Algorytmy i struktury danych

Drzewa zrównoważone

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

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Drzewa zrównoważone

- drzewo doskonale zrównoważone to takie, w którym dla każdego węzła rozmiary poddrzew różnią się co najwyżej o jeden
- drzewo zrównoważone — długość dowolnej ścieżki z węzła do liścia różni się co najwyżej o 1 od wysokości tego węzła
- drzewo w przybliżeniu zrównoważone — długość dowolnej ścieżki z węzła do liścia różni się co najwyżej 2 razy od wysokości tego węzła

Drzewa zrównoważone

- złożoność operacji na drzewie jest proporcjonalna do jego wysokości
- aby zatem operacje te były efektywne, należy kontrolować wysokość drzewa — celem jest utrzymanie wysokości proporcjonalnej do logarytmu liczby węzłów (wtedy operacje zajmują $O(\log n)$)
- równoważenie — przywracanie drzewu właściwości, zapewniających logarytmiczną wysokość

Drzewa AVL

- AVL: Aldelson-Velskii i Landis
- kryterium równoważenia: drzewo jest zrównoważone wtedy, i tylko wtedy, gdy dla każdego węzła wysokości dwóch jego poddrzew różnią się co najwyżej o 1
- można pokazać, że wysokość takiego drzewa z n węzłami $h(n)$ będzie

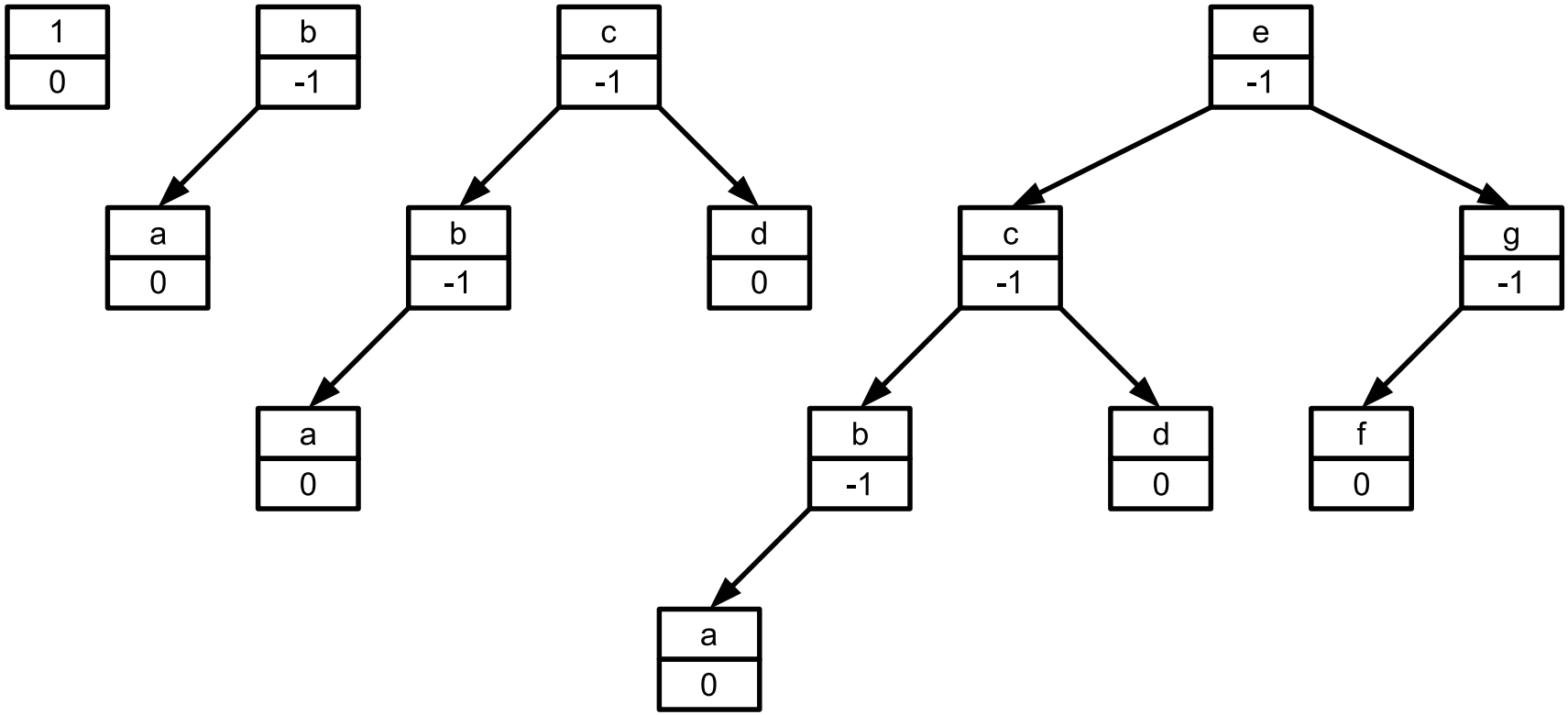
$$\log(n+1) \leq h(n) \leq 1.4404 \log(n+2) - 0.328$$

- drzewo takie będzie więc co najwyżej 45% wyższe od doskonale zrównoważonego
- w każdym węźle przechowujemy dodatkową informację (*wywazenie*): -1, gdy lewe poddrzewo jest wyższe, 0 gdy poddrzewa są równej wysokości, $+1$ gdy prawe drzewo jest wyższe

Drzewa AVL

- "pesymistycznym" przypadkiem drzewa AVL jest drzewo Fibonacciego
- drzewo takie powstaje poprzez dołączenie do korzenia dwóch poddrzew Fibonacciego stopnia o 1 mniejszego i o 2 mniejszego

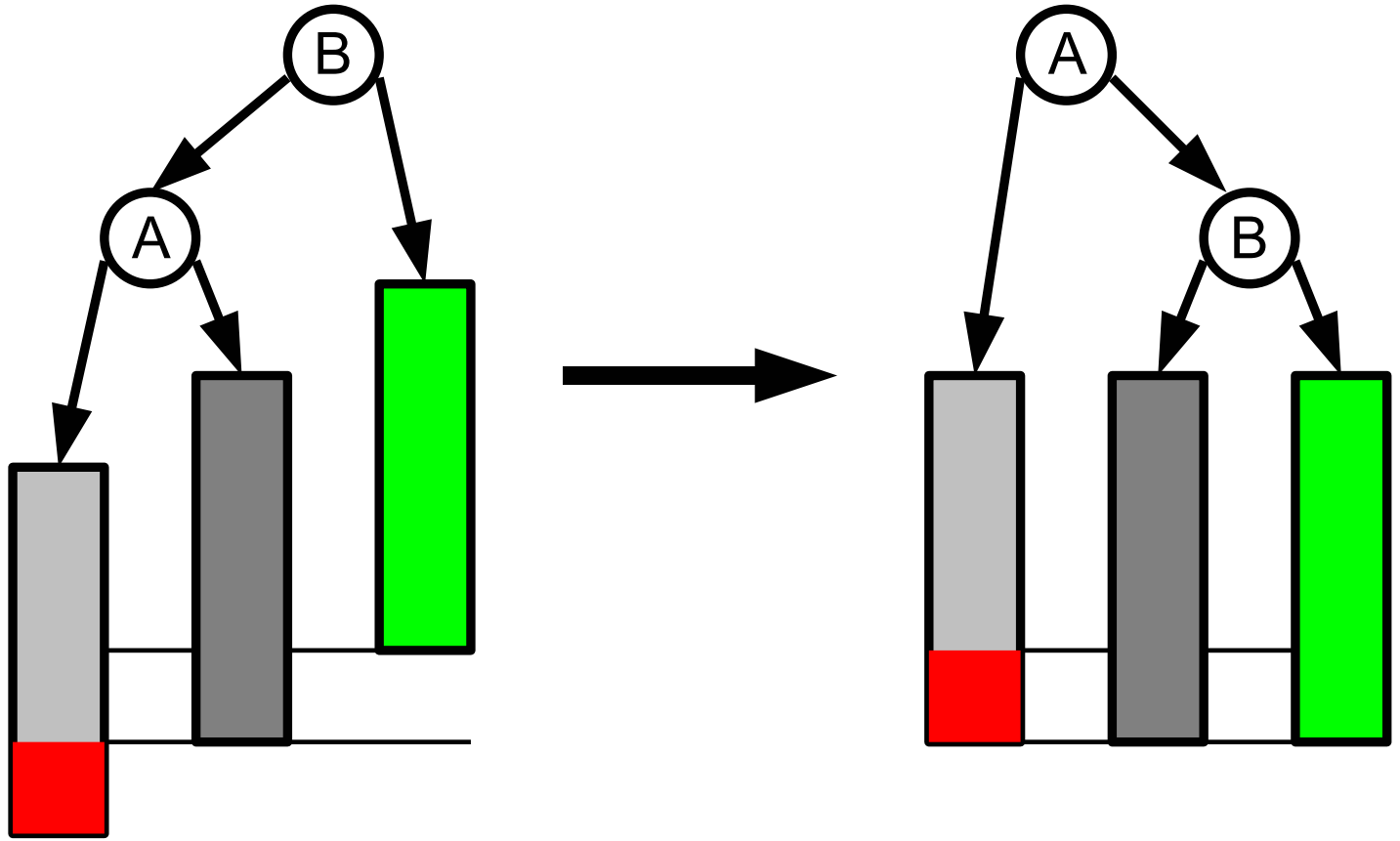
Drzewa AVL



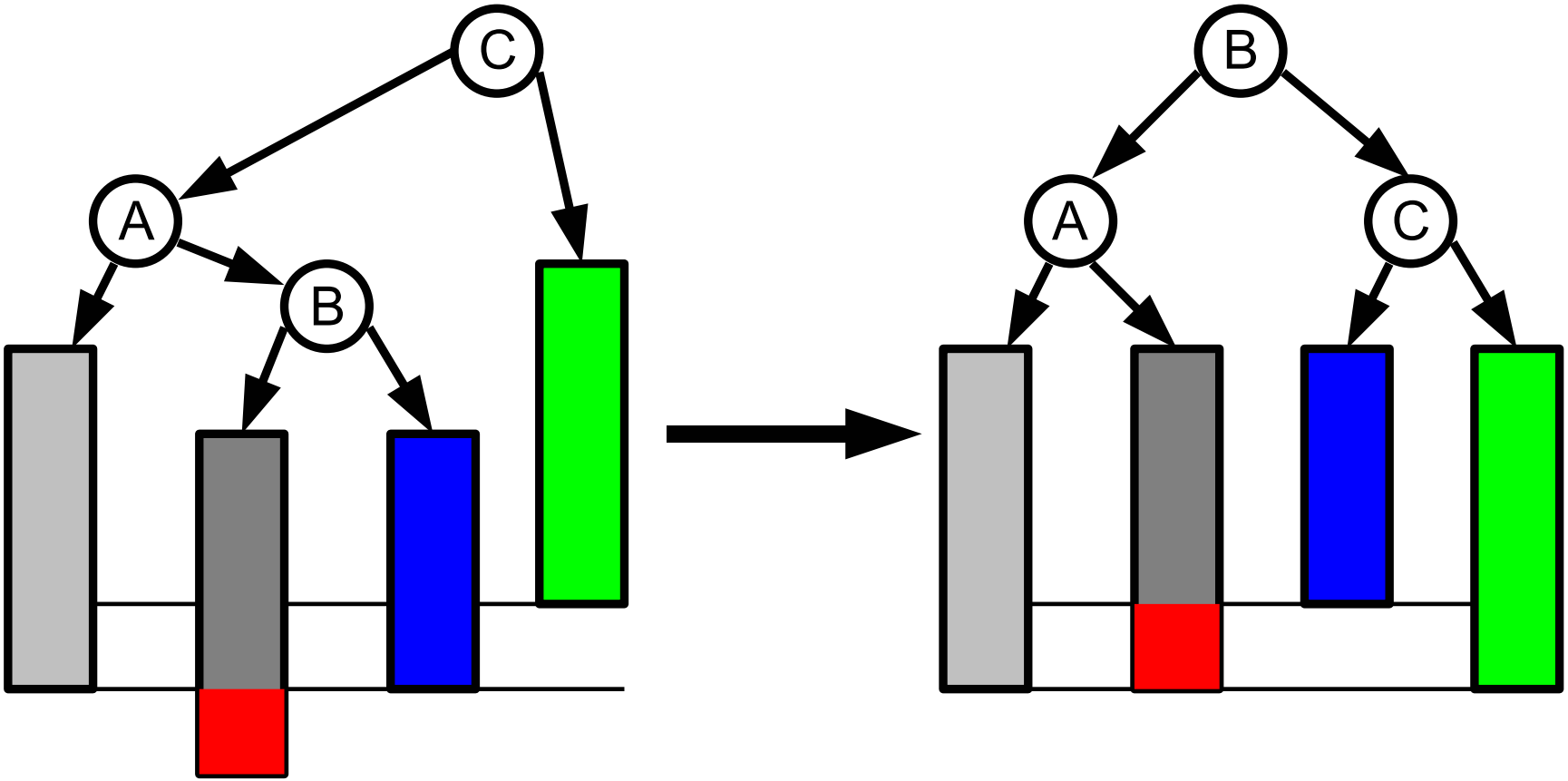
Wstawianie węzła

- wstawianie rozpoczynamy od znalezienia miejsca, w które należy wstawić węzeł (procedura wstawiania w zwykłe drzewo binarne)
- załóżmy, że wstawiliśmy węzeł w lewe poddrzewo węzła *parent*, wówczas:
  - jeżeli $parent.wywazenie = +1$ — zrównoważenie drzewa się poprawia,
  - jeżeli $parent.wywazenie = 0$ — lewe poddrzewo staje się wyższe od prawego, ale kryterium zrównoważenia jest nadal zachowane,
  - jeżeli $parent.wywazenie = -1$ — kryterium zrównoważenia zostaje złamane i należy przebudować drzewo

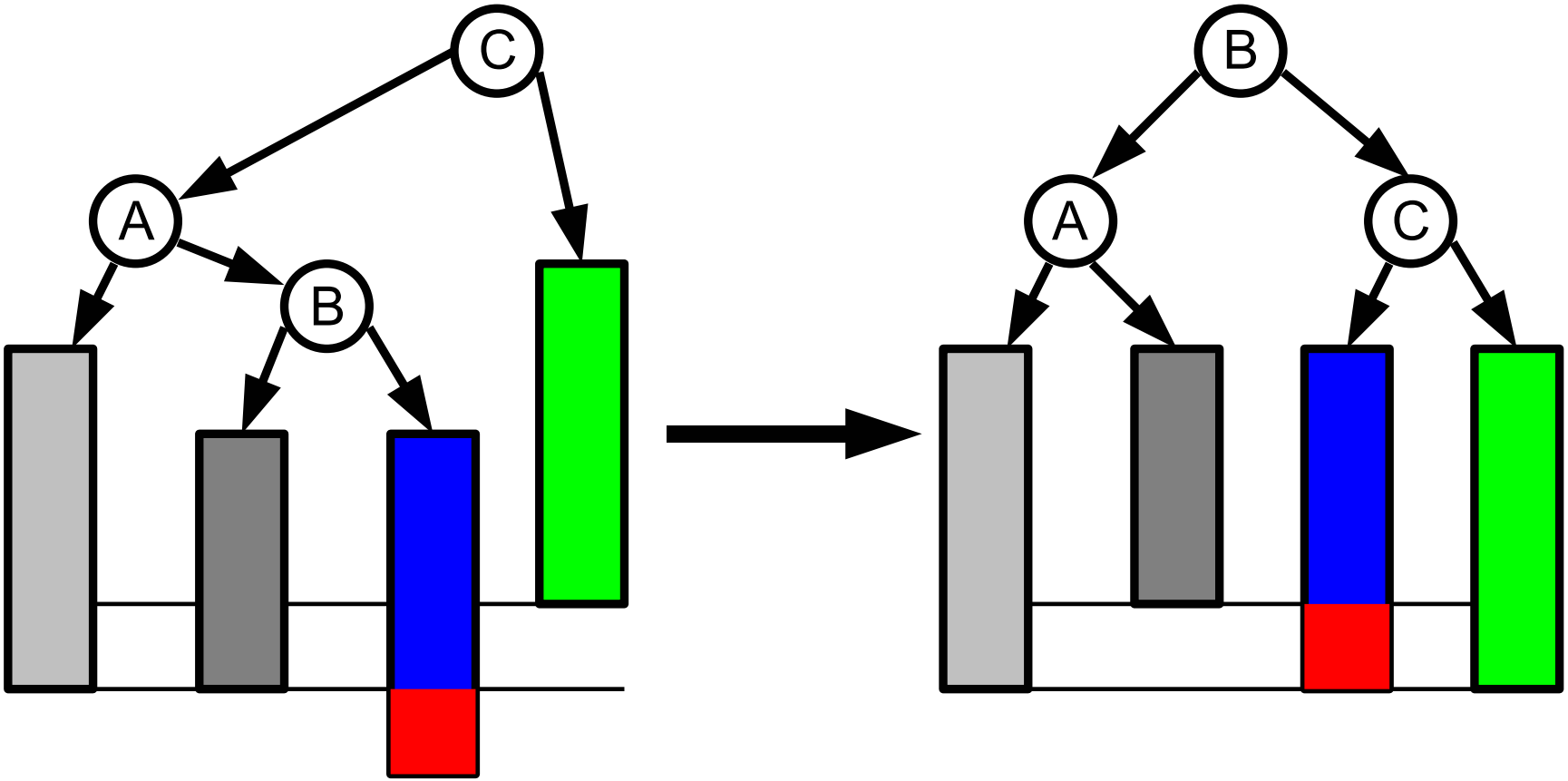
Przebudowanie drzewa



Przebudowanie drzewa



Przebudowanie drzewa



Wstawianie węzła

- po wstawieniu węzła musimy cofnąć się po ścieżce wyszukiwania, i w każdym węźle sprawdzić zrównoważenie (procedura wstawiająca do poddrzewa musi zwracać informację, czy zwiększyła się wysokość drzewa)

Wyważenie

```
WywazL(a)
  if a.wywazenie = −1 then
        b = a.parent
        if b.parent ≠ nullptr then
              if b.parent.left = b then b.parent.left = a
              else b.parent.right = a
        else
              root = a
        end if
        a.parent = b.parent
        b.parent = a
        b.left = a.right
        if b.left ≠ nullptr then b.left.parent = b
        a.right = b
        a.wywazenie = 0
        b.wywazenie = 0
  else
        {a.wywazenie ≥ 0}
        ...
  end if
```

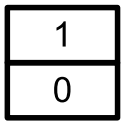
Wyważenie

$\{$**else** $a.wywazenie \geq 0\}$
$b = a.right$
$c = a.parent$
if $c.parent \neq nullptr$ **then**
    if $c.parent.left = c$ **then** $c.parent.left = b$
    else $c.parent.right = b$
else
    $root = b$
end if
$b.parent = c.parent$
$a.right = b.left$
if $a.right \neq nullptr$ **then** $a.right.parent = a$
$c.left = a.right$
if $c.left \neq nullptr$ **then** $c.left.parent = c$
$b.left = a$
$a.parent = b$
$b.right = c$
$c.parent = b$
$a.wywazenie = 0$
$b.wywazenie = 0$
$c.wywazenie = 0$

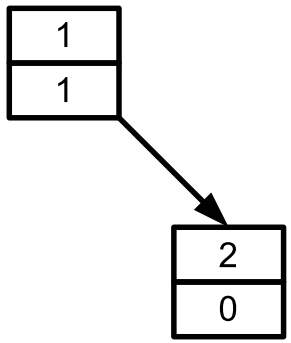
Wstawianie węzła

```
Wstaw(root, klucz)
 1: if klucz < root.key then
 2:         if root.left = nullptr then
 3:                 root.left =  NewNode;
 4:                 root.left.klucz = klucz
 5:                 root.left.wywazenie = 0
 6:                 inc = TRUE
 7:         else
 8:                 inc =Wstaw(root.left, klucz)
 9:         end if
10:         if inc then
11:                 root.wywazenie = root.wywazenie − 1
12:                 inc = FALSE
13:                 if root.wywazenie = −2 then WywazL(root.left)
14:                 else if root.wywazenie = −1 then inc = TRUE
15:         end if
16: else if klucz > root.key then
17:         symetrycznie...
18: else
19:         error klucz już istnieje
20: end if
21: return inc
```

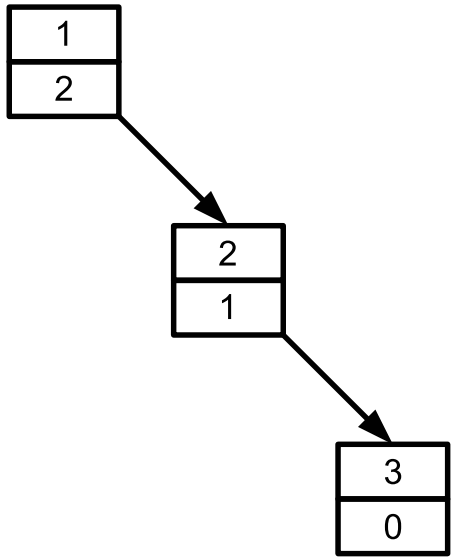
Wstawianie do AVL



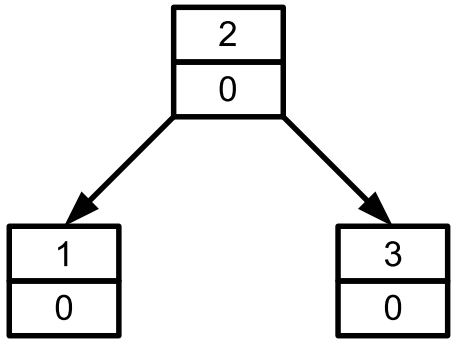
Wstawianie do AVL



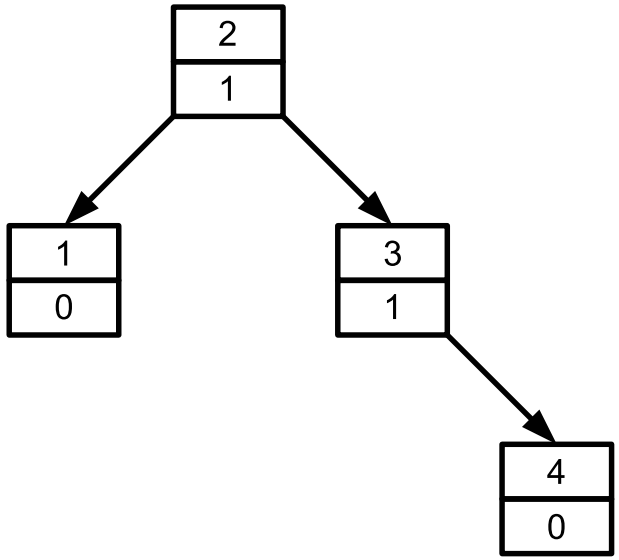
Wstawianie do AVL



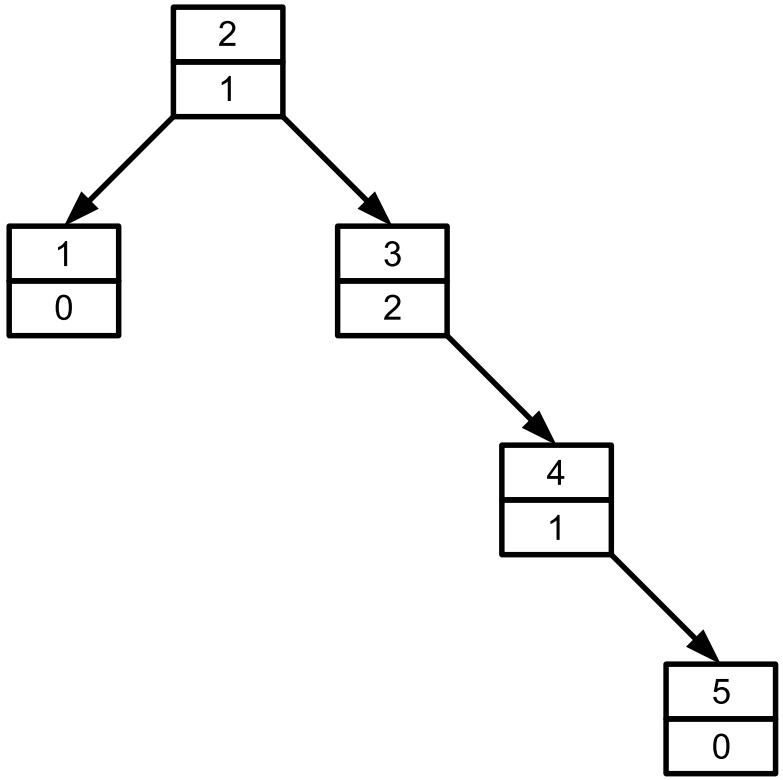
Wstawianie do AVL



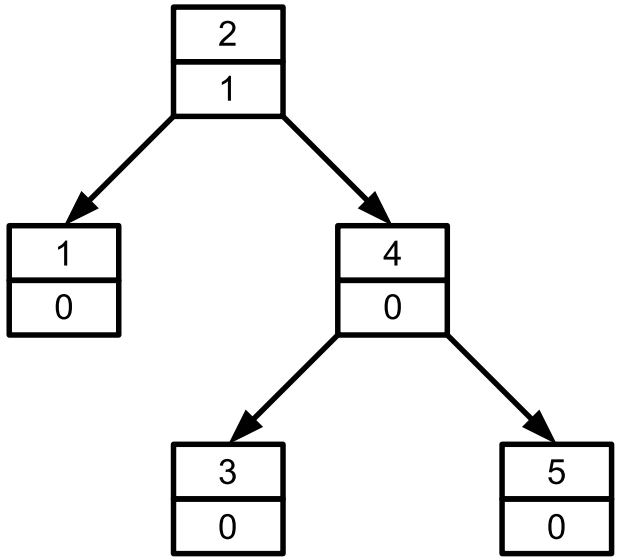
Wstawianie do AVL



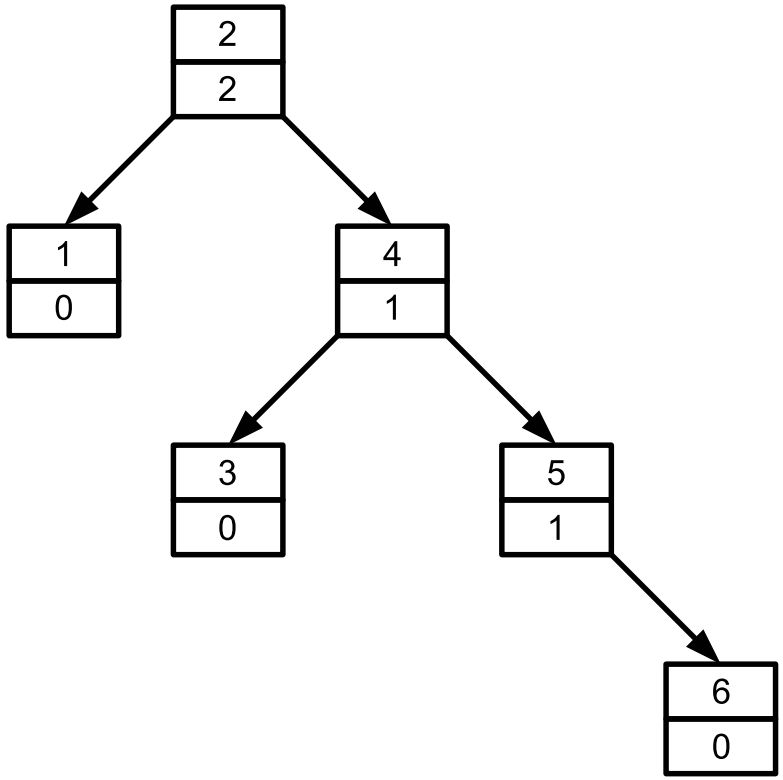
Wstawianie do AVL



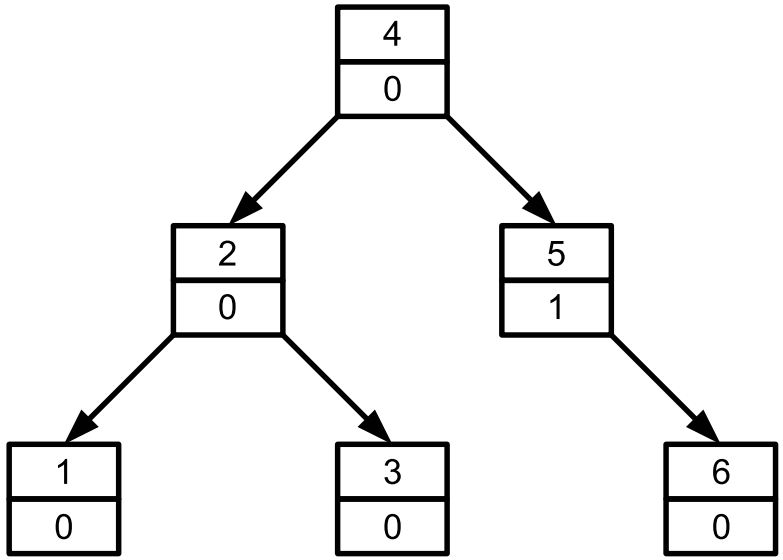
Wstawianie do AVL



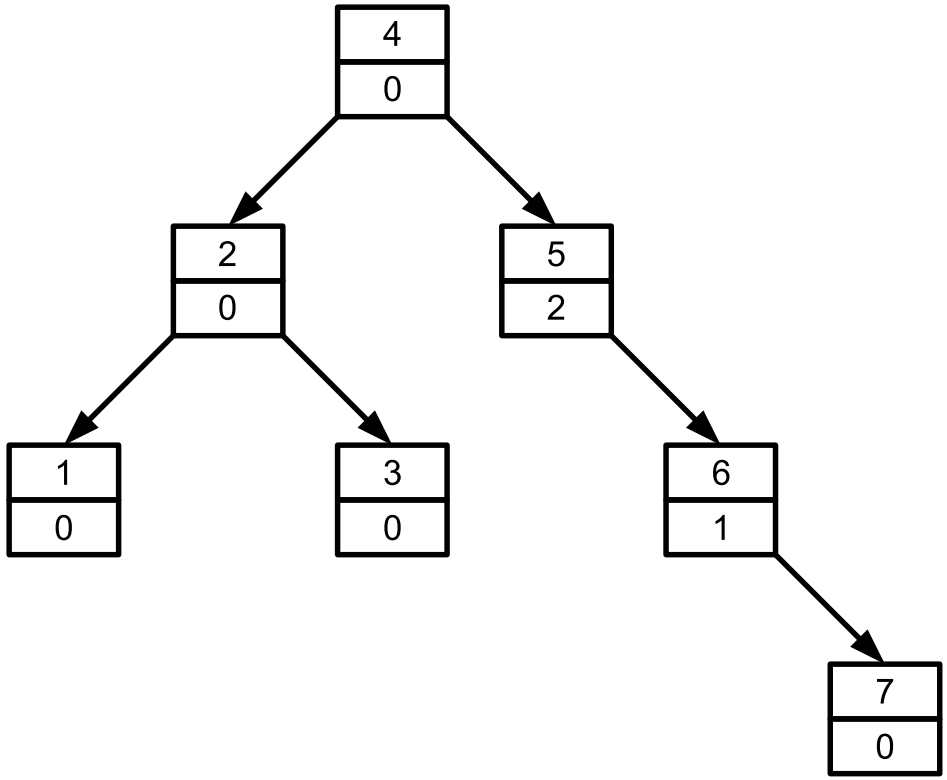
Wstawianie do AVL



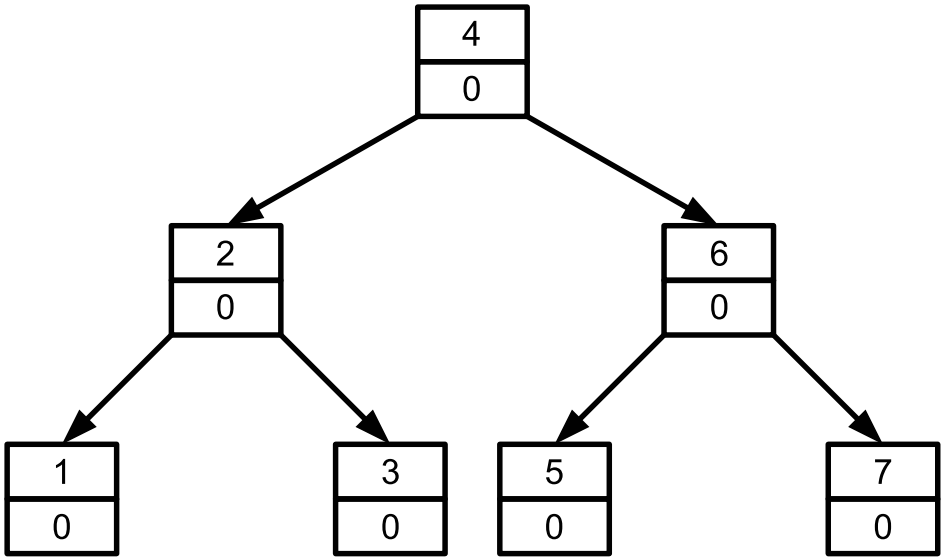
Wstawianie do AVL



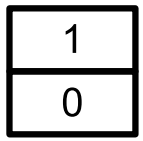
Wstawianie do AVL



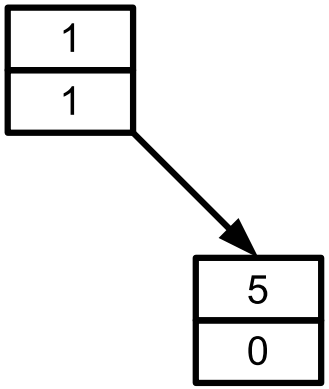
Wstawianie do AVL



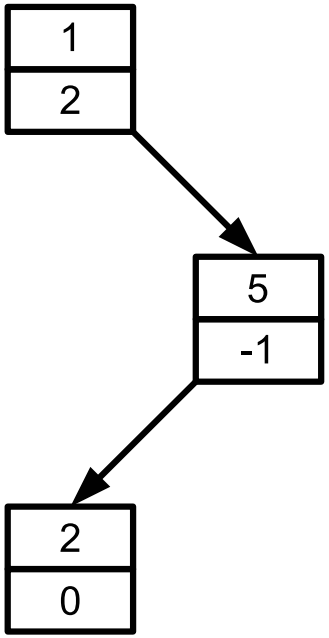
Wstawianie do AVL



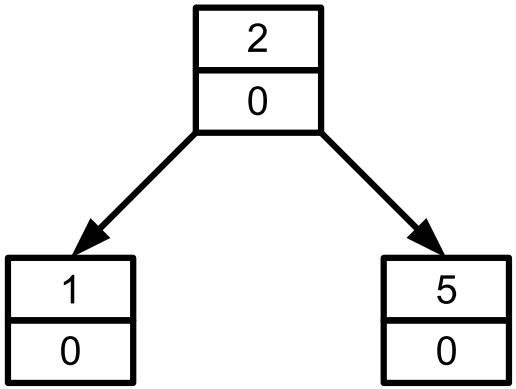
Wstawianie do AVL



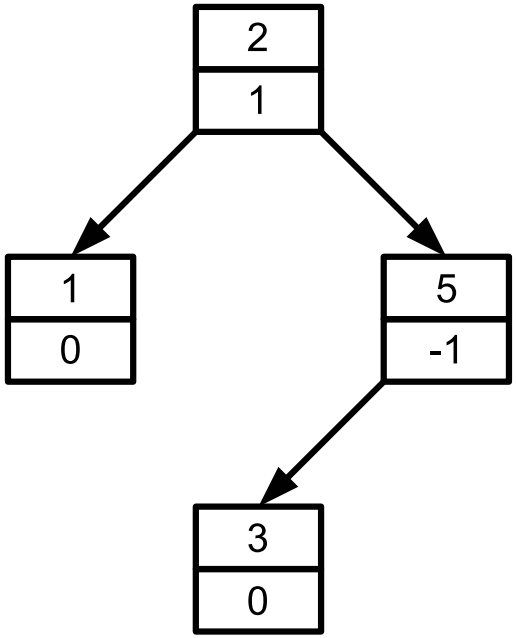
Wstawianie do AVL



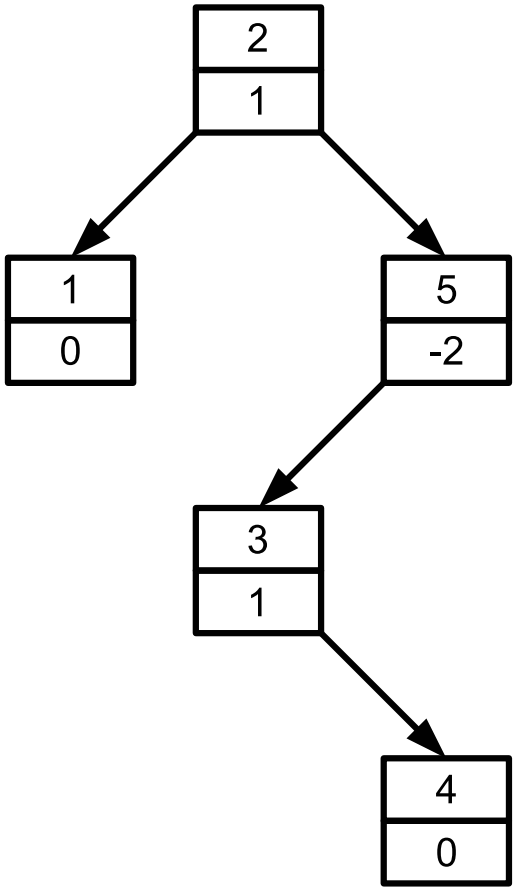
Wstawianie do AVL



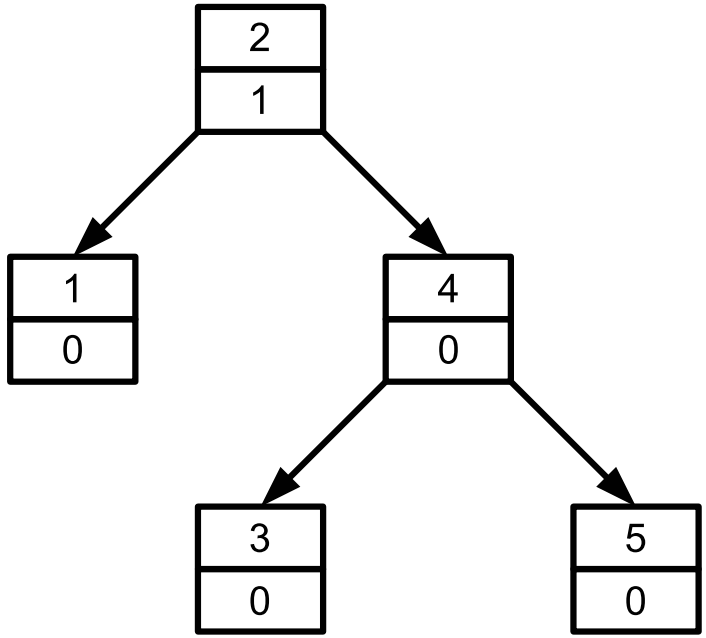
Wstawianie do AVL



Wstawianie do AVL



Wstawianie do AVL



Wstawianie węzła

- empiryczne testy:
  - oczekiwana wysokość $h(n) = \log(n) + c$, gdzie $(c \approx 0.25)$
  - średnio na dwa wstawienia konieczne jest jedno zrównoważenie
  - obydwa rodzaje zamian są równo prawdopodobne

Usuwanie z drzewa

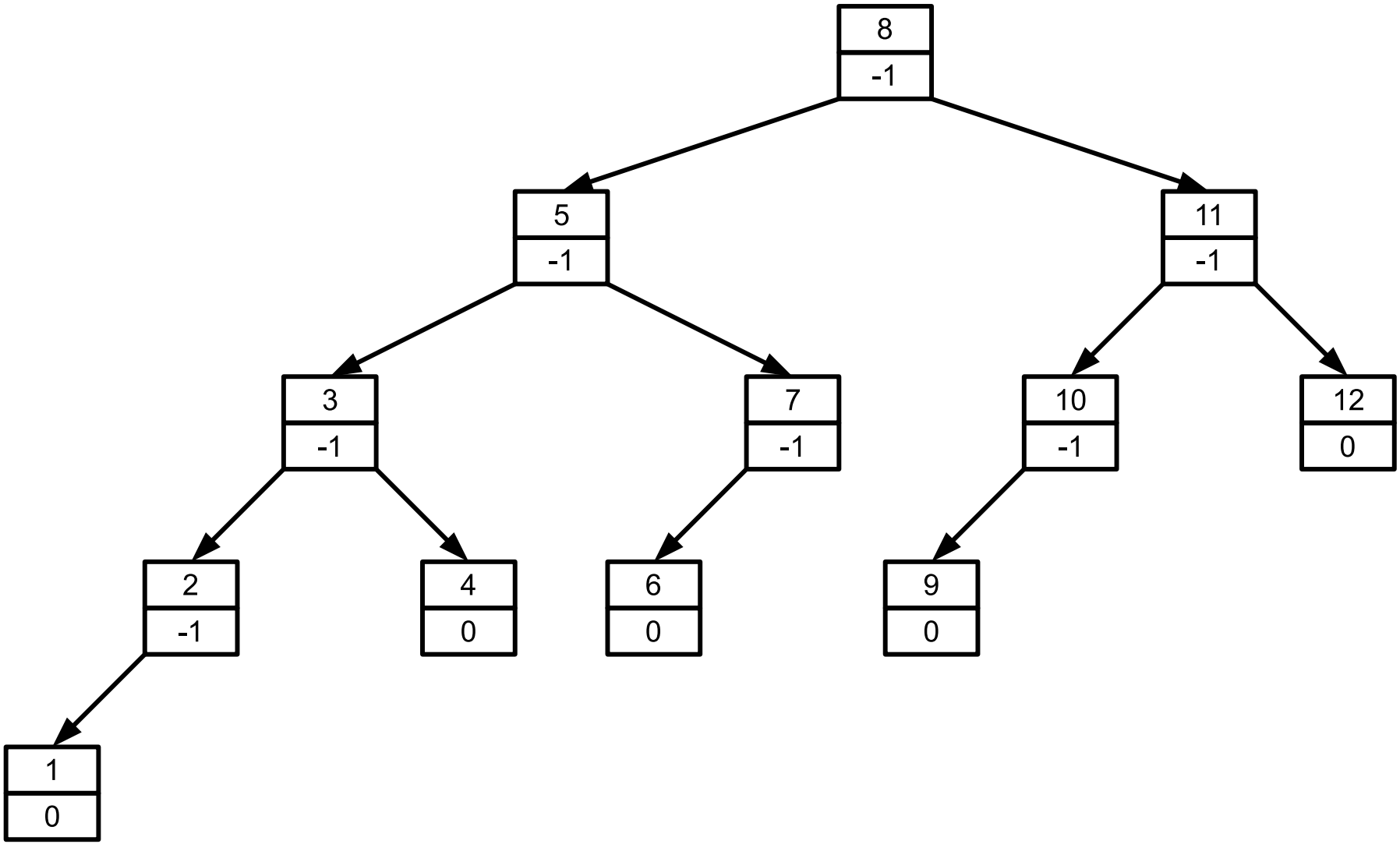
- najpierw usuwamy węzeł stosując procedurę usuwania ze zwykłego drzewa binarnego
- usunięcie może zaburzyć zrównoważenie, należy więc drzewo poprawić

Usuwanie z drzewa

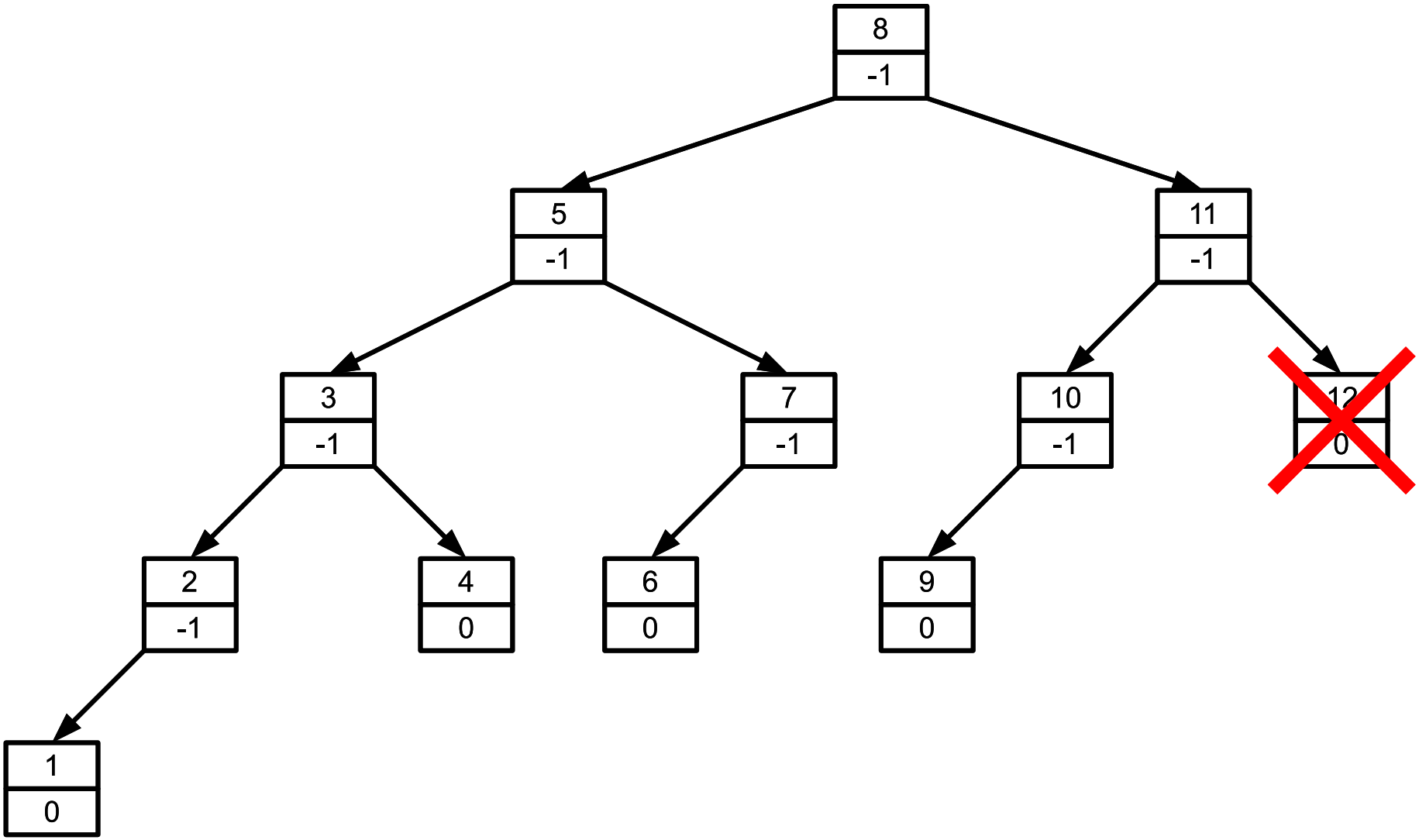
`Usun`(*root*, *klucz*)

1: `UsunBST`(*root*, *klucz*); niech *parent* będzie rodzicem usuniętego węzła, a *lewy* oznacza, czy usunięty węzeł był lewym potomkiem swojego rodzica
2: $dec = TRUE$
3: **while** *dec* **and** $parent \neq nullptr$ **do**
4: **if** *lewy* **then**
5: $parent.wywazenie+ = 1$
6: **if** $parent.wywazenie == 1$ **then** $dec = FALSE$
7: **else if** $parent.wywazenie == 2$ **then** `WywazP`(*parent.right*)
8: **else**
9: $parent.wywazenie- = 1$
10: **if** $parent.wywazenie == -1$ **then** $dec = FALSE$
11: **else if** $parent.wywazenie == -2$ **then** `WywazL`(*parent.left*)
12: **end if**
13: **if** $parent.parent \neq nullptr$ **then** $lewy = (parent.parent.lewy = parent)$
14: $parent = parent.parent$
15: **end while**

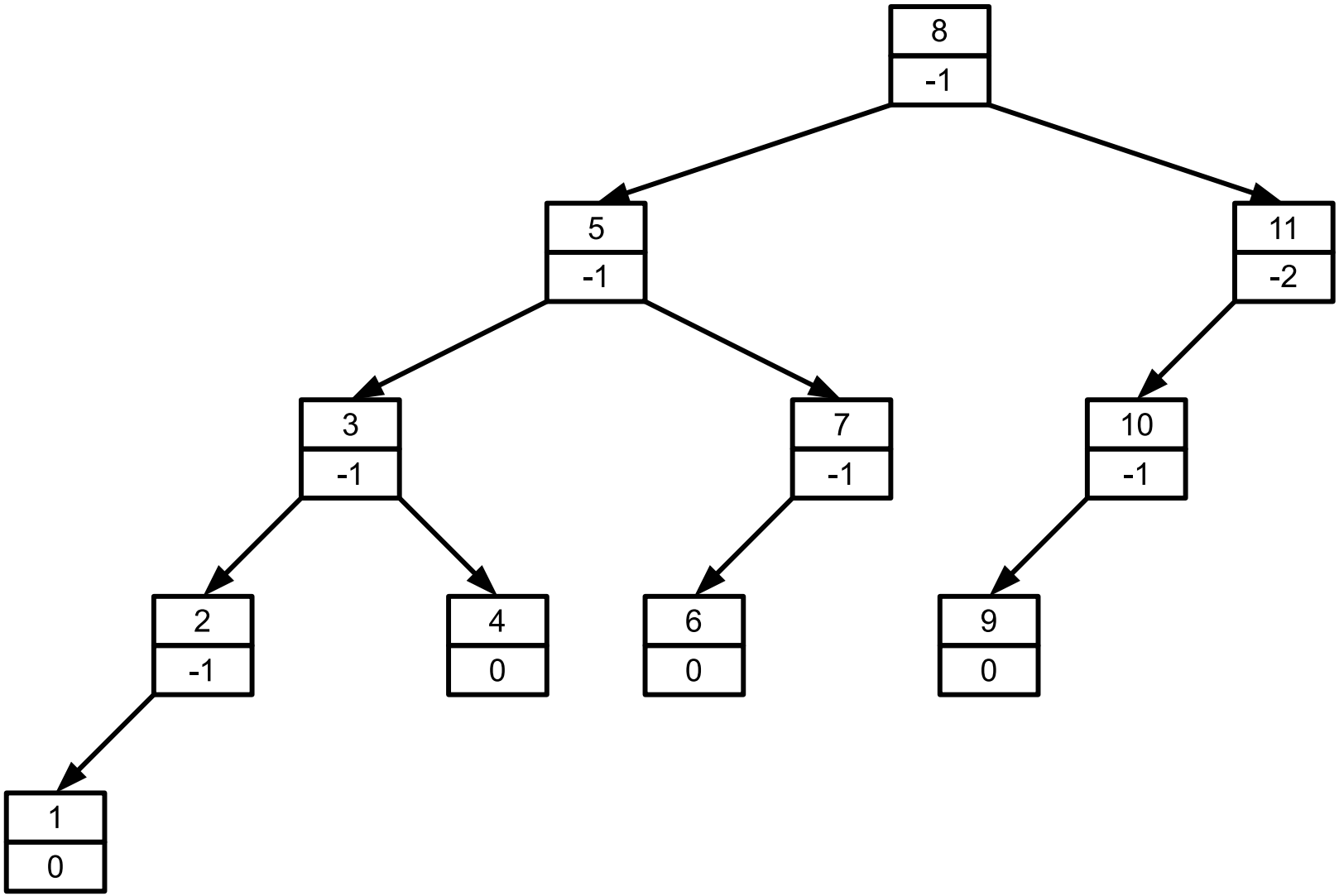
Usuwanie z AVL



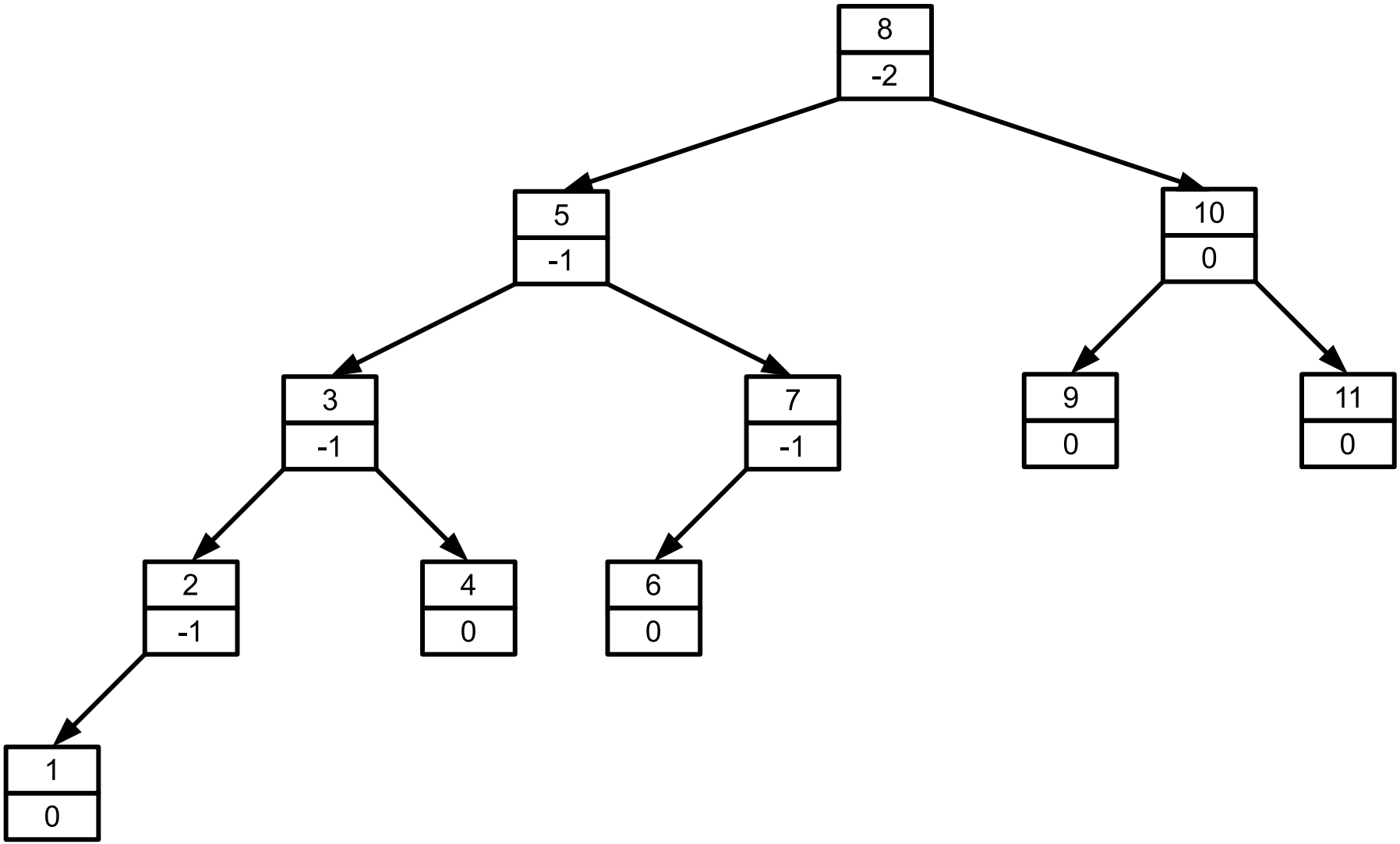
Usuwanie z AVL



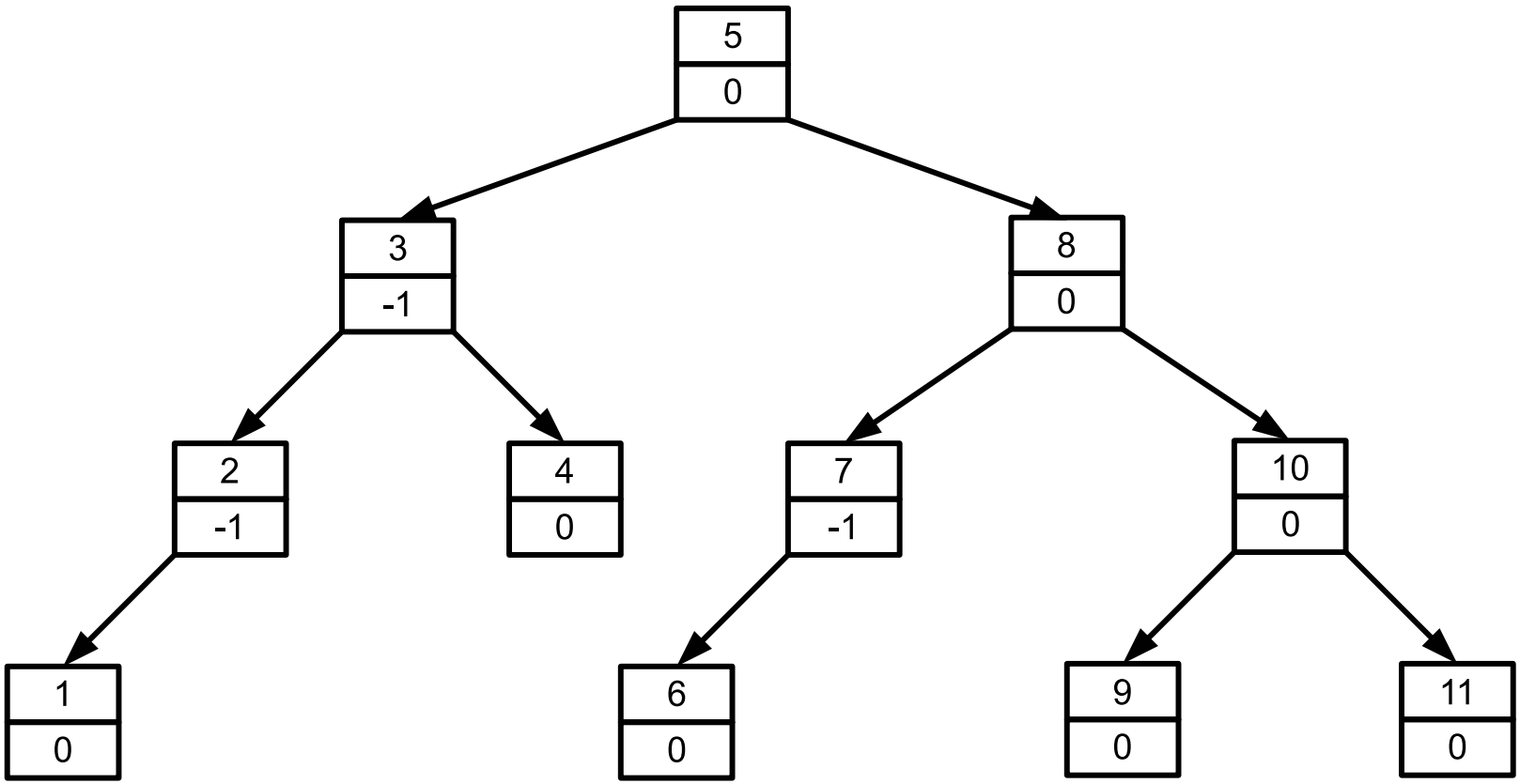
Usuwanie z AVL



Usuwanie z AVL



Usuwanie z AVL



Usuwanie z drzewa

- usunięcie węzła może doprowadzić do wykonania równoważenia w każdym węźle na ścieżce do korzenia (np. usunięcie skrajnie prawego węzła w drzewie Fibonacciego)
- wyniki empiryczne: jedna zamiana co ok. pięć usunięć węzłów

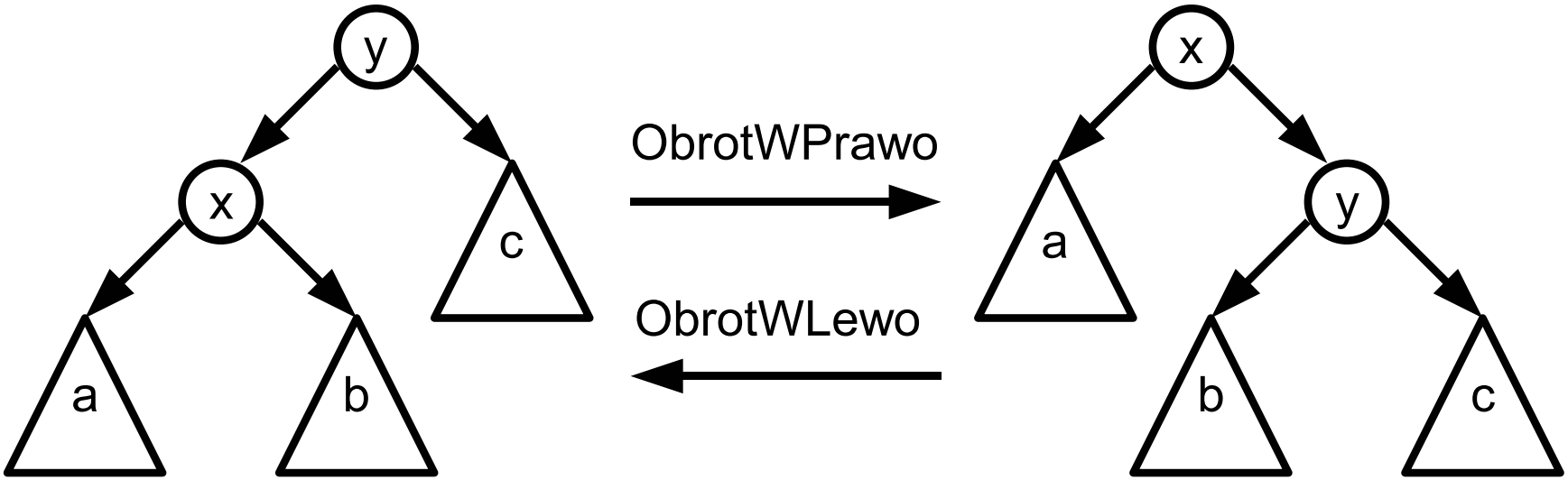
Drzewa czerwono-czarne

- w każdym węźle, oprócz klucza, przechowujemy także jego kolor, który może być czerwony lub czarny
- drzewa czerwono-czarne są w przybliżeniu zrównoważone — każda ścieżka jest co najwyżej dwa razy dłuższa od dowolnej innej
- drzewo czerwono-czarne o n węzłach ma wysokość co najwyżej $2\log(n+1)$

Drzewa czerwono-czarne

1. każdy węzeł jest albo czerwony, albo czarny
2. każdy liść jest czarny (przy czym liśćmi są tu puste wskaźniki)
3. jeżeli węzeł jest czerwony, to oba jego węzły potomne są czarne
4. każda prosta ścieżka z ustalonego węzła do liścia ma tyle samo czarnych węzłów

Rotacja węzła



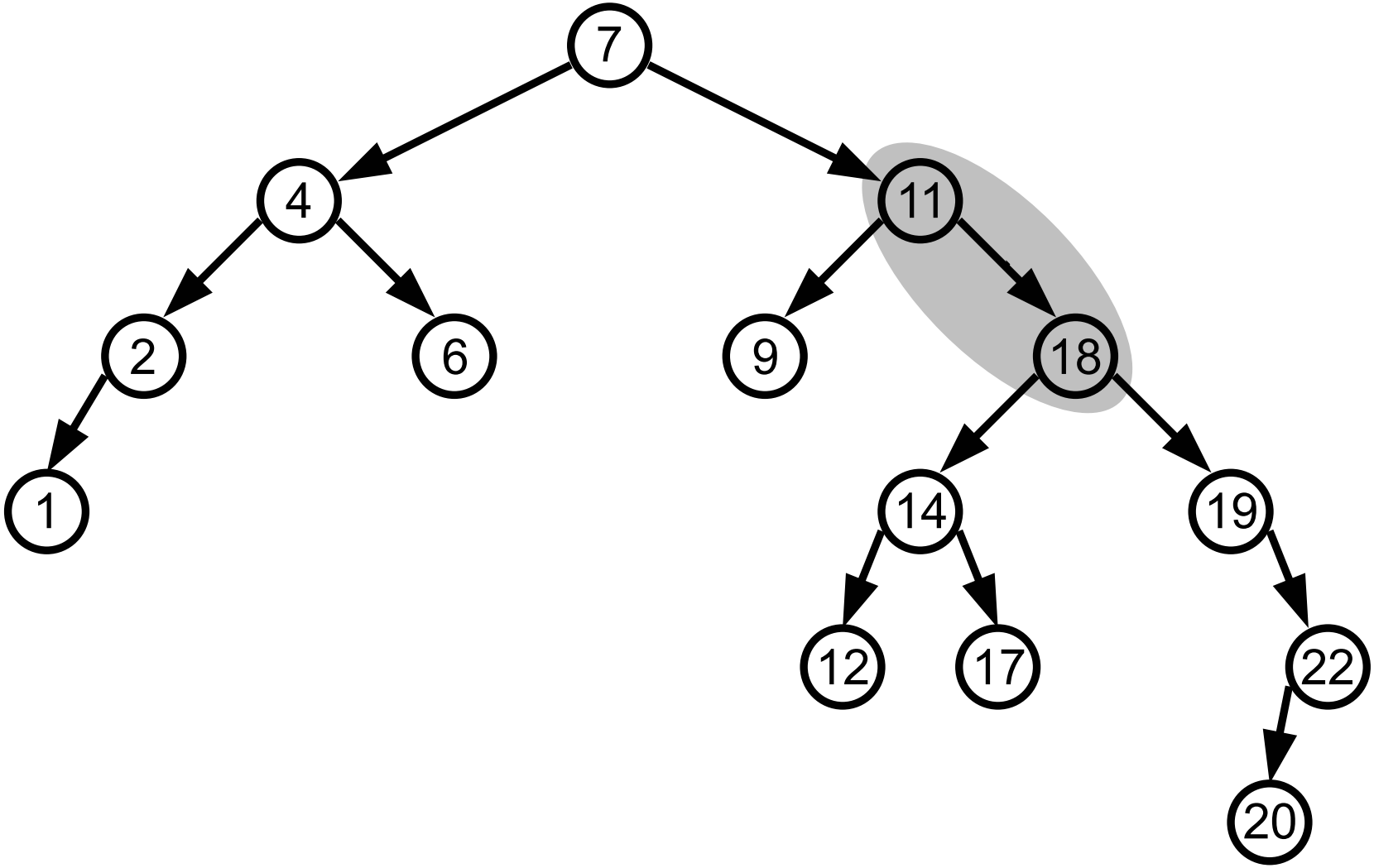
Rotacja węzła

```
ObrotWLewo(T, w)
 1: t = w.right
 2: w.right = t.left
 3: if t.left ≠ nullptr then
 4:         t.left.parent = x
 5: end if
 6: t.parent = w.parent
 7: if w.parent = nullptr then
 8:         T.root = t
 9: else if w = w.parent.left then
10:         w.parent.left = t
11: else
12:         w.parent.right = t
13: end if
14: t.left = w
15: w.parent = t
```

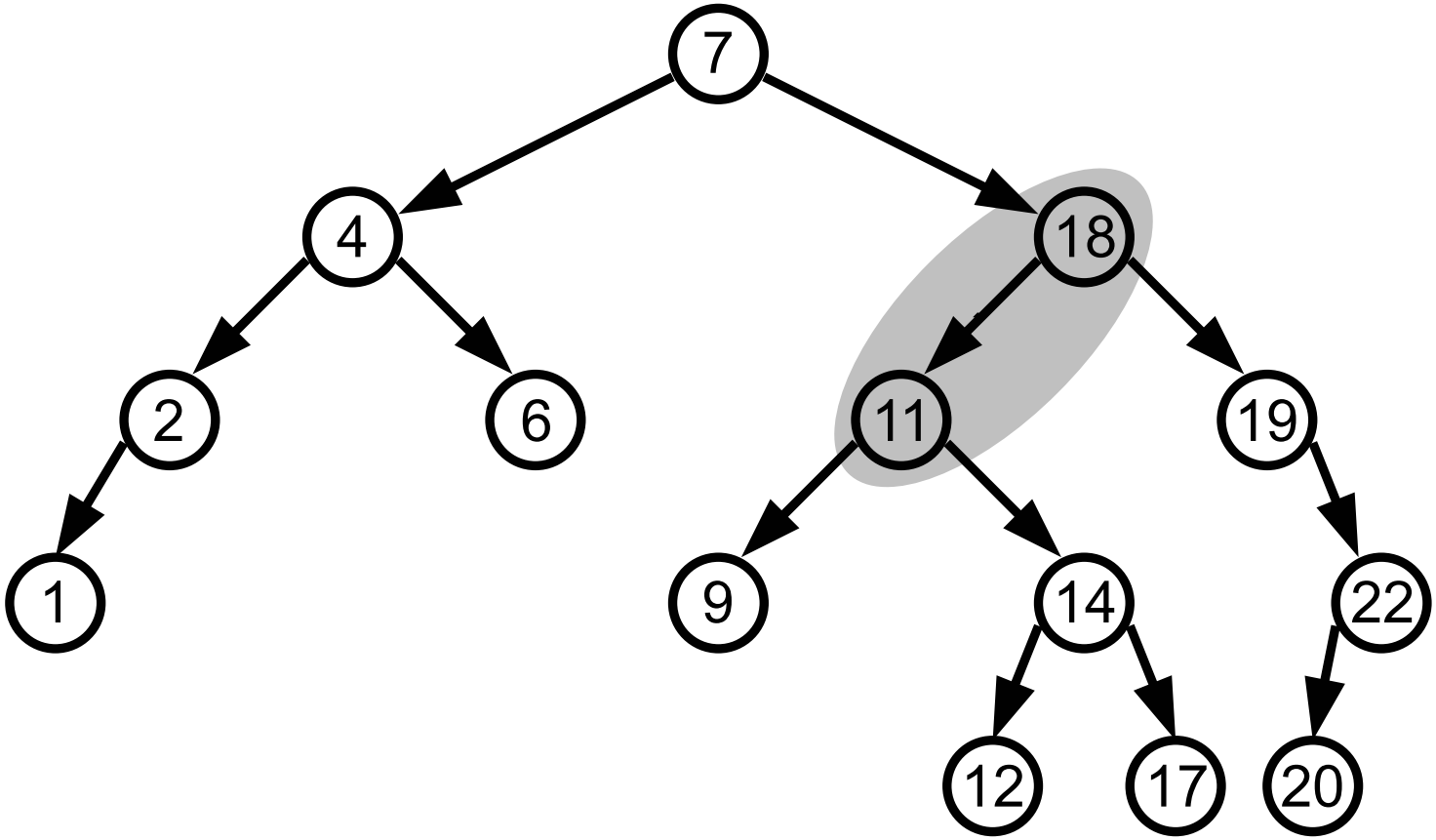
Rotacja węzła

```
ObrotWPrawo(T, w)
 1: t = w.left
 2: w.left = t.right
 3: if t.right ≠ nullptr then
 4:         t.right.parent = x
 5: end if
 6: t.parent = w.parent
 7: if w.parent = nullptr then
 8:         T.root = t
 9: else if w = w.parent.right then
10:         w.parent.right = t
11: else
12:         w.parent.left = t
13: end if
14: t.right = w
15: w.parent = t
```

Rotacja węzła



Rotacja węzła



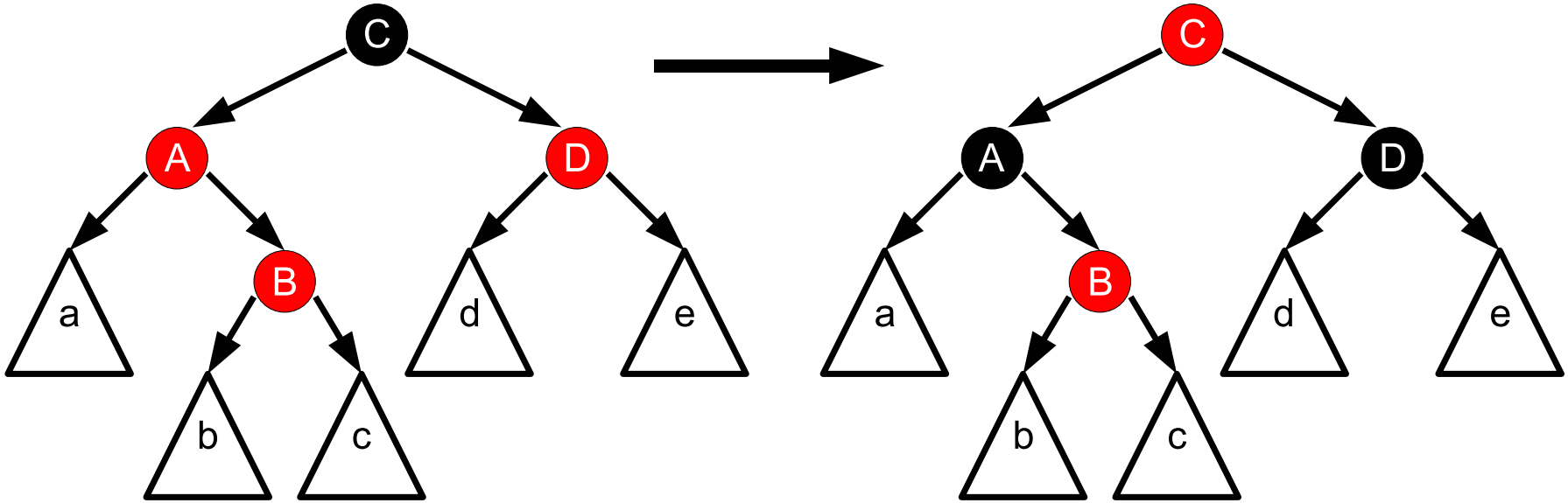
Wstawianie węzła

- na początek wstawiamy węzeł do drzewa tak, jakby było to zwykłe drzewo binarne
- wstawiony węzeł kolorujemy na czerwono
- może to zaburzyć właściwość nr 3 — jeżeli węzeł jest czerwony, to oba jego węzły potomne są czarne
- stanie się tak, gdy rodzic wstawionego węzła jest czerwony
- wówczas “przenosimy” zaburzenie piętro wyżej
- załóżmy, że rodzic wstawionego węzła jest lewym potomkiem swojego rodzica

Wstawianie węzła

- przypadek 1: rodzic rodzica węzła ma dwoje czerwonych potomków (brat rodzica jest czerwony)
  - wówczas "przenosimy" jego czarny kolor na potomków
  - przechodzimy do rodzica rodzica węzła — stał się czerwony i gdy jego rodzic też jest czerwony, trzeba poprawiać dalej

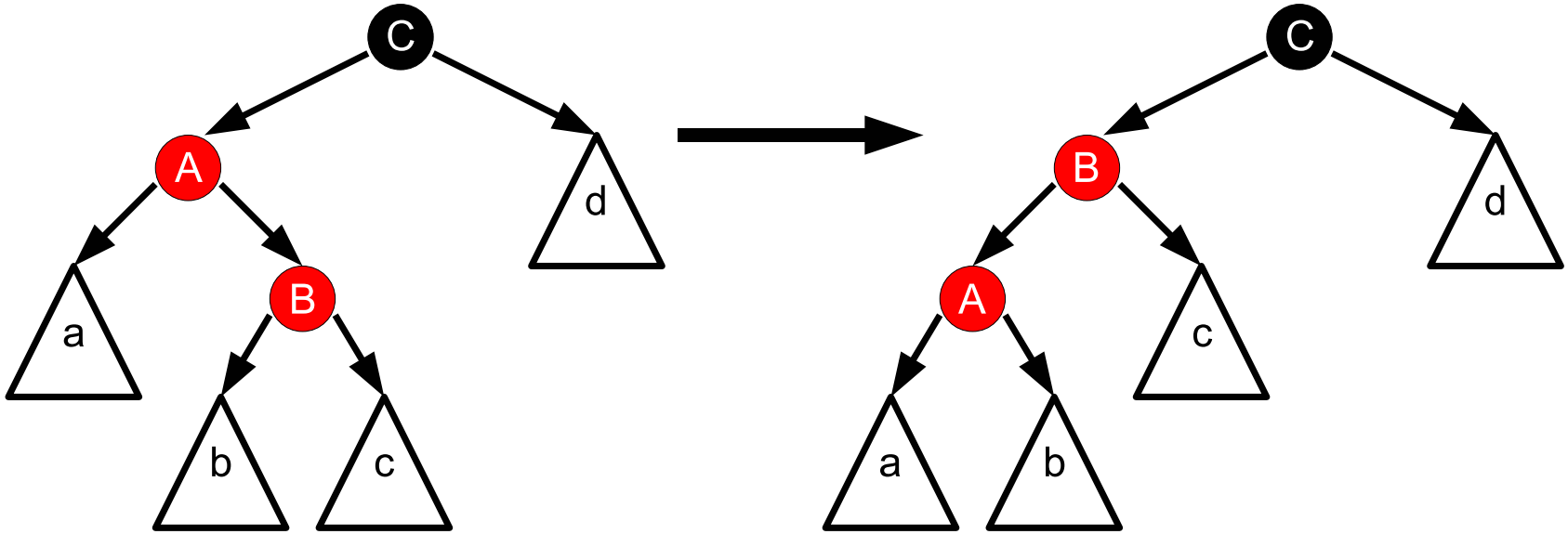
Przypadek 1



Wstawianie węzła

- przypadek 2: brat rodzica jest czarny, a bieżący węzeł jest prawym potomkiem swojego rodzica
  - wykonujemy obrót w lewo rodzica węzła
  - bieżącym węzłem staje się były rodzic węzła
  - przechodzimy do przypadku 3

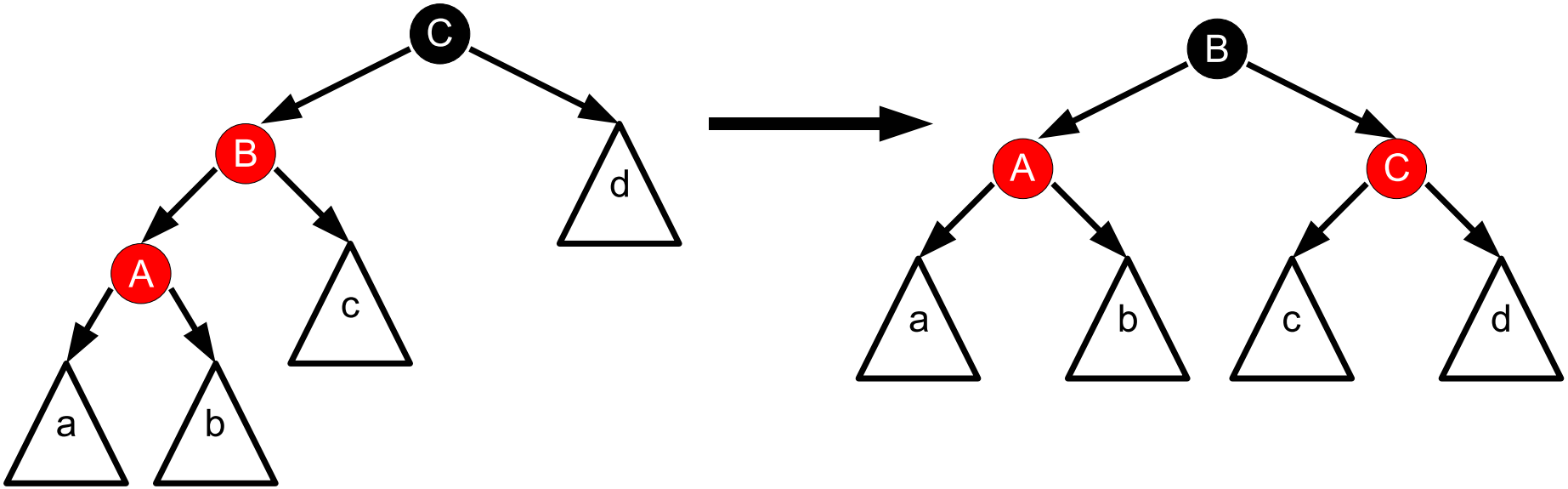
Przypadek 2



Wstawianie węzła

- przypadek 3: brat rodzica jest czarny, a bieżący węzeł jest lewym potomkiem swojego rodzica
  - przekolorowujemy rodzica węzła na czarno, a jego rodzica na czerwono
  - wykonujemy obrót w prawo rodzica rodzica węzła
  - przerywamy pętlę — nie trzeba już nic poprawiać

Przypadek 3



Wstawianie węzła

```
Wstaw(T, w)
 1: WstawBST(T, w)
 2: w.color = RED
 3: while w ≠ T.root and w.parent.color = RED do
 4:         if w.parent = w.parent.parent.left then
 5:                 t = w.parent.parent.left
 6:                 if t.color = RED then
 7:                         w.parent.color = BLACK
 8:                         t.color = BLACK
 9:                         w.parent.parent.color = BLACK
10:                         w = w.parent.parent
11:                 else
12:                         if w = w.parent.right then
13:                                 w = w.parent
14:                                 ObrotWLewo(T, w)
15:                         end if
16:                         w.parent.color = BLACK
17:                         w.parent.parent.color = RED
18:                         ObrotWPrawo(T, w.parent.parent)
19:                 end if
20:         else
21:                 analogicznie jak wyżej, zamieniając miejscami left i right
22:         end if
23: end while
24: T.root.color = BLACK
```

Wstawienie węzła

- złożoność procedury: $O(\log n)$
- każde wstawienie wykona co najwyżej dwie rotacje (w przypadku 2 i przypadku 3)

Usuwanie węzła

- węzeł usuwamy tak, jak w zwykłym drzewie binarnym
  (ale kopiując węzeł, nie kopiujemy koloru)
- jeżeli faktycznie usunięty węzeł jest czerwony, nie ma konieczności naprawy drzewa (nie są naruszone żadne właściwości)
- jeżeli faktycznie usuwany węzeł jest czarny, należy poprawić drzewo (skraca się część czarnych ścieżek)

Usuwanie węzła

- aby ułatwić sobie obsługę warunków brzegowych, można puste wskaźniki z liści przekierować na dodatkowy, czarny węzeł
- przed dostępem do rodzica takiego węzła należy ustawić go na odpowiednią wartość

Usuwanie węzła

`Usun`(T, w)

1: `UsunBST`(T, w) $\rightarrow$, niech w wskazuje na potomka usuniętego węzła
2: **if** usunięty węzeł był czarny **then**
3: `Napraw`(T, w)
4: **end if**

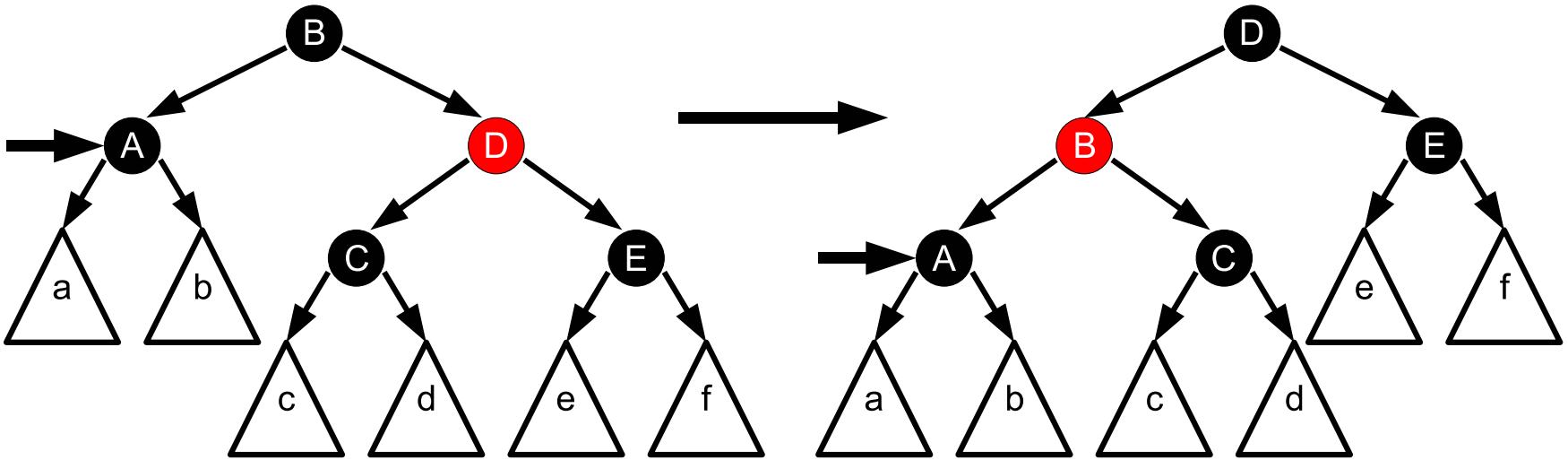
Naprawa drzewa

- traktujemy potomka usuniętego węzła tak, jakby był "podwójnie" czarny (otrzymał "czarną jednostkę" od usuniętego rodzica) — wtedy właściwość drzewa jest zachowana
- przesuwamy "czarną jednostkę" w górę drzewa
- jeżeli dotrzemy z nią do czerwonego węzła — przekolorowujemy go na czarno
- jeżeli dojdziemy do korzenia zapominamy o niej ("wypada" z drzewa)
- wykonujemy odpowiednie rotacje, aby zachować właściwość drzewa

Naprawa węzła

- przypadek 1: węzeł ma czerwonego brata
- przypadek 1 sprowadzamy do przypadku 2, 3 lub 4, obracając rodzica i zamieniając jego kolor z bratem

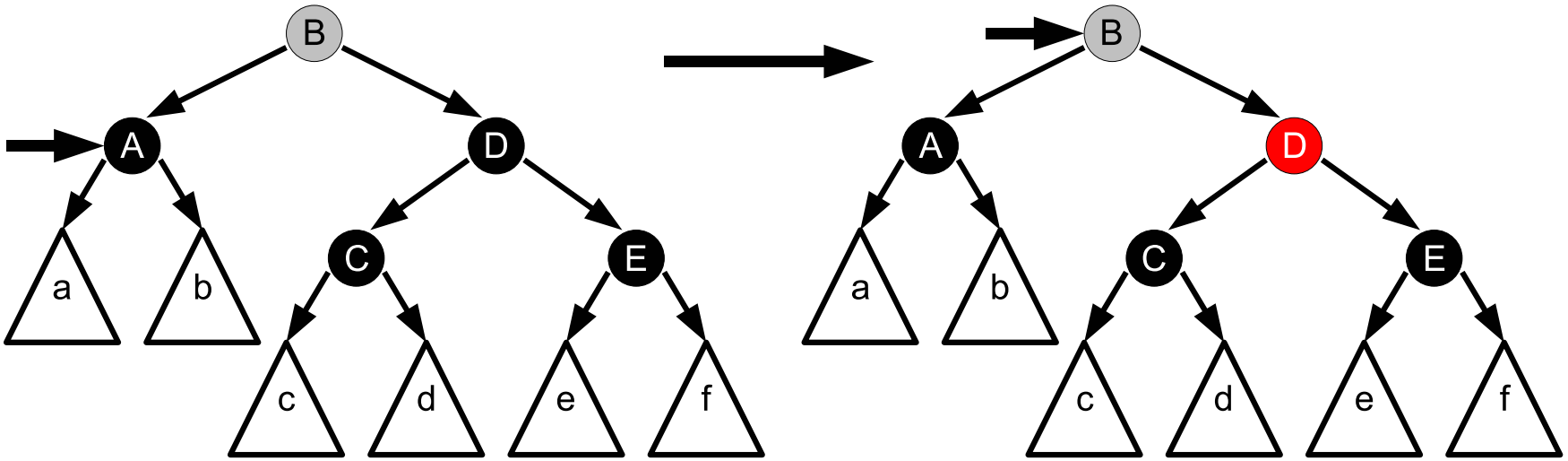
Przypadek 1



Naprawa węzła

- przypadek 2: brat jest czarny i ma dwóch czarnych potomków
- z węzła i jego brata wyciągamy po jednej “czarnej jednostce” (z węzła usuwamy “wirtualną”, brata przekolorowujemy na czerwono)
- rodzic otrzymuje nadmiarową “czarną jednostkę” i kontynuujemy poprawianie w górę drzewa

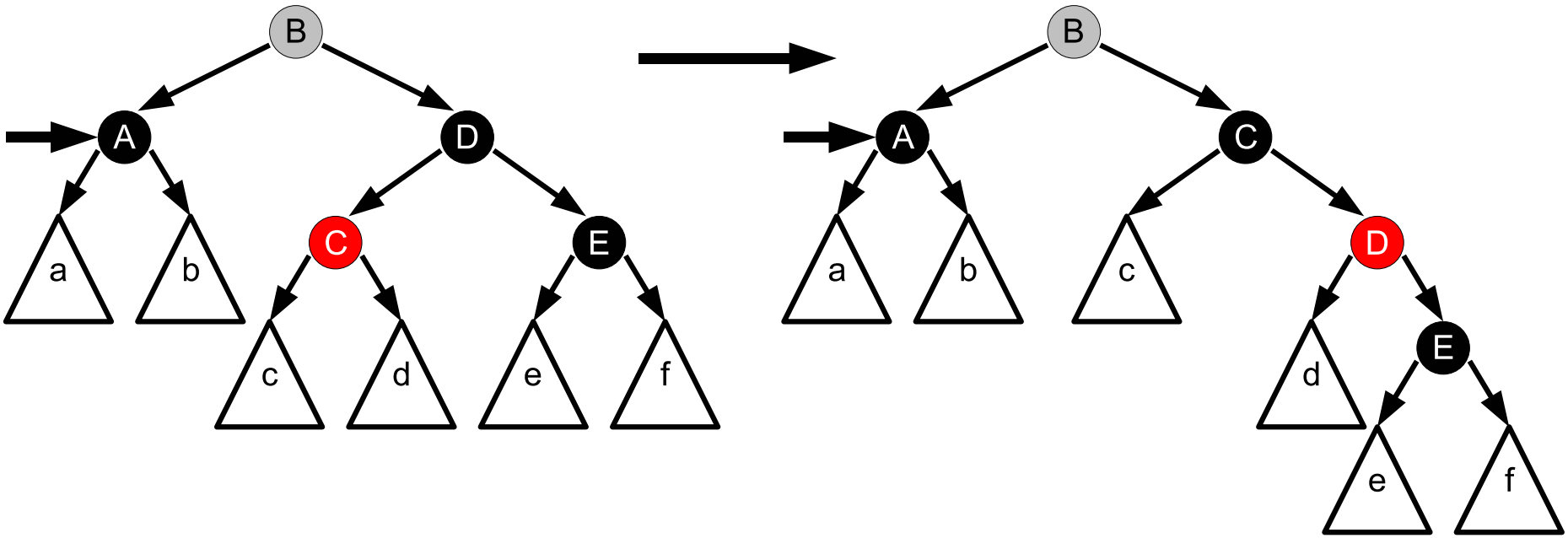
Przypadek 2



Naprawa węzła

- przypadek 3: brat jest czarny, jego lewy potomek jest czerwony, prawy jest czarny
- obracamy brata w prawo, i zamieniamy kolorami "starego" brata z nowym
- sprowadzamy ten przypadek do przypadku 4

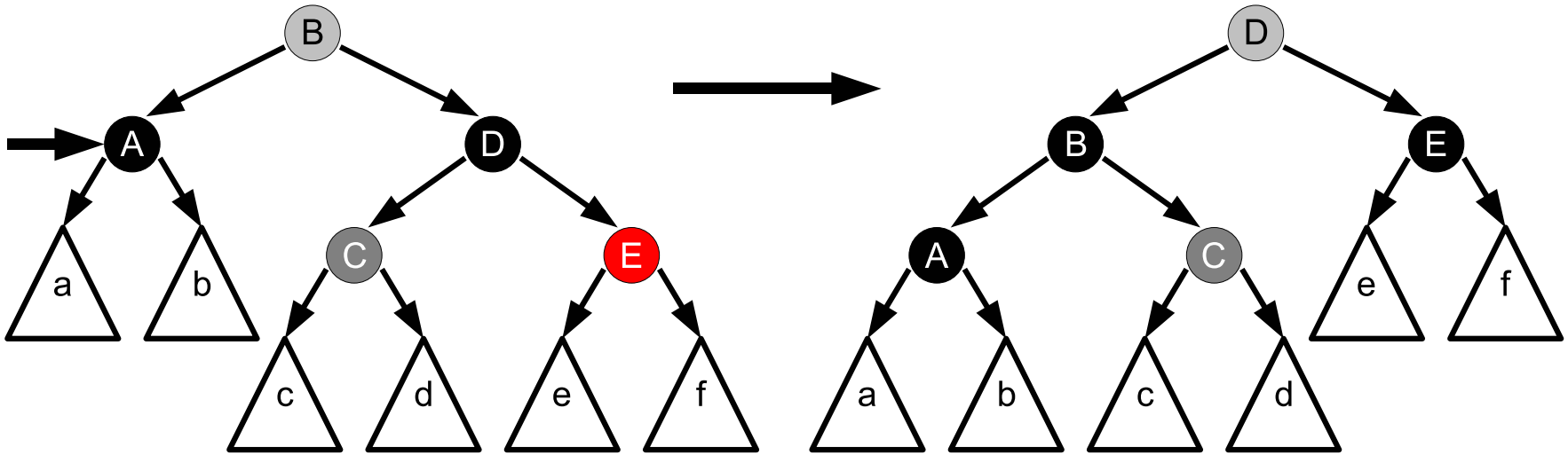
Przypadek 3



Naprawa węzła

- przypadek 4: brat jest czarny, jego prawy potomek jest czerwony
- wykonujemy odpowiednie obroty, aby móc przekolorować czerwony węzeł na czarno
- przerywamy poprawianie (wstawiliśmy nadmiarową czarną jednostkę w rzeczywiste miejsce)

Przypadek 4



Naprawa węzła

```
Napraw(T, w)
 1: while w ≠ T.root and w.color = BLACK do
 2:         if w = x.parent.left then
 3:                 b = w.parent.left
 4:                 if b.color = RED then {Przypadek 1}
 5:                         b.color = BLACK
 6:                         w.parent.color = RED
 7:                         ObrotWLewo(T, w.parent)
 8:                         b = w.parent.right
 9:                 end if
10:                 if b.left.color = BLACK and b.right.color = BLACK then {Przypadek 2}
11:                         b.color = RED
12:                         w = w.parent
13:                 else
14:                         if b.right.color = BLACK then {Prz. 3}
15:                                 b.left.color = BLACK
16:                                 b.color = RED
17:                                 ObrotWPrawo(T, b)
18:                                 b = w.parent.right
19:                         end if
20:                         b.color = w.parent.color {Przypadek 4}
21:                         w.parent.color = BLACK
22:                         b.right.color = BLACK
23:                         ObrotWLewo(T, w.parent)
24:                         w = T.root
25:                 end if
26:         else
27:                 analogicznie jak wyżej, zamieniając rolami left i right
28:         end if
29: end while
30: w.color = BLACK
```

Usuwanie węzła

- złożoność procedury: $O(\log n)$
- każde usunięcie wykona co najwyżej trzy rotacje
  - przypadek 1 wykonuje 1 rotację i przechodzi albo do przypadku 2 (z sytuacją, która przerwie pętlę), albo do przypadku 3 lub 4
  - przypadek 2 nie wykonuje rotacji
  - przypadki 3 i 4 wykonają co najwyżej 2 rotacje i przerywają pętlę

Porównanie AVL i RBT

- wysokość drzewa AVL $\leq 1.4404 \log(n+2) - 0.328$
- usuwanie węzła z AVL wymaga $O(\log n)$rotacji
- wysokość RBT $\leq 2 \log(n+1)$
- usuwanie węzła z RBT wymaga co najwyżej 3 rotacji
- drzewo AVL lepiej nadaje się do statycznych struktur (częste wyszukiwania, rzadkie modyfikacje)
- RBT lepiej nadaje się do struktur dynamicznych (rzadkie wyszukiwania, częste modyfikacje)

B-drzewa

- w każdym węźle (poza korzeniem) może znajdować się od $t-1$ do $2t-1$ kluczy (t – minimalny stopień drzewa)
- każdy węzeł jest albo liściem, albo zawiera $m+1$ potomków, gdzie m to liczba kluczy w węźle
- węzły wiedzą, czy są liśćmi (dodatkowe pole)
- wszystkie liście występują na tym samym poziomie
- dostęp do elementu wymaga $O(\log n)$czasu, ale t określa podstawę logarytmu
- B-drzewa są uogólnieniem drzew poszukiwań binarnych

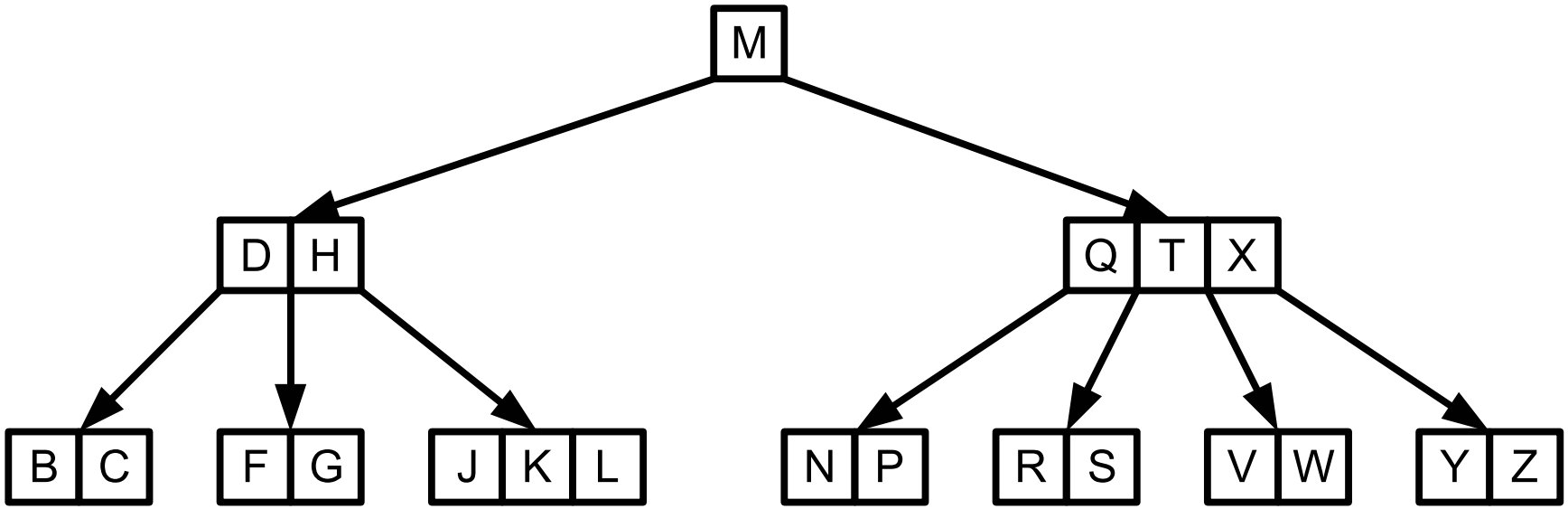
B-drzewa

- szczególnie efektywne w przypadku przechowywania drzewa na dysku, gdzie odczyt/zapis trwa bardzo długo (dłużej niż operacja $O(n)$ w pamięci)
- chcemy zminimalizować liczbę dostępów do dysku
- węzeł = strona/sektor/klaster
- np. B-drzewo o minimalnym stopniu 500 zawierające 1000000000 kluczy będzie miało wysokość 2 (dotarcie do konkretnego klucza wymaga tylko 3 odczytów z pamięci zewnętrznej)
- dodatkowo korzeń może być cały czas przechowywany w pamięci — zmniejsza to o 1 liczbę koniecznych odczytów (modyfikacja korzenia wymaga zapisania go na dysk)

B-drzewa

- węzeł z m kluczami posiada $m + 1$ potomków
- pierwszy potomek zawiera klucze mniejsze od pierwszego klucza w węźle
- drugi potomek zawiera klucze zawierające się między pierwszym a drugim kluczem
- ...
- ostatni potomek zawiera klucze większe od ostatniego klucza w węźle

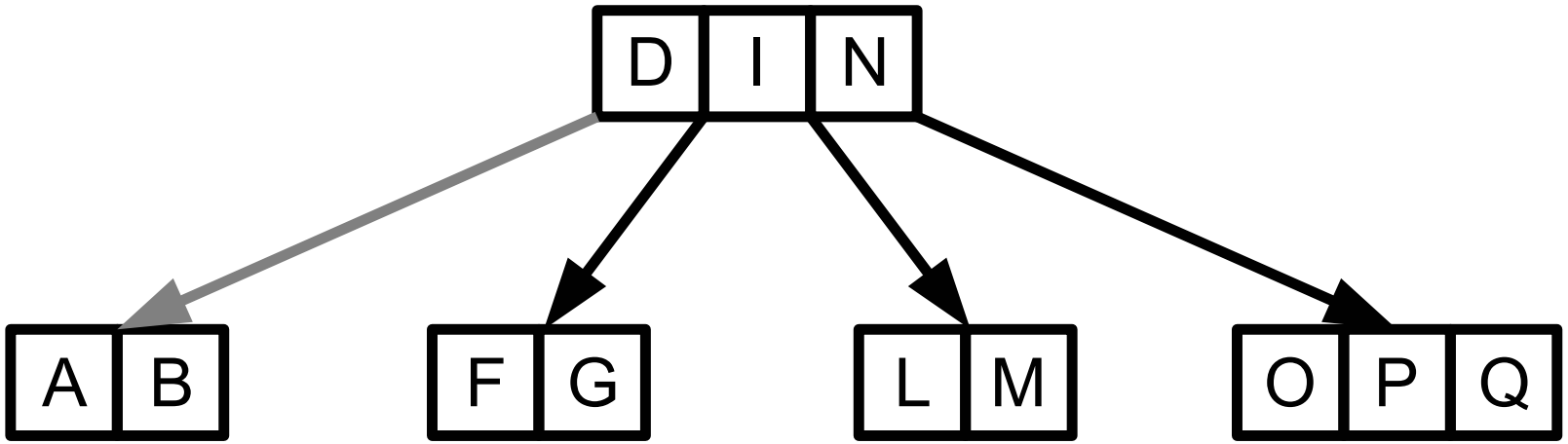
B-drzewo



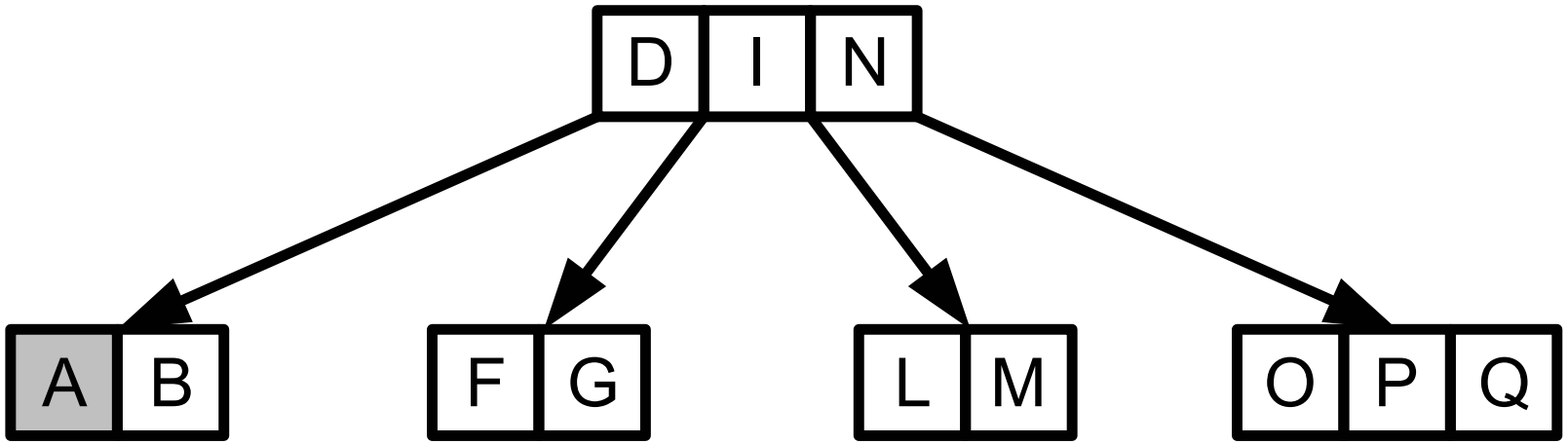
Przeglądanie B-drzewa

```
Wypisz(w)
1: for i = 1 to w.m do
2:         if not w.leaf then Wypisz(w.son[i])
3:         print w.key[i]
4: end for
5: Wypisz(w.son[w.m + 1])
```

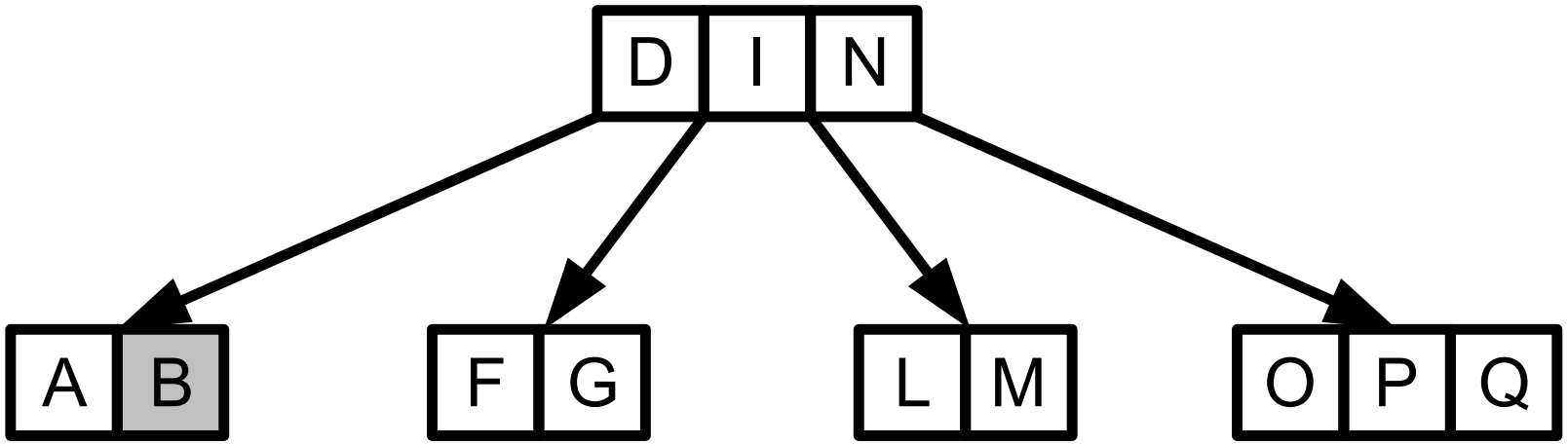
Przeglądanie B-drzewa



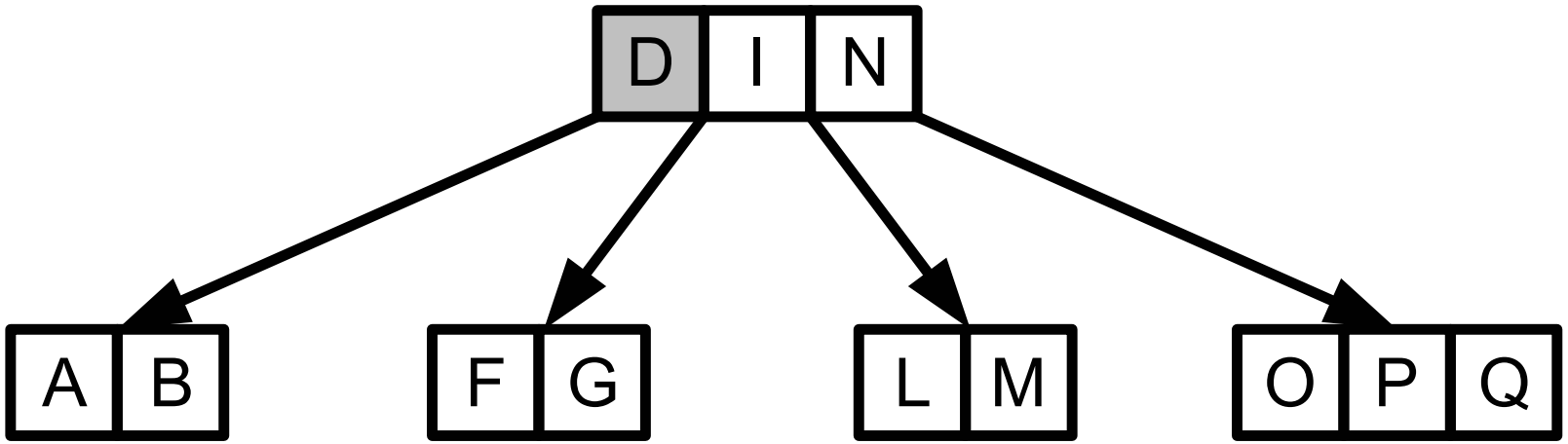
Przeglądanie B-drzewa



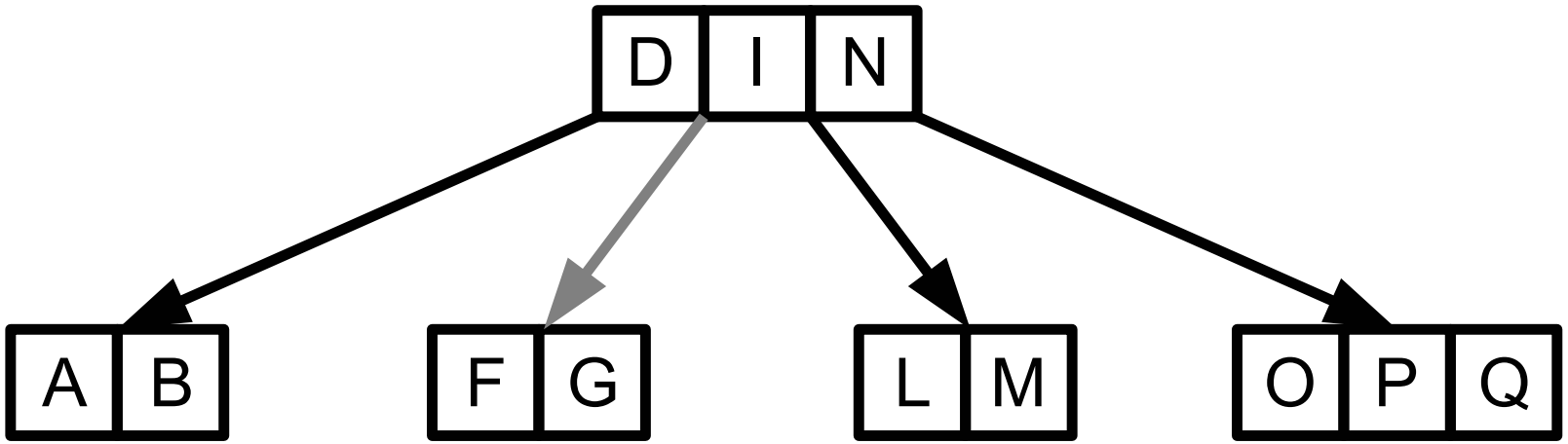
Przeglądanie B-drzewa



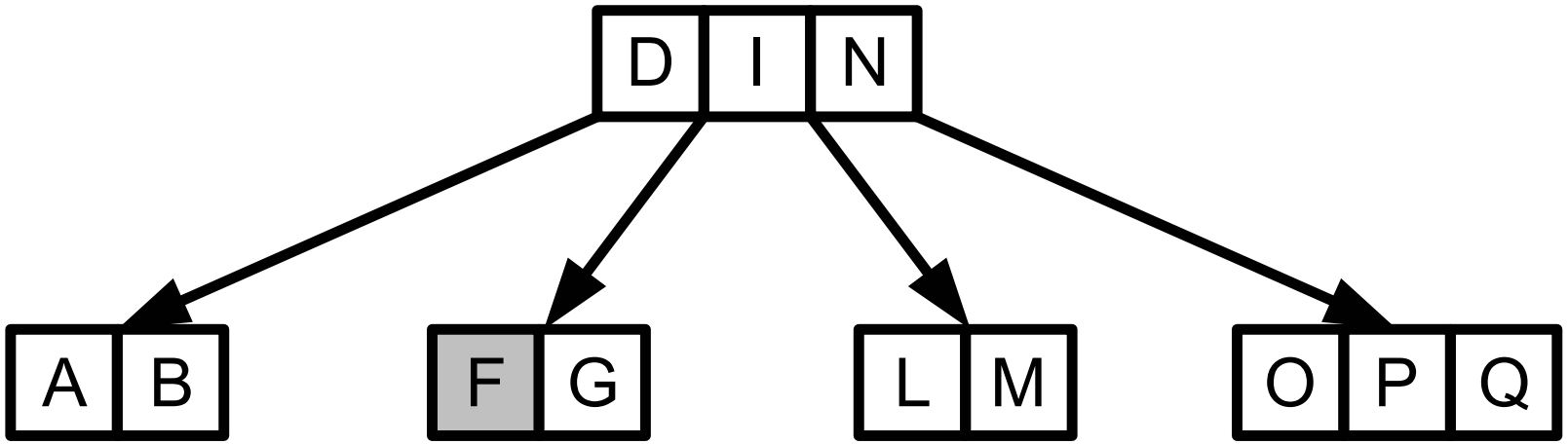
Przeglądanie B-drzewa



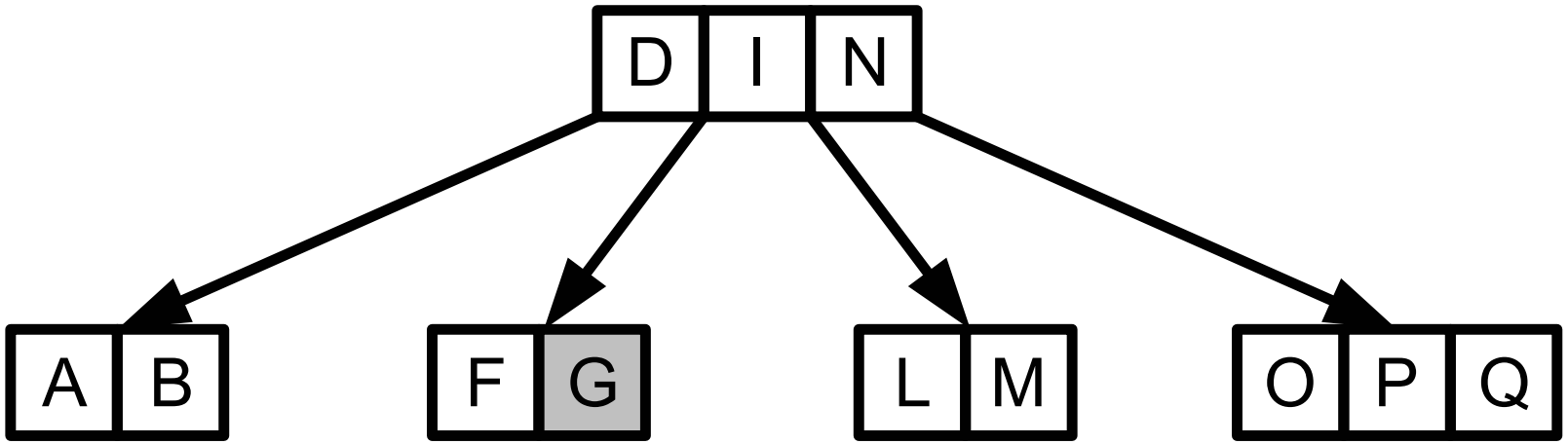
Przeglądanie B-drzewa



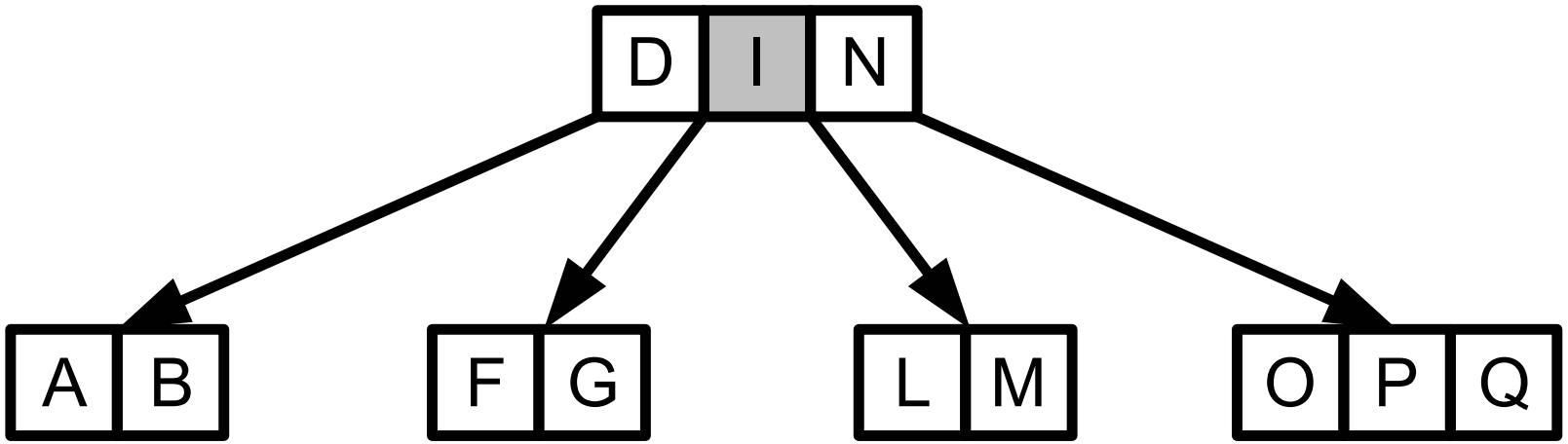
Przeglądanie B-drzewa



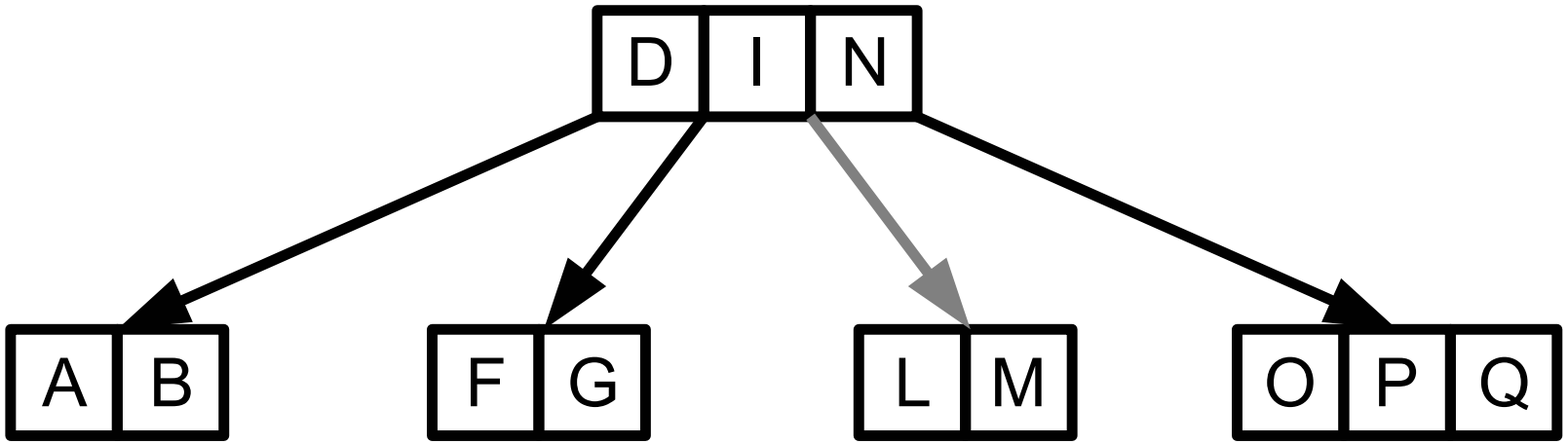
Przeglądanie B-drzewa



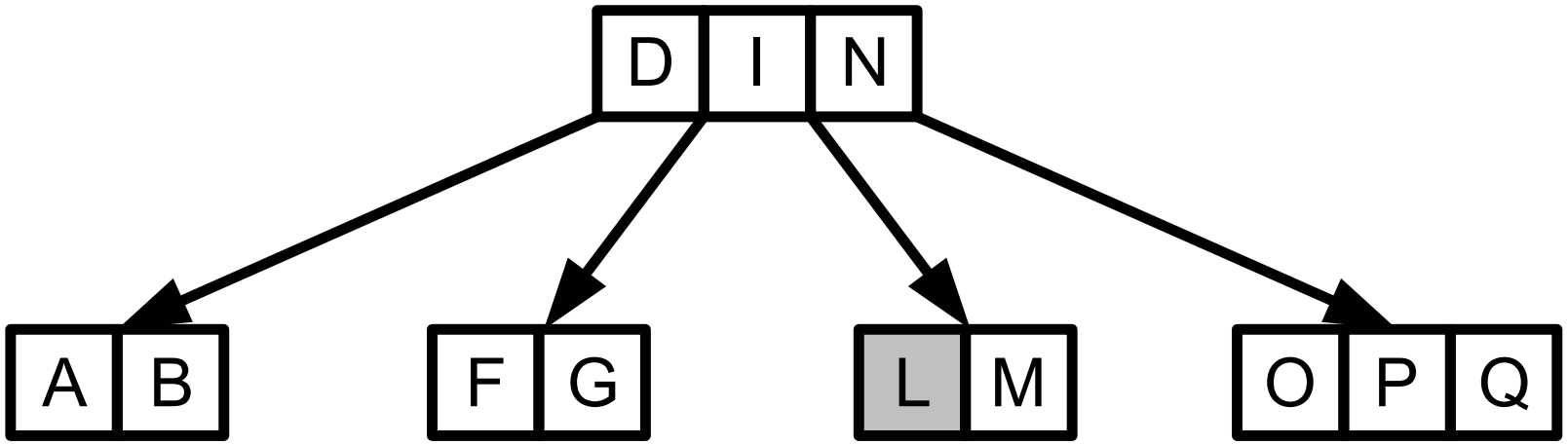
Przeglądanie B-drzewa



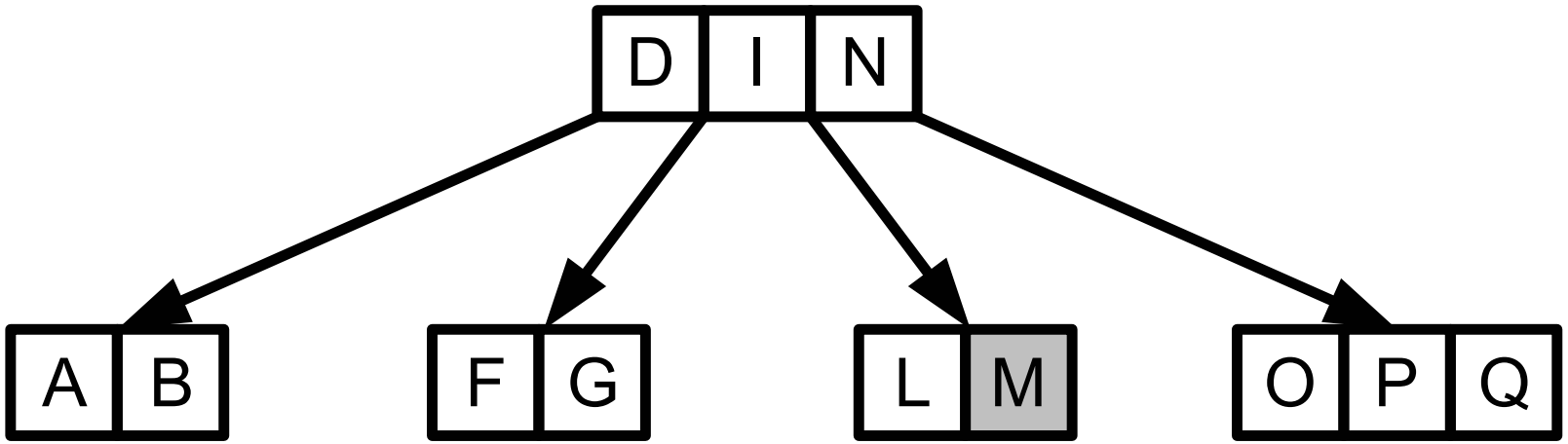
Przeglądanie B-drzewa



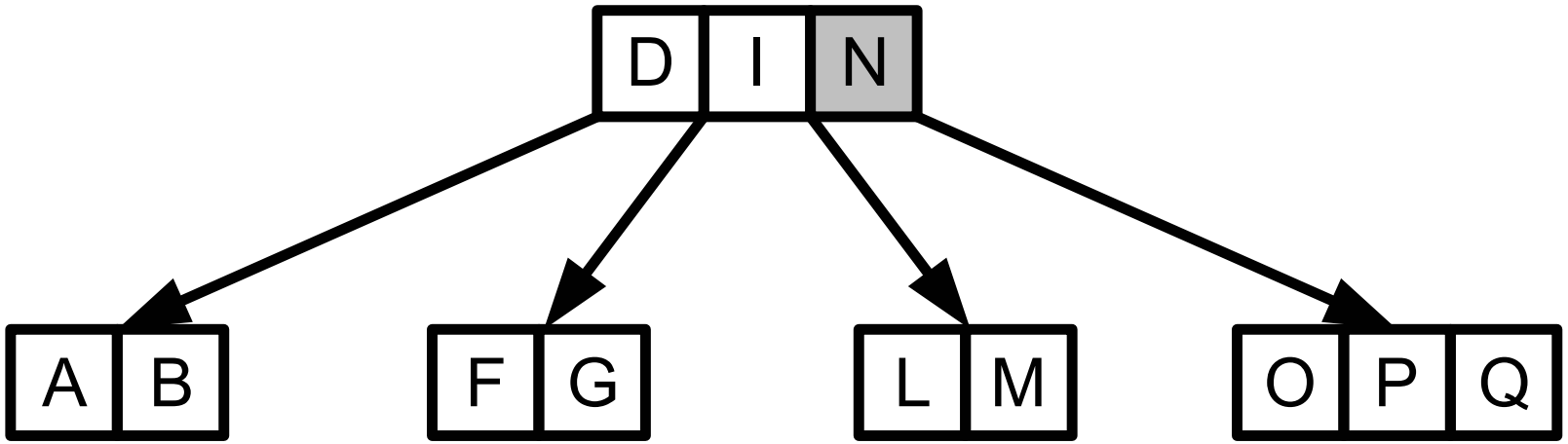
Przeglądanie B-drzewa



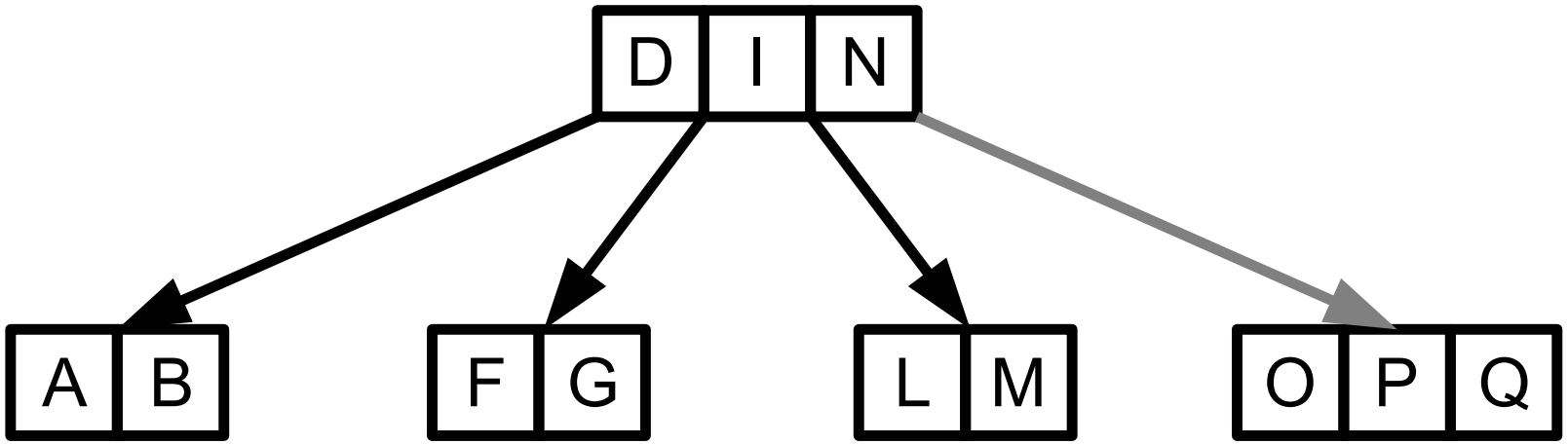
Przeglądanie B-drzewa



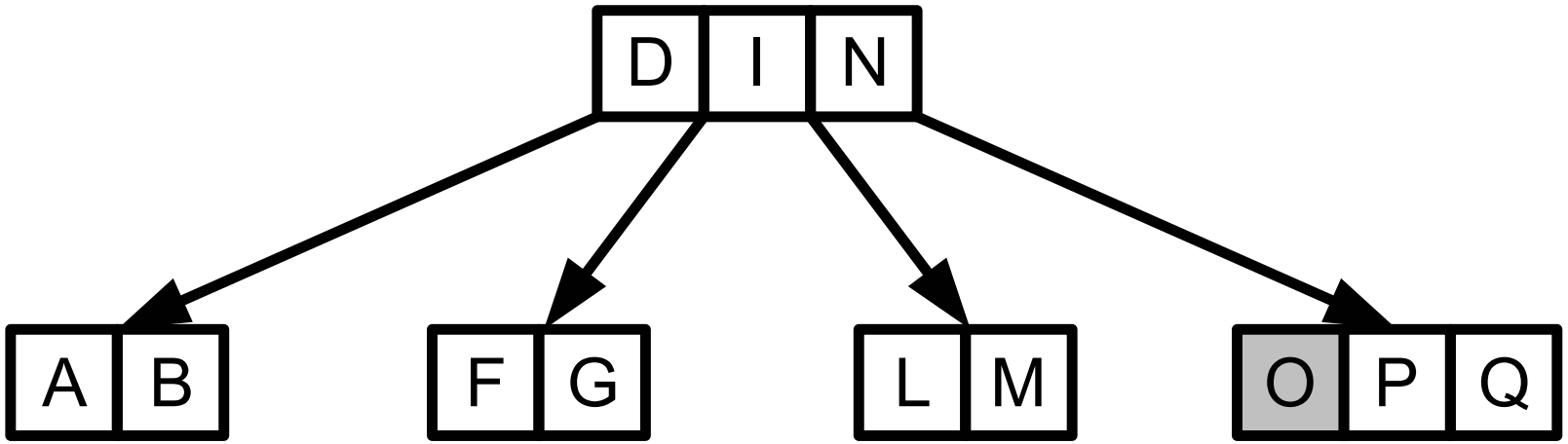
Przeglądanie B-drzewa



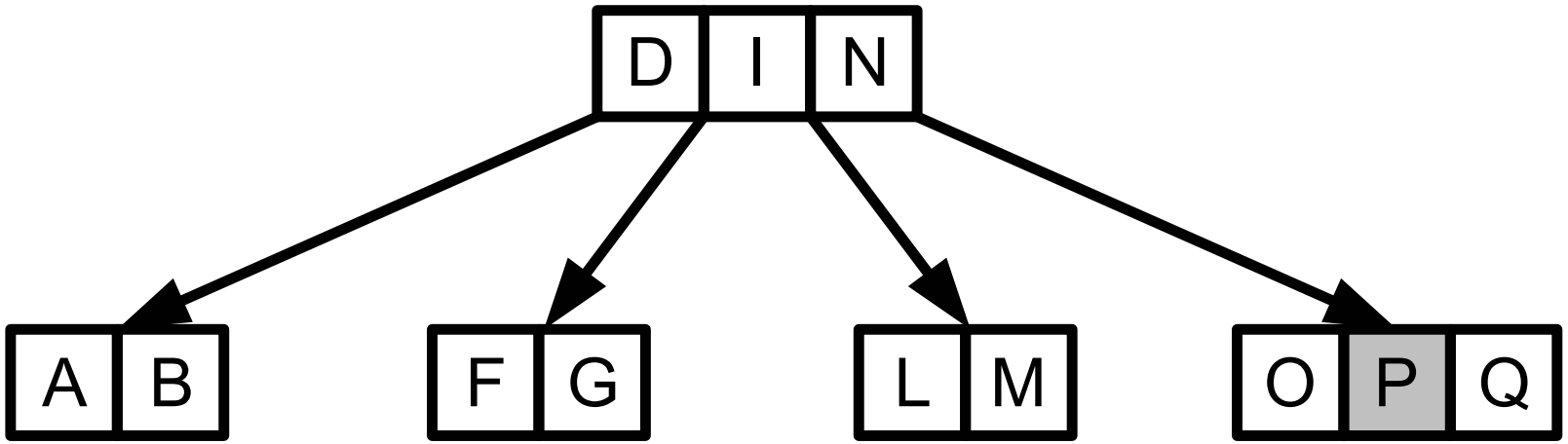
Przeglądanie B-drzewa



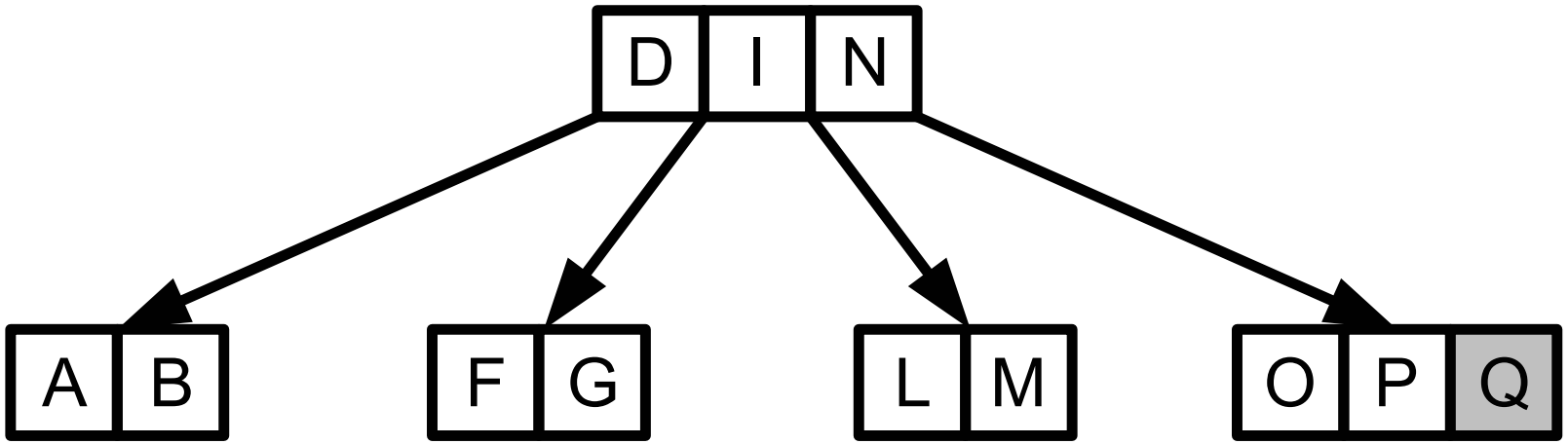
Przeglądanie B-drzewa



Przeglądanie B-drzewa



Przeglądanie B-drzewa



Wyszukiwanie w B-drzewie

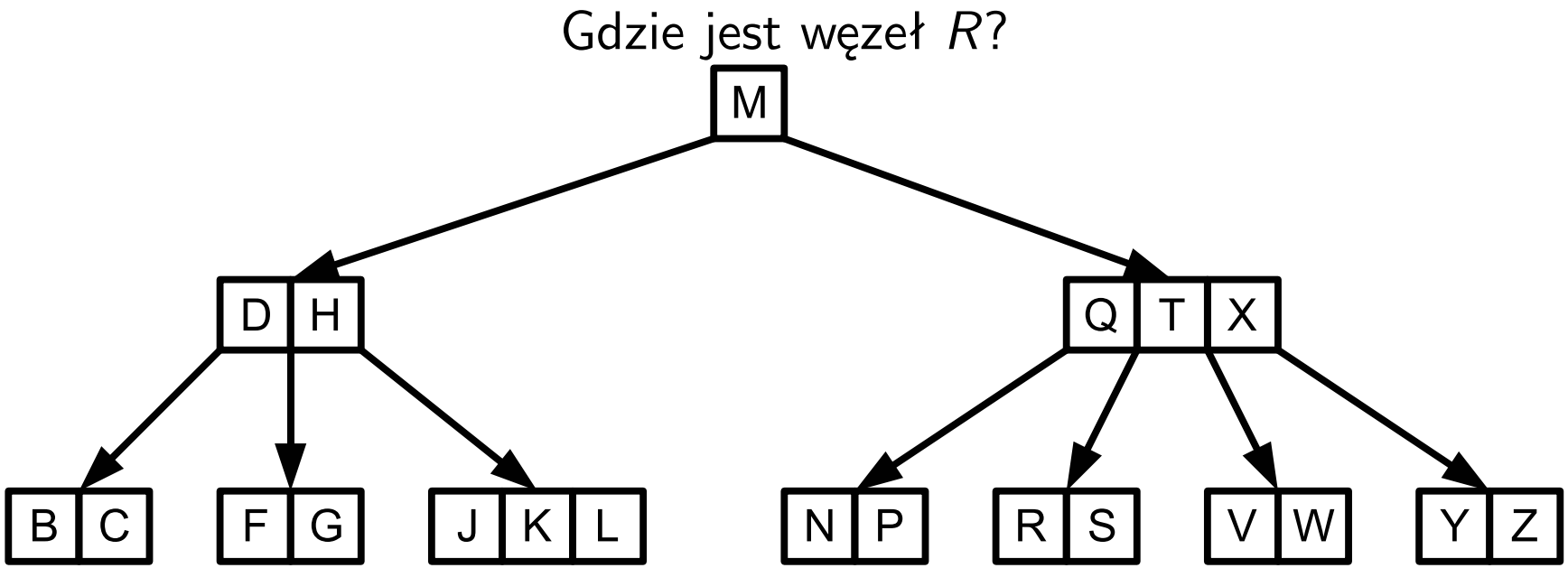
- poszukując danego klucza, zaczynamy od korzenia
- przeglądamy klucze w bieżącym węźle (liniowo lub binarnie)
- klucz będzie znajdował się albo w bieżącym węźle, albo w i-tym potomku, gdzie $w.key[i-1] < klucz < w.key[i]$
- w tym wypadku wczytujemy do pamięci operacyjnej i-tego potomka, i w nim kontynuujemy poszukiwanie
- wymagany czas to $O(t \log_t n)$

Wyszukiwanie w B-drzewie

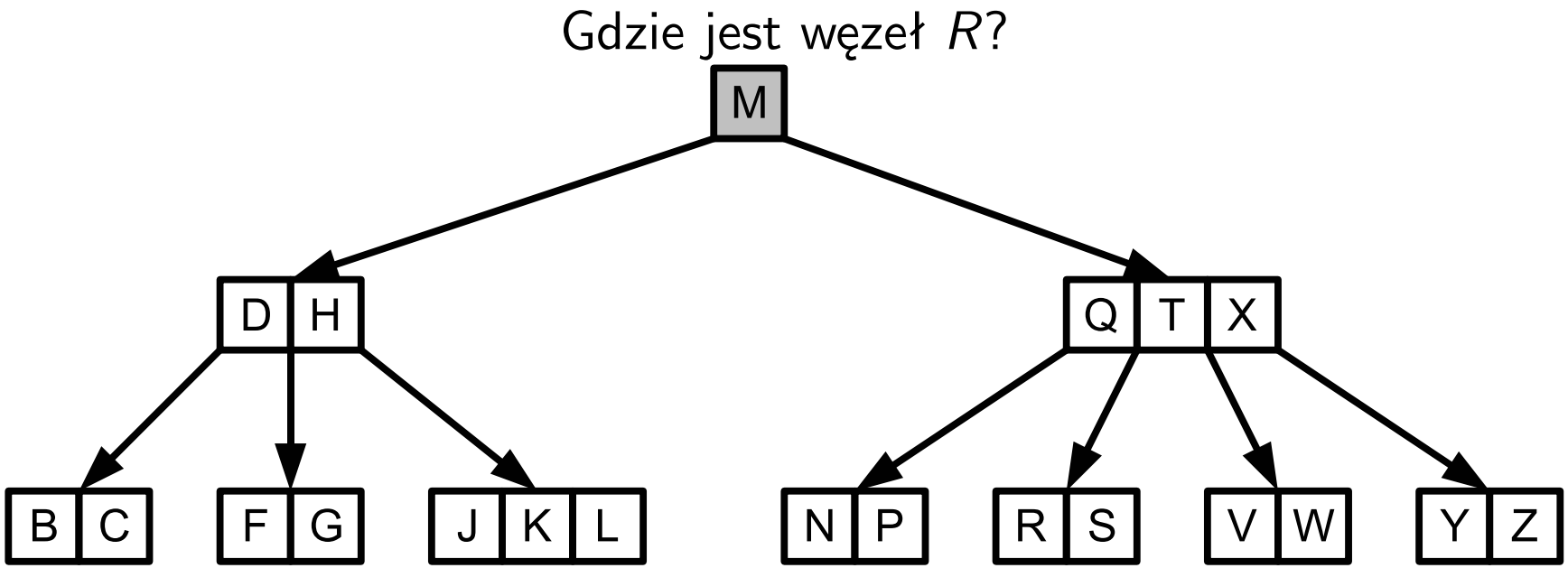
`Znajdz`(w, *klucz*)

```
 1: i = 1
 2: while i ≤ w.m and k > w.key[i] do
 3:         i = i + 1
 4: end while
 5: if i ≤ w.m and k = w.key[i] then
 6:         return (w, i)
 7: end if
 8: if w.leaf then
 9:         return nullptr
10: else
11:         wczytaj do pamięci węzeł w.son[i]
12:         return Znajdz(w.son[i], klucz)
13: end if
```

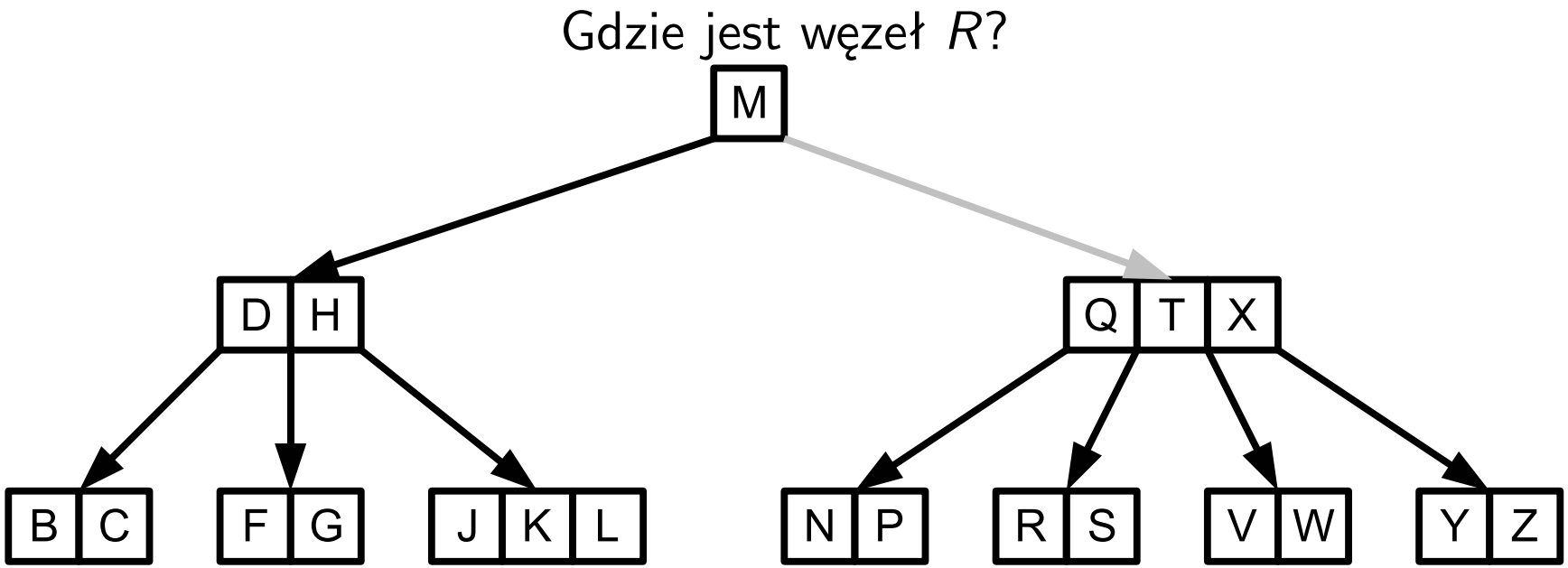
Wyszukiwanie w B-drzewie



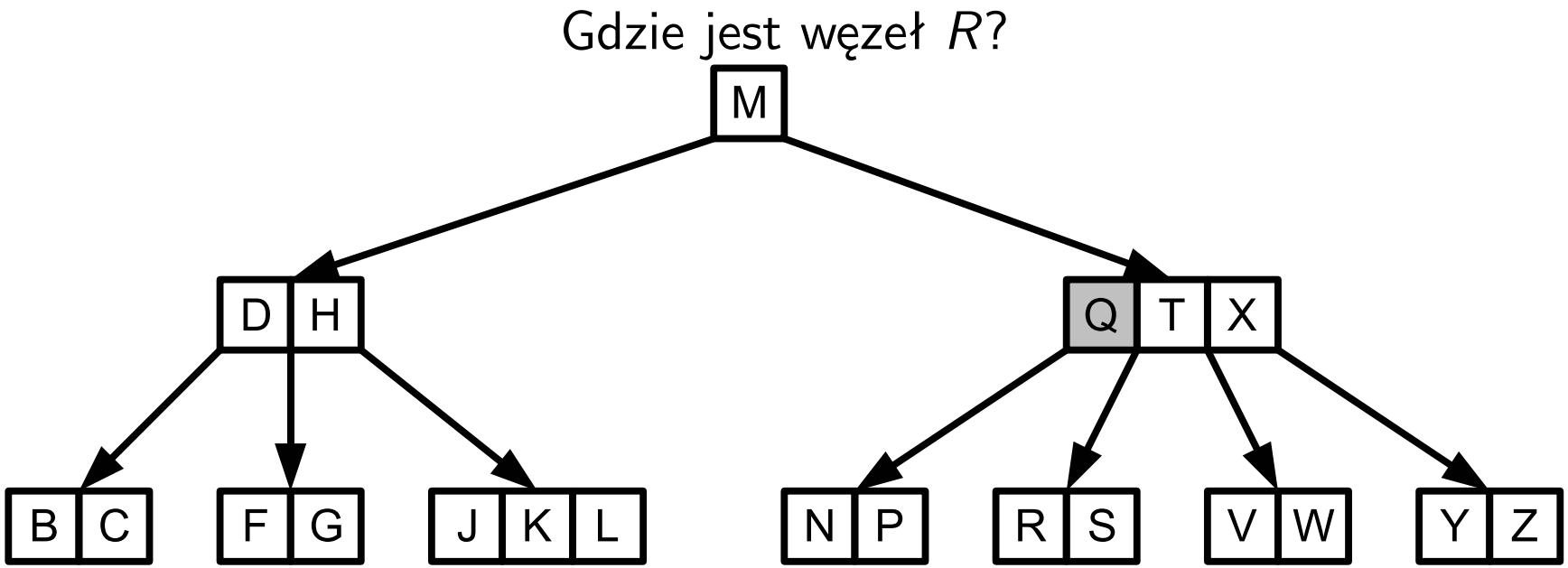
Wyszukiwanie w B-drzewie



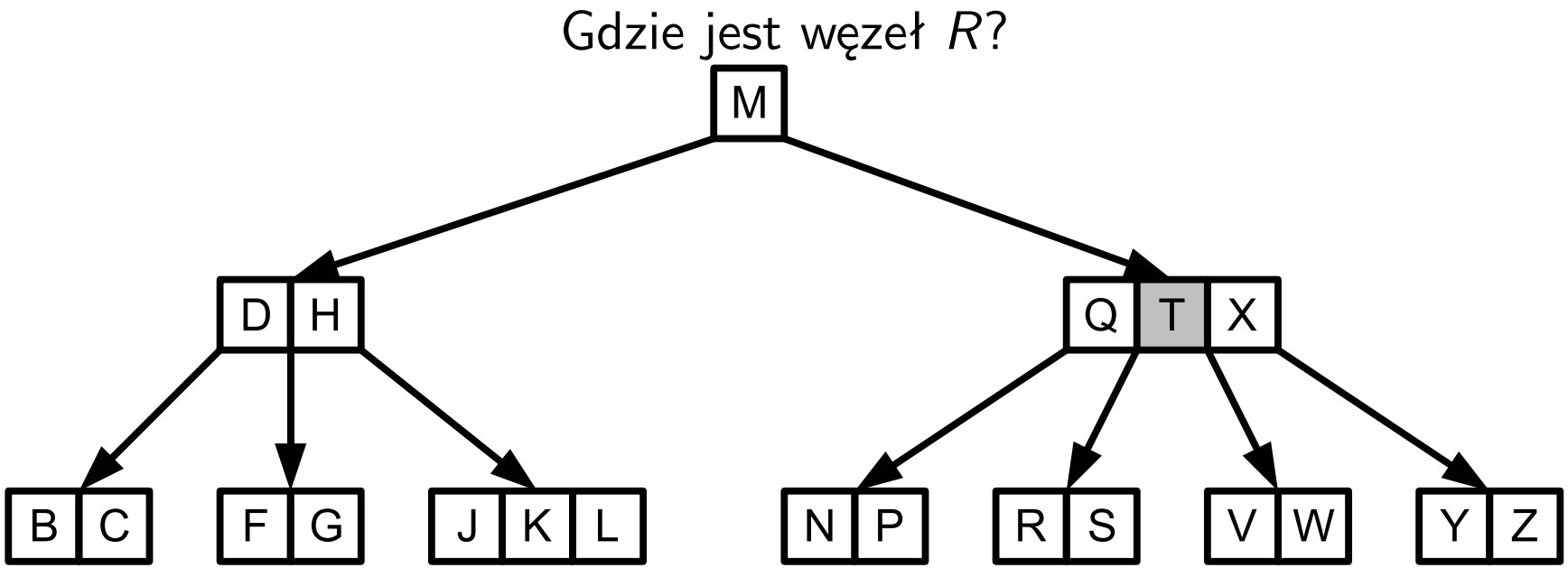
Wyszukiwanie w B-drzewie



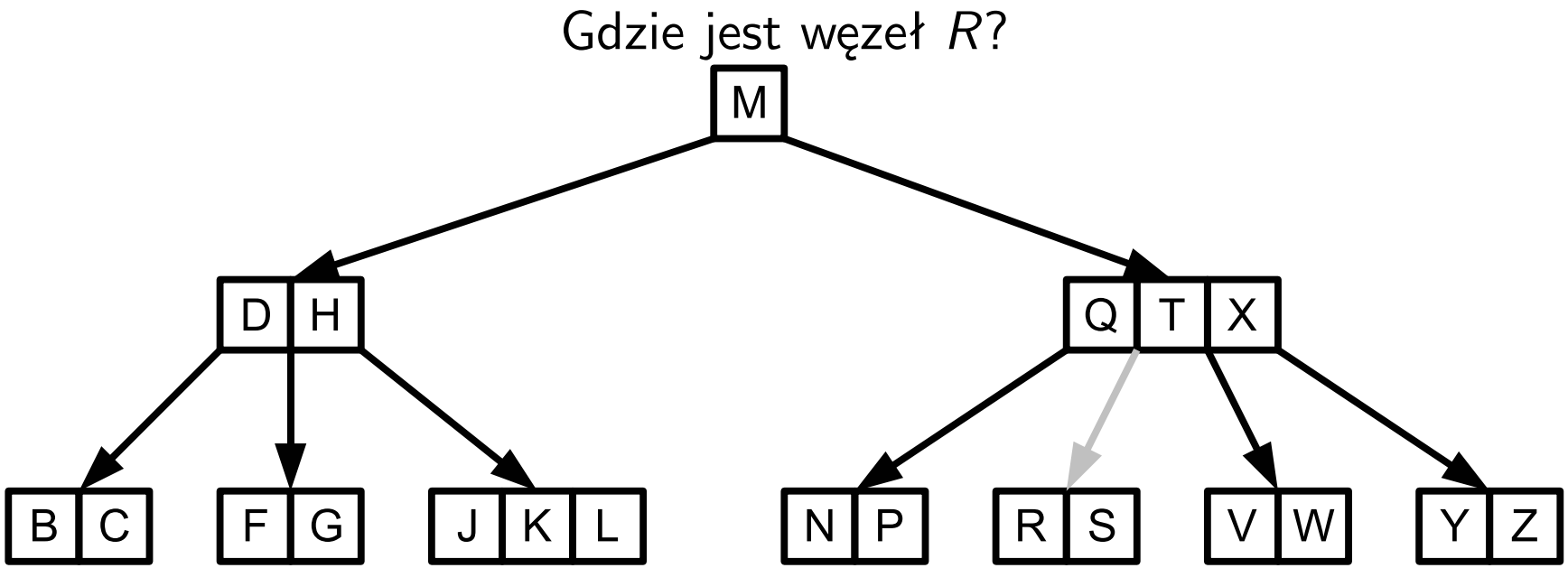
Wyszukiwanie w B-drzewie



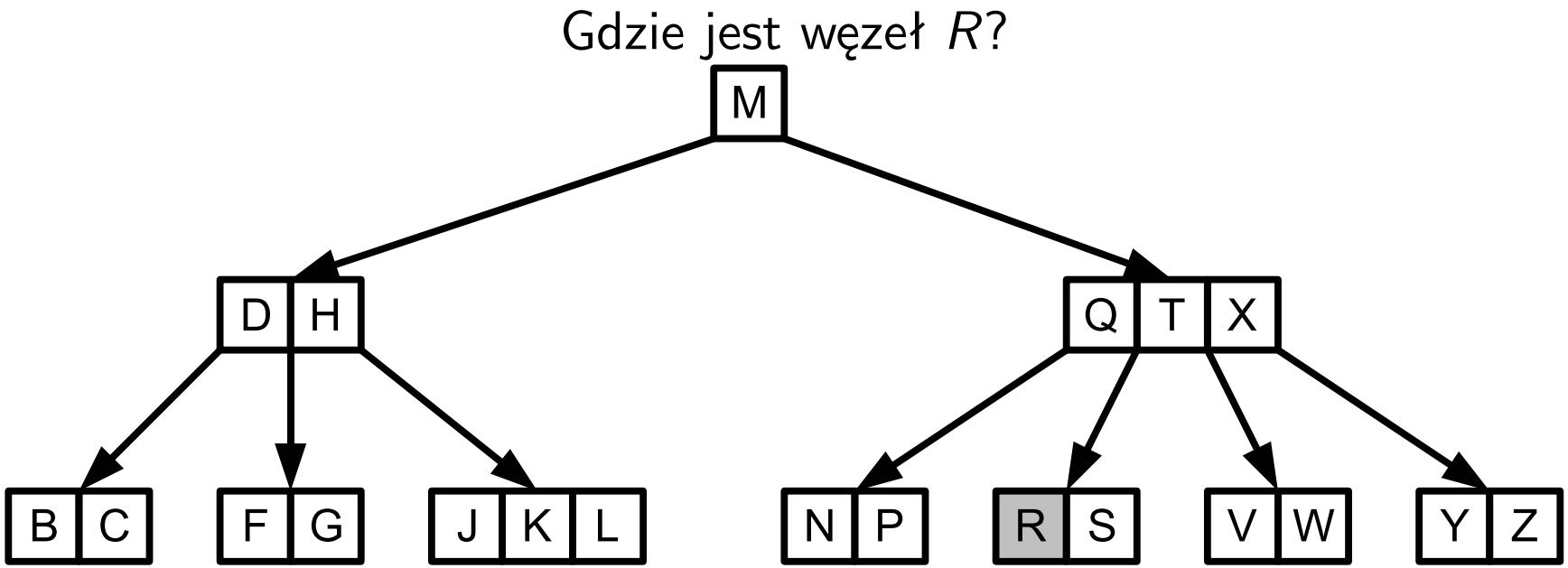
Wyszukiwanie w B-drzewie



Wyszukiwanie w B-drzewie



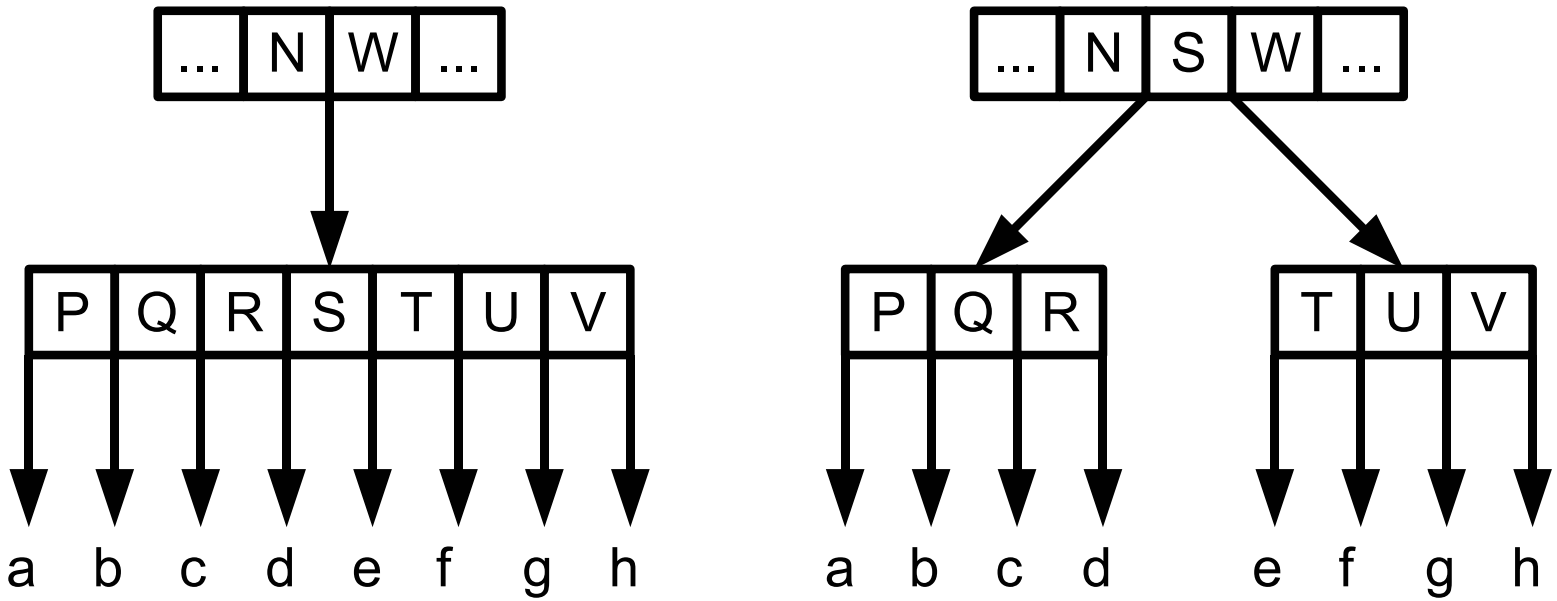
Wyszukiwanie w B-drzewie



Rozbijanie węzła

- wstawianie klucza do B-drzewa wykorzystuje operację rozbicia węzła
- rozbijany będzie pełny węzeł (zawierający $2t - 1$) węzłów
- rezultatem są dwa nowe węzły
  - pierwszy zawiera pierwsze $t - 1$ kluczy (i t potomków) węzła początkowego
  - drugi zawiera ostatnie $t - 1$ kluczy (i t potomków) węzła początkowego
- środkowy klucz (t-ty) zostaje wstawiony do rodzica węzła (zakładamy, że rodzic nie jest pełny)
- jeżeli węzeł nie ma rodzica (jest korzeniem) to tworzymy nowy korzeń zawierający jeden klucz, i dwoje potomków

Rozbijanie węzła



Rozbijanie węzła

`Rozbij`($parent$, i, w) (i to numer potomka w w $parent$)

1: $z =$ `NewNode()`
2: $z.leaf = w.leaf$
3: $z.m = t - 1$
4: $z.key[j] = w.key[j + t]$ dla $j = 1, \ldots t - 1$
5: **if not** $w.leaf$ **then** $z.son[j] = w.son[j + t]$ dla $j = 1, \ldots t - 1$
6: $w.m = t - 1$
7: **for** $j = parent.m + 1$ **down to** i **do**
8: $\quad parent.key[j + 1] = parent.key[j]$
9: $\quad parent.son[j + 1] = parent.son[j]$
10: **end for**
11: $parent.key[i] = w.key[t]$
12: $parent.m = parent.m + 1$
13: zapisz na dysk węzły w, z i $parent$

Wstawianie do B-drzewa

- wykonujemy tylko jedno zejście od korzenia do liścia
- jeżeli po drodze natrafimy na pełen węzeł, rozbijamy go
- wymagany czas to $O(t \log_t n)$

Wstawianie do B-drzewa

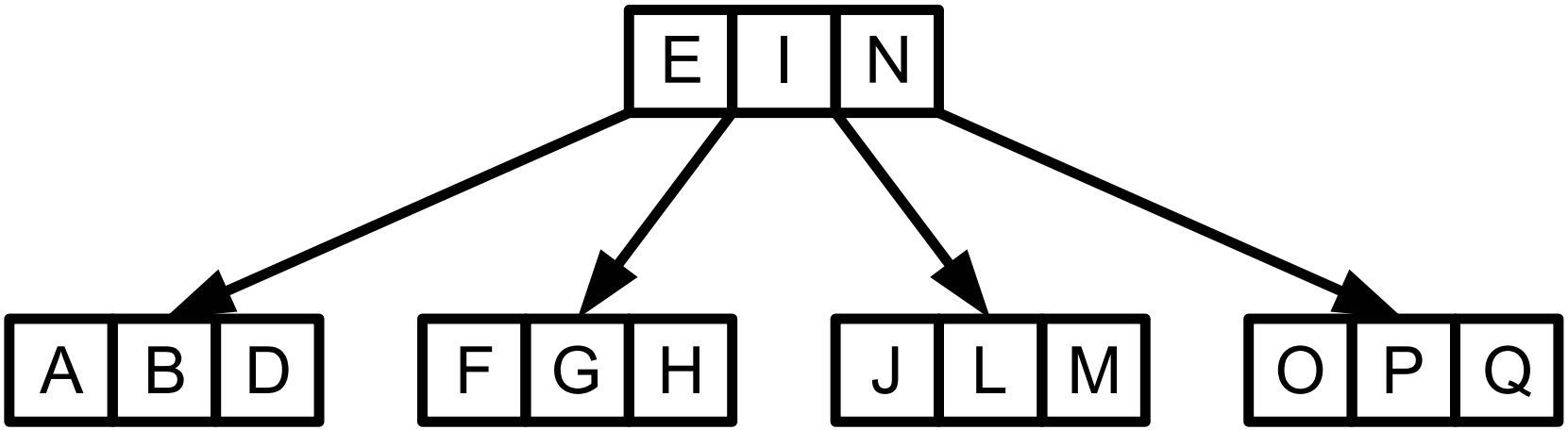
```
Wstaw(T, klucz)
 1: r = T.root
 2: if r.m = 2t − 1 then
 3:         s = NewNode()
 4:         T.root = s
 5:         s.leaf = FALSE
 6:         s.m = 0
 7:         s.son[1] = r
 8:         Rozbij(s, 1, r)
 9:         WstawNiePelny(s, klucz)
10: else
11:         WstawNiePelny(r, klucz)
12: end if
```

Wstawianie do B-drzewa

```
WstawNiePelny(w, klucz)
 1: i = w.m
 2: if w.leaf then
 3:         while i ≥ 1 and klucz < w.key[i] do
 4:                 w.key[i + 1] = w.key[i]
 5:                 i = i − 1
 6:         end while
 7:         w.key[i + 1] = klucz
 8:         w.m = w.m + 1
 9:         zapisz na dysk węzeł w
10: else
11:         while i ≥ 1 and klucz < w.key[i] do i = i − 1
12:         i = i + 1
13:         wczytaj do pamięci węzeł w.son[i]
14:         if w.son[i].m = 2t − 1 then
15:                 Rozbij(w, i, w.son[i])
16:                 if klucz > w.key[i] then i = i + 1
17:         end if
18:         WstawNiePelny(w.son[i], klucz)
19: end if
```

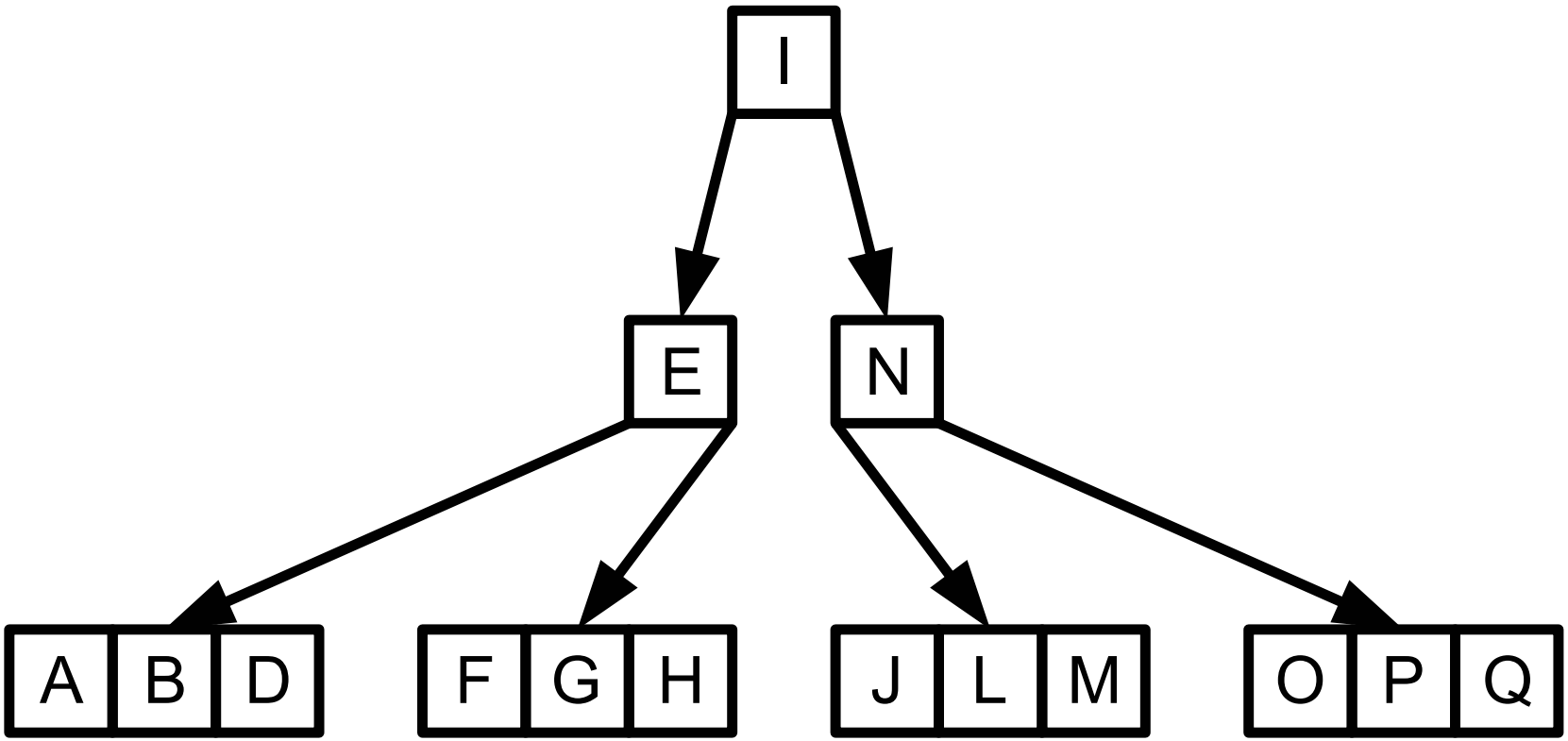
Wstawianie do B-drzewa

Wstawiamy C, $t = 2$



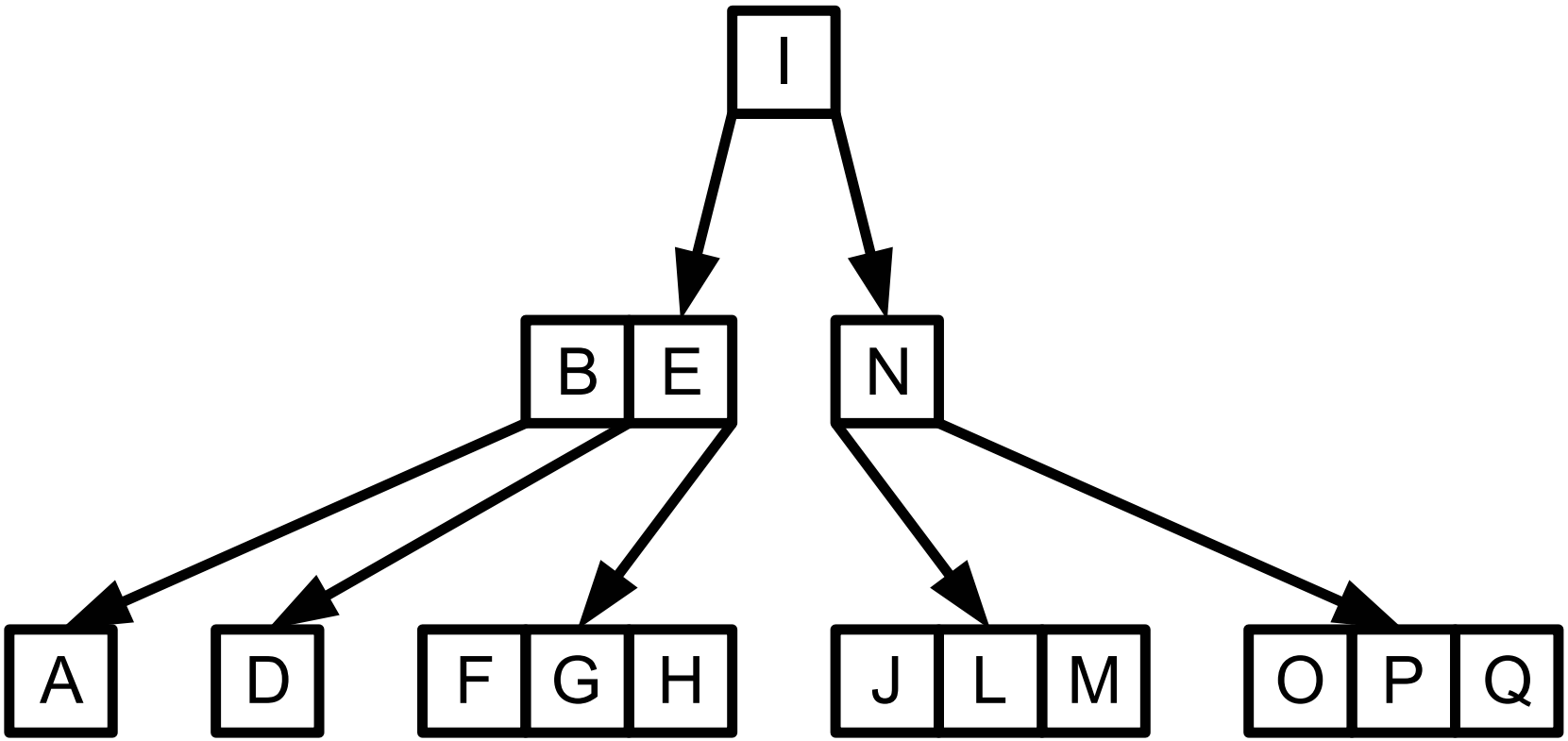
Wstawianie do B-drzewa

Wstawiamy C, $t = 2$



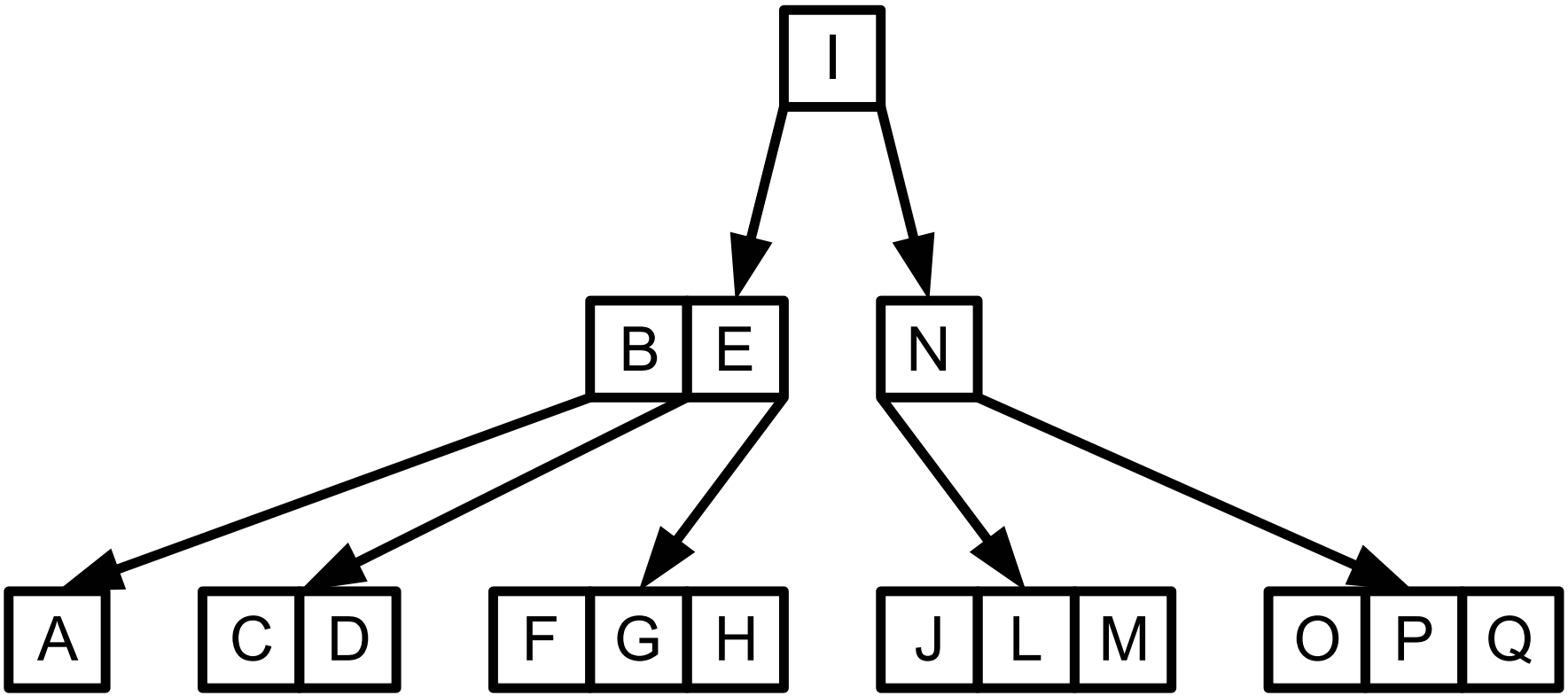
Wstawianie do B-drzewa

Wstawiamy C, $t = 2$



Wstawianie do B-drzewa

Wstawiamy C, $t = 2$



Usuwanie z B-drzewa

- w większości przypadków usunięcie klucza z drzewa wymaga tylko pojedynczego zejścia w dół drzewa
- może jednak się zażyć, że wymagany będzie powrót w górę drzewa
- wymagany czas to $O(t \log_t n)$
- niezmiennik: w wywołaniu usunięcia dla węzła różnego od korzenia, węzeł ten ma zawsze co najmniej t kluczy

Usuwanie z B-drzewa

- jeżeli klucz k jest w węźle w i w jest liściem, usuń klucz k z w
- jeżeli klucz k jest w węźle w i w jest węzłem wewnętrznym:
  - niech u będzie synem w poprzedzającym k; jeżeli u ma co najmniej t kluczy, to w poddrzewie o korzeniu w u wyznacz poprzednik k' dla klucza k; rekurencyjnie usuń k' i zastąp k przez k' w węźle w
  - symetrycznie, niech v będzie synem w następującym po k; jeżeli v ma co najmniej t kluczy, to w poddrzewie o korzeniu w v wyznacz następnik k' dla klucza k; rekurencyjnie usuń k' i zastąp k przez k' w węźle w

Usuwanie z B-drzewa

- jeżeli klucz k jest w węźle w i w jest węzłem wewnętrznym (c.d.):
  - jeżeli u i v mają tylko po $t-1$ kluczy, to przenieś k i całą zawartość węzła v do węzła u — k i wskaźnik do v zostaną usunięte z w; zwolnij pamięć zajmowaną przez v i usuń rekurencyjnie k z u

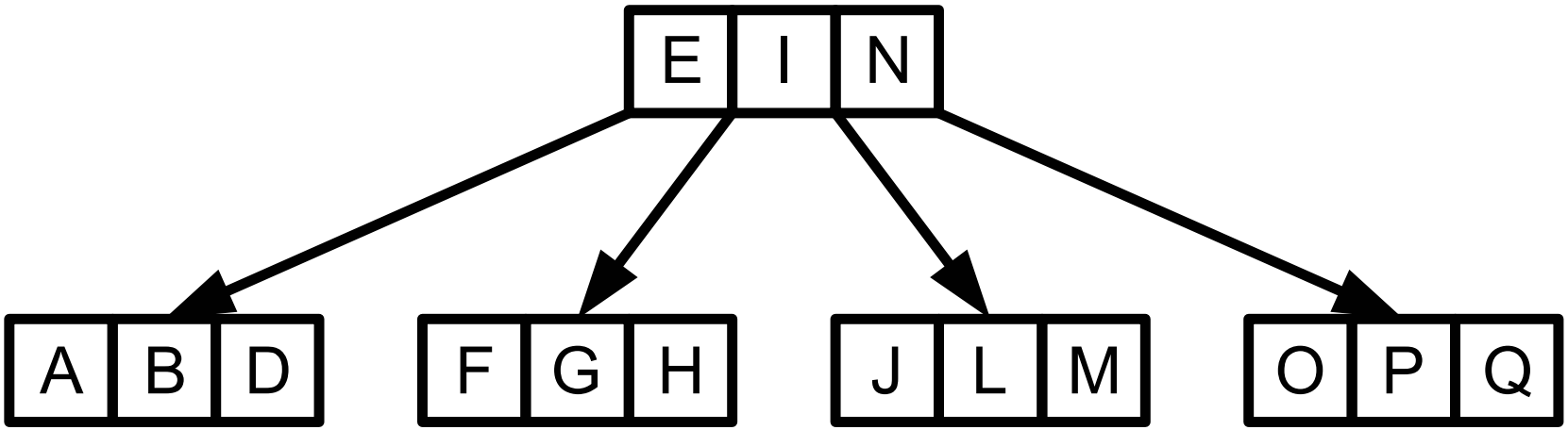
Usuwanie z B-drzewa

- jeżeli k nie występuje w wewnętrznym węźle w to wyznacz korzeń $w.son[i]$ poddrzewa, w którym k powinien się znajdować
- jeżeli $w.son[i]$ ma tylko $t-1$ kluczy:
  - jeżeli jeden z jego sąsiednich braci ma t kluczy, umieść w $w.son[i]$ dodatkowy klucz, przesuwając odpowiedni klucz z w, a w jego miejsce przenosząc klucz z lewego lub prawego brata (tego, który zawiera t kluczy); przesuń z wybranego brata do $w.son[i]$ wskaźnik do odpowiedniego syna

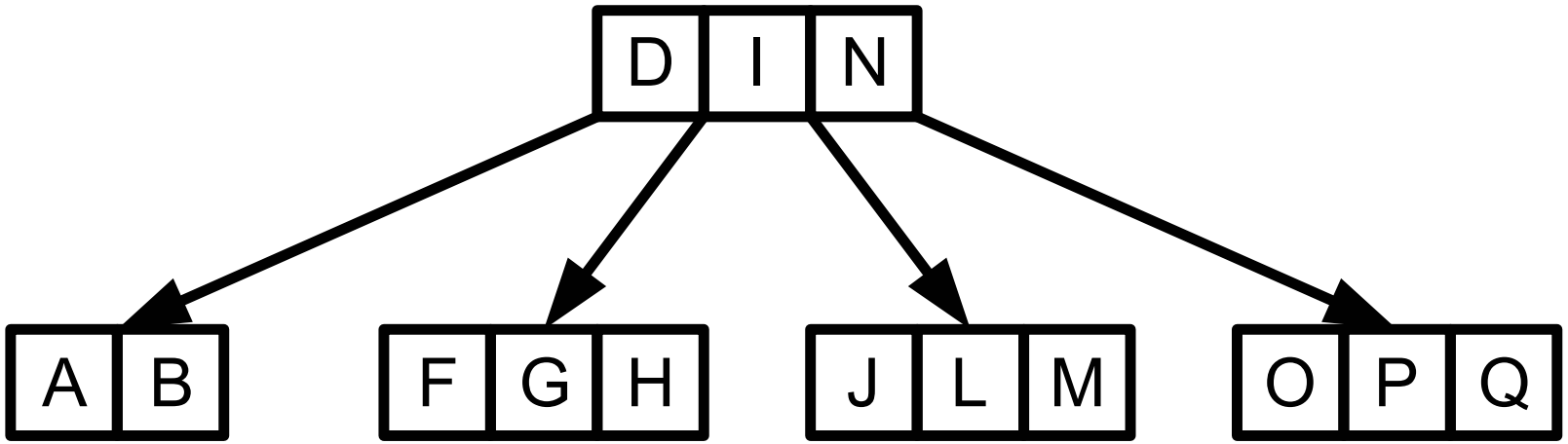
Usuwanie z B-drzewa

- jeżeli $w.son[i]$ ma tylko $t-1$ kluczy (c.d.):
  - jeżeli w $w.son[i]$ i sąsiedni bracia mają po $t-1$ kluczy, połącz $w.son[i]$ z jednym z sąsiednich braci, przesuwając odpowiednie klucz z w do nowo powstałego węzła (przesunięty klucz będzie kluczem środkowym)
- usuń rekurencyjnie k z właściwego poddrzewa

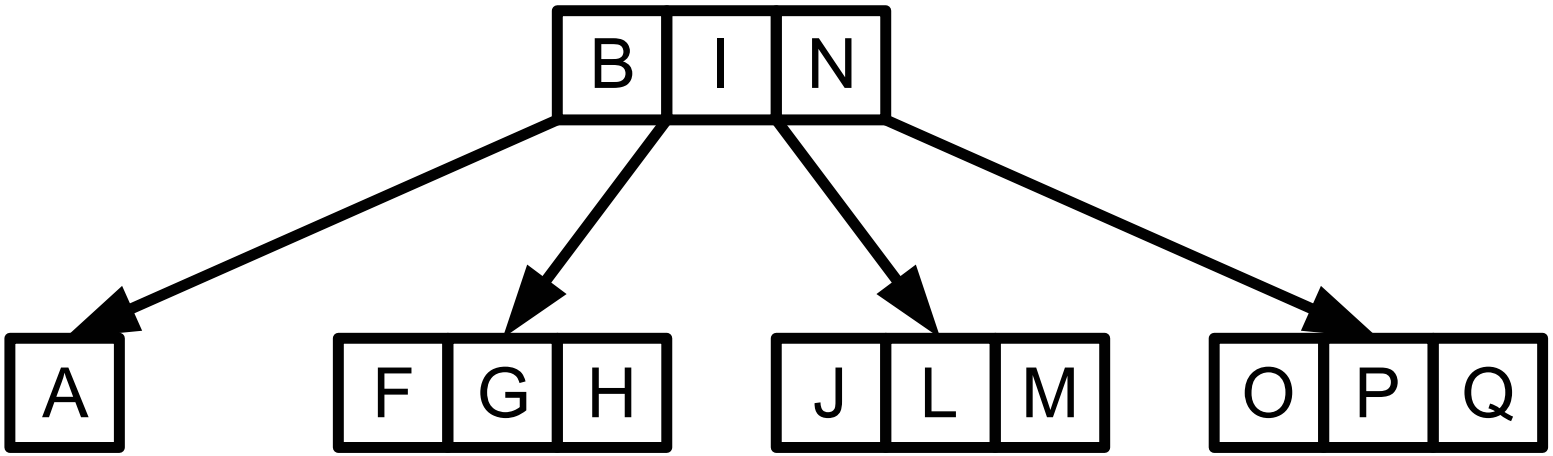
Usuwanie z B-drzewa



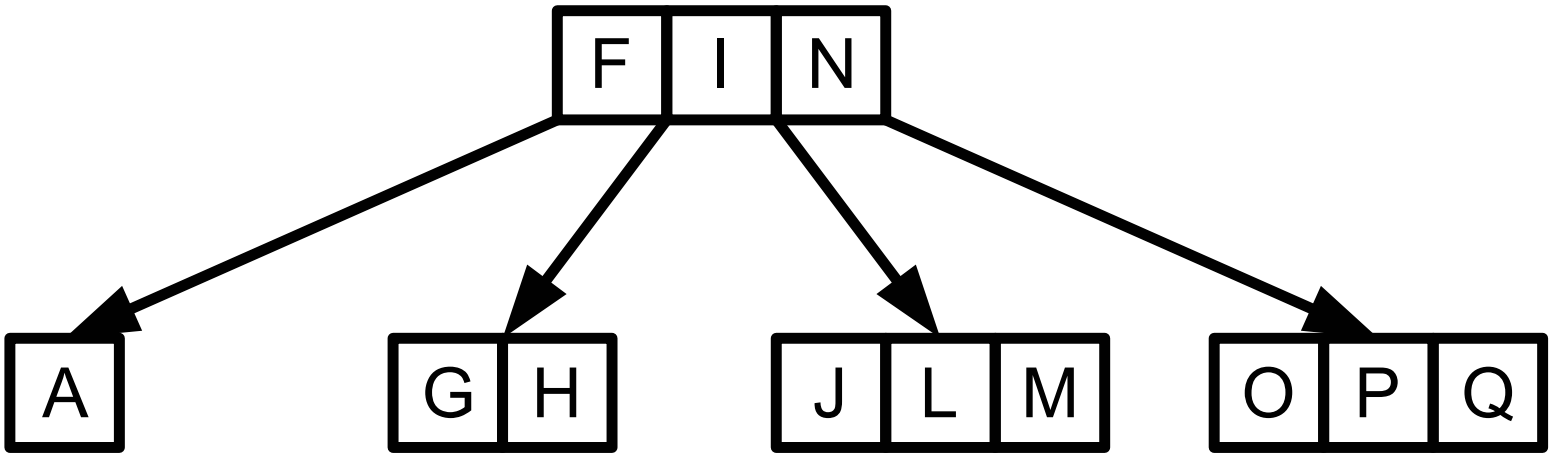
Usuwanie z B-drzewa



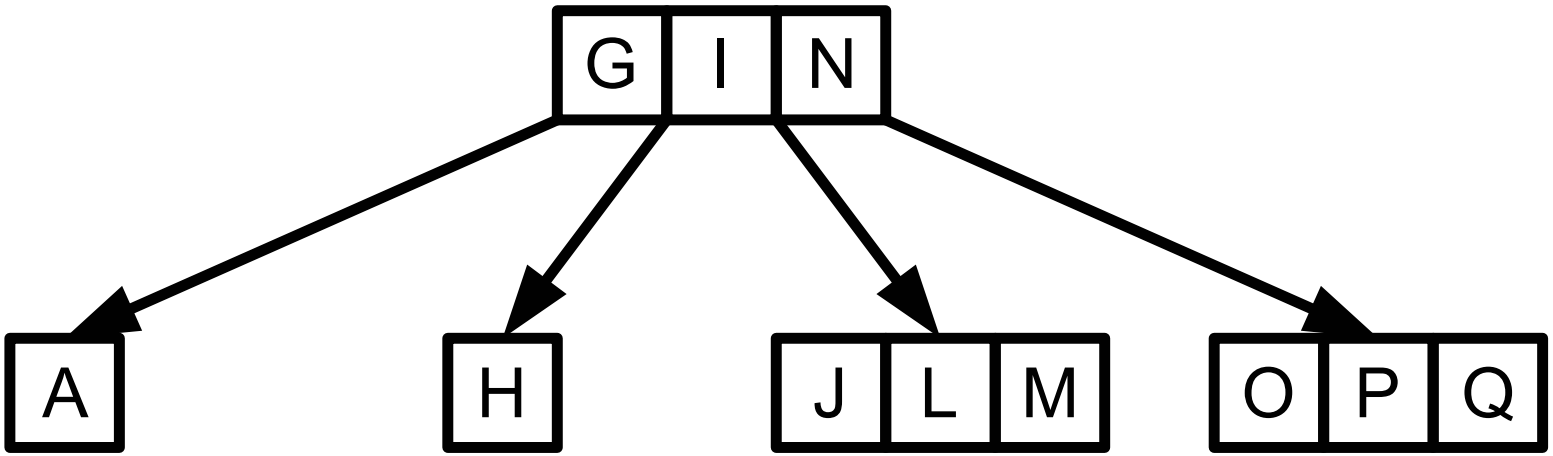
Usuwanie z B-drzewa



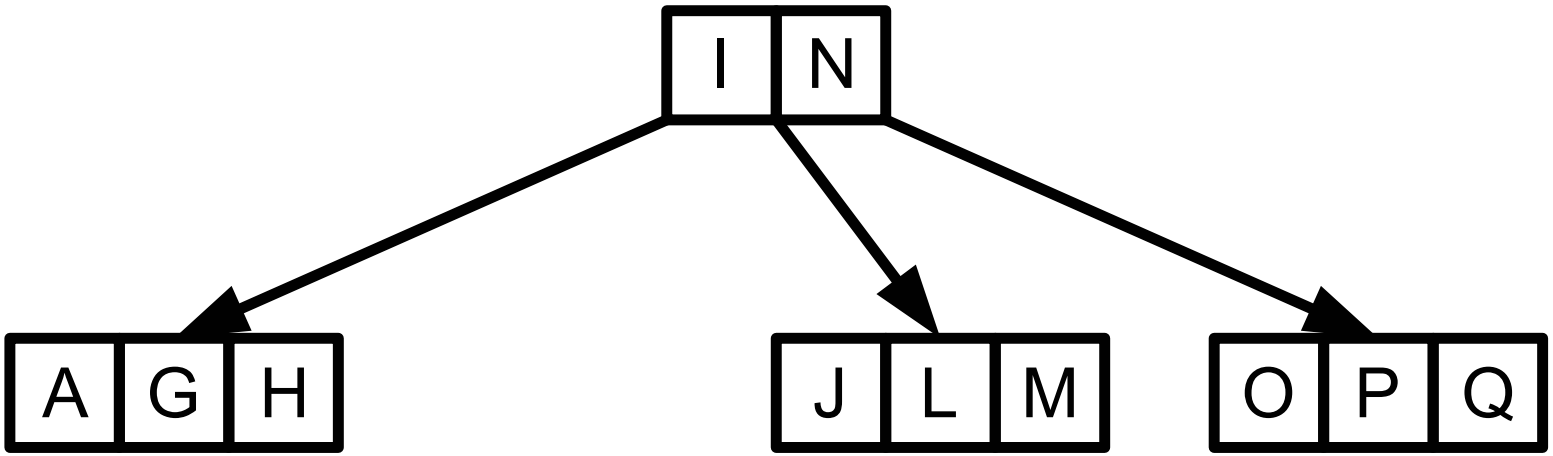
Usuwanie z B-drzewa



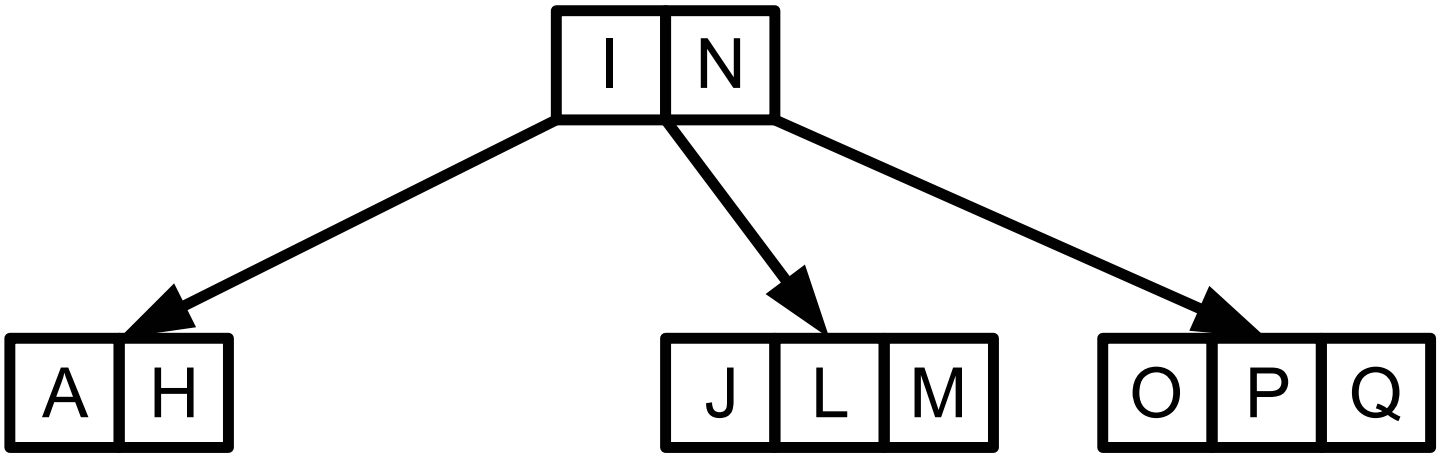
Usuwanie z B-drzewa



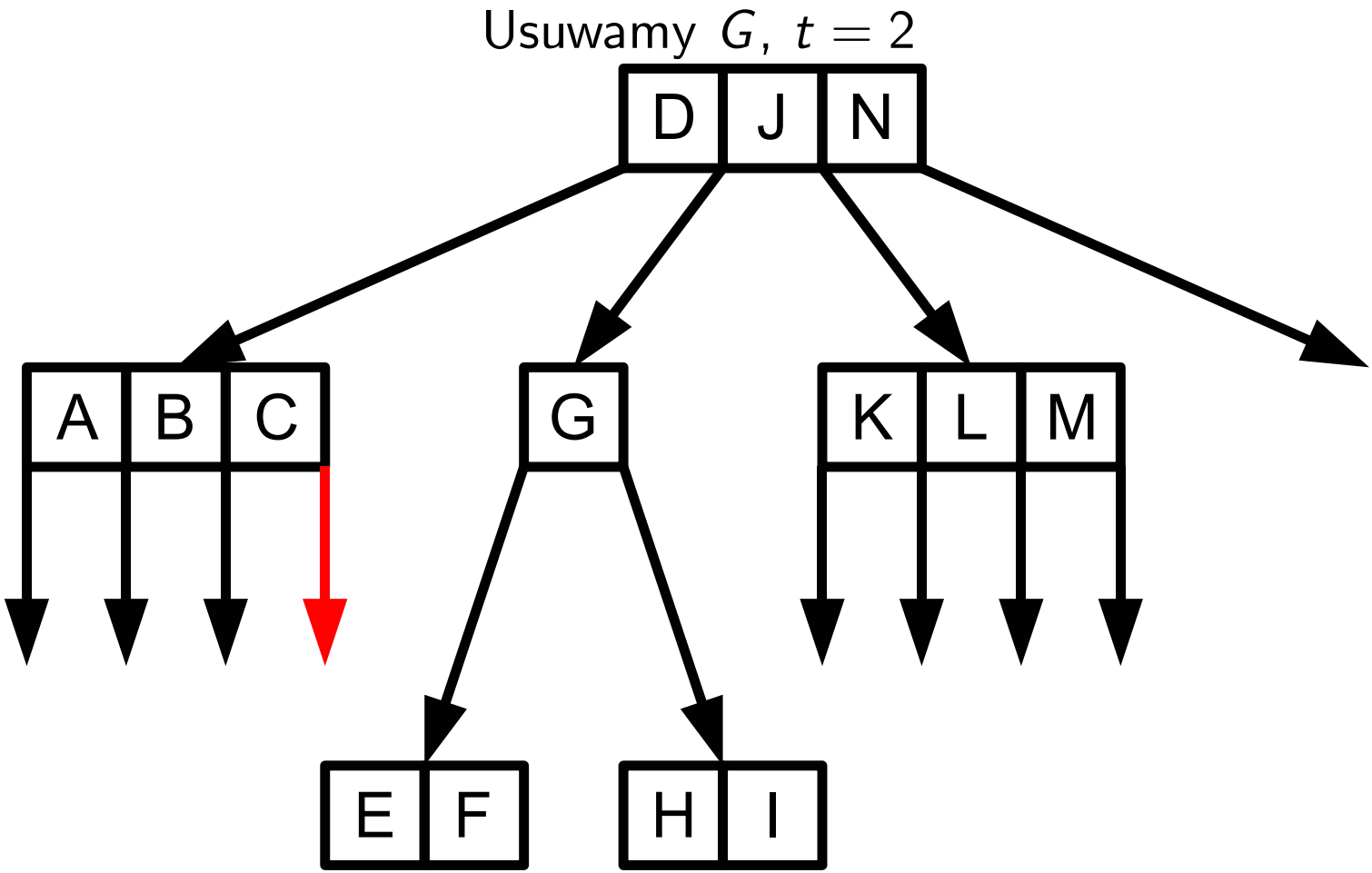
Usuwanie z B-drzewa



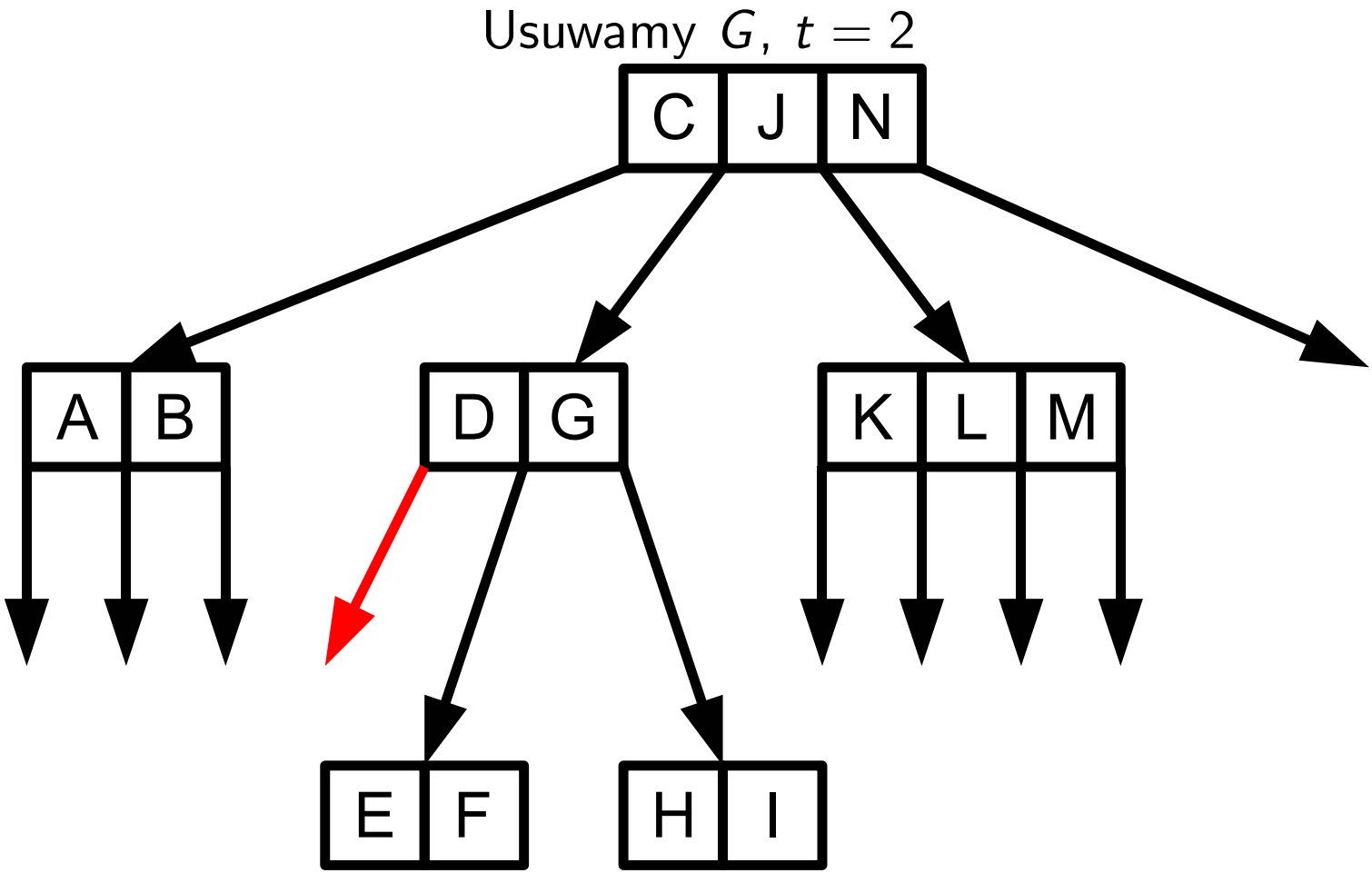
Usuwanie z B-drzewa



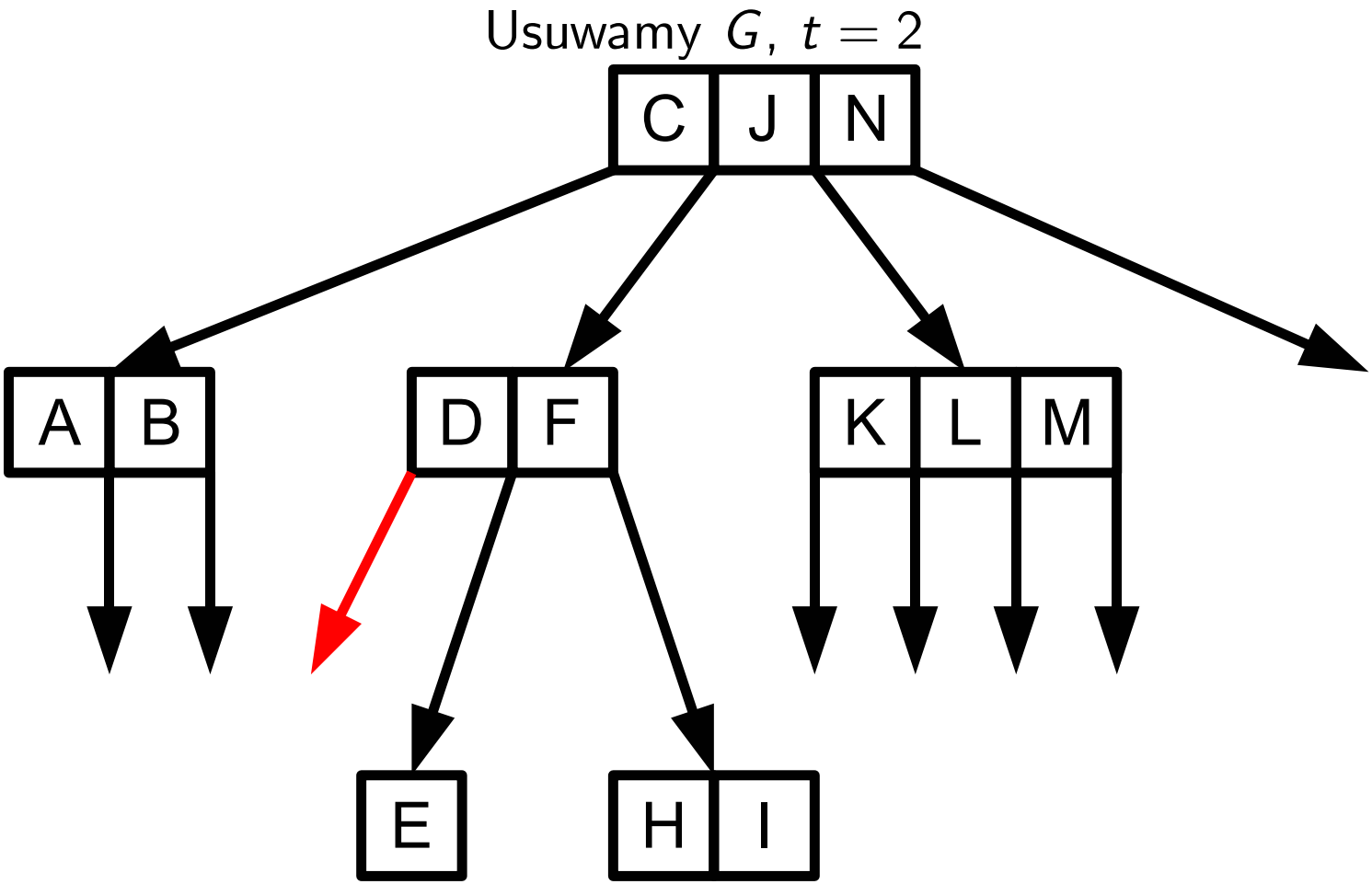
Usuwanie z B-drzewa



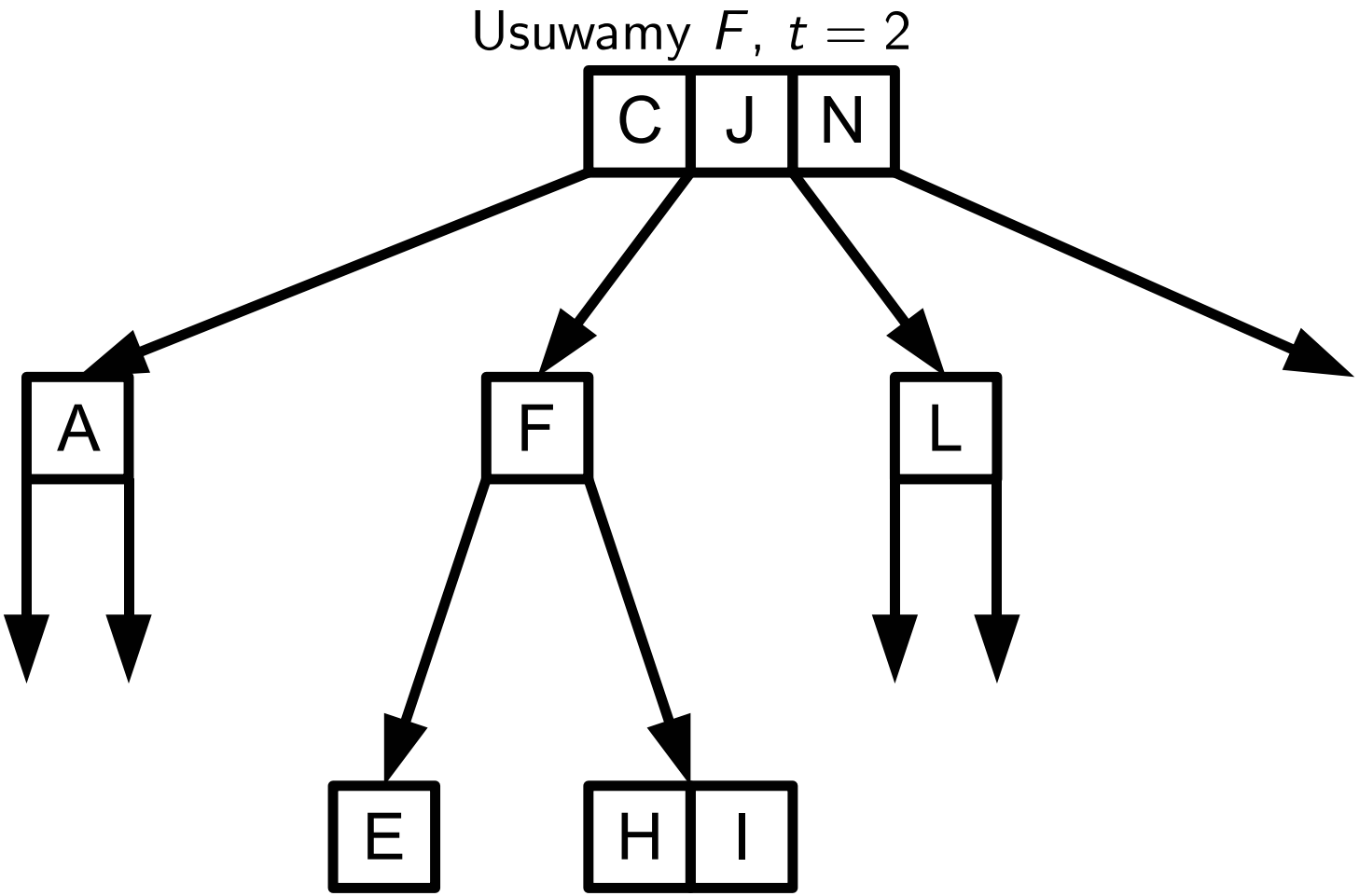
Usuwanie z B-drzewa



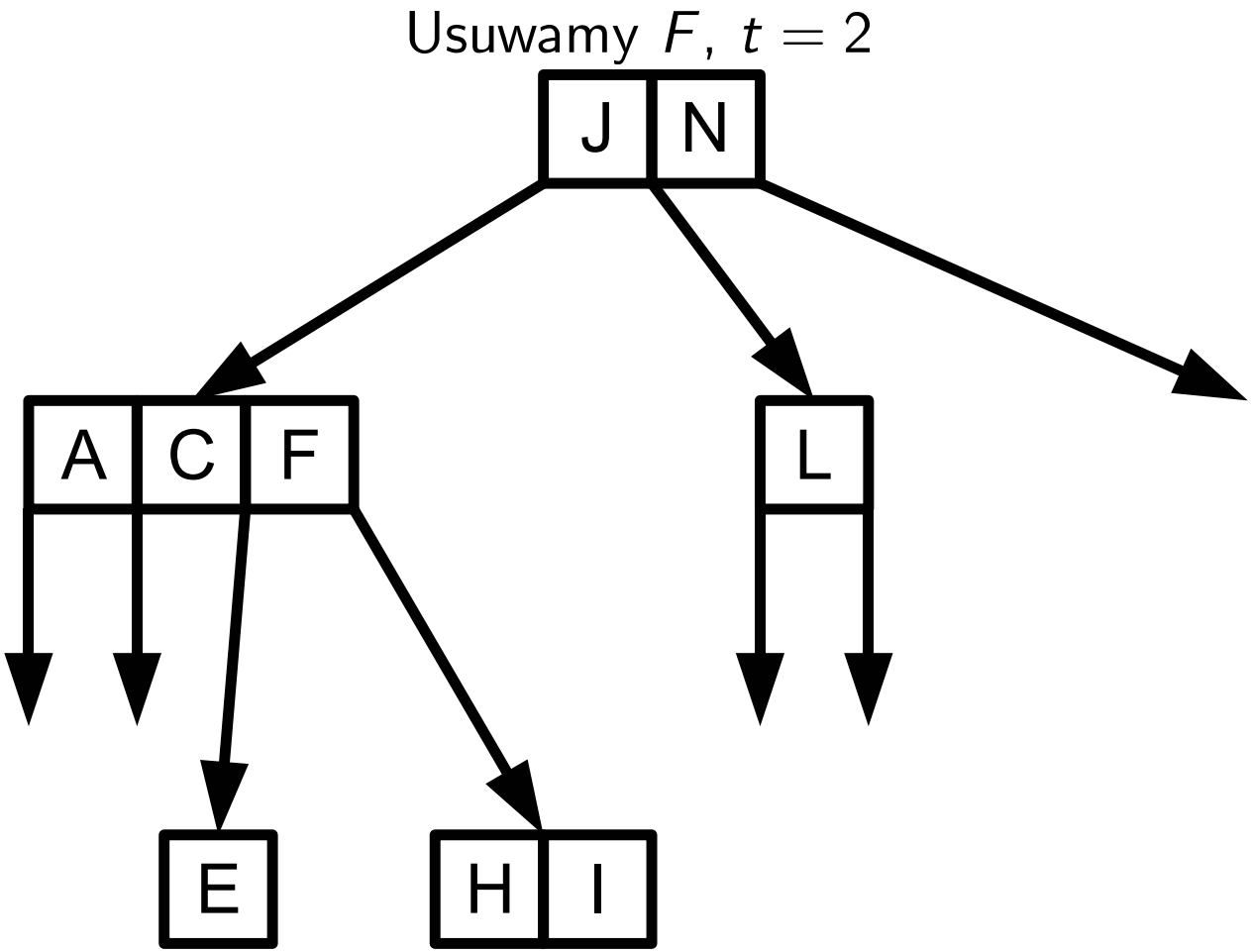
Usuwanie z B-drzewa



Usuwanie z B-drzewa

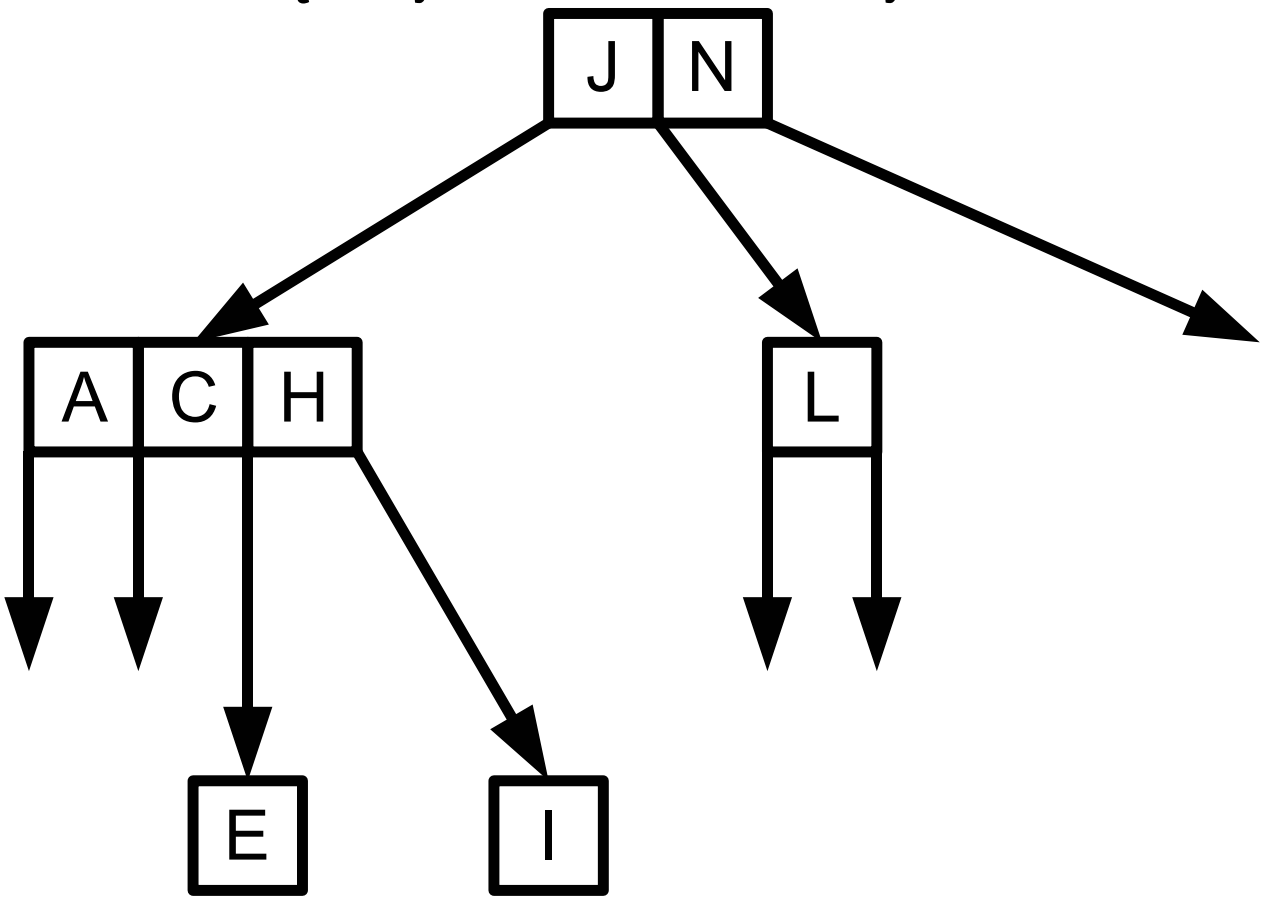


Usuwanie z B-drzewa

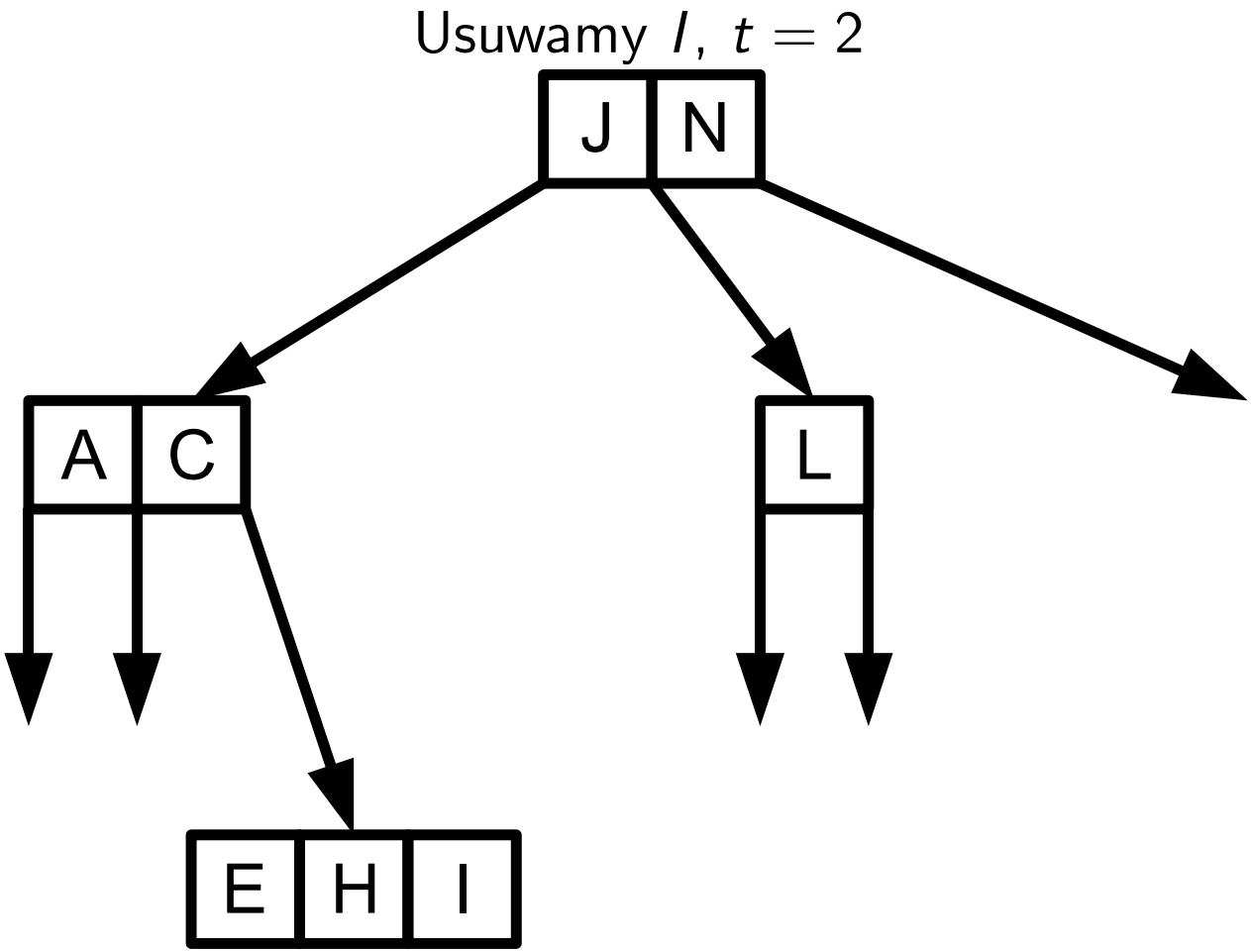


Usuwanie z B-drzewa

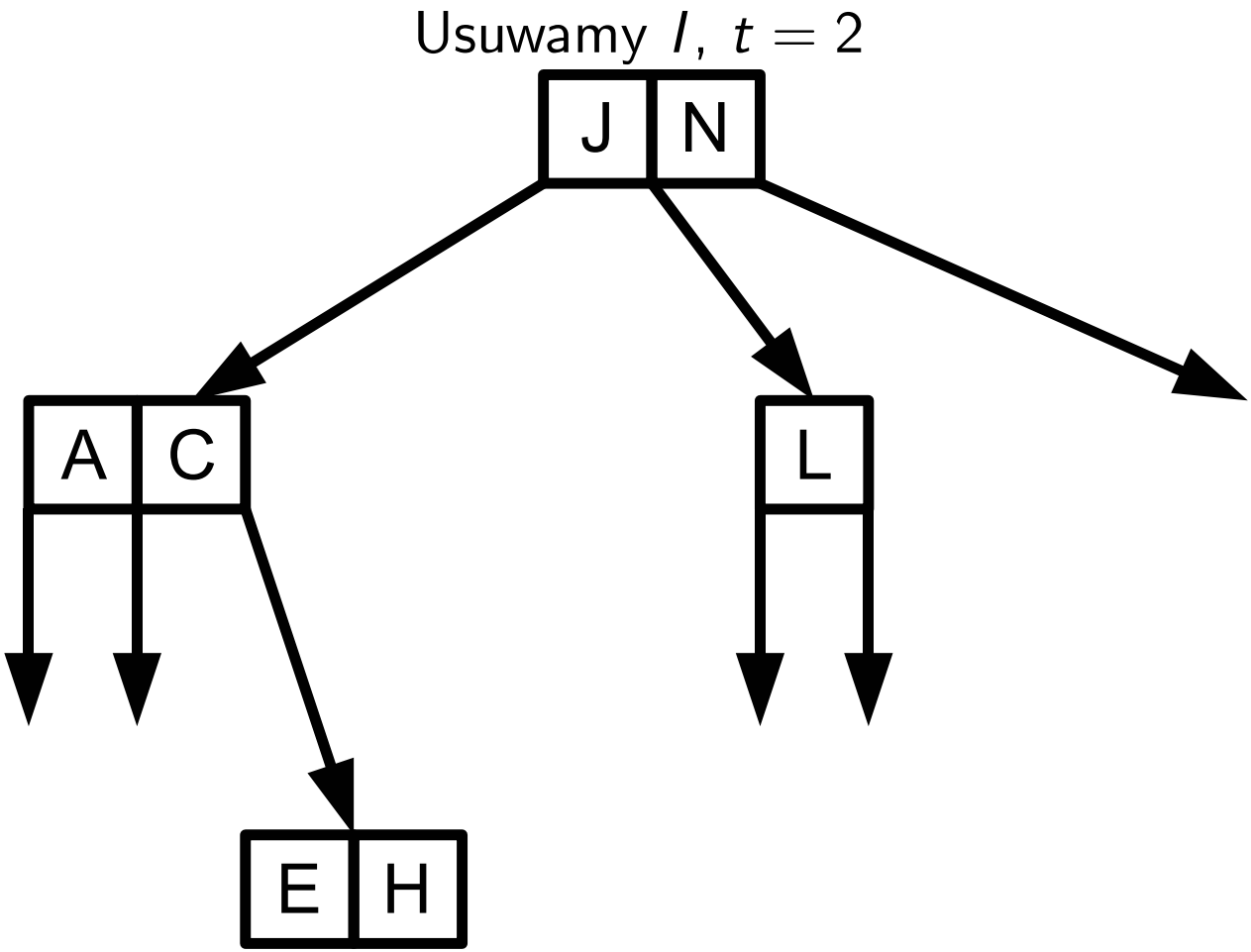
Usunęliśmy F, teraz usuwamy I, $t = 2$



Usuwanie z B-drzewa



Usuwanie z B-drzewa



Inne drzewa równoważone

- 2–3 drzewa
- AA-drzewa
- drzewa *splay*
- B^+ drzewa
- ...