AISD lista 7

Dominik Szczepaniak

May 28, 2024

1 Zadanie 1

Ponieważ w naszej operacji Union(A, B) przypisujemy A jako ojca B, to mamy O(1). Jeśli wszystkie Union są przed find to najpierw wykonamy wszystkie Uniony w złożoności O(1).

Teraz mamy same operacje Find(x). Załóżmy, że elementów jest n. Załóżmy, że wszystkie wierzchołki są na początku odznaczone. Zaznaczamy wierzchołek tylko gdy przejdziemy przez niego jakimś findem (również gdy szliśmy z kogoś niżej do góry). W takim razie widać, że jeżeli nie ma żadnych unionów po wykonaniu finda, to jeśli wierzchołek będzie zaznaczony raz to będzie od razu podpięty pod ojca zbioru. Czyli każdy wierzchołek możemy zaznaczyć co najwyżej raz każdym findem, czyli mamy złożoność co najwyżej O(n).

Union - O(n) + Find O(n) = O(2n) = O(n)

2 Zadanie 2

Możemy użyc drzewa Splay. Jeśli mamy insert to normalnie sobie insertujemy. Jeśli mamy min(i) to splayujemy pierwsza wartość większą niż i, i usuwamy lewe drzewo. Jak mamy deletemin to po prostu idziemy na maxa na lewo i usuwamy lewą wartość. Jeśli tutaj będziemy mieć długą ścieżkę w prawo to nam to nic nie wadzi, bo zawsze usuwamy minimalny element, więc usuniemy po prostu korzeń.

Problem - możemy usuwać i inserować ostatni element na zmiane i mamy ciągle złożoność O(n), więc łącznie $O(n^2)$

==========

Generalnie to jeżeli każda liczba może być dodana maksymalnie raz, to wszystkich liczb jest maksymalnie n.

W takim razie operacji deleteMin oraz Min nie może być więcej niż n, bo każda liczba jest dodana maksymalnie raz, a te operacje muszą usunąć przynajmniej jedną liczbę.

W takim razie możemy zrobić zwykłe drzewo AVL / CzerwonoCzarne które normalnie sobie insertuje liczby a później usuwa albo najmniejszą liczbę albo wszystkie liczby po kolei. Wtedy mamy złożoność O(nlogn).

==========

Dla każdego zbioru chcemy sobie zachować mina ktory obowiazuje

Czyli dla sigma1 min D

sigmie 1 przypisujemy tego min

ale dla sigma1 D min

nie przypisujemy mina nikomu

jesli poprzedni min byl mniejszy niz obecny to jest spoko, a jesli byl wiekszy to bedziemy go i tak juz dobrze pamietac

Niech k będzie ilością deleteMin

Wtedy mamy ciąg $sigma = sigma_1 Esigma_2 Esigma_3 ... sigma_k Esigma_k+1$, gdzie każda $sigma_j$ 1 <= j <= k+1 to ciąg insertów.

Czyli ostatni zbiór trzymamy na "śmieci" - wartości już znalezione lub te których i tak nigdy nie będzie.

Za każdym razem gdy wyjmiemy liczbe ze zbioru to możemy ją wrzucić do zbioru "śmieci"

Inicjalizujemy ciąg zbiorów dla union finda tak, żeby set nazywający się j zawierał liczby z $sigma_j$

Do tego trzymamy dwie tablice PRED i SUCC które tworzymy do zrobienia podwójnie skierowanej linked listy posortwanej dla tych wartości j dla których zbiory nazwane j istnieją.

Na początku PRED[j] = j-1 dla 1 <= j <= k+1 i SUCC[j] = j+1 dla 0 <= j <= k.

for i = 1 until n do:

j = find(i)

if j != m+1:

extracted[j] = i

let l be smallest value greater than j for which set K_l exists

$K_l = union(K_j, K_l)$ i usuwamy K_j

Dowód:

Załóżmy, że extracted jest nieprawidłowe. Niech x = extracted[j] będzie najmniejszą wartością dla której extracted[j] jest nieprawidłowe. Niech prawidłowa wartość będzie w tablicy "correct" i niech ta wartość nazywa się y. Mamy dwie możliwości - x $<$ y oraz x $>$ y.

a) Niech x $>$ y. Wtedy y nie może wystąpić w extracted, bo inaczej byłoby najmniejszą wartością dla której wynik jest nieprawidłowy. Ponieważ już przepracowaliśmy y przed przepracowaniem x musi być w zbiorze m+1. Ale jeśli correct[j] = y to y musi być gdzieś w K_i gdzie i $<$ j. Ponieważ extracted[j] nie miało wartości gdy było procesowane y, to nie moglibyśmy wrzucić wartości y do zbioru m+1, bo łączymy się tylko z setami większymi od nas i nie zrobiliśmy uniona z K_j, więc x $>$ y nie może zachodzić.

b) Załóżmy, że x $<$ y. Mówimy, że element x musi wystąpić w tablicy correct. Oczywiście x musi wystąpić przed j-tym wyjęciem w oryginalnym wejściu ponieważ nasz algorithm nigdy nie przesuwa zbiorów elementów wstecz. Jeśli nie wyodrębniliśmy x do j-tej selekcji to optymalne rozwiązanie powinno wybrać x zamiast y dla j-tej selekcji, ponieważ x jest mniejszy. Dlatego optymalne rozwiązanie musiało wyodrębnić x dla jakiegoś i $<$ j. Ale to oznacza, że extracted[i] trzyma jakieś z $>$ x. Z podobnych powodów jak powyżej nie mogliśmy przesunąć x poza zbiór K_i, ponieważ extracted[i] byłoby puste w momencie w którym x został wybrany. Więc ponieważ łączymy się tylko ze zbiorami powyżej nas i K_i nie został jeszcze połączony nie możemy umieścić x w extracted[j] przed extracted[i], dlatego nie możemy mieć x $<$ y.

W takim razie extracted[j] = correct[j], więc algos jest poprawny.

Złożoność:

Robimy n setów make-set

Robimy uniony n-m razy (gdzie m to liczba deleteMin)

Dla każdego seta trzymamy 3 wartości - number, prev, next (trzyma je każdy, ale potrzebuje je trzymac tylko przedstawiciel)

number mówi o j w K_j i może być łatwo ustawiony na samym początku tworzenia setów

prev wskazuje na poprzedniego reprezentanta (reprezentanta K_{j-1}), next następnego reprezentanta

Gdy robimy union na dwóch setach j i l, gdzie l to zbiór który dalej istnieje po j, to ustawiamy number na maximum number z dwóch reprezentantów, czyli l. Prev ustawiamy na prev z setu j, a next z prev z setu j ustawiamy na nowy number, next na next z setu l i prev z next z setu l na nowy reprezentant

Mamy n razy find-set - O(nlog*n)

Później "let l be smallest value greater than j for which set K_l exists" to wystarczy pójść do next dla j O(1)

Później mamy Union() co najwyżej m razy, czyli łącznie mamy n unionów Mamy więc O(nlog*n) razy.

3 Zadanie 3

Z każdego drzewa robimy union set gdzie set odpowiada drzewu.

każdy node ma v.p - parent v.d - depth w góre (czyli ile jest do korzenia) suma v.d ma być równa głębokości

```
MAKE_TREE(v)
    v = ALLOCATE_NODE()
    v.d = 0
    v.p = v
    return v

FIND_SET(v)
    if v != v.p
        (v.p, d) = FIND_SET(v.p)
        v.d = v.d + d
        return (v.p, v.d)
    return (v, 0)

LINK(r, v)
    (x, d_1) = FIND_SET(r)
    (y, d_2) = FIND_SET(v)
    if x.rank > y.rank
        y.p = x
        x.d = x.d + d_2 + y.d
    else
        x.p = y
        x.d = x.d + d_2
        if x.rank == y.rank
            y.rank = y.rank + 1
```

4 Zadanie 4

No generalnie to jeżeli zwiększymy stałą przy operacji find to zwiększymy złożoność. Tutaj nam się dodaje kolejny log, a złożoność i tak była log* = log(log(log(...))) czyli dodanie jednego loga generalnie nie będzie nas jakoś bardzo boleć, bo i tak mamy log*. Czyli możemy użyc tego samego rozumowania, ponieważ tutaj dochodzi tylko jeden logarytm, stąd trzeba zastosować inną stałą do operacji find w tym zadaniu.

5 Zadanie 5

Dostajemy wierzchołek v i dla każdego wierzchołka chcemy umieć odpowiedzieć ile krawędzi początkowych musimy usunąć aby ten wierzchołek i v nie było połączone żadną krawędzią.

Idziemy od tyłu i dodajemy krawędzie tworząc sobie drzewa w union findzie. Za każdym razem gdy dodamy zrobimy union to chcemy zrobić finda na jednym z dwóch wierzchołków które dodaliśmy (żeby drzewa były postaci ojciec i synowie i nie było głębokości 2). Jeśli nagle trafiamy w union do jakiejś krawędzi w v to robimy tak, że jeśli mamy Union(v, w) to jeśli przed unionem w find(w) = find(v) to nie robimy nic, a w.p.p jeśli w było ojcem swojego uniona to dla każdego syna aktualizujemy v na maxa z obecnych, a jeśli nie było korzeniem swojego uniona to idziemy do ojca i robimy to z ojcem.