Zadanie 1. (2pkt)

Podaj nierekurencyjną wersję procedury Quicksort, która
- działa w miejscu, tj. poza tablicą z danymi (int A[n]) używa tylko stałej (niezależnej od n) liczby komórek typu int (zakładamy, że $max(n, max\{A[i] \mid i = 1, \dots, n\})$ jest największą liczbą jaką może pomieścić taka komórka),
- czas jej działania jest co najwyżej o stały czynnik gorszy od czasu działania wersji rekurencyjnej.

```
Quicksort(A, n)
l = 1, r = n // wskaźniki na końce przedziałów
while (true)
        if (r – l < 3) // przedział długości <= 3, tak jakby dno "rekurencji"
                sort(A[l,…,r])
                l = r + 2 // lewy koniec następnego przedziału
                r = r + 3 // prawy koniec następnego przedziału (chwilowo +1)
                // wyznaczanie prawdziwego prawego końca przedziału
                while (r <= n and A[r] <= A[l])
                        r++
                if (r == n)  // posortowaliśmy już całą tablicę
                        break
        else
                pivot = ChosePivot(A, l, r) // zwraca indeks pivota przed podziałem
                s = partition(A, pivot, l, r) // zwraca indeks pivota po podziale
                m = find_index_of_max(A, s+1, r) // indeks największego elementu
                // KLUCZOWE: największy element prawego przedziału zawsze
                // będzie 1  elementem tego przedziału – odzyskuje wskaźniki
                swap(A[m], A[s+1])
                r = s – 1 // powtarzamy proces dla lewego przedziału

// zwraca indeks pivota na przedziale [l, r] w A
ChoosePivot(A, l, r)
if (r – l < 6) // krótki przedział, pivotem będzie jego 1 element
        return 1
// dla dłuższych przedziałów pivotem będzie mediana 3 elementów
x = A[l], y = A[(l+r)/2], z = A[r]
m = median of (x, y, z)
return index of m in A
```

```
// ustawia elementy mniejsze od pivota na jego lewo a większe na prawo
partition(A[1,…,n], p, l, r)
x = A[p]
i = p
j = r
while (i < j) // dopóki istnieją elementy ze złej strony pivota
        while (A[j] > x) // szukanie elem. z prawej, który powinien być z lewej
                j = j – 1
        while (A[i] < x) // szukanie elem. z lewej, który powinien być z prawej
                i = i + 1
        if (i < j) // znaleziono 2 elementy z przeciwnie złych stron
                Swap(A[i], A[j]) // po zamianie oba będą po dobrej stronie
        else // wszystkie elementy są po dobrej stronie pivota
                return j
```

Zadanie 3. (1pkt)

Podaj algorytm sprawdzający izomorfizm drzew nieukorzenionych.

Pomysł:
Najpierw znajdujemy wierzchołki centralne obu drzew poprzez usuwanie liści tak długo, aż zostaną nam 1 lub 2 wierzchołki. Jeśli T1 będzie miał inną ilość wierzchołków centralnych niż T2, to na pewno nie są izomorficzne.
Jeśli oba mają po 1 wk. cen. to wywołujemy procedurę 4.3 z notatki 10.

Bez zmniejszenia ogólności możemy założyć, że obydwa drzewa mają tę samą:

- wysokość,
- liczbę liści na każdym poziomie.

1. $\forall_{v\ -\ \text{liść w}\ T_i}\ kod(v) \leftarrow 0$
2. **for** $j \leftarrow depth(T_1)$ **downto** 1 **do**
3. $\quad S_i \leftarrow$ zbiór wierzchołków T_i z poziomu j nie będących liśćmi
4. $\quad \forall_{v \in S_i}\ key(v) \leftarrow$ wektor $\langle i_1, \ldots, i_k \rangle$, taki że
   - $i_1 \leq i_2 \leq \ldots \leq i_k$
   - v ma k synów $u_1, \ldots, u_k$ i $i_l = kod(u_l)$
5. $\quad L_i \leftarrow$ lista wierzchołków z S_i posortowana leksykograficznie według wartości key
6. $\quad L_i^{'} \leftarrow$ otrzymany w ten sposób uporządkowany ciąg wektorów
7. $\quad$ **if** $L_1^{'} \neq L_2^{'}$ **then return** ("nieizomorficzne")
8. $\quad \forall_{v \in L_i}\ kod(v) \leftarrow 1 + rank(key(v), \{key(u) \mid u \in L_i\})$
9. $\quad$ Na początek L_i dołącz wszystkie liście z poziomu j drzewa T_i
10. **return** ("izomorficzne")

Twierdzenie 2 *Izomorfizm dwóch ukorzenionych drzew o n wierzchołkach może być sprawdzony w czasie $O(n)$.*

Zadanie 4. (1.5 pkt)
Oszacuj oczekiwany czas działania Algorytmu Hoare'a (znajdowania mediany w ciągu). Mile widziane będzie zastosowanie metody Fredmana (z artykułu załączonego na stronie wykładu).

HackMD Atiluj

Zadanie 5. (2pkt)
Niech h(v) oznacza odległość wierzchołka v do najbliższego pustego wskaźnika w poddrzewie o korzeniu v. Rozważ możliwość wykorzystania drzew binarnych, równoważonych poprzez utrzymywanie następującego warunku:
$h(lewy\ syn\ v) \geq h(prawy\ syn\ v)$ dla każdego wierzchołka v,
do implementacji złączalnych kolejek priorytetowych.

Wykorzystamy 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).
$h(lewy\ syn\ v) \geq h(prawy\ syn\ v)$ jest spełnione dzięki lewicowości drzewa.
Operacja insert polega na stworzeniu kopca 1-elementowego, składającego się z elementu, który chcemy wstawić. Następnie 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.

```
Scal(T1, T2)
if (T1 == null)
        return T2
else if (T2 == null)
        return T1
if (T1.root.key > T2.root.key)
        T1.right = Scal(T1.right, T2)
        if (T1.left == null) // naprawianie lewicowości
                swap(T1.left, T1.right) // lewe poddrzewo ma być wyższe
                T1.height = 1
        else if (T1.right.height > T1.left.height) // naprawianie lewicowości
                swap(T1.left, T1.right) // lewe poddrzewo ma być wyższe
                T1.height = T1.left.height + 1
        return T1
else
        T2.right = Scal(T1, T2.right)
```

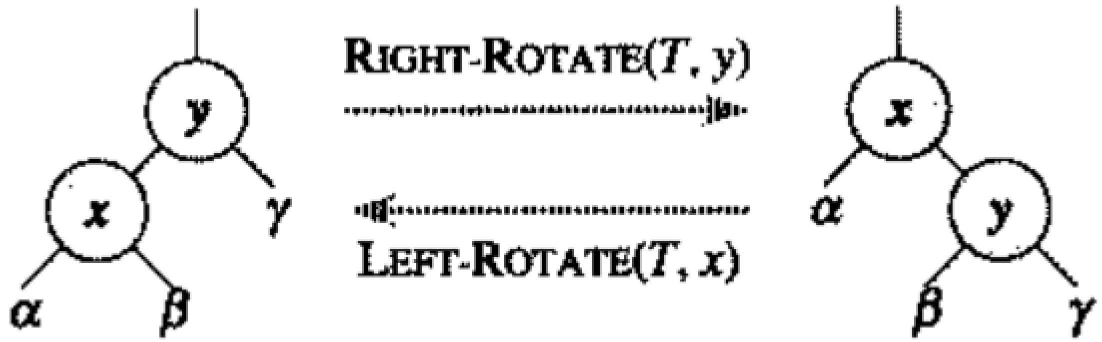
```
    if (T2.left == null) // naprawianie lewicowości
        swap(T2.left, T2.right) // lewe poddrzewo ma być wyższe
        T2.height = 1
    else if (T2.right.height > T2.left.height) // naprawianie lewicowości
        swap(T2.left, T2.right) // lewe poddrzewo ma być wyższe
        T2.height = T2.left.height + 1
    return T2
```

Zadanie 6. (0,5pkt)

Udowodnij, że każde drzewo BST można przekształcić operacjami rotacji w dowolne inne drzewo BST.

Własność 1 BST:
Każdą rotację można odwrócić, tzn. Left-rotate(Right-rotate(T)) = T

Własność 2 BST:
Rotacje nie zmieniają kolejności pre-order elementów.
Przypomnienie: wypisywanie elementów w kolejności pre-order polega na wypisywaniu najpierw elementów lewego poddrzewa, następnie korzenia, a na końcu elementów prawego poddrzewa.



Jak widać, w obu drzewach kolejność wypisywania elementów pre-order będzie następująca: alpha, x, beta, y, gamma.

Własność 3 BST (kluczowa dla dowodu):
Każde drzewo BST można przekształcić do maksymalnie niezbalansowanego drzewa BST, w którym każdy wierzchołek ma tylko lewego syna lub jest liściem.
Dowód:
Na obrazku powyżej widać, że jak wykonamy na prawym drzewie rotację w lewo
to wysokość wierzchołka x albo zmaleje (jeśli h(y)>=alpha) albo nie zmieni się.
Zatem można rekurencyjnie (zaczynając od prawych poddrzew, potem przechodząc do lewych) „usuwać” prawe poddrzewa dokonując rotacji w lewo.
Na koniec otrzymamy drzewo BST z własności 3.

Dowód zadania:
Jak mamy drzewa BST A oraz B, to znajdujemy ciąg rotacji $r_1, r_2, \dots, r_m$, który doprowadza B do drzewa z własności 3. Następnie przekształcamy A do tej postaci za pomocą rotacji $r'_m, r'_{m-1}, \dots, r'_2, r'_1$, gdzie r'_i to rotacja przeciwna do r_i. To pokazuje, że dowolne 2 drzewa BST można przekształcić jedno w drugie.

Zadanie 7. (2pkt)

Napisz procedurę Split(T, k) rozdzielającą drzewo AVL T na dwa drzewa AVL: jedno zawierające klucze mniejsze od k i drugie zawierające pozostałe klucze. Jaka jest złożoność Twojej procedury?

```
Split(T, k) // dzieli drzewo AVL na 2 drzewa t1, t2, że ∀x ∈ t1 < k ≤ ∀x ∈ t2
if (T == null)
        return <null, null>
else if (k == T.value)
        return <T.left, Join(null, k, T.right)>
else if (k < T.value)
        LL, LR = Split(T.left, k)
        return <LL, Join(LR, T.value, T.right)>
else
        RL, RR = Split(T.right, k)
        return <Join(T.left, T.value, RL), RR>

// balansuje 2 drzewa AVL różnych wysokości tak, aby można je było połączyć
wraz z korzeniem w 1 drzewo AVL
JoinRight(TL, k, TR) // zakładamy, że height(TL) ≥ height(TR) + 2
if (height(TL.right) <= height(TR) + 1) // prawie zbalansowane
        T' = Node(TL.right, k, TR)
        if (height(T') <= height(TL.left) + 1)
                return Node(TL.left, TL.value, T')
        else
                T2 = Node(TL.left, TL.value, RotateRight(T'))
                return RotateLeft(T2)
else // mocno niezbalansowane, szukamy drzewa o podobnej wysokości co TR
        T' = JoinRight(TL.right, k, TR)
        T2 = Node(TL.right, TL.value, T')
        if (height(T') <= height(TL.left) + 1)
                return T2
        else
                return RotateLeft(T2)
```

```
// łączy 2 drzewa AVL oraz korzeń w 1 drzewo AVL
Join(TL, k, TR)
if (height(TL) > height(TR) + 1)
        return JoinRight(TL, k, TR)
else if (height(TR) > height(TL) + 1)
        return JoinLeft(TL, k, TR)
return Node(TL, k, TR)
```

Złożoności:
Split: O(logn), bo Join prawie zawsze działa w O(1), jeden wyjątek - 2 if, który jest O(logn), ale on wywoła się tylko raz, bo w AVL klucze się nie powtarzają.

Join: O(logn), w tym O(1) dla 2 drzew podobnych rozmiarów
(a takie przekazuje Split)

JoinRight/Left: $O(|height(TL) - height(TR)|)$, czyli z zakresu O(1) do O(logn)

Najpierw dowód dla Joina, który jednocześnie pokaże JoinRight:
Jeśli wysokości TR,TL są +-1 równe, to oba ify się nie wykonają, a procedura tworzenia węzła jest w czasie stałym, bo działa na wskaźnikach.
W przeciwnym przypadku wywołujemy procedurę JoinRight gdy TL jest wyższe lub JoinLeft gdy TR jest wyższe.
W obu wersjach jeśli pierwszy if jest spełniony to wykonujemy 0 lub 2 rotacje, zatem wtedy czas jest stały.
Jeśli trafimy do drugiego ifa to najpierw szukamy takiego skrajnie prawego poddrzewa TL, które jest podobnej wysokości, co oznacza $O(|height(TL) - height(TR)|)$ wywołań rekurencyjnych, a potem jak w 1 ifie reszta operacji jest w czasie stałym, stąd taka złożoność czasowa.

Dowód dla Splita:
Dowód przeprowadzimy gdy wszystkie poddrzewa łączone są po lewej stronie.
Nazwiemy je (od dołu do góry) $T_1, T_2, T_3, \dots, T_l$, wtedy zachodzi $r(T_1) \le r(T_2) \le r(T_3) \le \cdots \le r(T_l)$, gdzie r(T) to ilość elementów w lewym poddrzewie T. Łączymy T_1, T_2 w T_2', T_2', T_3 w T_3' itd. aż połączymy wszystkie poddrzewa w jedno. Wykonamy zatem l-1 operacji Join.
Własnością Join, zwracającego T jest to, że dla argumentów TL,k,TR zachodzi $\max\big(r(TL), r(TR)\big) \le r(T) \le \max\big(r(TL), r(TR)\big) + 1$, stąd pojedynczy Join wykona $O(|r(T_{i+1}) - r(T_i')|)$ operacji, stąd łączny koszt Splita to:

$$\sum_{i=1}^{l} |r(T_{i+1}) - r(T_i')| \le \sum_{i=1}^{l} (r(T_{i+1}) - r(T_i') + 2) = O\big(r(T)\big) = O(logn)$$

Zadanie 8. (1,5pkt)
Zaproponuj strukturę danych do pamiętania zbioru liczbowego i wykonywania na nim operacji: insert, delete, mindiff. Ostatnia z tych operacji zwraca jako wynik najmniejszą różnicę między dwoma elementami zbioru.

Drzewo AVL z dodatkowymi polami w każdym wierzchołku:
1. minDiff – najmniejsza różnica 2 liczb tego poddrzewa,
2. minVal – najmniejsza wartość poddrzewa,
3. maxVal – największa wartość poddrzewa.

Po każdej operacji Insert/Delete aktualizujemy najpierw min/maxVal na podstawie min/maxVal dzieci, a następnie minDiff korzystając ze zaktualizowanych wartości min/maxVal oraz wartości minDiff dzieci.

```
// aktualizowanie wartości minimalnej drzewa
UpdateMinVal(T)
if (T.left == null)
        T.minVal = T.key
else
        T.minVal = T.left.minVal

// aktualizowanie wartości maksymalnej drzewa
UpdateMaxVal(T)
if (T.right == null)
        T.maxVal = T.key
else
        T.maxVal = T.right.maxVal

// aktualizowanie minimalnej różnicy drzewa
UpdateMinDiff(T)
if (T == null or (T.left == null and T.right == null))
        T.minDiff = +inf
else if (T.left == null)
        T.minDiff = min(T.right.minDiff, T.right.minVal – T.value)
else if (T.right == null)
        T.minDiff = min(T.left.minDiff, T.value – T.left.maxVal)
else
        T.minDiff = min(T.left.minDiff, T.right.minDiff,
                        T.value – T.left.maxVal, T.right.minVal - T.value)
```

```
// aktualizuje min/max wartości oraz minDiff po rotacji w prawo
// zakładamy, że wywołuje się w ostatniej linijce standardowego rightRotate
UpdateRightRotate(T)
	UpdateMinVal(T.right) // update y
	T.maxVal = T.right.maxVal // update x
	UpdateMinDiff(T.right) // update y
	UpdateMinDiff(T) // update x

// aktualizuje min/max wartości oraz minDiff po rotacji w lewo
// zakładamy, że wywołuje się w ostatniej linijce standardowego leftRotate
UpdateLeftRotate(T)
	UpdateMaxVal(T.left) // update x
	T.minVal = T.left.minVal // update y
	UpdateMinDiff(T.left) // update x
	UpdateMinDiff(T) // update y

// Balansowanie drzewa AVL
Balance(T)
balance = GetBalance(T)
if (balance > 1 and GetBalance(T.left)>=0) // Left Left
	return rightRotate(T)
else if (balance < -1 and GetBalance(T.left)<=0) // Right Right
	return leftRotate(T)
else if (balance > 1) // Left Right
	T.left = leftRotate(T.left)
	return rightRotate(T)
else if (balance < -1) // Right Left
	T.right = rightRotate(T.right)
	return leftRotate(T)
return T

// Zaktualizuj wartości min/max/minDiff po Insert/Delete
UpdateValues(T)
	UpdateMinVal(T)
	UpdateMaxVal(T)
	UpdateMinDiff(T)
```

```
// Wstawianie elementu do drzewa AVL
Insert(T, key)
if (T == null) // znaleźliśmy liść do którego trzeba wstawić klucz
	return newNode(key)
if (key < T.value) // wstawianie do lewego poddrzewa
	T.left = Insert(T.left, key)
else // wstawianie do prawego poddrzewa
	T.right = Insert(T.right, key)
UpdateValues(T) // zaktualizuj min/maxVal, minDiff
UpdateHeight(T)
return Balance(T)

// Usuwanie elementu z drzewa AVL
Delete(T, key)
if (T == null) // nie znaleźliśmy elementu do usunięcia
	return null
if (key < T.value) // usuwanie z lewego poddrzewa
	T.left = Delete(T.left, key)
else if(key > T.value) // usuwanie z prawego poddrzewa
	T.right = Delete(T.right, key)
else // usuwanie aktualnego elementu
	if (T.left == null)
		return T.right
	else if (T.right == null)
		return T.left
	T.value = T.right.minVal
	T.right = Delete(T.right, T.value)
if (T == null)
	return null
UpdateValues(T) // zaktualizuj min/maxVal, minDiff
UpdateHeight(T)
return Balance(T)

// Tworzenie nowego liścia
newNode(key)
node.height = 1
node.left = null
node.right = null
node.minVal = key // minimalna wartość drzewa
node.maxVal = key // maksymalna wartość drzewa
```

```
node.value = key // wartość korzenia drzewa
node.mindiff = +inf // minimalna różnica drzewa
return node
```

Zadanie 9. (1.5pkt)

Bolesną dolegliwością związaną z drzewami AVL jest konieczność poświęcenia dwóch bitów w każdym węźle na pamiętanie współczynnika zrównoważenia. Zastanów się, czy aby na pewno mamy do czynienia z „koniecznością”

Wskazówka od MK – lewa wysokość >= prawa wysokość
Wskazówka od PRz – rozwiązać najpierw łatwiejsze zadanie z 1 bitem
// Do dokończenia