AISD lista 6

Dominik Szczepaniak

September 20, 2024

1 Zadanie 3

Ukorzenimy drzewo w centroidzie i uzyjemy algorytmu z wykladu dla drzew ukorzenionych.

2 Zadanie 4

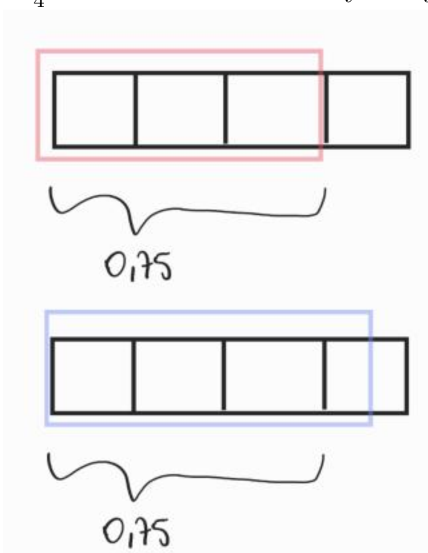
ALGORYTM HOARE'A

```
def mediana(arr, l, r, k): # mediana to k-ty najmniejszy element

    if (k > 0 and k <= r - l + 1):

        index = partition(arr, l, r)

        if (index - l == k - 1):
            return arr[index]
        if (index - l > k - 1):
            return mediana(arr, l, index - 1, k)
        return mediana(arr, index + 1, r, k - index + l - 1)
    return INT_MAX
```

Rozpatrzmy rozmiary kolejnych podtablic powstałych w wyniku podziału.

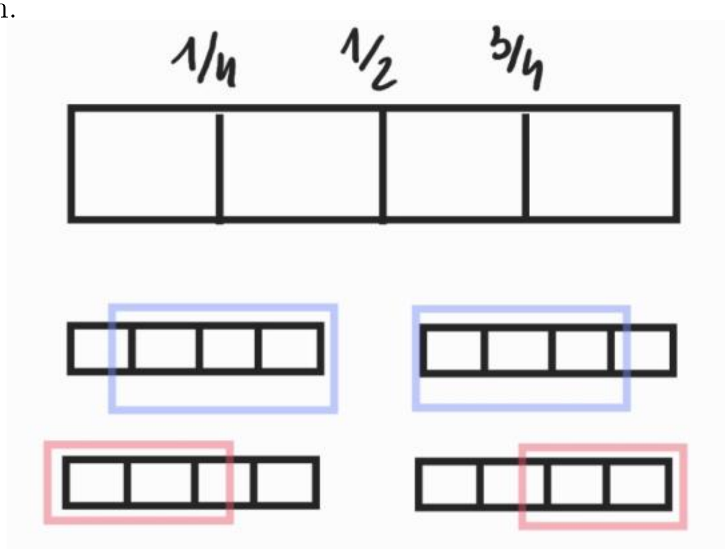
Z artykułu wiemy, że optymistyczny rozmiar kolejnej tablicy to conajwyżej $\frac{3}{4}$ rozmiaru rodzica. Jest to podział zrównoważony.

Analaogicznie podziała wykraczający rozmiarem nowo powstałej tablicy poza $\frac{3}{4}$ rozmiaru rodzica nazywamy podziałem niezrównoważonym.



Powstałą tablicy która jest zrównoważona będziemy oznaczać jako czerwone dziecko.

Tablicę, które jest niezrównoważona będziemy nazywać niebieskim dzieckiem.



Podział zrównoważony otrzymamy z prawdopodobieństwem co najmniej $\frac{1}{2}$.

Wynika to z jednostajnego rozkładu prawdopodobieństwa.

Pivot może zajmować jedną z n pozycji. Jeżeli zwrócimy uwagę na rozmiar powstałej w w wyniku takiego podziału lewej podtablicy, to otrzymamy, że ten podział może mieć długość od 0 do $n-1$.

Niezrównoważony podział wystąpi, gdy lewy podział będzie mniejszy niż niż $\frac{1}{4}n-1$ lub większy niż $\frac{3}{4}n$. A na to jzansa wynosi również $\frac{1}{2}$.

Tak jak ustaliliśmy dzeci czerwone to zrównoważone dzieci, a niebieskie to dzieci niezrównoważone. W takim razie w momencie, gdy przedstawiamy to za pomocą drzewa, korzeń będzie czerwony. Każdy rodzic będzie miał jedynie jefno dziecko, ponieważ w momencie gdy 'odcinamy' w algorytmie jakąś cześć tablicy to ta odcięta cześć juz nas nie interesuje bo wiemy że tam nie będzie k-tego najmniejszego elemetnu (mediany).

Stwórzym coś takiego jak poziom czerwonych podtablic.

Oznaczym to zmienną red_level_numer, gdzie numer będzie odpowiednią liczbą oznaczającą poziom zagłębienia, np. korzeń znaduje sie na poziomie red_level_0.

Łatwo zauważyć, żę rozmiar podtablicy na poziomie red_level_l jest niewiększy niż

$len(tab\_red\_level\_l) \leq \left(\frac{3}{4}\right)^l \cdot n$

Jest tak ponieważ wiemy, że dziecko czerwonej tablicy, które też jest czerwone może być conajwyżej $\frac{3}{4}$ rozmiaru rodzica.

Teraz rozważmy niebieskie podtablice-dzieci.

Będziemy chcieli je liczyć przy okazji czerwonych podtablic. W ten sposób oszacujemy złożoność.

Logiczne jest to że skoro wiemy, ze szansa na podział niezrównoważony *wynosi* $\frac{1}{2}$ to szansa na to że czerwony wierzchołek będzie miał niebieskie dziecko również wynosi $\frac{1}{2}$.

Wtedy zamiast wyliczyć to dziecko na poziomie red_level_l w czasie równym długosci tablicy - oznaczmy jako m, to wyliczymy go w czasie $m + m'$ gdzie $m' \in (\frac{3}{4}m, m)$.

W takim razie wiemy, że dla poziomu l mamy

$E(l) = \frac{1}{2}m + \frac{1}{2}(m + m')$

$E(l) = m + \frac{1}{2}m'$

Wartość oczekiwana czasu wyliczenia takiej podtablicy to $m + \frac{1}{2}m'$ oszacujmy z góry m' przez m. (W najgorszym wypasku niebiekie dziecko będzie prawie wielkości swojego rodzica)

$E(l) \leq red\_level\_l\_size + \frac{1}{2}(red\_level\_l\_size)$

Analogicznie będziemy musieli dodać kolejne niebieskie dzieci. Prawdopodobieństwo k-tego niebieskiego dziecka (k-tego wynika) jest *równe* $\frac{1}{2^k}$. Zatem wartość oczekiwana rozwija sie w szereg geometryczny

$E(l) \leq red\_level\_l\_size(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots) \leq 2 \cdot red\_level\_l\_size$

Czyli wartość oczekiwana wyliczenia każdego czerwonego dziecka na poziomie $red - l$ jest liniowa - $O(n)$.

Teraz policzmy złożoność całego algorytmu.

Czyli wyliczenie wszystkich podtablic na kolejnych red_level

$\left(1 + \frac{3}{4} + \ldots + \left(\frac{3}{4}\right)^l\right) \cdot O(n) \leq \frac{1}{1-\frac{3}{4}} \cdot O(n) = 4 \cdot O(n) = O(n)$

3 Zadanie 5

Kolejka priorytetowa to struktura, która przechowuje dane, które posiadaja określony priorytet. Zastosowanie takiej struktury umożliwia łatwy dostęp do elementu o najwyższym piorytecie.

Wykonujemy na nich takie operacje jak:

isEmpty()

insert(T, x)

deleteMax(T)

Dodatkowo w poleceniu zadania mowa jest o kolejce złączalnej, która dodatkowo umożliwia łączenie dwóch kolejek w jedną.

Z warunku jak został podany w zadaniu wnioskuje, że powinniśmy rozważyć drzewo lewicowe, które jest kopcem binarnym (wierzchołek może mieć dzieci: 0,1 lub 2, a każdy wierzchołek spełnia warunek że rodzic jest większy od dziecka).

Warunek świadczący o lewicowości drzewa mówi że najkrótsza możliwa droga do liści w lewym poddrzewia ma być dłuższa od najkrótszej możliwej drogi do liśćia w poddrzewie prawym.

Operacja insert polega na stworzeniu kopca jedno-elementowego, składającego się z elementu, który chcemy wsawić. Natępnie łączymy oba drzewa.

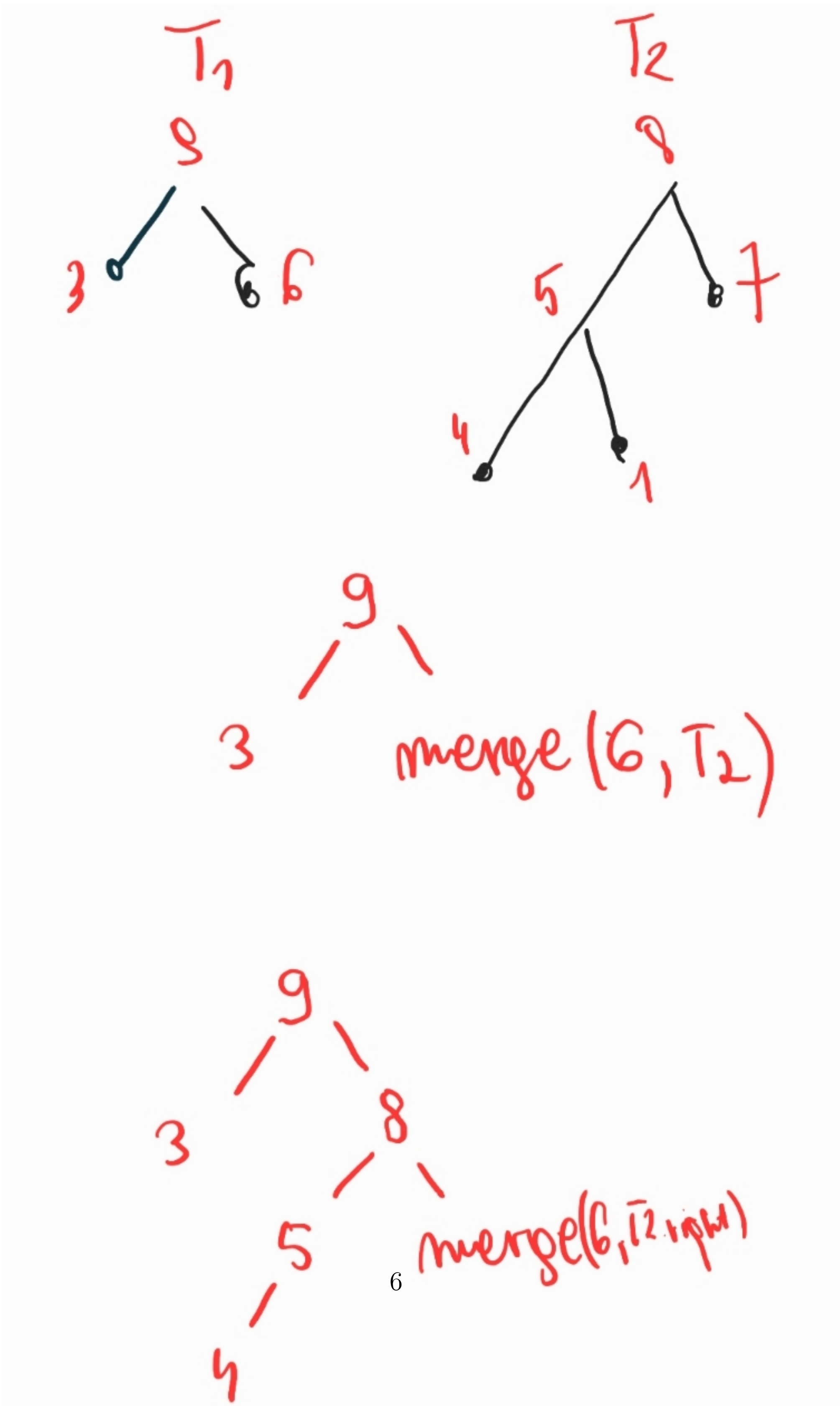
Operacja deleteMax polega na usunięciu korzenia. Widać, że w lewym poddrzewie korzenia jak i prawym poddrzewie korzenia mamy drzewa lewicowe, dlatego wystarczy wykonać na nich operację złączenia.

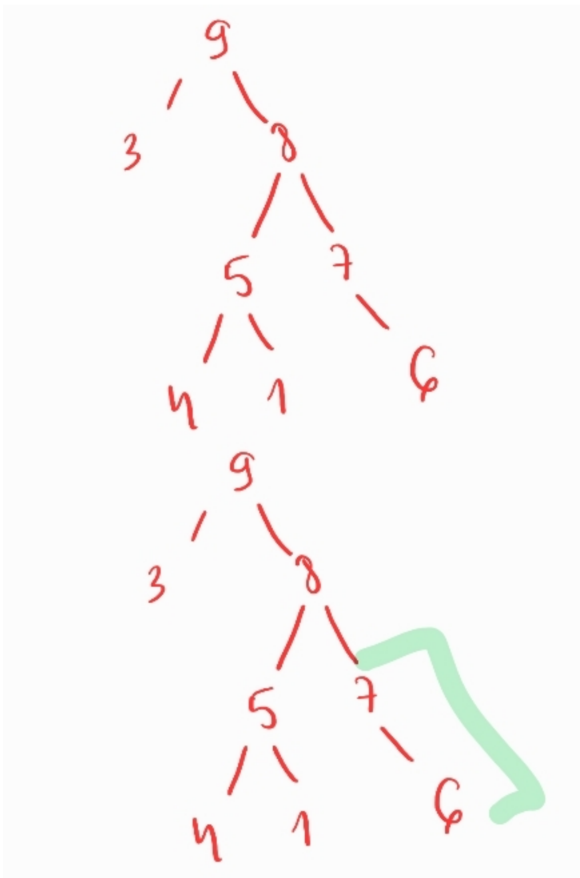
Własności:

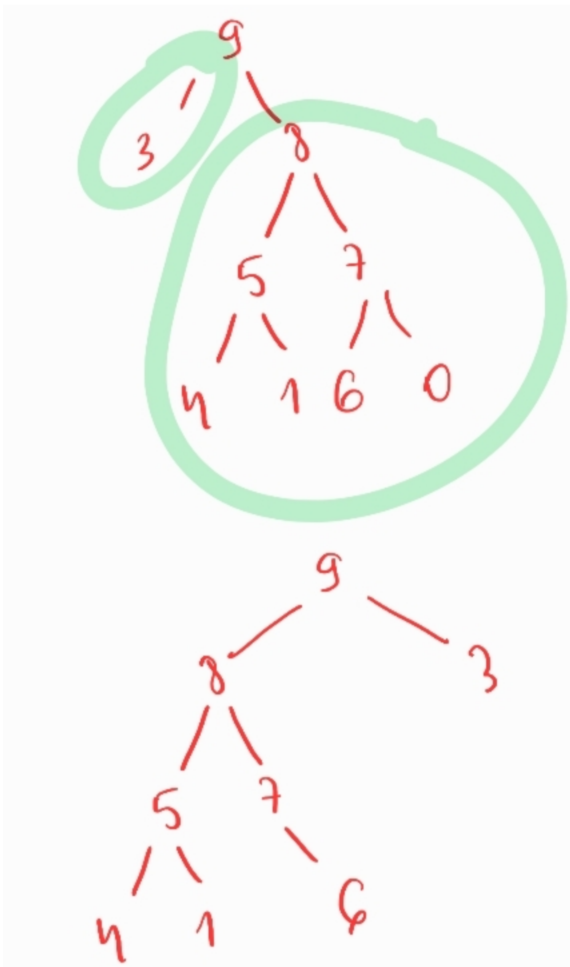
Możemy zauważyć, że jeśli przeprowadzmy ścieżkę zawsze przez prawe dziecko to najszybciej dostaniemy sie do liścia. W takim razie jeżeli długość tej ścieżki będzie równa l to łatwo zauważyć, że w całym drzewie będziemy mieli przynajmniej $2^l - 1$ wierzchołków. Wiemy, że najkrótsza ścieżka do pustego liścia lewego dziecka jest conajmniej tak długa jak najkrótsza ścieżka prawego, stad $2 * 2^l - 1 = 2^l$.

Zatem $n \geq 2^l$, a stąd $logn \geq l$

```
def scal(T1, T2):
    if isEmpty(T1):
        return T2
    if isEmpty(T2):
        return T1

    if T1.root.val > T2.root.val:
        T1.right = scal(T1.right, T2)

        if T1.left == Null:
            swap(T1.left, T1.right)
            T1.height = 1
        elif T1.right.height > T1.left.height:
            swap(T1.left, T1.right)
            T1.height = T1.right.height + 1
        return T1
    else
        T2.right = scal(T2.right, T1)
        if T2.left == Null:
            swap(T2.left, T2.right)
            T2.height = 1
        elif T2.right.height > T2.left.height:
            swap(T2.left, T2.right)
            T2.height = T2.right.height + 1
        return T2
```


T1
T2
9
3
6
8
5
7
4
1
merge(6, T2)
merge(6, T2.right)


9
3
8
5
7
4
1
6
9
3
8
5
7
4
1
6



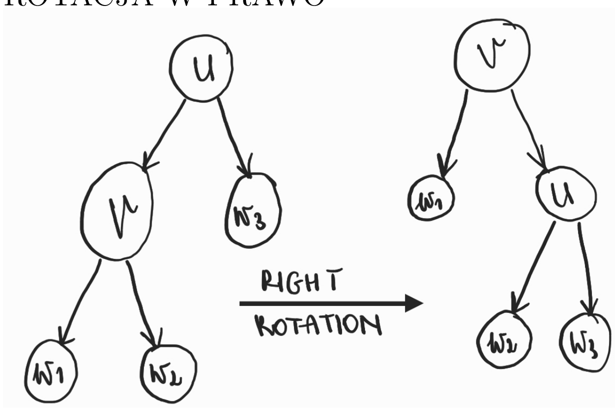
4 Zadanie 6

Drzewo BST to drzewo, które ma strukturę drzewa binarnego. Każdy węzeł posiada atrybut key, left, right, father. Klucze w tym drzewie sa przechowywane w taki sposób, aby jeśli y jest węzłem znajdującym się w lewy poddrzewie węzła x, to y.key $\leq$ x.key. Jeśli y jest węzłem znajudjącym się w prawym poddrzewie węzła x, to y.key $\geq$ x.key.

Na drzewach przeszukiwań binarnych możemy przeprowadzić wiele operacji. Insert, Delete, Search, Max, Min, Transplant

Możemy również przeprowadzić operacje rotacji w lewo i w prawo. Są to operacje odwrotne.

ROTACJA W PRAWO



ROTACJA W LEWO to odwrotna operacja do rotacji w prawo

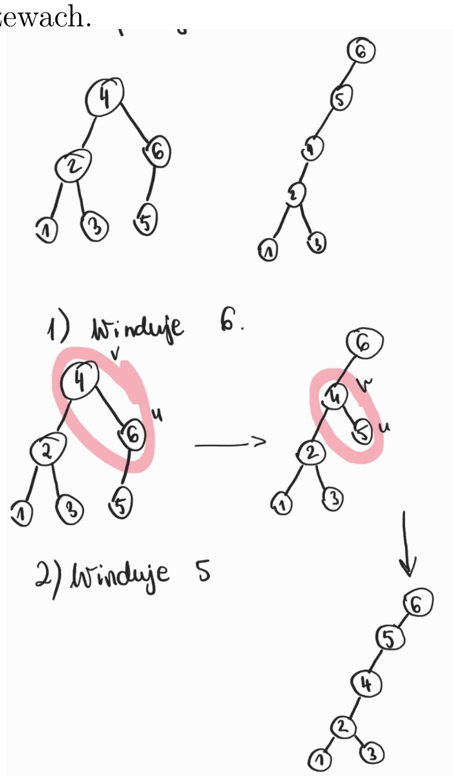
Rozpiszmy jak wygląda rotacja w lewo

```
def left_rotate(T, v):
    u = v.right
    v.right = u.left

    if u.left != null:
        u.left.father = v
    u.father = v.father

    if v.father == null:
        T.root = u
    else if v == v.father.left:
        v.father.left = u
    else:
        v.father.right = u

    u.left = v
    v.father = u
```

wierzchołek v to wierzchołek, który chcemy zrotować z wierzchołkiem, który jest jego prawym synem. Oznaczać go będziemy jako u.

Pokażemy, że dowolne drzewo BST da się przekształcić w inne dowolne drzewo BST (,które jest z budowane z tych samych kluczy).

Wystarczy pokazać, że wiedząc jaki jest korzeń drzewa w drzewie docelowym jesteśmy wstanie "wywindować" rotacjami odpowiedni wierzchołek do korzenia w drzewie startowym, a następnie zrobić to samo na jego pod drzewach.



Rozwiązanie to działa ponieważ za każdym razem gdy rotujemy wierzchołek, który chcemy aby był korzeniem, z jego rodzicem to zwiększamy poziom jego położenia, a co za tym idzie kiedyś dotrzemy do korzenia.

Zatem plan działania jet taki:

1) Patrzymy na drzewo docelowe i patrzymy jaki jest oczekiwany korzeń.

2) Znajudjemy wartość tego korzenia w drzewie startowym.

3) Używamy algorytmu do podciągnięcia wierzchołka do korzenia.

Wiemy, że po wykonaniu jednego takiego ciągu operacji oba drzewa (startowe i docelowe) mają w prawym i lewym poddrzewie odpowiednio takie same wartości. WYnika to z własności drzew BST.

Jedyny co może się między nimi różnić to kształt tych poddrzew.

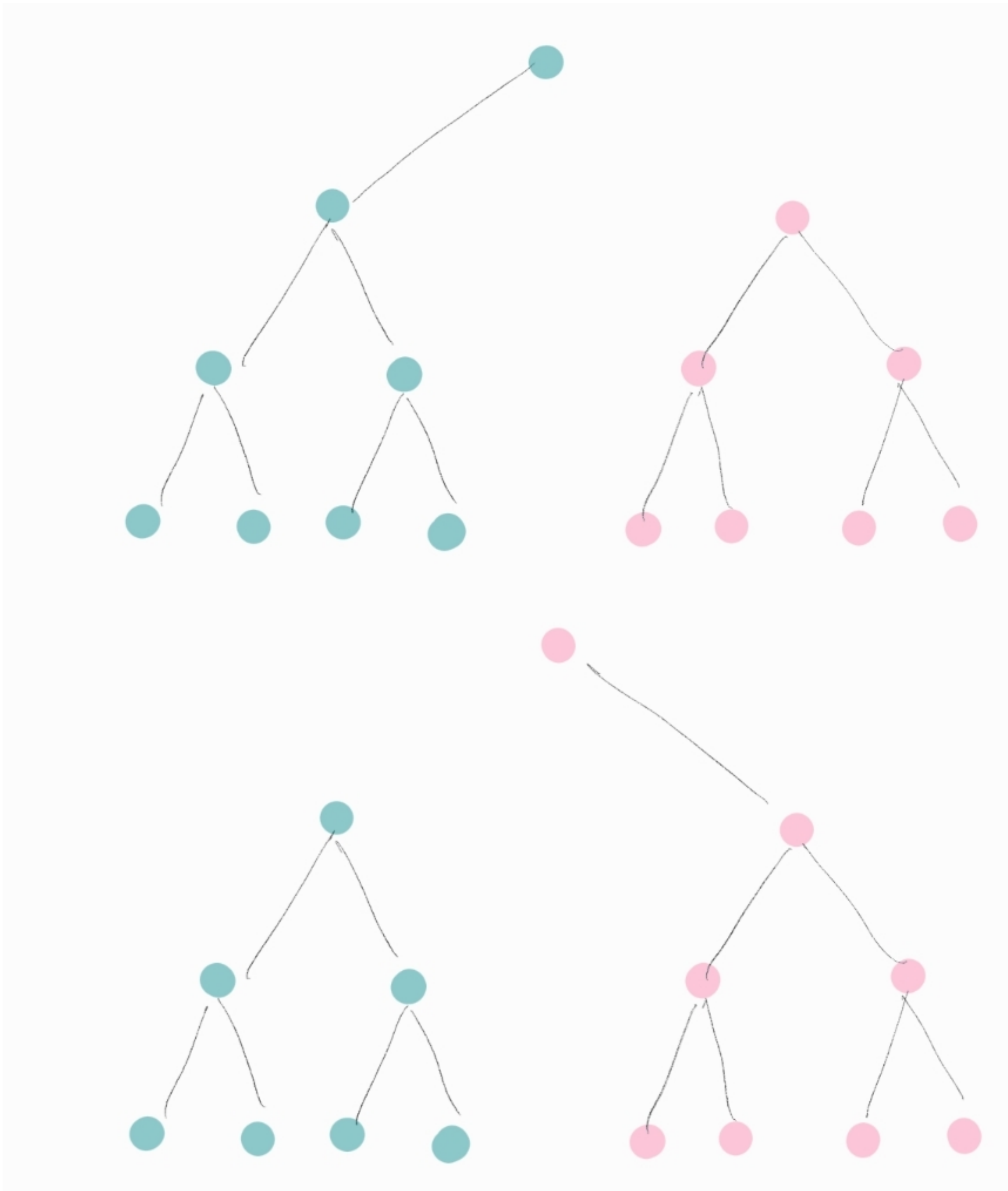
W takim razie należy kontunować wywołanie tych 3 kroków na dwóch poddrzewach drzewa startowego

Złożoność czasowa: O(n*logn)

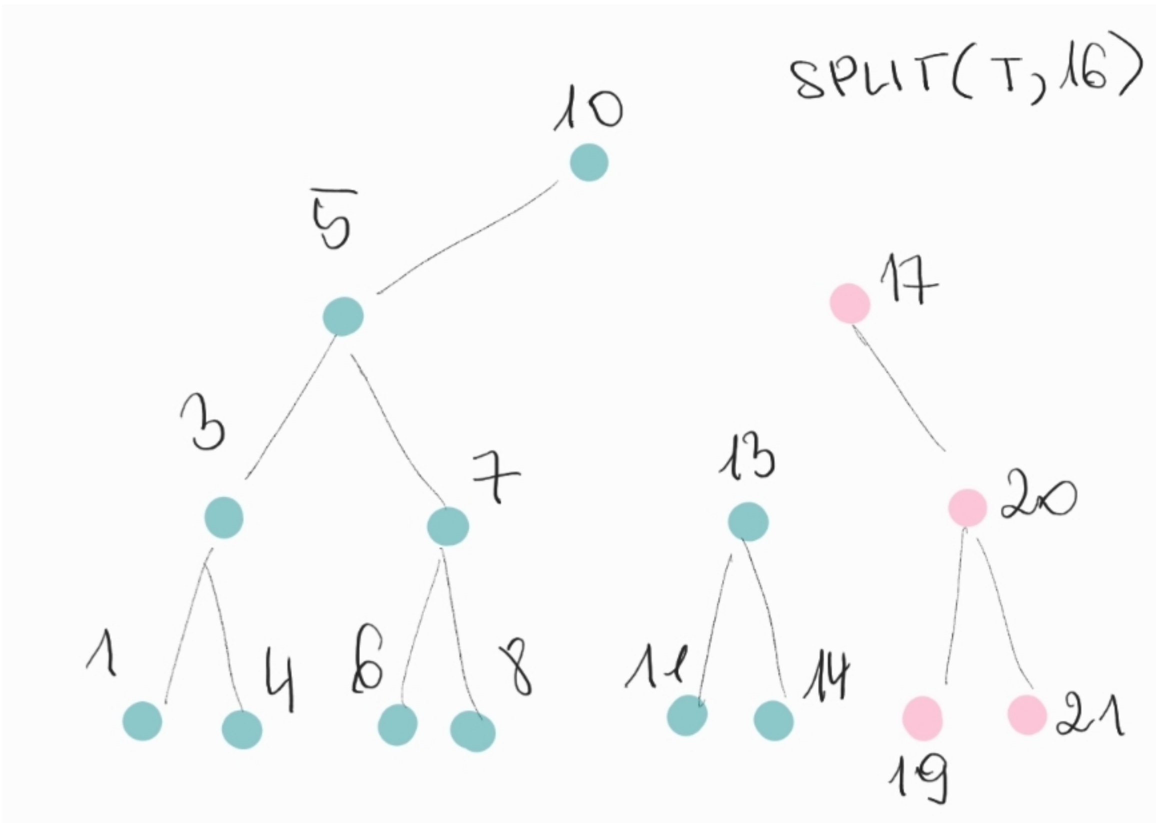
5 Zadanie 7

IDEA:

Będziemy schodzić po drzewie od korzenia w dół i w każdym takim kroku odpowiednio rozdzielimy je na dwa poddrzewa. Jedno o elementach mniejszych niż klucz i drugie większe badź równe kluczowi.



Przykład;

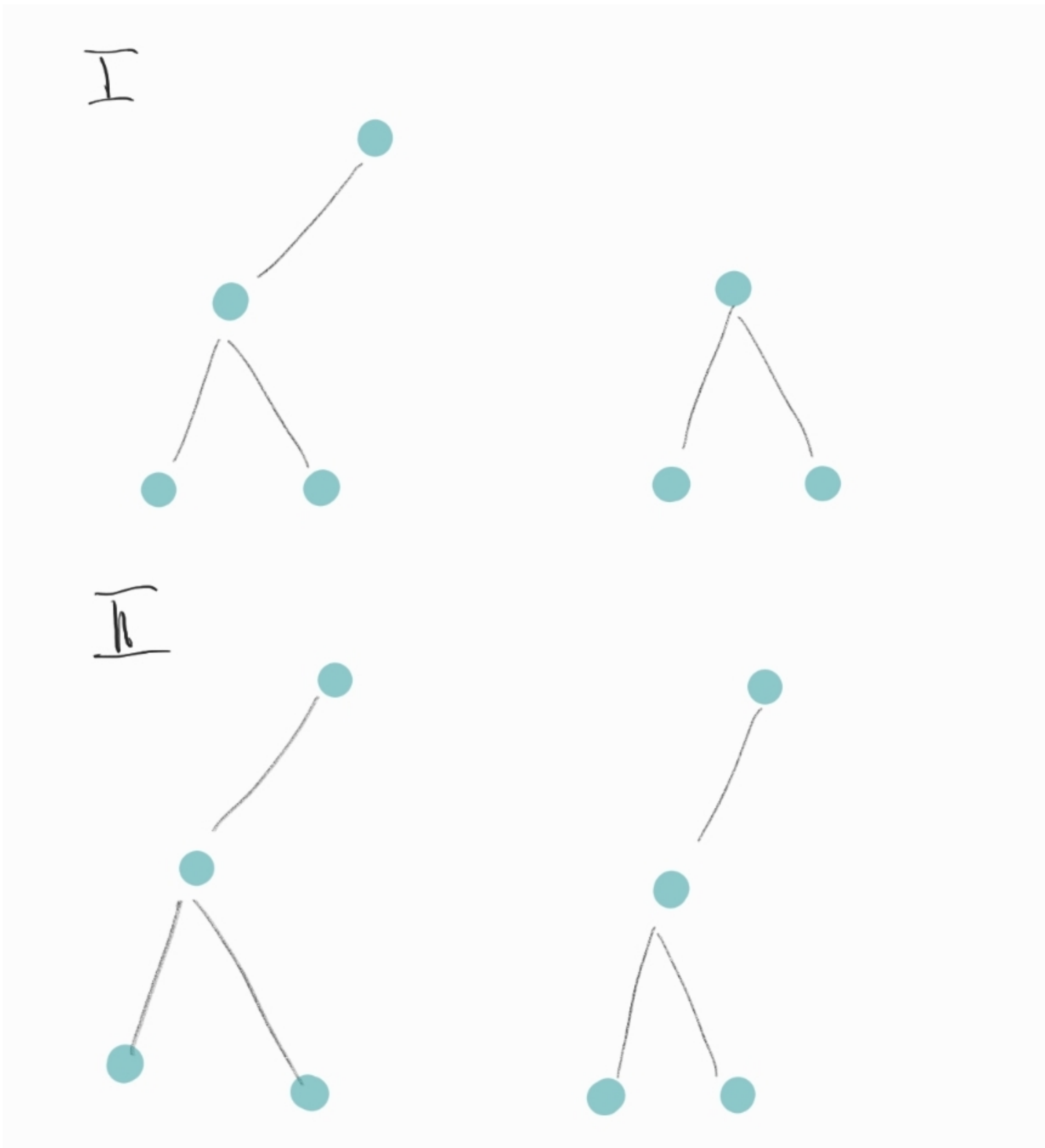


Procedura podziału zajmie nam $O(lgn)$, ponieważ wysokość drzewa jet logarytmiczna.

Stworzymy operacje Join(T1, x, T2), która jako argumenty przyjmuje dwa drzewa, które są drzewami AVL.

Element x jest większy od wszystkich elelemtnów w drzewie T1 i mniejszy od wszystkich elementów w drzewie T2.

Mamy dwa przypadki:



1) Joinujemy dwa drzew z czego jedno jest avl drugie ma "wystającego" rodzica. Wtedy "wystającego" rodzica odcinamy, drzewo staje sie drzewem AVL a odciety korzeń zapisujemy pod x.

2) Joinujemy dwa drzewa, które nie są AVL, z jedynm z nich postępujemy

tak samo jak w przypadku 1, a w drugim odcinamy korzeń i po operacji join wykonujemy operacje insert na drzewie AVL.

Łącząc dwa drzewa AVL nie mamy pewności czy całe drzewo będzie AVL. Wiemy za to, że zaborzenie takiego drzewa będzie na poziomie korzenia. Zatem musimy wykonać odpowiednią ilość rotacji.

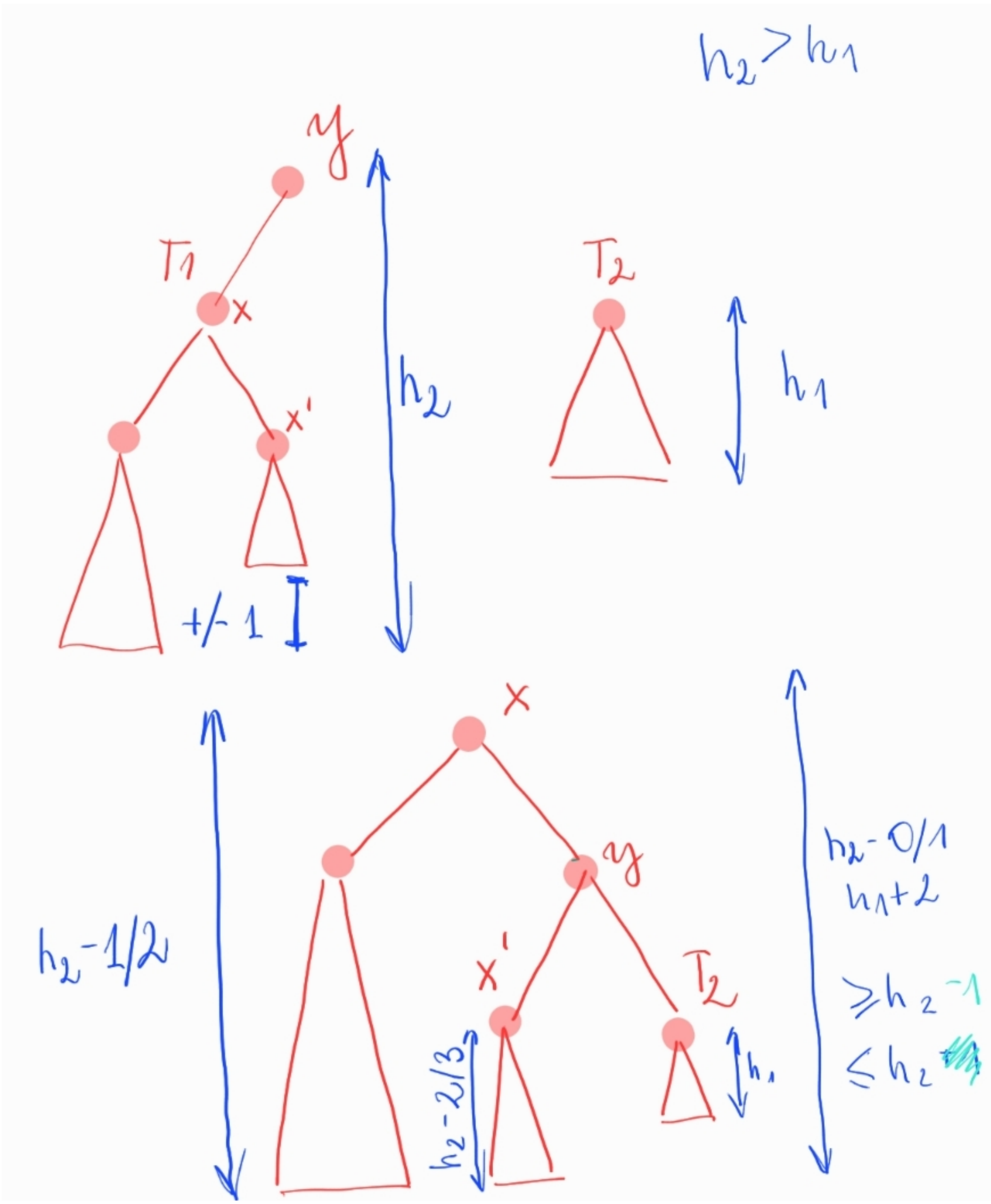
Jeśli balans jest dodatni, czyli lewe drzewo zaburza to wykonujemy prawą rotację, wpp. wykonujmey lewą rotację.

Joinować będziemy dwa drzewa o najmniejszej wysokości, tak aby wykonać jak najmniej rotacji. Oznaczmy wysokości drzew od najmniejszego do największego - $h_1, h_2, h_3, ..., h_m$ i wiemy, że $m \leq log_2 n$

Policzmy ile roatacji wykonujemy:

$h_2 - h_1 + h_3 - h'_2 + h_4 - h'_3 + \ldots + h_m - h_{m-1} \leq h_m - h_1 \leq h_m$

$-h_1 + (h_2 - h_2) + (h_3 - h_3) + \ldots + (h_{m-1} - h_{m-1}) + h_m = h_m - h_1 \leq h_m$
$\leq \log_2 n$


h2 > h1
y
T1
x
x'
h2
T2
h1
+/- 1
x
y
h2 - 1/2
x'
T2
h2 - 2/3
h1
h2 - 0/1
h1+2
≥ h2 -1
≤ h2 +1

$$(h_2 - h_1) + (h_3 - h_2') + (h_4 - h_3')$$

$$\downarrow$$

$$h_2' \geq h_2 - 1$$

$$\leq h_2$$

$$h_2 - h_1 + h_3 - (h_2 - 1) + h_4 - (h_3 - 1)$$

$$\cancel{h_2} - h_1 + \cancel{h_3} - \cancel{h_2} + 1 + \cancel{h_4} - \cancel{h_3} + 1$$

$$h_m - h_1 + 1 + 1 + 1 + 1 \ldots$$

$$\underbrace{h_m - h_1}_{\leq \log n} + \log n \leq 2 \log n$$

$$\Theta(\log n)$$

6 Zadanie 8

Proponowana struktura: drzewa czerwono-czarne

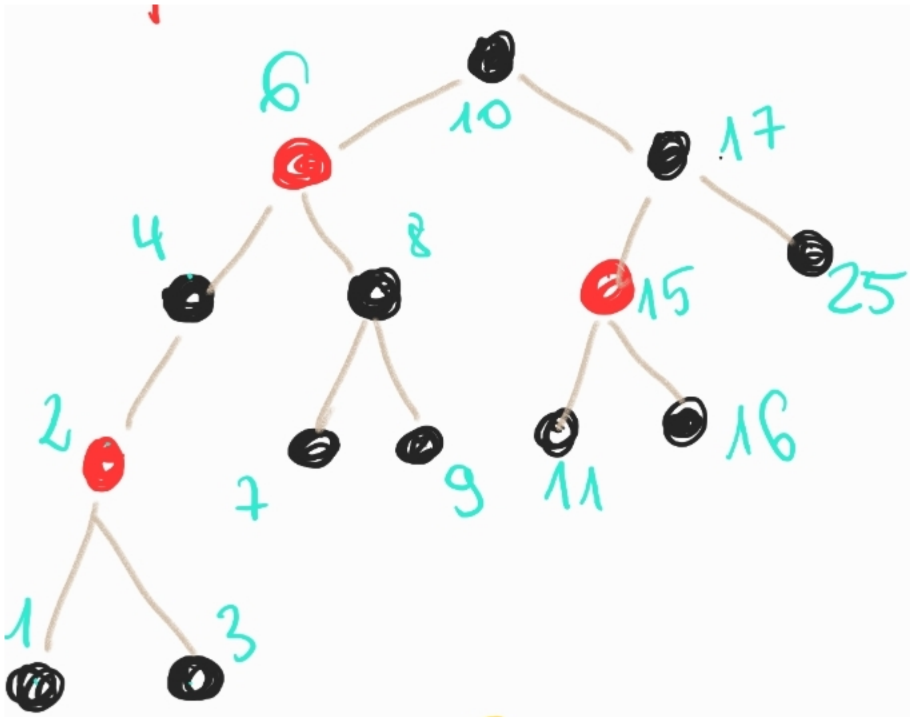
Drzewo czerwono czarne to drzewo BST o włsnościach:

- Każdy węzeł jest albo czerwony albo czarny
- Korzeń jest czarny
- Gdy węzeł jest czerwony, to obaj jego synowie są czarni
- Każda ścieżka z ustalonego węzła do liścia ma tyle samo czarnych węzłów

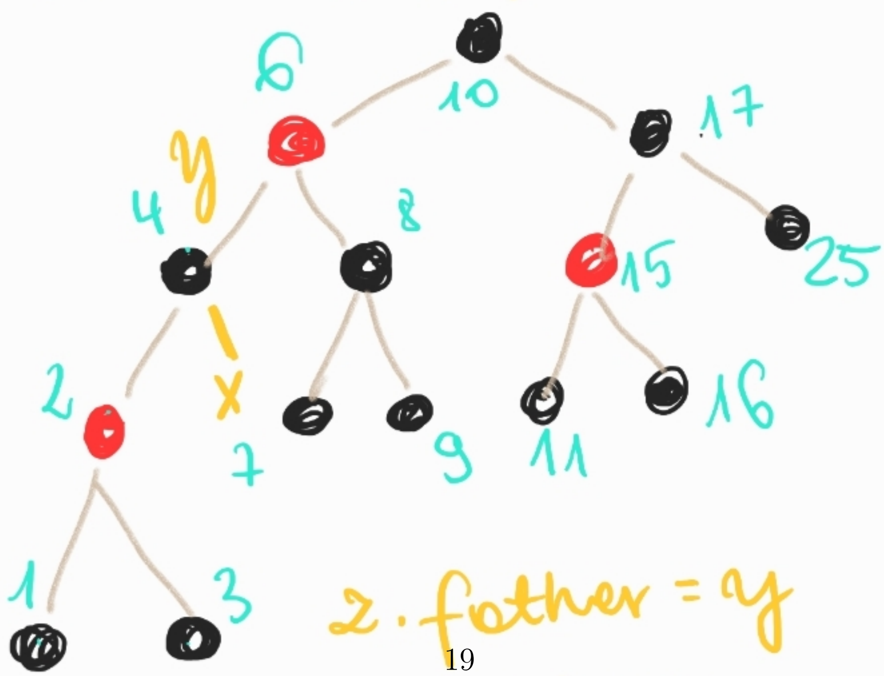
Dodatkowo będziemy pamiętać wartość minDiff - przy dodawaniu nowego wierzchołka musimy znaleźć pierwszy element większy od niego i pierwszy element mniejszy od niego i wziąć mniejszą z różnic.

OPERACJA INSERT

```
def insert(T, z):
    minDiff = min(minDiff, ceiling(z) - z)
    minDiff = min(minDiff, z - floor(z))
    y = T.nil
    x = T.root
    #rozpoczynamy przeglad w korzeniu i przeciagamy na dol odpowiedn
w lewo
    #lub w prawo az napotkamy wartosc nil; tam wstawimy nasz element
    while x != T.nil:
        y = x
        if z.key < x.key:
            x = x.left
        else x = x.right
    z.father = y
    #jesli okaze sie ze element nie ma ojca to jest to korzen
    if y == T.nil:
        T.root = z
    #ustawiamy odpowienio elemetn na lewo lub prawe dziecko
    elif z.key < y.key:
        y.left = z
    else y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = 'RED'
    insert_fix(T, z)
```



INSERT (T, 5)



z.father = y

y.right = z

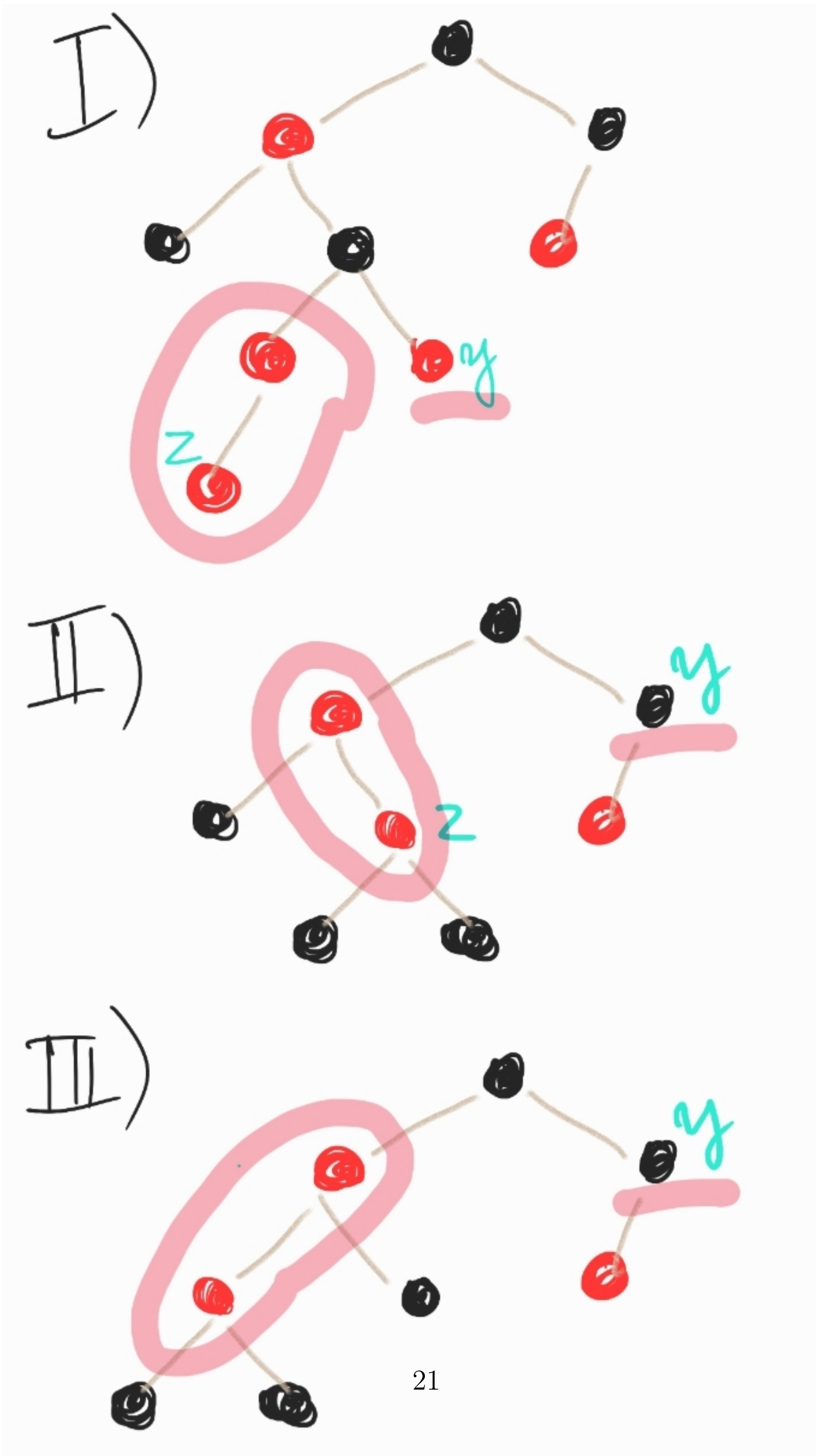
z.left = T.nil

Mamy 3 przypadki, które niszczą nam struktuę drzewa czerwono-czarnego:

1) brat ojca wstawionego elemetnu jest czerwony

2) brat ojca wstawionego elemetnu jest czarny i wstawiony element jest prawym synem

3) brat ojca wstawionego elemetnu jest czarny i wstawiony element jest lewym synem


I)
y
z
II)
y
z
III)
y

```
def insert_fix(T, z):
    if z.father == z.father.father.left:
        y = z.father.father.rigth
        if y.color == 'RED':
            z.father.color = 'BLACK'
            y.color = 'BLACK'
            z.father.father.color = 'RED'
            z = z.father.father
        else
            if z == z.father.right:
                z = z.father
                left_rotate(T,z)
            z.father.color = 'BLACK'
            z.father.father.color = 'RED'
            right_rotate(T, z.father.father)
    else
        y = z.father.father.left
        if y.color == 'RED':
            z.father.color = 'BLACK'
            y.color = 'BLACK'
            z.father.father.color = 'RED'
            z = z.father.father
        else
            if z == z.father.left:
                z = z.father
                left_rotate(T,z)
            z.father.color = 'BLACK'
            z.father.father.color = 'RED'
            right_rotate(T, z.father.father)
    T.root.color = 'BLACK'
```

OPERACJA DELETE

operacja transplant służy do przesuwania poddrzew w drzewie wyszukiwań binarnych. Wstawiw jedno poddrzewo w miejsce drugiego w jego ojcu. Ojciec u staje się ojcem v, a v zostaje synem ojca u.

```
def transplant(t, u, v):
    if u.father == T.nil:
        T.root = v
```

```
    elif u == u.father.left:
        u.father.left = v
    else u.father.right = v
    v.father = u.father

def delete(T, z):
    y = z
    y-original-color = y.color
    #gdy lewo poddrzewo jest puste
    if z.left == T.nil
        x = z.right
        transplant(T, z, z.right)
    #gdy prawe poddrzewo jest puste
    elif z.right == T.nil
        x = z.left
        transplant(T, z, z.left)
    else
        y = minimum(z.right)
        y-orginal-color = y.color
        x = y.right
        if y.father == z:
            x.father = y
        else
            transplant(T, y, y.right)
            y.right = z.right
            y.right.father = y
        tranplant(T, z, y)
        y.left = z.left
        y.left.father = y
        y.color = z.color
    if y-orginal-color == 'BLACK'
        delete-fix(T, x)

def delete-fix(T, x):
    while x != T.root and x.color == 'BLACK':
        if x == x.father.left:
            w = x.father.right
            if w.color == 'RED':
```

```
            w.color = 'BLACK'
            x.father.color = 'RED'
            left_rotate(T, x.father)
            w = x.father.right
        if w.left.color == 'BLACK' and w.right.color = 'BLACK':
            w.color = 'RED'
            x = x.father
        else
            if w.right.color == 'BLACK':
                w.left.color = 'BLACK'
                w.color = 'RED'
                right_rotate(T, w)
                w = x.father.right
            w.color = x.father.color
            x.father.color = 'BLACK'
            w.right.color = 'BLACK'
            left_rotate(T, x.father)
            x = T.root
    else
        w = x.father.left
        if w.color == 'RED':
            w.color = 'BLACK'
            x.father.color = 'RED'
            left_rotate(T, x.father)
            w = x.father.left
        if w.right.color == 'BLACK' and w.left.color = 'BLACK':
            w.color = 'RED'
            x = x.father
        else
            if w.left.color == 'BLACK':
                w.right.color = 'BLACK'
                w.color = 'RED'
                right_rotate(T, w)
                w = x.father.left
            w.color = x.father.color
            x.father.color = 'BLACK'
            w.left.color = 'BLACK'
            left_rotate(T, x.father)
```

```
                x = T.root
    x.color = 'BLACK'
```

OPERACJA FLOOR I CEILING

```
def ceiling(T, z):
    if T == none:
        return none
    if T.value == z:
        return z
    if T.value < z:
        return ceiling(T.right, z)
    left_celing = ceiling(T.left, z)
    if left_ceiling is not none:
        return left_ceiling
    else:
        return T.value


def floor(T, z):
    if T == none:
        return none
    if T.value == z:
        return z
    if T.value > z:
        return floor(T.left, z)
    right_floor = floor(T.right, z)
    if right_floor is not none:
        return right_floor
    else:
        return T.value
```

7 Zadanie 9

Mamy klasyczne drzewo AVL z wartościami w node:
-val - wartość liczby w wierzchołku
-size - ilość wierzchołków w poddrzewie

$-even_sum - sumaelementwnapozycjachparzystych$
$-odd_sum - sumaelementwnapozycjachnieparzystych$

Drzewo AVL jest porządkowane na podstawie wielkości lewych poddrzew. Przejście in-order jest równorzędne z wypisaniem liczb po kolei. (t.j. pierwsza liczba jest maksymalnie po lewej, druga jest jej ojcem, trzecia jest prawym synem drugiej itd.)

Parzystość pozycji jest uzależniona od łącznej ilości wierzchołków na lewo od niej.

Operacje:
- insert(i, a):
Robimy split który dzieli na dwa poddrzewa:
a) z pozycjami mniejszymi od i
b) z pozycjami większymi równymi i
Tworzymy nowy node z val = a
Mergujemy lewe poddrzewo, node i prawe poddrzewo
O(logn) - bo find(i)
- delete(i):
Splitujemy drzewo na pozycji i.
Usuwamy node i
Mergujemy pozostale dwa drzewa
O(logn) - bo find(i)
- find(i):
Przechodzimy drzewo ze względu na rozmiary lewych poddrzew i znajdujemy miejsce i.
O(logn)
$-sum_of_even():$
Bierzemy wartość z korzenia
O(1)

Utrzymywanie poprawnych informacji:
Po każdym insert, delete albo rotacji updatujemy size, $even_sum$ i odd_sum.
Mamy node N z lewym synem L i prawym synem R to:
$N_parz = size(L)mod2$
$size(N) = size(L) + size(R) + 1$
jeśli N_parz jest parzysty to nie zmieniają się parzystości w R, więc:
$even_sum(N) = even_sum(L) + even_sum(R) + N.val$
$odd_sum(N) = odd_sum(L) + odd_sum(R)$
jeśli N_parz jest nieparzysty to zmieniają się parzystości w R:
$swap(odd_sum(R), even_sum(R))$

$even_sum(N) = even_sum(L) + even_sum(R)$
$odd_sum(N) = odd_sum(L) + odd_sum(R) + N.val$

8 Zadanie 10

Dla każdego wierzchołka pamiętamy czy sąsiad jest większy
0 - nie jest większy
1 - większy
jeśli 2x mamy 0 to jest równe
jeśli mamy w jednym 1 a w drugim 0 to wiemy który większy