Tomasz Waleń
Treść zadania, Opracowanie

Adam Karczmarz
Program

Dostępna pamięć: 64 MB. *OI, Etap II, dzień drugi, 10.02.2011*

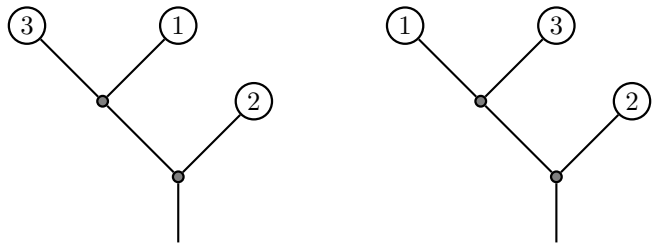
Rotacje na drzewie

Ogrodnik Bajtazar zajął się hodowlą rzadkiego drzewa o nazwie **Rotatus Informatikus**. *Ma ono bardzo ciekawe własności:*

- *Drzewo składa się z prostych gałęzi, rozgałęzień i liści. Wyrastający z ziemi pień drzewa jest również gałęzią.*
- *Każda gałąź jest zakończona u góry rozgałęzieniem lub liściem.*
- *Z rozgałęzienia na końcu gałęzi wyrastają dokładnie dwie dalsze gałęzie — lewa i prawa.*
- *Każdy liść drzewa zawiera jedną liczbę całkowitą z zakresu $1..n$. Liczby w liściach nie powtarzają się.*
- *Za pomocą pewnych zabiegów ogrodniczych, dla dowolnego rozgałęzienia można wykonać tzw.* **rotację**, *czyli zamienić miejscami lewą i prawą gałąź.*

Korona drzewa *to ciąg liczb całkowitych, który otrzymujemy, czytając liczby zawarte w liściach drzewa od lewej do prawej.*

Bajtazar pochodzi ze starego miasta Bajtogrodu i jak wszyscy jego mieszkańcy bardzo lubi porządek. Zastanawia się, jak za pomocą rotacji jak najlepiej uporządkować swoje drzewo. Uporządkowanie drzewa mierzymy liczbą **inwersji** *zawartych w jego koronie, tj. dla korony $a_1, a_2, \ldots, a_n$ wyznaczamy liczbę takich par (i, j), $1 \leqslant i < j \leqslant n$, dla których $a_i > a_j$.*



Rys. 1: Oryginalne drzewo (po lewej) o koronie $3, 1, 2$ zawiera dwie inwersje. Po rotacji otrzymujemy drzewo (po prawej) o koronie $1, 3, 2$, które zawiera tylko jedną inwersję. Każde z tych drzew ma 5 gałęzi.

Napisz program, który wyznaczy minimalną liczbę inwersji zawartych w koronie drzewa, które można otrzymać za pomocą rotacji wyjściowego drzewa Bajtazara.

Wejście

W pierwszym wierszu standardowego wejścia znajduje się jedna liczba całkowita n ($2 \leqslant n \leqslant 200\,000$), oznaczająca liczbę liści drzewa Bajtazara. W kolejnych wierszach znajduje się opis drzewa. Drzewo definiujemy rekurencyjnie:

- *jeśli na końcu pnia (czyli gałęzi, z której wyrasta drzewo) znajduje się liść z liczbą całkowitą p ($1 \leqslant p \leqslant n$), to opis drzewa składa się z jednego wiersza zawierającego jedną liczbę całkowitą p,*
- *jeśli na końcu pnia znajduje się rozgałęzienie, to opis drzewa składa się z trzech części:*
  - *pierwszy wiersz opisu zawiera jedną liczbę 0,*
  - *po tym następuje opis lewego poddrzewa (tak, jakby lewa gałąź wyrastająca z rozgałęzienia była jego pniem),*
  - *a następnie opis prawego poddrzewa (tak, jakby prawa gałąź wyrastająca z rozgałęzienia była jego pniem).*

W testach wartych przynajmniej 30% punktów zachodzi dodatkowy warunek $n \leqslant 5\,000$.

Wyjście

W jedynym wierszu standardowego wyjścia należy wypisać jedną liczbę całkowitą: minimalną liczbę inwersji w koronie drzewa, które można otrzymać za pomocą jakiegoś ciągu rotacji wyjściowego drzewa.

Przykład

Dla danych wejściowych:	*poprawnym wynikiem jest:*
3	1
0	
0	
3	
1	
2	

Wyjaśnienie do przykładu: *Rysunek 1 ilustruje drzewo z przykładu.*

Rozwiązanie

Zliczanie inwersji w permutacjach

Zadanie ma bliski związek z bardzo znanym problemem: zliczaniem inwersji w permutacjach. Jak wskazuje nazwa, polega on na zliczeniu, dla danej permutacji p liczb od 1 do n, wszystkich par (i, j), takich że $1 \leqslant i < j \leqslant n$ oraz $p(i) > p(j)$.

Takie klasyczne zadanie ma wiele ciekawych rozwiązań, w szczególności algorytm korzystający z techniki „dziel i zwyciężaj”. Jak możemy podejść do problemu tym sposobem? Zapiszmy permutację w postaci ciągu i podzielmy go na dwa spójne podciągi możliwie równej długości. Wówczas inwersje będzie można podzielić na trzy grupy: leżące w lewym podciągu, leżące w prawym podciągu oraz te inwersje (i, j), dla których i jest indeksem w lewym, a j w prawym podciągu. Pierwsze dwa przypadki

chcielibyśmy rozważyć przez wywołania rekurencyjne konstruowanej funkcji. Aby było to możliwe, musimy jednak nieco uogólnić nasz problem — będziemy zliczać inwersje w dowolnym różnowartościowym ciągu liczb naturalnych. Szczęśliwie nie wpływa to negatywnie na dotychczas przeprowadzone rozumowanie.

Pozostaje nam zliczenie inwersji trzeciego rodzaju. Przydatne będzie do tego proste spostrzeżenie. Otóż zbiór takich inwersji zależy tylko od tego, jaki jest zbiór wartości w lewym, a jaki w prawym podciągu, nie zależy zaś od uporządkowania tych wartości. Co więcej, jeśli obydwa ciągi byłyby uporządkowane rosnąco, zliczenie tych inwersji byłoby łatwe: można byłoby przystosować do tego zadania algorytm scalania dwóch posortowanych ciągów. Dzięki temu, niejako przy okazji, zapewniamy sobie również, że wywołania rekurencyjne posortują lewy i prawy podciąg. Pożądane przez nas założenie o uporządkowaniu obydwu złączanych ciągów będzie więc spełnione.

Otrzymaliśmy tym samym algorytm, który tak naprawdę jest nieskomplikowaną modyfikacją algorytmu sortowania przez scalanie (*MergeSort*). W szczególności, działa w czasie $O(n \log n)$ i pamięci $O(n)$.

Analiza właściwego problemu

Wróćmy jednak do naszego wyjściowego zadania. W jaki sposób możemy wyliczyć minimalną liczbę inwersji dla drzewa? Na początku bardzo nieznacznie zmodyfikujmy definicję inwersji w tym przypadku. Niech nie będzie to para (i, j) indeksów w koronie, lecz para etykiet (x, y), taka że $x > y$ i liść o etykiecie x jest w koronie wcześniej niż ten o etykiecie y.

Spróbujmy podejść do problemu podobnie jak poprzednio. Tym razem nie będziemy jednak dzielić na pół, lecz z skorzystamy z naturalnego podziału zadanego przez strukturę drzewa — na prawe i lewe poddrzewa. Inwersje dzielą się więc na trzy grupy: te w lewym poddrzewie, te w prawym poddrzewie i te, których pierwszy liść leży w lewym poddrzewie, a drugi w prawym. Spoglądając na taki algorytm zliczania inwersji z globalnej perspektywy, dostrzegamy, że inwersję (x, y) liczymy wtedy, gdy znajdujemy się w najniższym wspólnym przodku liści o tych etykietach.

Wprowadźmy wobec tego użyteczną notację. Dla każdego wierzchołka v wejściowego drzewa przez inv_v oznaczmy liczbę inwersji (x, y) takich, że x jest etykietą liścia w lewym poddrzewie v, a y w prawym poddrzewie v. Oczywiście dla v będących liśćmi drzewa zachodzi $inv_v = 0$, a liczba inwersji w całym drzewie to suma wartości inv_v po wszystkich węzłach v.

W naszym zadaniu nie mamy jednak po prostu policzyć inwersji, lecz zminimalizować ich liczbę, mając do dyspozycji rotacje. Zauważmy jednak, że pojedyncza rotacja w wierzchołku v wpływa jedynie na wartość inv_v, a wszystkie wartości inv_u dla pozostałych wierzchołków ($u \neq v$) pozostają bez zmian. Stąd każdy wierzchołek możemy traktować oddzielnie. Aby w pełni opisać wpływ rotacji na liczbę inwersji, pozostaje stwierdzić, jak zmienia się inv_v przy rotacji w węźle v. Rozważamy oczywiście pary liści, których najniższym wspólnym przodkiem jest v. Nietrudno dostrzec, że wśród nich inwersjami stają się wówczas dokładnie te pary, które nie tworzyły wcześniej inwersji. Wobec tego, jeśli lewe poddrzewo v ma l_v, a prawe r_v liści, to po wykonaniu rotacji liczba inwersji w v jest równa $l_v \cdot r_v - inv_v$.

Te obserwacje pozwalają nam spostrzec, że stosując zachłanną strategię rotacji w każdym z węzłów, otrzymamy optymalne rozwiązanie (dla każdego wierzchołka wykonujemy rotację wtedy, gdy powoduje to zmniejszenie wartości inv_v). Podsumowując, minimalną liczbę inwersji drzewa możemy obliczyć za pomocą wzoru:

$$\sum_{v \in T} \min(inv_v,\, l_v \cdot r_v - inv_v).$$

Niestety nie wszystkie składniki powyższej sumy są łatwe do obliczenia. O ile wartości l_v oraz r_v można wyznaczyć prostym obejściem drzewa, o tyle efektywne wyznaczenie wartości inv_v jest bardziej skomplikowane.

Wyznaczanie liczby inwersji w węzłach

Pozostaje nam teraz stworzyć możliwie efektywny algorytm, który wyznaczy wartości inv_v dla wszystkich wierzchołków drzewa. Zauważmy, że przytoczony na początku opisu algorytm wyznaczający liczbę inwersji w ciągu tak naprawdę wyznacza liczbę inwersji w węzłach pewnego zrównoważonego drzewa binarnego zbudowanego nad tym ciągiem. Stanowi tym samym dobry punkt wyjściowy do naszego algorytmu.

Zdefiniujemy rekurencyjną funkcję *MergeTree*. Funkcja ta dla zadanego wierzchołka v obliczy wartości inv_v w całym poddrzewie, a także zwróci strukturę danych reprezentującą wartości wszystkich liści poddrzewa. Poniżej przedstawiamy szkielet takiego rozwiązania:

```
1:  function MergeTree(v)
2:  begin
3:    niech l oznacza lewego syna v, a r prawego syna v;
4:    Val_l := MergeTree(l);
5:    Val_r := MergeTree(r);
6:    { zgodnie z wcześniejszą definicją inv_v oznacza liczbę par (x, y) }
7:    { takich, że x > y i x ∈ Val_l, y ∈ Val_r }
8:    (inv_v, Val_v) := MergeValues(Val_l, Val_r);
9:    wynik := wynik + min(inv_v, l_v · r_v − inv_v);
10:   return Val_v;
11: end
```

Nadal jednak nie jest to kompletny opis, brakuje realizacji funkcji *MergeValues*.

Proste scalanie

Spróbujmy zastosować podejście, które sprawdziło się przy zliczaniu inwersji w ciągu, czyli do reprezentacji wartości liści użyjmy posortowanych ciągów (przechowywanych w listach lub tablicach dynamicznych). Funkcję *MergeValues* możemy wówczas wykonać w czasie $O(l_v + r_v)$. Niestety prowadzi to do algorytmu, który jest pesymistycznie kwadratowy. Przykładowo, na drzewie o głębokości $\Theta(n)$, w którym każdy lewy syn

jest liściem, algorytm działa w czasie $\Theta(n^2)$. Przedstawione rozwiązanie zostało zaimplementowane w plikach `rots1.cpp` oraz `rots2.pas`. Programy te otrzymywały tylko ok. 25 punktów, ponieważ wykorzystywały listy, które działają dość wolno.

Czas kwadratowy jest jednak o wiele łatwiejszy do osiągnięcia. Zauważmy, że gdyby nasza funkcja *MergeValues* działała w czasie $O(l_v \cdot r_v)$ i siłowo sprawdzała każdą parę liści o najniższym wspólnym przodku w v, wciąż pesymistyczny czas działania pozostałby kwadratowy. Każdą parę liści w całym drzewie sprawdzalibyśmy bowiem dokładnie raz. Takie rozwiązanie można znaleźć w plikach `rots3.cpp` i `rots4.pas`. Ze względu na prostotę sprawowało się nieco lepiej i dostawało ok. 30 punktów.

Efektywne scalanie

Skoncentrujmy się zatem na stworzeniu struktury danych, która pozwoli na efektywniejszą realizację funkcji *MergeValues*. Zacznijmy od opisu cech, które musi posiadać taka struktura danych.

Chcemy przechowywać zbiory kluczy o wartościach ze zbioru $\{1, \ldots, n\}$. Dodatkowo, struktura powinna umożliwiać operację *MergeValues*, która dla danych struktur A i B zwróci nową strukturę zawierającą wszystkie klucze z A i B. Pewnym ułatwieniem może być założenie, że zbiory kluczy z A i B są rozłączne.

Pierwszym rozwiązaniem problemu może być użycie drzew zrównoważonych (np. drzew czerwono-czarnych czy AVL, opisanych np. w książce [22] oraz na stronie `http://wazniak.mimuw.edu.pl`). Drzewa te umożliwiają wykonywanie w czasie $O(\log n)$ m.in. następujących operacji:

- wstawienie nowego elementu do struktury,
- wyznaczenie liczby kluczy w strukturze o wartościach większych niż x.

Możemy też zaimplementować operację scalania drzew A i B, tak by wymagała czasu $O(\min(|A|, |B|) \cdot \log(|A|+|B|))$. Nie jest to trudne — wystarczy, że wstawimy wszystkie klucze z mniejszego drzewa do większego i zwrócimy tak zmodyfikowane drzewo. W takim samym czasie możemy również zliczać liczbę inwersji za pomocą drugiej spośród operacji dostarczanych przez drzewo.

Zastanówmy się chwilę, jak zastosowanie takiej struktury wpłynie na całkowitą złożoność rozwiązania. Operacją dominującą jest wstawienie klucza do struktury. Możemy jednak zauważyć, że w trakcie działania całego algorytmu każdy element może być wstawiany co najwyżej $\log n$ razy. Faktycznie, jeśli wstawiamy jakiś element, to łączymy jego strukturę ze strukturą co najmniej tego samego rozmiaru, czyli po każdym kolejnym wstawieniu tego konkretnego elementu, jego struktura jest przynajmniej dwukrotnie większa. Ponieważ koszt wstawienia elementu szacuje się przez $O(\log n)$, więc całkowity koszt algorytmu wynosi $O(n \log^2 n)$. Złożoność pamięciowa jest jednak bardzo dobra — liniowa.

Niestety, struktura słownikowa dostępna w bibliotece STL języka C++ nie udostępnia operacji zapytania o liczbę kluczy o wartościach mniejszych/większych niż x, co powoduje, że jesteśmy skazani na własną implementację. Jest to niestety skomplikowane i pracochłonne zadanie jak na pięciogodzinną sesję. Jego rozwiązanie pozwalało

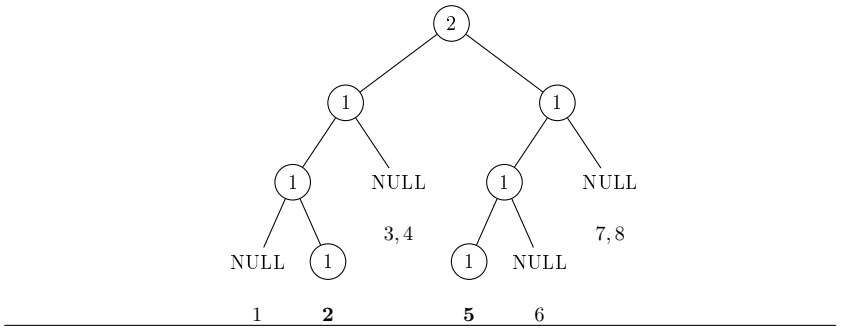
jednak cieszyć się maksymalną punktacją. Przykładowa implementacja znajduje się w plikach `rot2.cpp` oraz `rot3.pas`.

Prostsza struktura danych – drzewo przedziałowe

Spróbujmy wobec tego użyć struktury znacznie łatwiejszej w implementacji, której wykorzystanie często pozwala uchronić się przed implementacją drzew zrównoważonych. Mowa o drzewie przedziałowym[1]. Tym razem nie będzie to zwyczajne statyczne drzewo przedziałowe implementowane standardowo w tablicy, lecz dynamicznie alokowane, reprezentowane jako struktura wskaźnikowa. Jest to w zasadzie zwykłe drzewo przedziałowe, w którym nie przechowujemy poddrzew zawierających 0 kluczy (a dokładniej, całe takie poddrzewa są reprezentowane przez wartość NULL). Każdy węzeł drzewa utrzymuje trzy atrybuty:

- *count* — liczba kluczy w poddrzewie,
- *left* — wskaźnik do lewego syna (lub NULL),
- *right* — wskaźnik do prawego syna (lub NULL).

Przykładowe drzewo tego typu można zobaczyć na rysunku 1. Dzięki takiemu oszczędnemu gospodarowaniu pamięcią reprezentacja k kluczy z przedziału $\{1, \ldots, n\}$ wymaga jedynie pamięci rzędu $O(k \log n)$. W szczególności, struktura zawierająca 1 klucz zajmuje pamięć rzędu $\Theta(\log n)$.



Rys. 1: Przykładowe drzewo przedziałowe dla zakresu $\{1, \ldots, 8\}$ zawierające klucze $\{2, 5\}$. Wewnątrz węzłów podane są wartości atrybutów *count*.

Zauważmy, że skonstruowana przez nas struktura potrafi wykonać dokładnie te operacje, o których pisaliśmy w poprzedniej sekcji, w dokładnie takim samym czasie jak struktury słownikowe. W czym, skoro nie w ograniczeniu funkcjonalności, tkwi wobec tego sekret jej prostoty? Musimy z góry znać zbiór kluczy, jakie pojawiają się w całym algorytmie, i jeśli rozmiar tego zbioru to N, czas operacji na strukturze A jest dla drzew przedziałowych rzędu $O(\log N)$, zaś np. dla drzew AVL rzędu $O(\log |A|)$.

[1] Więcej o drzewach przedziałowych można dowiedzieć się np. w opracowaniu zadania *Tetris 3D* z XIII Olimpiady Informatycznej [13] lub na stronie `http://was.zaa.mimuw.edu.pl/?q=node/8`. Drugie ze źródeł porusza również tematykę dynamicznie alokowanych drzew przedziałowych.

W naszym przypadku nie odgrywa to jednak roli przy oszacowaniu złożoności czasowej — pozostaje równa $O(n \log^2 n)$. Poświęćmy jeszcze chwilę złożoności pamięciowej: jeśli przy łączeniu drzew od razu zwolnimy pamięć po niepotrzebnych węzłach, żaden klucz nie będzie nigdy w dwóch różnych drzewach. Stąd łącznie będziemy potrzebować $O(n \log n)$ węzłów. Pozwala to, co prawda z niewielkim zapasem, zmieścić się w limicie pamięci. Większa złożoność pamięciowa to jednak wyraźna wada w porównaniu ze strukturami słownikowymi.

Efektywniejsze scalanie drzew przedziałowych

Poprzednio scalaliśmy dwa drzewa przedziałowe przez wstawianie elementów mniejszego do większego. Spróbujmy zastosować nieco inne podejście, którego idea jest bardzo naturalna dla scalania statycznych drzew przedziałowych (cały czas pracujemy nad ustalonym uniwersum kluczy). W takich drzewach każdy element tablicy reprezentującej drzewo odpowiada zawsze za określony przedział wartości uniwersum. Zatem wartość *count* w węźle drzewa po scaleniu to po prostu suma wartości *count* w odpowiednich węzłach scalanych drzew.

W drzewach dynamicznie alokowanych zachodzi ta sama zależność — trzeba tylko wziąć pod uwagę, że część poddrzew jest zastąpionych przez NULL. Innymi słowy, jeśli kształt ścieżki z korzenia do węzła dla pewnych dwóch węzłów różnych drzew jest taki sam, to węzły te odpowiadają za ten sam przedział. Na przykład, lewy syn prawego syna korzenia odpowiada za trzecią ćwiartkę uniwersum.

Oparty na tych spostrzeżeniach algorytm scalania wygodnie zapisać rekurencyjnie. Dla danych dwóch drzew przedziałowych A i B tworzymy nowy węzeł, którego lewe poddrzewo jest efektem scalenia lewych poddrzew A, B, zaś prawe — prawych. Musimy jeszcze ustawić atrybut *count* w nowym węźle, który powinien przyjąć wartość $count(A) + count(B)$. Po wykonaniu scalenia węzły reprezentujące korzenie A i B nie są już dalej potrzebne, więc możemy zwolnić zajmowaną przez nie pamięć[2].

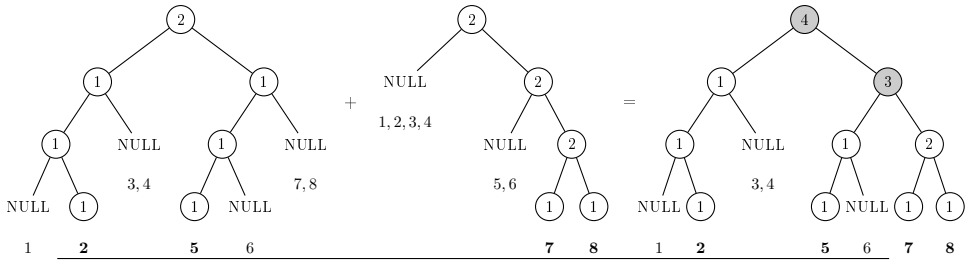
Na razie nie widać, czemu nasze rozwiązanie miałoby być choćby równie efektywne jak poprzednie. Jednak nie opisaliśmy jeszcze scalania poddrzew, z których przynajmniej jedno jest puste (reprezentowane przez NULL). Jest to oczywiście bardzo proste — wynikiem jest drugie ze scalanych poddrzew. Aby podać ten wynik, nie potrzeba schodzić w głąb tego poddrzewa, wystarczy więc do tego czas stały.

Funkcja *MergeValues* powinna również zwracać liczbę inwersji pomiędzy strukturami. Możemy to osiągnąć, sumując (dla wszystkich wywołań rekurencyjnych) iloczyny liczby kluczy w prawym poddrzewie drzewa A i lewym poddrzewie drzewa B. Jeśli jedno z poddrzew jest puste, cały iloczyn to oczywiście zero. Również aby podać tę wartość, nie musimy zagłębiać się w drugie drzewo.

Poniżej przedstawiamy pseudokod opisanego rozwiązania, natomiast na rysunku 2 zobrazowany jest efekt działania funkcji.

[2]W praktyce warto nie tworzyć nowego węzła, lecz wykorzystać ponownie jeden z usuwanych. Dzięki temu w całym algorytmie nowe węzły będziemy tworzyli tylko podczas budowania drzew reprezentujących jednoelementowy zbiór kluczy. To zaś pozwoli zaalokować na początku działania odpowiednią ich liczbę. Takie podejście znacząco przyspiesza program.

```
1:  function MergeValues(A,B)
2:  begin
3:    if A = NULL then return (0, B);
4:    if B = NULL then return (0, A);
5:    inv := count(A.right) · count(B.left);
6:    (inv_l, l) := MergeValues(A.left, B.left);
7:    (inv_r, r) := MergeValues(A.right, B.right);
8:    t := nowy węzeł drzewa przedziałowego;
9:    t.count := count(A) + count(B);
10:   t.left := l;
11:   t.right := r;
12:   zwolnij pamięć odpowiadającą węzłom A i B;
13:   return (inv_l + inv_r + inv, t);
14: end
```



Rys. 2: Przykład działania funkcji *MergeValues* dla drzew zawierających klucze $A = \{2, 5\}$ oraz $B = \{7, 8\}$. Wynikiem jest drzewo zawierające klucze $C = \{2, 5, 7, 8\}$. Kolorem szarym zostały oznaczone nowe węzły utworzone w trakcie scalania.

Zastanówmy się teraz, jaka jest złożoność powyższego rozwiązania. Nietrudno dostrzec, że złożoność pojedynczego wykonania funkcji *MergeValues* szacuje się przez liczbę węzłów mniejszego z drzew, tzn. $O(\min(|A|, |B|) \cdot \log n)$, więc złożoność całości nie jest gorsza niż poprzednio. Można jednak pokazać, że jest istotnie lepsza. Tutaj przychodzi z pomocą analiza kosztu zamortyzowanego z użyciem funkcji potencjału (więcej informacji na temat tej techniki dowodzenia można odnaleźć w książce [22]). W naszym przypadku jako funkcję potencjału możemy wybrać liczbę węzłów drzew przedziałowych. Początkowo mamy n drzew przedziałowych, z których każde zawiera dokładnie jeden klucz (każde z takich drzew odpowiada liściowi z drzewa danego w treści zadania), stąd początkowy potencjał wynosi $\Theta(n \log n)$.

Musimy teraz wykazać, że żadne wywołanie funkcji *MergeValues* nigdy nie zwiększy potencjału, a jeśli po jej wykonaniu potencjał zmniejszy się o wartość Δ, to wykonana praca nie będzie większa niż $c \cdot (\Delta + 1)$ dla pewnej stałej $c > 0$.

Rozważmy dwa możliwe przypadki:

- jeden z argumentów A lub B przyjmuje wartość NULL, w tym przypadku zmiana potencjału Δ wynosi 0, jednak tutaj wykonujemy stałą pracę; wystarczy więc dobrać odpowiednio dużą stałą c, tak aby zachodziło żądane ograniczenie na wykonaną pracę;

- w przypadku, gdy oba drzewa są niepuste, wykonujemy rekurencyjnie pracę co najwyżej $c \cdot (\Delta_l + 1)$ (dla lewych poddrzew) oraz $c \cdot (\Delta_r + 1)$ (dla prawych poddrzew), przy czym $\Delta_l + \Delta_r = \Delta - 1$ (usuwamy dwa węzły A i B, a tworzymy nowy węzeł t). Stąd także w tym przypadku wykonana praca nie przekracza $c \cdot (\Delta + 1)$.

Podsumowując, całkowity czas poświęcony wywołaniom *MergeValues* nie przekracza początkowej wartości potencjału, czyli $O(n \log n)$. Taka jest również i złożoność pamięciowa tego rozwiązania. Opisany algorytm zaimplementowano w plikach `rot5.cpp` oraz `rot7.cpp`. Oba programy dostają 100 punktów. Drugi jest jednak blisko trzykrotnie szybszy dzięki lepszemu zarządzaniu alokacją i zwalnianiem pamięci.

A może jeszcze coś da się poprawić?

Co ciekawe, to nie jest jeszcze koniec optymalizacji, które możemy uzyskać w tym zadaniu. Można bowiem zredukować złożoność pamięciową do $O(n)$, przy zachowaniu złożoności czasowej $O(n \log n)$. Takie rozwiązanie nie było oczywiście wymagane, ale stanowi wartą wzmianki ciekawostkę.

Jeden z przykładów takiego podejścia opiera się na nieznacznie bardziej skomplikowanej implementacji drzew przedziałowych. Wystarczy bowiem zredukować pamięć potrzebną do przechowywania k kluczy z $O(k \log n)$ do $O(k)$. Ten cel możemy zrealizować przez kompresję do jednej krawędzi ścieżek utworzonych z węzłów o dokładnie jednym synu. Ta sama technika jest wykorzystywana np. w drzewach sufiksowych.

Oczywiście, trudności implementacyjne przy scalaniu struktur są o wiele większe, program nie jest jednak bardzo długi. Aby dało się odtworzyć skompresowaną ścieżkę, w węźle trzeba dodatkowo pamiętać np. indeks, jaki otrzymałby w statycznej implementacji drzewa. To podejście, dzięki zastosowaniu niestandardowych operacji na bitach, pozwala nawet na łączenie drzew bez rozwijania ścieżek, co nie jest wymagane do osiągnięcia pożądanej złożoności czasowej i pamięciowej, ale w praktyce wyraźnie przyspiesza program. Opisane rozwiązanie zostało zaimplementowane w pliku `rot6.cpp`.

Testy

Zadanie było oceniane przy użyciu 11 zestawów danych testowych. Specyfika zadania wymagała zastosowania dosyć agresywnego grupowania testów.

Nazwa	n	Opis
rot1a.in	20	mały test poprawnościowy
rot1b.in	19	małe drzewo słabo zrównoważone
rot1c.in	20	drzewa o zbliżonych bądź skrajnie różnych wielkościach poddrzew (w każdym miejscu)
rot1d.in	20	zrównoważone drzewo o względnie dużej liczbie koniecznych inwersji

Nazwa	n	Opis
rot1e.in	2	minimalny test, odpowiedź 0
rot2a.in	101	drzewo zrównoważone u góry, u dołu zakończone krótkimi „ścieżkami”
rot2b.in	110	ścieżka z przyłączonymi ścieżkami
rot2c.in	126	losowe drzewo zrównoważone
rot2d.in	115	ścieżka z doczepionymi parami liści
rot3a.in	756	drzewo losowe, słabo zrównoważone
rot3b.in	879	drzewo jeszcze słabiej zrównoważone, z doczepionymi krótkimi ścieżkami
rot3c.in	856	drzewo dobrze zrównoważone
rot3d.in	832	rozgałęziona ścieżka, w korzeniu połączona z losowym drzewem
rot4a.in	3 885	pojedyncza ścieżka, która rozgałęzia się na mniejsze losowe drzewa
rot4b.in	4 159	drzewo słabo zrównoważone, ze sporą liczbą krótkich ścieżek
rot4c.in	4 387	dwie różnie skomplikowane ścieżki, połączone w korzeniu
rot4d.in	4 291	długa ścieżka z kilkoma małymi drzewkami
rot5a.in	11 009	losowa ścieżka w dół, test na poprawność
rot5b.in	12 474	losowe drzewo słabo zrównoważone z odrostami
rot5c.in	30 129	drzewo bardzo dobrze zrównoważone
rot5d.in	13 045	skomplikowane drzewo, zawierające ścieżki zarówno losowe, jak i posortowane
rot6a.in	50 768	drzewo z dużą liczbą odstających ścieżek
rot6b.in	47 809	drzewo rozrasta się ostatecznie w 4 długie ścieżki
rot6c.in	53 002	duży test poprawnościowy, drzewo losowe z długą ścieżką na końcu
rot6d.in	57 009	duży test przeciwko drzewom BST bez równoważenia
rot7a.in	80 987	ścieżka rozgałęzia się na kilkadziesiąt innych ścieżek
rot7b.in	83 998	korzeń rozdziela się na posortowaną ścieżkę z nieposortowanymi odrostami i na pomieszaną ścieżkę z posortowanymi odrostami
rot7c.in	82 345	ścieżka z odrastającymi sporymi losowymi drzewami
rot7d.in	87 509	test przeciwko słabym BST, kilkanaście odrastających posortowanych ścieżek
rot8a.in	120 859	duże drzewo ze sporą liczbą dosyć krótkich odrastających ścieżek

Nazwa	n	Opis
rot8b.in	117 098	także duże drzewo, lecz teraz odrasta od niego więcej mniejszych ścieżek
rot8c.in	119 877	dwie ścieżki z odrastającymi małymi drzewkami, połączone w korzeniu
rot8d.in	118 567	test przeciwko słabym BST, posortowane odrosty
rot9a.in	172 098	skomplikowane drzewo o dużej minimalnej liczbie inwersji
rot9b.in	173 790	proste drzewo losowe o małej minimalnej liczbie inwersji
rot9c.in	169 123	skomplikowany test poprawnościowy
rot9d.in	174 072	długie posortowane odrosty, przeciwko słabym BST
rot9e.in	170 000	drzewo zrównoważone
rot10a.in	189 680	10 długich ścieżek, niektóre posortowane, inne losowe
rot10b.in	191 998	większy wariant testu *7b*
rot10c.in	190 067	większy wariant testu *7c*
rot10d.in	190 800	test przeciwko słabym BST, podobny do *6d*, lecz dużo większy
rot10e.in	193 983	drzewo zrównoważone
rot11a.in	199 998	na początku drzewo „dzieli się” po równo, ostatecznie jednak rozdziela się na 16 długich ścieżek
rot11b.in	200 000	test poprawnościowy, większy wariant testu *8b*
rot11c.in	200 000	duży test na drzewo zrównoważone
rot11d.in	200 000	skomplikowany test podobny do *11a*, lecz na dole więcej możliwych różnych scenariuszy
rot11e.in	200 000	prosta, losowa ścieżka maksymalnej długości
rot11f.in	200 000	drzewo doskonale zrównoważone