Algorytmy i Struktury Danych, 11. ćwiczenia

2018-12-19

Spis treści

1 Join i Split na 2–3–4 drzewach

`http://courses.csail.mit.edu/6.046/spring04/handouts/ps5-sol.pdf`
Ogólnie o 2–3–4 drzewach:
`https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13b.pdf`

2 Przejście z 2–3–4 drzew na czerwono-czarne

- węzeł z jednym kluczem zamieniany jest na: czarny węzeł,
- węzeł z dwoma kluczami zamieniany jest na dwa węzły: czarny i jego czerwonego syna (możemy dowolnie wybrać lewy czy prawy),
- węzeł z trzema kluczami jest zamieniany na trzy węzły: czarny węzeł z dwoma czerwonymi synami.

3 B-drzewa definicja

- każdy węzeł ma następujące pola n, $c[]$, $key[]$,
- każdy węzeł wewnętrzny utrzymuje n kluczy i $n+1$ wskaźników do synów,
- klucze są uporządkowane rosnąco,

- klucz w poddrzewie $c[i]$ mają wartości pochodzą z przedziału $[key[i-1], key[i]]$ (definiujemy $key[0] = -\infty$, $key[n+1] = \infty$),
- wszystkie liście mają leżą na tej samej głębokości,
- każdy węzeł zawiera nie więcej niż $2t-1$ kluczy,
- każdy węzeł oprócz korzenia zawiera co najmniej $t-1$ kluczy.

B-drzewa usuwanie

- jeśli klucz k jest w węźle x i x jest liściem, to usuń k z węzła,
- jeśli klucz k jest w węźle wewnętrznym x, to:
  - niech y_1 syn x poprzedzający k, y_2 syn x występujący po k, k_1 poprzednik k w drzewie, k_2 następnik k w drzewie,
  - jeśli węzeł y_1 ma co najmniej t kluczy, to rekurencyjnie usuń k_1 i zastąp k przez k_1,
  - w przeciwnym przypadku, jeśli węzeł y_2 ma co najmniej t kluczy, to rekurencyjnie usuń k_2 i zastąp k przez k_2,
  - w przeciwnym przypadku, y_1 i y_2 mają po $t-1$ kluczy, scal węzeł y_1, klucz k i węzeł y_2 otrzymując węzeł y', usuń rekurencyjnie k z węzła y.
- jeśli klucz k nie występuje w węźle wewnętrznym x, to:
  - znajdź odpowiednie poddrzewo y w którym może znajdować się k,
  - jeśli y ma co najmniej t kluczy, usuń rekurencyjnie k z y,
  - wpp., jeśli y ma $t-1$ kluczy, ale jeden z sąsiadów y ma t kluczy, to dodaj jeden klucz do y (jeden klucz przechodzi z x do y, jeden z brata y do x),
  - wpp., scal y z dowolnym bratem i usuń k z tak utworzonego węzła (jeśli x jest korzeniem, to może to spowodować zmniejszenie wysokości drzewa).

4 Pokazać, że przy pomocy rotacji można zawsze przejść z jednego BST do drugiego

Za pomocą rotacji możemy dowolne drzewo BST zamienić na listę (i na odwrót). Wstarczy tak długo jak drzewo zawiera węzeł z lewym synem, wykonać na nim (i lewym synie) prawą rotację.

5 ASD Zadania — wzbogacanie struktur danych

Zadanie 3.24

Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie $O(\log n)$ następujących operacji na zbiorze S:

- $makeset(S)$ :: $S := \emptyset$
- $insert((x, y), S)$ :: $S := S \cup \{(x, y)\}$
- $minx(S)$ :: usunięcie z S pary (x, y) o najmniejszej pierwszej składowej,
- $miny(S)$ :: usunięcie z S pary (x, y) o najmniejszej drugiej składowej,
- $searchx(x, S)$ :: wyznaczenie takiej pary $(a, b) \in S$, że $x = a$,
- $searchy(y, S)$ :: wyznaczenie takiej pary $(a, b) \in S$, że $y = b$.

Rozwiązanie: dwa drzewa AVL (jedno ze współrzędnymi x, drugie ze współrzędnymi y), dodatkowo każdy węzeł trzyma dowiązanie do odpowiadającego mu węzła w drugim drzewie.

Zadanie 3.25

Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie $O(\log n)$ następujących operacji na zbiorze S:

- $construct(S)$ :: utworzenie ciągu pustego S,
- $insert(S, x)$ :: $S := S \cup \{(x)\}$,
- $delete(S, x)$ :: $S := S - \{(x)\}$,
- $search(S, x)$ :: sprawdzenie, czy x znajduje się w zbiorze S,
- $elem(S, i)$ :: wyznaczenie i–tego co do wielkości elementu zbioru S,
- $numb(S, x)$ :: wyznaczenie numeru elementu x w zbiorze S (względem wielkości).

Rozwiązanie: drzewo AVL z atrybutami rozmiar poddrzewa.

Zadanie 3.26

Zaprojektuj strukturę danych do wykonywania ciągów następujących operacji (dla elementów x pochodzących z dowolnego zbioru liniowo uporządkowanego):

- $initialization$ :: $S_i = \emptyset$ dla $i = 1, 2, \ldots, n$,
- $insert(S_i, x)$ :: $S_i := S_i \cup \{(x)\}$, pod warunkiem, że x nie występuje w żadnym zbiorze S_j, $1 \leq j \leq n$,
- $deletemin(S_i)$ :: usunięcie ze zbioru S_i najmniejszego elementu,
- $find(x)$ :: wyznaczenie numeru zbioru do którego należy element x.

Rozwiązanie: S_i jako zwykłe kopce, dodatków utrzymujemy słownik par $(x, numerzbioru)$

Zadanie 3.27

Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie $O(\log n)$ następujących operacji na ciągu S:

- $construct(S)$ :: utworzenie ciągu pustego S,
- $insert(S, i, x)$ :: wstawienie x na i-te miejsce w ciągu S, tzn. $S_i = x$ pod warunkiem, że $i \leq |S| + 1$,
- $sum(S, i, j)$ :: obliczenie sumy $\sum_{k=i}^{j} S_k$,

Rozwiązanie: AVL z dodatkowym atrybutem suma elementów poddrzewa.

Zadanie 3.28

Rozwiązanie: AVL z atrybutem rozmiar poddrzewa.

Zadanie 3.29

Zaprojektuj strukturę danych umożliwiającą wykonywanie w czasie $O(\log n)$ następujących operacji na zbiorze S zawierającym przedziały liczb rzeczywistych $[l, r]$:

- $empty(S)$ :: $S = \emptyset$,
- $add(S, I)$ :: $S = S \cup \{I\}$,
- $delete(S, I)$ :: $S = S - \{I\}$,
- $is(S, x)$ :: sprawdzenie czy element x należy do jakiegoś przedziału w zbioru S;
- $intersect(S, I)$ :: sprawdzenie czy przedział I ma niepuste przecięcie z jakimś przedziałem należącym do S.

Rozwiązanie: Utrzymujemy słownik z parami (x, z) (gdzie x to liczba rzeczywista, a z +1 lub -1). Dodatkowo każdy węzeł ma dodatkowy atrybut *suma* oznaczający sumę wartości z w poddrzewie. Możemy w takim drzewie w czasie $O(\log n)$ obliczyć $sum(q)$ oznaczającą sumę wszystkich atrybutów z par (x, z), takich, że $x \leq q$.

- $add(S, I)$ – dodajemy do słownika pary $(l, +1)$ i $(r, -1)$,
- $delete(S, I)$ – usuwamy ze słownika pary $(l, +1)$ i $(r, -1)$,
- $is(S, x)$ – jeśli słownik zawiera pary $(x, +1)$ lub $(x, -1)$ to zwracamy $true$, wpp. obliczamy $sum(x)$ i jeśli suma jest > 0 to zwracamy $true$, jeśli $sum(x) \leq 0$, to zwracamy $false$.
- $intersect(S, I)$, jeśli $is(S, l)$ lub $is(S, r)$ to zwracamy $true$, jeśli istnieje w słowniu para (x, z), t.że $l \leq x \leq r$, to zwracamy $true$, wpp zwracamy $false$.

6 Klasówka 2007 (1), zadanie 1

Opracuj strukturę danych, która pozwala wykonywać następujące operacje:

- Ini(k):: inicjacja struktury danych i ustalenie długości krotek liczb całkowitych na k
- Insert($< a_1, a_2, \ldots, a_k >$):: dodaje do struktury krotkę $< a_1, a_2, \ldots, a_k >$
- Min:: podaje najmniejszą leksykograficznie krotkę w strukturze
- ExtractMin:: usuwa najmniejszą leksykograficznie krotkę ze struktury

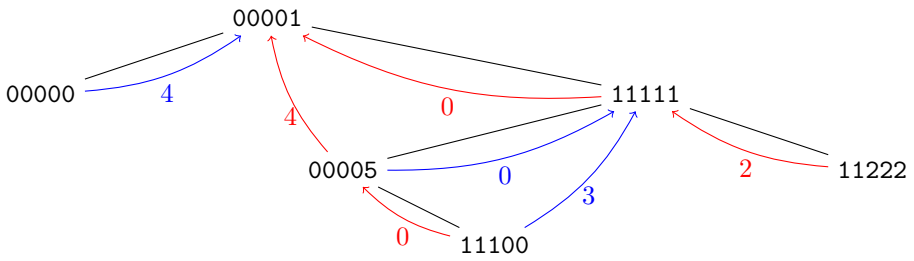
W Twoim rozwiązaniu operacje Insert i ExtractMin powinny by wykonywane w czasie $O(\log n + k)$

Rozwiązanie:

Jako bazową strukturę będziemy używać drzew Czerwono-czarnych. Dzięki temu wysokość struktury nie będzie przekraczać $O(\log n)$, oraz co później okaże się istotne liczba rotacji czy wstawianiu/usuwaniu kluczy jest rzędu $O(1)$.

W każdym węźle utrzymujemy następujące informacje:

- *key*: krotkę $< a_1, \ldots, a_k >$,
- p_{left} wskaźnik do najbliższego przodka będącego po lewej stronie,
- p_{right} wskaźnik do najbliższego przodka będącego po prawej stronie,
- lcp_{left} najdłuższy wspólny prefiks pomiędzy kluczem z węzła i kluczem z p_{left},
- lcp_{right} najdłuższy wspólny prefiks pomiędzy kluczem z węzła i kluczem z p_{right},



Rysunek 1: Przykładowe drzewo dla krotek $k = 5$, wskaźniki p_{left} zaznaczone są na czerwono, wskaźniki p_{right} na niebiesko. Wartości lcp_{left} i lcp_{right} zaznaczone są jako etykiety krawędzi.

W jaki sposób aktualizować takie dodatkowe atrybuty?

- dodawanie nowego liścia – musimy wyznaczyć atrybuty p_{left} i p_{right} (w czasie $O(\log n)$), następnie obliczamy wartości lcp_{left} i lcp_{right} (w czasie $O(k)$),

- usuwanie skrajnie prawego węzła (najmniejszy klucz w drzewie) – jeśli węzeł jest liściem to nie musimy nic zrobić, jeśli ma prawego syna (z własności drzew czerwono czarnych prawy syn musi być już liściem) to poprawiamy w nim wartości p_{left} i p_{lcp} (zmieniając je na NULL)

- rotacje – jeśli wykonujemy rotację węzłów x i $y = parent(x)$ to zauważamy, że musimy jedynie zaktualizować część atrybutów z x i y, wartości atrybutów w pozostałych węzłach zostają bez zmian. Koszt aktualizacji pojedynczego węzła wynosi $O(k)$.

Jak zrealizować operację $Insert$ w czasie $O(\log n + k)$?

Musimy pokazać, że dzięki dodatkowym atrybutom *lcp* uda nam się znacząco przyspieszyć porównywanie krotki z węzłami podczas przechodzenia po drzewie od korzenia do miejsce w którym nowa krotka powinna być wstawiona. Dzieje się, tak dlatego, że dzięki wartościom *lcp* możemy często w czasie $O(1)$ porównać wstawianą krotkę z wartością w węźle.

Algorithm 1: Insert(r, $A =< a_1, \ldots, a_k >$)

```
v := r; lcp_l = NULL; lcp_r = NULL
while v ≠ NullNode do
    niech p_l = p_left(v), p_r = p_right(v)
    niezmiennik: lcp_l = NULL lub LCP(A, key(p_l)) = lcp_l ≤ lcp_left(v)
    niezmiennik: lcp_r = NULL lub LCP(A, key(p_r)) = lcp_r ≤ lcp_right(v)
    if lcp_l != NULL and lcp_l < lcp_left(v) then
        v = right(v)
    else if lcp_r != NULL and lcp_r < lcp_right(v) then
        v = left(v)
    else
        l = max(coalesce(lcp_l, 0), coalesce(lcp_r, 0))
        while l < k and a_{l+1} = key(v)[l + 1] do
            l = l + 1
        if l = k then
            krotka już istnieje w drzewie
        else if a_{l+1} < key(v)[l + 1] then
            v = left(v); lcp_r = l
        else if a_{l+1} > key(v)[l + 1] then
            v = right(v); lcp_l = l
```