Zadanie 1

Zaprojektuj strukturę danych, która umożliwia wydajne wykonywanie następujących operacji na dynamicznym ciągu liczbowym *C*:

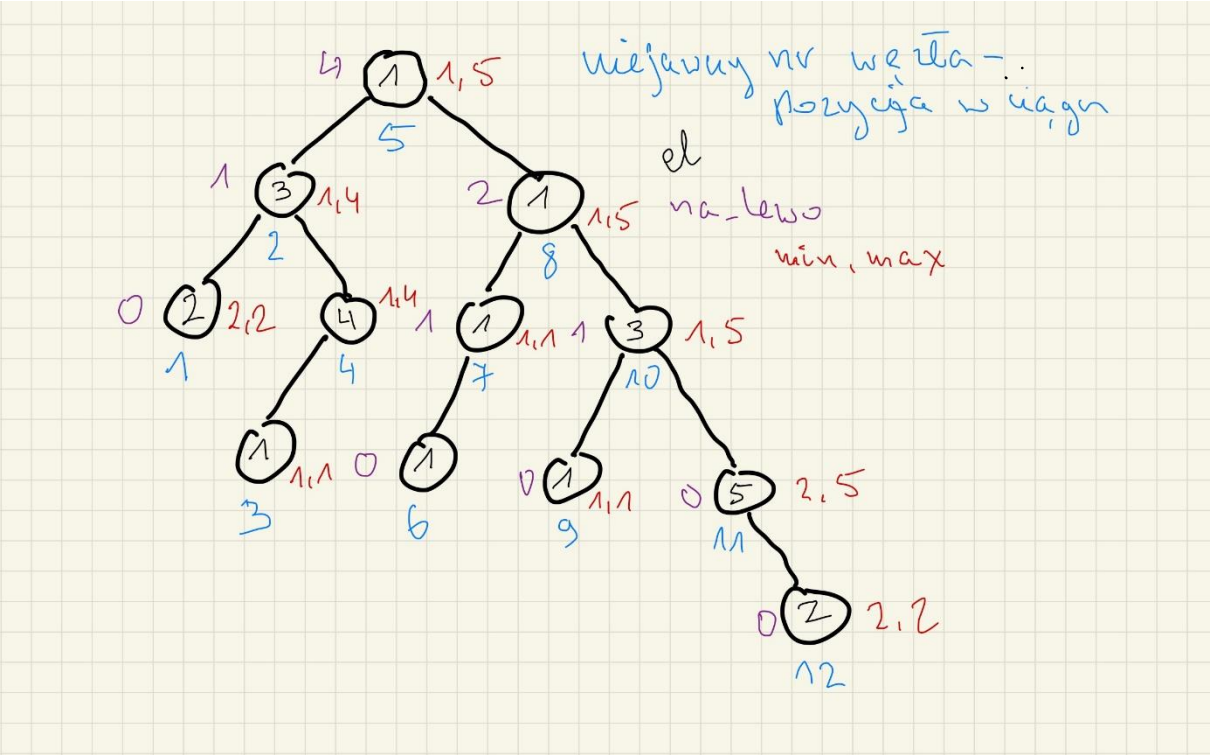
a) init:: utwórz pusty ciąg *C* (operacja wykonywana tylko raz na samym początku);
b) Element(*i*):: podaj wartość elementu z pozycji *i* w ciągu *C*, $1 \le i \le |C|$;
c) insert(*i*, *x*):: wstaw element *x* jako *i*-ty element ciągu *C* (za elementem z dotychczasowej pozycji *i*-1 a przed dotychczasowym elementem z pozycji *i*), $1 \le i \le |C|+1$;
d) delete(*i*):: usuń element z pozycji *i* z ciągu *C*, $1 \le i \le |C|$;
e) isConstant(*i*, *j*):: sprawdź, czy w podciągu *C*[*i*..*j*] wszystkie elementy są takie same, $1 \le i \le j \le |C|$;
f) mostCommon:: podaj wartość najczęściej pojawiającego się elementu w ciągu *C* (w przypadku kilku takich elementów wystarczy podać wartość tylko jednego z nich).

Rozwiązanie

Skupmy się najpierw na punktach a) – e}. Ciąg implementujemy standardowo z wykorzystaniem zrównoważonego drzewa wyszukiwań binarnych (AVL, czerwono-czarne, splay), w którym zrównoważenie przywraca się z użyciem rotacji. W takiej implementacji i-ty element ciągu znajduje się w i-tym węźle w porządku in-order. Ważne, że pojedyncza rotacja (podwójna składa się z dwóch pojedynczych) zachowuje porządek in-order. Żeby umieć nawigować po drzewie w poszukiwaniu i-tego elementu potrzebujemy w każdym węźle dodatkowy atrybut *na_lewo* mówiący, ile jest węzłów w lewym poddrzewie takiego węzła. Dodatkowo będziemy chcieli wiedzieć, czy wszystkie elementy ciągu zapisane w poddrzewie są takie same. Zauważmy, że tworzą one spójny podciąg zapisanego w drzewie ciągu. W tym celu każdy węzeł wyposażymy w dwa dodatkowe atrybuty *min* i *max*, których wartościami są odpowiednio wartość najmniejszego i wartość największego elementu ciągu zapisanego w poddrzewie o korzeniu w tym węźle. Jeśli *min* = *max*, to wszystkie elementy w poddrzewie są takie same. Jeśli do reprezentacji ciągu użyjemy drzewa AVL, to każdy węzeł będzie miał atrybuty:
el – element zapisany w węźle
lewy, prawy, rodzic – wskaźniki odpowiednio do lewego i prawego dziecka oraz rodzica
wr – wskaźniki zrównoważenia
na_lewo – liczba węzłów w lewym poddrzewie
min, max – wartości odpowiednio najmniejszego i największego elementu zapisanego w poddrzewie

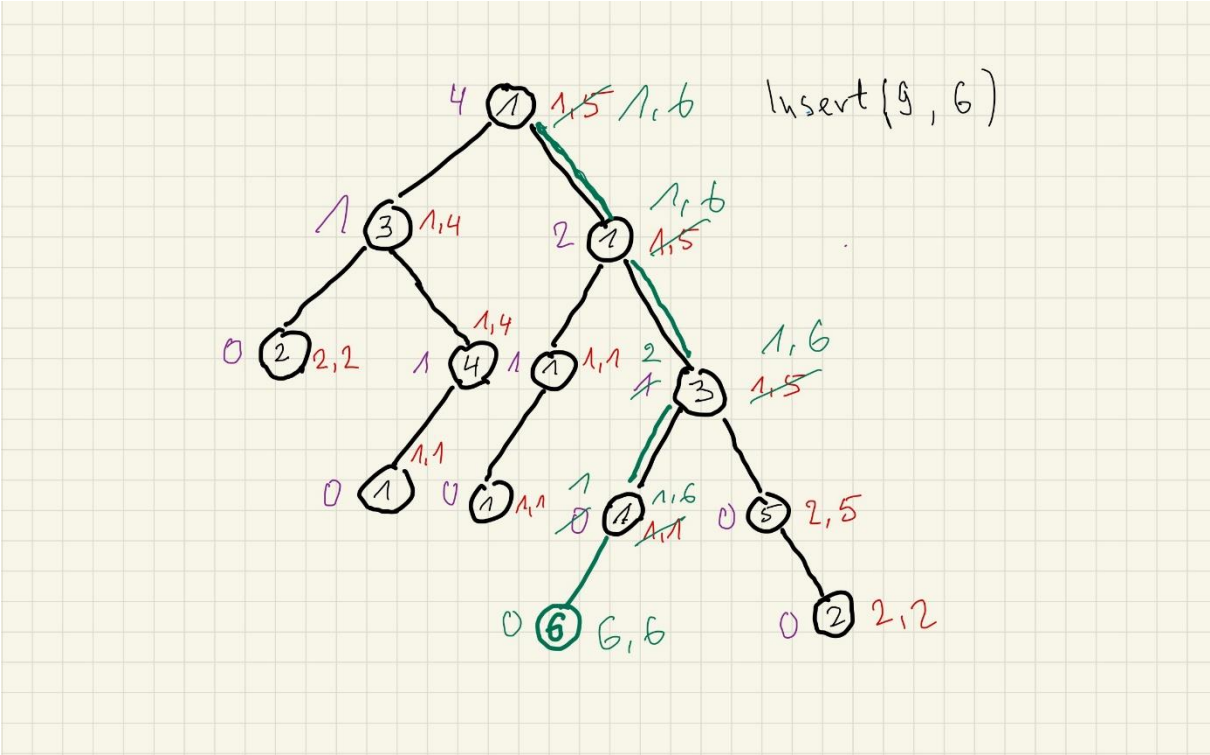
Przykład

Ciąg 2, 3, 1, 4, 1, 1, 1, 1, 1, 3, 5, 2 i jego reprezentacja w AVL-drzewie.



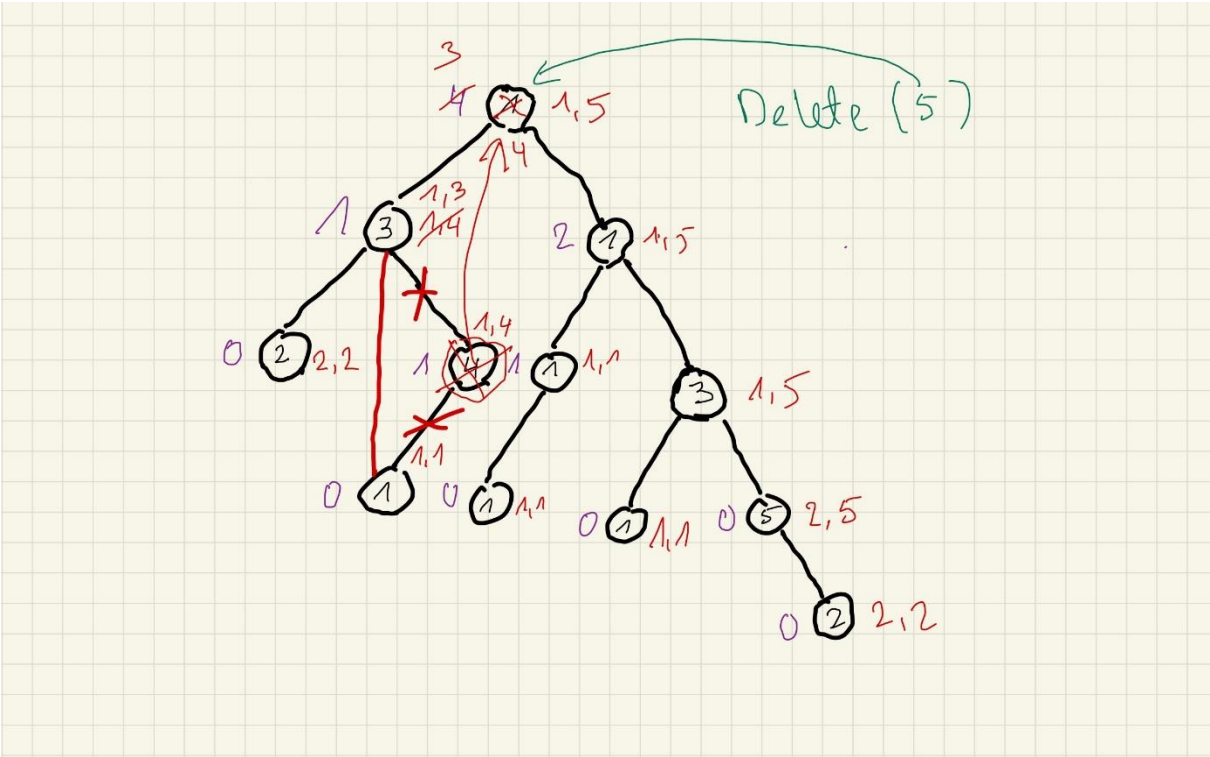
a) drzewo inicjujemy jako puste
b) Załóżmy, że jesteśmy w węźle v i szukamy i-tego węzła w porządku in-order w poddrzewie (i-tego elementu w podciągu zapisanego w tym poddrzewie).
Jeśli v.na_lewo + 1 = i, to szukany element ciągu ma wartość v.el.
Jeśli v.na_lewo >= i, to szukamy i-tego węzła w lewym poddrzewie v, którego korzeniem jest v.lewy.
Jeśli v.na_lewo + 1 < i, to szukamy węzła o numerze i - v.na_lewo – 1 w prawym poddrzewie v o korzeniu v.prawy.
Operację Element(i) zaczynamy od korzenia całego drzewa.
Koszt – O(wysokość drzewa) = O(log |C|) .
c) Żeby wstawić nowy element x na pozycji i-tej szukamy w drzewie v węzła o numerze i-1 (gdy i = 1, szukamy węzła o numerze 1). Dodajemy nowy węzeł u jako ostatni na skrajnie lewej ścieżce w prawym poddrzewie v (gdy i = 1, dodajemy nowy węzeł na lewo od węzła o numerze 1). Zapisujemy u.el = x, u.na_lewo = 0, u.min = x, u.max = x. Wracamy w górę drzewa i aktualizujemy wartości atrybutów w węzłach na ścieżce do korzenia. Zauważ, że wartość atrybutu w węźle może być zaktualizowana na podstawie informacji zapisanych w tym węźle i jego dzieciach. Pozostaje jeszcze przywrócić zrównoważenie, o czym za chwilę.
Koszt – O(wysokość drzewa) = O(log |C|).

Przykład

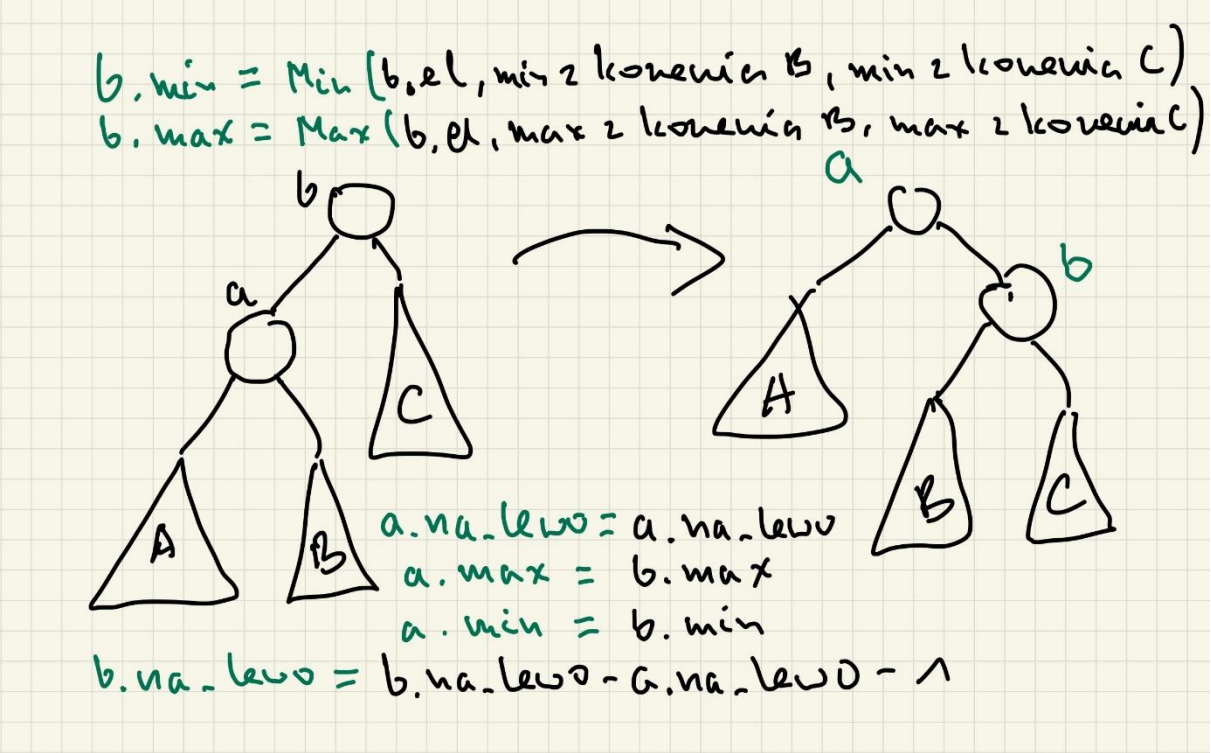


d) Najpierw postępujemy tak, jak byśmy usuwali węzeł o numerze i w drzewa BST, pamiętając o aktualizacji atrybutów w węzłach na ścieżce od miejsca faktycznie usuniętego węzła do korzenia. Potem przywracamy zrównoważenie.
Koszt – O(wysokość drzewa) = O(log |C|).

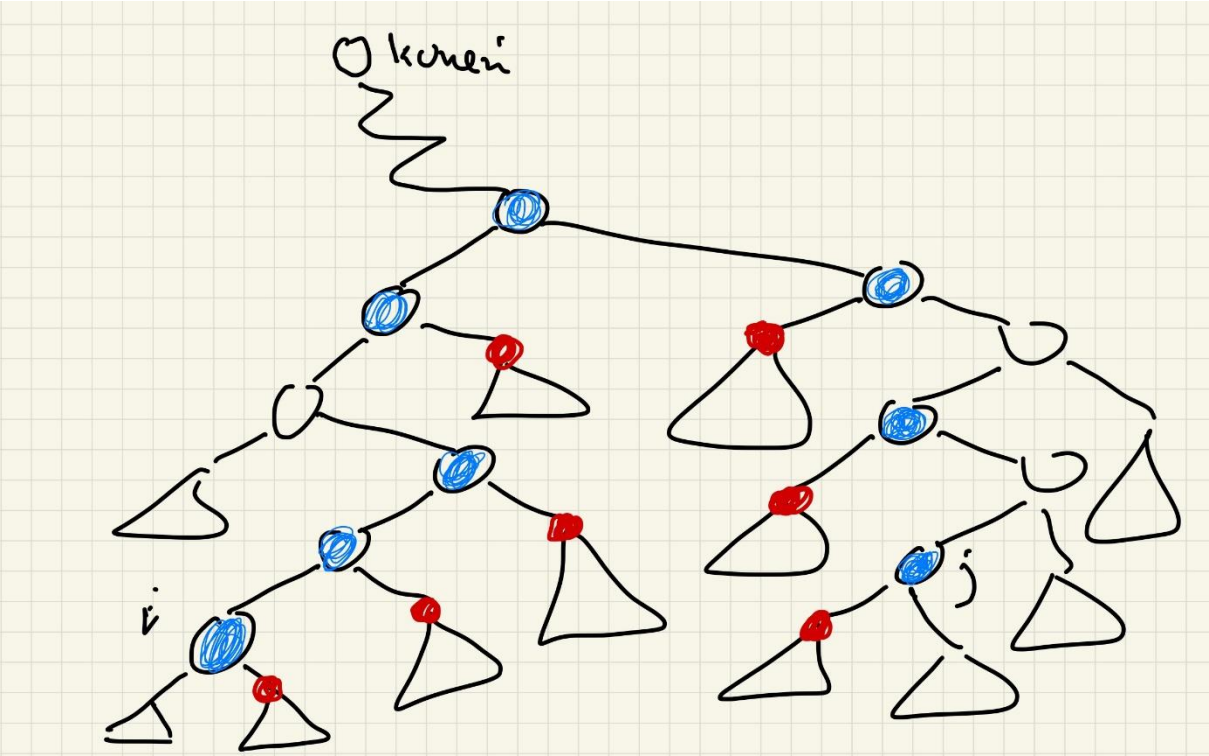
Przykład



Pokażemy teraz, że pojedynczą rotację można wykonać tak, żeby zapewnić właściwe własności atrybutów.



e) Należy znaleźć węzeł o numerze i oraz węzeł o numerze j, a potem ich najgłębszego wspólnego przodka. Na ścieżkach od i oraz j do najgłębszego wspólnego przodka wyznaczamy węzły z przedziału [i,j] odpowiednio te dla których i jest w ich lewym oraz te dla których j jest w prawym poddrzewie. Następnie sprawdzamy, czy wszystkie elementy w tych węzłach mają taką sama wartość x oraz czy w korzeniach prawych poddrzew tych węzłów dla i oraz w korzeniach lewych poddrzew tych węzłów dla j mamy x = min = max.
Koszt jak poprzednio – O(wysokość drzewa) = O(log |C|).



f) Ostatni punkt jest najprostszy. Bierzemy dodatkowe zrównoważone drzewo BST, w którym przechowujemy różne wartości elementów wraz z ich krotnościami. Ponadto te wartości przechowujemy w kolejce priorytetowej typu max, gdzie priorytety to krotności wartości elementów w ciągu. Dla każdej wartości mamy jej dowiązanie do wystąpienia w kolejce priorytetowej. Operacje na kolejce priorytetowej to Max, Insert, DecreaseKey, IncreaseKey oraz Delete (wskazanego elementu), przy czym usuwamy każdą wartość, której krotność spada do zera. Wszystkie te operacje można wykonać w pesymistycznym czasie O(log |C|) na drzewach lewicowych, kolejkach dwumianowych, czy kopcach zupełnych implementowanych z użyciem wskaźników (nie znamy ograniczenia na długość ciągu) zamiast w tablicy.

Zadanie 2

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.

a) Zaproponuj algorytm, który w czasie liniowym obliczy długość najdłuższego super-dobrego podciągu danego ciągu A.
b) 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, $1 \le i \le |A|+1$
Delete(A,i):: usuń i-ty element z A, $1 \le i \le |A|$
SuperGood(A,i,j):: sprawdź, czy podciąg A[i..j] jest super-dobry, $1 \le i \le j \le |A|$

Zadanie posiada pewnie wiele rozwiązań. Zaprezentujemy takie, w którym ciągi nad zbiorem {-1, 0, 1} interpretujemy jako wyrażenia nawiasowe, gdy wartości 1 odpowiada nawias otwierający (, wartości -1 – nawias zamykający), 0 to symbol „obojętny" #. Ciąg z przykładu ma zatem postać

A =[1 1 0 -1 0 0 1 -1 1 1 -1]
((#) # # ( ) (( )

Podciąg super-dobry, to poprawnie zbudowane wyrażenie nawiasowe (gdy pominiemy #). Najdłuższym super dobrym podciągiem A jest podciąg zaczynający się na pozycji 2-giej i kończący na pozycji 8-ej –

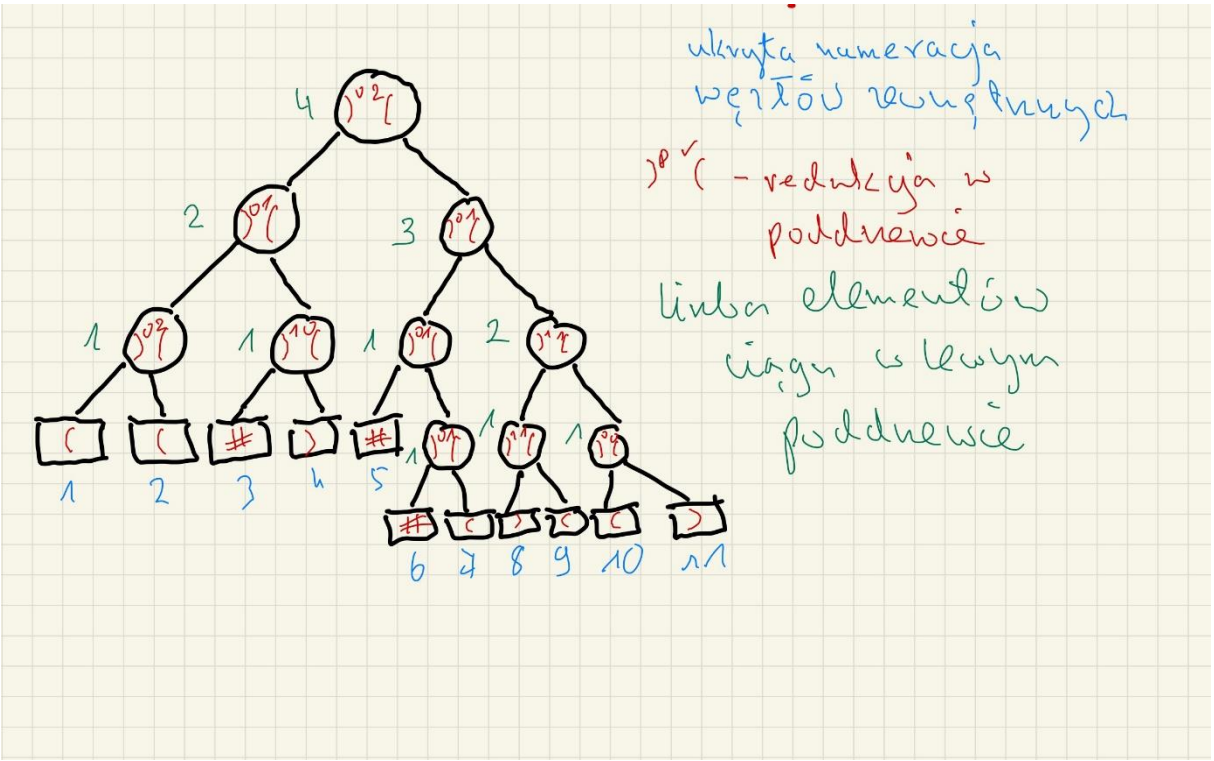
(#) # # ( ).

a) Przeglądamy ciąg od strony lewej do prawej trzymając na stosie nawiasy otwierające, które nie znalazły jeszcze odpowiadających im nawiasów zamykających. Zakładamy, że przed pierwszym nawiasem, pomiędzy nawiasami i za ostatnim nawiasem na stosie znajduje się liczba o wartości równej długości poprawnego wyrażenia nawiasowego pomiędzy tymi nawiasami. Dodatkowego na zmiennej max pamiętamy długość najdłuższego dotychczas wykrytego poprawnego wyrażenia nawiasowego. Początkowo max = 0. Oto zawartość stosu po przetworzeniu pierwszych 7 elementów ciągu A (stos rośnie z lewa na prawo): 0 (5 (0.

   Niech S będzie stosem i niech k będzie aktualną liczba elementów na stosie i niech E będzie kolejnym rozważanym z ciągu A. W zależności od tego czym jest E wykonujemy jedną z akcji:
   - E = (=> na stos wstawiamy kolejno (i 0 a następnie przechodzimy do kolejnego elementu w ciągu A
   - E = # => S[k] := S[k]+1, ponieważ długość poprawnie zbudowanego wyrażenia z prawej strony wzrosła o 1 (obojętny symbol #), aktualizujemy max - max := Max(max, S[k]) i przechodzimy do następnego elementu A
   - E =) =>
   jeśli k = 1 to S[k] := 0 – każde następne poprawnie zbudowane wyrażenie zacznie się na prawo od rozważanej pozycji w ciągu A;
   jeśli k > 1 to S[k-2] := S[k-2]+2+S[k], k := k – 2, max := Max(max, S[k])
   … $\#^{S[k=2]}$ ($\#^{S[k]}$)
   i przechodzimy do następnego elementu ciągu A.

Złożoność w oczywisty sposób liniowa – dla każdego elementu ciągu A wykonujemy stałą liczbę operacji.

b) Ciąg A, jako wyrażenie nawiasowe będziemy reprezentowali z pomocą zrównoważonego drzewa wyszukiwani binarnych, np. AVL. Jednak w przeciwieństwie do poprzedniego zadania założymy, że elementy ciągu A znajdują się w |A| węzłach zewnętrznych AVL drzewa, kolejno od strony lewej do prawej. W każdym węźle wewnętrznym zapamiętamy „redukcję” wyrażenia nawiasowego umieszczonego w węzłach zewnętrznych poddrzewa o korzeniu w tym węźle, powstałego po usunięciu wszystkich par pasujących do siebie nawiasów i symboli obojętnych. Każda taka redukcja ma postać $)^p(^r$, co oznaczę, że p nawiasów zamykających i r nawiasów otwierających nie znalazło w tym poddrzewie swoich odpowiedników. Dodatkowo drzewo wzbogacamy dodając do każdego węzła wewnętrznego informację ile jest węzłów zewnętrznych w jego lewym poddrzewie. Taka informacja pozwoli nam łatwo docierać do wskazanej pozycji w ciągu (patrz zadanie poprzednie). Poniżej wzbogacone przykładowe AVL-drzewo dla ciągu A.



Nietrudno zauważyć, że pojedyncza rotacja pozwala poprawnie, lokalnie zaktualizować wartości atrybutów. Stąd wstawianie, usuwanie, poszukiwanie i-tego elementu można wykonać w czasie O(log |A|).

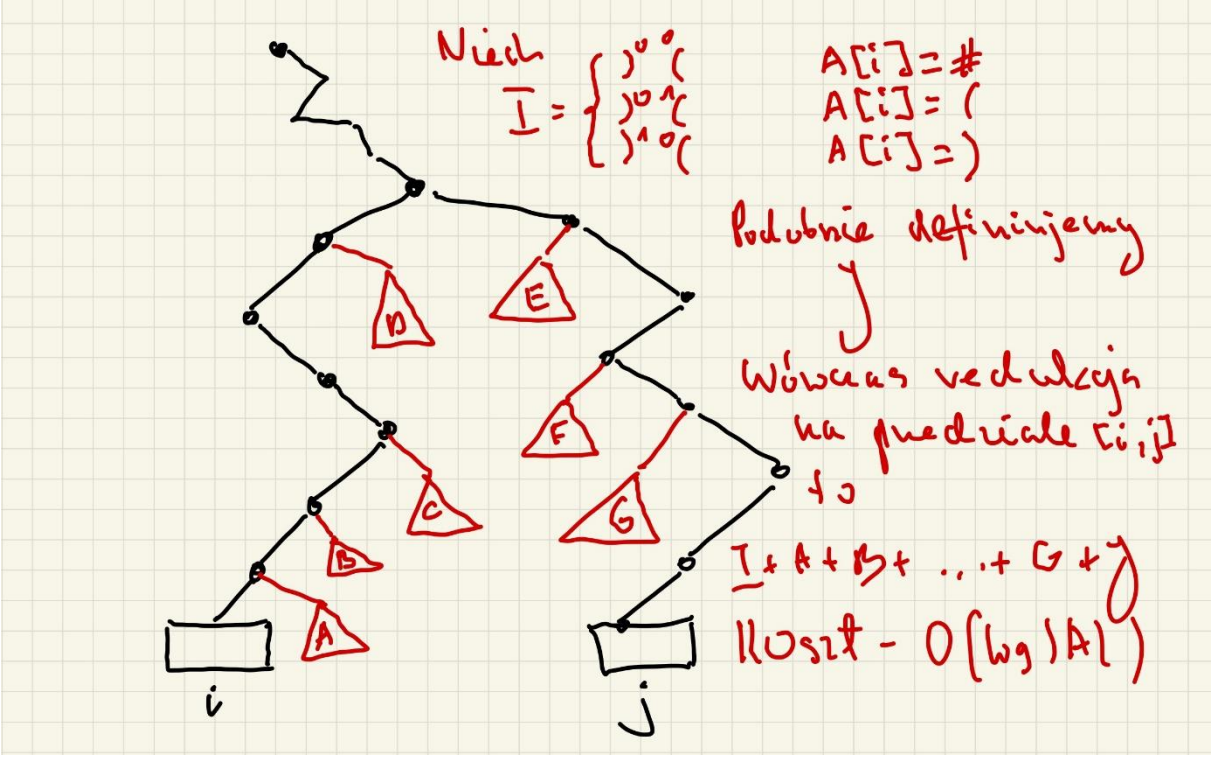
Operację SuperGood(i,j) wykonujemy podobnie do operacji isCostant, obliczając redukcję na przedziale [i,j]. Załóżmy, że mamy redukcję na przedziale [a,b] zapisana w korzeniu drzewa A oraz redukcję dla przedziału [b+1,c] zapisaną w korzeniu drzewa B. Wówczas redukcja na przedziale [a,c] zapisujemy jako A+B i obliczamy jak następuje:

redukcja A - $)^i\ {}^j($

redukcja B - $)^{s}\ {}^{t}($

A+B = $)^{i + \max(0,s-j)}\ {}^{t + \max(0,j-s)}($

Poniżej ilustracja sposobu obliczenia redukcji na przedziale [i,j].



Zadanie 3

Na początkowo pustej szachownicy o wymiarach *N*x*N* dwaj gracze na przemian stawiają białe i czarne pionki. Gdy dwa pionki o tym samym kolorze znajdą się obok siebie (w przypadku powstania kilku takich par pionków wybieramy TYLKO JEDNĄ, dowolną z nich), zdejmujemy postawiony dopiero co pionek i zmieniamy kolor drugiego pionka w parze na przeciwny. Jeśli powstała w ten sposób co najmniej jedna nowa para sąsiednich pionków o tym samym kolorze, to postępujemy podobnie jak poprzednio – zdejmujemy dopiero co przekolorowany pionek, a jego sąsiada w parze kolorujemy na kolor przeciwny. Proces ten kontynuujemy, aż nie będzie sąsiednich pionków tego samego koloru. Podaj zamortyzowany koszt jednego posunięcia w grze, gdy postawienie pionka, zdjęcie pionka z planszy, zmiana koloru pionka są operacjami jednostkowymi.

Dokonaj analizy kosztu zamortyzowanego

a) metodą księgowania
   Niezmiennik: każdy pionek na szchownicy ma 2 złote:
   - 1 zł na zdjęcie go z planszy
   - 1 zł na przekolorowanie sąsiada
   Koszt zamortyzowany wynosi 3 zł: 1 zł na postawienie pionka na szachownicy + 2 zł na zachowanie niezminnika. Za usuwanie pionków z szachownicy i przekolorowanie sąsiadów płacimy złotówkami przypisanymi do usuwanych pionków.
b) metodą funkcji potencjału
   Definiujemy potencjał struktury równy 2 x liczba pionków na szachownicy. Potencjał jest nieujemny i równy 0 dla pustej szachownicy. Załóżmy teraz, że po postawieniu

pionka na planszy wykonujemy k usunięć/przekolorowań pionków. Wówczas koszt zamortyzowany tej operacji wynosi:
1 (postawienie pionka) + 2k (usuwanie/przekolorowane) + 2 – 2k (różnica potencjału) = 3

Zadanie 4

Mamy trzy, początkowo puste stosy S_1, S_2, S_3. Wykonujemy ciąg ruchów, z których każdy polega na dodaniu do wybranego dowolnie stosu jednego żetonu (operacja Push). Jeśli po dodaniu nowego żetonu, wysokości dwóch różnych stosów i oraz j są takie same oraz większe od 0, rozpoczynamy proces czyszczenia stosów realizowany za pomocą procedury Oczyść(i,j).

```
Oczyść(i,j)::
begin
  k := indeks stosu różny od i oraz j;
  repeat
    Push(Sj,Pop(Si));
    if (h(Si) > 0) AND (h(Si) = h(Sk)) then begin a := j; j := k; k := a end
    else
      if (h(Sj) = h(Sk)) then begin a := i; i := k; k := a end
  until h(Si) = 0
end;
```

Uwaga: Pop(S) jest funkcją, której wynikiem jest element usuwany ze stosu S; h(S) oznacza wysokość stosu S. W tym zadaniu operacjami elementarnymi są operacje stosowe Push i Pop.

a) Udowodnij, że operacja czyszczenia zawsze się kończy.

Wystarczy zauważyć, że w każdym obrocie pętli różnica wysokości najwyższego i najniższego stosu rośnie. Ta różnica nie może być większa od liczby żetonów w strukturze. Stąd wynika, że operacja czyszczenia jest skończona.

b) Ile wynosi koszt zamortyzowany jednego ruchu z uwzględnieniem operacji czyszczenia stosów.

Zastosujemy metodę potencjału. Potencjał struktury definiujemy jako
4 x liczba żetonów na najniższym stosie + 2 x liczba żetonów na stosie o „środkowej wysokości". Nietrudno zauważyć, że podczas wykonywania oczyść zasadniczo wykonujemy dwa rodzaje operacji:

Zdejmujemy żeton z najniższego stosu i kładziemy go na stos środkowy – 1 jednostka potencjału za Pop + 1 jednostka potencjału za Push + 2 jednostki potencjału przenosimy na środkowy stos.

Zdejmujemy żeton ze środkowego stosu i przenosimy na stos najwyższy – 1 jednostka potencjału za zdjęcie ze środkowego stosu + 1 jednostka potencjału na przeniesienie na najwyższy stos. Załóżmy, że przenosimyy k żetonów z najniższego stosu na środkowy stos i

k’ żetonów ze stosu środkowego na najwyższy. Wówczas koszt zamortyzowany jednego ruchu wynosi:

1 (koszt położenia żetonu na któryś ze stosów)

+

2k + 2k’ (koszt przekładania żetonów)

+

[4,2,0] (zapewnienie odpowiedniego potencjału: 4, gdy pierwszy żeton kładziemy na najniższy stos, 2 – na środkowy, 0 – na najwyższy)

+

-2k (żetony z najniższego stosu przenosimy na środkowy)

-2k’ (żetony ze środkowego stosu przenosimy na najwyższy)

=

1 + [4,2,0].

Potencjał jest nieujemny i dla stosów bez żetonów jego ma wartość 0.