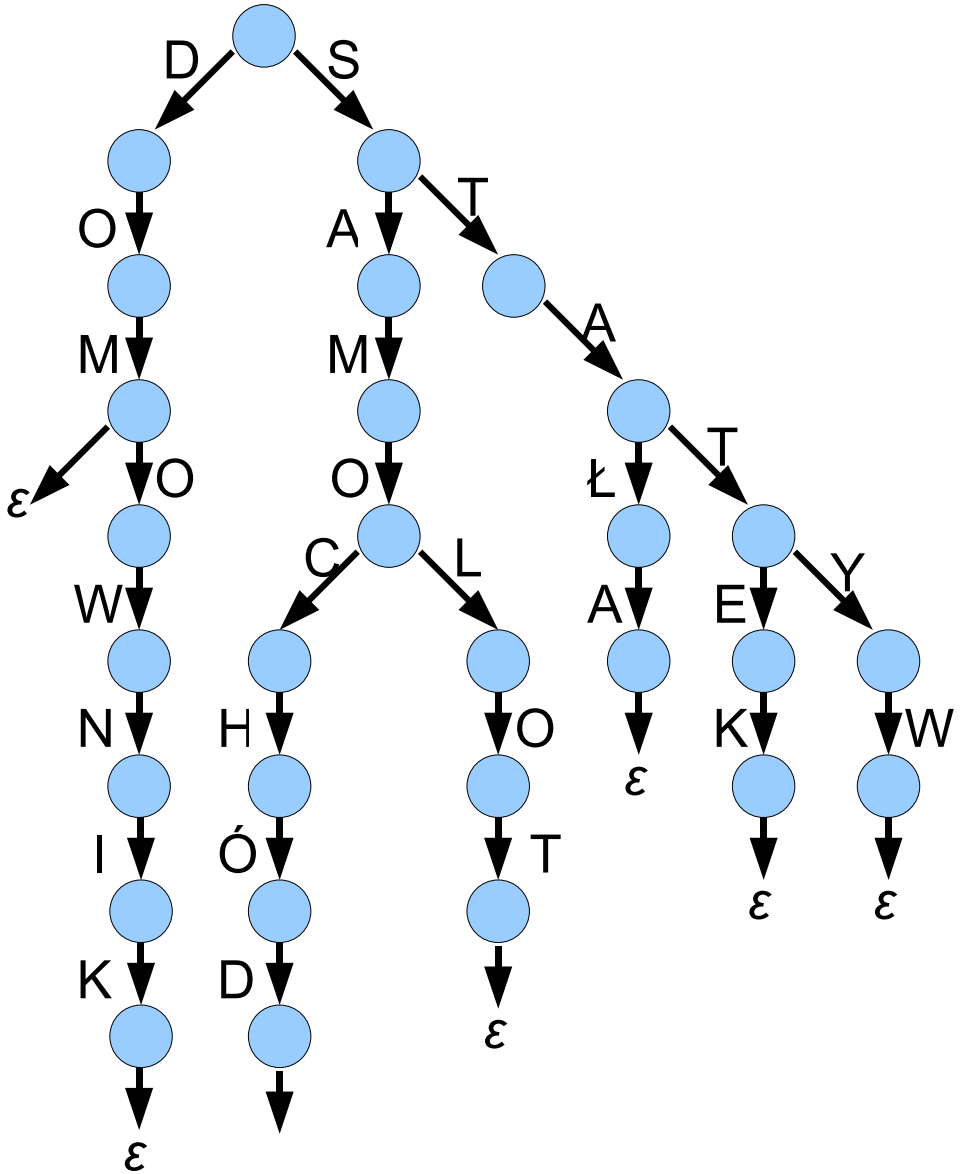
Drzewa słownikowe (TRIE)

- klucz dzielimy na litery (gdy przechowujemy liczby, literą może być pojedyncza cyfra lub grupa kilku cyfr, w przypadku adresu IP literą może być jeden oktet)
- każdy węzeł drzewa posiada maksymalnie tyle potomków, ile wynosi rozmiar alfabetu, krawędzie prowadzące do potomków są poetykietowane kolejnymi literami
- każdy węzeł posiada dodatkowo znacznik określający, czy słowo kończące się w tym węźle należy do słownika (oraz ewentualnie dodatkowe dane, powiązane z kluczem)

Drzewa słownikowe (TRIE)

- przechodząc po drzewie, w każdym kroku wybieramy potomka wskazywanego przez krawędź z etykietą równą bieżącej literze
- usuwając węzeł, należy sprawdzić, czy jest to ostatni potomek rodzica — jeżeli tak, należy usunąć także rodzica, itd.

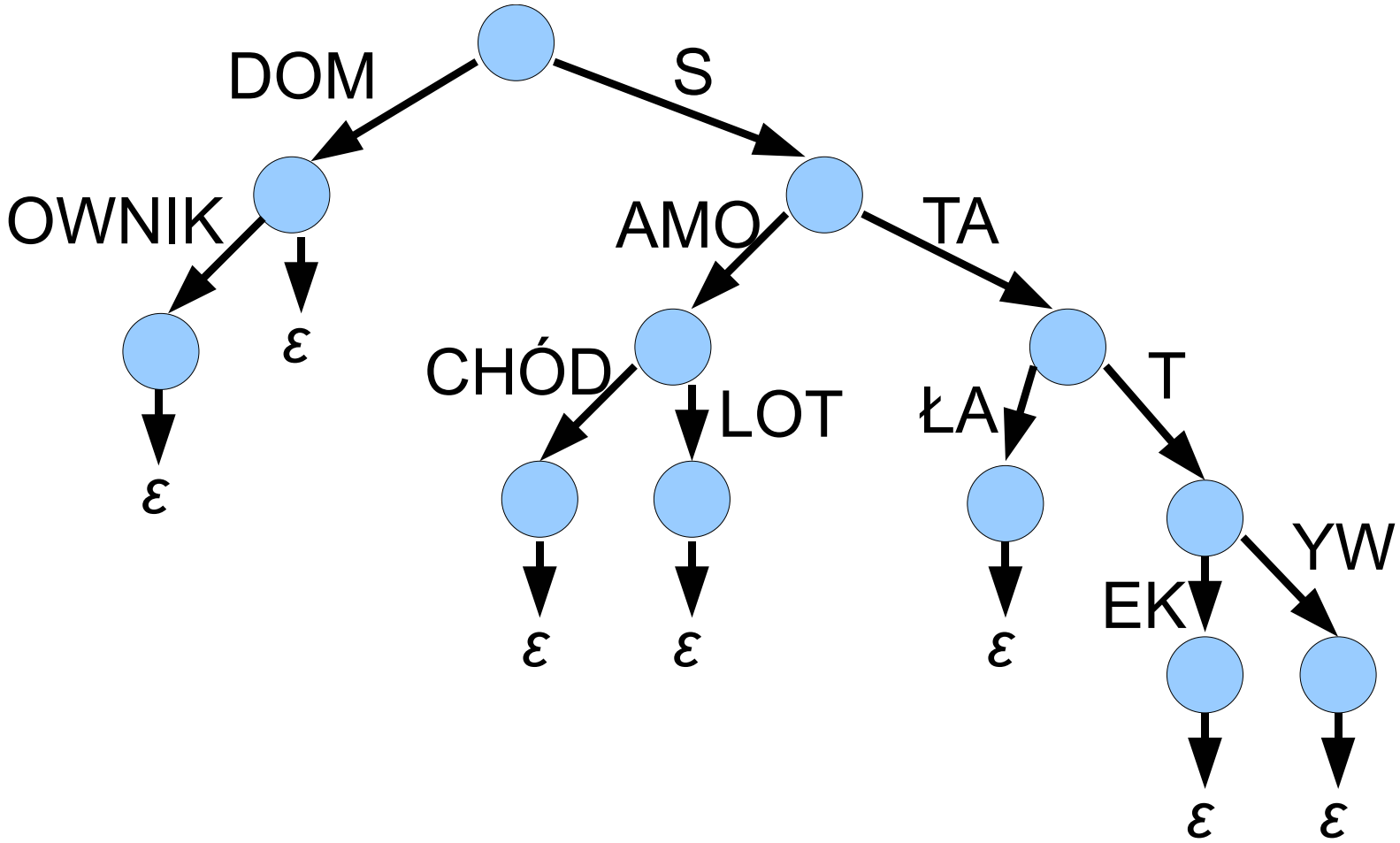
Drzewa słownikowe (TRIE)



Kompresja ścieżki

- w przypadku rzadkiego drzewa struktura będzie alokowała sporo nieużywanej pamięci
- kompresja ścieżek — ścieżki "zwijamy" w jedną krawędź
- etykietami są teraz napisy (zamiast pojedynczych liter)
- komplikuje to jednak operacje na drzewie (np. podczas dodawania czasem trzeba dodać węzeł wewnątrz krawędzi)

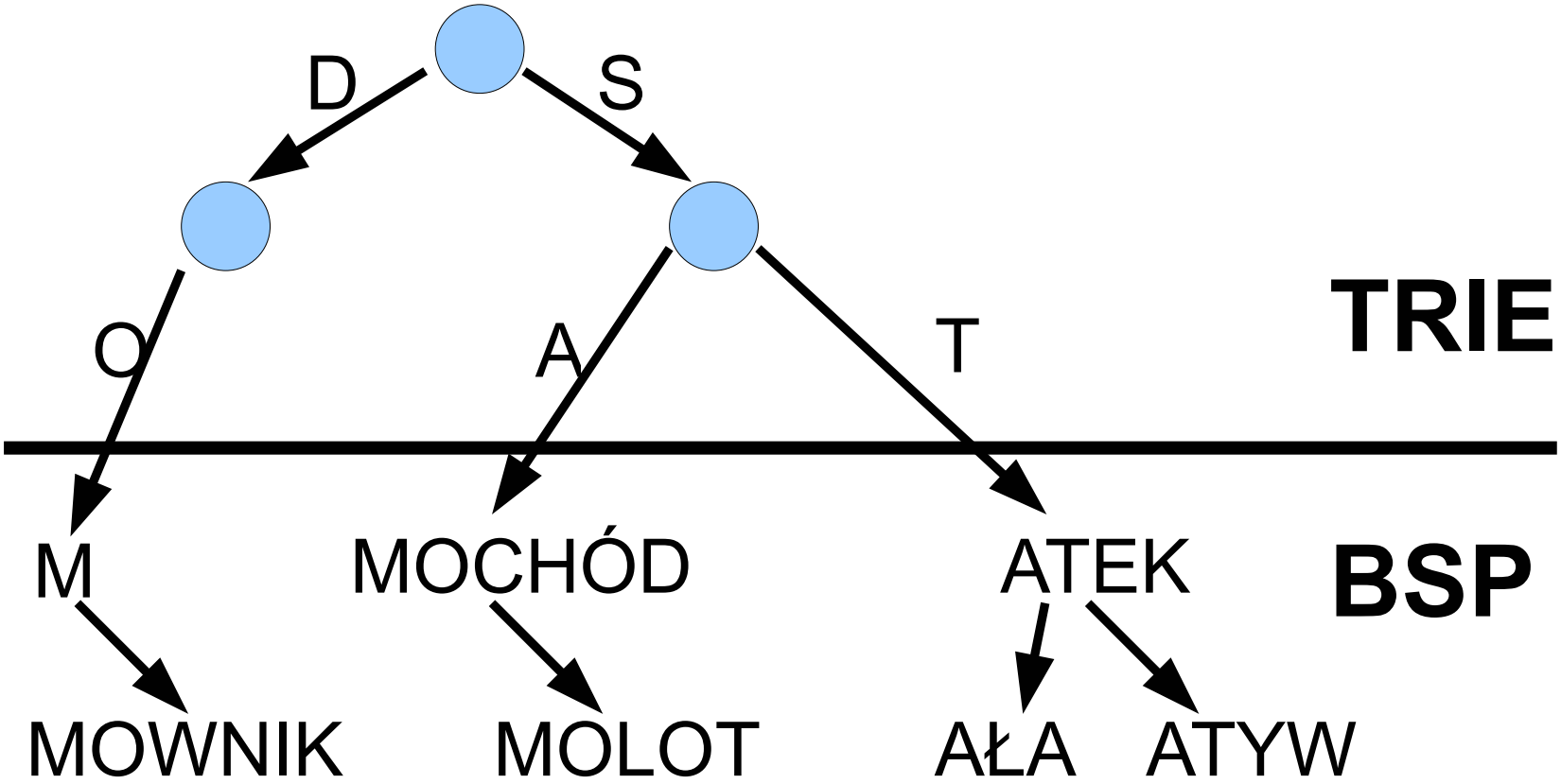
Kompresja ścieżki



Drzewa heterogeniczne

- większość wyrazów różni się na kilku początkowych literach — w słowniku istnieje dużo różnych kombinacji pierwszych kilku liter
- końcówki wyrazów są mniej zróżnicowane

Drzewa heterogeniczne



Drzewa heterogeniczne

- większość wyrazów różni się na kilku początkowych literach — w słowniku istnieje dużo różnych kombinacji pierwszych kilku liter
- końcówki wyrazów są mniej zróżnicowane

Radix tree

- zwykłe drzewo słownikowe, w którym kluczami są liczby całkowite o ustalonej liczbie bitów
- dzielimy liczby na litery po k bitów (np. 32 bitowe liczby na 4 litery po 8 bitów)
- ostatni poziom, zamiast wskaźników, przechowuje dane powiązane z kluczami

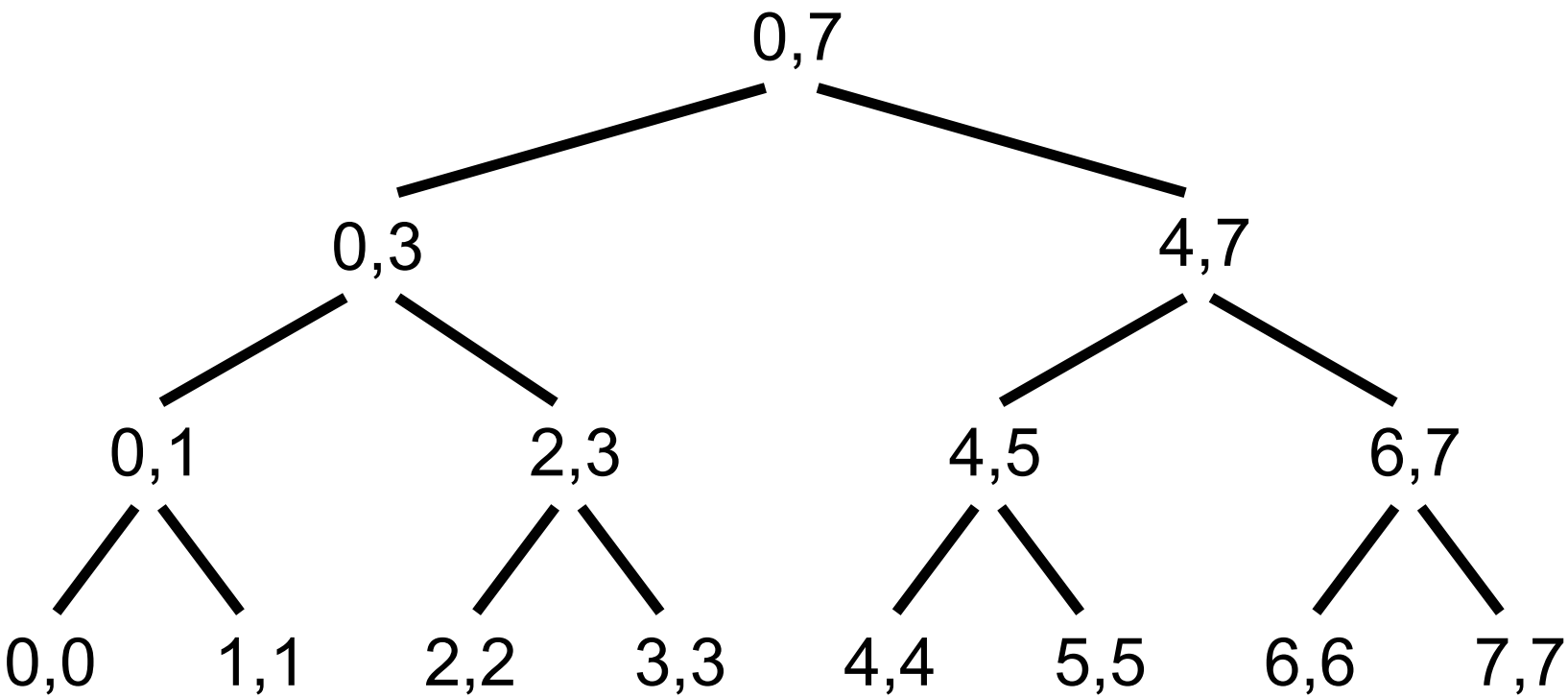
Drzewo przedziałowe

- mamy n danych: $a_0, \ldots, a_{n-1}$
- potrzebujemy pewnej "zagregowanej" informacji na temat dowolnego podprzedziału $[0, n-1]$, np.
  - minimum
  - maksimum
  - suma
  - średnia
  - itp.
- wiele zapytań o różne podprzedziały
- naiwne podejście: $O(nm)$ gdzie m to liczba zapytań

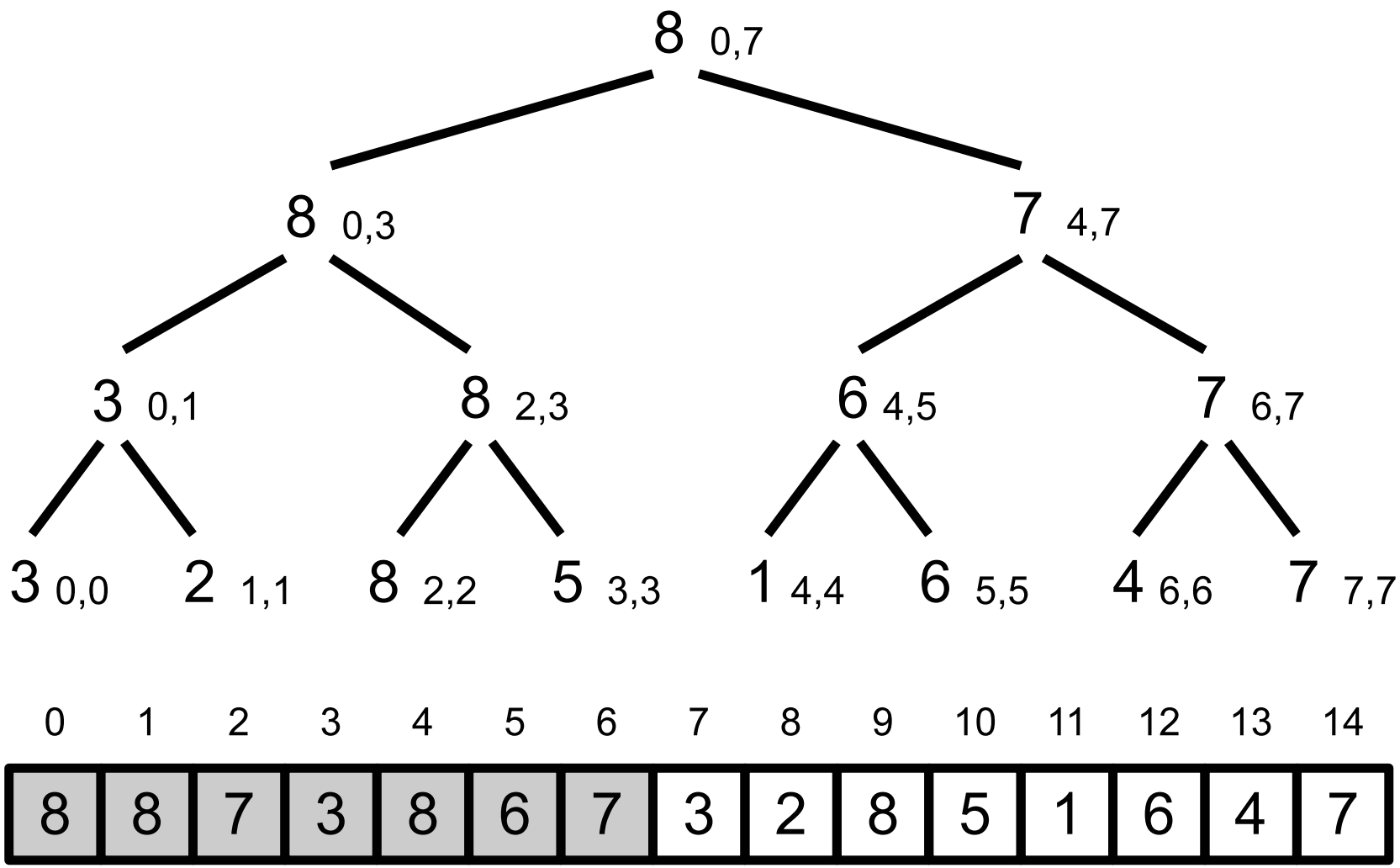
Drzewo przedziałowe

- drzewo przedziałowe =
  kopiec (struktura) + wzbogacone drzewo binarne
- każdy węzeł odpowiada pewnemu podprzedziałowi
- liście przechowują dane dla jednoelementowych podprzedziałów
- węzły wewnętrzne przechowują dane dla sumy podprzedziałów swoich dzieci
- jeżeli $n \neq 2^k$ uzupełniamy nieistotnymi elementami
  ($-\infty$ dla maksimum, $+\infty$ dla minimum, 0 dla sumy itp.)

Drzewo przedziałowe



Drzewo przedziałowe



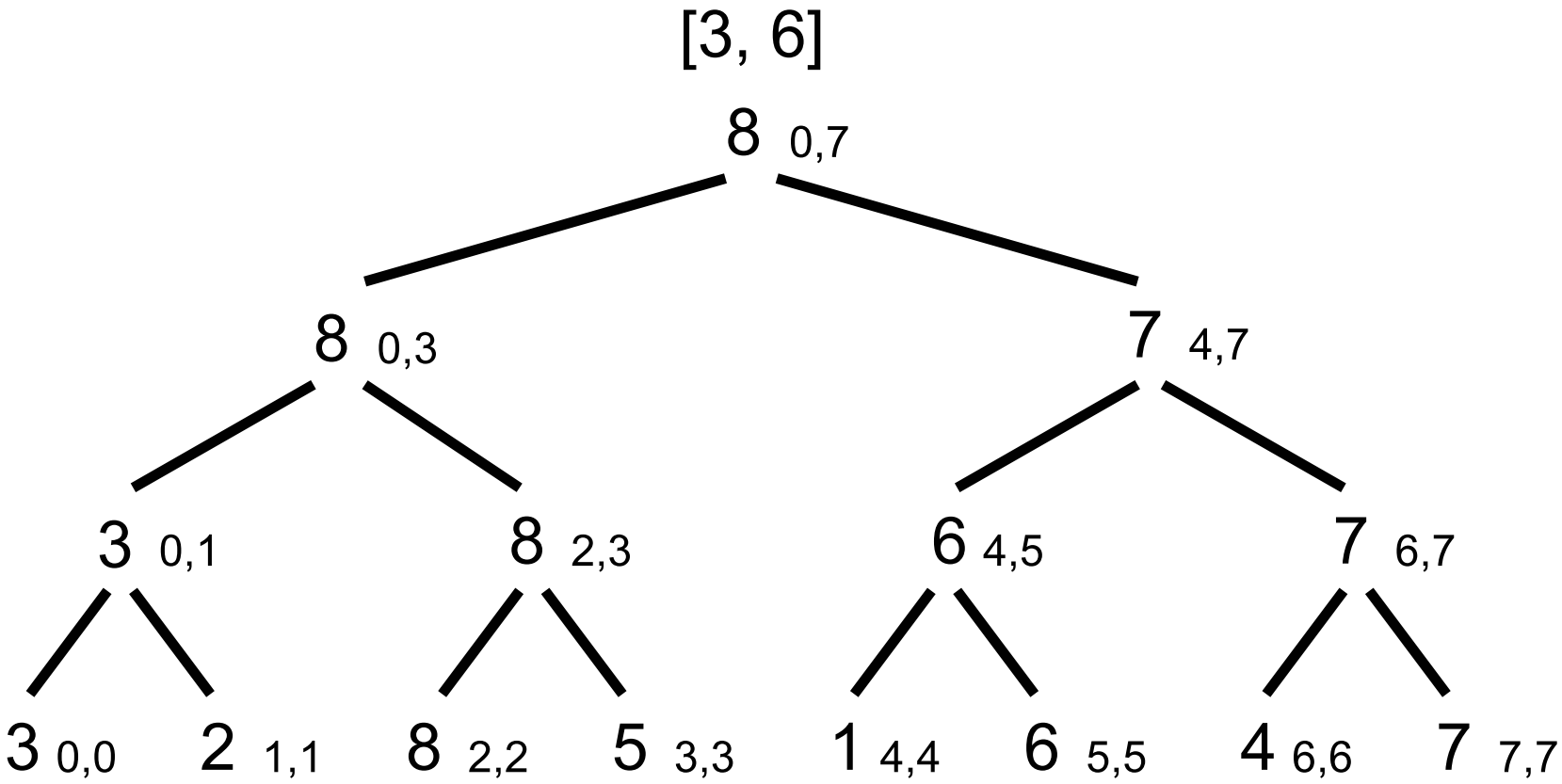
Drzewo przedziałowe

- budowa drzewa — podobnie jak budowa kopca
- zamiast własności kopca — odpowiednio minimum, maksimum czy suma dzieci
- umożliwia zmianę wartości elementów (jaka złożoność?)

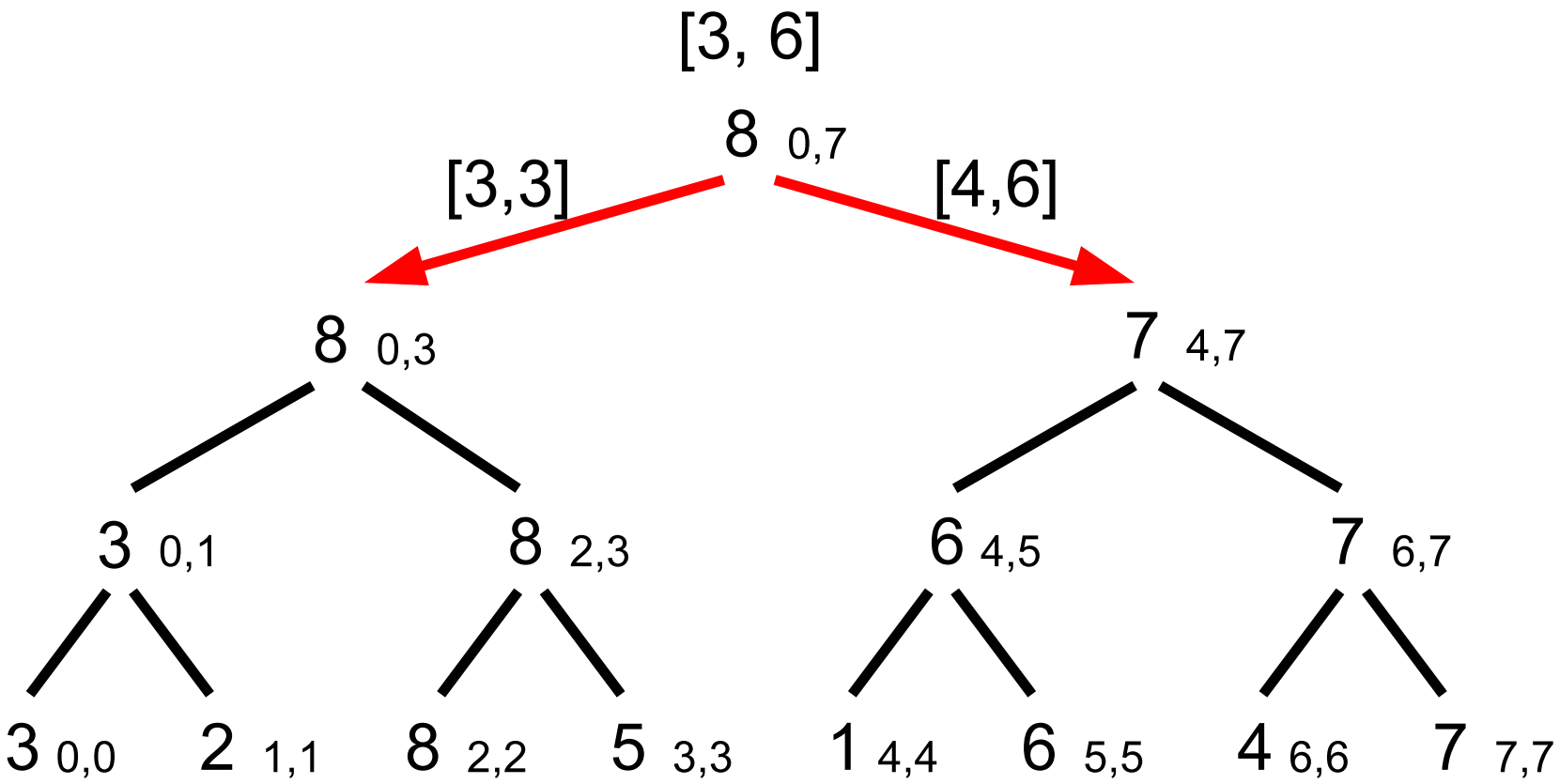
Drzewo przedziałowe — zapytania

- zadając pytanie o zagregowaną wartość na podprzedziale $[a, b]$
  - startujemy w korzeniu
  - jeżeli podprzedział jest identyczny z przedziałem węzła, zwracamy wartość z węzła
  - jeżeli podprzedział zawiera się wewnątrz jednego z dzieci przechodzimy do tego dziecka
  - jeżeli podprzedział ma część wspólną z lewym i prawym dzieckiem rekurencyjnie schodzimy do obu dzieci, odpowiednio ograniczając podprzedział
- złożoność $O(\log n)$
- ile razy "rozgałęzimy" wyszukiwanie (ostatni punkt)?

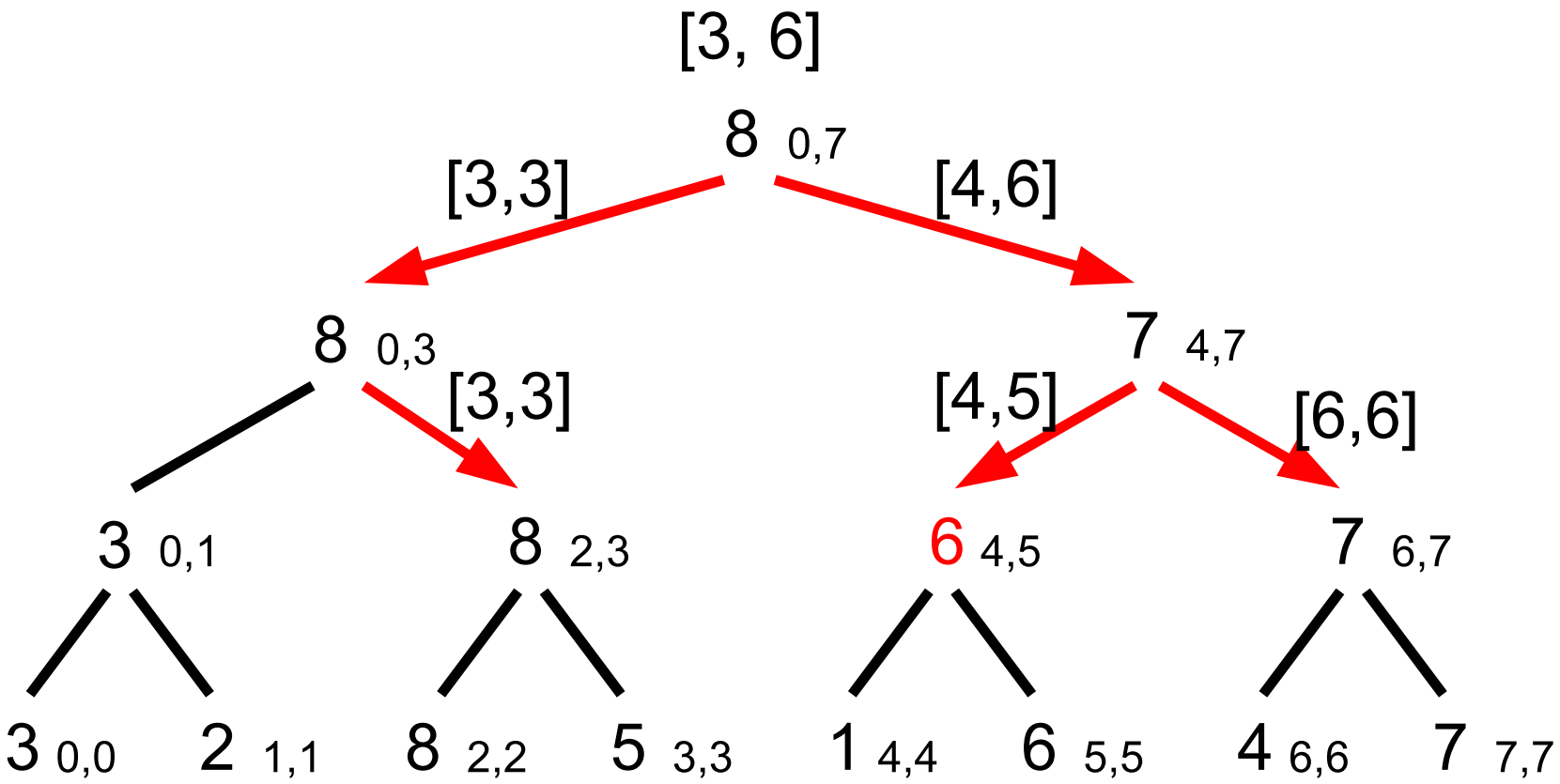
Drzewo przedziałowe



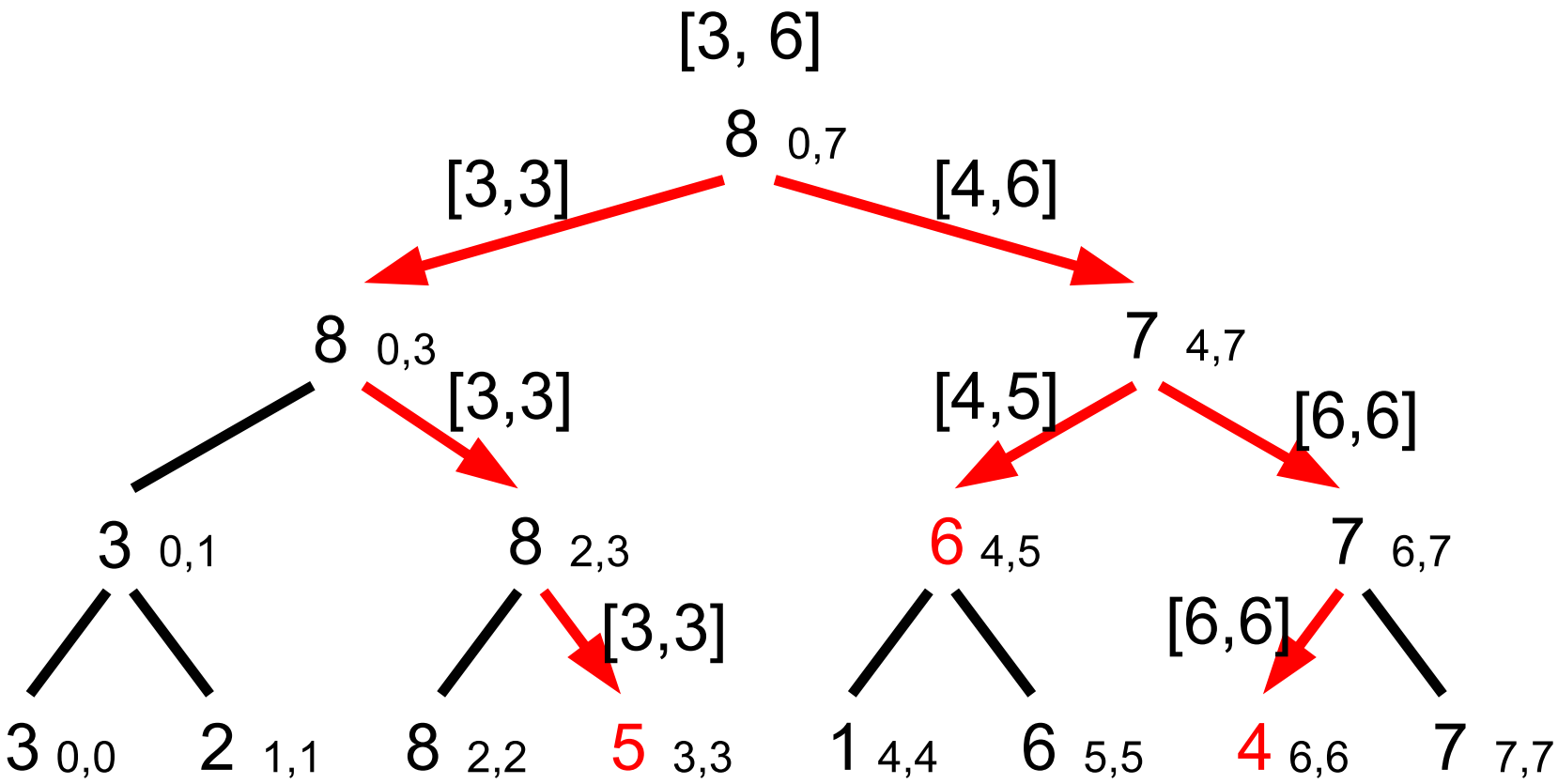
Drzewo przedziałowe



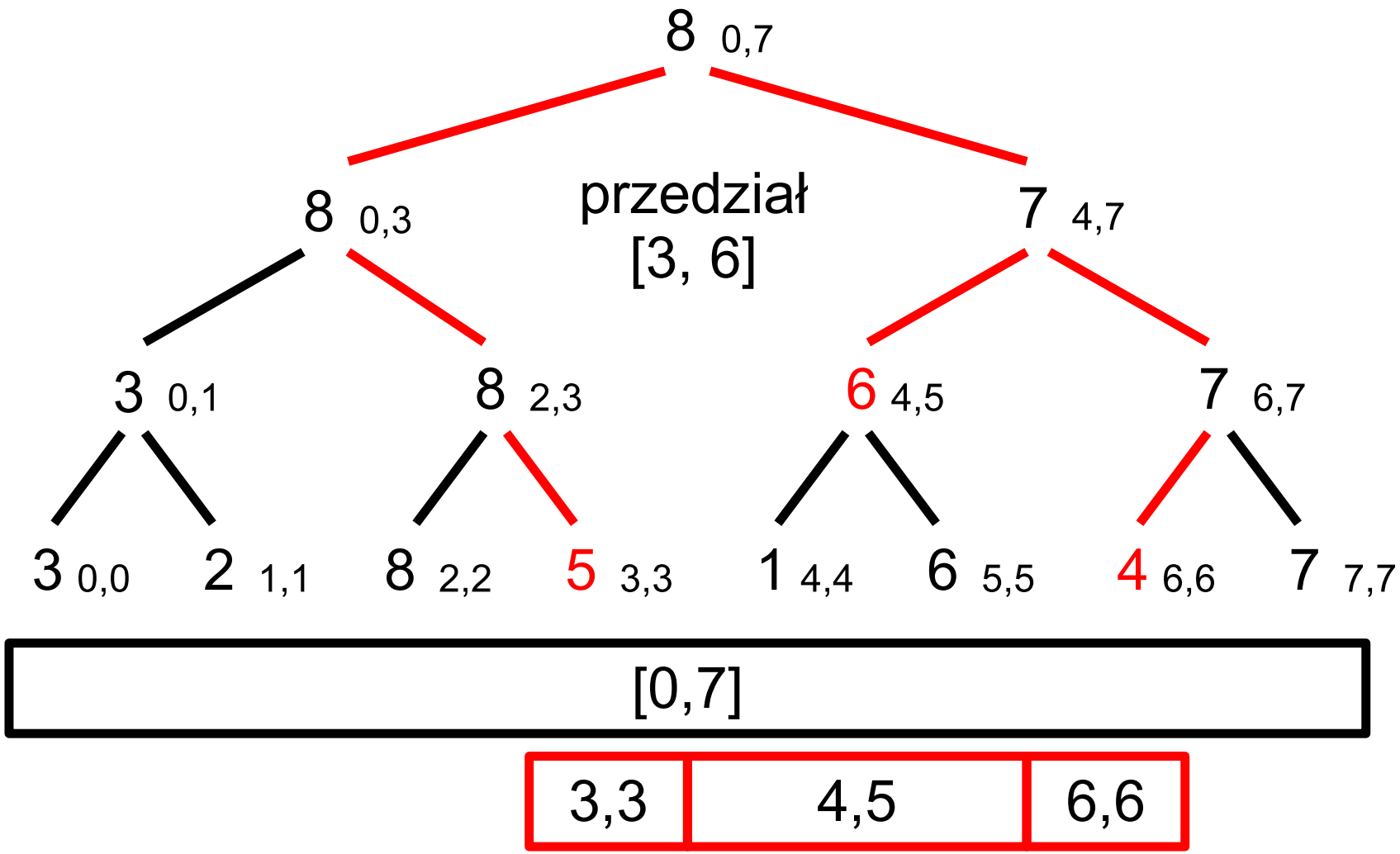
Drzewo przedziałowe



Drzewo przedziałowe



Drzewo przedziałowe



Drzewo przedziałowe

- złożoność odpowiedzi na m zapytań: $O(n + m \log n)$; budowa drzewa zajmuje $O(n)$