Wrocław
University
of Science
and Technology

Struktury danych

Wykład 7

Słowniki, binarne drzewa poszukiwań

dr inż. Jarosław Rudy

HR EXCELLENCE IN RESEARCH

Słowniki (1)

- Słownik (mapa, tablica asocjacyjna) jest ADT przechowującym elementy w postaci pary klucz-wartość.
- Klucze mogą być dowolnego typu (najczęściej są to liczby lub łańcuchy tekstowe).
- Słownik mapuje zbiór kluczy na zbiór wartości (jak funkcja matematyczna).
  - Część kluczy może nie mieć wartości (funkcja częściowa).
- Generalnie każdy klucz ma co najwyżej jedną wartość.
  - Multimapa – uogólnienie słownika, w którym z jednym kluczem może być związane kilka wartości (np. kolekcja).

Słowniki (2)

Typowe operacje na słowniku:

- insert(k,v) – dodanie do słownika elementu o kluczu k i wartości v.
  - Jeśli element dla klucza k już istnieje, to zostanie nadpisany przez v.
- find(k), lookup(k) – zwraca $S[k]$, czyli element mapowany przez klucz k w słowniku S.
  - Jeśli taki element nie istnieje, to najczęściej zwracana jest specjalna wartość lub podnoszony jest wyjątek.
- delete(k), remove(k) – usunięcie elementu o kluczu k, usuwa mapowanie klucza.

Słowniki (3)

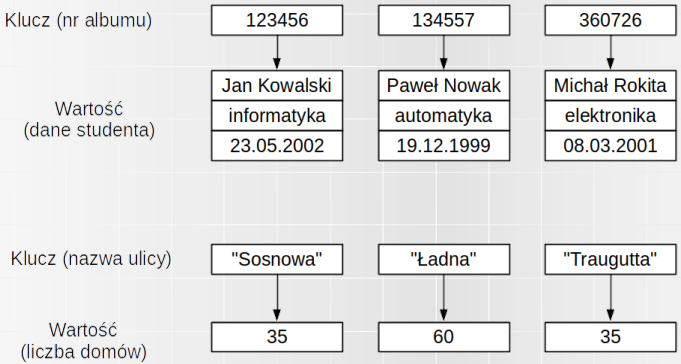
- exists(k) – zwraca czy klucz k ma przypisaną wartość w słowniku.
- size() – zwraca rozmiar słownika.
- empty() – zwraca czy słownik ma jakikolwiek przypisany klucz.
- keys() – zwraca (enumeruje) listę przypisanych kluczy słownika lub iterator na taką listę.
- values() – zwraca listę wartości słownika lub iterator na taką listę.

Słowniki (4)

- Podstawowa definicja słownika nie wymaga, by na kluczach definiować porządek (relacje $<$, $\leqslant$ itd.), ani by przechowywać je w jakimś porządku.
  - Niektóre z implementacji słowników będą jednak tego wymagały!
- Można jednak zdefiniować uporządkowany słownik, w którym elementy zachowują porządek przy listowaniu. Istnieją 2 powszechne definicje:
  - Porządek określony przez sortowanie kluczy (niezależny od wstawiania).
  - Porządek określony przez kolejność wstawiania (niezależny od klucza).

Słowniki (5)

Przykłady mapowania klucz-wartość w słownikach.



Słowniki (6)

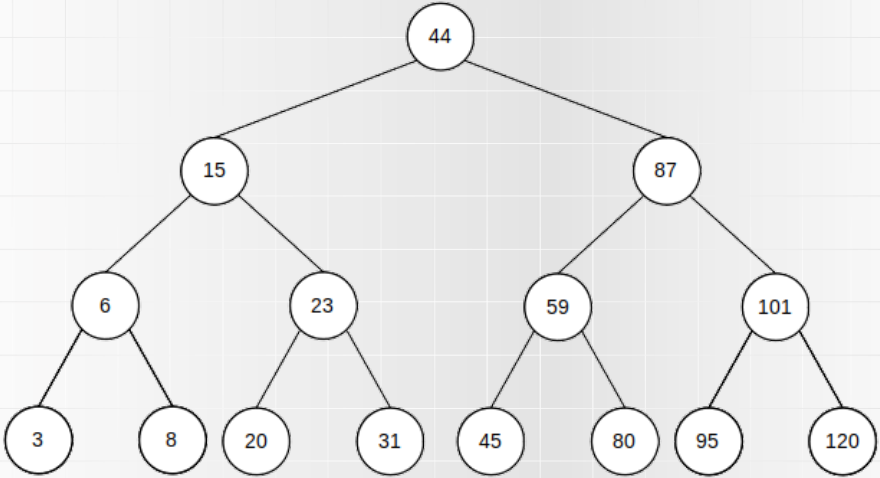
- Słownik zakłada szybkie wyszukiwanie elementu po zadanym kluczu.
- Dla implementacji z użyciem np. listy, średni i pesymistyczny czas wyszukiwania wyniesie $O(n)$.
- Listy z przeskokiem lub samoorganizujące się ulepszają przypadek średni do $O(\log n)$, ale pesymistycznie wciąż jest $O(n)$.
- Szybsze wyszukiwanie można uzyskać implementując słownik jako:
  - Binarne drzewo poszukiwań.
  - Tablicę mieszającą.

Binarne drzewo poszukiwań (1)

Drzewo binarne (binary search tree, BST):

- Drzewo binarne w którym klucz węzła jest większy niż klucz lewego syna i mniejszy niż klucz prawego syna.
  - Wymaga zdefiniowania porządku dla każdej pary kluczy.
- Wypisując BST metodą in-order otrzymamy elementy posortowane wg klucza (słownik uporządkowany).

Binarne drzewo poszukiwań (2)

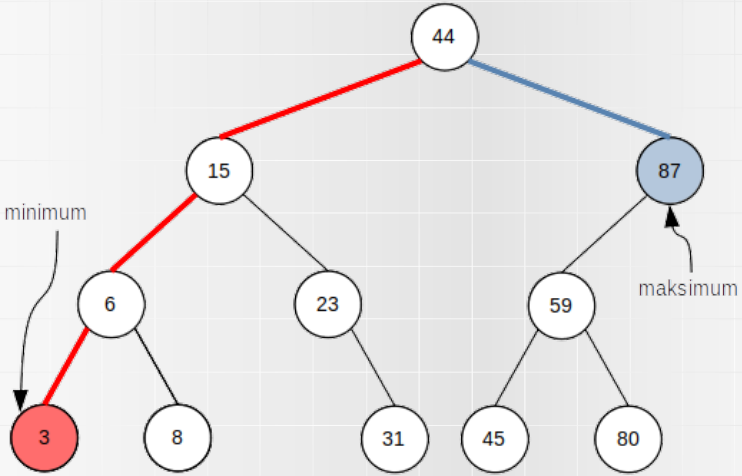


Binarne drzewo poszukiwań (3)

Operacje pomocnicze:

- minimum() – znajdowanie elementu minimalnego w drzewie:
  - Podążamy lewym podrzewem dopóki istnieje.
  - Jeśli nie ma lewego syna, aktualny węzeł jest szukanym minimum.
- maximum() – analogicznie (podążamy prawym poddrzewem do końca).

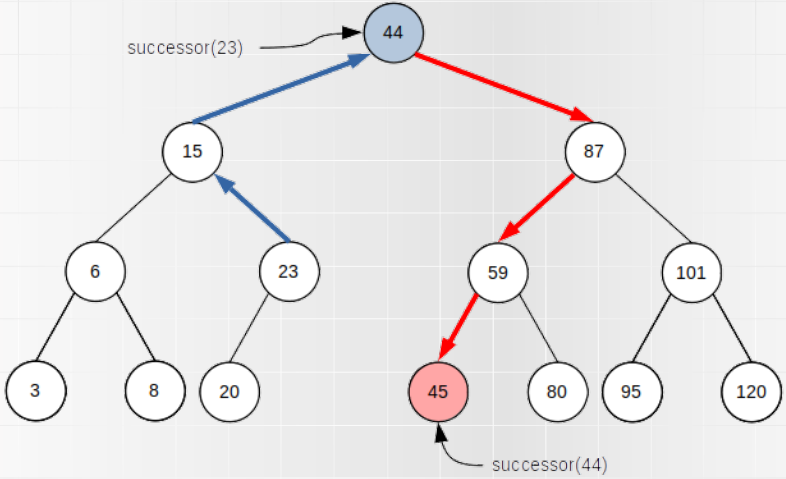
Binarne drzewo poszukiwań (4)



Binarne drzewo poszukiwań (5)

- successor(w) – znajdywanie następnika węzła w tj. węzła następnego w kolejności posortowanej (mającego najmniejszy klucz większy od $w.\mathrm{key}$):
  - Jeśli istnieje $w.\mathrm{right}$, to $\mathrm{successor}(w) \leftarrow \mathrm{minimum}(w.\mathrm{right})$.
  - Jeśli $w.\mathrm{right}$ nie istnieje, to następnikiem w jest pierwszy z przodków w, dla którego w leży w lewym poddrzewie.
    - Innymi słowy, zaczynając od $v \leftarrow w$ przechodzimy cyklicznie do rodzica ($v \leftarrow v.\mathrm{parent}$), do czasu aż $v.\mathrm{parent.left} = v$. Wtedy $v.\mathrm{parent}$ jest szukanym następnikiem.
- predecessor(w) – znajdywanie poprzednik węzła w tj. węzła poprzedniego w kolejności posortowanej (mającego największy klucz mniejszy od $w.\mathrm{key}$):
  - Analogicznie do następnika.

Binarne drzewo poszukiwań (6)



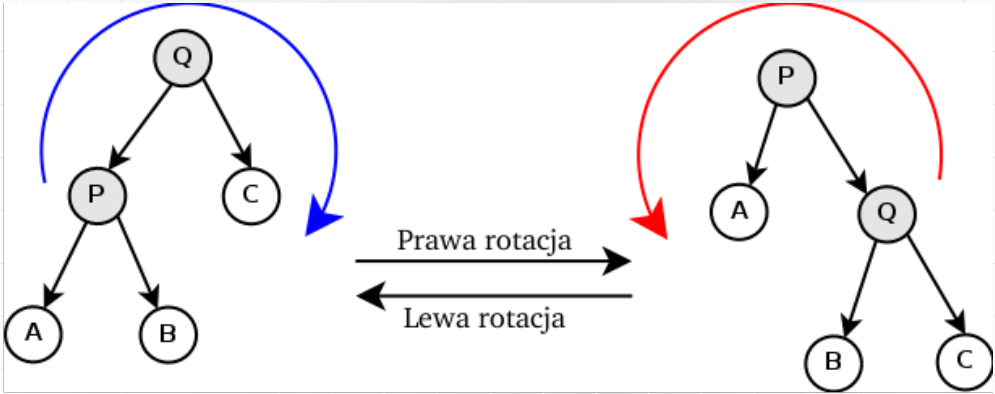
Binarne drzewo poszukiwań (7)

- Rotację definiujemy względem korzenia w pewnego poddrzewa (lub względem krawędzi łączącej ten korzeń go z jego synem).
- Rozróżniamy dwie podstawowe rotacje:
  - rotateLeft(w)
    - $\text{temp} \leftarrow w.\text{right.left}$
    - $w.\text{right.left} \leftarrow w$
    - $w.\text{right} \leftarrow \text{temp}$

Binarne drzewo poszukiwań (8)

- rotateRight(w)
  - $\text{temp} \leftarrow w.\text{left.right}$
  - $w.\text{left.right} \leftarrow w$
  - $w.\text{left} \leftarrow \text{temp}$
- Rotacja zmniejsza głębokość jednego wierzchołka/poddrzewa kosztem zwiększenia głębokości innego wierzchołka/poddrzewa.
- Lewa rotacja jest operacją odwrotną do prawej rotacji i vice versa.
- Rotacje zachowują porządek kluczy (przegląd in-order).

Binarne drzewo poszukiwań (9)



W obu przypadkach porządek kluczy to $A < P < B < Q < C$.

Binarne drzewo poszukiwań (10)

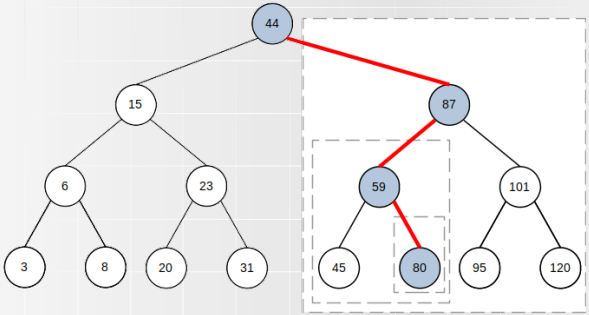
Operacja find(k):

- $w \leftarrow \mathrm{root}$
- Dopóki w istnieje (np. nie jest null):
  - Jeśli $k = key(w)$, zwracamy w (znaleziono klucz).
  - Jeśli $k < key(w)$, to $w \leftarrow left(w)$ (przechodzimy do lewego poddrzewa).
  - Jeśli $k > key(w)$, to $w \leftarrow right(w)$ (przechodzimy do prawego poddrzewa).
- Klucza nie ma w drzewie, zwróć odpowiednią wartość (null, wyjątek itp.).

Możliwa jest również wersja iteracyjna.

Binarne drzewo poszukiwań (11)

Przykład szukania w BST wartości o kluczu równym 80.



Jeśli lewe i prawe podrzewa zawsze mają podobny rozmiar, to z każdą decyzją odrzucamy ok. połowę pozostałych węzłów do sprawdzenia (zasada algorytmu przeszukiwania binarnego).

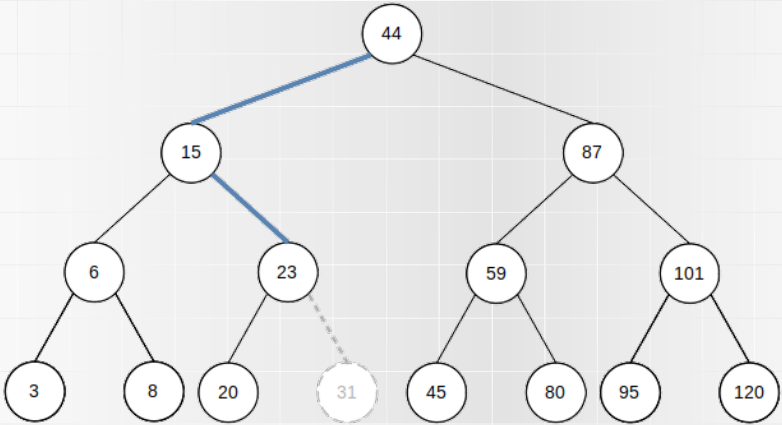
Binarne drzewo poszukiwań (12)

Operacja insert(k,v):

- Tworzymy nowy element $e = (k, v)$.
- Jeśli drzewo jest puste, to dodajemy e jako korzeń.
- W przeciwnym razie, znajdujemy miejsce do wstawienia węzła.
  - Odbywa się to prawie identycznie jak w operacji find(k).
  - Ponieważ k nie ma w drzewie, to natrafimy w końcu na null (brak poddrzewa w którym powinno być k).
  - Wpisujemy e w miejsce nulla (tj. jako odpowiedni syn liścia).

Binarne drzewo poszukiwań (13)

Przykład dodawania do drzewa węzła o kluczu 31.



Binarne drzewo poszukiwań (14)

Operacja delete(k):

- Znajdujemy węzeł w do usunięcia poprzez find(k).
- Usuwamy znaleziony węzeł w.
- Konieczność reorganizacji drzewa, możliwe 3 przypadki:
  - Jeśli w nie miał synów (był liściem), to nie robimy nic.
  - Jeśli w miał jednego syna, to syn zastępuje w.
  - Jeśli w miał dwóch synów, to wstawiamy w miejsce w jego następnika.

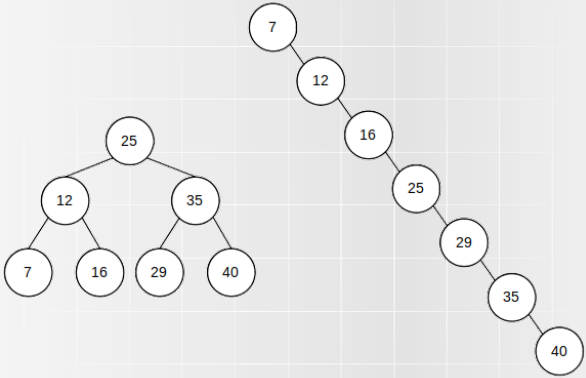
Binarne drzewo poszukiwań (15)

Złożoność operacji, zakładając drzewo o wyoskości h z n elementami:

- insert(k,v), find(k), exists(k) i delete(k):
  - średnio $O(\log n)$,
  - pesymistycznie $O(h)$.
- keys() i values():
  - średnio i pesymistycznie $\Theta(n) \in O(n)$.
- size() i empty():
  - średnio i pesymistycznie $O(1)$.

Binarne drzewo poszukiwań (16)

Niekorzystne wstawianie zwiększa rozmiar drzewa. W najgorszym przypadku mamy $h = n$ (drzewo binarne degraduje się do listy).



Binarne drzewo poszukiwań (17)

- Możemy rozważać drzewo zrównoważone. 2 definicje:
  - Równoważenie wysokością: dla każdego węzła wysokość obu jego poddrzew różni się co najwyżej o stałą:
    - Doskonale zrównoważone, jeśli wysokość różni się najwyżej o 1.
  - Równoważenie wagą: dla każdego węzła jego waga różni się najwyżej o stały czynnik od wagi synów.
    - Waga węzła to rozmiar jego poddrzewa plus 1.

Binarne drzewo poszukiwań (18)

Niektóre sposoby równoważenia drzew:

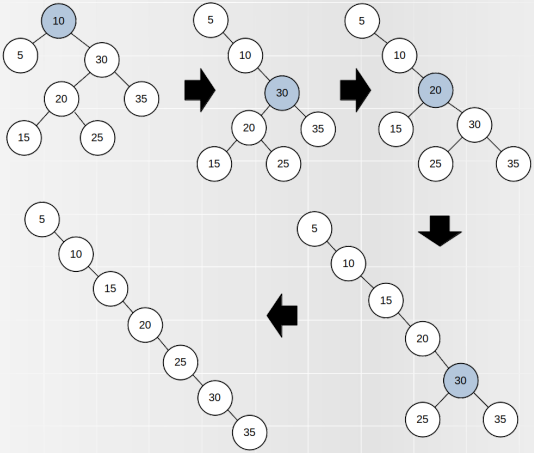
- Algorytm równoważenia DSW.
- Drzewa samorównoważące się np.:
  - Drzewa AVL.
  - Drzew czerwono-czarne.
  - B-drzewa.

Algorytm DSW (1)

- Algorytm równoważy zadane drzewo (gwarantując, że $h \in O(\log n)$) w dwóch fazach z użyciem rotacji.
  - W fazie pierwszej wykorzystywane są wielokrotne prawe rotacje, wykonywne od korzenia.
    - W wyniku drzewo zostaje zamienione w listę (tzw. kręgosłup, vine).
  - W fazie drugiej iteracyjnie stosujemy lewe iteracje na co drugim węźle wzdłuż prawej gałęzi drzewa.
    - Zabieg ten stosowany jest wielokrotnie, za każdym razem zmniejszając wysokość drzewa o połowę.

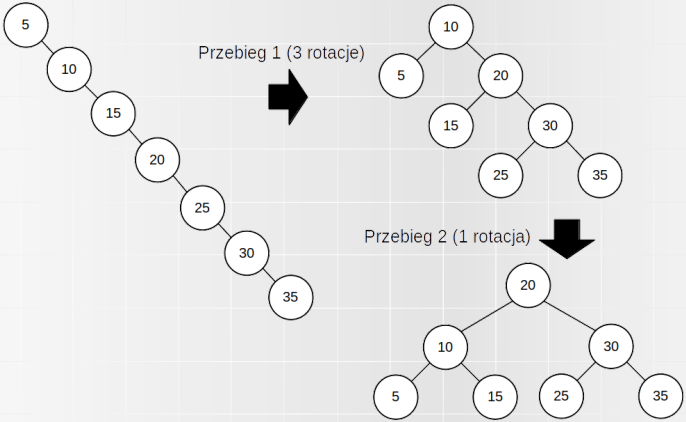
DSW (2)

Przykład fazy pierwszej



DSW (2)

Przykład fazy drugiej



Algorytm DSW (4)

- Obie fazy działają w czasie $O(n)$.
- Niskie zapotrzebowanie na pamięć ($O(1)$).
- Uzyskujemy drzewo doskonale zrównoważone.
- Brak potrzeby sortowania lub dalszej dekompozycji drzewa.
- Konieczność ręcznego stosowania.

Drzewa AVL (1)

- Samorównoważące się BST.
- Każdy wierzchołek przechowuje współczynnik zrównoważenia (różnica wysokości jego lewego i prawego poddrzewa, 2 bity).
  - Wartości -1, 0 oraz 1 są w porządku (ale zmiana może wymagać propagacji tej informacji).
  - Wartości -2 oraz 2 wymagają naprawy poziomu wyważenia węzłów.
- Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas $O(\log n)$.

Drzewa AVL (2)

- Operacja wstawiania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - Wyważenie aktualizowane na 0 kończy proces bez konieczności dalszych zmian.
  - Wyważenie aktualizowane na -2 lub 2 oznacza, że drzewo straciło własność AVL. Naprawa przebiega z użyciem 1–2 rotacji, po których algorytm się kończy.
  - Wyważenie aktualizowane na -1 lub 1 wymaga akualizacji zmiany w rodzicu (kontynuujemy algorytm).
- Czas $\Theta(\log n)$.

Drzewa AVL (3)

- Operacja usuwania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - Wyważenie aktualizowane na -1 lub 1 kończy proces bez konieczności dalszych zmian.
  - Wyważenie aktualizowane na -2 lub 2 oznacza, że drzewo straciło własność AVL. Naprawa przebiega z użyciem 1–2 rotacji, po czym proces należy kontynuować.
  - Wyważenie aktualizowane na 0 wymaga akualizacji zmiany w rodzicu (kontynuujemy algorytm).
- Czas $O(\log n)$.

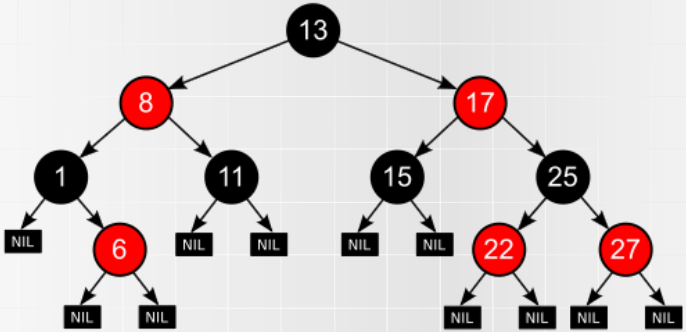
Drzewa czerwono-czarne (1)

- Samorównoważące się BST.
- Każdy węzeł ma kolor (1 bit) a drzewo musi spełniać następujące własności:
  - Każdy węzeł jest czerwony lub czarny.
  - Korzeń jest czarny.
  - Każdy liść jest czarny.
  - Synowie czerwonego są czarni.
  - Dla węzła w ścieżki od w do jego potomków-liści mają tyle samo węzłów czarnych.
- Dla tego drzewa przez liść rozumiemy null (nil)!

  Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas $O(\log n)$.

Drzewa czerwono-czarne (2)

Przykład drzewa czerwono-czarnego.



Dzięki własności, dla każdego węzła w jego najdłuższa ścieżka do liścia jest co najwyżej 2x dłuższa niż najkrótsza.

Drzewa czerwono-czarne (3)

- Operacja wstawiania:
  - Początek działa jak dla zwykłego BST.
  - Nowy węzeł kolorowany jest wstępnie na czerwono.
  - Mogły zostać złamane niektóre własności drzewa.
    - Przeprowadzamy korektę zależnie od występującego przypadku (kilka możliwych), w czasie $O(1)$, w tym co najwyżej jedna rotacja.
- Czas $O(\log n)$.

Drzewa czerwono-czarne (4)

- Operacja usuwania:
  - Początek działa jak dla zwykłego BST.
  - Dalsza część jest bardziej skomplikowana.
    - Możliwe jest wiele przypadków, zależnie od koloru usuwanego węzła i liczby jego dzieci.
    - Jednakże przywracanie własności drzewa czerwono-czarnego przy usuwaniu zawsze wymaga co najwyżej 2 rotacji.
  - Czas $O(\log n)$.

AVL vs red-black tree

- Oba drzewa zapewniają find(), insert() i delete() w czasie $O(\log n)$.
- Drzewa AVL:
  - Są lepiej wyważone i w praktyce find() jest dla nich szybsze.
  - Większy koszt operacji insert() oraz delete() – możliwa konieczność przywracania własności wzdłuż całej wysokości drzewa.
- Drzewa czerwono-czarne:
  - Nie gwarantują doskonałego wyważenia.
  - Koszt naprawy własności drzewa dla insert() i delete() jest $O(1)$.
- Wybór zależny od tego których operacji spodziewamy się więcej.

Drzewa zrównoważone

- Zbalansowane BST zapewnia wiele operacji w czasie $O(\log n)$, w tym znalezienie minimum i maksimum.
- Takie BST może więc posłużyć do implementacji kolejki priorytetowej.
  - Zaletą jest możliwość implementacji jednocześnie operacji extract-min jak i extract-max (kolejka priorytetowa „dwustronna").
  - Wadą jest mniejsza szybość:
    - Operacje BST mają zwykle większą stałą niż operacje kopcowe.
    - Reprezentacja kopca z użyciem tablicy dynamicznej lepiej współpracuje z pamięcią podręczną procesora niż BST.