

Algorytmy i Struktury Danych, 9. ćwiczenia – rozwiązania zadań serii 7 BIS (wersja 1.0.1)

2024-12-04

Zadanie 7.1

Implementacja słownika. a) Dwuwymiarowa tablica. b) Tablica z rosnącymi blokami.

Rozwiązanie:

a) <https://en.wikipedia.org/wiki/Beap>

Szukaj(x):: zaczynamy w skrajnie prawy elemencie pierwszego wiersza $(k, 1)$, będąc w węźle (i, j) jeśli $a[i - 1, j] \geq x$ to przejdź do $(i - 1, j)$ wpp. przejdź do $(i, j + 1)$.

Wstaw(x):: wstawiamy nową wartość w wolne miejsce na ostatniej przekątnej, po czym wykonujemy operację analogiczną do heapUp, czyli jeśli $a[i, j] < \min(a[i - 1, j], a[i, j - 1])$ to zamieniamy $a[i, j]$ z większą z wartości $a[i - 1, j], a[i, j - 1]$ i kontynuujemy poprawienie.

Usuń(x):: zastępujemy usuwaną wartość używając y — ostatniego klucza z ostatniej przekątnej, jeśli $x \geq y$ to postępujemy analogicznie jak przy wstawianiu, jeśli $x \leq y$ to wykonujemy operację analogiczną do heapDown, czyli jeśli $a[i, j] > \min(a[i + 1, j], a[i, j + 1])$ to zamieniamy $a[i, j]$ z mniejszą z wartości $a[i + 1, j], a[i, j + 1]$ i kontynuujemy poprawienie.

b) Dla bloku zawierającego przesunięty cyklicznie ciąg rosnący, możemy w czasie $O(\log n)$:

- znaleźć minimum
- znaleźć maksimum
- sprawdzić czy blok zawiera dany klucz x

Szukaj(x):: czas $O(\log^2 n)$, za pomocą wyszukiwania binarnego znajdź blok taki, że $\min(B_i) \leq x \leq \max(B_i)$. Sprawdź czy B_i zawiera x .

Szukaj(x):: czas $O(\log n)$, za pomocą wyszukiwania binarnego znajdź dwa sąsiednie bloki B_i i B_{i+1} , które mogą zawierać klucz x a następnie sprawdź w czasie $O(\log n)$ czy klucz występuje w którymś z nich.

Korzystamy z faktu, że dla dowolnego elementu y z bloku B_j :

- jeśli $y < x$ to element x na pewno nie występuje w $B_1, \dots, B_{j-1}$,
- jeśli $y > x$ to element x na pewno nie występuje w $B_{j+1}, B_{j+2}, \dots$,

Te obserwacje pozwalają kontynuowanie wyszukiwania binarnego tak długo jak zares składa się z co najmniej trzech bloków. Ponieważ możemy porównywać z dowolnym elementem z bloku (np. pierwszym) i unikamy ustalania min / max, to koszt porównania x z jednym blokiem to $O(1)$.

Wstaw(x):: czas $O(\sqrt{n} \log n)$, za pomocą wyszukiwania binarnego znajdź blok taki, że $\min(B_i) \leq x \leq \max(B_i)$, wstaw x do bloku B_i przesuwając pozostałe elementy, wypchnij $\max(B_i)$ do bloku B_{i+1} (co spowoduje reakcję łańcuchową i przesłanie dalej $\max(B_j)$). Ponieważ dopuszczamy przesunięcia cykliczne w blokach stąd obsługa każdego z $\max(B_i), \max(B_{i+1}), \dots$ wymaga tylko czasu $O(\log n)$.

Usuń(x):: analogicznie jak wstawianie, ale brakujący element zastępujemy przez $\min(B_{i+1})$.

Zadanie 7.2

Danych jest n par liczb całkowitych, które się różnią na każdej pozycji. Pierwsze elementy par to klucze, zaś drugie to priorytety. Innymi słowy, mamy n różnych kluczy i n różnych priorytetów.

- Jednoznaczność drzewa które jest zarówno BST jak i kopcem typu MAX.
- Konstrukcja drzewa z a) poprzez $O(n)$ rotacji jeśli drzewo wejściowe jest BST.
- Algorytm znajdujący w czasie liniowym ciąg rotacji dla b).

Rozwiązanie:

a)

- Korzeń = element r z największym priorytetem.
- Jedno z dzieci korzenia = element e z największym priorytetem spośród pozostałych elementów. Jeśli $r.key > e.key$, to e musi być lewym dzieckiem, w innym przypadku prawym dzieckiem.
- Drugie dziecko = element f z największym priorytetem spośród elementów z kluczem mniejszym niż $r.key$ (jeśli f jest lewym dzieckiem, tzn. jeśli $r.key < e.key$) lub w innym przypadku spośród elementów z kluczem większym niż $r.key$.

Wybór korzenia jest jednoznaczny. Wybór dzieci też jednoznaczny. Rekurencyjnie pokazujemy jednoznaczność wyboru pozostałych wierzchołków. (Nota bene: Czasem korzeń może mieć tylko jedno dziecko, np. drzewo może okazać się ścieżką.)

b) Dowolne drzewo BST możemy za pomocą co najwyżej $n - 1$ rotacji przekształcić w listę (tzn. drzewo, którego każdy węzeł wewnętrzny ma tylko prawego syna). Niech S_T to sekwencja rotacji przekształcająca T w listę, niech S_{MAX} to sekwencja rotacji przekształcająca T_{MAX} w listę, rozwiązaniem będzie $S_T + reverse(S_{max})$.

c)

1. Rozwiązanie: Jak punkt b (czyli wygeneruj sekwencję rotacji zamieniającą drzewo BST na listę i dodaj odwróconą sekwencję rotacji zamieniającą kopiec BST na listę). Musimy pokazać jednak liniowy algorytm, który dla zadanego drzewa BST wygeneruje kopiec:

Algorytm 1: Obliczanie Kopca

Input: drzewo BST o kluczach $k_1, \dots, k_n$ i priorytetach $p_1, \dots, p_n$

Output: Drzewo K z porządkiem BST wg kluczy $k_1, \dots, k_n$ i porządkiem kopcowym (MAX) wg priorytetów $p_1, \dots, p_n$

K = drzewo o jednym węźle (k_1, p_1)

foreach $i \in \{2, \dots, n\}$ **do**

 niech $v_1, \dots, v_k$ skrajnie prawa ścieżka w drzewie K ($v_1 = root(K)$)

$j = k$

while $j > 0$ **and** $p_i > p(v_j)$ **do**

$j = j - 1$

 wstaw do K nowy węzeł (k_i, p_i) jako prawe dziecko v_j (jeśli $j = 0$

 jako nowy korzeń), lewy syn to v_{j+1} (jeśli $j < k$)

2. Rozwiązanie: Algorytm na dwa kroki:

1. Zamieniamy wejściowe BST za pomocą rotacji na ścieżkę skierowaną całkowicie na prawo.
2. Zaczynając od korzenia, wykonujemy po kolei dla każdego wierzchołka Up-Heap używając do tego rotacji.

Dowód poprawności : Ponieważ używaliśmy tylko rotacji, to wyjściowe drzewo jest nadal BST. Krok 2) zapewnia nam że drzewo jest też kopcem.

Trzeba jeszcze pokazać że kroki 1) i 2) możemy zrobić w czasie $O(n)$. Ponieważ każdy krok naszego algorytmu to rotacja, wystarczy pokazać że wykonujemy w sumie $O(n)$ rotacji.

Krok 1): Niech π będzie ścieżką zaczynającą się przy korzeniu i idącą maksymalnie tylko w prawo. (Z początku może być że π jest tylko korzeniem.) Dopóki π nie zawiera wszystkie wierzchołki, to musi zawierać jakiś wierzchołek v z lewym dzieckiem. Rotujemy w prawo wokół v . Przez to π staje się dłuższe o jeden wierzchołek. Więc po najwyżej $n - 1$ rotacjach π zawiera wszystkie wierzchołki.

Krok 2): Up-heap dla danego wierzchołka v działa w ten sposób że tak długo jak priorytet ojca v jest mniejszy, to wykonujemy rotację w lewo wokół tegoż ojca. Skutkuje to tym że ojciec stanie się dzieckiem v (bo v był prawym dzieckiem ojca), a dziadek wierzchołka v stanie się nowym ojcem v . Ponieważ wszyscy (aktualni) przodkowie v leżą na ścieżce π , to v będzie prawym dzieckiem nowego ojca.

Obserwujemy że podgraf oparty na wierzchołkach, dla których już wywołaliśmy Up-Heap, jest zarówno drzewem BST jak i kopcem. Ponadto przy każdej rotacji ścieżka π maleje o jeden wierzchołek a nowe już nie dochodzą. Więc w sumie nie może być więcej niż $n - 1$ rotacji.

Zadanie 7.3

Pokaż, w jaki sposób wykonywać wydajnie operacje Join i Split na AVL-drzewach wyszukiwań binarnych:

Rozwiązanie:

Algorytm 2: Join(T, x, T') w czasie $O(\log n)$

WLOG $height(T) \geq height(T')$

Zlokalizuj na skrajnie prawej ścieżce T wierzchołek v , taki, że

$height(T_v) = height(T')$

zastąp T_v przez drzewo o korzeniu z kluczem x , lewym synem T_v i prawym synem T'

popraw zbalansowanie wierzchołków na ścieżce od x do korzenia T .

Jeśli dokładniej przeanalizujemy czas działania *Join* to zauważmy, że ta operacja działa w czasie $O(1 + |h(T) - h(T')|)$.

Split(T, x) w czasie $O(\log n)$:: Zlokalizuj węzeł x , po jego usunięciu drzewo rozpada się na $O(\log n)$ drzew (i pojedynczych kluczy) mniejszych / większych od x . Te dwie rodziny drzew można scalać w $T_{<x}$, $T_{>x}$ używając Join (uwaga scalamy drzewa w kolejności od najmniejszych do największych).

Zadanie 7.4

Zaprojektuj strukturę danych dla dynamicznego zbioru domkniętych przedziałów liczbowych S , umożliwiającą wydajne wykonywanie operacji:

Search($S, [a, b]$):: podaj wskaźnik do wystąpienia $[a, b]$ w S ; jeśli $[a, b]$ nie ma w S odpowiedzią jest NULL

Insert($S, [a, b]$):: $S := S \cup \{[a, b]\}$

Delete($S, [a, b]$):: $S := S \setminus \{[a, b]\}$

Intersect($S, [a, b]$):: sprawdź, czy w S jest przedział z niepustym przecięciem z $[a, b]$

Rozwiązanie: Search/Insert/Delete można zaimplementować za pomocą zwykłego drzewa które utrzymuje pary (a, b) w porządku leksykograficznym.

Operacja Intersect:

Utrzymujemy dodatkowe drzewo, które będzie utrzymywało krotki (x, f) (z porządkiem leksykograficznym), gdzie x to współrzędna na osi X a f to flaga $\{0, 1\}$. Każdej krotce przypisana jest jej wartość $wart$. Jeśli w trakcie działania algorytmu wartość jakiejś krotki będzie równa 0, to jest ona usuwana z drzewa. Dodatkowo wzbogacamy drzewo dwa dodatkowe atrybuty: $suma$ - suma wszystkich wartości z poddrzewa (łącznie z wartością bieżącego węzła), $maxPref$ - maksymalna suma z prefiksu wszystkich wartości z całego poddrzewa. Dodatkowe atrybuty możemy aktualizować za pomocą wzorów:

$$suma(v) = suma(left(v)) + wart(v) + suma(right(v))$$

$$maxPref(v) = \max \begin{cases} maxPref(left(v)) \\ suma(left(v)) + wart(v) \\ suma(left(v)) + wart(v) + maxPref(right(v)) \end{cases}$$

Gdy do struktury danych dodawany jest nowy przedział $[a, b]$ to zmieniamy wartość krotek:

- dla krotki $v = (a, 0)$ zmieniamy $wart(v) := wart(v) + 1$, jeśli taka krotka nie istnieje to ją dodajemy (z wartością 1)
- dla krotki $v = (b, 1)$ zmieniamy $wart(v) := wart(v) - 1$, jeśli taka krotka nie istnieje to ją dodajemy (z wartością -1)

Analogicznie przy usuwaniu przedziału $[a, b]$:

- dla krotki $v = (a, 0)$ zmieniamy $wart(v) := wart(v) - 1$, jeśli taka krotka nie istnieje to ją dodajemy (z wartością -1)
- dla krotki $v = (b, 1)$ zmieniamy $wart(v) := wart(v) + 1$, jeśli taka krotka nie istnieje to ją dodajemy (z wartością +1)

Uwaga! przy aktualizacji krotek musimy pamiętać aby usuwać te węzły drzewa których wartość $wart(v) = 0$.

Za pomocą dodatkowych atrybutów jesteśmy w stanie w czasie $O(\log n)$ zaimplementować następujące operacje:

- $suma(x, f)$ - suma wartości wszystkich krotek nie większych niż (x, f) , możemy zauważyć, że w naszym przypadku $suma(x, 0)$ oznacza liczbę przedziałów które zawierają wartość x ,
- $maxPref(x_0, f_0, x_1, f_1)$ - maksymalna suma prefiksowa wartości wszystkich krotek nie mniejszych niż (x_0, f_0) i nie większych niż (x_1, f_1)

Operacja $Intersect(a, b)$ sprowadza się do warunku logicznego:

$$maxPref(a, 0, b, 0) > 0$$

Ewentualnie można też sprawdzać, czy $suma(a, 0) > 0$ lub $suma(b, 0)$, lub istnieje krotka $(x, 0) : a \leq x \leq b$ taka, że $wart((x, 0)) > 0$.

Zadanie 7.5

Zaprojektuj strukturę danych dla dynamicznego ciągu liczbowego $x_1, x_2, \dots, x_n$ umożliwiającą wydajne wykonywanie następujących operacji:

Ini():: zainicjuj ciąg jako pusty

Delete(i):: usuń i -ty element ciągu

Insert(i,a):: wstaw liczbę a jako i -ty element ciągu

Find(i):: wskaż i -ty element ciągu

ParitySum():: podaj sumę wszystkich elementów na pozycjach parzystych w ciągu

Rozwiązanie: Drzewo czerwono-czarne z elementami wstawianymi wg. liczby elementów w poddrzewie. Dodatkowe atrybuty:

- klucz (ale bez porządku BST),
- liczba elementów w poddrzewie,
- suma elementów o indeksach parzystych w poddrzewie,
- suma elementów o indeksach nieparzystych w poddrzewie,

Zadanie 7.6

Niech A będzie skończonym, dynamicznie zmieniającym się ciągiem, którego elementami są liczby ze zbioru $\{-1, 0, 1\}$. Podciąg kolejnych elementów A nazwiemy dobrym, gdy suma jego elementów jest równa 0. Podciąg jest super-dobry, gdy jest dobry i suma elementów w każdym jego prefiksie jest nieujemna.

Przykład: W ciągu $A = [1, 1, 0, -1, 0, 0, 1, -1, 1, 1, -1]$, podciąg $-1, 1, 1, -1$ jest dobry, ale nie super-dobry. Super-dobrym podciągiem jest na przykład $1, 0, -1$.

- Zaproponuj algorytm, który w czasie liniowym obliczy długość najdłuższego super-dobrego podciągu danego ciągu A .
- Zaproponuj strukturę danych, która pozwoli na wydajne wykonywanie następujących operacji na A :

Ini(A):: $A := []$; wykonywana tylko raz, na początku

Insert(A,e,i):: wstaw nowy element e jako i -ty element w A , $1 \leq i \leq |A| + 1$

Delete(A,i):: usuń i -ty element z A , $1 \leq i \leq |A|$

SuperGood(A,i,j):: sprawdź, czy podciąg $A[i, \dots, j]$ jest super-dobry, $1 \leq i \leq j \leq |A|$

Rozwiązanie:

a) Niech $psum[i] = \sum_{j=1}^i a[j]$.

Obserwacje:

- maksymalne (w sensie zawierania) ciągi superdobre nie mają części wspólnej,
- jeśli $a[i..j]$ jest maksymalnym ciągiem superdobrym, to $i = 1$ lub $a[i-1] = -1$,
- jeśli $a[i..j]$ jest maksymalnym ciągiem superdobrym, to $i = j$ lub $psum[j-1] > psum[j]$,

Niech $A_x = \{i : psum[i] = x\}$, dla każdego x możemy w czasie $O(|A_x|)$ sprawdzić maksymalne ciągi superdobre $a[i..j]$, dla $i, j \in A_x$. Ponieważ $\sum_x |A_x| = n$ to cały algorytm ma złożoność $O(n)$.

b) Wzbogacamy węzły drzewa o: *count*:: liczba elementów w poddrzewie, *sum*:: suma elementów w poddrzewie, *minpsum*:: minimalna suma prefiksowa ciągu elementów w poddrzewie,

Każdy z tych atrybutów możemy aktualizować w czasie $O(1)$:

- $count(v) = 1 + count(left(v)) + count(right(v))$
- $sum(v) = key(v) + sum(left(v)) + sum(right(v))$
- $minpsum(v) = \min \begin{cases} minpsum(left(v)) \\ sum(left(v)) + key(v) \\ sum(left(v)) + key(v) + minpsum(right(v)) \end{cases}$