Algorytmy i Struktury Danych

Słowniki

(c) Marcin Sydow

Zawartość wykładu

- definicja słownika
- analiza naiwnych implementacji słownika
- tablice mieszające
- własności funkcji mieszającej
- analiza operacji słownika zaimplementowanych na tablicy mieszającej
- sposoby rozwiązywania kolizji
- definicja słownika uporządkowanego
- definicja i własności drzewa BST
- działanie i analiza operacji słownika uporządkowanego na drzewie BST
- ograniczenia drzew BST
- definicja i opis motywacji oraz pomysłu drzewa AVL

Definicja słownika

Słownik jest abstrakcyjną strukturą danych służącą do operowania na parach klucz-wartość o następującym zbiorze operacji:

- search(K key)
  (zwraca wartość związaną z kluczem key)[1]
- insert(K key, V value) // umieszcza nową parę klucz-wartość w słowniku
- delete(K key) // kasuje parę związaną z kluczem key

(zakłada się, że klucze są unikatowe)

[1] może zwracać specjalną wartość lub rzucać wyjątek, jeśli takiego klucza nie ma w słowniku

Przykłady zastosowań

- baza kontaktów
  (np. klucz: osoba, wartość: numer telefonu lub odwrotnie)
- system konfiguracyjny: kolekcja par cecha-wartość (property-value)
- kompilatory i interpretery języków programowania: klucz: nazwa zmiennej, wartość: typ i adres w pamięci
- słownik języka obcego: klucz: wyraz w języku bazowym, wartość: znaczenie w innym języku
- etc.

Implementacje

- "naiwne": tablice lub listy (posortowane lub nie)
- tablice mieszające (ang. hash tables)
- drzewa wyszukiwań binarnych (ang. binary search trees (BST))
- drzewa AVL
- samoorganizujące drzewa BST*
- (a,b)-drzewa* (w szczególności: 2-3-trees)
- B-drzewa*
- etc.

* - nie omawiane w tym kursie

Proste implementacje słownika

Dwie tablice: keys i values. Tablica keys trzyma klucze a "równoległa" tablica values pod odpowiadającymi indeksami trzyma wartości. Klucze mogą być posortowane albo nie.

- nieposortowane:
  search: **O(n)**; insert: O(1); delete: **O(n)**
- posortowane:
  search: O(log n); insert **O(n)**; delete **O(n)**

(rozmiar danych: liczba elementów (n); op. dom.: porównanie klucza)

Niestety przy takiej prostej implementacji niektóre operacje mają nieakceptowalnie wysoką liniową złożoność czasową.
(sytuacja nie jest lepsza gdy zamiast tablic używamy list dowiązaniowych)

Potrzebna jest bardziej efektywna implementacja. Przykładem efektywnej implementacji są **tablice mieszające** (ang. hash tables)

Adresowanie bezpośrednie

Załóżmy, że klucze są liczbami naturalnymi z zakresu [0, ..., m-1]

Wtedy słownik można prosto zaimplementować jako tablicę, gdzie kluczem jest indeks i pod nim trzymana jest odpowiadająca temu kluczowi wartość.
W takim przypadku wszystkie operacje słownika mają znakomitą **stałą złożoność czasową** $O(1)$!

Implementacja ta stwarza jednak 2 problemy:

- ilość zużywanej pamięci jest proporcjonalna do najwyższej możliwej potencjalnej wartości klucza (m) a nie liczby przechowywanych aktualnie kluczy
- rozwiązanie działa tylko dla kluczy będących liczbami naturalnymi

Oba problemy można rozwiązać rozszerzając powyższy pomysł do tzw. tablic mieszających.

Tablice mieszające (ang. hashtables)

Tablice mieszające rozszerzają pomysł adresowania bezpośredniego o jeden dodatkowy krok:
Do przeliczenia wartości klucza na indeks w tablicy (pod którym jest odpowiadająca mu wartość) stosuje się tzw. **funkcję mieszającą**:

$$hash : U \rightarrow [0, ..., m-1]$$

(gdzie U to uniwersum wszystkich możliwych kluczy)

Dzięki temu pomysłowi rozwiązuje się oba problemy:

- ilość zużywanej pamięci jest teraz proporcjonalna do parametru m (a nie rozmiaru uniwersum U)
- typ klucza może być dowolny (jest typem argumentu funkcji mieszającej)

Kolizje

Ponieważ parametr m jest na ogół niższy niż rozmiar uniwersum |U|, więc nieuniknione jest, że funkcja mieszająca nie jest różnowartościowa (a więc dla różnych kluczy k1 i k2 musi zachodzić hash(k1) == hash(k2).

Sytuację taką nazywamy **kolizją**. Stwarza ona pewien problem, gdyż na jedno miejsce w tablicy mieszającej (odpowiadające różnym kluczom) przypada więcej niż jedna wartość.

Do metod rozwiązywania problemu kolizji należą m.in.:

- metoda mieszania wielokrotnego (w przypadku natrafienia na zajęte miejsce mieszanie powtarza się *w sposób odtwarzalny* aż do znalezienia pierwszego wolnego miejsca). Wada: w tablicy można przechowywać maksymalnie m elementów.
- metoda łańcuchowa: w każdym miejscu tablicy umieszczana jest (np. dowiązaniowa) lista elementów (można więc na każdym miejscu umieścić więcej niż jeden element), która może być liniowo przeszukiwana

Wymagane własności funkcji mieszającej

1. musi być obliczalna bardzo szybko (w szczególności w stałym czasie, niezależnym od liczby kluczy)
2. musi "równomiernie rozkładać obciążenie" w tablicy (ang. uniform load), czyli dla klucza wylosowanego z rozkładu jednostajnego z uniwersum U każda wartość z zakresu [0,...,m-1] musi być jednakowo prawdopodobna.

Własność pierwsza jest pożądana z oczywistych względów (efektywność), natomiast druga zapewnia, że przeciętna długość list (w przypadku trafiania kluczy na te same pozycje) będzie możliwie krótka.

Współczynnik obciążenia α zdefiniowany jest jako iloraz $\alpha = n/m$, gdzie n oznacza liczbę par (klucz-wartość) przechowywaną aktualnie w tablicy mieszającej.

Wlasność 2 powyżej, zapewnia, że pesymistyczna złożoność operacji słownikowych na tablicy mieszającej jest bliska $O(\alpha)$ (gdyż wszystkie listy mają zbliżoną długość do α)

Przykład prostej funkcji mieszającej

Jeśli klucze są liczbami całkowitymi, to najprostszym przykładem funkcji mieszającej jest funkcja `modulo m`, tzn.:

$$hash(key) = key \bmod m$$

Gdyż dla dowolnej liczby całkowitej key, wartość $key \bmod m \in [0, ..., m-1]$, jest to funkcja szybko obliczalna[2] oraz wszystkie wartości [0,...,m-1] są "jednakowo obciążone".

W niektórych zastosowaniach (np. sprawdzaniu integralności kodów źródłowych) od funkcji mieszającej wymaga się dodatkowo, aby wartość mieszająca była była trudna do odwrócenia (czyli ustalenia jaki był argument). Funkcja "modulo" nie spełnia tego warunku, ale w praktyce stosuje się wyrafinowane funkcje mieszające spełniające ten warunek (np. funkcja MD5 używana w systemach unixowych)

[2] szczególnie dla m będących potęgami 2 – wtedy wystarczy wziąć ostatnie $log_2(m)$ bitów liczby key w reprezentacji binarnej

Podsumowanie tablic mieszających

Tablice mieszające jako implementacja słownika mają następujące zalety:

- efektywność operacji słownika (złożoność czasowa $O(\alpha)$, gdzie α to współczynnik obciążenia tablicy, jeśli funkcja mieszająca spełnia warunek jednorodnego obciążenia)
- elastyczny sposób równoważenia złożoności pamięciowej i czasowej za pomocą parametru m – rozmiaru tablicy (im wyższy tym niższa przeciętna złożoność czasowa, ale wyższe zużycie pamięci), który można dobrać w zależności od spodziewanej liczby przechowywanych par klucz-wartość.

W przypadku, gdy klucze są typu uporządkowanego, mogą być przydatne operacje opierające się na porządku: np. znajdź maksymalny/minimalny klucz lub znajdź następny/poprzedni najbliższy klucz do podanego.

Tablice mieszające nie wspierają efektywnej implementacji takich dodatkowych operacji (tzn. miałyby one złożoność liniową)

Słownik uporządkowany (ang. Dynamic Ordered Set)

Słownik uporządkowany to abstrakcyjna struktura danych, będąca rozszerzeniem słownika, zdefiniowana następującymi operacjami:

- search(K key)
- insert(K key, V value)
- delete(K key)
- K minimum() // zwróć minimalny klucz w słowniku
- K maximum() // zwróć maksymalny klucz w słowniku
- predecessor(K key) // zwóć klucz będący bezpośrednim poprzednikiem podanego
- successor(K key) // zwróć klucz będący bezpośrednim następnikiem podanego

Zakładamy, że typ klucza K jest liniowo uporządkowany

Do najprostszych efektywnych implementacji słownika uporządkowanego należy drzewo BST.

Drzewo wyszukiwań binarnych BST (ang. Binary Search Tree)

Drzewo BST jest drzewem binarnym, gdzie każdy węzeł przechowuje pewien klucz (z przypisaną wartością) i spełniony jest następujący **warunek porządku BST**:

Dla każdego węzła x, klucz w tym węźle jest niemniejszy, niż wszystkie klucze w lewym poddrzewie węzła x oraz niewiększy, niż wszystkie klucze w prawym poddrzewie węzła x.

Uwaga: w drzewie binarnym nawet jeśli dany węzeł ma tylko jednego syna, to jest jasno określone, czy jest to syn lewy czy prawy.

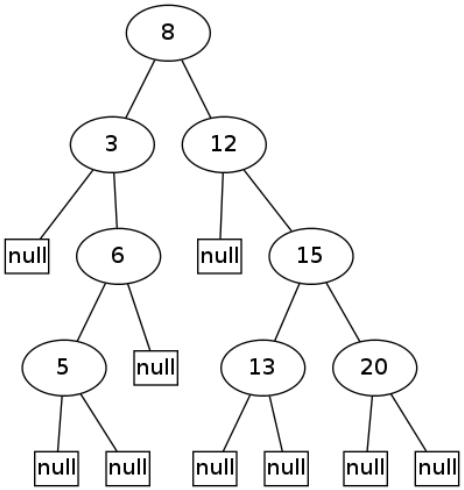
Pytanie sprawdzające: Gdzie znaleźć klucz minimalny (maksymalny)?

Implementacja drzewa BST

Drzewo BST można zaimplementować za pomocą struktury dowiązaniowej, gdzie każdy węzeł jest obiektem klasy posiadającym następujące pola:

- key (przechowuje klucz)
- value (przechowuje wartość)
- parent (wskaźnik do rodzica - w przypadku korzenia ustawione na null)
- left (wskaźnik do lewego syna - w przypadku liścia ustawione na null)
- right (wskaźnik do prawego syna - w przypadku liścia ustawione na null)

Przykładowe drzewo BST



(w węzłach pokazano tylko klucze)

Obserwacje:

- wolne miejsca oznaczone są wskaźnikami “null”
- drzewo nie musi być zupełne (tzn. wolne miejsca mogą być nie tylko na ostatnim poziomie)
- klucz minimalny (maksymalny) można znaleźć idąc od korzenia skrajnie w lewo (prawo) (tutaj jest to odpowiednio 3 i 20)

Implementacja operacji słownika uporządkowanego na BST

- search(key): zacznij od korzenia i dopóki nie dojdziesz do "null", porównuj key z kluczem bieżącego węzła: jeśli jest równy, to zwróć wartość przechowywaną w tym węźle; jeśli mniejszy to idź do lewego syna; jeśli większy to idź do prawego syna; (podobnie jak w przypadku algorytmu wyszukiwania binarnego – stąd nazwa); jeśli dojdziesz do "null", to szukanego klucza nie ma w drzewie
- insert(key, value): zacznij od korzenia i postępuj tak jak w operacji `search`, aż znajdziesz odpowiedni węzeł (wtedy uaktualnij wartość) lub null (wtedy wstaw nowy węzeł w miejsce null, odpowiednio podpinając wskaźnik do i od rodzica)
- delete(key): zacznij od korzenia i podobnie jak w search; gdy znajdziesz węzeł do usunięcia możliwe są 3 warianty (na nast. slajdzie)

Implementacja operacji słownika uporządkowanego na BST, c.d.

- minimum()/maximum(): zacznij od korzenia i idź skrajnie w lewo/prawo - ostatni węzeł przed "null" zawiera minimum/maksimum
- predecessor(key) (successor(key)): znajdź węzeł zawierający klucz key (używając search); jeśli węzeł ma lewego (prawego) syna, to poprzednik (następnik) stanowi maksimum (minimum) w jego lewym (prawym) poddrzewie; jesli nie ma takiego syna, to poprzednik (następnik) jest najbliższym przodkiem (w górę drzewa), z którego trzeba było zejść w prawo (lewo)

Warianty operacji delete

Po zidentyfikowaniu w drzewie BST węzła do usunięcia, następuje jeden z wariantów, w zależności od tego, ilu synów ma ten węzeł:

- nie ma synów: po prostu usuwamy węzeł i uaktualniamy wskaźnik od rodzica na "null"
- ma 1 syna: usuwamy węzeł i "podpinamy" jego jedynego syna (wraz z całym poddrzewem) do rodzica usuniętego węzła (odpowiednio aktualizując wszystkie niezbędne wskzaźniki)
- ma 2 synów: usuwamy węzeł i zastępujemy go węzłem x zawierającym klucz, który jest bezpośrednim następnikiem klucza usuwanego węzła; następnie podpinamy jedynego syna węzła x do rodzica węzła x;

Pseudokod operacji search


```
searchIterative(node, key): \\ wywołanie dla node == root
  while ((node != null) and (node.key != key))
    if (key < node.key) node = node.left
    else node = node.right
  return node

***

searchRecursive(node, key): \\ wywołanie dla node == root
  if ((node == null) or (node.key == key)) return node
  if (key < node.key) return search(node.left, key)
  else return search(node.right, key)
```

Minimum i Maximum


```
minimum(node): \\
  while (node.left != null) node = node.left
  return node

maximum(node): \\ wywołanie dla node == root
  while (node.right != null) node = node.right
  return node

successor(node):
  if (node.right != null) return minimum(node.right)
  p = node.parent
  while ((p != null) and (node == p.right)
    node = p
    p = p.parent
  return p
```

(predecessor analogicznie do successor)

Pseudokod operacji insert


```
insert(node, key):
  if (key < node.key) then
    if node.left == null:
      n = create new node with key
      node.left = n
    else: insert(node.left, key)
  else: // (key >= node.key)
    if node.right == null:
      n = create new node with key
      node.right = n
    else: insert(node.right, key)
```

Pseudokod operacji delete

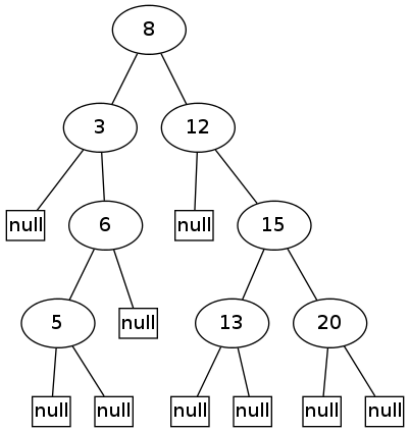

```
procedure delete(node, key)
  if (key < node.key) then
    delete(node.left, key)
  else if (key > node.key) then
    delete(node.right, key)
  else begin { key = node.key
    if node is a leaf then
      deletesimple(node)
   else
     if (node.left != null) then
       find x = the rightmost node in node.left
       node.key:=x.key;
       delete1(x);
     else
       proceed analogously for node.right
       (we are looking for the leftmost node now)
```

Example of a helper delete1 Implementation

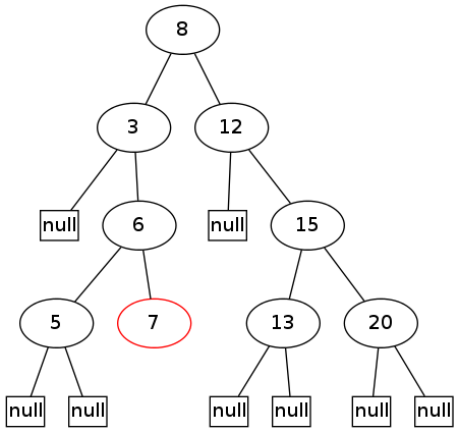

```
// delete1: for nodes having only 1 son

 procedure delete1(node)
 begin
    subtree = null
    parent = node.parent
    if (node.left != null)
      subtree = node.left
    else
      subtree = node.right

    if (parent == null)
     root = subtree
    else if (parent.left == node)  // node jest lewym synem
      parent.left = subtree
    else // node jest prawym synem
      parent.right = subtree
```

Przykład operacji insert

Oryginalne drzewo:

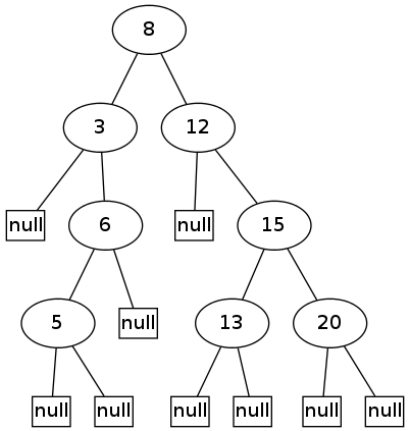


Drzewo po operacji `insert(7,...)`:

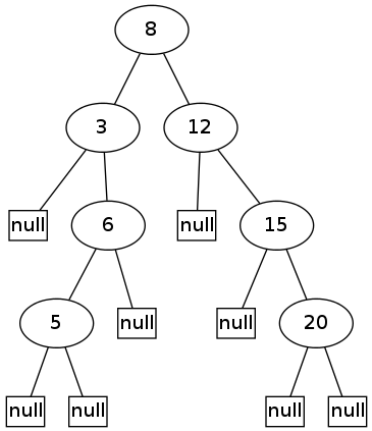


Przykład operacji delete: nie ma synów

Oryginalne drzewo:

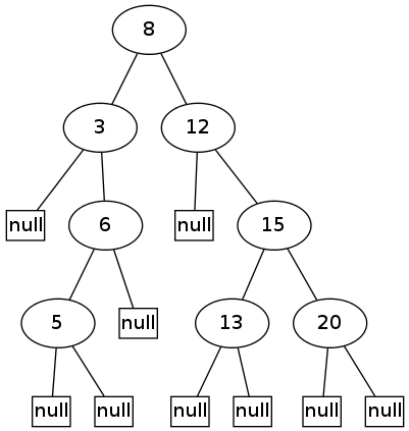


Drzewo po operacji `delete(13)`

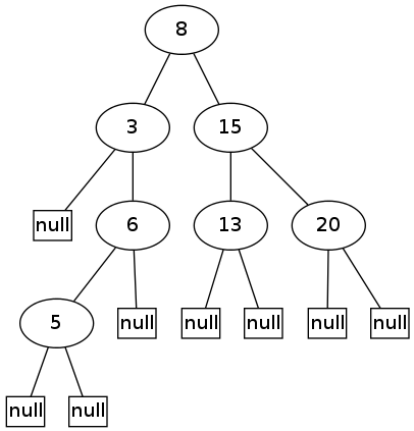


Przykład operacji delete: ma 1 syna

Oryginalne drzewo:

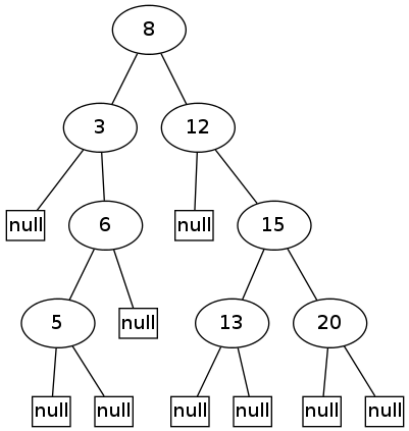


Drzewo po operacji `delete(12)`:

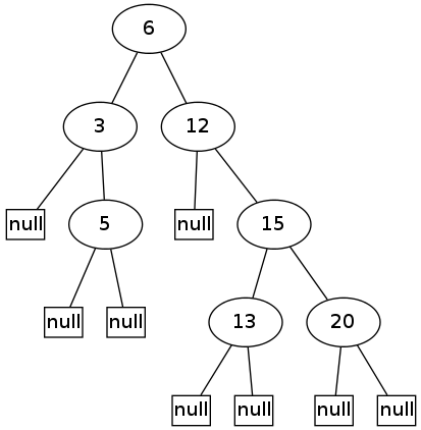


Przykład operacji delete: ma 2 synów

Oryginalne drzewo:



Drzewo po operacji `delete(8)`
(wariant z poprzednikiem – poprzednikiem 8 w tym drzewie było 6):



(wariant z następnikiem jako ćwiczenie)

Analiza przeciętnej złożoności czasowej operacji na BST

rozmiar danych: n - liczba elementów w drzewie BST; operacja dominująca: porównanie kluczy

Dla wszystkich operacji słownika uporządkowanego na drzewie BST, **liczba porównań jest proporcjonalna do wysokości drzewa** (zawsze przechodzimy od korzenia w dół + ewentualne dodatkowe operacje o koszcie stałym)
Można udowodnić, że **wysokość losowego drzewa BST**[3] **jest logarytmiczna względem n** ($O(log(n))$.
A zatem **przeciętna złożoność czasowa** wszystkich operacji na BST jest: **A(n)=O(log(n))**.

[3] czyli takiego, do którego klucze wstawiane są w losowym porządku, a dokładniej: każda permutacja kluczy przychodzących jest jednako prawdopodobna

Pesymistyczna złożoność czasowa operacji na BST

Niestety, ponieważ drzewo BST nie musi być zupełne, jego wysokość w pesymistycznym wypadku może wynosić $O(n)$ (np. jedna długa gałąź, bez odgałęzień).

Wynika z tego, że pesymistyczna złożoność czasowa operacji na BST jest niestety **liniowa** $W(n) = O(n)$.

Aby zaradzić temu problemowi, zaprojektowano wiele ulepszeń i rozszerzeń drzew BST, które gwarantują logarytmiczną pesymistyczną wysokość drzewa.

Drzewo AVL

Drzewo AVL (od nazwisk twórców: Adelson-Velskij, Landis) jest drzewem BST z dodatkowym warunkiem. Warunek ten wprowadza dla każdego węzła pojęcie **współczynnika zrównoważenia** (bf - od ang. balance factor).

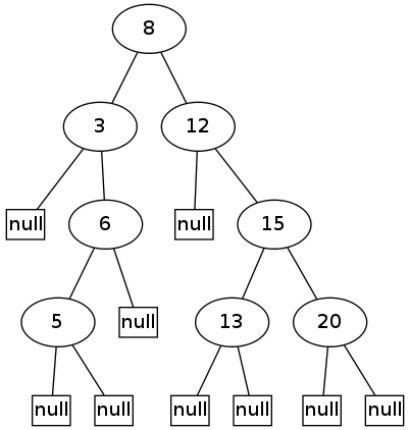
Współczynnik zrównoważenia dla węzła x zdefiniowany jest jako różnica wysokośći lewego poddrzewa węzła x i prawego poddrzewa węzła x.

Definicja AVL jest następująca: jest to drzewo BST, które dodatkowo ma własność, że w każdym węźle x, $bf(x) \in \{-1, 0, 1\}$, czyli że wysokości poddrzew nie mogą się różnić o więcej niż 1.

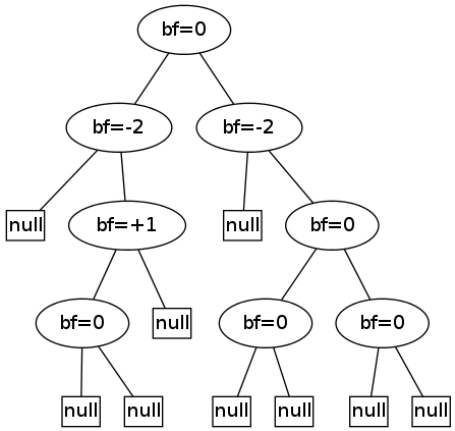
Można udowodnić, że pesymistyczna wysokość drzewa AVL wynosi **O(log(n)**, a więc wszystkie operacje słownika uporządkowanego na drzewie AVL będą miały również **pesymistyczną złożoność czasową logarytmiczną**

Przykład obliczania współczynników zrównoważenia

Przykładowe drzewo:



Drzewo z obliczonymi współczynnikami zrównoważenia (bf):



(drzewo to nie jest drzewem AVL, bo niektóre współczynniki są poza zbiorem dozwolonych wartości $\{-1, 0, 1\}$)

Implementacja drzewa AVL

Aby zapewnić warunek zrównoważenia, w drzewie AVL, po każdej operacji modyfikującej (przebiegającej tak samo jak dla drzewa BST) **dokonuje się sprawdzenia warunku zrównoważenia**, tak aby po operacji wszystkie węzły miały odpowiednie wartośći bf.

Aby to osiągnąć, przegląda się wszystkie współczynniki bf od zmodyfikowanego węzła **w górę** (tylko te mogą mieć zaburzone bf o najwyżej 1 po operacji insert lub delete) i w przypadku natrafienia na $bf == 2$ lub $bf = -2$, dokonuje się **naprawy tego fragmentu drzewa**. Naprawa w węźle x ma postać tzw. **rotacji**[4], która w prosty sposób przebudowuje lokalny fragment drzewa tak, aby przywrócić odpowiednie wartości bf. Rotacje są zaprojektowane tak, że koszt każdej z nich jest stały (O(1)) a w efekcie wszystkie operacje słownika uporządkowanego na drzewie AVL mają **pesymistyczną złożoność logarytmiczną**.

Zaproponowano też wiele innych "udoskonalonych" implementacji słownika uporządkowanego (np. drzewa samo-organizujące się, etc.) zapewniających logarytmiczną złożość czasową operacji.

[4] rotacje nie są omawiane w tym wykładzie

Podsumowanie: słowniki i słowniki uporządkowane

- tablice mieszające są efektywną implementacją słowników, ale nie zapewniają efektywnej (lepszej niż liniowa) implementacji niektórych operacji słownika uporządkowanego
- drzewo BST jest najprostszą implementacją słownika uporządkowanego zapewniającą efektywną przeciętną złożoność czasową operacji słownika uporządkowanego, ale niestety mają linową pesymistyczną złożoność czasową tych operacji
- drzewo AVL jest rozszerzeniem drzewa BST o warunek zrównoważenia i zapewnia logarytmiczną pesymistyczną złożoność czasową tych operacji
- zaproponowano też wiele innych konkretnych struktur danych efektywnie implementujących operacje słownika uporządkowanego (drzewa samo-organizujące się, B-drzewa, AB-drzewa, drzewa B+ i wiele innych)

Przykładowe pytania/problemy

- definicja słownika
- analiza naiwnych implementacji słownika
- tablice mieszające
- własności funkcji mieszającej
- analiza operacji słownika zaimplementowanych na tablicy mieszającej
- sposoby rozwiązywania kolizji
- definicja słownika uporządkowanego
- definicja i własności drzewa BST
- działanie i analiza operacji słownika uporządkowanego na drzewie BST
- mając dane drzewo BST, pokazać jak wygląda po konkretnej operacji insert, delete (we wszystkich wariantach)
- ograniczenia drzew BST
- definicja i opis motywacji oraz pomysłu drzewa AVL
- mając dane drzewo binarne, obliczyć współczynniki bf dla wszystkich węzłów i powiedzieć czy jest to drzewo AVL
- jak zapewnia się zrównoważenie drzewa AVL i jaka jest z tego korzyść?

Dziękuję za uwagę