Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Kinga Władzińska

ALGORYTMY & STRUKTURY DANYCH

1. POJĘCIA WSTĘPNE
   - 1.1 algorytmy
   - 1.2 rekurencja vs iteracja
   - 1.3 pseudokod

2. ODWROTNA NOTACJA POLSKA - ONP

3. ALGORYTMY DLA TABLIC
   - 3.1 insertion sort
   - 3.2 selection sort
   - 3.3 bubble sort
   - 3.4 shell sort
   - 3.5 quick sort
   - 3.6 partition

4. KOPCE I ALGORYTMY KOPCA
   - 4.1 heapify
   - 4.2 heap sort
   - 4.3 heap sort
   - 4.4 merge sort
   - 4.5 kolejka priorytetowa

5. TABLICE HASZOWANE
   - 5.1 basic idea
   - 5.2 funkcje haszujące
   - 5.3 kolizje
   - 5.4 uniwersum kluczy

6. TREES
   - 6.1 binary tree
   - 6.2 K-ary tree
   - 6.3 drzewo generalne
   - 6.4 depth first search
   - 6.5 breadth first search
   - 6.6 binarne drzewo poszukiwań
   - 6.7 drzewa zrównoważone
   - 6.8 drzewa czerwono-czarne
   - 6.9 B- drzewo

7. KODOWANIE HUFFMANA

8. REPREZENTACJA GRAFU
   - 8.1 minimalne drzewo spinające
   - 8.2 algorytm Kruskala
   - 8.3 algorytm Dijkstry
   - 8.4 algorytm Prima
   - 8.5 cykl eulera

9. KOPCE I DRZEWA CD.
   - 9.1 koszt operacji
   - 9.2 drzewa dwumianowe
   - 9.3 kopce dwumianowe
   - 9.4 kopiec fibonacciego

1. POJĘCIA WSTĘPNE

1.1 - algorytmy

różnice w algorytmach

- szybkość
- ilość pamięci na dane
- precyzja
- jakie dane
- komplikacje
- stabilność
- złożoność

symbole

Θ - funkcja rośnie w tym samym tempie
Ω - f. rośnie szybciej / tak samo
ω - f. rośnie szybciej
O - f. rośnie tak samo / wolniej
o - f. rośnie wolniej

złożoność obliczeniowa

↳ liczba obliczeń wyrażona jako funkcja rozmiarowa

× czasowa
liczba operacji wyrażona jako funkcja rozmiaru danych (f. wykładnicza)

× pamięciowa
rozmiar dodatkowej pamięci (bez danych wejściowych) wymagany przez algorytm wyrażony jako funkcja rozmiaru danych

1.2 - rekurencja vs iteracja

każdy algorytm rekurencyjny można zapisać w sposób iteracyjny i odwrotnie

Potencjalne problemy związane z rekurencją:

- głębokość rekurencji (→ złożoność pamięciowa)
- nadmiarowe obliczenia (→ czas obliczeń)

Stos pada przy zbyt głębokiej rekurencji (na stosie zapisywane są adresy powrotu rekurencji). Na stosie przechowywane są również zmienne lokalne, adresy tablic (czyli oryginalne tablice NIE kopie)

1.3 - pseudokod

- podstawienie = porównanie ==
- instrukcje złożone kończymy :
- bloki zaznaczamy wcięciem (zamiast {/}, begin / end)
- linie kontynuujemy przez / na końcu
- komentarz zaczynamy od #
- instrukcja warunkowa if warunek : czynność
  if warunek : czynność else : czynność

- pętla iteracyjna for zmienna in lista : czynność
- dla skrótu else : if → elif
- pętla while warunek : czynność else : czynność
- przerwanie pętli break, kontynuacja pętli continue
- nic nie rób → pass

> zakres w Pythonie jest listą indeksowaną od 0
> iterowanie

for element in A :

> do definiowania zakresów służy funkcja range

range (1, 10) → zwraca zakres 1, 2, 3, 4, 5, 6, 7, 8, 9

Odwrotna Notacja Polska

Obliczanie wartości wyrażenia ONP:

Wyrażenie ONP	Element	Operacja	Stos
3 5 2 * +			---
3 5 2 * +	3	na stos	3 ---
3 5 2 * +	5	na stos	5 3 ---
3 5 2 * +	2	na stos	2 5 3 ---
3 5 2 * +	*	pobierz 2 i 5 mnóż 5 * 2 wynik na stos	10 3 ---
3 5 2 * +	+	pobierz 10 i 3 dodaj 3 + 10 wynik na stos	13 ---

Przekształcenie wyrażenia arytmetycznego na wyrażenie ONP:

Wyrażenie	Element	Operacja	Stos	Wyjście
(a + b * c - d) / (e + f) =	start		---	
(a + b * c - d) / (e + f) =	(	na stos	(---	
(a + b * c - d) / (e + f) =	a	na wyjście	(---	a
(a + b * c - d) / (e + f) =	+	na stos	+ (---	a
(a + b * c - d) / (e + f) =	b	na wyjście	+ (---	a b

(a + b * c - d) / (e + f) =	*	na stos	* + (---	a b
(a + b * c - d) / (e + f) =	c	na wyjście	* + (---	a b c
(a + b * c - d) / (e + f) =	-	ze stosu na wyjście na stos	- + (---	a b c *
(a + b * c - d) / (e + f) =	d	na wyjście	- + (---	a b c * d
(a + b * c - d) / (e + f) =	)	ze stosu na wyjście ze stosu na wyjście usuń ze stosu	---	a b c * d - +
(a + b * c - d) / (e + f) =	/	na stos	/ ---	a b c * d - +
(a + b * c - d) / (e + f) =	(	na stos	(/ ---	a b c * d - +
(a + b * c - d) / (e + f) =	e	na wyjście	(/ ---	a b c * d - + e
(a + b * c - d) / (e + f) =	+	na stos	+ (/ ---	a b c * d - + e
(a + b * c - d) / (e + f) =	f	na wyjście	+ (/ ---	a b c * d - + e f
(a + b * c - d) / (e + f) =	)	ze stosu na wyjście usuń ze stosu	/ ---	a b c * d - + e f +
(a + b * c - d) / (e + f) =	koniec	ze stosu na wyjście	---	a b c * d - + e f + /

2. ODWROTNA NOTACJA POLSKA - ONP

↳ sposób zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczony jest po operandach (zapis postfiksowy), a nie pomiędzy nimi (zapis infiksowy) lub przed operandami jak w zwykłej notacji polskiej (zapis prefiksowy).

ONP ułatwia wykonywanie na komputerze obliczeń z nawiasami i zachować kolejność działań.

W normalnym zapisie arytmetycznym operatory mają „ważność". W ONP operator występuje po swoich argumentach. Dzięki czemu nawiasy stają się zbędne

$3+5*2 \longrightarrow 352*+ = 310+ = 13$
$(3+5)*2 \longrightarrow 35+2* = 82* = 16$

Do obliczenia wartości wyrażenia zapisanego w ONP potrzebujemy stosu.

- Wyrażenie ONP przeglądamy od lewej do prawej.
- jeżeli napotkamy liczbę to umieszczamy ją na stosie
- jeżeli napotkamy operator to ze stosu pobieramy dwie liczby (ostatnie) wykonujemy na nich działanie i wynik umieszczamy z powrotem na stosie
- po skończeniu przeglądania wyrażenia na szczycie stosu będzie znajdował się wynik działania

Example

Wyrażenie ONP	352*+	$\boxed{3}$52*+	3$\boxed{5}$2*+	35$\boxed{2}$*+	352$\boxed{*}$+	+
element	—	$\boxed{3}$	$\boxed{5}$	$\boxed{2}$	*	+
operacja	—	na stos	na stos	na stos	pobież 2 i 5 pomnóż 5·2 na stos	3+10
stos	—	3	53	253	10 3	13 WYNIK

Założenia ONP:

+ dodawanie
- odejmowanie
* mnożenie
/ dzielenie

wyrażenie musi być poprawne - algorytm nie sprawdza jego poprawności

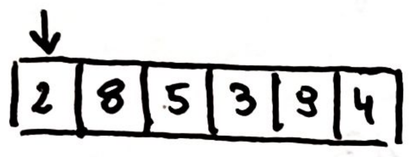
3. ALGORYTMY DLA TABLIC

3.1 - *insertion sort*

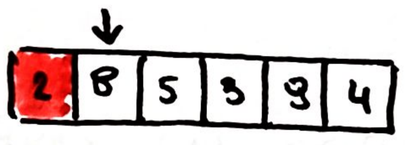
Step 1 Pracuj od lewej do prawej. Analizuj każdy element i porównuj do elementów po jego lewej stronie

Step 2 Wstaw element w odpowiednim miejscu w tablicy

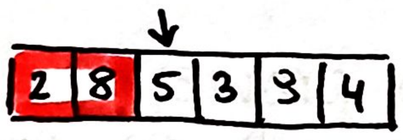
* tablica utworzy dwie partycje posortowaną i nieposortowaną. posortowana partycja będzie kolorowana.



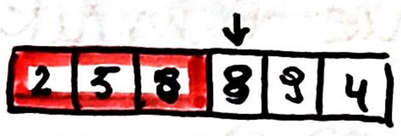
2 nic nie ma po lewej stronie
SORTED!



8 > 2
SORTED!



5 > 2
5 < 8
SORTED



3 > 2
3 < 5
SORTED

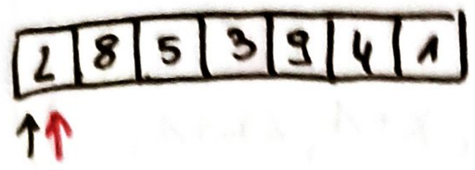
9 jest w dobrym miejscu

2 < 4
3 < 4
5 > 4
SORTED

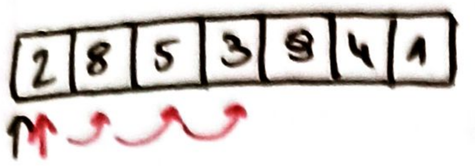
Time complexity
worst case scenario $O(n^2)$

$n \rightarrow$ ilość elementów w tablicy

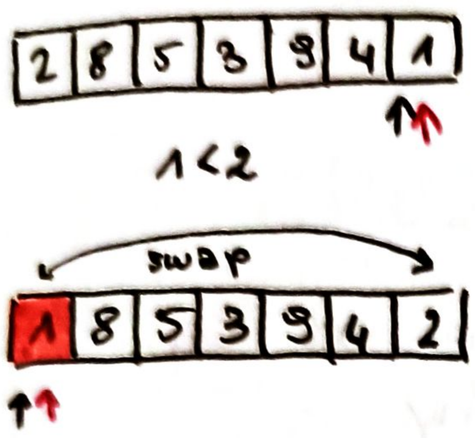
3.2 ~ selection sort



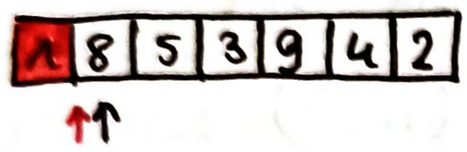
Step 1 Podczas każdej iteracji wybieramy najmniejszy element z nieposortowanej partycji i przenosimy go do posortowanej partycji



sorted partition — current minimum — current item

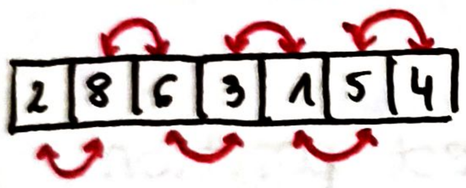


Step 2 Po znalezieniu najmniejszego elementu wstawiamy go na początek nieposortowanej partycji tym samym dodając go na koniec posortowanej partycji

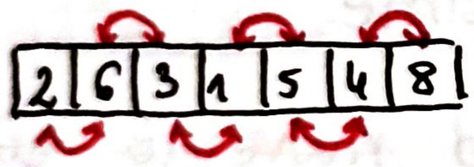


Time complexity : $O(n^2)$
↳ ilość elementów w tablicy

3.3 - bubble sort

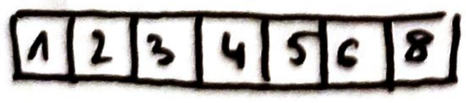


Step 1 Iterując po tablicy porównuj elementy i swapuj je miejscami jeżeli to konieczne



Powoduje to „wytarbelkowanie" największej liczby na koniec tablicy

Time complexity : $O(n^2)$



3.4 shell sort

↳ poprawiony insertion sort

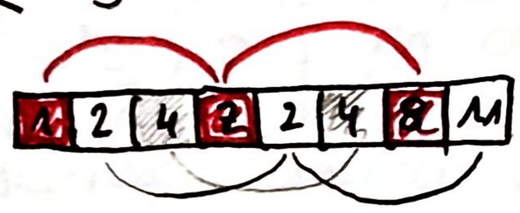
$k = \log_2 n$

Step 1 wybierz krok k

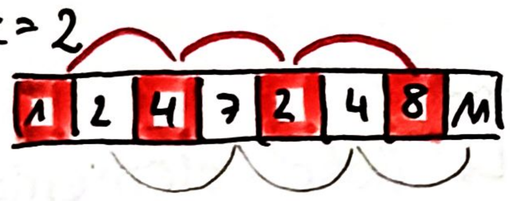
Step 2 zastosuj insertion sort dla ~~podtablic~~ $(1, k+1, 2k+1, \dots)$, $(2, k+2, 2k+2, \dots)$

Step 3 zmniejsz k aż k==1

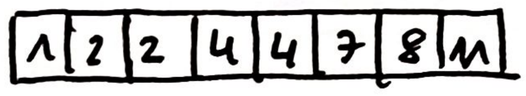
k = 3



k = 2



k = 1



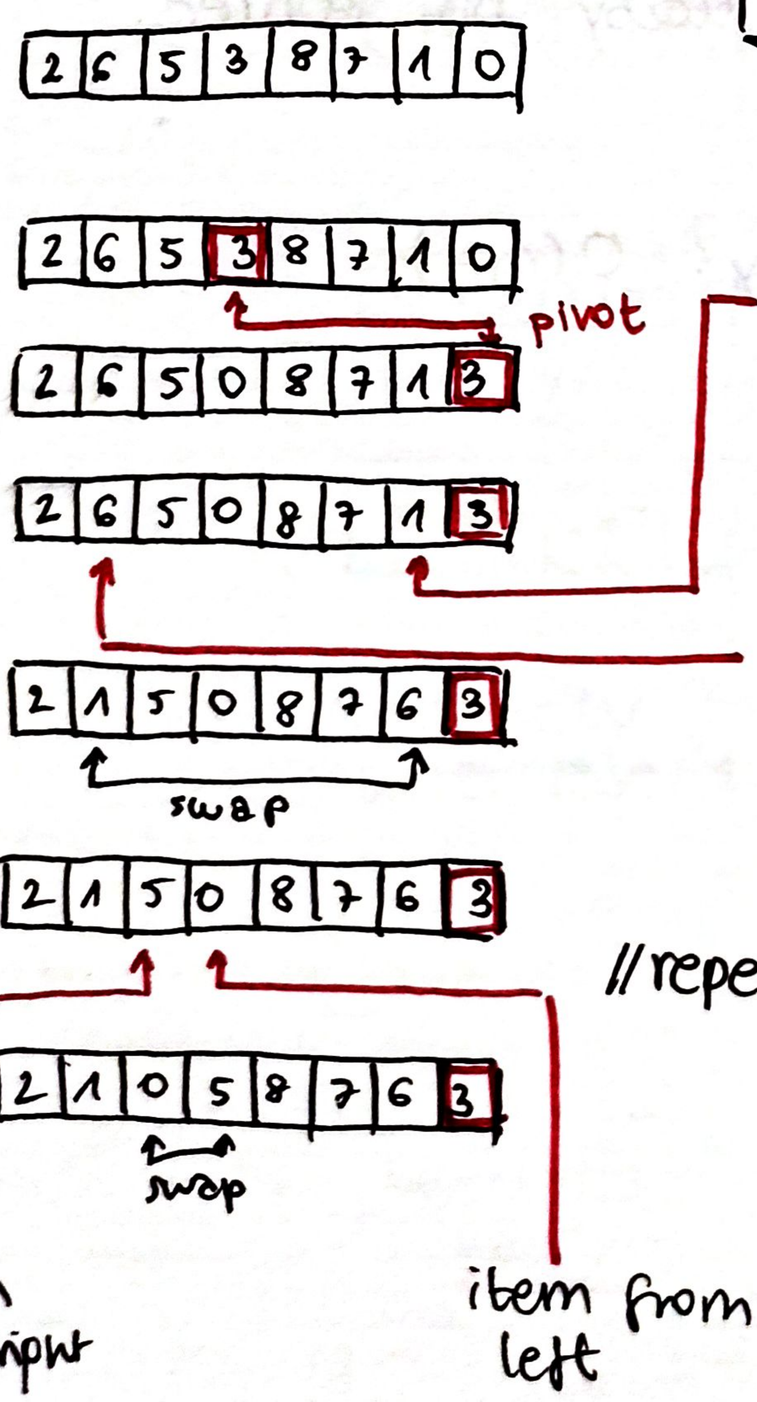
Time complexity:
jest bardzo zależne od tego jakie k dobierzemy

- k = 1 → insertion sort
- $k = 2^i - 1 \rightarrow O(n^{1.2})$
- $k = 2^r \cdot 3^q \rightarrow O(n \log^2 n)$
  $r, q \in \mathbb{N}$

3.5 quick sort

pivot - element in array that is placed in correct position. Meaning putting smaller elements before x and bigger after. All this should be done in linear time.

Step 1 Choose a pivot and move it at the end of array



Step 2 Next we look from left side to find item from right and starting from right side to find left side item

Step 3 Stop when index of itemFromLeft > index of itemFromRight

Step 4 Swap element itemFromLeft with our pivot

2	1	0	5	8	7	6	3

swap

2	1	0	3	8	7	6	5

As you can see our pivot is in the right spot. All numbers on right are larger and all numbers on left are smaller

Now we redo the whole process but on individual partitions (each partition is divided by pivot)

2	1	0	3	8	7	6	5

2	1	0

pivot

8	7	6	5

pivot

8	5	6	7

itemFromRight itemFromLeft

6	5	8	7

swap

6	5	8	7

itemFromLeft itemFromRight

6	5	7	8

HOW CHOOSE A PIVOT?

median-of-three
in this method we take first, last and middle numbers in array

2	6	5	3	8	7	1	0	4

we sort them properly and choose the middle item as a pivot (4)

Time complexity:
worst case: $O(n^2)$
average case: $O(n \cdot \log n)$

3.6 - partition

Step 1 Wybieramy element x (granicę podziału)

Step 2 Idąc od początku przedziału szukamy pierwszego elementu większego lub równego x.

Step 3 Idąc od góry przedziału szukamy ostatniego elementu mniejszego lub równego x

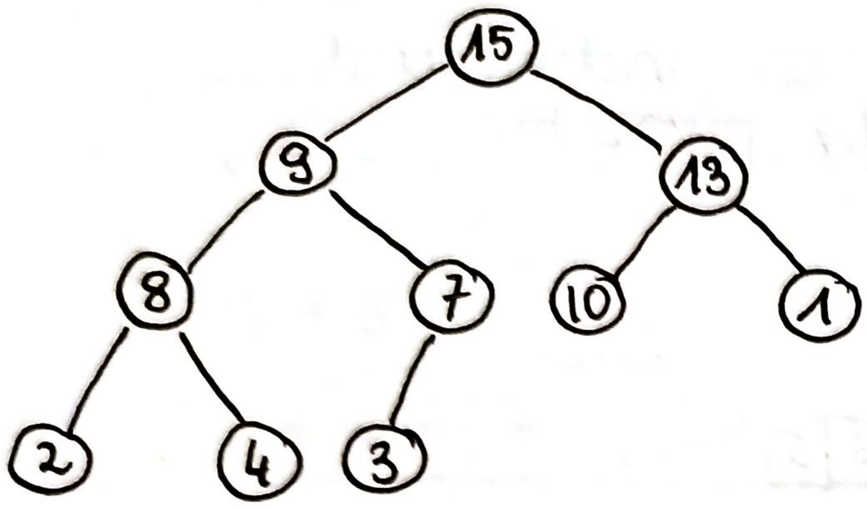
Step 4 zamieniamy znalezione elementy

Step 5 kończymy gdy poszukiwania dotrą do tego samego elementu

Pivot ( ) → wypisuje index prawego operatora jak już mną się wskaźniki → koniec sortowania

4. KOPIEC I ALGORYTMY KOPCA

↳ każdy element jest nie mniejszy / większy niż jego dzieci



15	9	13	8	7	10	1	2	4	3

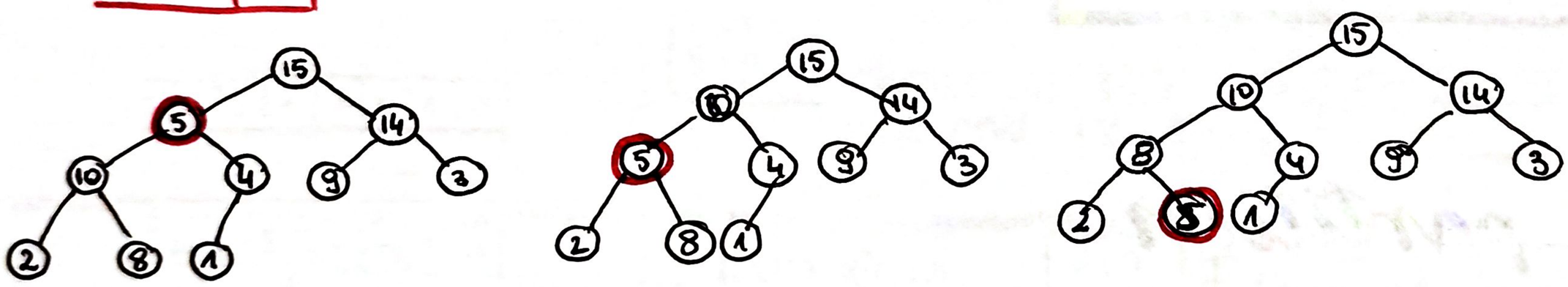
właściwość kopca może nie być spełniona tylko dla korzenia

4.1 - heapify

Step 1 rozpocznij od korzenia (w = root)

Step 2 if spełniona własność dla w i synów:
koniec
else
zmień w z większym z synów w
powtórz 2) dla nowego w

Example

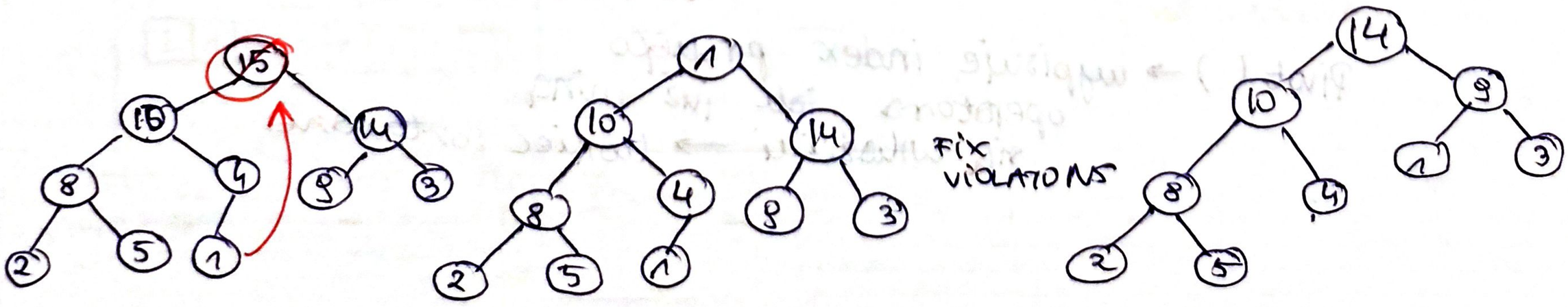


4.2 - heap sort

max heap → has a relationship parent > child

build-max-heap → tworzy max heap z nieposortowanej tablicy

USUWANIE Z KOPCA



Tworzenie kopca z całej tablicy
> przywracamy własność kopca dla coraz większych kopców

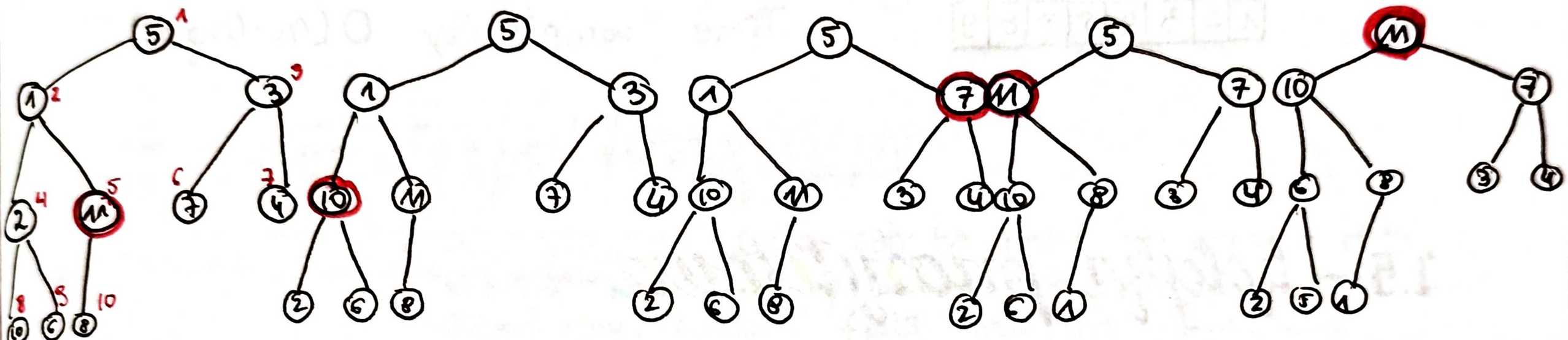
Step 1 Dla kolejnych (coraz mniejszych) kopców: wybór największego elementu (korzeń kopca)

Step 2 Zmiana z ostatnim elementem

Step 3 Zmniejszenie rozmiarów kopca

Step 4 Przywrócenie własności kopca

BUDOWA KOPCA



5	1	3	2	11	7	4	10	6	8

11	10	7	6	8	3	4	2	5	1

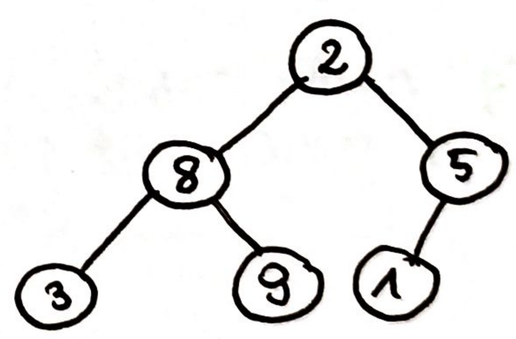
4.3 - heap sort

Step 1 Create max heap

Step 2 Remove largest item

Step 3 Place item in sorted partition

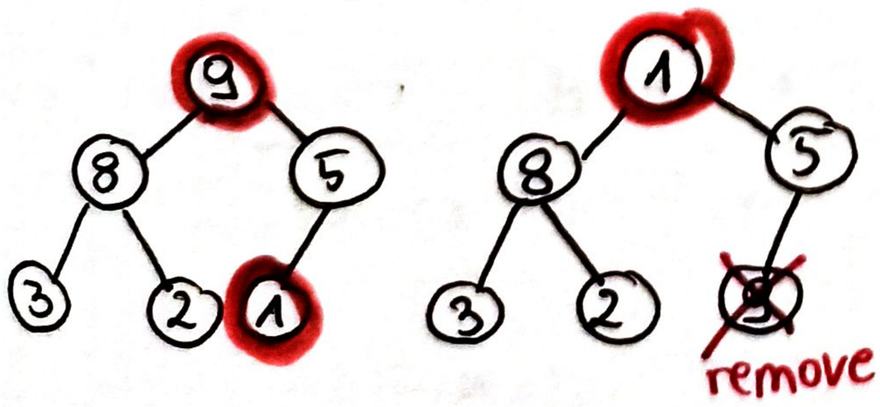
2	8	5	3	9	1



// build-max-heap

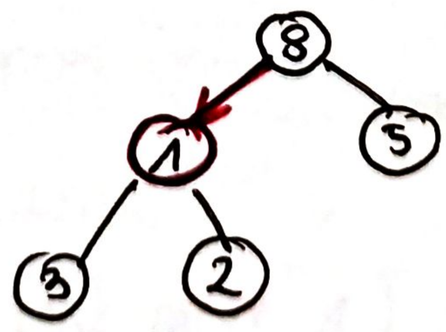
9	8	5	3	2	1

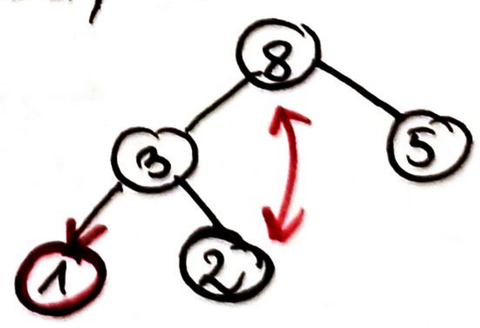
swap



1	8	5	3	2	9

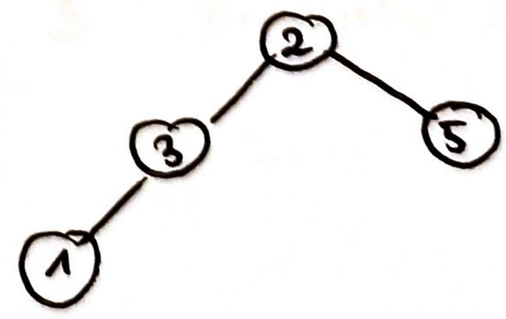
// heapify → part of heap is already sorted

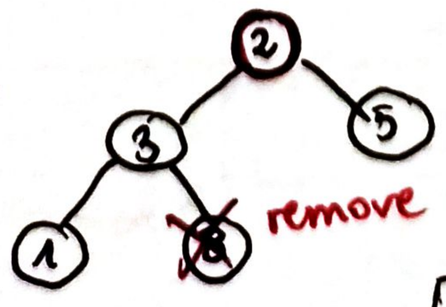




1	8	5	3	2	9

swap

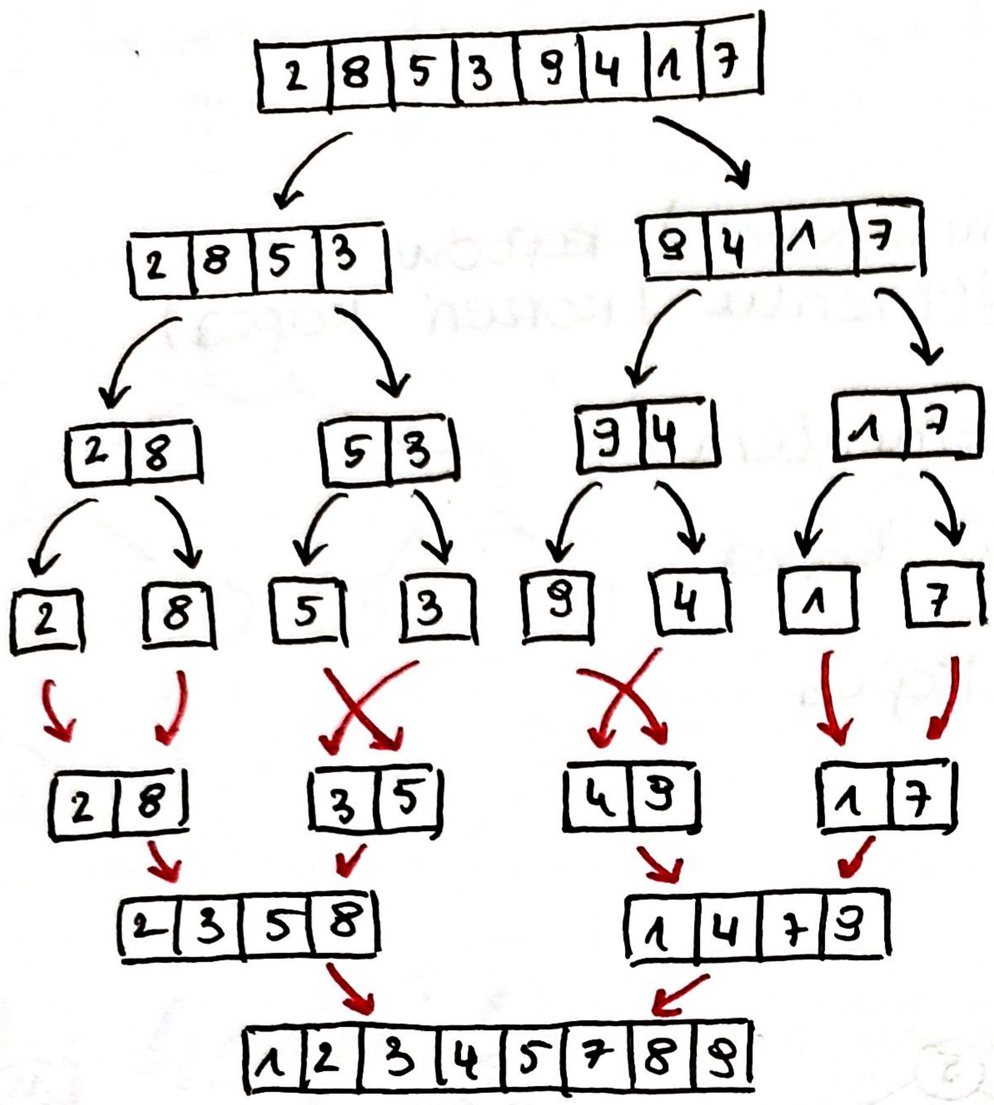




1	2	5	3	8	9

Time complexity: $O(n \cdot \log n)$

4.4 - merge sort



Step 1 Podziel tabelę / tabelki aż zostanie podzielona na indywidualne elementy

Step 2 Porównaj wartości parami (lub trójkami) i scal je w tymczasowe tabele

tymczasowe tablice są posortowane

Time complexity $O(n \cdot \log n)$

4.5 - kolejka priorytetowa

wymagane operacje:

- dodaj element o priorytecie x
- pobierz element o najwyższym priorytecie
- przeczytaj element o najwyższym priorytecie (bez usówania)

5. TABLICE HASZOWANE

5.1 - basic idea

Examples:

- students are assigned a unique roll number to retrive information about them
- books in libraries have unique number to retrive info about them

To store key/value pair, you can use an array. However, in cases where the keys are large and cannot be used directly as an index, you should use hashing.

In hashing, large keys are conveyed into small keys by using hash functions.

5.2 - funkcje haszujące

↳ any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.

Good hashing mechanism:

- EASY TO COMPUTE - it should be easy to compute and must not become an algorithm itself
- UNIFORM DISTRIBUTION - it should provide a uniform distribution across the hash table and should not result in clustering
- LESS COLLISIONS - collision occur when pairs of elements are mopped to the same hash value. These should be avoided

haszowanie modularne

$$h(k) = k \cdot \text{mod } m$$

→ rozmiar tablicy

haszowanie przez mnożenie

$$h(k) = \boxed{m}\,(k\boxed{A} \text{ mod } 1)$$

potęga 2 lub 10

$A \in \langle 0,1 \rangle$

haszowanie przez randomizację

$$h(k) = \text{Rand}(k)$$

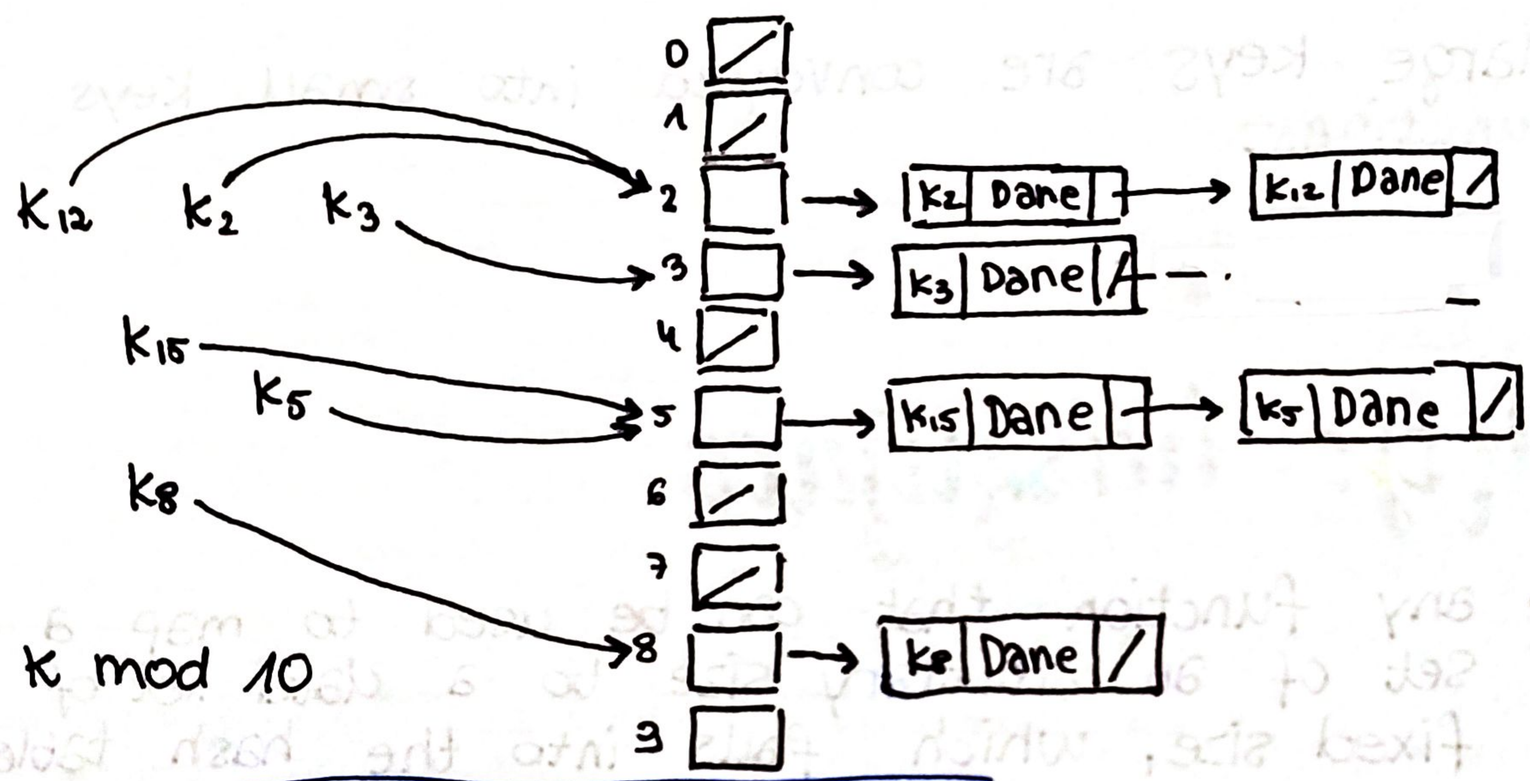
haszowanie uniwersalne

H - rodzina funkcji haszujących

Na początku losujemy jedną funkcję haszującą z całej rodziny i stosujemy do tej pory $h \in H$.

5.3 - kolizje

łańcuchowe oddzielanie



$h(k) = k \bmod 10$

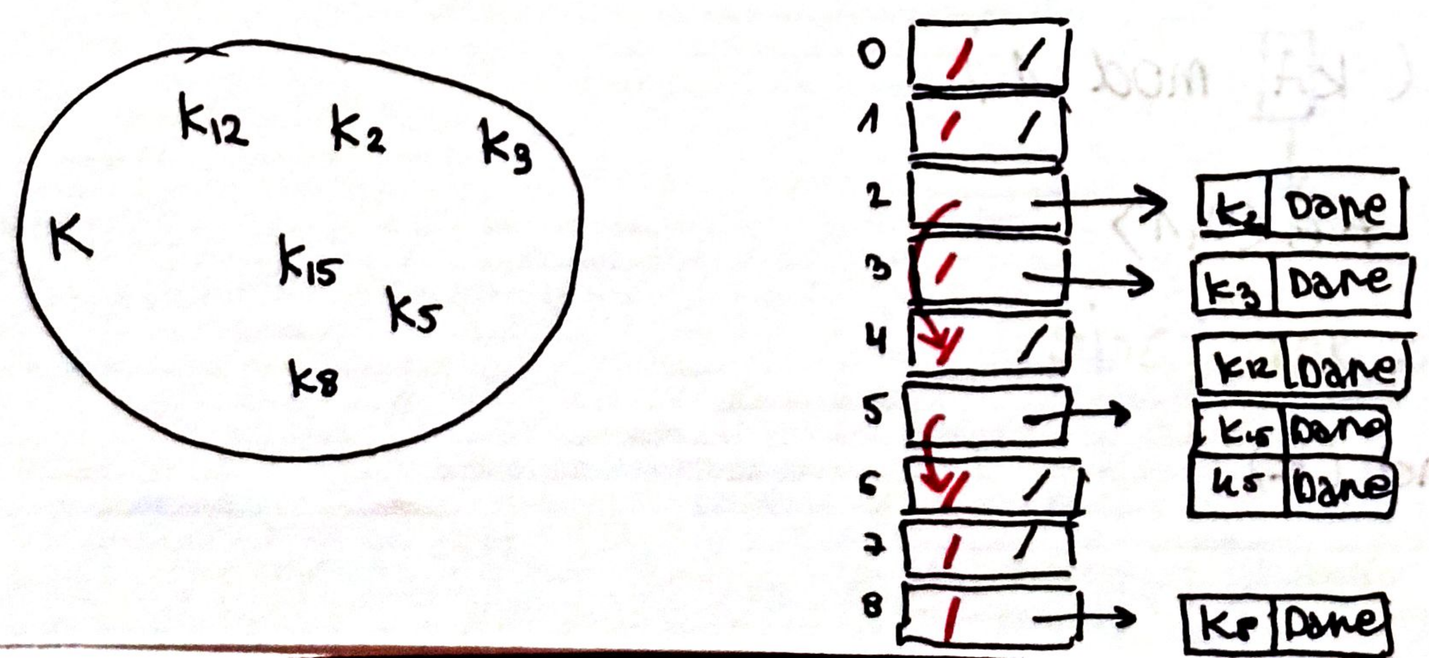
α - współczynnik zapełnienia

$\alpha = \frac{\text{ilość elementów}}{\text{rozmiar tablicy}}$

Time complexity
worst case $\Theta(n)$
expected case $\Theta(1+\alpha)$

łańcuchowanie bezpośrednie

- listy tworzone są w obszarze tablicy
- w przypadku dodawania nowego elementu do listy wybierana jest pierwsza wolna komórka



Adresowanie otwarte

- dla rozwiązywania konfliktów nie stosuje się list
- wszystkie elementy (wskaźniki) zapisywane są bezpośrednio w tablicy

$$h(k,i) = (h'(k) + g(i))$$

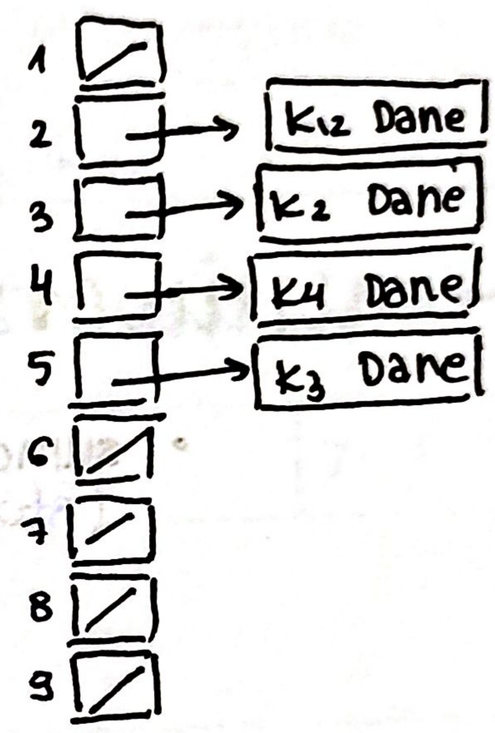
klucz ← k; i → rozmiar próby $(0 \ldots N-1)$ → rozmiar tablicy

- aby umieścić element w tablicy sprawdzamy $h(k,0)$ jeśli zajęte, to $h(k,1)$ aż do $h(k,N-1)$

Example

kolejność dodawania:
k_{12}, k_2, k_4, k_3

$h(k) = (k+i) \bmod 10$



Usuwanie elementów

Problem: Nie można wpisywać None!
Rozwiązanie:
- nowa stała np. DELETED
- należy wtedy zmodyfikować przeszukiwanie

```
if table[j] == None : return None
if table[j] == DELETED : continue
if table[j] == key : return table[j]
```

- i dodawanie

```
if table[j] == None or table[j] == DELETED :
```

LINEAR PROBING
EXPECTED SEARCH TIME

successful	$1 + \frac{1}{1-\alpha}$
usuccessful	$1 + \frac{1}{(1-\alpha)^2}$

DOUBLE HASH AND PSEUDORANDOM g FUNCTION

successful	$\alpha^{-1}\left(1 + \ln\left(\frac{1}{1-\alpha}\right)\right)$
unsuccessful	$\frac{1}{1-\alpha}$

Rodzaje adresowania otwartego

- liniowe : $h(k,i) = (h'(k) + i) \bmod m$
  → wadą jest grupowanie się elementów
- kwadratowe: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  → wadą jest grupowanie wtórne (mniej groźne)
- sześcienne: $h(k,i) = (h'(k) + c_1 i + c_2 i^2 + c_3 i^3) \bmod m$
  → wadą jest grupowanie wtórne
- dwukrotne: $h(k,i) = (h_1(k) + h_2(i)) \bmod m$
  → grupowanie wtórne jest znikome

5.4 - uniwersum kluczy

- sumowanie kodów ASCII liter
  (stosunkowo małe uniwersum)

  „ALA" = 65 + 76 + 65

- sumowanie par
  (są dziury w uniwersum bo znaki ≥ 32 i < 127)

  „ALA" = $65 \cdot 256 + 76 + 65 \cdot 256$

- suma iloczynów kodów par
  (jest nierówny rozkład wartości)

  „ALA" = $65 \cdot 76 + 65$

Przykład

Haszowanie liniowe modulo 9, tablica 9-elementowa.

Operacje:

+11, +8, +23, +7, +13, +32, -23, +41, +50, -41, +100.

+11: (11 mod 9 = 2)

		11						

+8 +23 +7 +13

		11		13	23		7	8

+32 (32 mod 9 = 5), adresowanie otwarte – wkładam w pierwsze wolne miejsce

		11		13	23	32	7	8

-23

		11		13	DELETED	32	7	8

+41

		11		13	41	32	7	8

+50 (50 mod 9 = 5), pierwsze wolne miejsce dopiero na początku tablicy

50		11		13	41	32	7	8

-41

50		11		13	DELETED	32	7	8

+100

50	100	11		13	DELETED	32	7	8

Na koniec uzupełniam puste pola NULL'ami

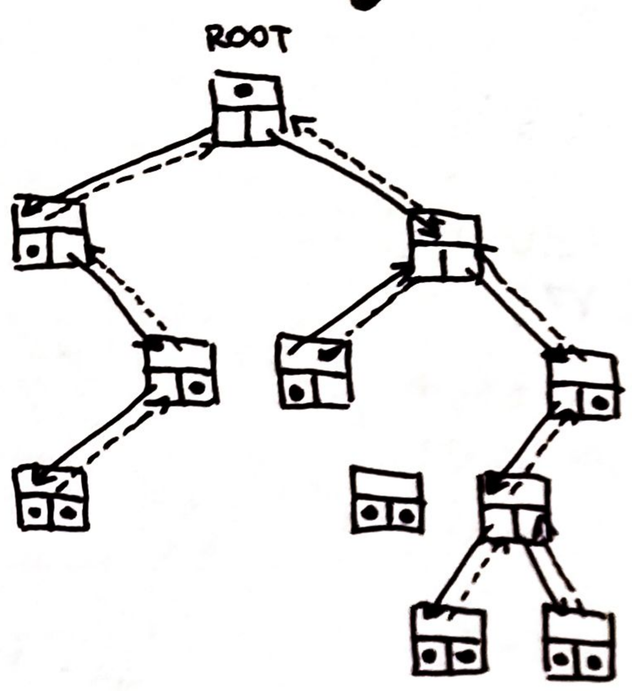
50	100	11	NULL	13	DELETED	32	7	8

6. TREES

rooted tree - tree with a defined root (starting node)

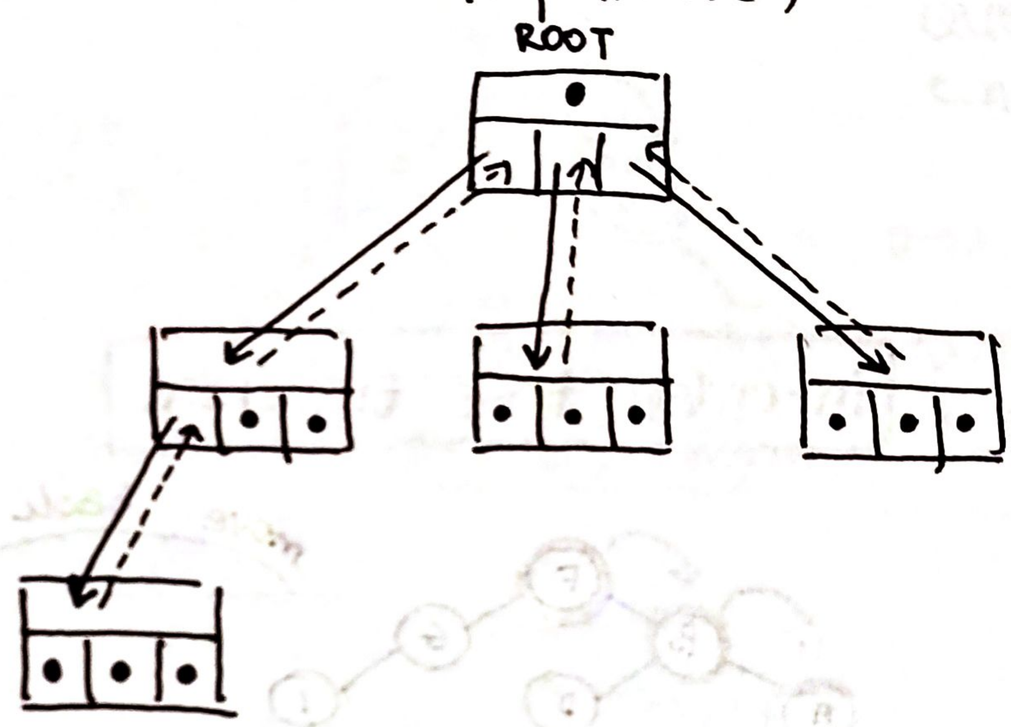
binary tree - node can have at most two children (two pointers)

6.1 - binary tree



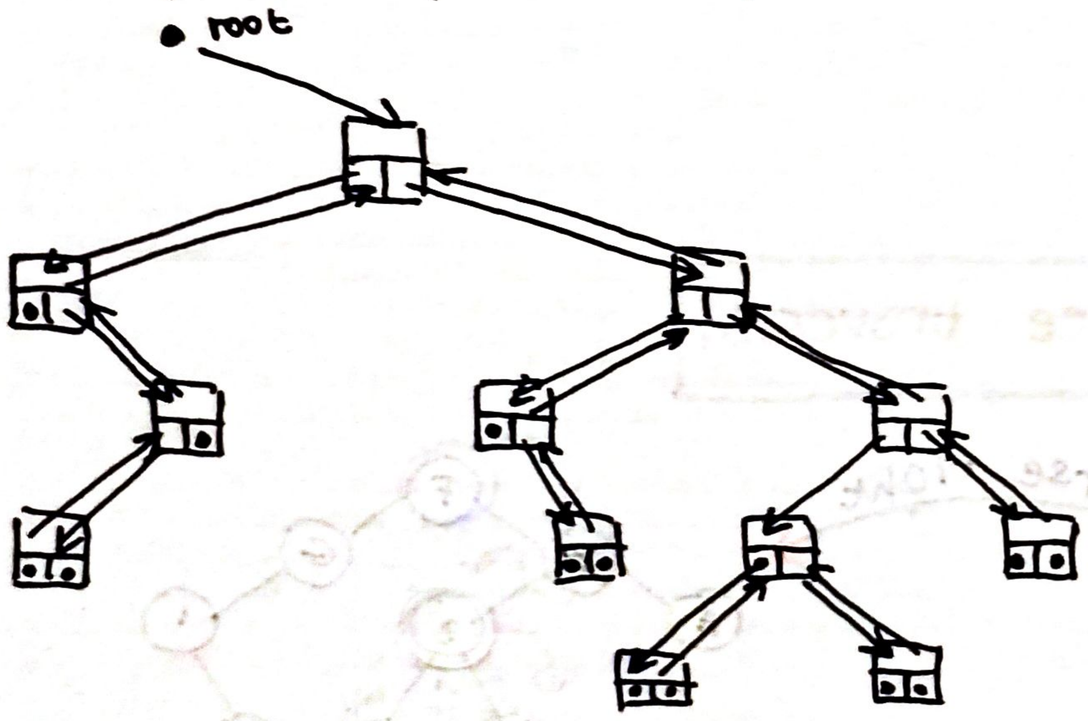
6.2 - K-ary tree

the node has at most K children (in implementation it has K pointers)



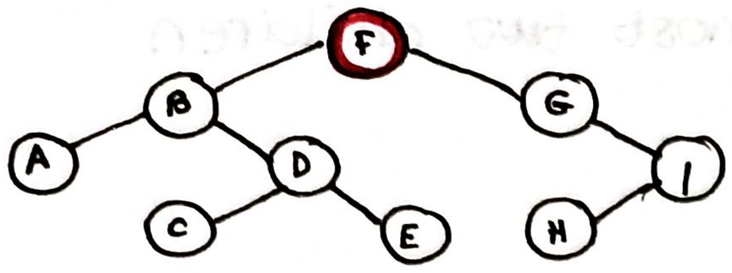
6.3 - drzewo generalne

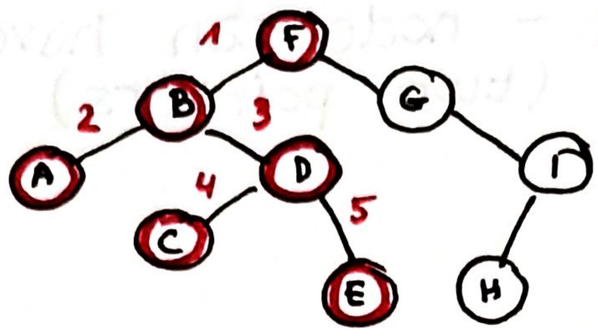
notacja „na prawo brat, na lewo syn"

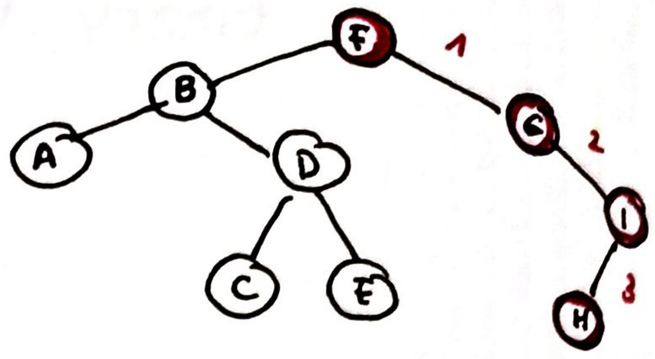


6.4 - depth first search

preorder tree traversal





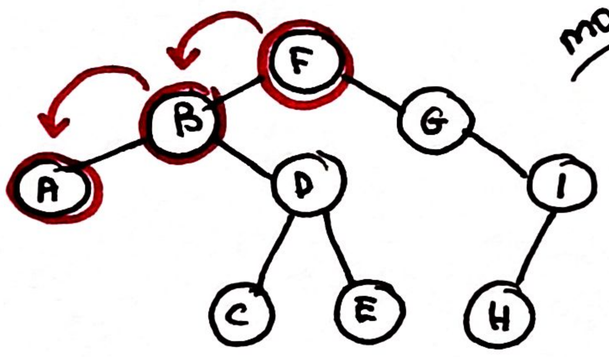


1) visit note
2) traverse left
3) traverse right

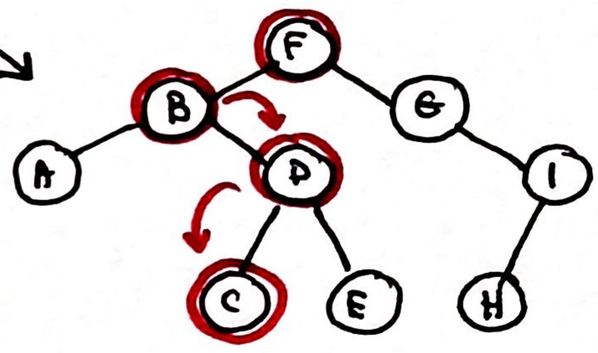
Time complexity $O(n)$

visit list : F, B, A, D, C, E, G, I, H

in-order tree traversal



move back

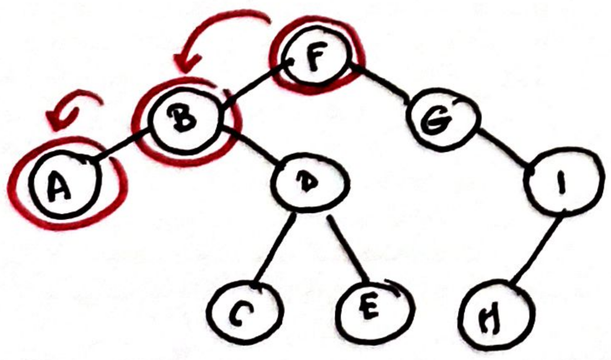


1) traverse left
2) visit node
3) traverse right

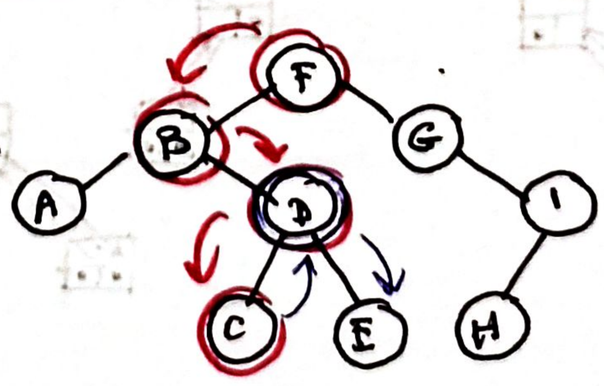
Time complexity $O(n)$

visit list : A, B, C, D, E, F, G, H, I

post-order tree traversal



traverse right



1) traverse left
2) traverse right
3) visit node

A, C, E, D, B, H, I, G, F

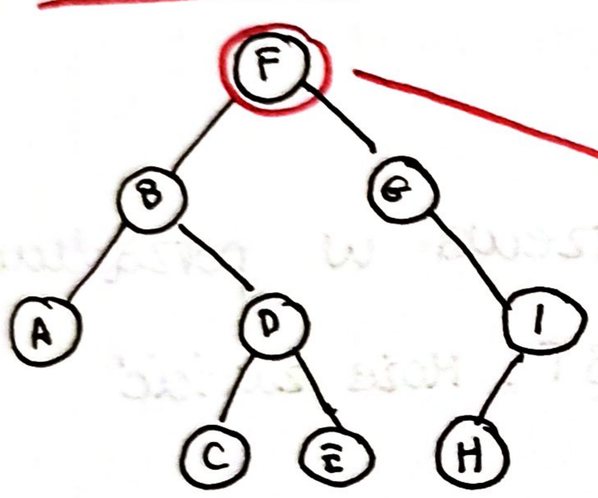
6.5 - breadth-first search

funkcja przegląda drzewo wszerz

level-order tree traversal

1) if queue empty add root
2) remove node from queue then add it's left and right child
3) if queue is empty again → end algorithm

Example

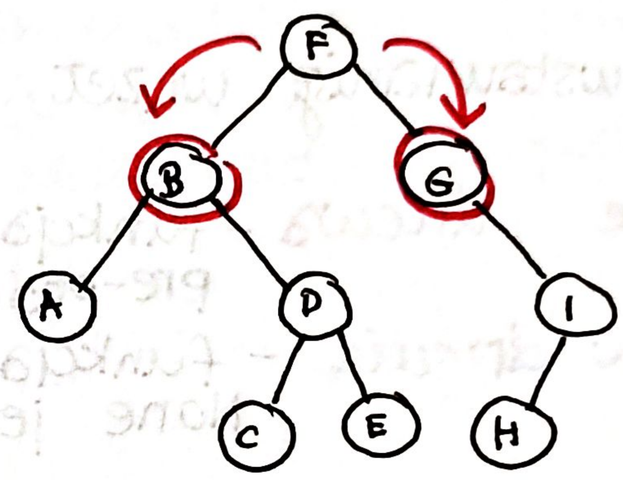


QUEUE: empty

add root

QUEUE: F

add left child →



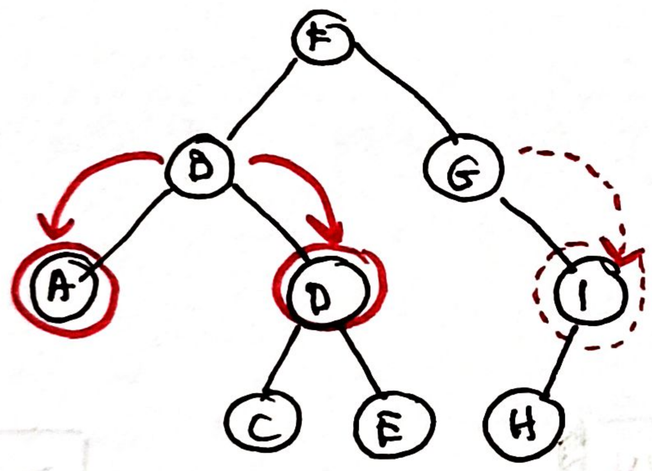
QUEUE: empty

add children

QUEUE: B, G

QUEUE: G

remove B from queue and print it (add B's children) →



add B's children

QUEUE G, A, D

Output: F, B, G, A, D, I, C, E, H

Time complexity

$O(n)$

number of nodes

6.6 - binarne drzewo poszukiwań

- uporządkowane, posortowane, drzewa binarne
- węzeł może mieć dwa podrzewa
- węzły po lewej są mniejsze od rodzica
- węzły po prawej są większe od rodzica
- wszystkie kluczowe operacje na drzewach mają złożoność $O(\underbrace{h(T)}_{\text{wysokość drzewa}})$

Wstawianie węzła

- wstawiany węzeł jest zawsze liściem

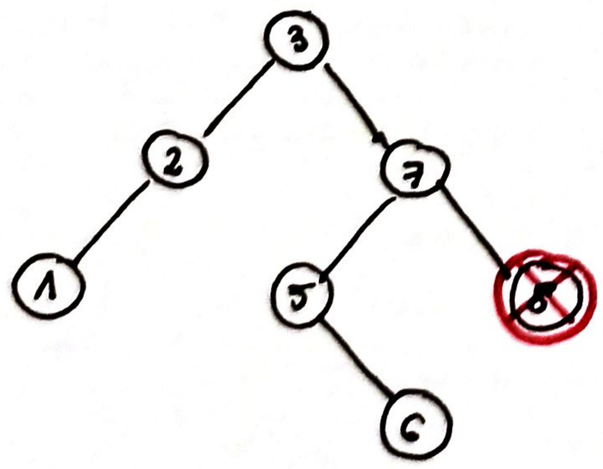
drukowanie drzewa - funkcja wypisuje zawartość drzewa w porządku pre-order lub in-order

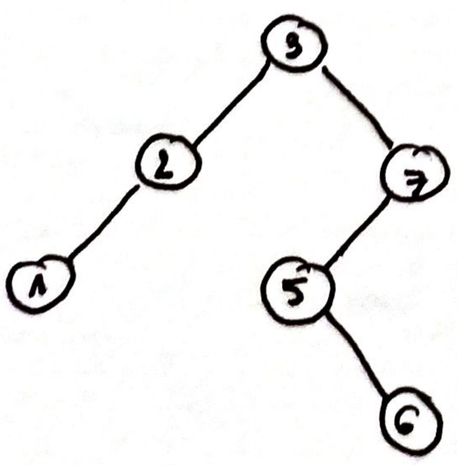
szukanie w drzewie - funkcja szuka klucza w BST. Może zwrócić None jeśli nie ma klucza k

Usuwanie węzła

BRAK SYNÓW

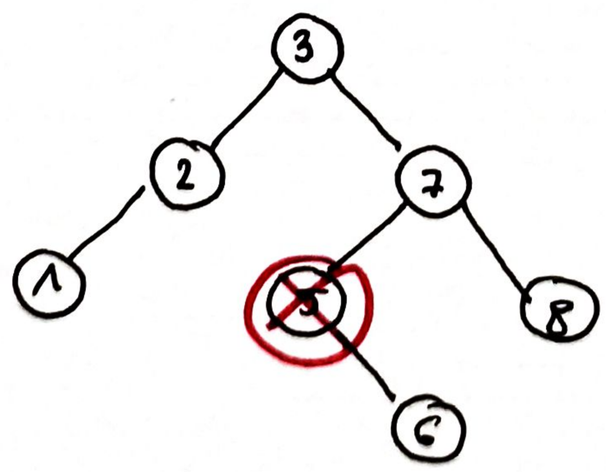
usuwany węzeł jest liściem - zastępujemy go przez None

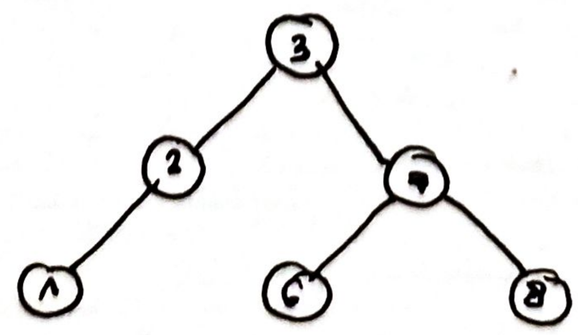




JEDEN SYN

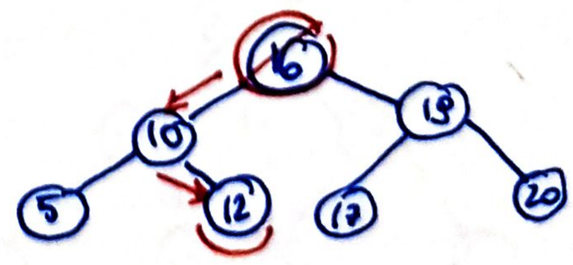
usuwamy go „kompresując gałąź" w drzewie

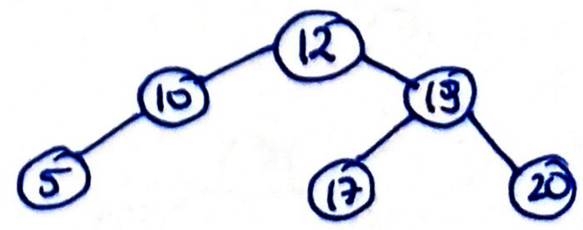




DWÓCH SYNÓW

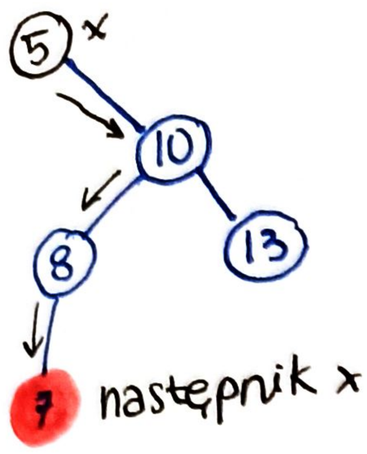
Find the largest element in left subtree and swap places with deleted node → delete the node





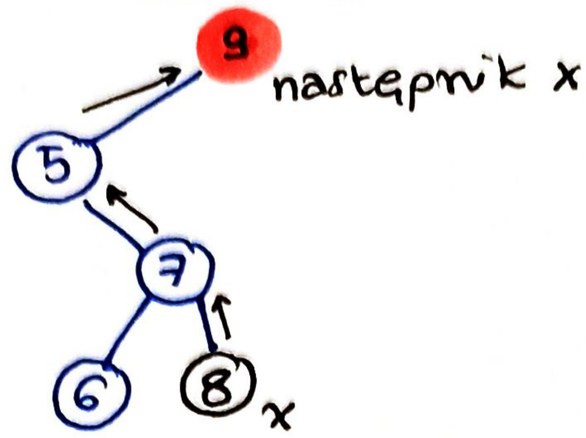
następnik – successor

Case 1



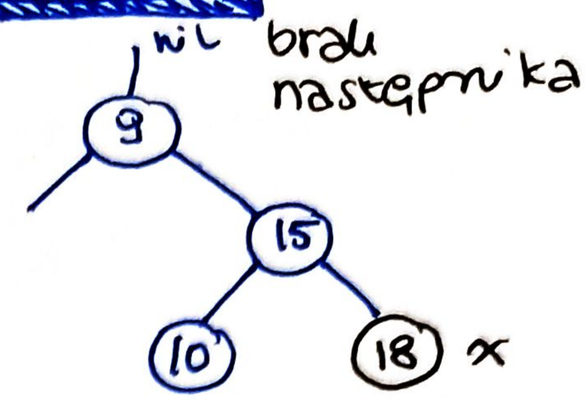
Węzeł x posiada prawego syna – następnikiem jest węzeł o minimalnym kluczu w poddrzewie, którego korzeniem jest prawy syn.

Case 2



Węzeł x NIE posiada prawego syna. Wtedy idąc w górę drzewa musimy znaleźć pierwszego ojca, dla którego nasz węzeł leży w prawym poddrzewie.

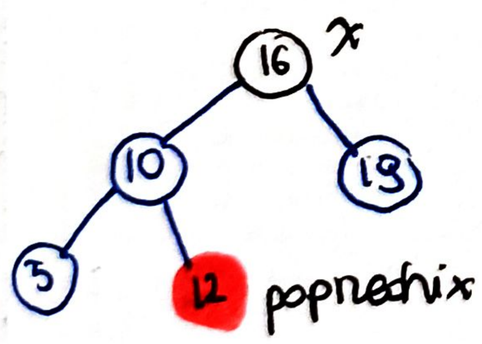
Case 3



Węzeł x nie posiada prawego syna i nie jest to case 2. Nie znajdujemy się w żadnym lewym poddrzewie → DUPA

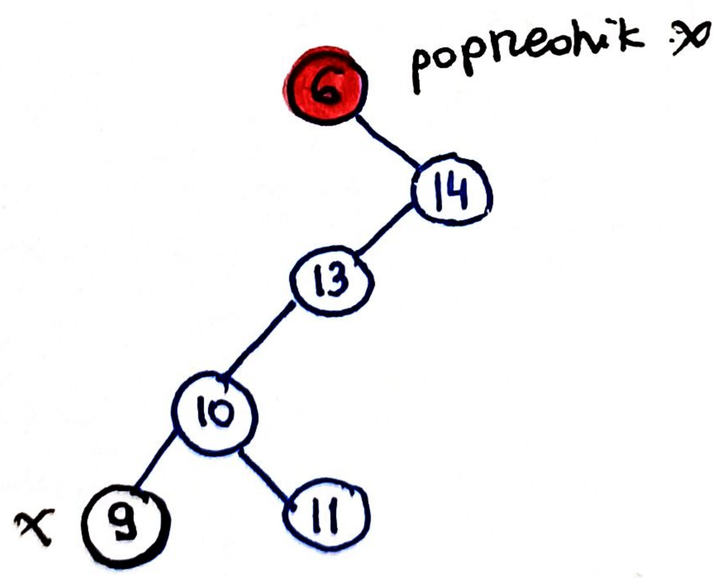
poprzednik – predecessor

Case 1



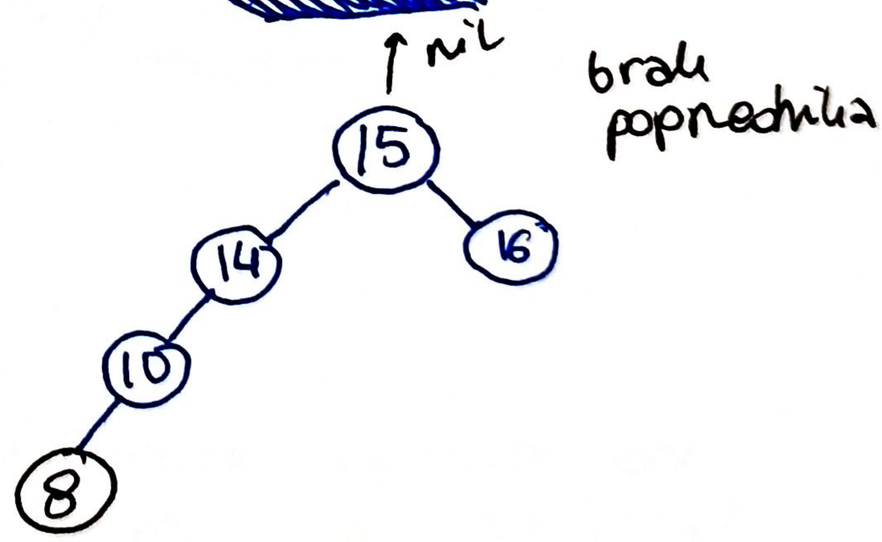
Zwracamy węzeł maksymalny od lewego syna

Case 2



Przemieszczamy się w górę drzewa, aż trafimy na węzeł dla którego x leży w prawej gałęzi

Case 3



Dupa → bo nie leży dla żadnego węzła w prawej gałęzi

6.7 - drzewa zrównoważone

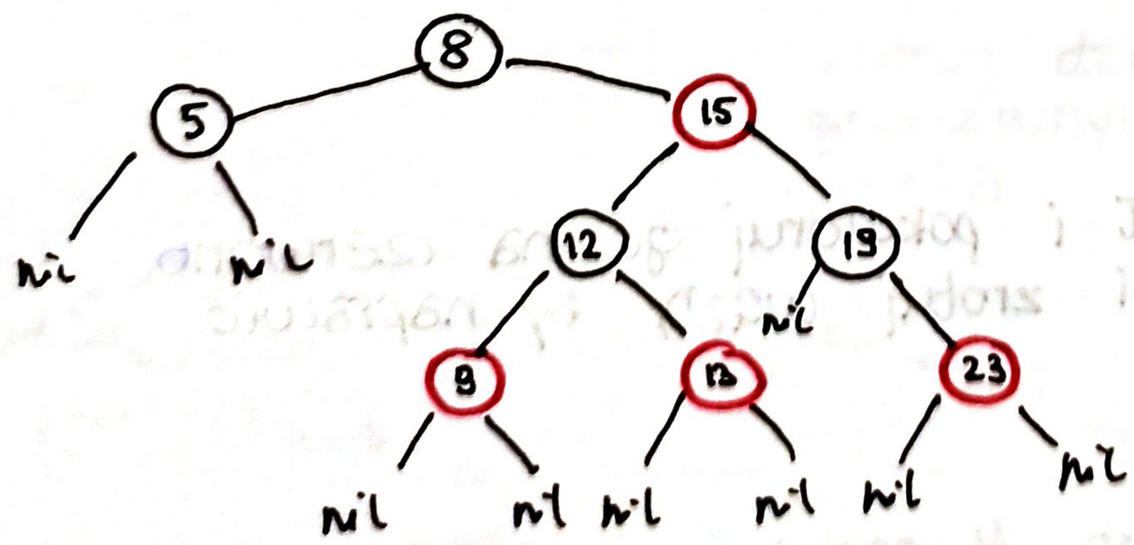
- czas operacji na BST (binary search tree) jest proporcjonalny do wysokości drzewa
- drzewo doskonale zrównoważone - dla dowolnego wierzchołka rozmiar lewego i prawego poddrzewa różnią się najwyżej o 1
- drzewo zrównoważone - długość dowolnej ścieżki z węzła do liści różni się od wysokości tego węzła najwyżej o 1
- drzewo w przybliżeniu zrównoważone - długość dowolnej ścieżki z węzła do liści różni się od wysokości tego węzła najwyżej 2 razy

6.8 - drzewa czerwono-czarne

- węzeł jest czerwony lub czarny
- korzeń i liście (NIL) są czarne
- jeżeli węzeł jest czerwony, wtedy dzieci są czarne
- wszystkie ścieżki prowadzące z jakiegoś węzła do NIL'ów zawierają tyle samo czarnych węzłów

  ↳ licząc czarne węzły w ścieżkach nie liczymy węzła początkowego

Example



1) nodes require one storage bit to keep track of colour

2) najdłuższa ścieżka jest nie dłuższa niż 2x najkrótsza ścieżka

Time complexity $O(\log n)$

Space complexity $O(n)$

Operacje:
- search
- insert
- remove

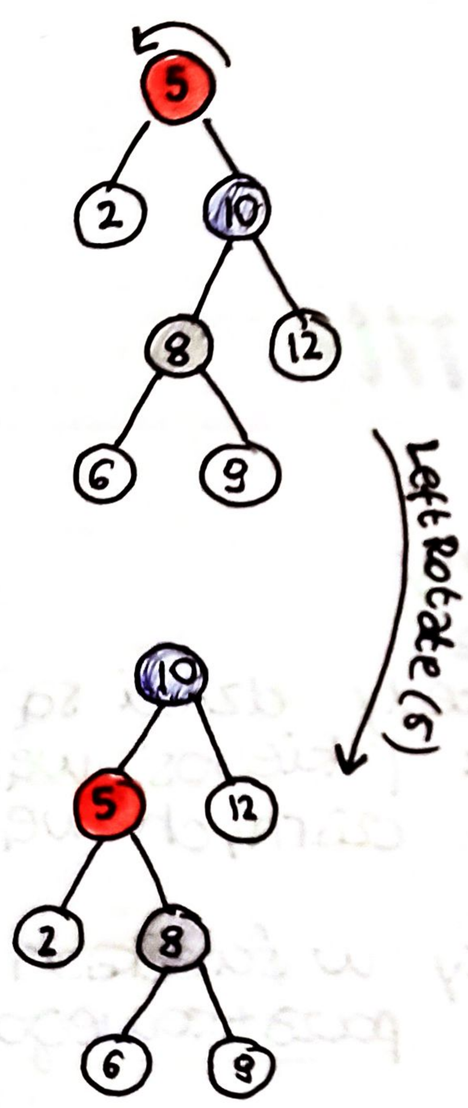
} require rotation

Definicja węzła

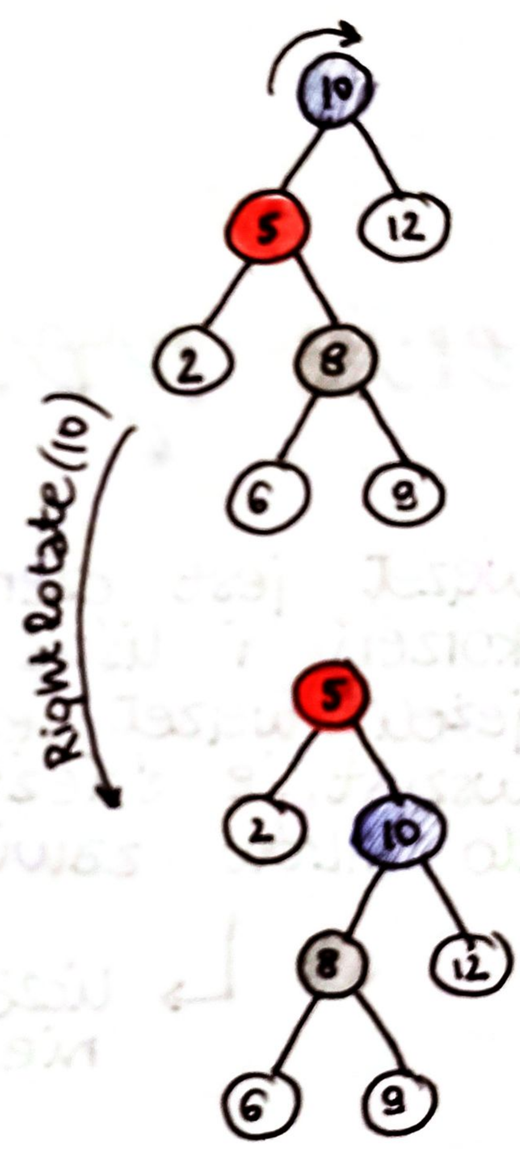
```
class NODE :
    data = None
    color = BLACK
    left = None
    right = None
    parent = none
```

Rotacje

Left-rotate



Right-rotate



Wstawianie węzła

1) wstaw węzeł i pokoloruj go na czerwono
2) przekoloruj i zrotuj węzły by naprawić naruszenia

↓

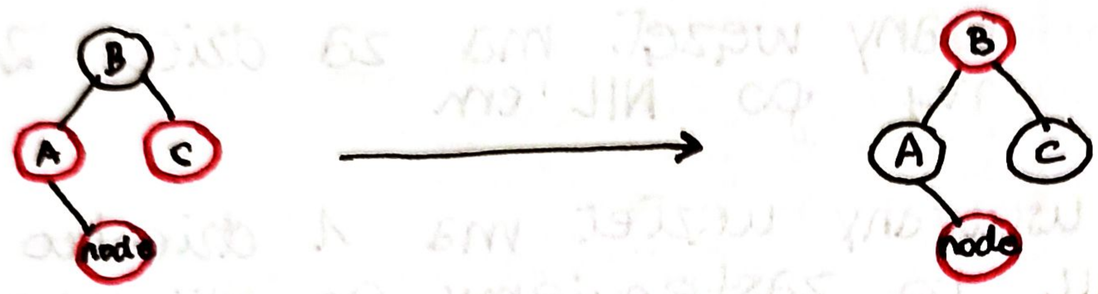
możliwe są 4 scenariusze

node = root

rozwiązanie : pokoloruj dodany węzeł na czarno

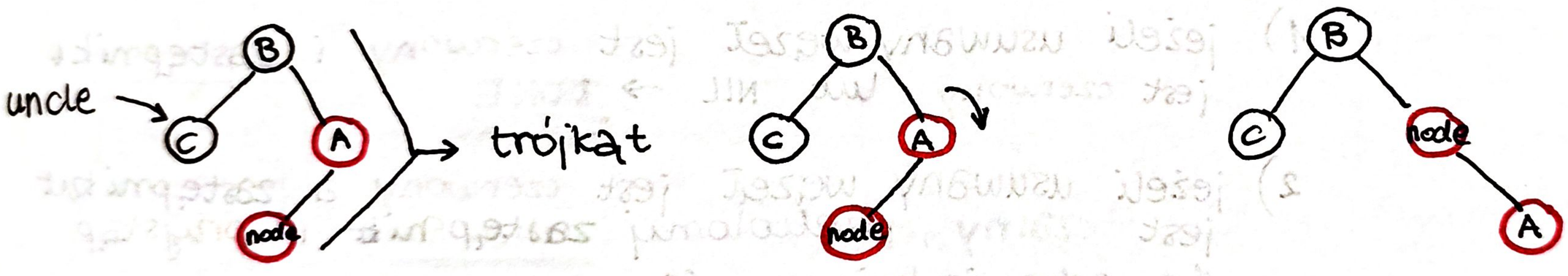
node's uncle is red

rozwiązanie : przekoloruj rodzica dodanego węzła, dziadka oraz wujka



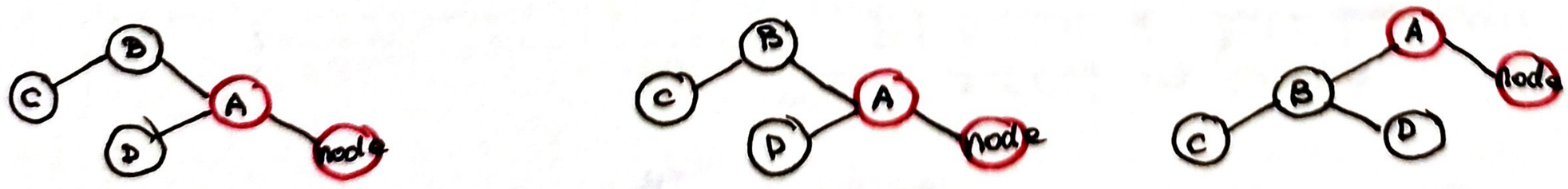
node's uncle is black (trójkąt)

rozwiązanie : zrotuj rodzica dodanego węzła w przeciwnym kierunku od dodanego węzła

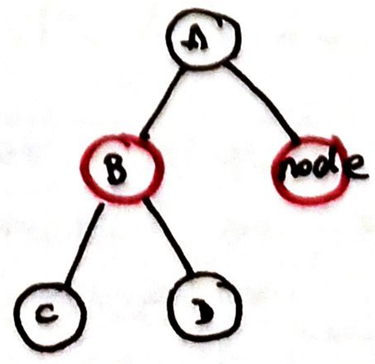


node's uncle is black (linia)

rozwiązanie : zrotuj dziadka dodanego węzła w przeciwnym kierunku od dodanego węzła



potem przekoloruj



Time complexity :

insert : O (log n)
colour : O (1)
violation fix : recolour O(1)
rotation O(1)

TOTAL: O (log n)

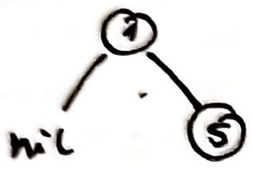
Usuwanie węzła

↳ po usunięciu węzła wszystkie własności drzewa czerwono-czarnego muszą zostać zachowane

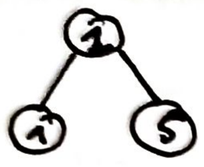
NIL node - every leaf is a special node called NIL (with no key) NIL is black
(wartownik utdy

Part 1

1) jeżeli usuwany węzeł ma za dzieci 2 liście NIL zastępujemy go NIL'em



2) jeżeli usuwany węzeł ma 1 dziecko NIL i jedno nie-NIL to zastępujemy go NIE-NILowym dzieckiem



3) jeżeli usuwany węzeł ma dwójkę nie-NILowych dzieci, set x to the replacement's right child before replacement is spliced out

Part 2

1) jeżeli usuwany węzeł jest czerwony i replacement jest czerwony lub NIL → **DONE**

2) jeżeli usuwany węzeł jest czerwony a replacement jest **czarny**, przekolorujemy replacement i przystąp do odpowiedniego Case

3) jeżeli usuwany węzeł jest **czarny** a jego replacement jest czerwony, przekolorujemy replacement na **czarno** → **DONE**

4) jeżeli usuwany węzeł jest **czarny** a jego replacement to NIL lub jest czarny przystąp do odpowiedniego Case.

Cases

0) węzeł x jest czerwony
1) węzeł x jest **czarny** i jego rodzeństwo w jest czerwone
2) węzeł x jest **czarny** a jego rodzeństwo w jest **czarne** i synowie w są **czarni**

3) węzeł x jest **czarny** i jego rodzeństwo w jest **czarne** i
   - x jest lewym dzieckiem i lewe dziecko w są czerwone i prawe dziecko w jest **czarne**

   lub
   - x jest prawym dzieckiem i prawe dziecko w są czerwone i lewe dziecko w jest **czarne**

4) węzeł x jest **czarny** i jego rodzeństwo w jest **czarne** i
   - jeżeli x jest lewym dzieckiem a prawe dziecko w jest czerwone

   lub
   - jeżeli x jest prawym dzieckiem i lewe dziecko w jest czerwone

Case 0

→ węzeł x jest czerwony

1) pokoloruj x na **czarno** → **DONE**

Case 1

→ węzeł x jest **czarny** i jego rodzeństwo w jest czerwony

1) przekoloruj w na **czarno**
2) przekoloruj rodzica x na czerwono
3) zrotuj rodzica x
   a) jeżeli x jest lewym dzieckiem zrotuj w LEWO
   b) jeżeli x jest prawym dzieckiem zrotuj w prawo
4)
   a) jeżeli x jest lewym dzieckiem ustaw w jako prawe dziecko rodzica x
   b) jeżeli x jest prawym dzieckiem ustaw w jako lewe dziecko rodzica x
5) z x i nowym w go to CASE 2, 3 lub 4

Case 2 → węzeł x jest **czarny** a jego rodzeństwo w jest **czarne** i dzieci w są **czarne**

1) przekoloruj w na czerwono
2) ustaw x jako rodzica x

2.1) jeżeli nowy x jest czerwony, przekoloruj x na **czarno**

2.2) jeżeli nowy x jest **czarny**, zdecyduj dalej który case pasuje CASE 1, 2, 3 lub 4 (pamiętaj, że jest teraz nowe w!)

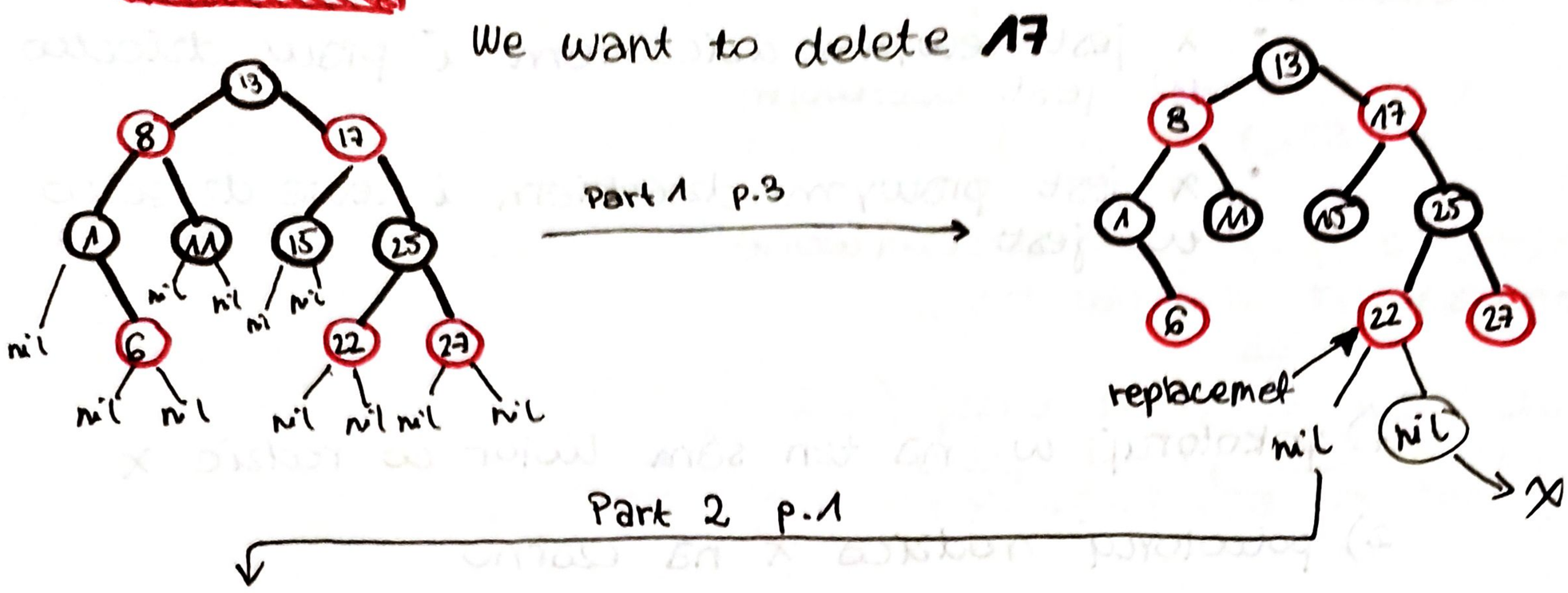
Case 3 → węzeł x jest **czarny** a jego rodzeństwo w jest **czarne** i

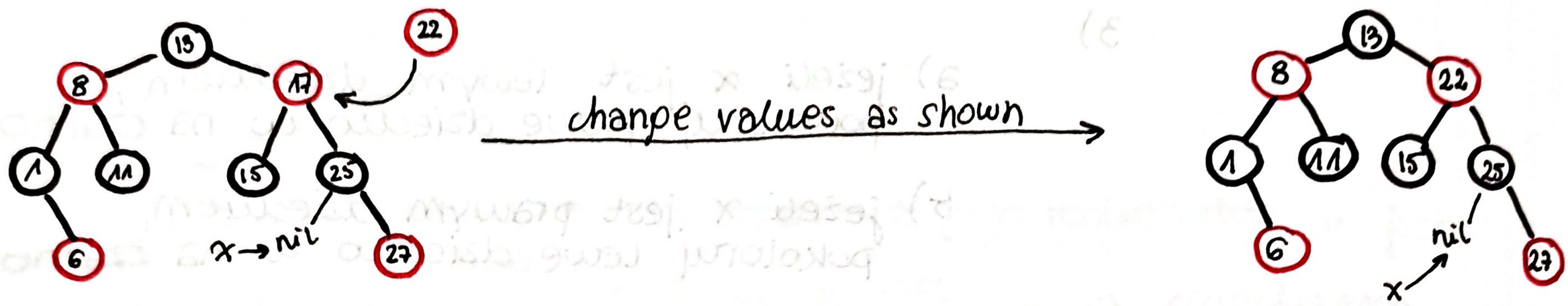
- jeżeli x jest lewym dzieckiem, lewe dziecko w jest czerwone a prawe dziecko w jest **czarne**
- jeżeli x jest prawym dzieckiem, lewe dziecko w jest **czarne** a prawe dziecko w jest czerwone

1) zmiana kolorów dzieci w
   a) jeżeli x jest lewym dzieckiem, pokoloruj lewe dziecko w na **czarno**
   b) jeżeli x jest prawym dzieckiem, pokoloruj prawe dziecko w na **czarno**
2) przekoloruj w na czerwono
3) rotacja w
   a) jeżeli x jest lewym dzieckiem zrotuj w w prawo
   b) jeżeli x jest prawym dzieckiem zrotuj w w lewo
4) zmień w
   a) jeżeli x jest lewym dzieckiem ustaw w na prawe dziecko rodzica x
   b) jeżeli x jest prawym dzieckiem ustaw w na lewe dziecko rodzica x
5) przejdź do CASE 4

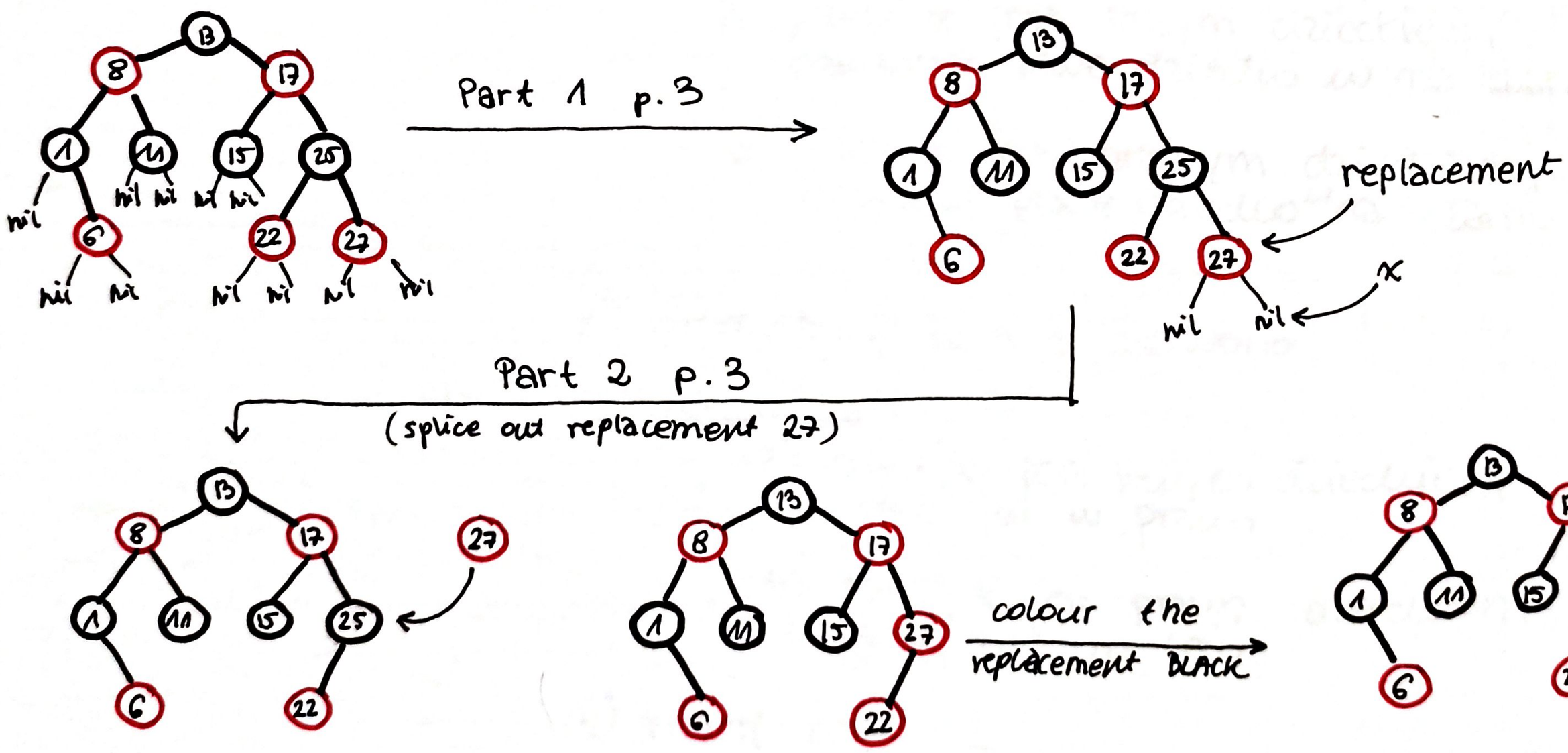
Examples

We want to delete **17**

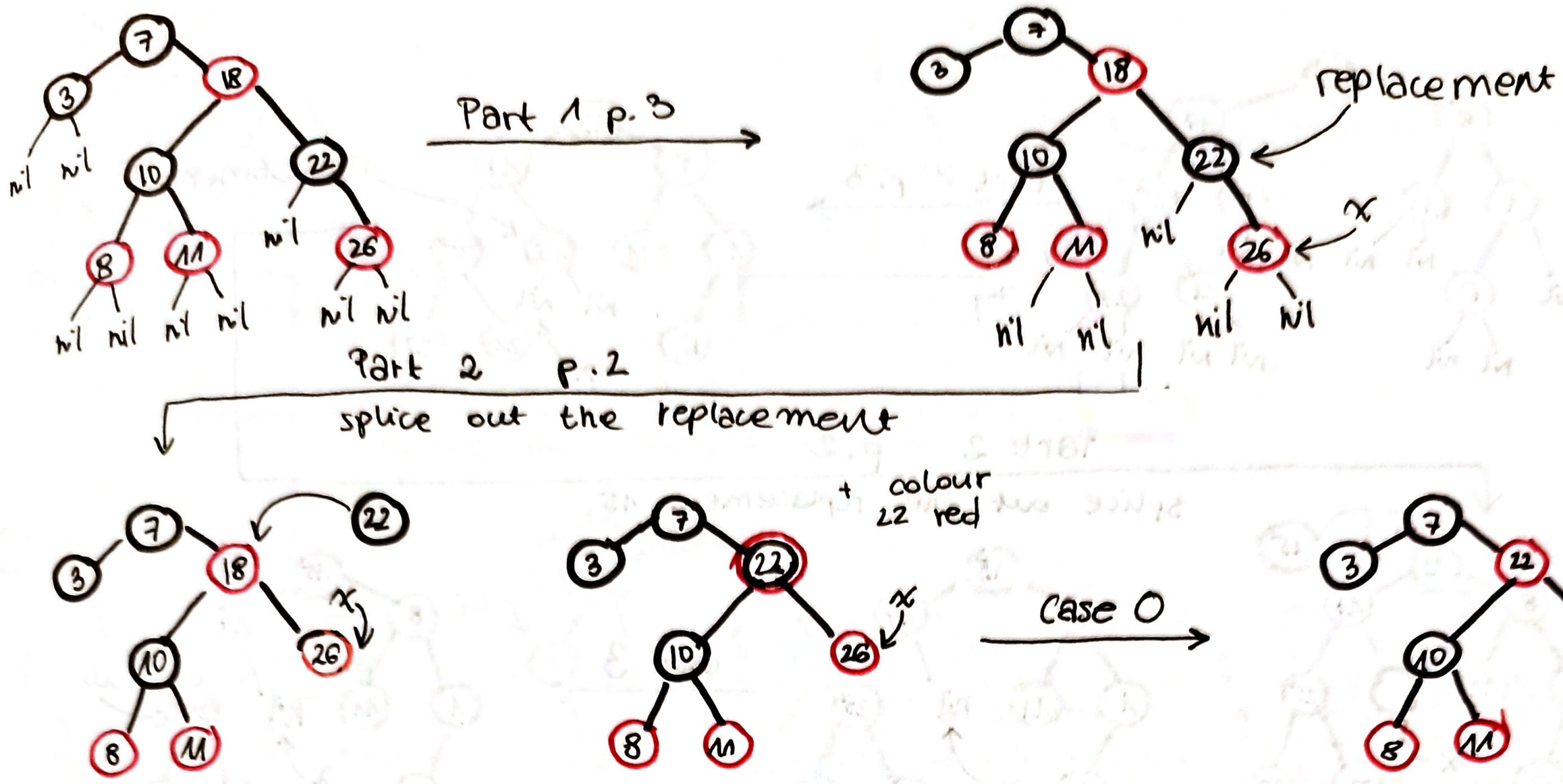




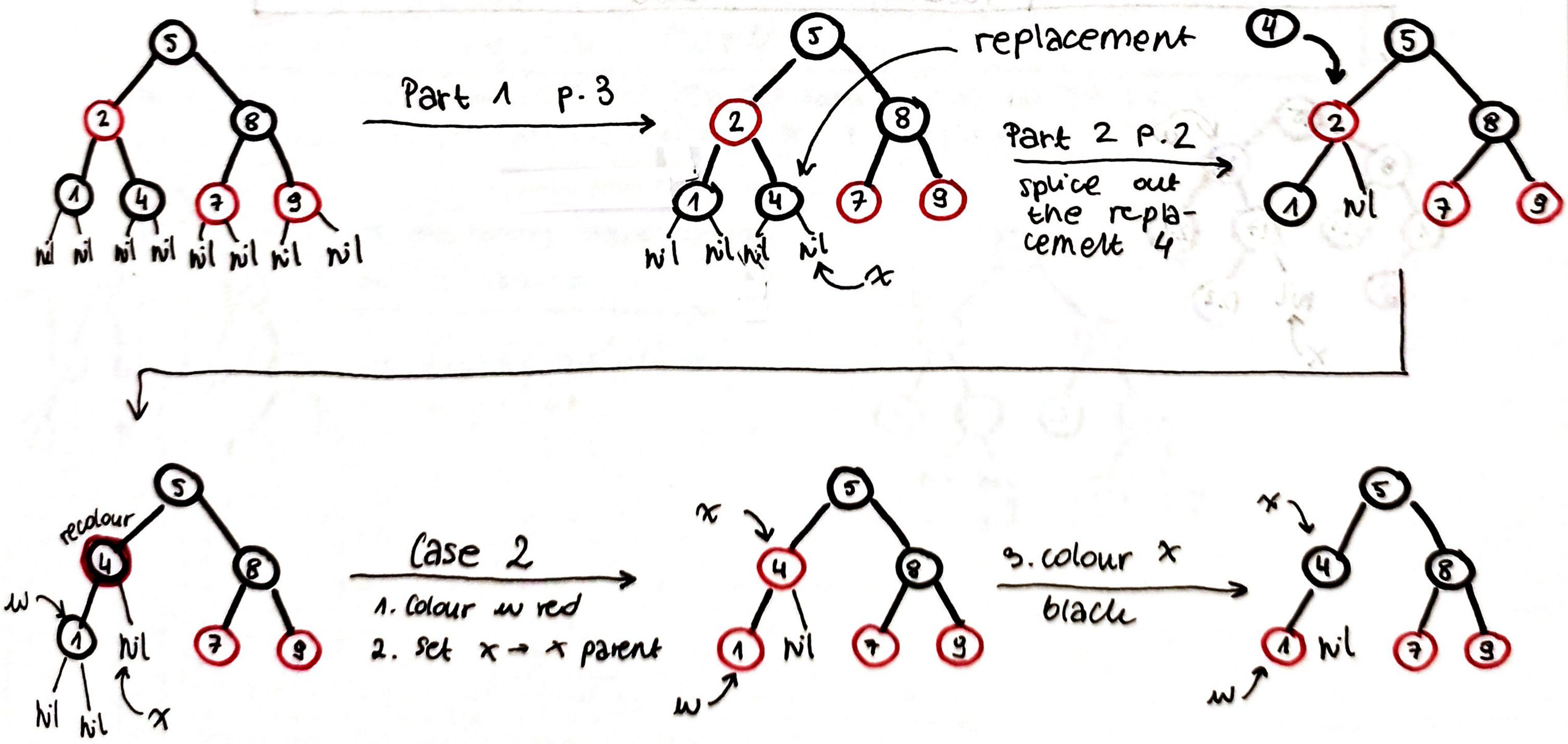
We want to delete **25**



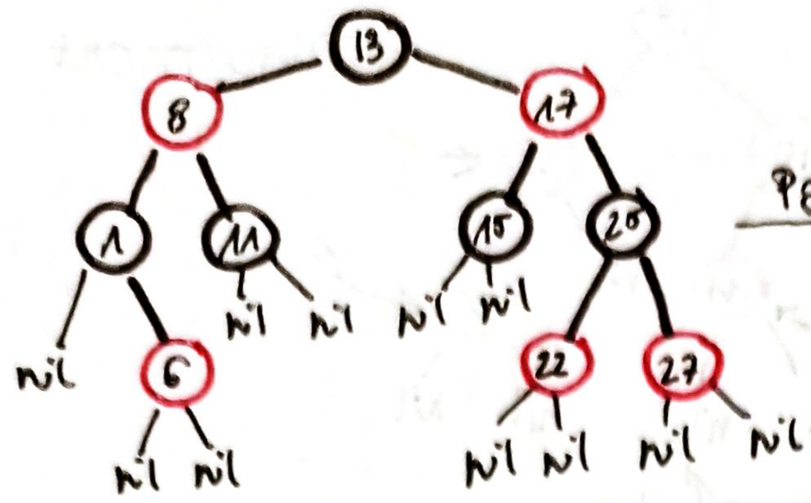
We want to delete **18**



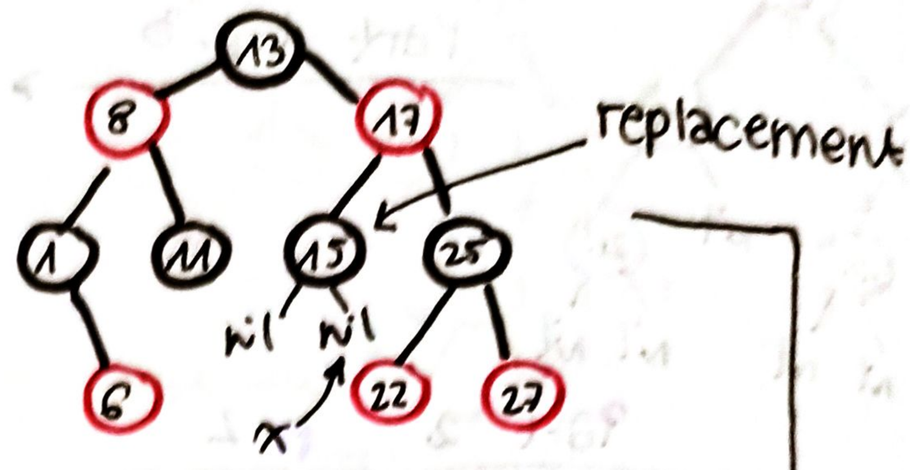
We want to delete **2**



We want to delete 13

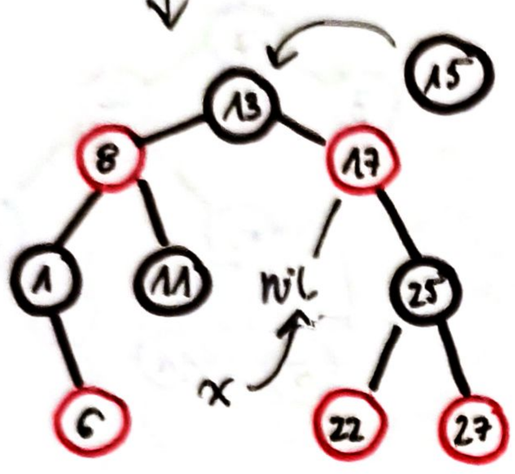


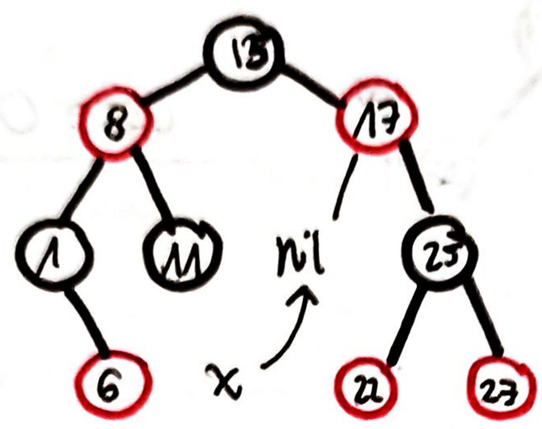
Part 1 p. 3 →



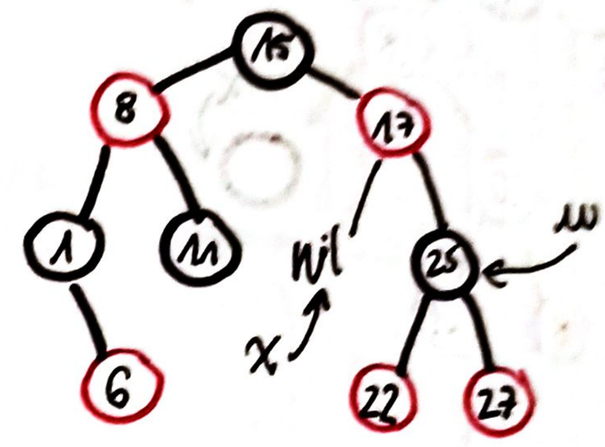
Part 2 p. 2

splice out the replacement 15

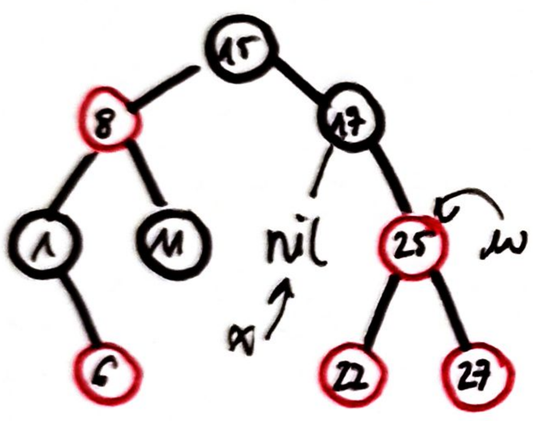




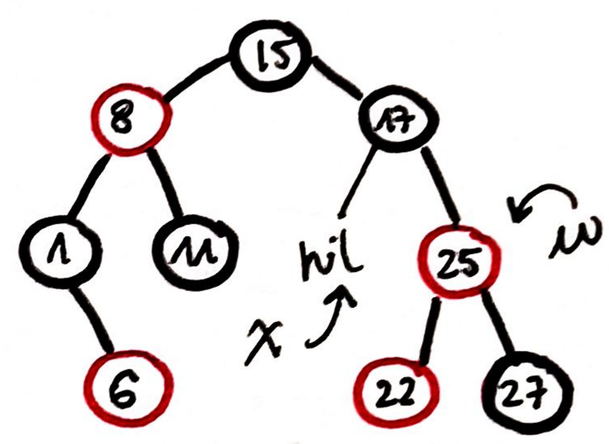
Case 4 →



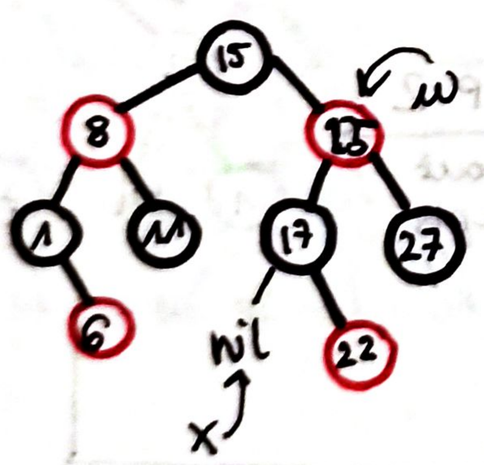
1. colour w na kolor rodzica x
2. colour rodzica x na czarno



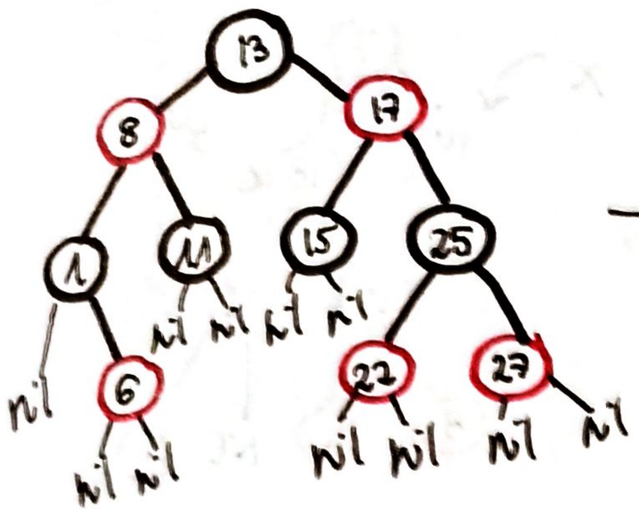
3.a pokoloruj prawe dziecko w na czerno →



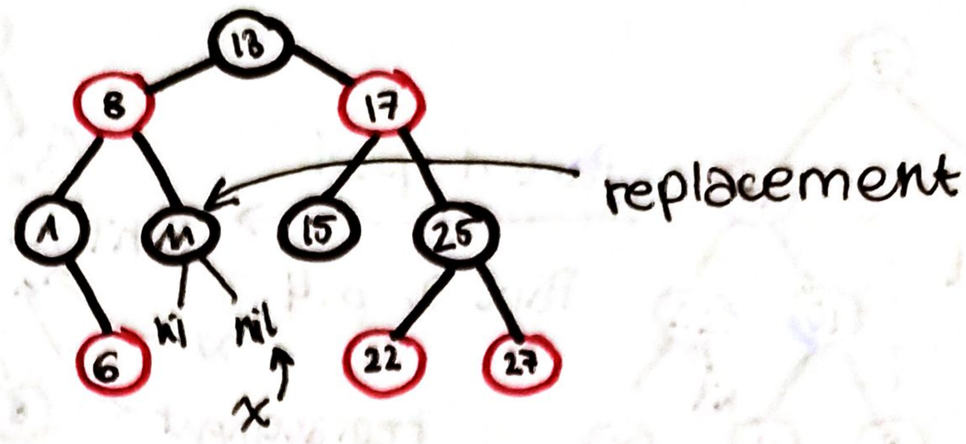
zrotuj rodzica x w lewo



We want to delete **8**

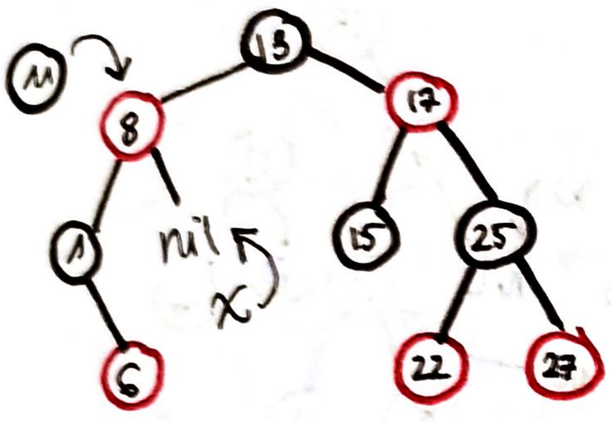


Part 1 p.3 →

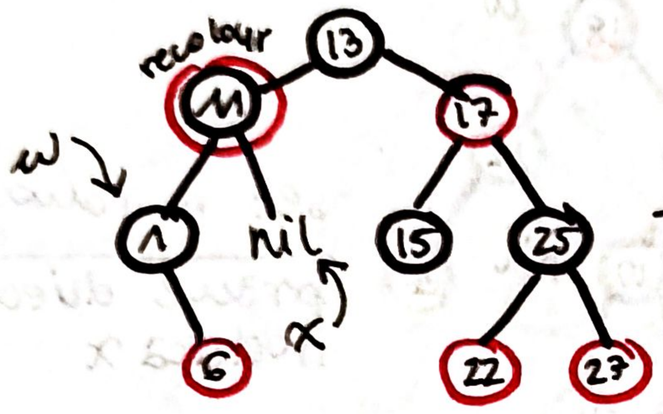


Part 2 p.2

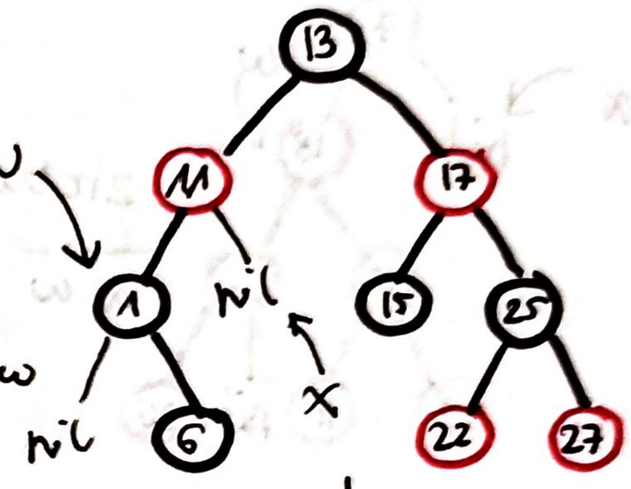
splice out the replacement 11



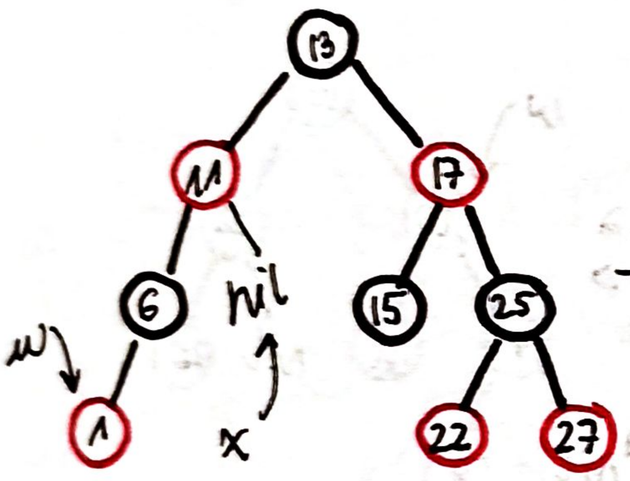
recolour



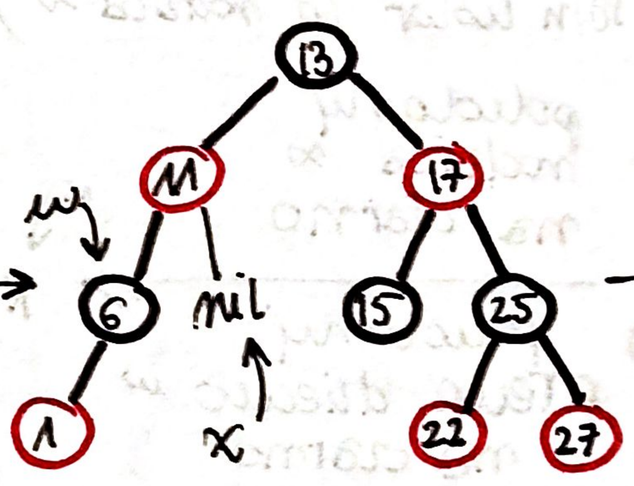
Case 3 :
1.2
pokoloruj prawe dziecko w na czarno →



przekoloruj w na czerwono, zrotuj w w lewo

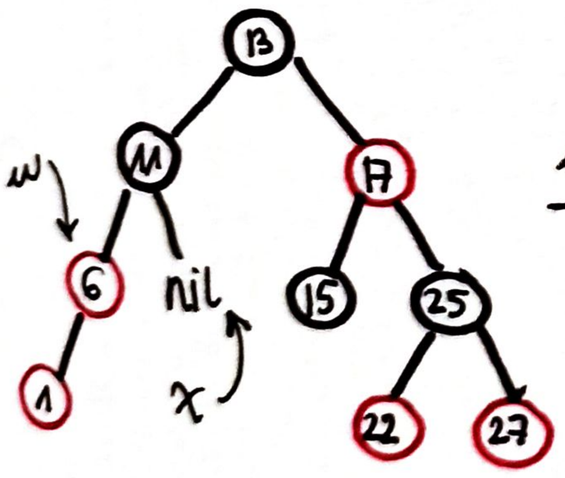


w = lewe dziecko rodzica x →

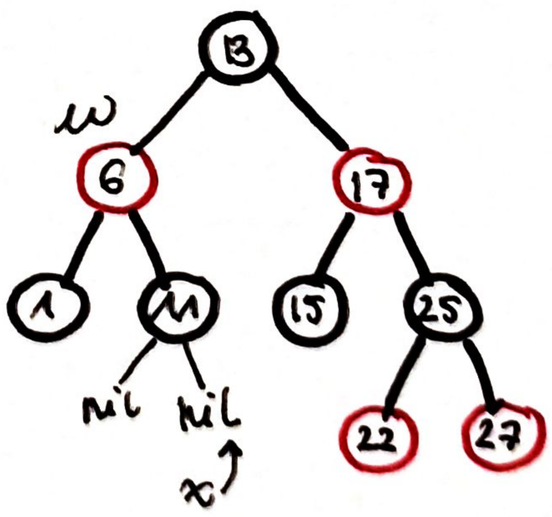


go to case 4

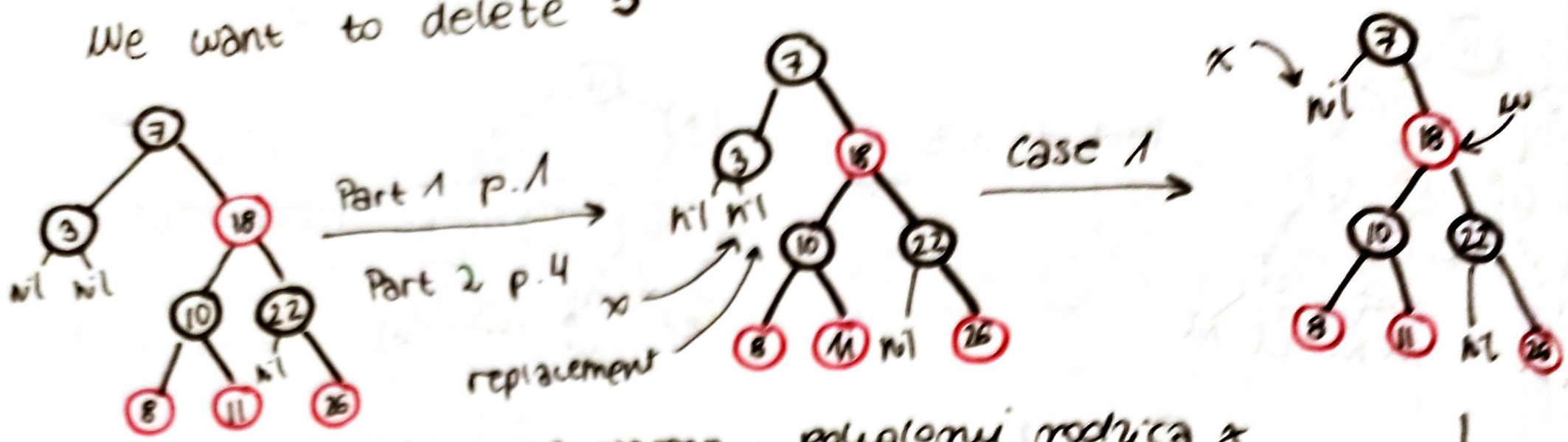
1. pokoloruj w na ten sam kolor co rodzic x
2. pokoloruj rodzica x na czarno



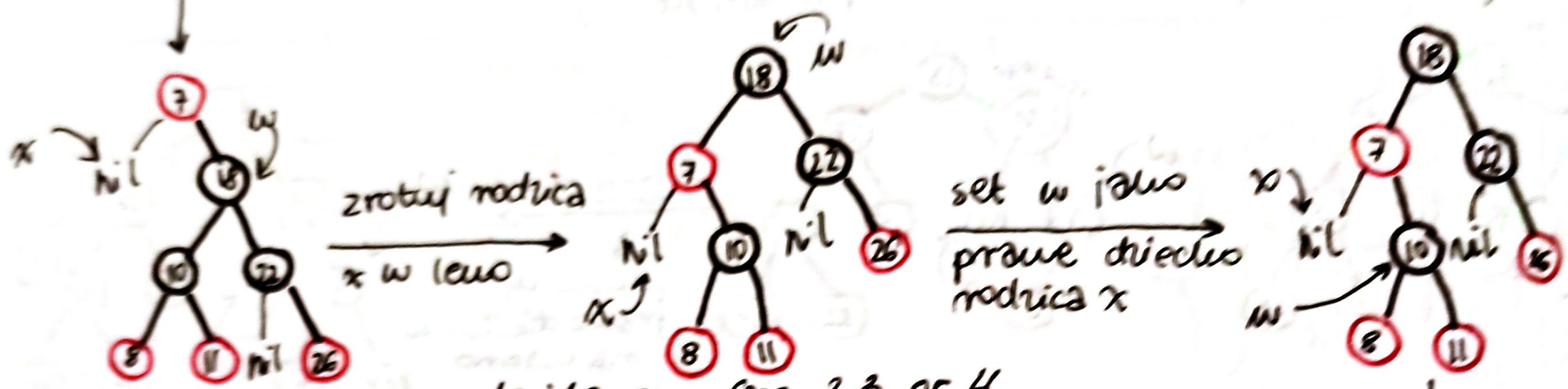
3. pokoloruj lewe dziecko w na czarno →
4. zrotuj rodzica x w prawo



We want to delete 3

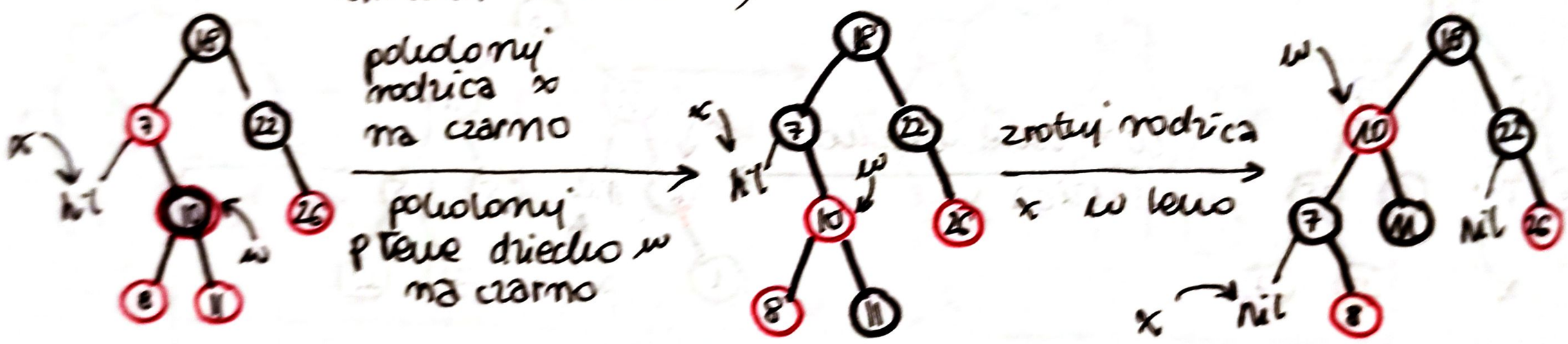


pokoloruj w na czarno, pokoloruj rodzica x na czerwono

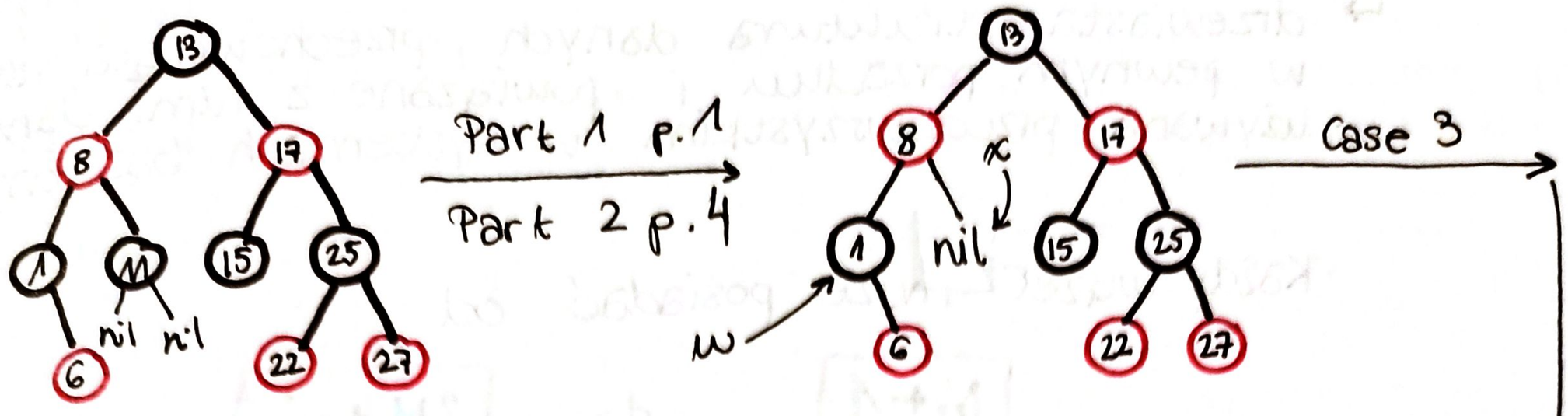


decide on Case 2, 3 or 4

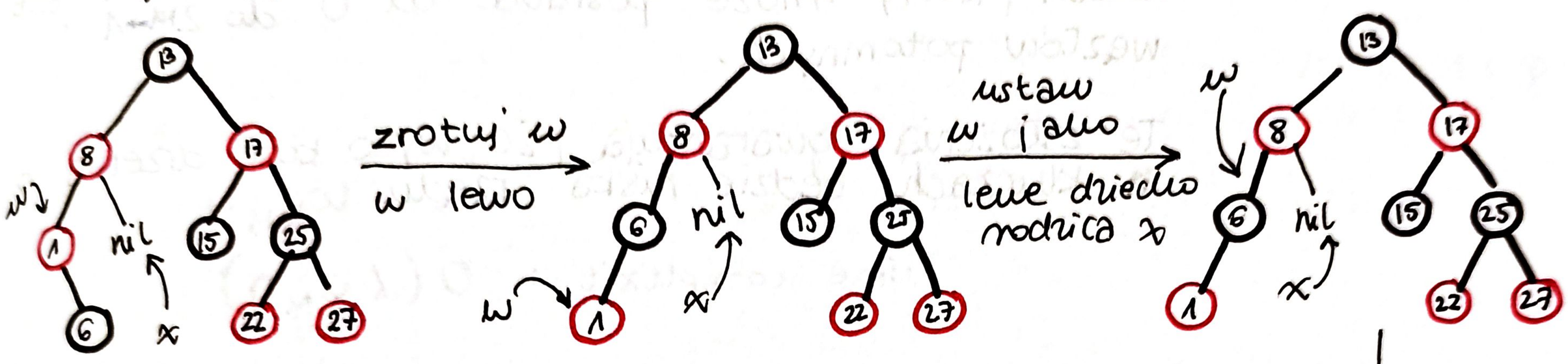
Case 4 (pokoloruj w na ten sam kolor co rodzica x)



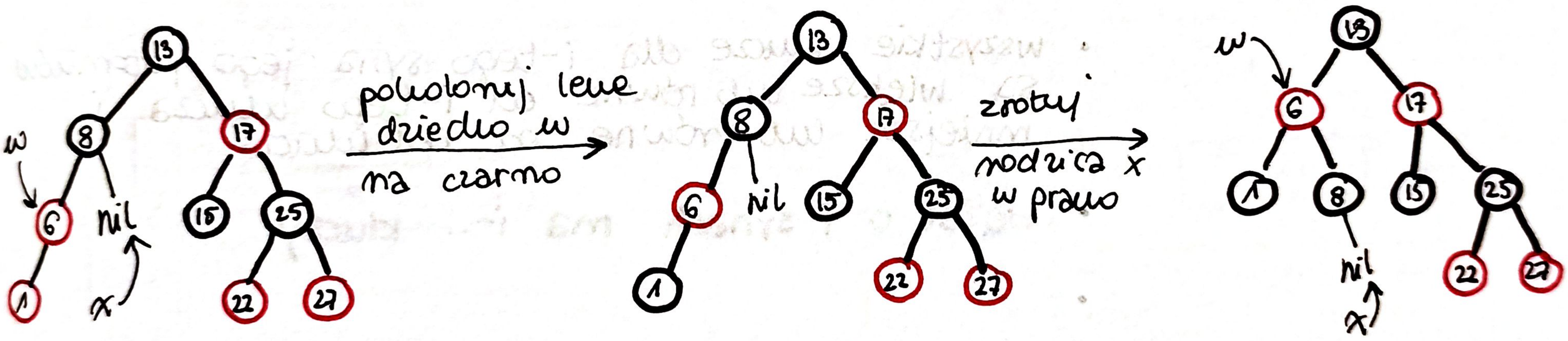
We want to delete **11**



pokoloruj prawe dziecko w na czarno
pokoloruj w na czerwono



Case 4 → pokoloruj w na ten sam kolor co rodzic x + pokoloruj rodzica x na czarno



6.9 - B-drzewo

↳ drzewiasta struktura danych, przechowująca klucze w pewnym porządku i powiązane z nimi dane, używana przedewszystkim w systemach baz danych

Każdy węzeł może posiadać od

$\boxed{T-1}$ do $\boxed{2T-1}$

węzłów, gdzie T to rząd B-drzewa. Wyjątkiem jest korzeń, który może posiadać od 0 do 2T-1 węzłów potomnych.

Te założenia gwarantują, że wysokość drzewa o n kluczach będzie niska, rzędu $\log_M n$

Time complexity: $O(\log_M n)$

Każde B-drzewo jest rzędu T co oznacza, że każdy z węzłów ma od T do 2T kluczy

- wszystkie klucze dla i-tego syna jego potomków są większe lub równe od i-tego klucza i mniejsze lub równe od i+1 klucza
- **węzeł o i synach ma i-1 kluczy**

$$N_{max} = \frac{1-(2T)^{h+1}}{1-2T}$$

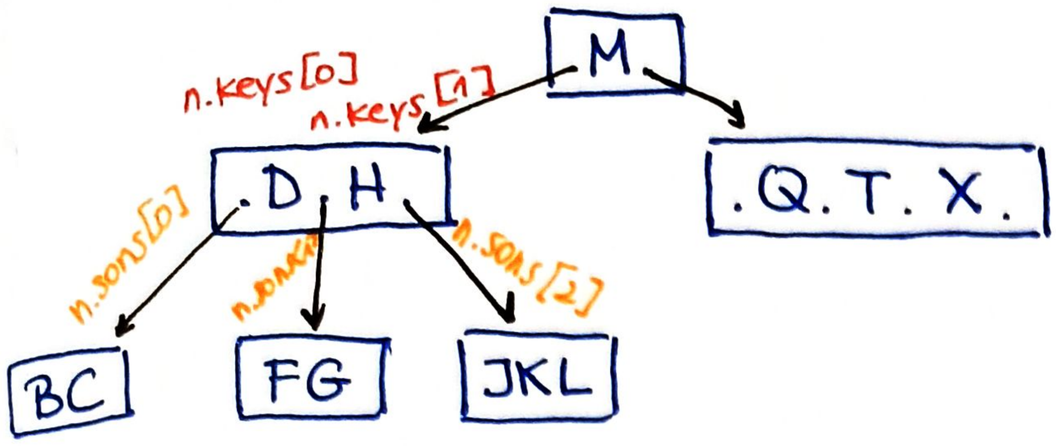
$$N_{min} = 1+2\left(\frac{1-T^h}{1-T}\right)$$

$$k_{max} = (2T)^{h+1} - 1$$

$$k_{min} = (2T)^h - 1$$

h - poziom
N - węzły
k - klucze
T - minimum degree

B-trees cd.



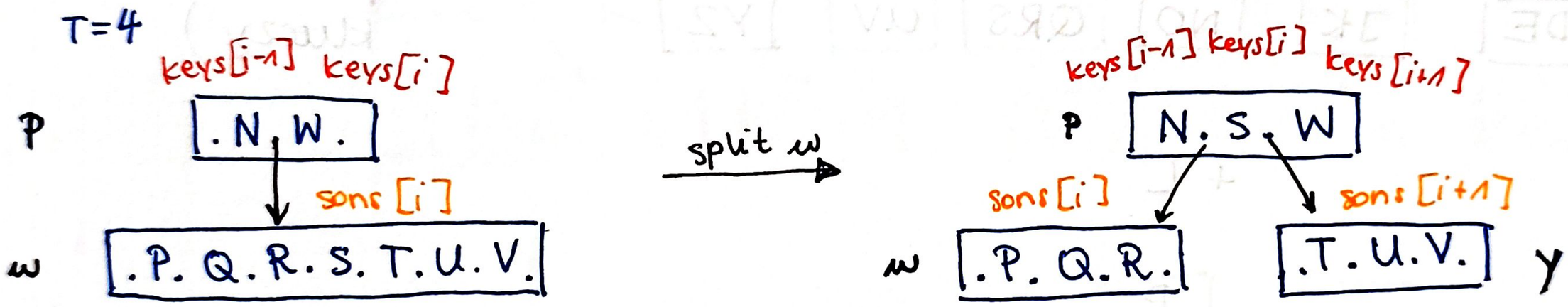
Rules:

When insert
do not enter to the maximal node (split first)

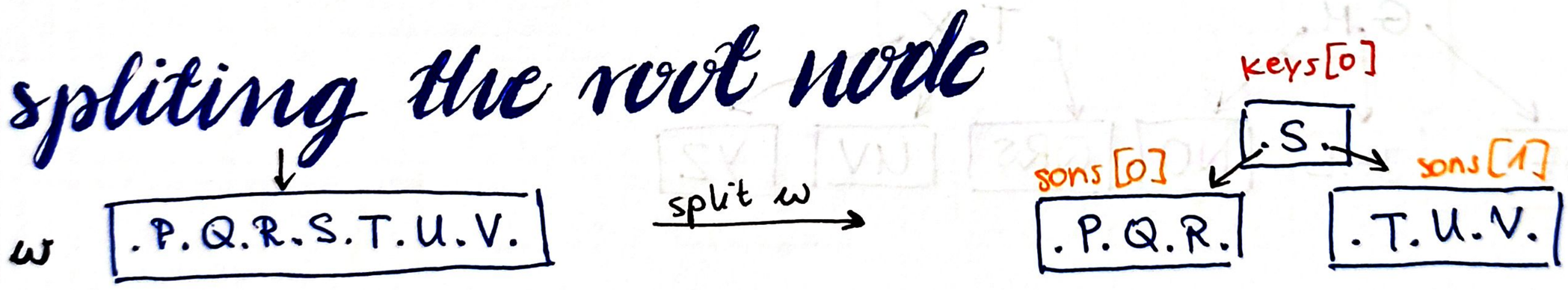
When deleting
do not enter to the minimal node (borrow a key or shift / merge)

spliting the node

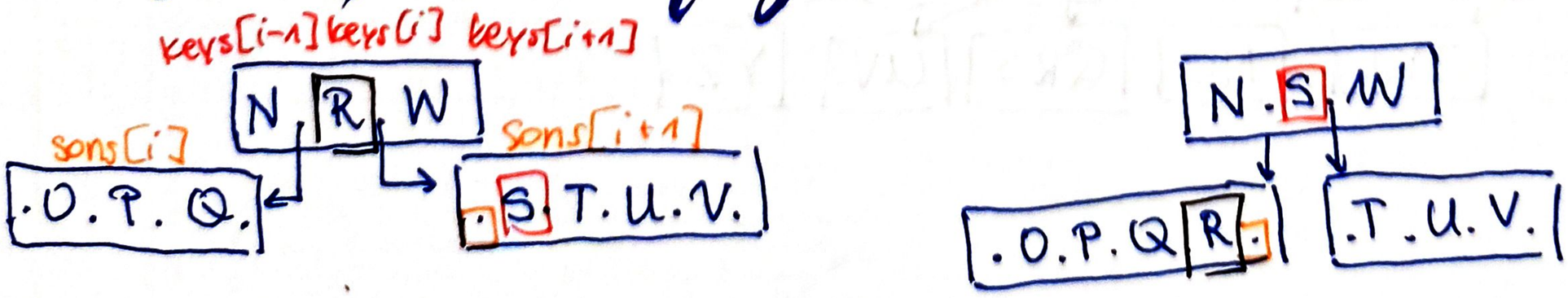
1. rozbijamy pełen węzeł w będący i-tym synem węzła p
2. środkowy z 2T-1 kluczy w wstawiamy do węzła p (przed element na pozycji i)
3. wskaźnik na nowy węzeł y wstawiamy do węzła p (przed element na pozycji i)
4. T-1 kluczy z w przepisujemy do y
5. T wskaźników z w przepisujemy do y
6. zwracamy nowy węzeł



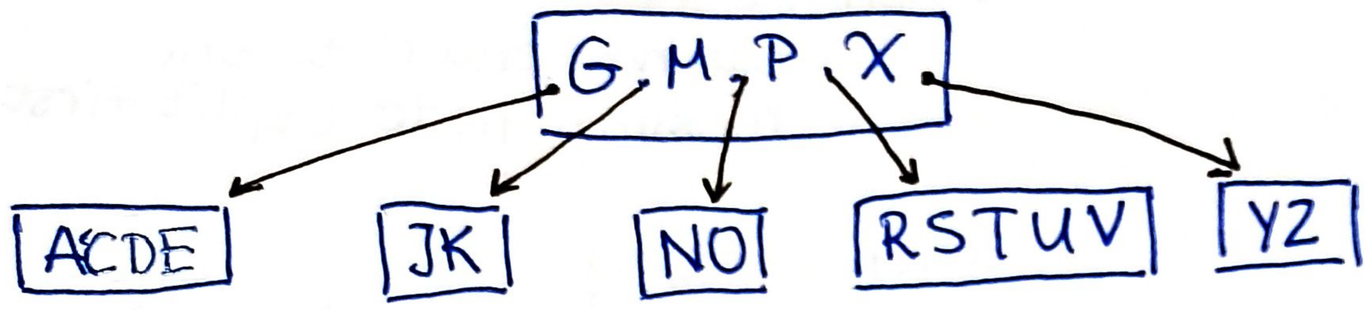
spliting the root node



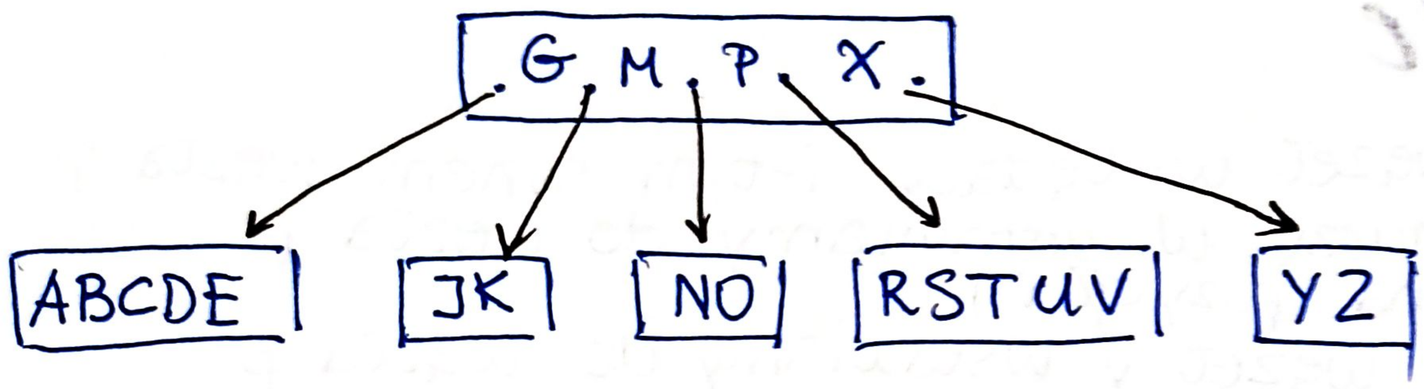
shift of the key



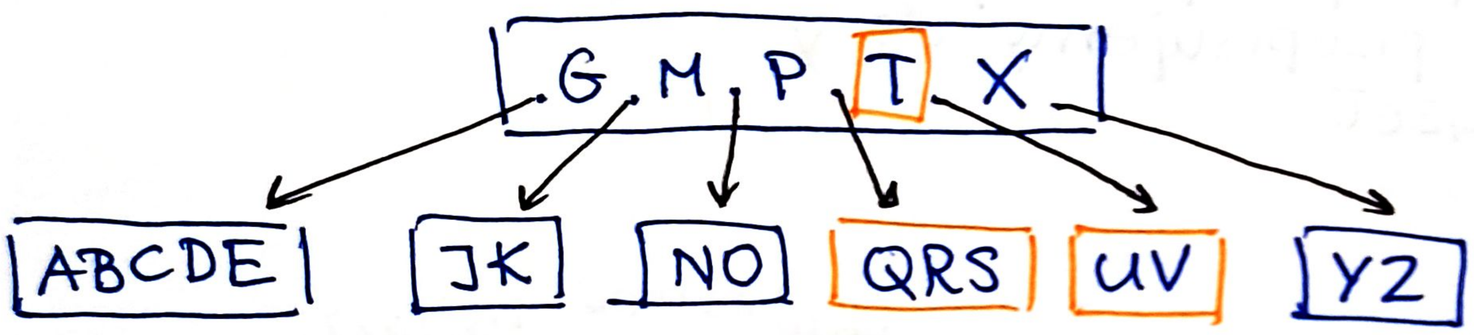
example T=3



+B



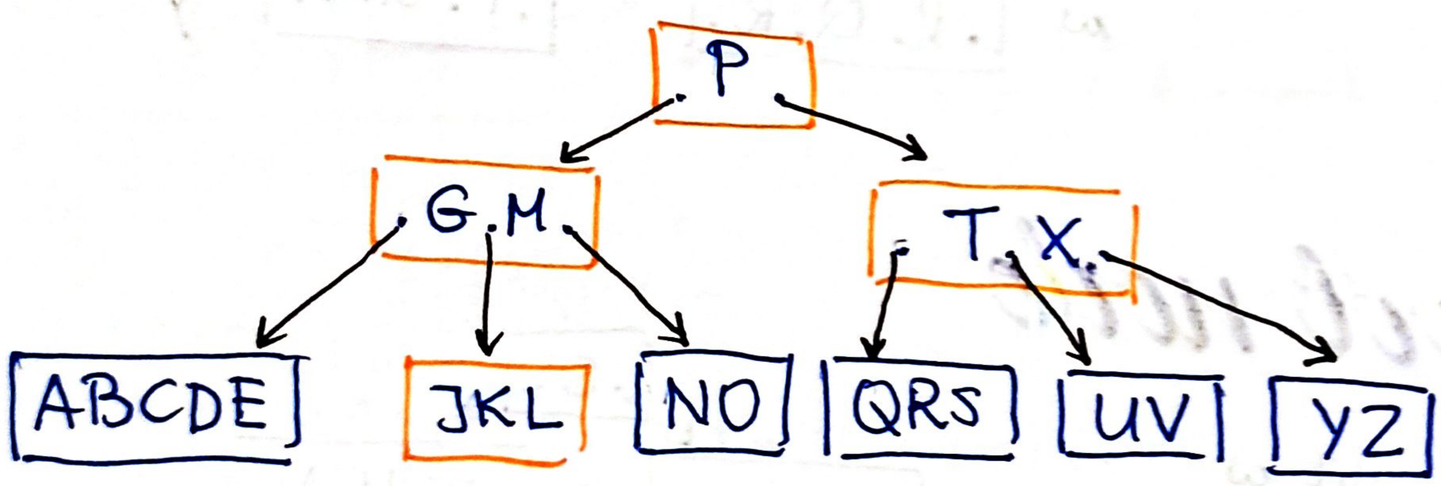
+ Q



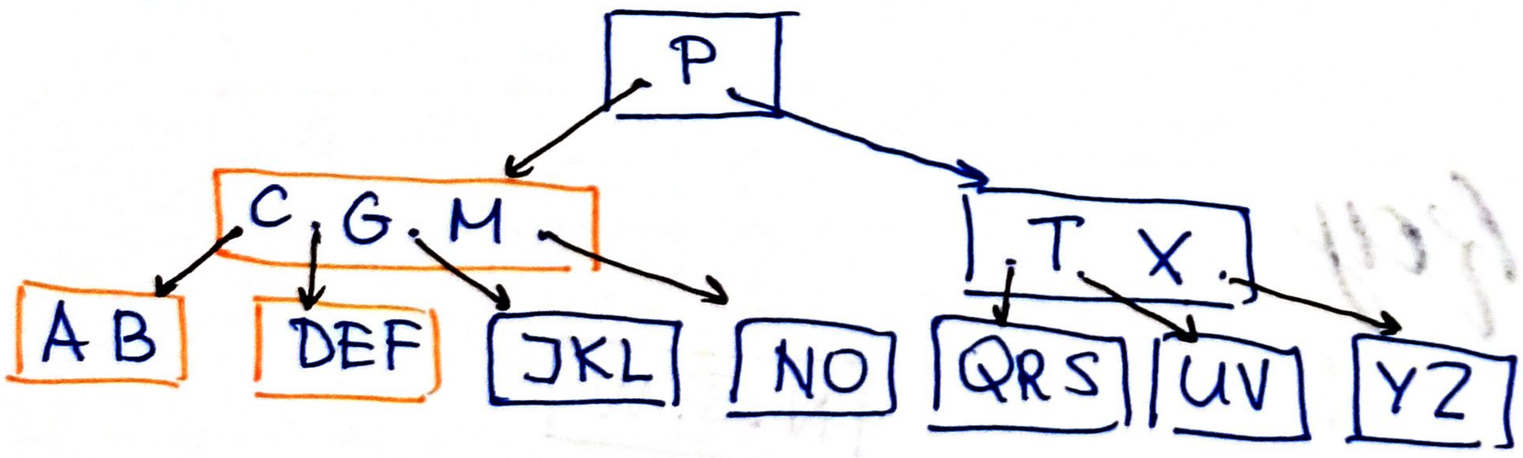
ważne

rozbijaj root jak widzisz że już ma prawie full (T-1 kluczy)

+ L

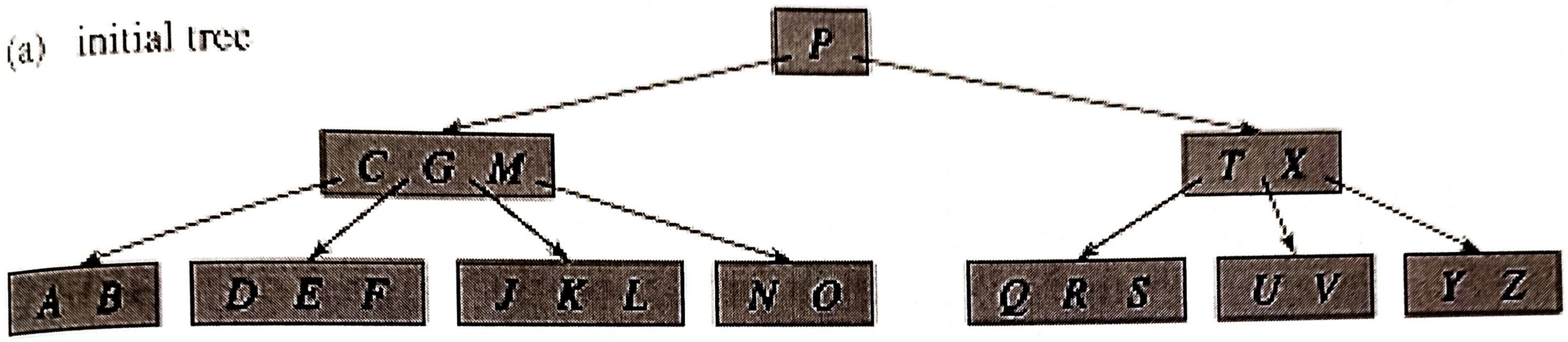


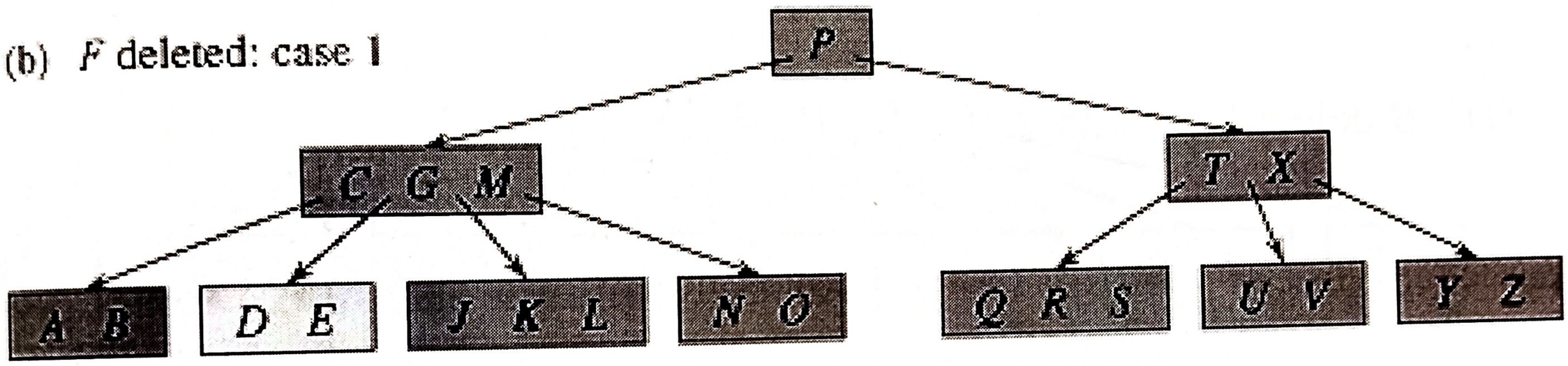
+ F

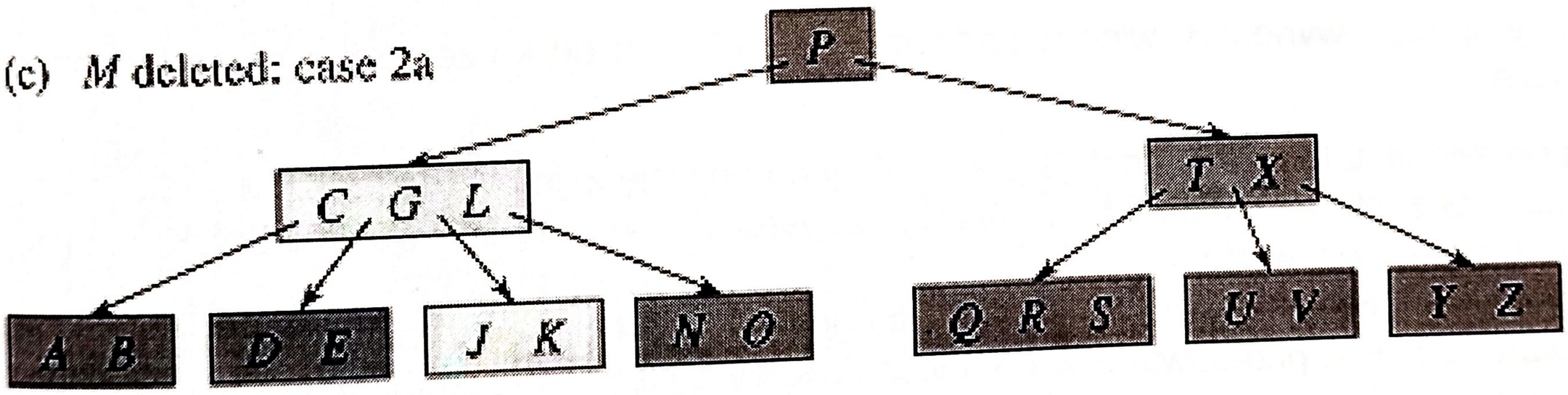


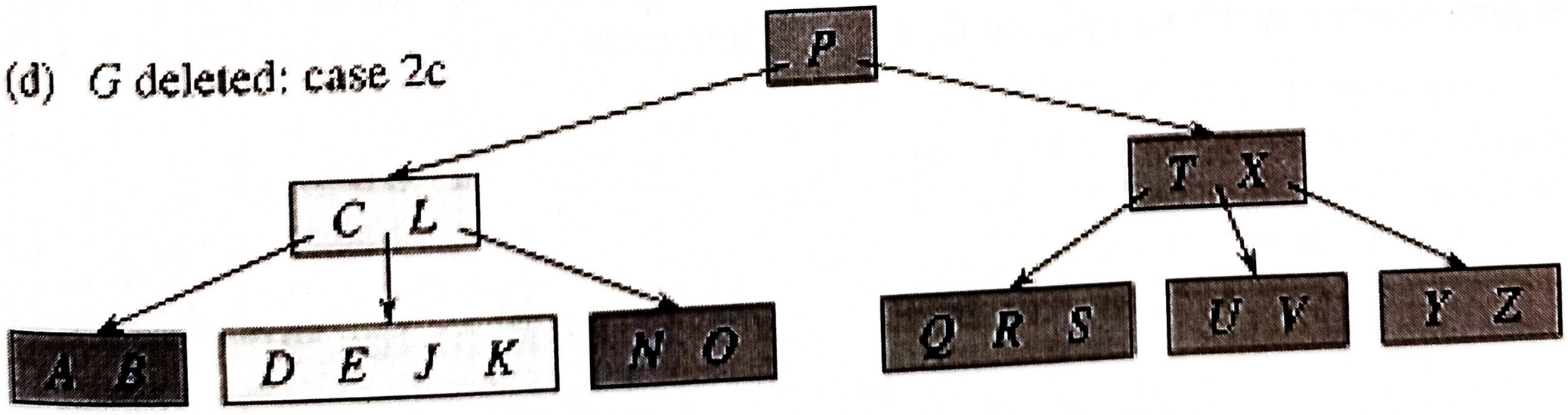
Usuwanie z B-drzewa o T=3

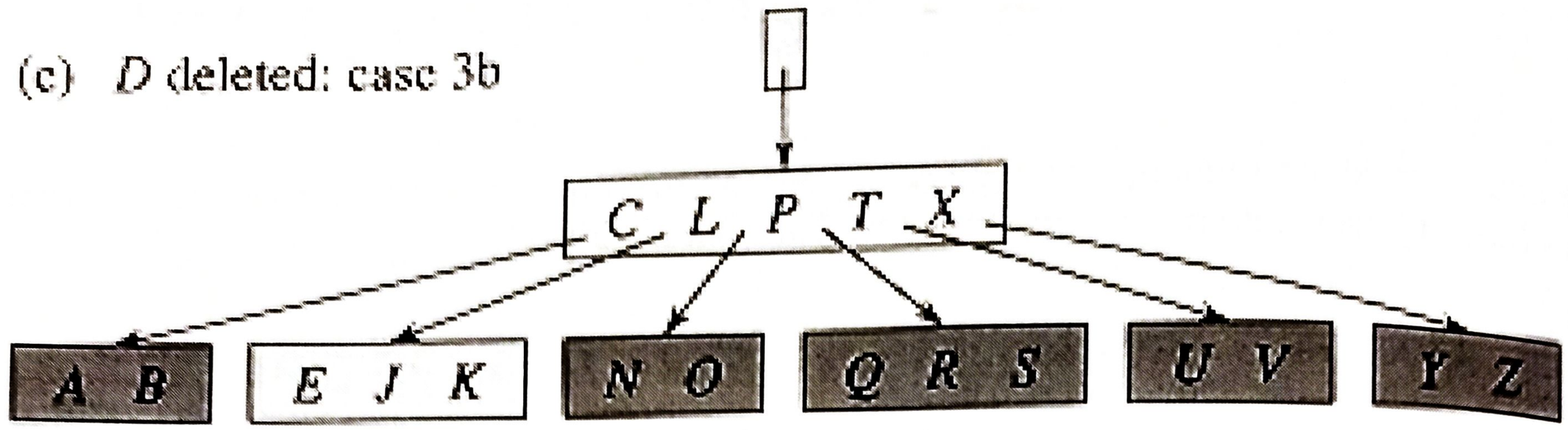
Jeśli usuwając klucz musimy przejść przez jakiś węzeł z minimalną liczbą kluczy – łączymy ten węzeł (nawet jeśli wcale nie z niego chcemy dany klucz usunąć).

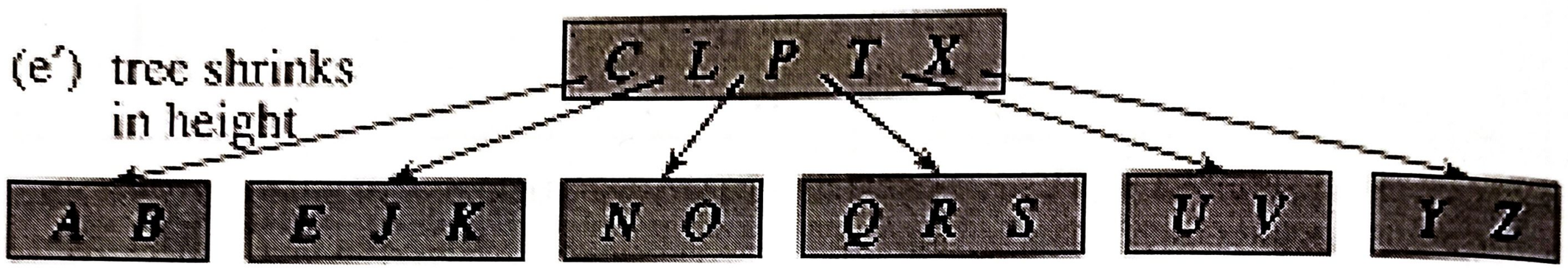


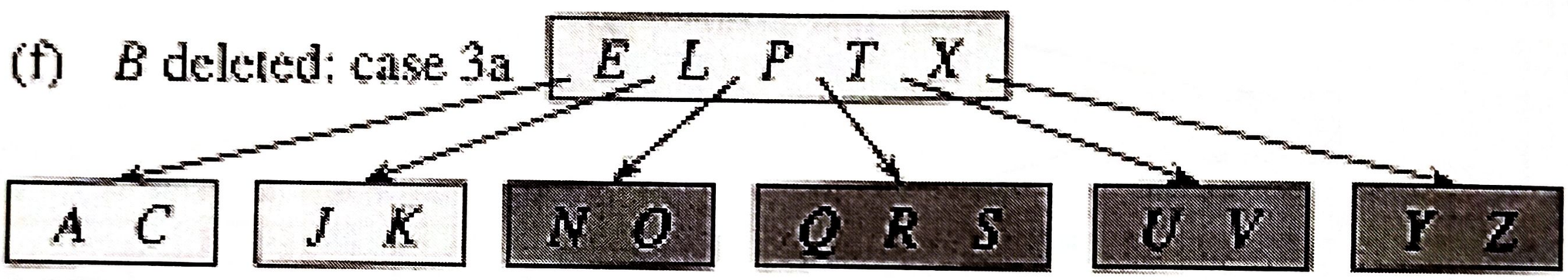










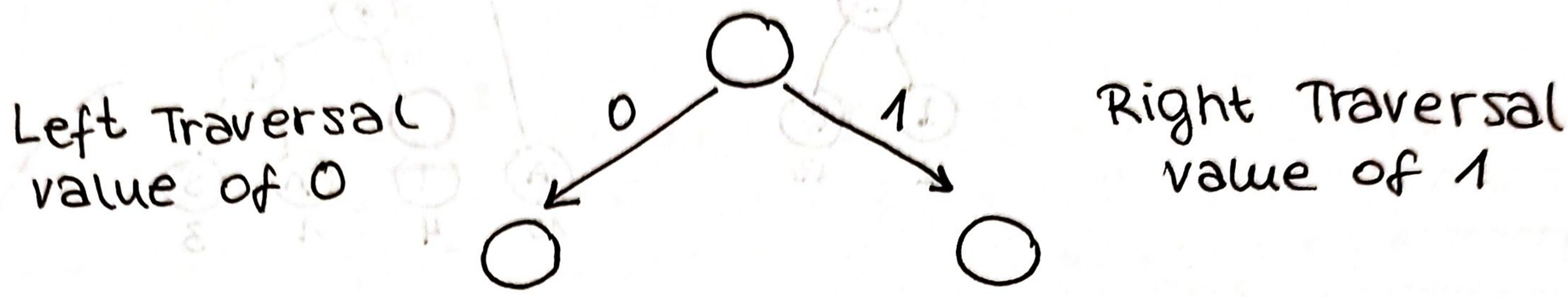


Minimalny stopień wynosi 3, więc w żadnym węźle różnym od korzenia nie może być mniej niż dwa klucze.

c) poprzednik L klucza M zostaje przesunięty na pozycję zajmowaną przez M.
d) klucz G zostaje przesunięty w dół i wykorzystany do utworzenia węzła DEGJK, a następnie jest z niego usuwany.
e) nie można wykonać zejścia rekurencyjnego do węzła CL, ponieważ są w nim tylko 2 klucze. Dlatego P jest przesuwane w dół i współtworzy z CL i TX węzeł CLPTX. Następnie D jest usuwane z liścia.
e) ‘Po wykonaniu e) zostaje usunięty korzeń drzewa i wysokość drzewa maleje o 1.
f) C jest przesuwane w dół na miejsce C, a E jest przesuwane w górę na miejsce C.

7. KODOWANIE HUFFMANA

- kodowanie o stałej długości – kody wszystkich znaków są jednakowe (np. A = 001 B = 010 C = 100)
- kodowanie o zmiennej długości – kody znaków mają różne długości (im rzadszy znak, tym większa długość kodu
- kod prefixowy (kod dowolnego znaku nie jest prefixem innego



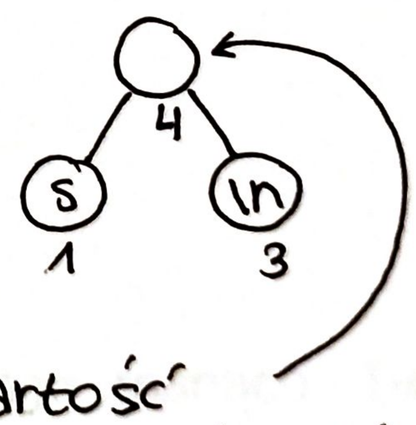
Jak działa kodowanie Huffmana?

Mamy do dyspozycji tabelę

CHAR	A	E	I	S	T	P	\n
FREQ	10	15	12	3	4	13	1

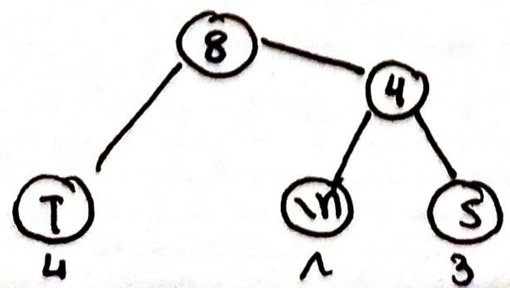
Step 1 Weź 2 chary o najmniejszej częstotliwości i stwórz z nich dwa liście w drzewie

CHAR	A	E	I	S	T	P	\n
FREQ	10	15	12	3	4	13	1



rodzic też ma wartość jest to suma częstotliwości jego dzieci, czyli w tym przypadku 3 + 1 = 4

Step 2 Weź kolejny char z najniższą częstotliwością i dodaj do drzewa

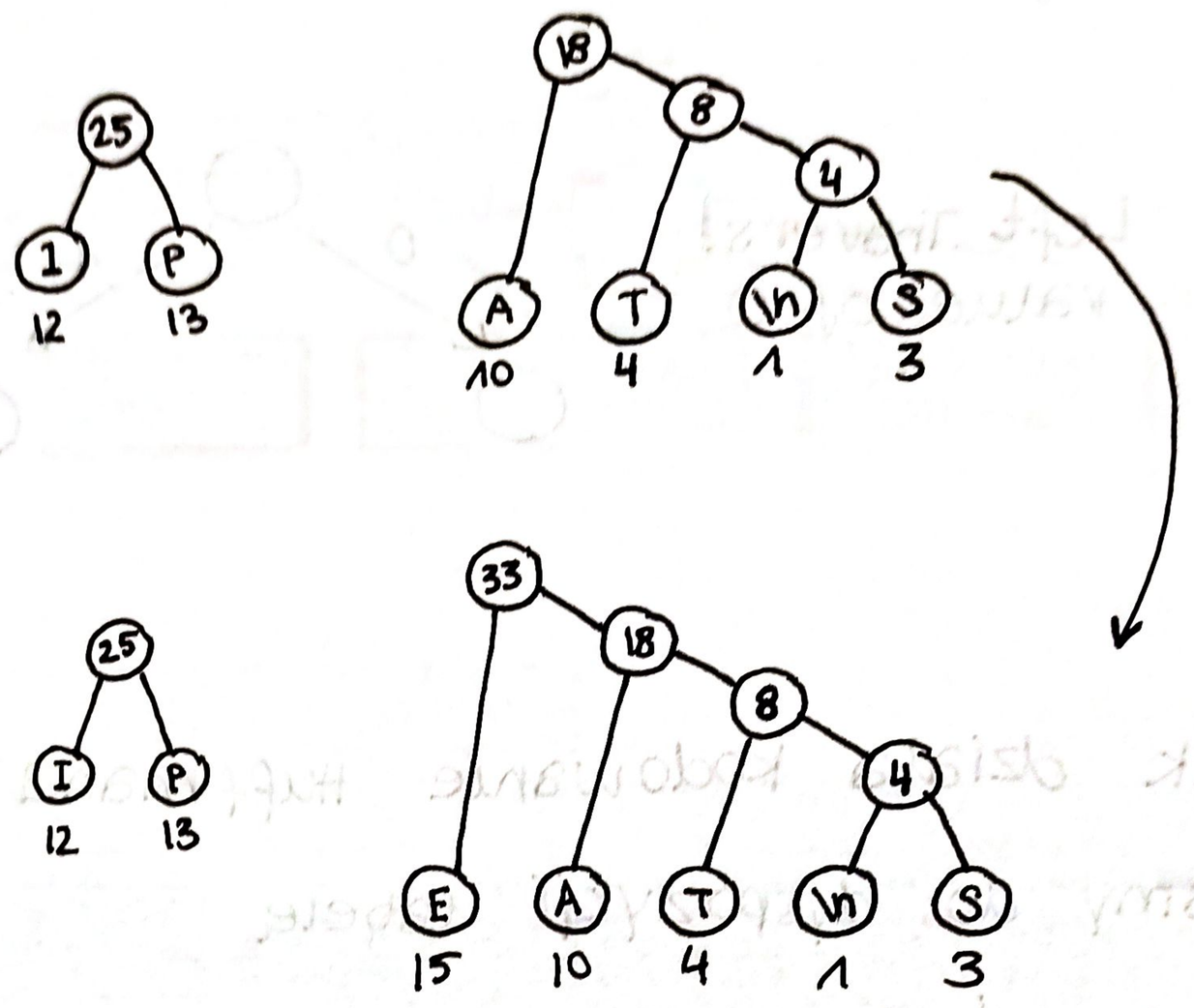


(powtarzaj ten krok, aż skończy się tabela

UWAGA! Jeżeli podczas tworzenia drzewa okaże się, że rodzic będzie miał większą wartość niż suma częstotliwości dzieci, to wtedy tworzymy nowe drzewo.

Liście chcemy dodawać do drzewa które ma najmniejszą wartość

Example

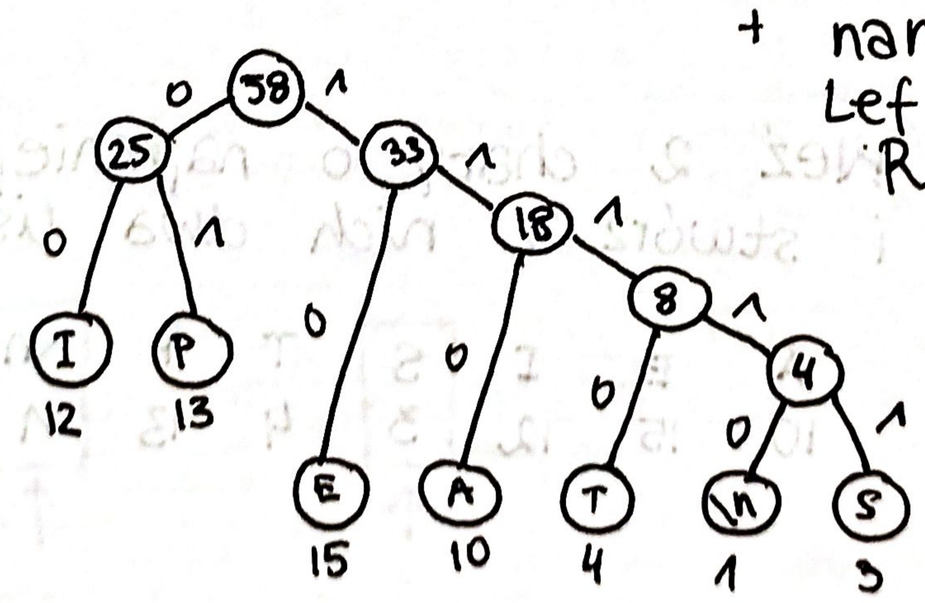


dodajemy char E o freq. 15

25 > 18

więc dodajemy do drzewa 18

Step 3 Połącz wszystkie powstałe drzewa w jedno.
+ nanieś wartości Left Traversal i Right traversal



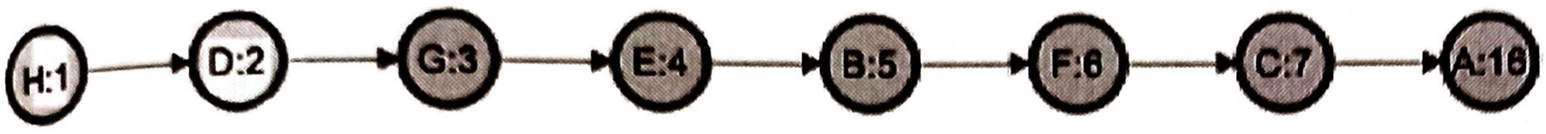
CHAR	A	E	I	S	T	P	\n
CODE	110	10	00	11111	1110	01	11110
FREQ	10	15	12	3	4	13	1
BITS	30	30	24	15	16	26	5

Kodowanie Huffmana

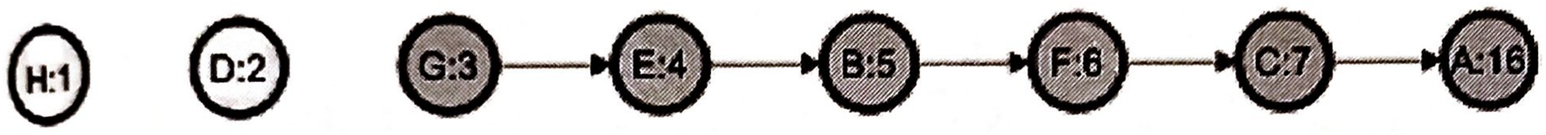
Kodowanie Huffmana (bezstratne) polega na przypisywaniu znakom występującym najczęściej w kodowanej wiadomości kodów krótkich. Znaki pojawiające się rzadziej otrzymują kody dłuższe. To zróżnicowanie długości pozwala otrzymać kod o mniejszej liczbie bitów, niż gdybyśmy stosowali kody o stałej długości.

Załóżmy, że chcemy zakodować następującą wiadomość:
ACBECAHCADFEGAFAGACBBADAAFAAEAGACAFABEFBCCFA

Obliczamy ilość wystąpień każdego znaku: A:16, B:5, C:7, D:2, E:4, F:6, G:3, H:1.

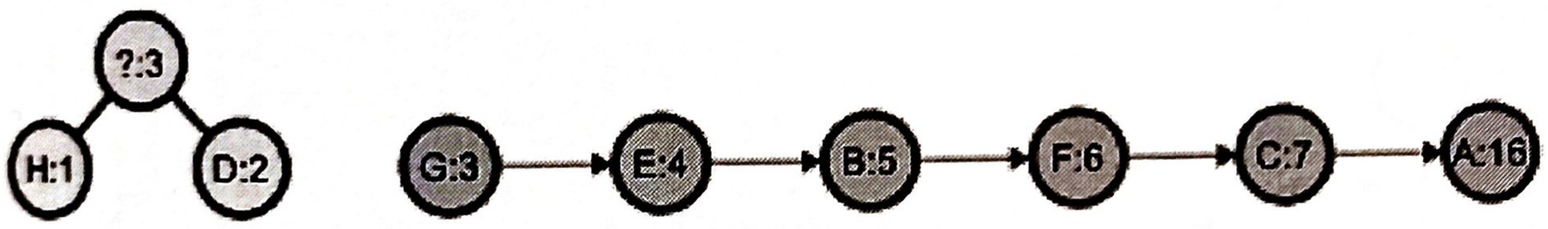


Tworzymy listę węzłów drzewa kodu bezprzystankowego. Na liście umieszczamy kolejne węzły, które w docelowym drzewie będą liśćmi. Każdy węzeł przechowuje kodowany znak oraz częstość wystąpień tego znaku w kodowanym tekście. Lista jest uporządkowana wg rosnącej liczby wystąpień. Dla naszego tekstu pierwszy węzeł zawiera znak H, który w tekście pojawia się jeden raz. Ostatni węzeł zawiera znak A o największej liczbie wystąpień, równej 16.

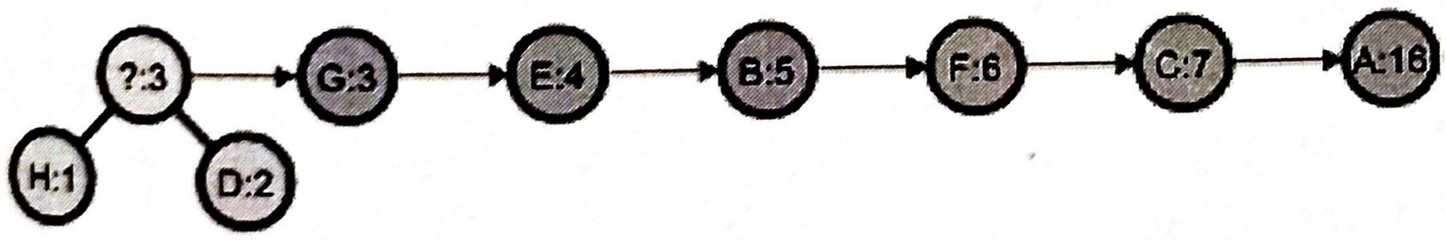


Dopóki na liście jest więcej niż jeden element, będziemy powtarzać następującą operację:

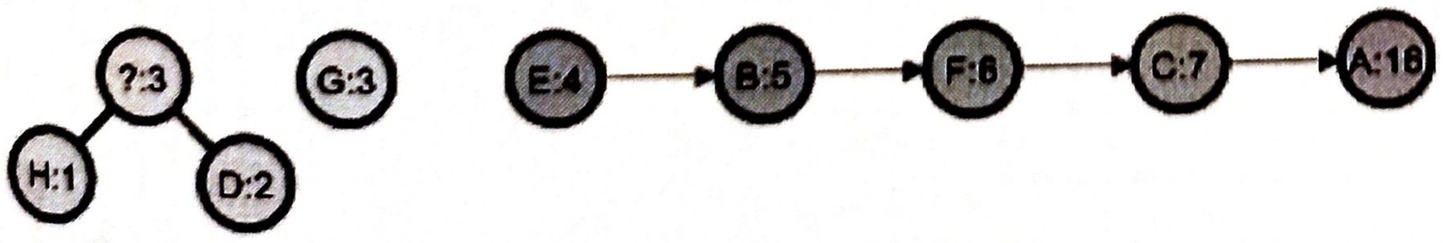
Z listy pobieramy dwa pierwsze węzły. Węzły te zawsze zawierają znaki o najmniejszej liczbie wystąpień. U nas są to węzły H:1 i D:2.



Tworzymy nowy węzeł i dołączamy do niego dwa pobrane z listy węzły. W nowym węźle zapamiętujemy sumę wystąpień znaków w dwóch pobranych węzłach. Węzeł nie musi pamiętać żadnego znaku, ponieważ w drzewie kodu bezprzystankowego będzie on węzłem wewnętrznym. Tutaj stosujemy znak '?".

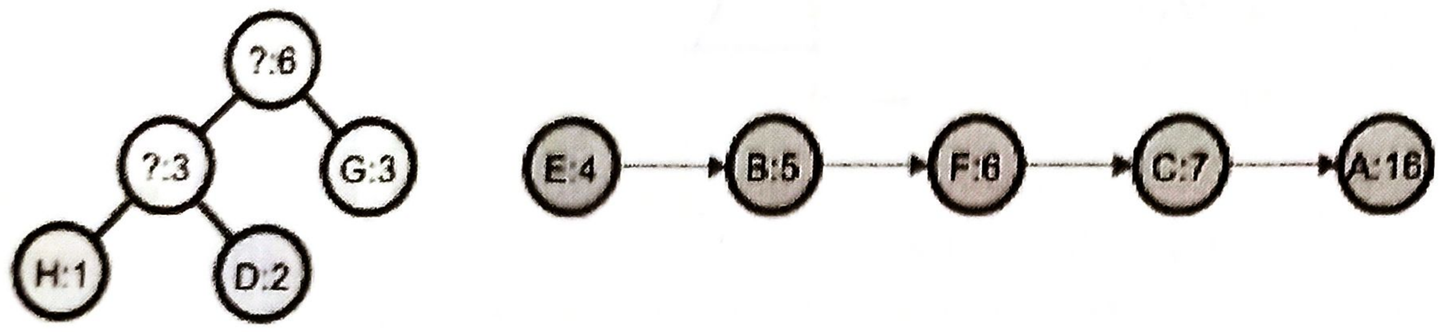


Utworzony węzeł dodajemy do listy tak, aby była wciąż uporządkowana rosnąco. Tutaj trafi na początek listy.

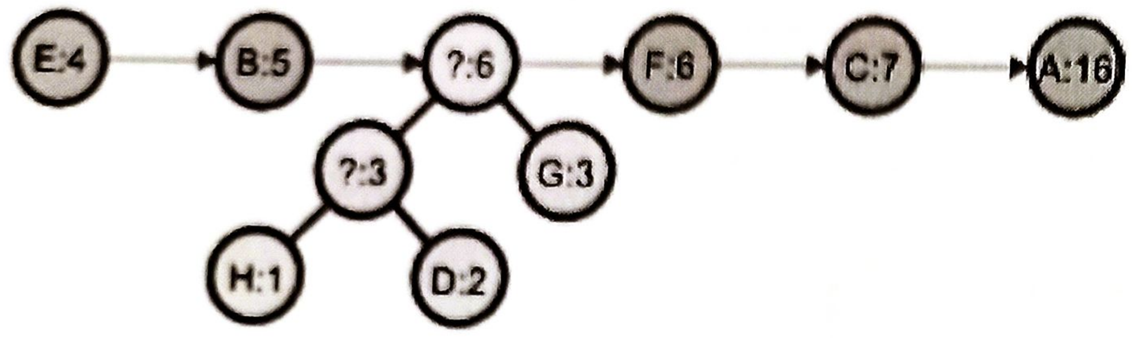


Powtarzamy powyżej opisane operacje dla dwóch pierwszych węzłów na liście:

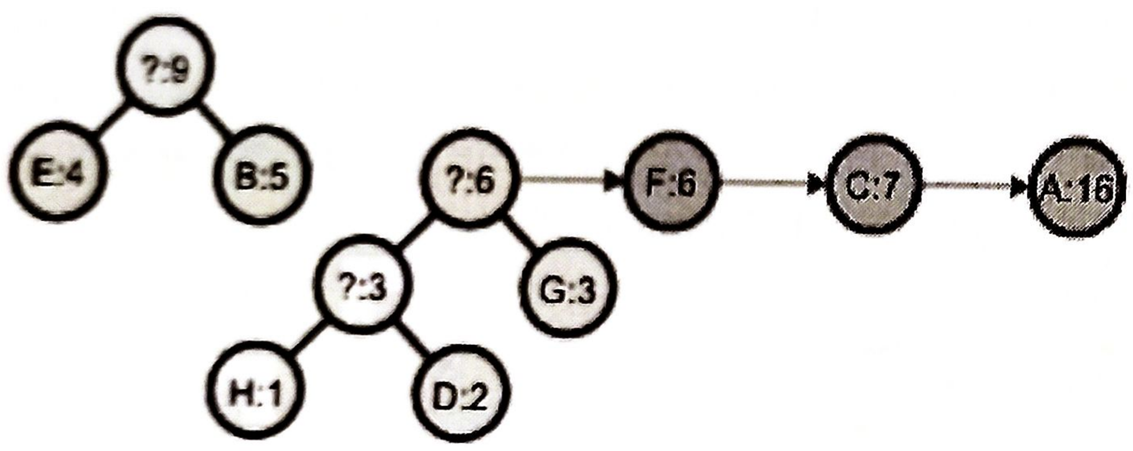
Odłączamy je od listy.



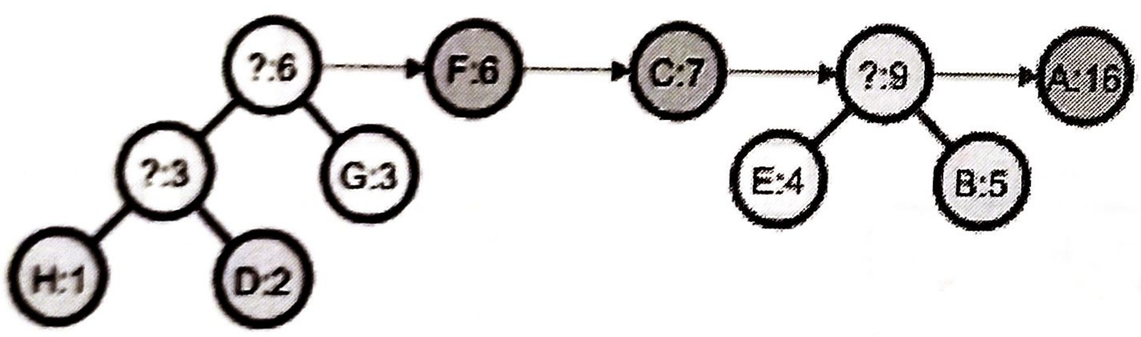
Tworzymy nowy węzeł z sumą wystąpień odłączonych węzłów. Dołączamy do niego te dwa węzły.



Węzeł wstawiamy na listę.

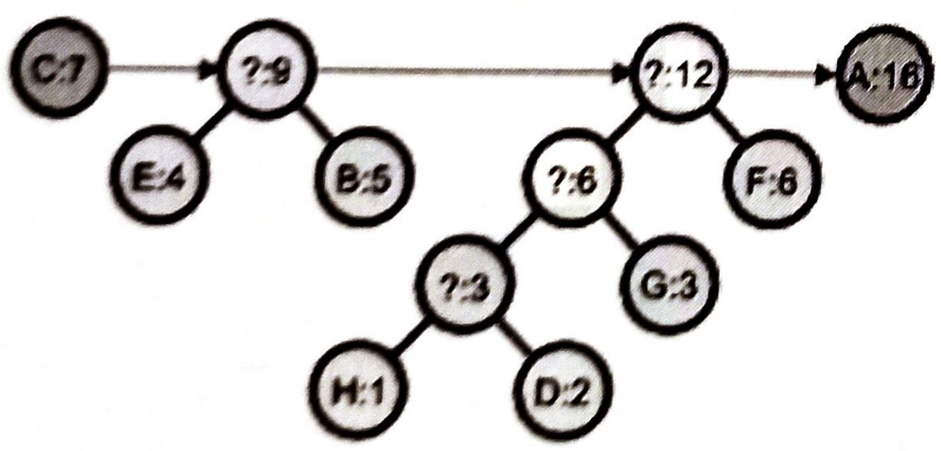


Powtarzamy te same operacje dla węzłów E:4 i B:5. Odłączamy je od listy. Tworzymy nowy węzeł. Umieszczamy w nim sumę wystąpień znaków E i B.



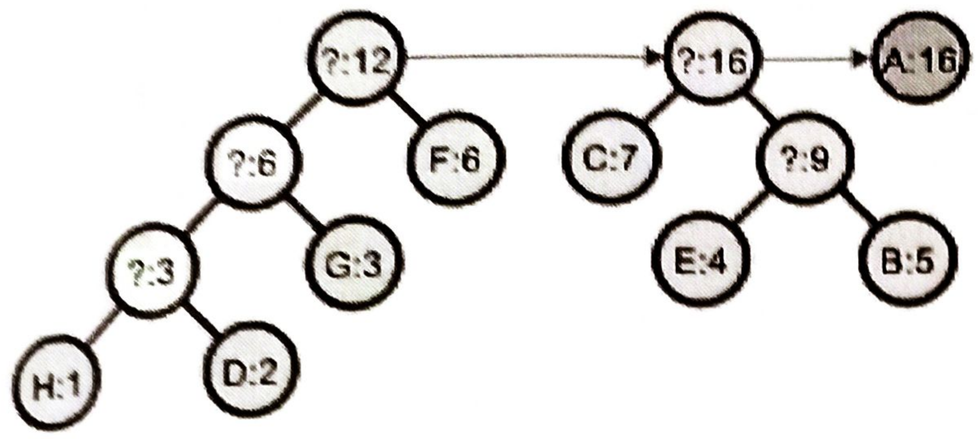
Po czym węzeł ten dołączamy do listy, zachowując jej uporządkowanie.

Wykonujemy to samo dla pary węzłów ?:6 i F:6.



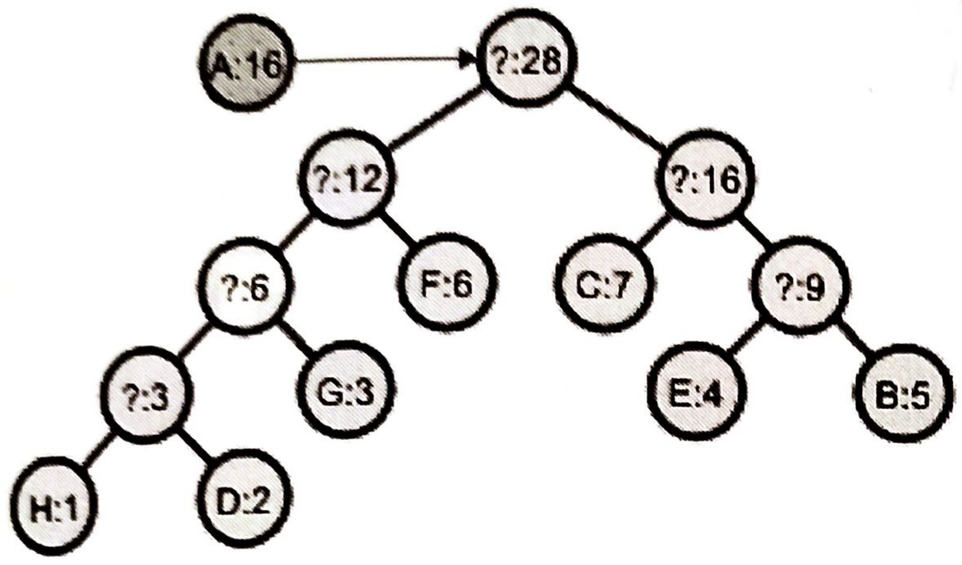
Nowy węzeł ?:12 wstawiamy na listę przed A:16.

To samo dla C:7 i ?:9.



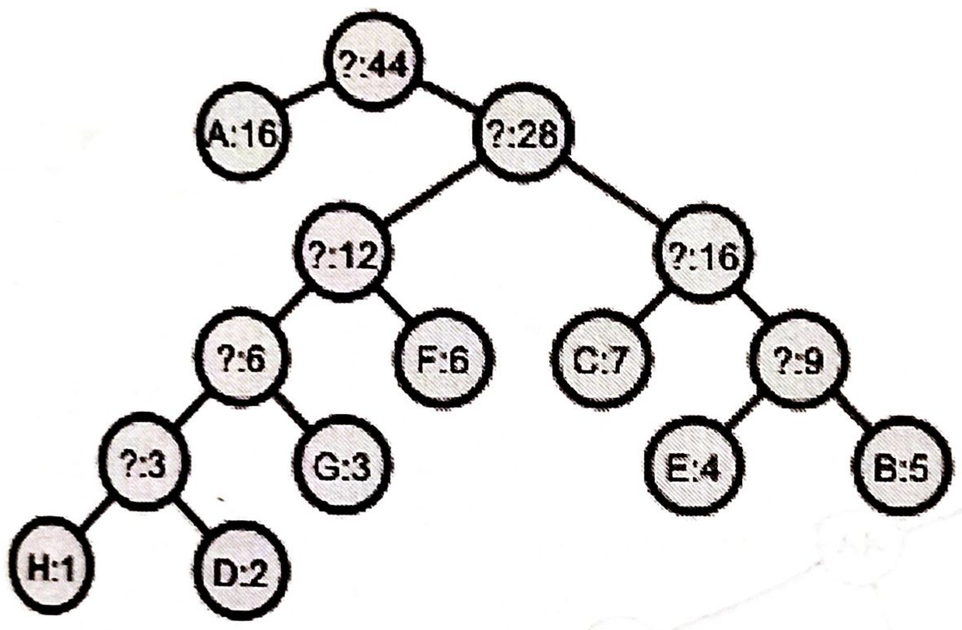
Nowy węzeł ?:16 znów trafia przed A:16.

To samo dla ?:12 i ?:16.

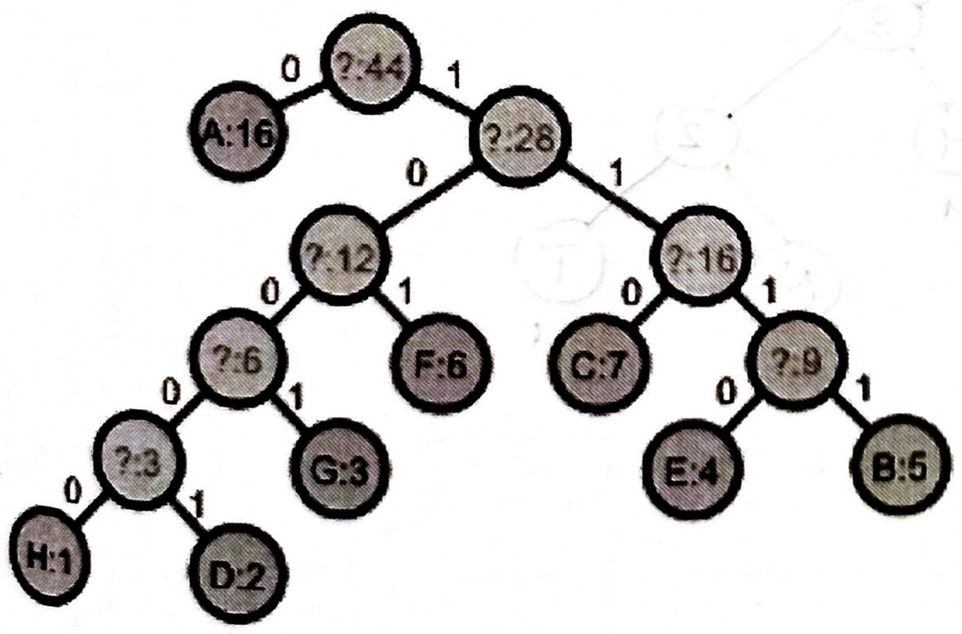


Tym razem nowy węzeł ?:28 trafia za A:16.

Na liście zostały tylko dwa węzły, A:16 i ?:26. Po raz ostatni wykonujemy dla nich opisane operacje.



W ich wyniku na liście pozostaje jeden węzeł, który jest korzeniem drzewa kodu bezprzystankowego Huffmana.



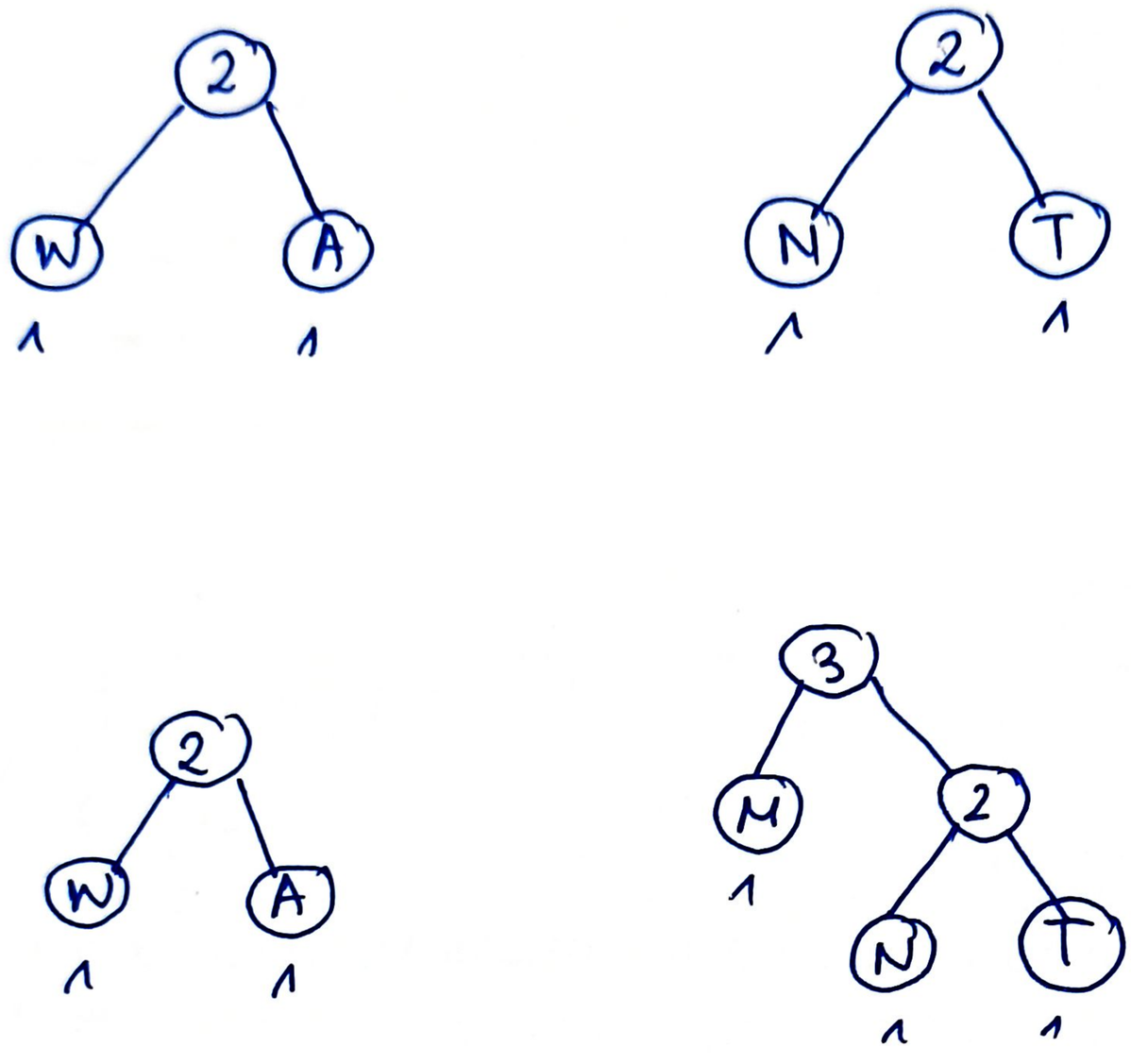
C → 110
D → 10001
E → 1110
F → 101
G → 1001
H → 10000

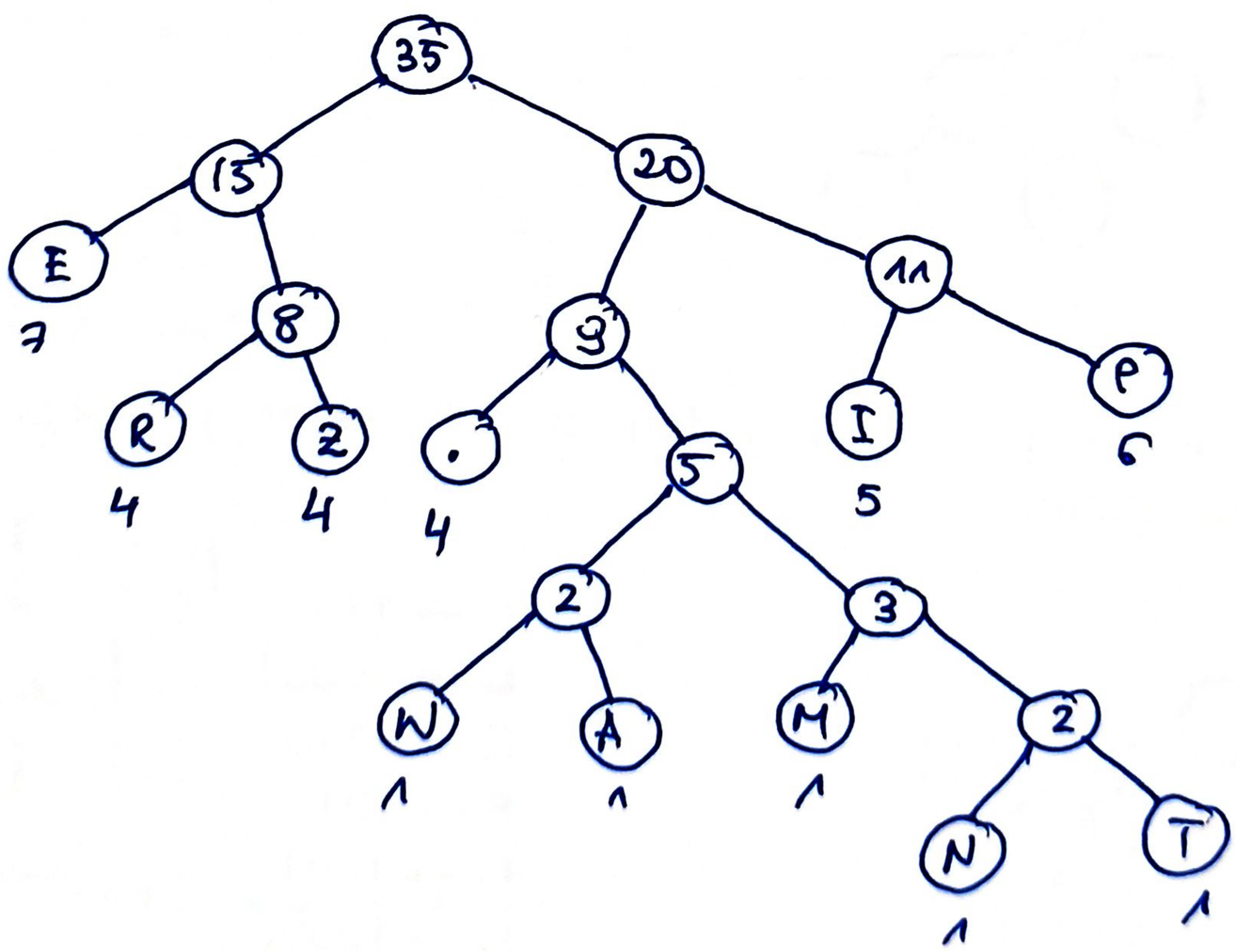
Z drzewa możemy odczytać kody dla poszczególnych znaków:

A → 0
B → 1111

Przykład Manusa

N	T	W	A	M	R	Z	.	I	P	E
1	1	1	1	1	4	4	4	6	6	7

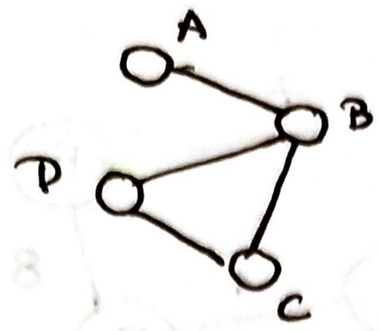


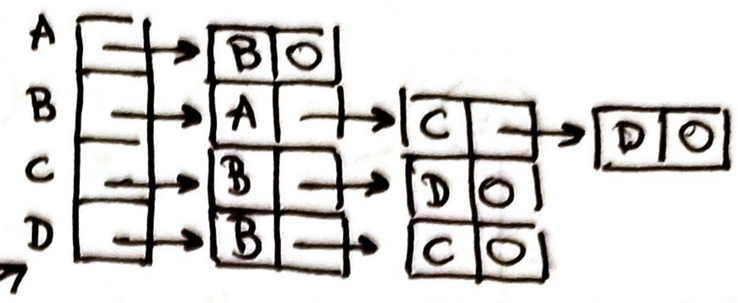


8. REPREZENTACJA GRAFU

	A	B	C	D
A		1	0	0
B			1	1
C				1
D				

macierz sąsiedztwa



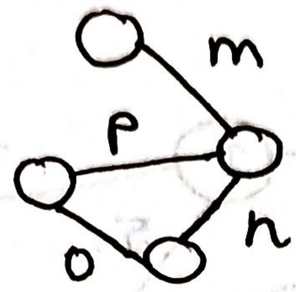


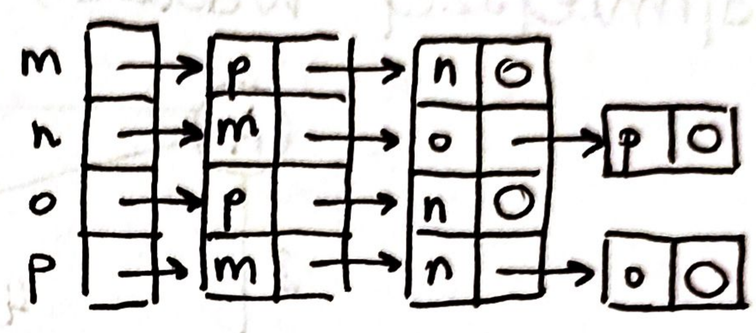
Lista sąsiedztwa

Jeżeli macierz jest symetryczna (graf nieskierowany) można przechowywać tylko jedną połówkę

	m	n	o	p
m	0	1	0	1
n	1	0	1	1
o	0	1	0	1
p	1	1	1	0

macierz incydencji

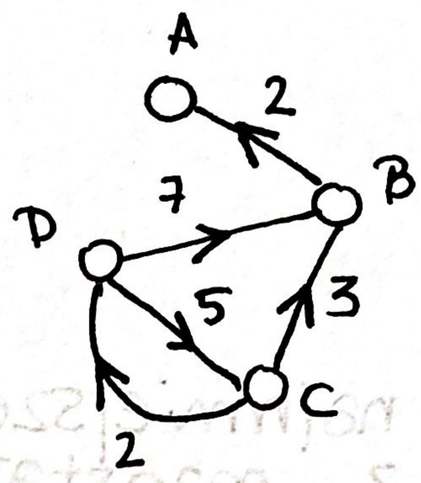


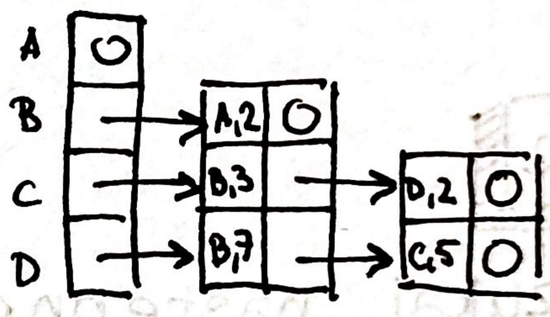


lista incydencji

	A	B	C	D
A	0	0	0	0
B	2	0	0	0
C	0	3	0	2
D	0	7	5	0

macierz sąsiedztwa



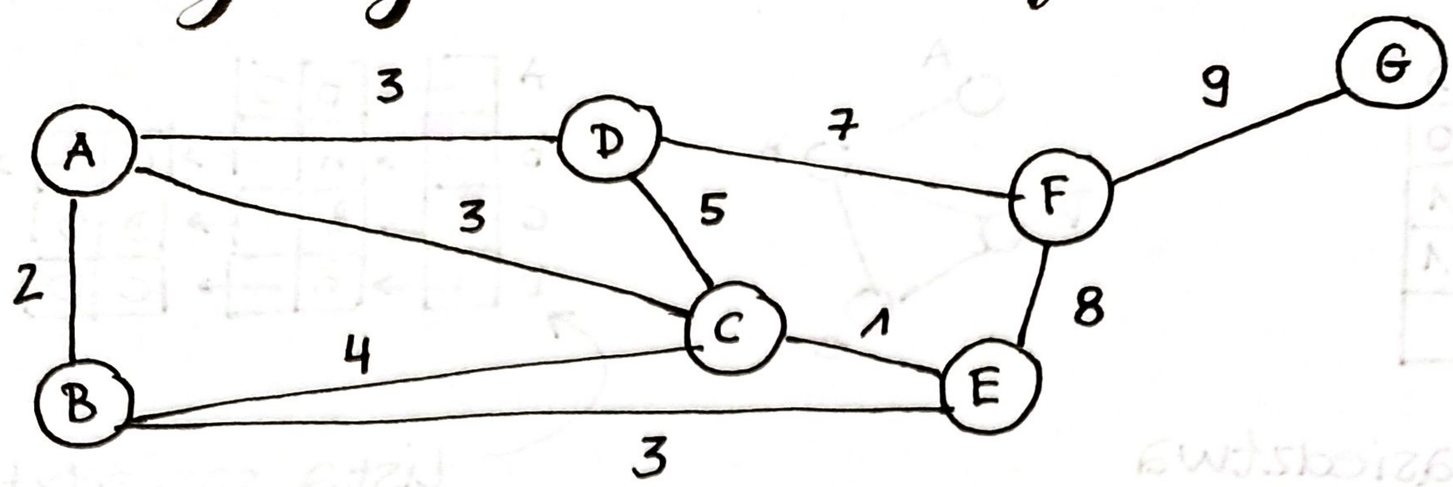


lista incydencji

8.1 - minimalne drzewo spinające

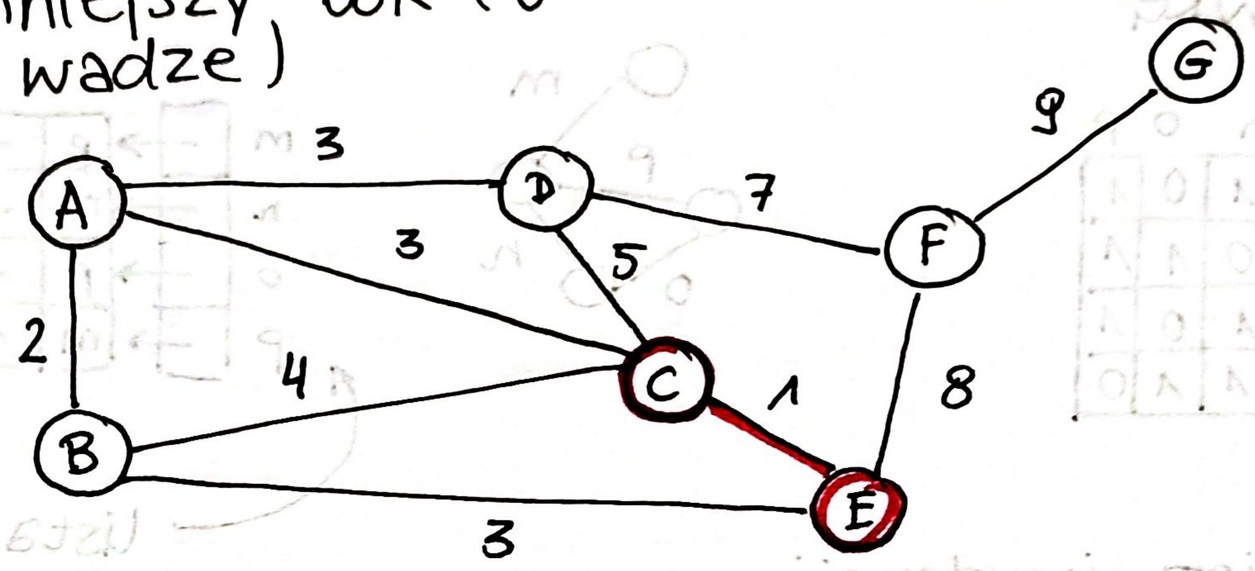
(MST) minimum spanning tree - drzewo rozpinające (spinające) danego grafu o najmniejszej z możliwych wag, tj. takie, że nie istnieje dla tego grafu inne drzewo rozpinające o mniejszej sumie wag krawędzi.

8.2 - algorytm Kruskala



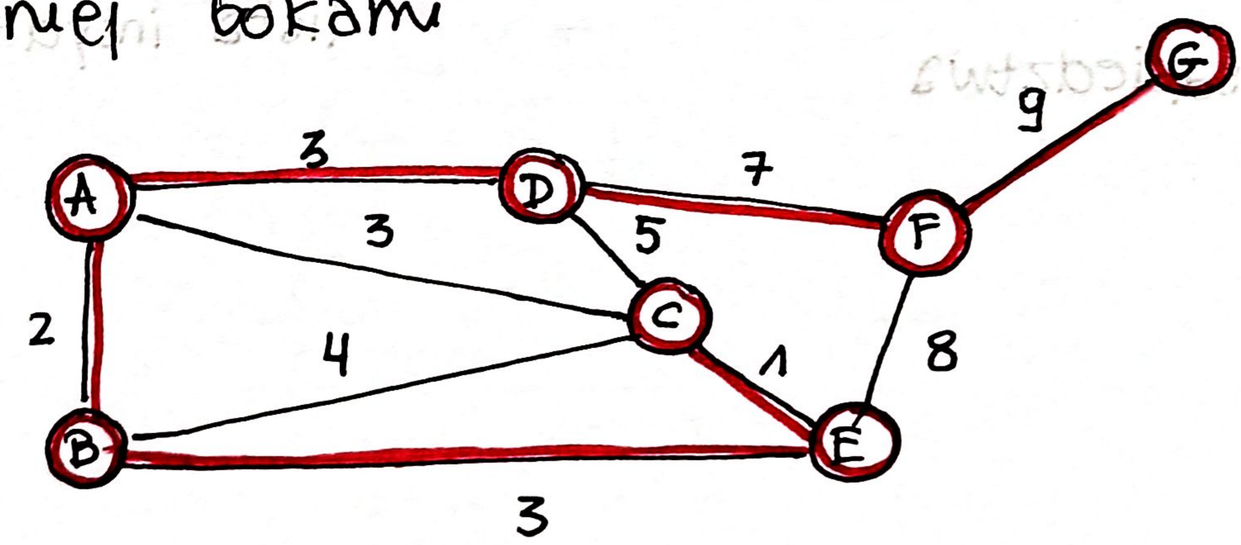
Step 1

Wybierz najmniejszy bok (o najmniejszej wadze)



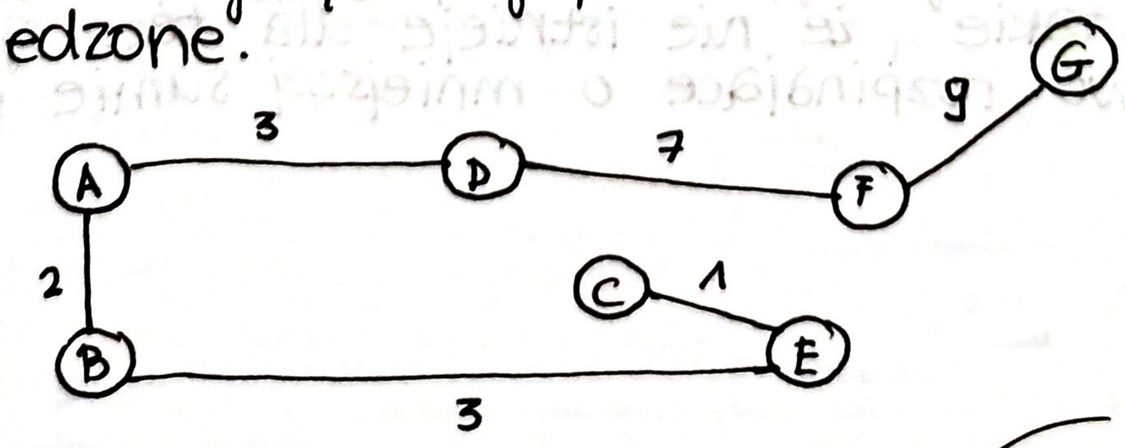
Step 2

Szukaj następnego najmniejszego boku, który nie stworzy cyklu z pozostałymi wybranymi wcześniej bokami



Step 3

Skończ algorytm gdy wszystkie wierzchołki zostaną odwiedzone.

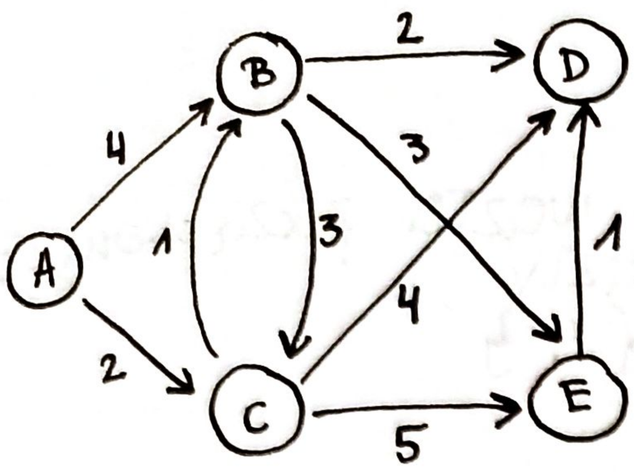


number of edges

Time complexity: $O(E \log E)$

8.3 - algorytm Dijkstry

↳ pokazuje najkrótszą drogę z jednego węzła do pozostałych



Step 1 Wybierz rozpoczynający wierzchołek (ex. A)
Będziemy używać tabeli poniżej żeby zapisywać odległości

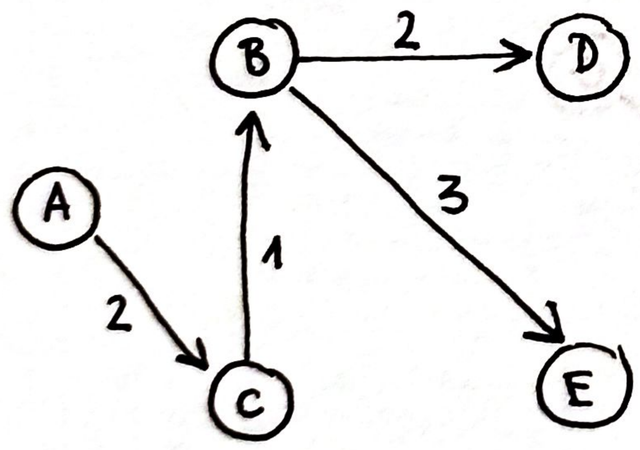
Unvisited nodes = {A, B, C, D, E}

ŚCIEŻKA	A	B	C	D	E
A	—	4, A	2, A	∞	∞
AC	—	3, C	—	6, C	7, C
ACB	—	—	—	5, B	6, B
ACBD	—	—	—	—	6, B
ACBE	—	—	—	—	—

jeżeli odwiedziliśmy dany węzeł nie musimy szukać innej drogi, chyba, że jest krótsza :)

Step 2 Sprawdź brzegi wychodzące z wybranego przez nas węzła i zaktualizuj tabelkę o ścieżki które prowadzą do najbliższych węzłów.

Step 3 Wybierz brzeg o najmniejszej wadze do wierzchołka trzeba dodać wszystkie koszty krawędzi, które łączą nasz węzeł początkowy z tym, do którego chcemy się przenieść



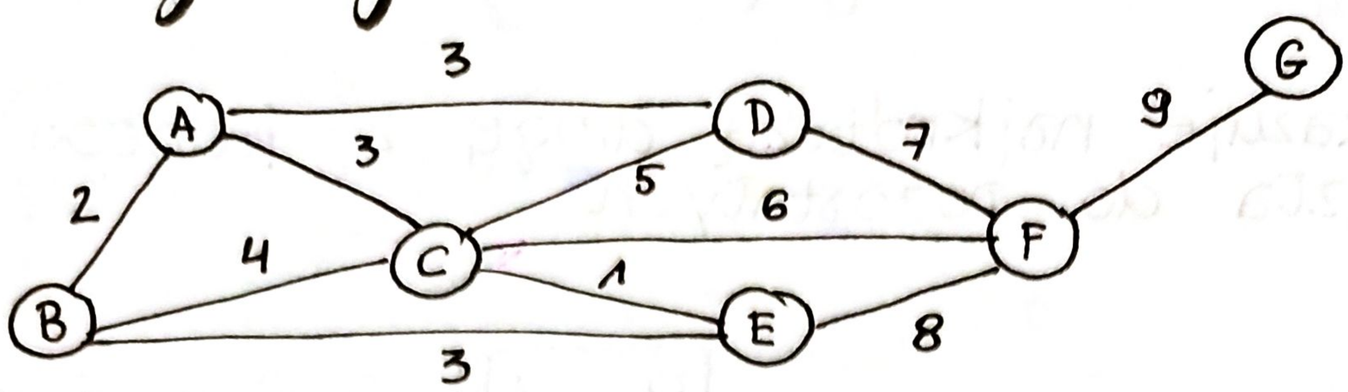
remember!
Teraz, pisząc koszt krawędzi przejścia do określonego wierzchołka, trzeba dodać koszt wszystkich krawędzi, które łączą nasz węzeł początkowy z tym, do którego chcemy się przenieść

Time complexity:

$$O(|E| + |V| \log |V|)$$

edges — vertexes

8.4 - algorytm Prima

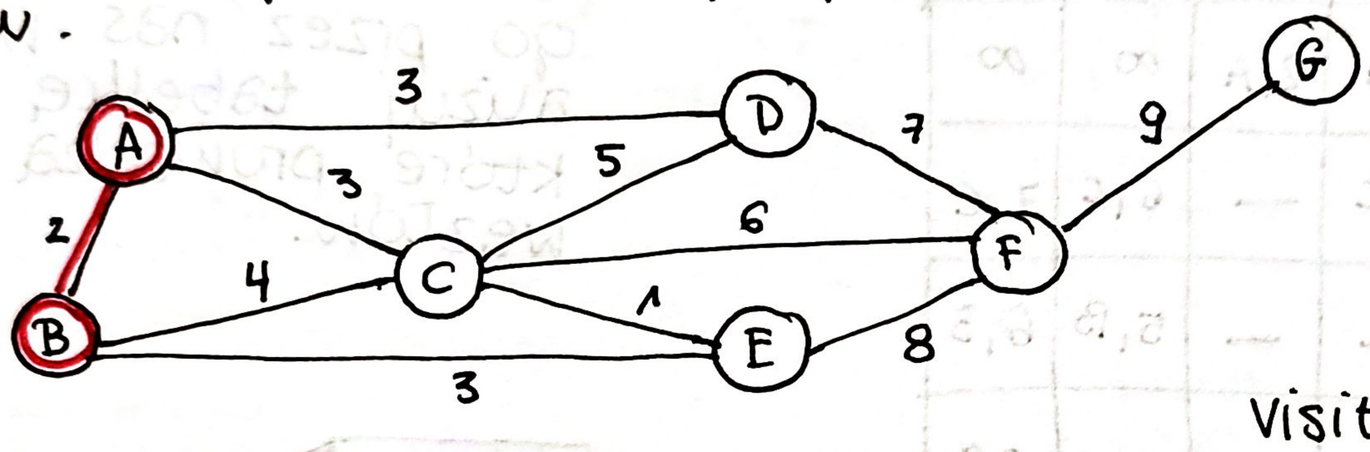


Step 1 Rozpoczynamy algorytm od wybrania węzła początkowego (np. A). Śledzimy odwiedzone węzły

Visited = {A}

Step 2

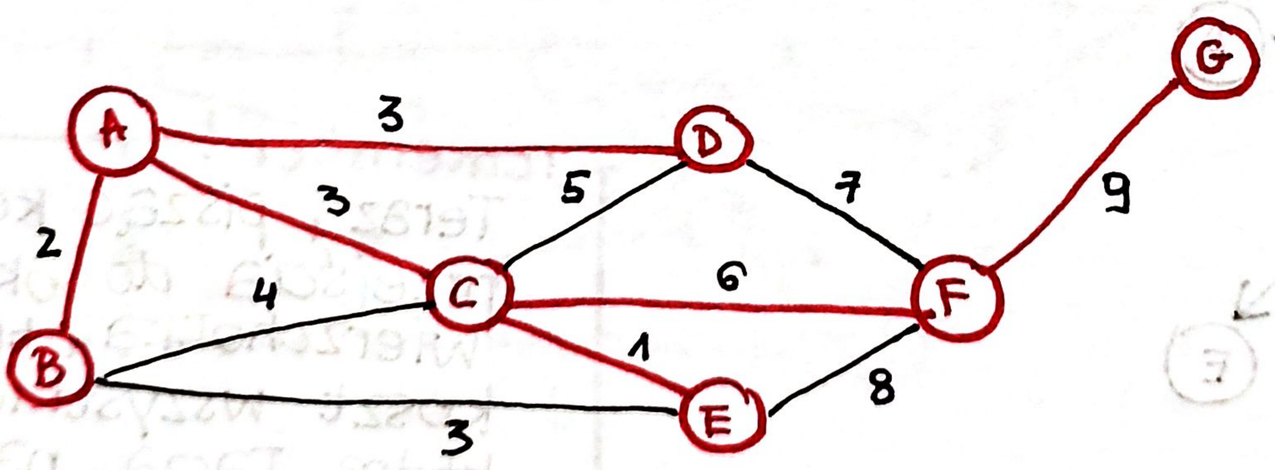
Algorytm Prima to **chciwy algorytm** więc będziemy wybierać brzegi o najmniejszej wadze do nieodwiedzonych węzłów.



Visited = {A,B}

Teraz patrzymy do jakich innych węzłów możemy dotrzeć z węzłów A i B. Kiedy mamy kilka brzegów o tej samej wadze możemy wybrać obojętnie który.

Kontynuuj w ten sposób za każdym razem, wybierając najmniejszą możliwą krawędź, która łączy się z nieodwiedzonym węzłem.



W ten sposób tworzy się MST (minimal spanning tree). Całkowita waga krawędzi to suma wszystkich krawędzi drzewa MST ($2+3+3+6+1+9=24$)

Time complexity: (zależy od struktury danych użytej do algorytmu)

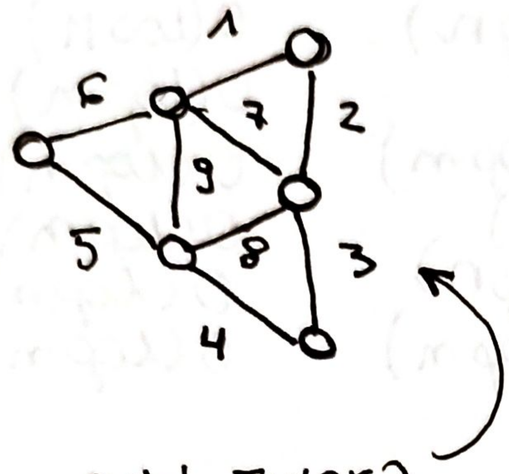
Wyszukiwanie macierzy sąsiedztwa $O(V^2)$

Sterta binarna i lista sąsiedztwa $O(V \log V + E \log V)$

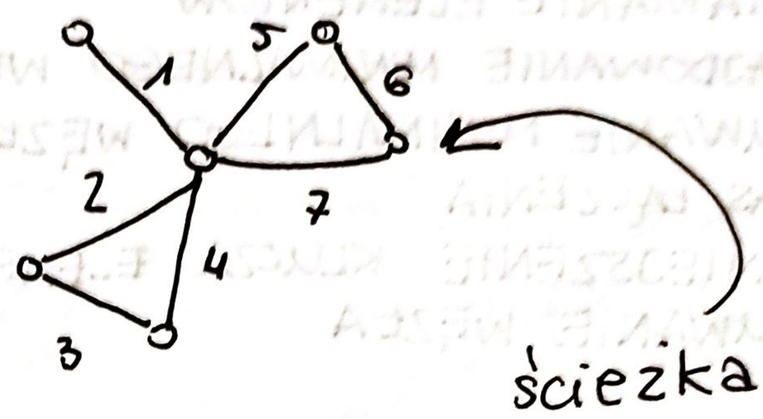
8.5 – cykl eulera

cykl Eulera – jeżeli istnieje ścieżka zamknięta zawierająca każdą krawędź grafu G.

ścieżka Eulera – ścieżka zawierająca każdą krawędź grafu G.



cykl Eulera



ścieżka Eulera

ALGORYTM FLEUREGO

- **Step 1** Wystartuj z wierzchołka o nieparzystym stopniu (jeśli jest)
- **Step 2** Wybierz dowolną krawędź, ale most wybieraj w ostateczności (usuń ją potem)
- **Step 3** Przejdź do kolejnego wierzchołka

→ szuka cyklu Eulera w grafie

Usuwając napotkane krawędzie i przemadrac na stosie napotkane wierzchołki.

cykl Eulera alg. ze stosem

1) wystartuj z dowolnego wierzchołka
2) jeżeli istnieją nie przechodzone dotąd krawędzie incydentne z bieżącym wierzchołkiem:
   - 2.1) wybierz dowolną krawędź
   - 2.2) przejdź do kolejnego wierzchołka i odłóż go na stos
3) w przeciwnym wypadku
   - 3.1) przenieś wierzchołek do rozwiązania i zdejmij go ze stosu
   - 3.2) przejdź do wierzchołka na szczycie stosu

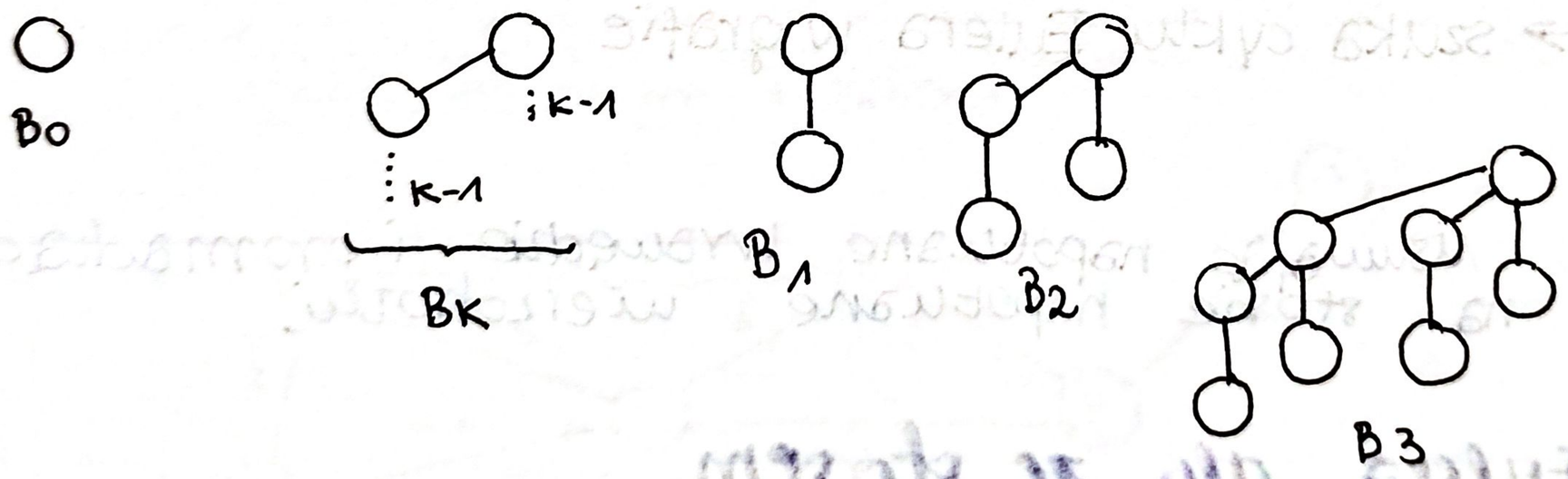
9. KOPCE & DRZEWA CD.

9.1 - koszt operacji

	KOPIEC BINARNY*	KOPIEC DWUMIANOWY*	KOPIEC FIBBONACIEGO**
WSTAWIANIE ELEMENTÓW	$O(\log n)$	$O(\log n)$	$O(1)$
ZNAJDOWANIE MINIMALNEGO WĘZŁA	$O(1)$	$O(\log n)$	$O(1)$
USUWANIE MINIMALNEGO WĘZŁA	$O(\log n)$	$O(\log n)$	$O(\log n)$
CZAS ŁĄCZENIA	$O(n)$	$O(\log n)$	$O(1)$
ZMNIEJSZENIE KLUCZA ELEMENTU	$O(\log n)$	$O(\log n)$	$O(1)$
USUWANIE WĘZŁA	$O(\log n)$	$O(\log n)$	$O(\log n)$

* koszt pesymistyczny
** koszt zamortyzowany

9.2 - drzewa dwumianowe



... i tak dalej

Właściwości

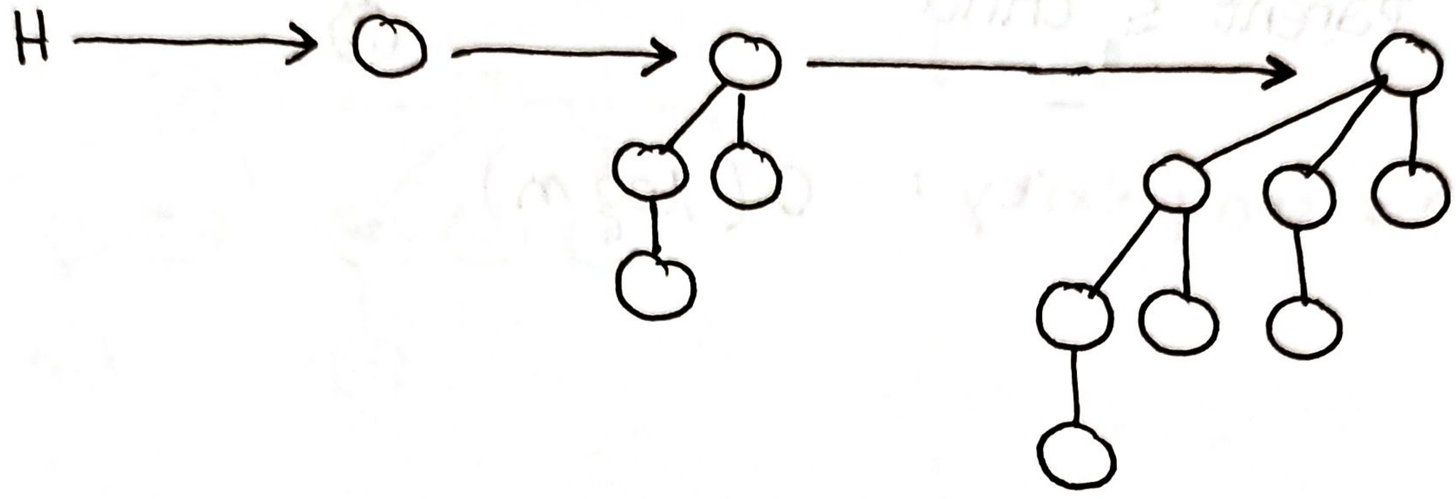
- liczba węzłów = 2^k
- wysokość = k
- na głębokości i znajduje się $\binom{k}{i}$ węzłów
- stopień korzenia jest k i jest największy w całym drzewie

9.3 - kopiec dwumianowy

↳ zawiera kolekcję drzew dwumianowych

→ każde drzewo w H jest uporządkowane kopcowo (klucz w węźle jest nie mniejszy niż klucz ojca)

→ w kopcu dwumianowym może być co najwyżej jedno drzewo danego stopnia



→ reprezentacja poszczególnych drzew w notacji na lewo syn na prawo brat

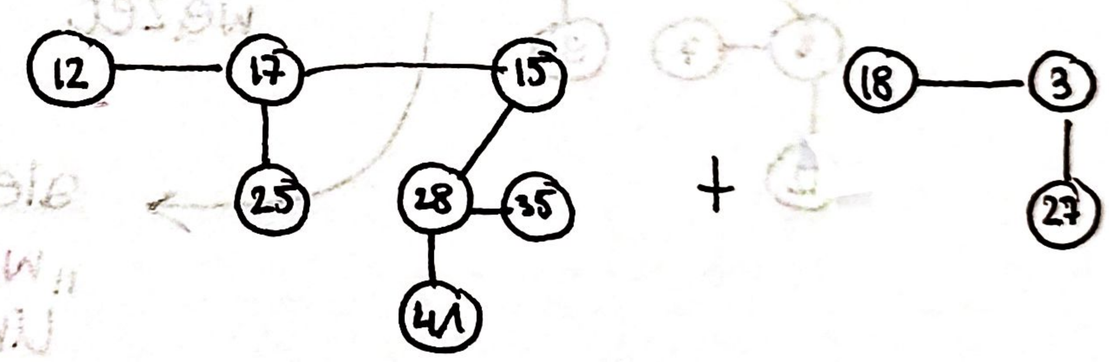
Fun property:

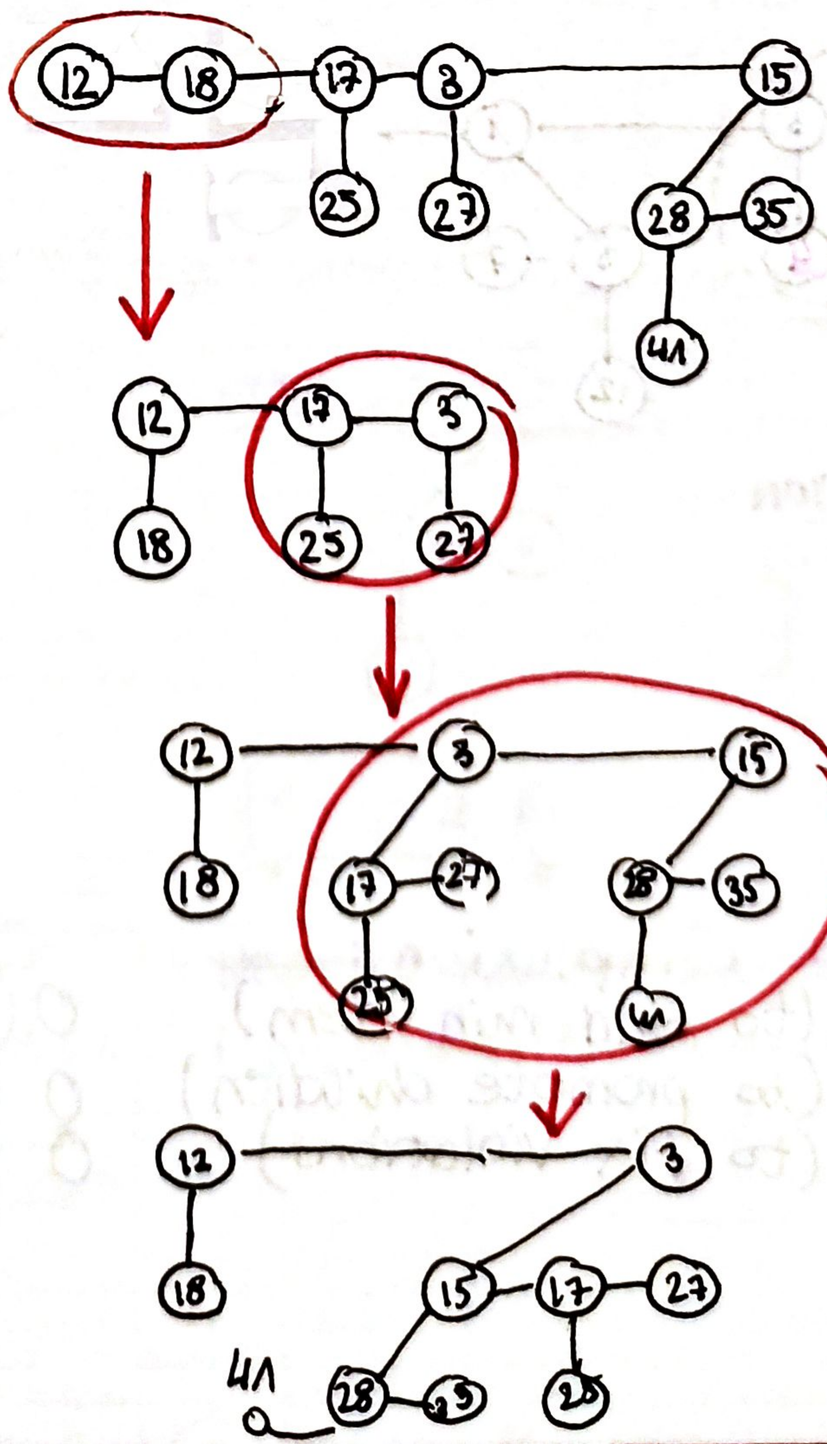
kopiec ma $2^{m+n+\dots}$ elementów, gdzie m, n... to stopnie drzew dwumianowych

! Pamiętaj, że w kopcu nie mogą być drzewa o tym samym stopniu

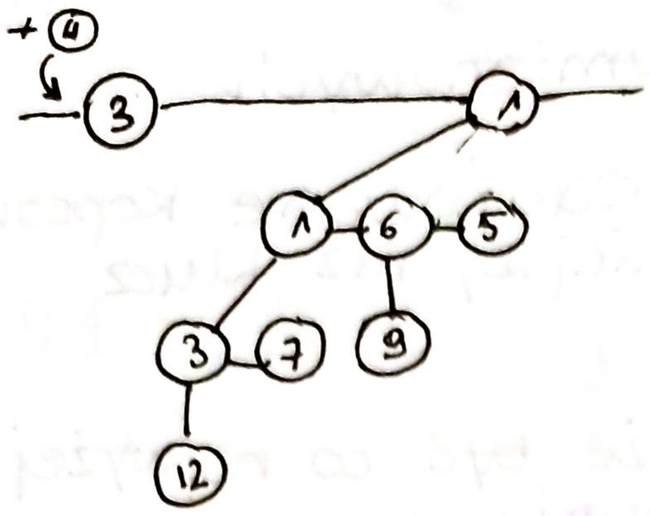
UNION

↳ dodawanie dwóch kopców

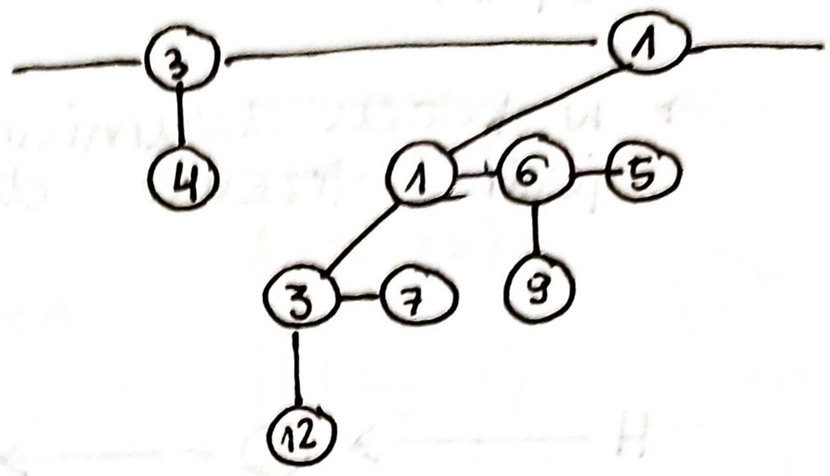




INSERT



drzewo dwumianowe nie pozwala na więcej niż jedno drzewo danego rozmiaru więc złączamy drzewa o tej samej wielkości

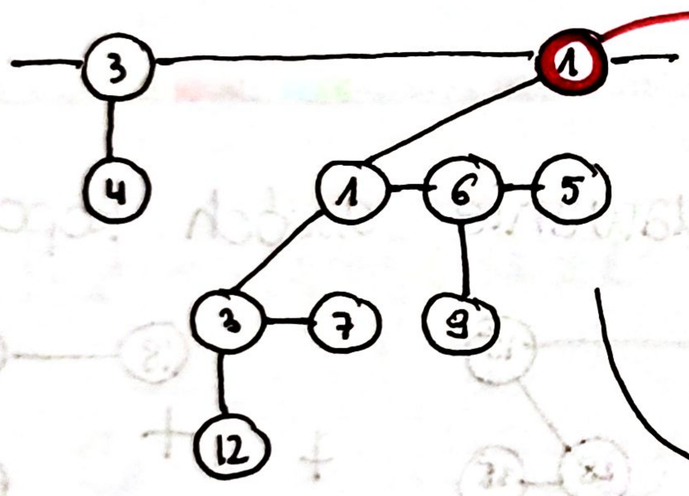


Pamiętaj o właściwości:

Parent ≤ child

Time complexity : $O(\log n)$

EXTRACT MIN



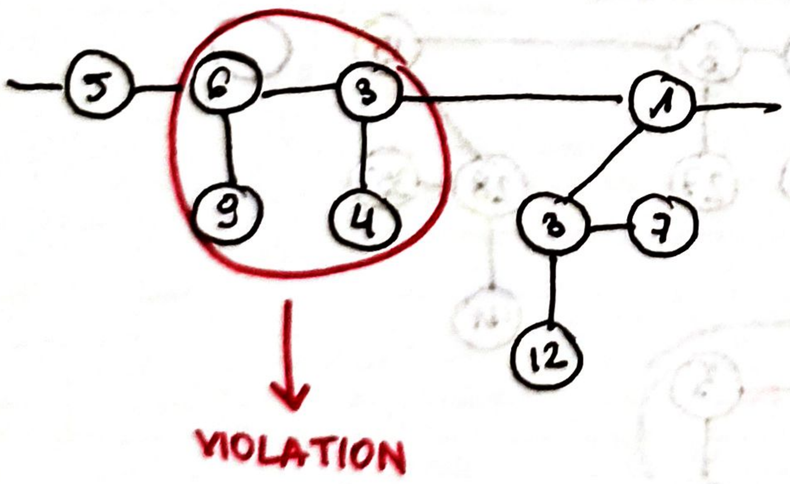
chcemy usunąć ten węzeł

Step 1

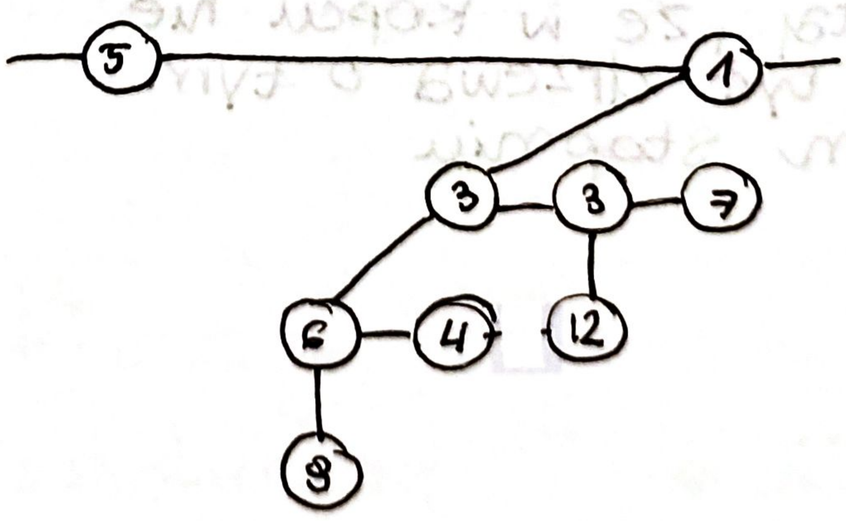
korzeń

Przeskanuj drzewo i znajdź najmniejszy

ale węzeł ma dzieci więc „wznosimy" je na główną unię



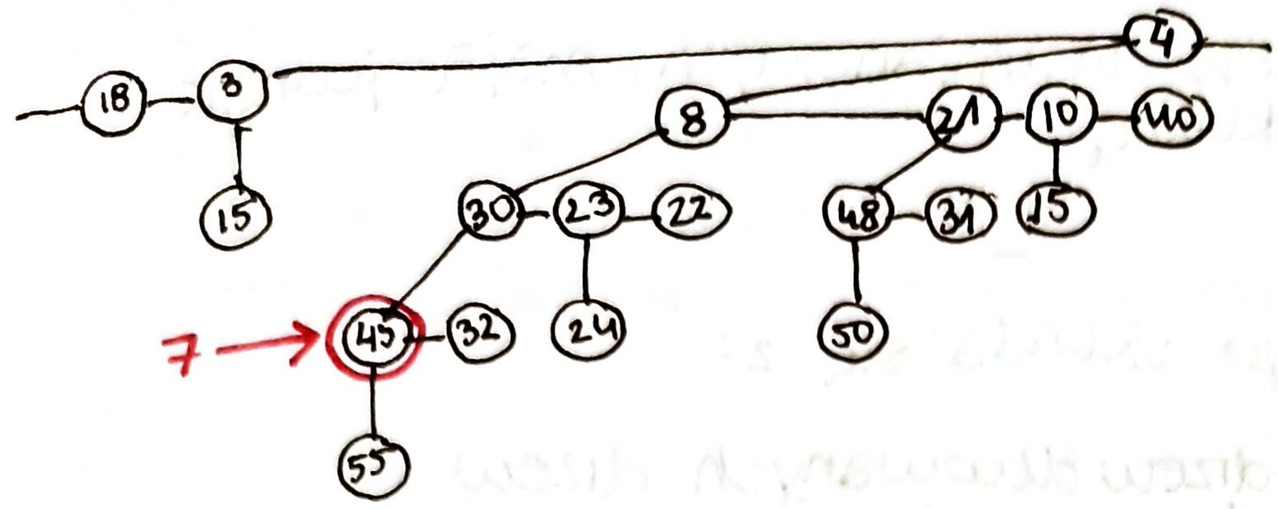
Step 2 Napraw wszystkie naruszenia



Time complexion :

+ (to find min item) $O(\log n)$
+ (to promote children) $O(\log n)$
+ (to fix violations) $O(\log n)$

DECREASEKEY

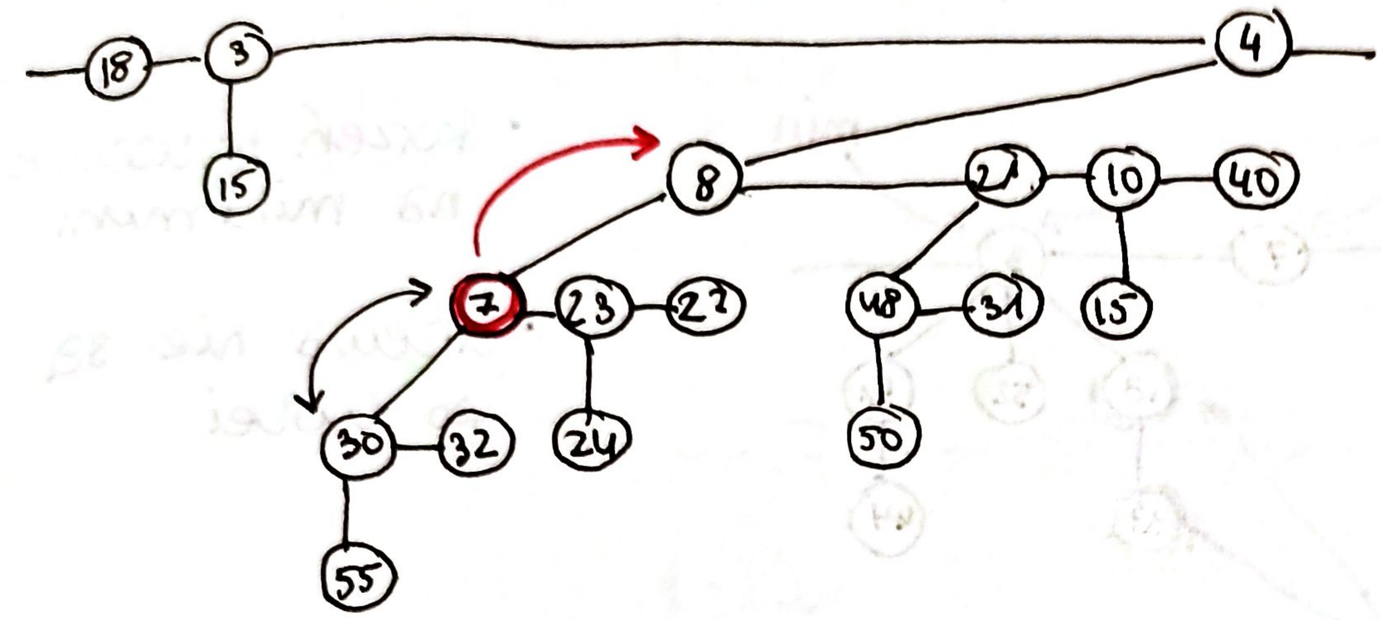


Time complexity:

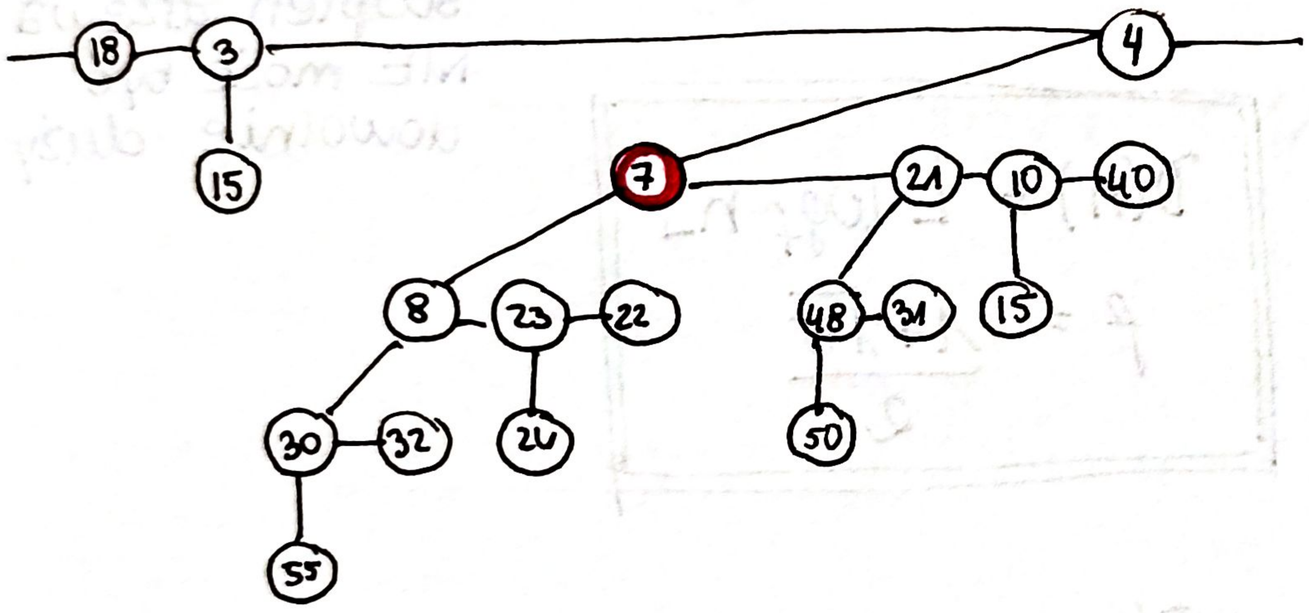
$O(\log n)$

↳ wysokość drzewa

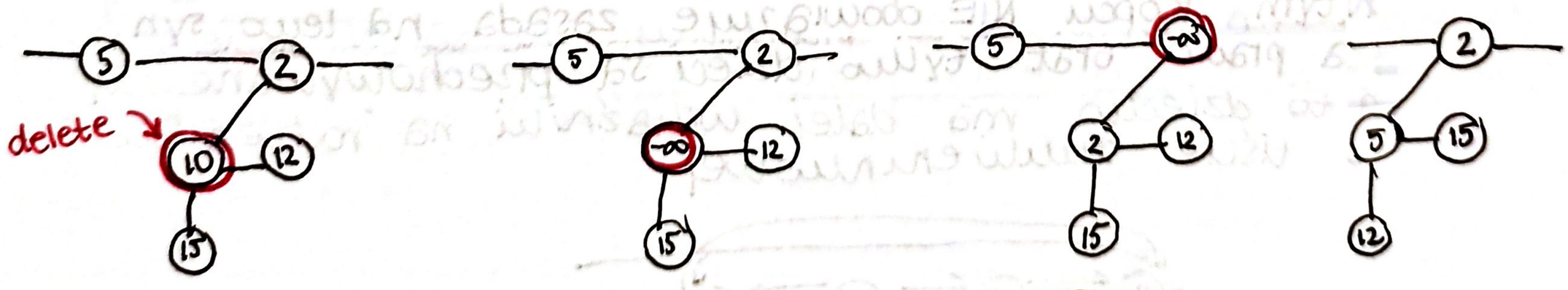
bubble up



bubble up



REMOVENODE



Step 1

Zmień wartość usuwanego węzła na minimalną

Step 2

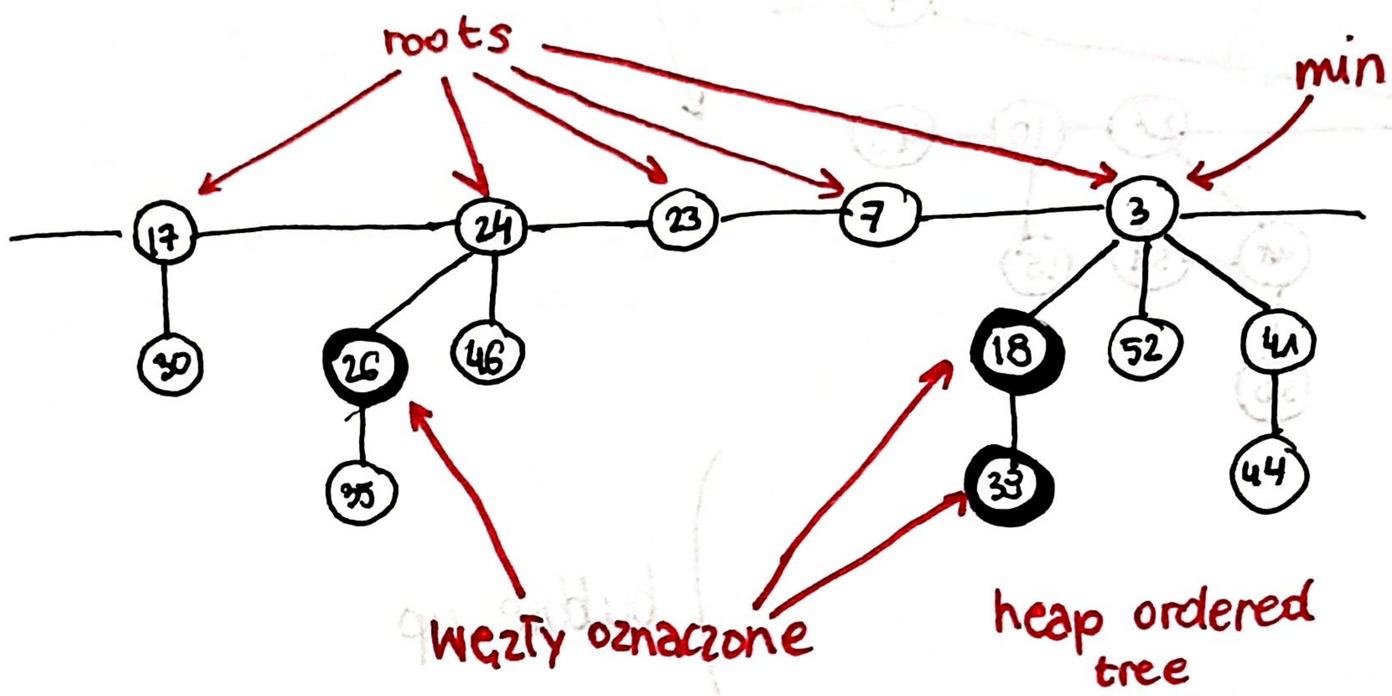
Napraw naruszenia i wykonaj ExtractMin

9.4 - Kopiec fibonacciego

↳ zbliżone do kopców dwumianowych, mają jednak luźniejszą strukturę

Kopiec fibonacciego składa się z:

- zbioru uporządkowanych drzew
- wskaźnik na najmniejszy element
- zbiór oznaczonych węzłów



- korzeń wskazuje na minimum
- drzewa nie są po kolei

stopień drzewa NIE może być dowolnie duży

Max degree in the tree

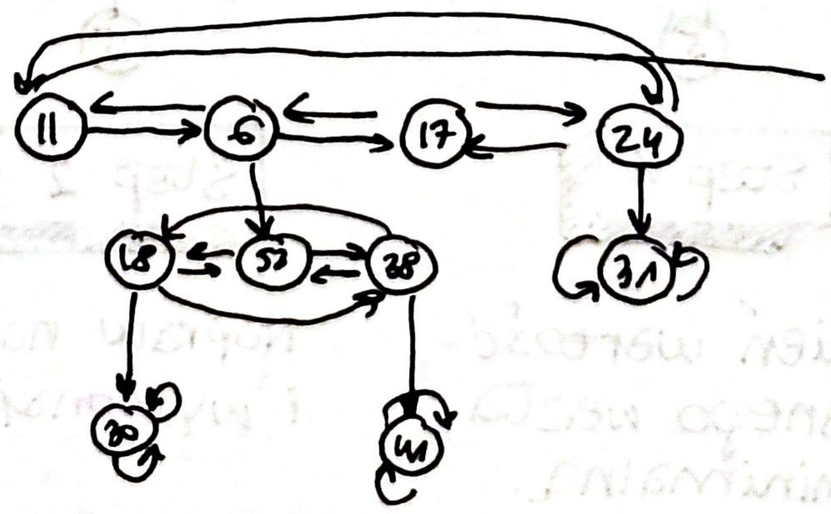
$$D(n) = \lfloor \log_f n \rfloor$$

$$f = \frac{1+\sqrt{5}}{2}$$

Notacja kopca

n	→	numer węzłów w kopcu
rank (x)	→	numer dzieci węzła x
rank (H)	→	maksymalna ranga węzła w kopcu
trees (H)	→	liczba drzew w kopcu H
marks (H)	→	ilość oznaczonych węzłów w kopcu H

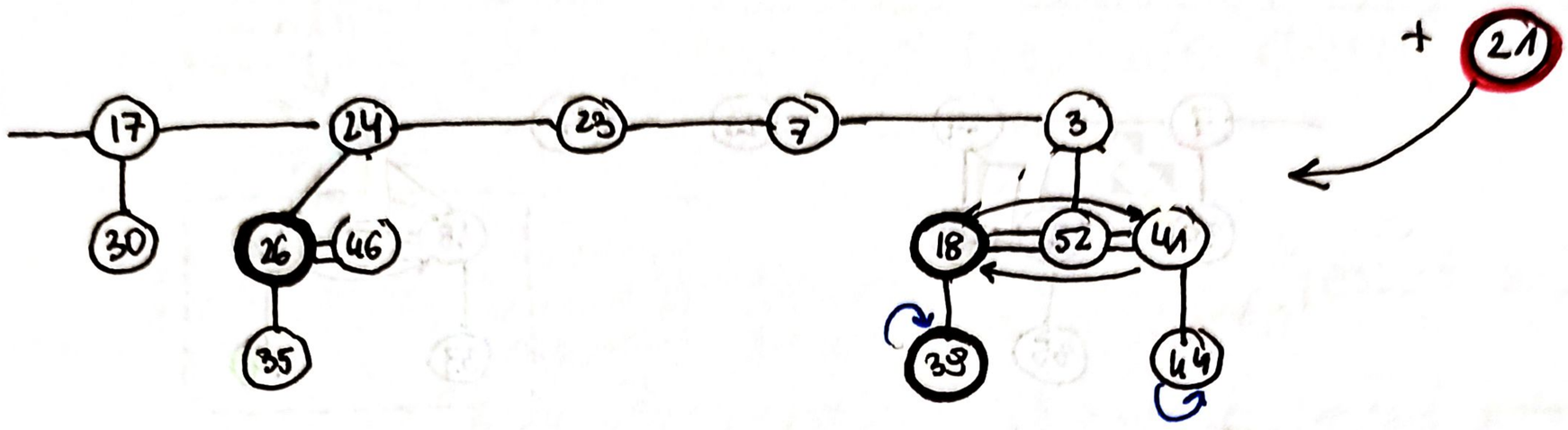
W tym kopcu dzieci przechowywane są w liście... (lista rodzeństwa). Jest wsk do jednego z dzieci, a to dziecko ma dalej wskaźniki na rodzeństwo w liście dwukierunkowej

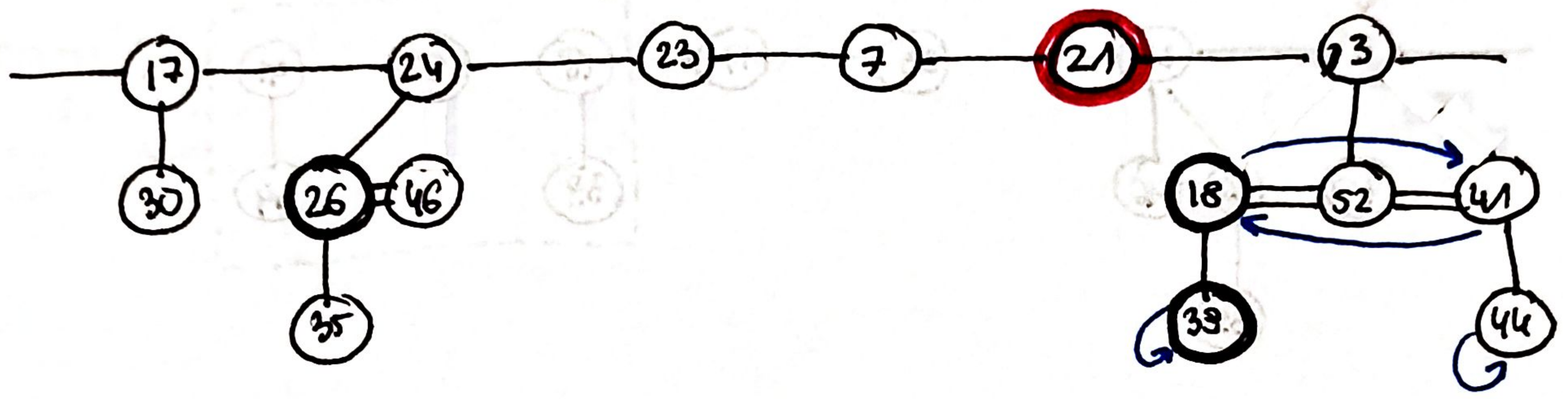


INSERT

Step 1 Stwórz nowe pojedyncze drzewo (singleton tree)

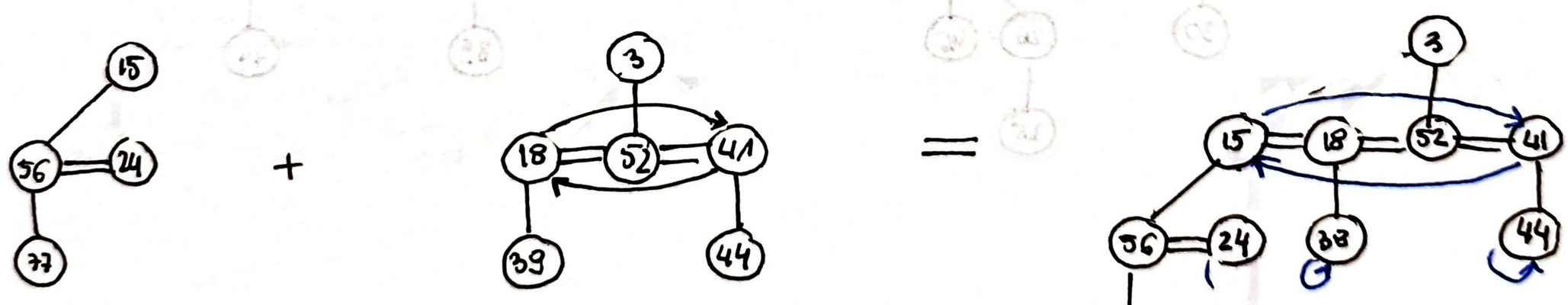
Step 2 Dodaj do listy korzeni, i zaktualizuj min pointer (jeżeli to konieczne)





Time complexity : $O(1)$

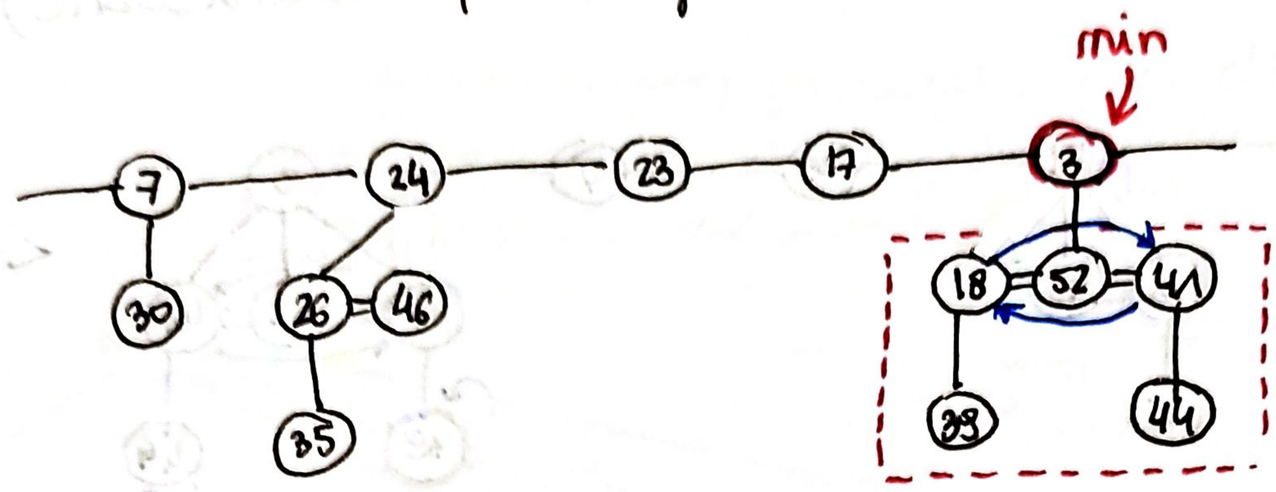
LINKING OPERATION

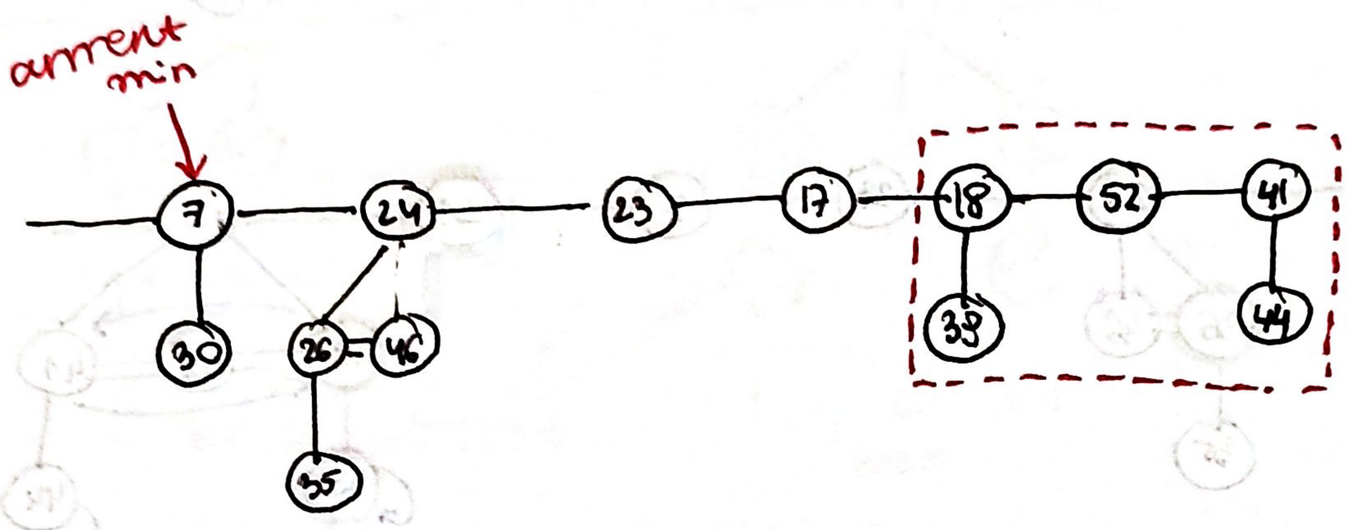


DELETEMIN

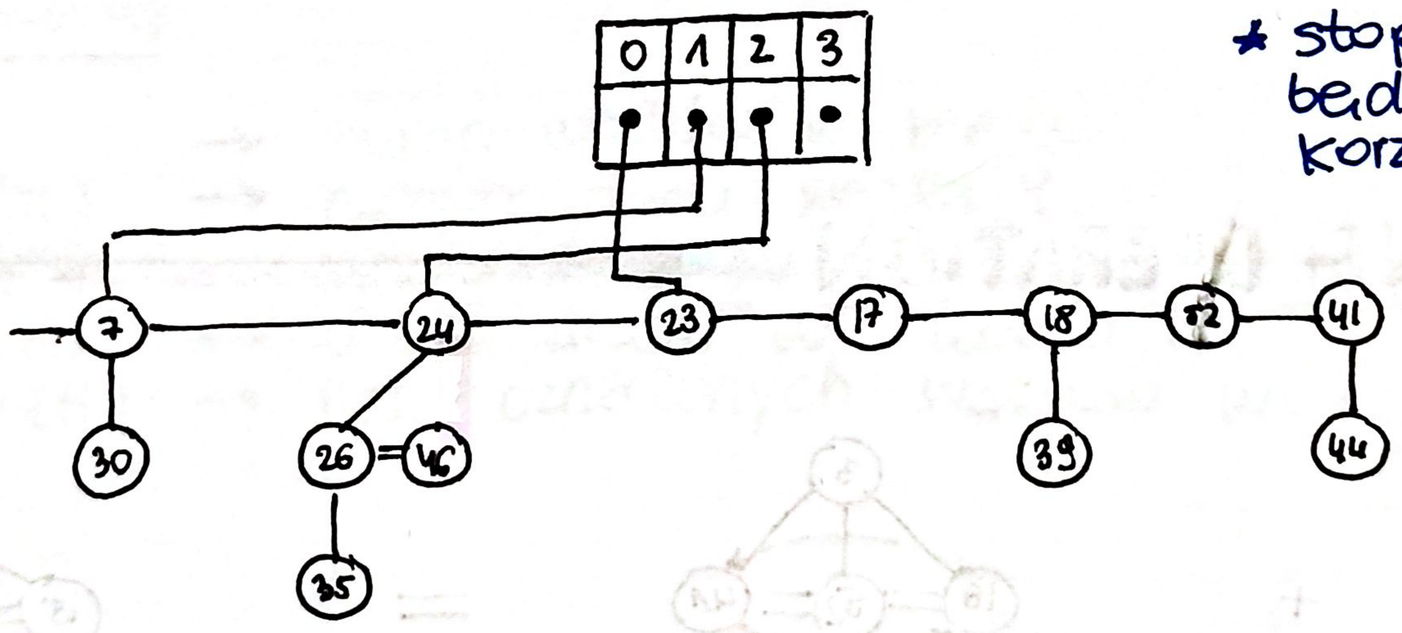
Step 1 Usuń minimalny węzeł, wepni dzieci węzła do listy korzeni i zaktualizuj min pointer

Step 2 złącz drzewa tak aby korzenie nie miały takiej samej rangi

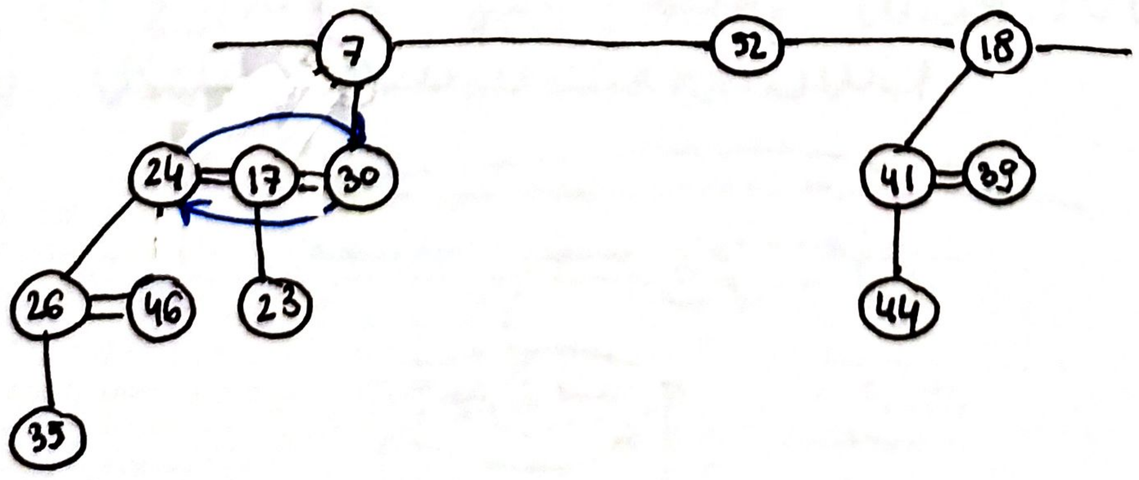




★ fajny patent na merge drzew tego samego stopnia to wstawianie do tablicy, jak są dwa drzewa do tego samego miejsca w tablicy → merge (usuń wszystko z tablicy i rozpocznij znowu analizę aż przejdziesz przez całą listę korzeni)



★ stopniem drzewa będzie liczba dzieci korzenia



Time complexity: $O(\text{rank}(H))$

DECREASE KEY

- jeżeli zmniejszenie wartości węzła nie powoduje naruszeń → po prostu zmniejsz wartość
- inaczej odetnij drzewo odchodzące od zmniejszanego węzła
- żeby zachować własności kopca i czas trwania wykonywanych operacji (żeby nie doszło do tego, że DeleteMin zajmie $\Omega(n)$)

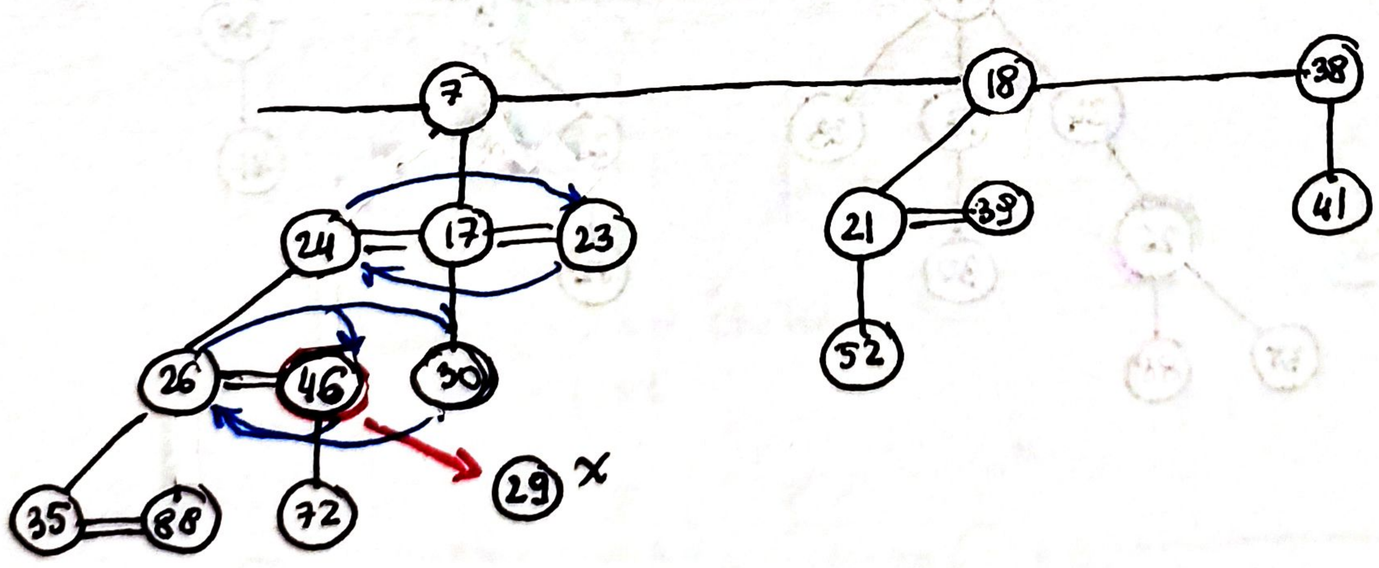
Rule 1 Strać 1 dziecko, jesteś węzłem oznaczonym LOSER

Rule 2 Strać 2 dzieci → jesteś wzniesiony do listy korzeni

pozwala nam to zapewnić pewną regularność drzew

Case 1 [HEAP-ORDER NIENARUSZONA]

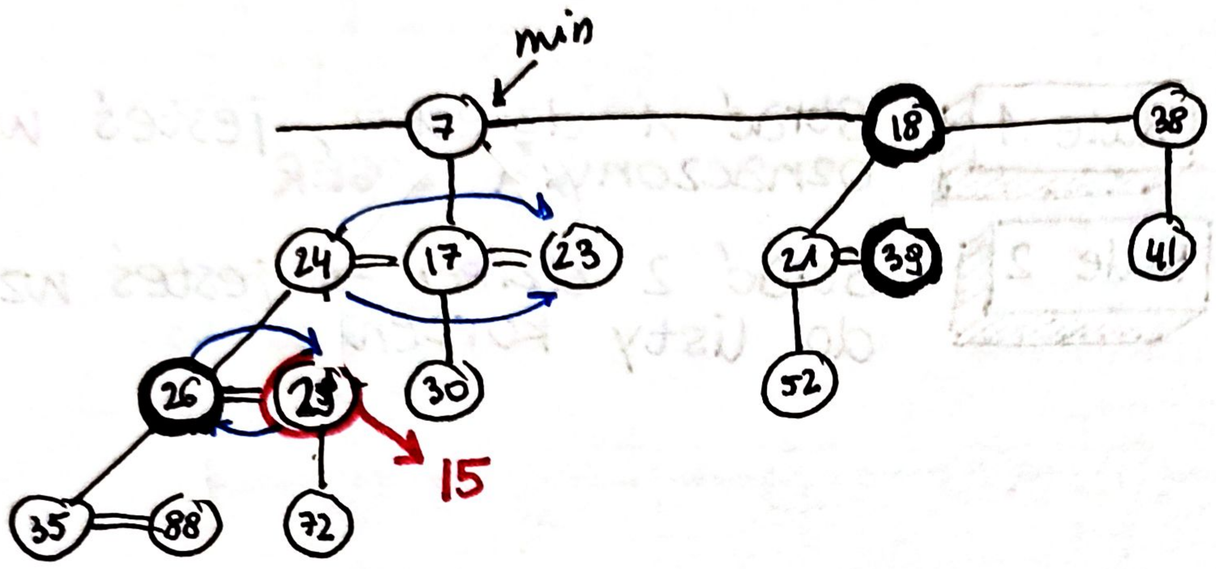
1) zmniejsz wartość węzła x
2) jeżeli to konieczne - zmień min pointer

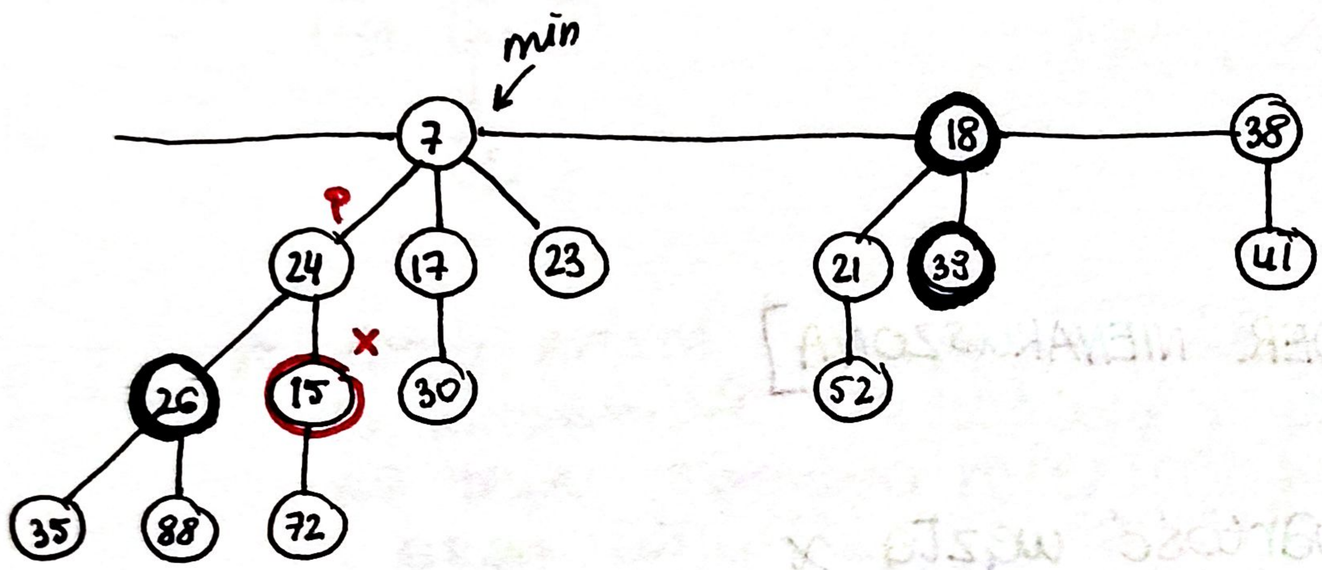


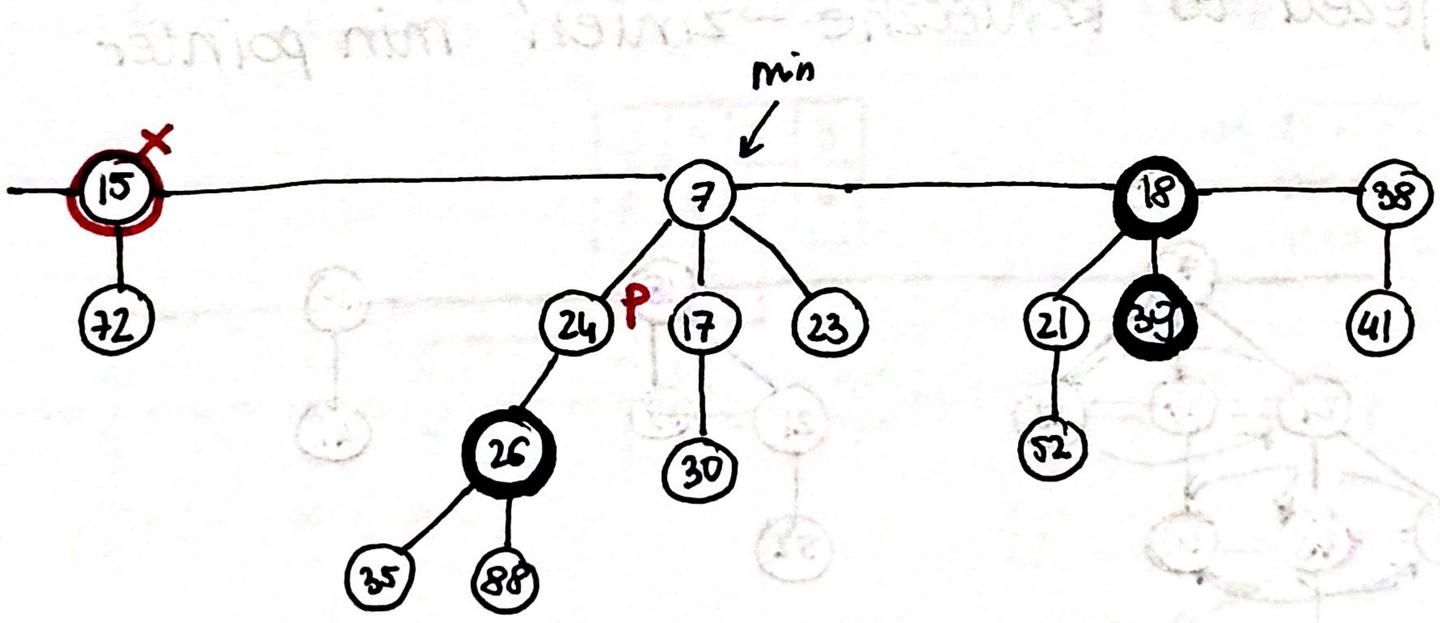
Case 2a

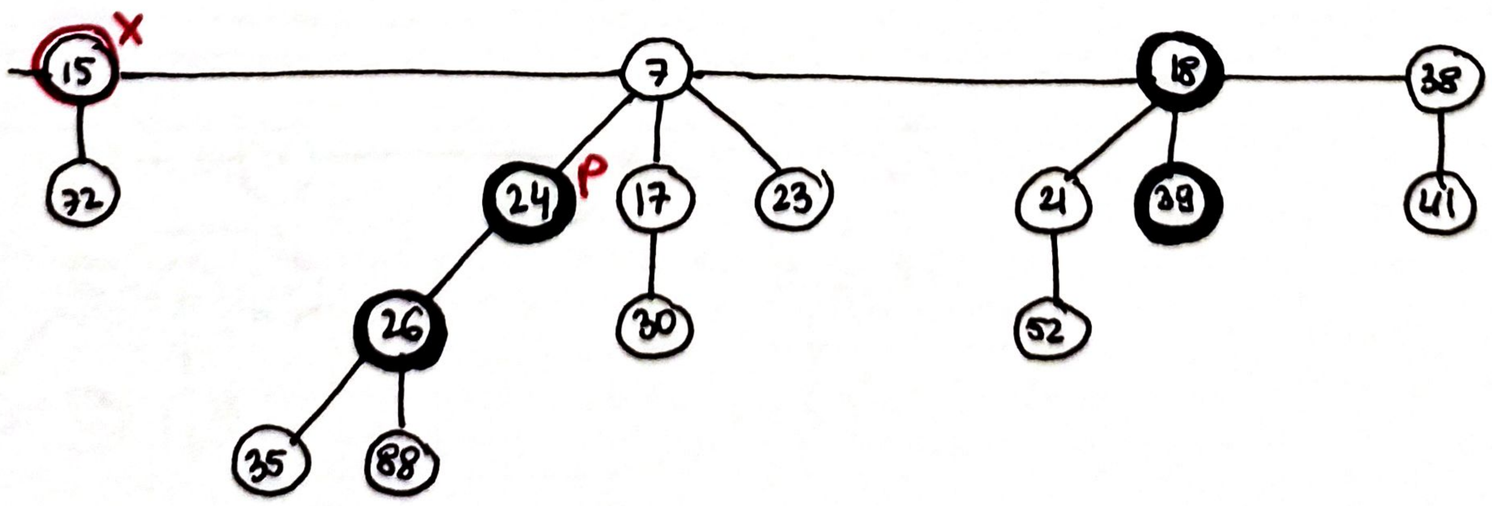
1) zmniejsz wartość węzła
2) utnij drzewo zakorzenione w x i dodaj je do listy korzeni, jeżeli jest oznaczone to odznacz

3) Jeżeli rodzic $x \to p$ jest nieoznaczony (nie stracił jeszcze żadnego dziecka) oznacz je i
Jeżeli rodzic $x \to p$ jest oznaczony (stracił już dziecko) wtop go w listę korzeni



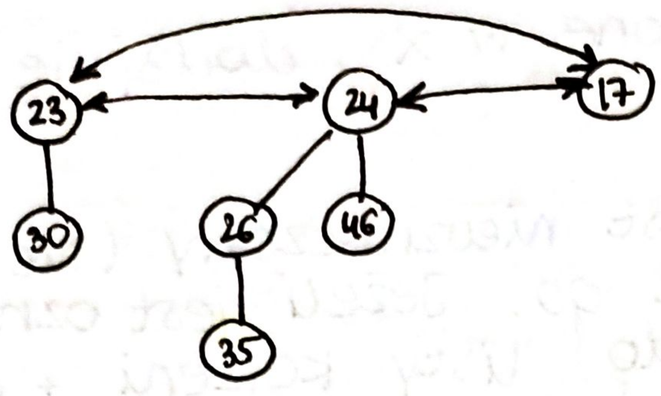


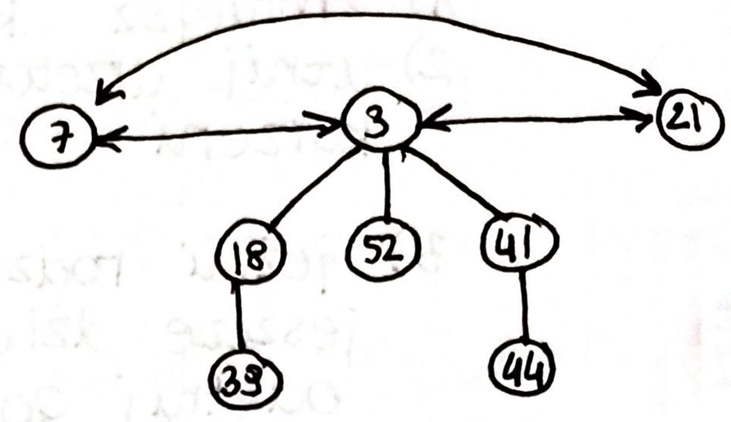


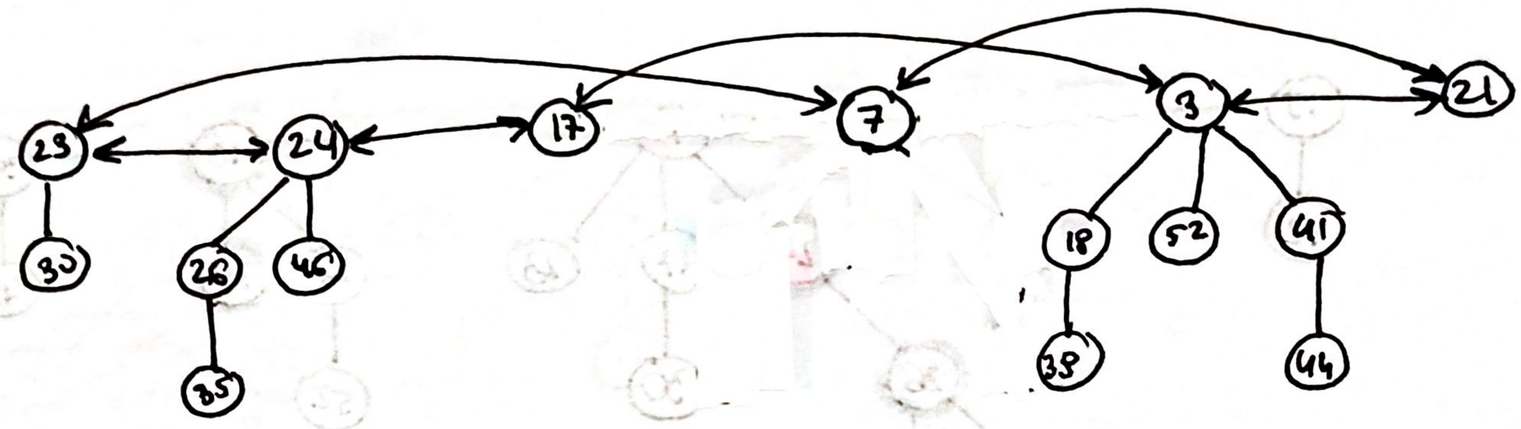


UNION

↳ combine two fibonacci heaps







+ update minimum root pointer

10. ZASTOSOWANIE ALGORYTMÓW W GRACH

10.1 - drzewo stanów gry

n-osobowa gra:

- drzewo topologiczne z wyróżnionym wierzchołkiem A nazywanym punktem początkowym
- funkcja wypłaty, która przyporządkowuje każdemu wierzchołkowi końcowemu pewien wektor n-wymiarowy
- rozbicie zbioru niekońcowych wierzchołków drzewa na n+1 zbiorów S0, S1, ... Sn, zwanymi zbiorami posunięć graczy
- funkcja za pomocą której możemy określić czy dany wierzchołek topologicznego drzewa gry jest wierzchołkiem końcowym

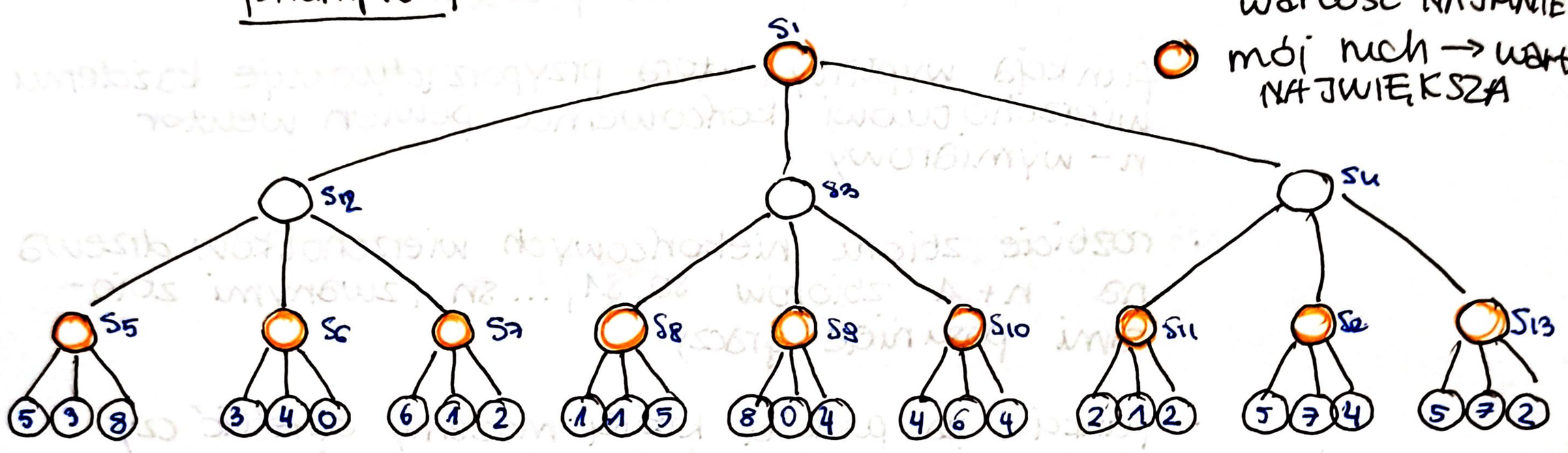
→ mówimy, że gra jest skończona, jeśli jej dendryt zawiera tylko skończoną ilość wierzchołków.

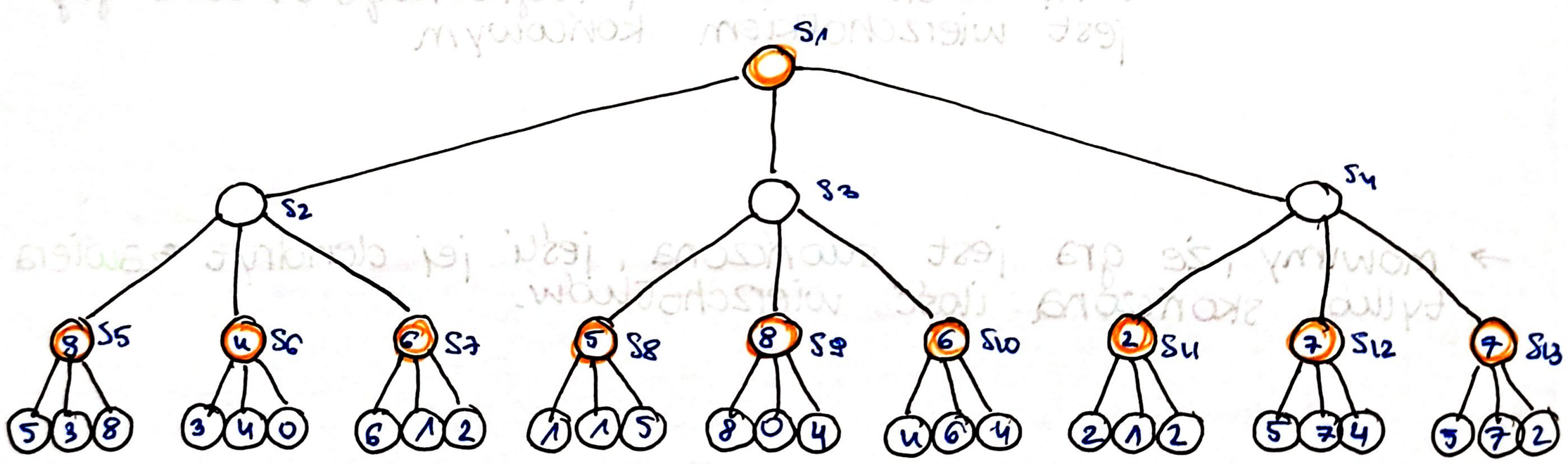
10.2 - minimax

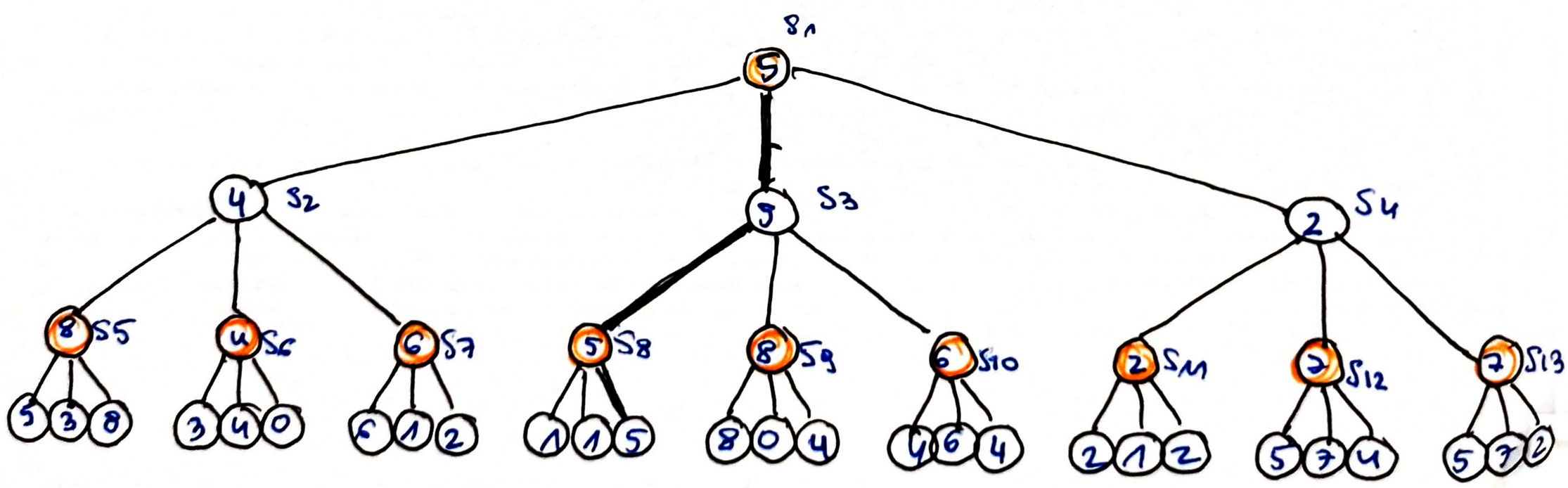
1. Określ d- głębokość przeszukiwania
2. Dla węzłów na głębokości d oceniamy w kategoriach dobrej dla programu
3. Dla węzłów wcześniejszych odpowiednio wybierz
   - dla gracza wybierz najlepszą sytuację
   - dla przeciwnika wybieraj te, które mają najgorszy skutek dla gracza

Example:

legenda
- ○ ruch przeciwnika → wartość NAJMNIEJSZA
- ● mój ruch → wartość NAJWIĘKSZA







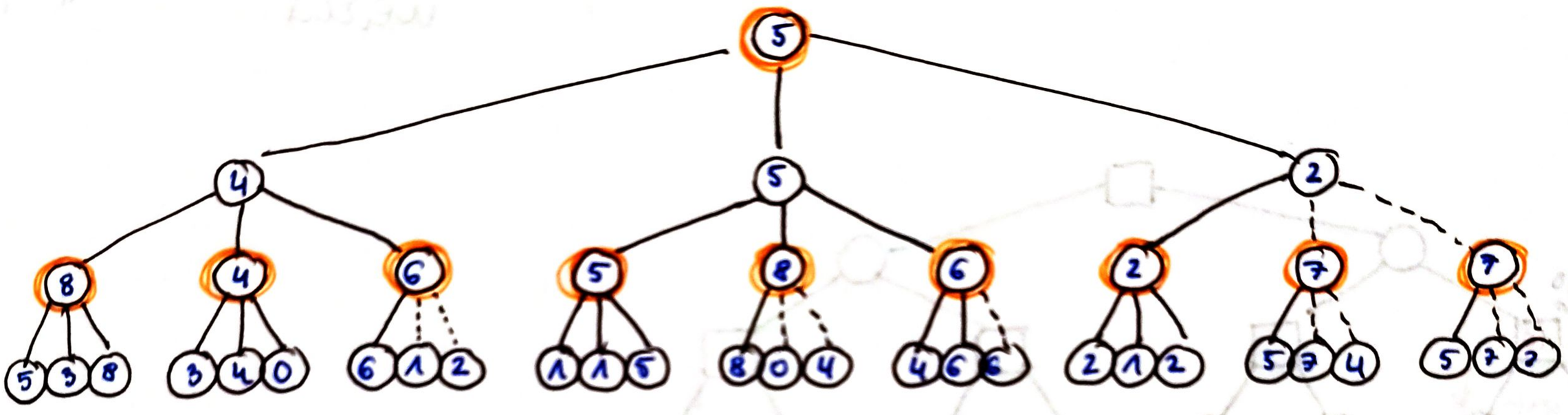
optymalna rozgrywka

10.3 alpha-beta pruning

1. α, β - reprezentują minimalny wynik (of the maximizing) i maksymalny wynik (of the minimizing player)

2. Początkowo $\alpha = -\infty$ i $\beta = \infty$ (obydwoje z graczy zaczynają z możliwie najgorszym wynikiem)

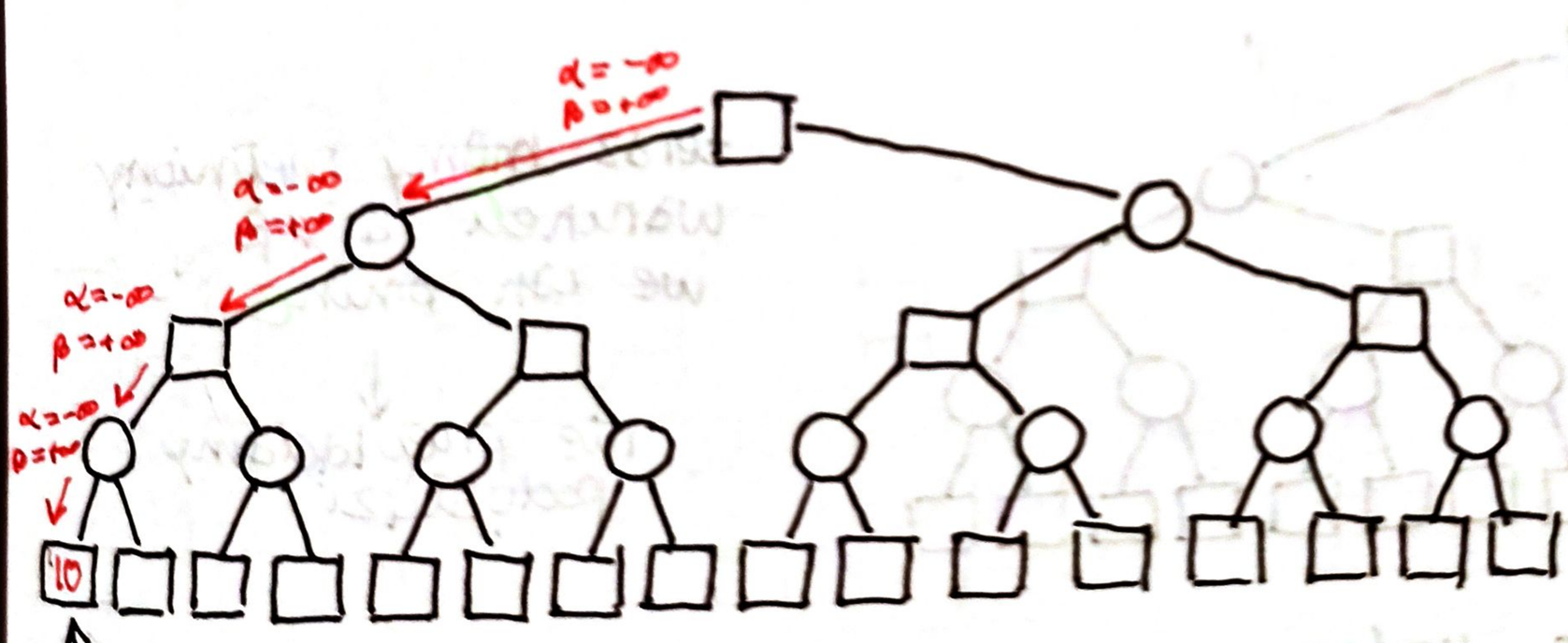
3. Ilekroć maksymalny wynik, jaki zapewnia gracz minimalizujący (β) staje się mniejszy niż minimalny wynik gracza maksymalizującego (α), gracz maksymalizujący ($\beta \leq \alpha$) nie musi brać pod uwagę potomków tego węzła, ponieważ nigdy nie zostaną osiągnięci w tej grze.



Example:
(step by step)

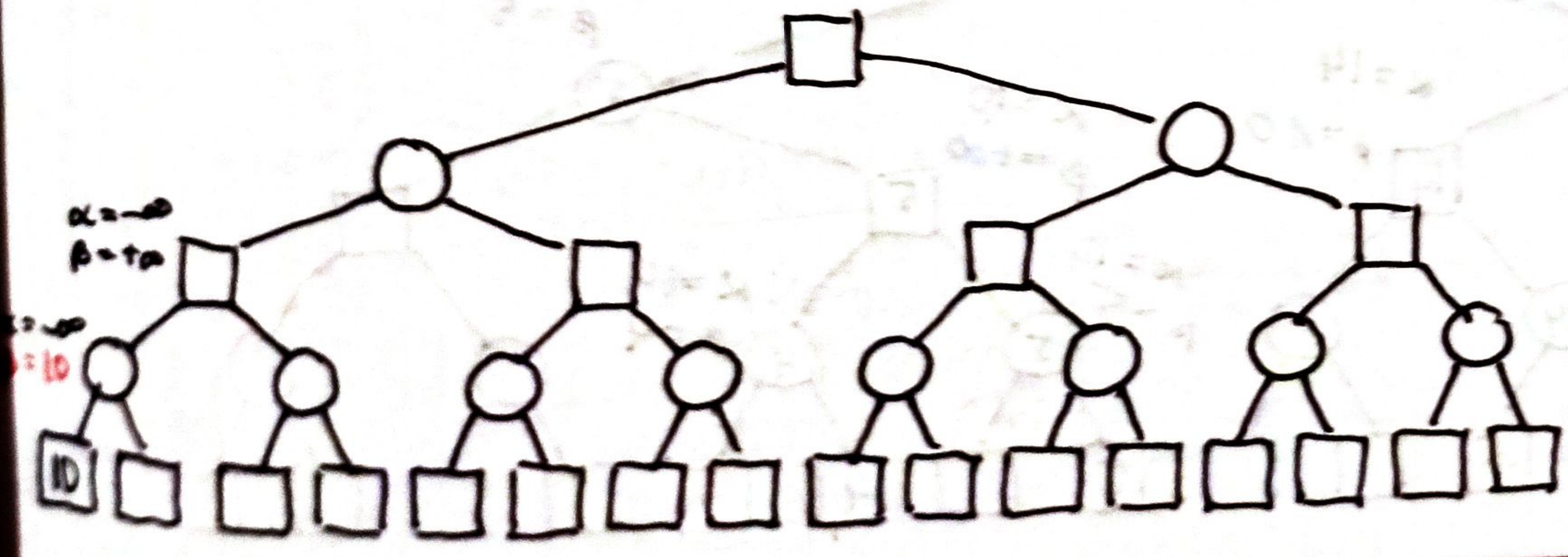
general condition:
if $\alpha \geq \beta$ we can prune

□ MAXIMIZING PLAYER
○ MINIMIZING PLAYER



Inicjujemy wartości dla $\alpha = -\infty$, $\beta = +\infty$ na samym początku

kiedy dochodzimy do końcowego wierzchołka obliczamy wartość heurystyczną

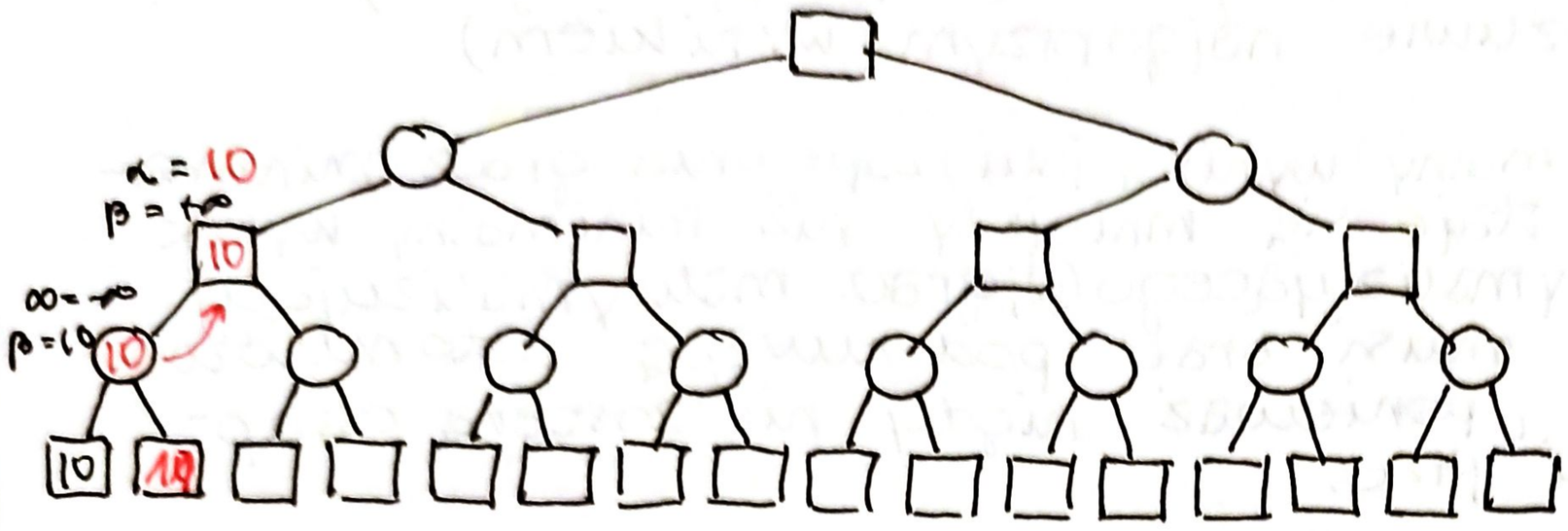


zmieniamy wartość β na 10. Ponieważ wiemy już że beta będzie ≤ 10

dlatego, że teraz $\alpha \not\geq \beta$ we can't prune

important

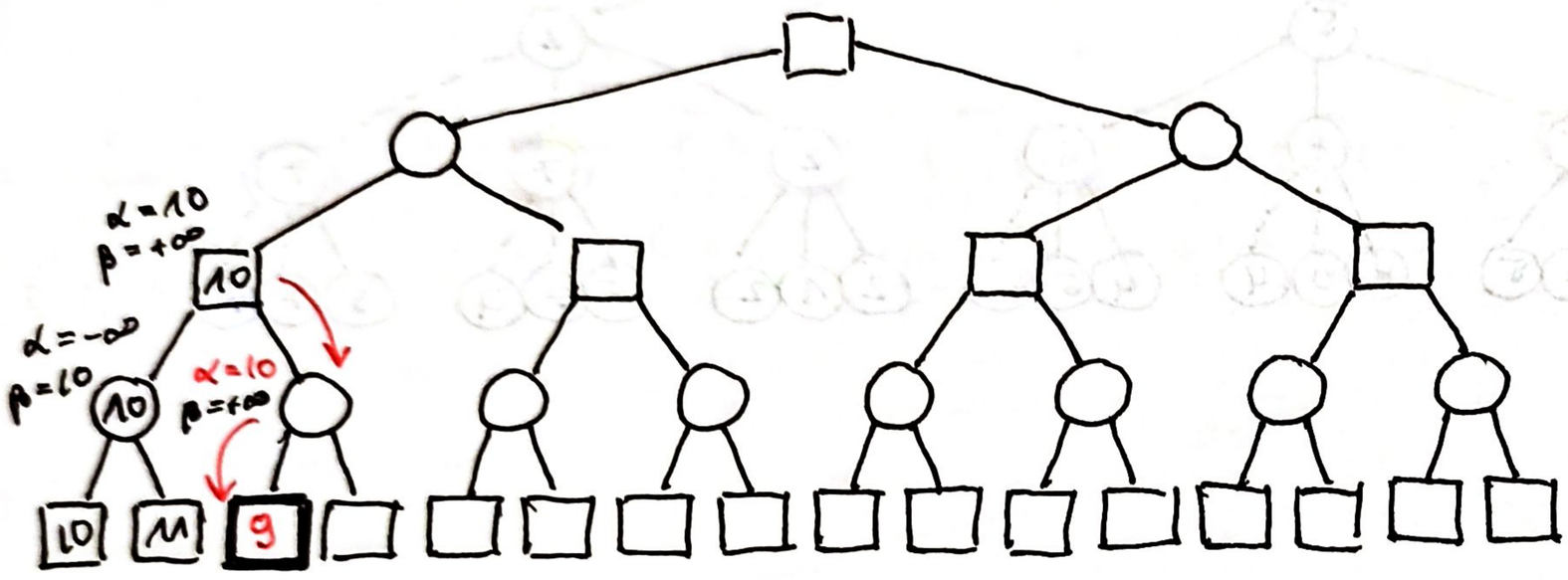
keep in mind that the min nodes (circles) can only modify the beta values and max nodes (squares) can only modify alpha values.



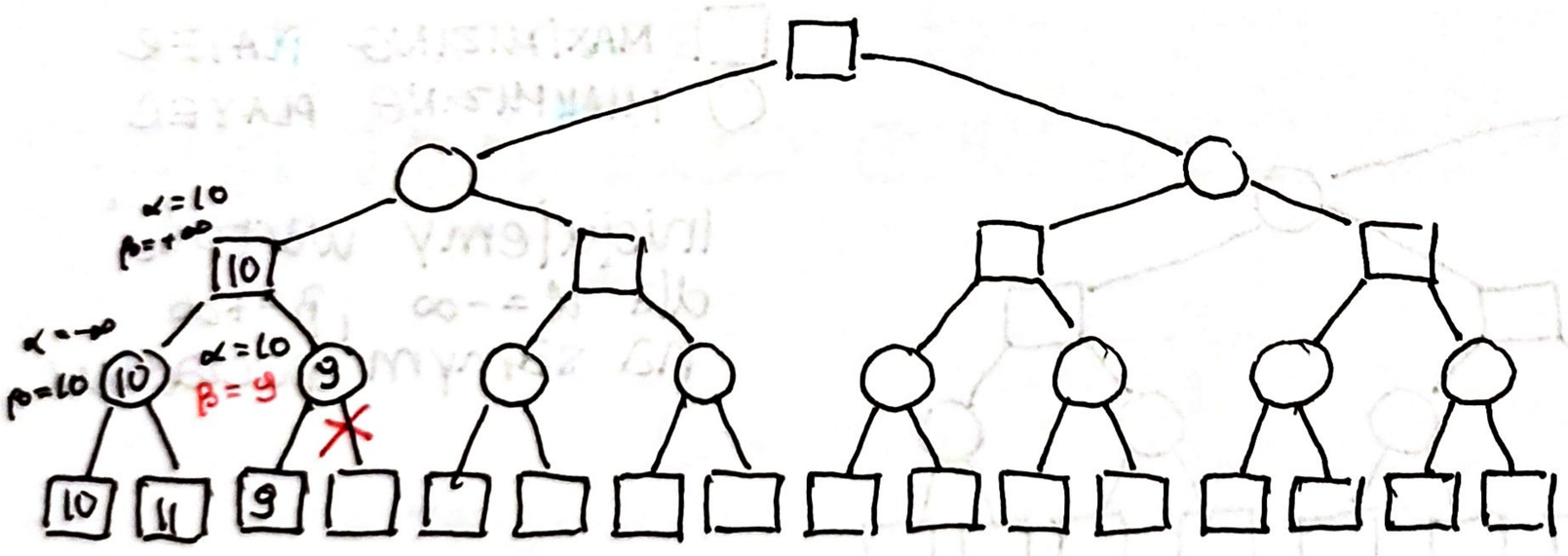
11 > 10 więc wartość β pozostaje 10

dlatego, że znowu α ≥ β we don't prune

lecimy do następnego węzła

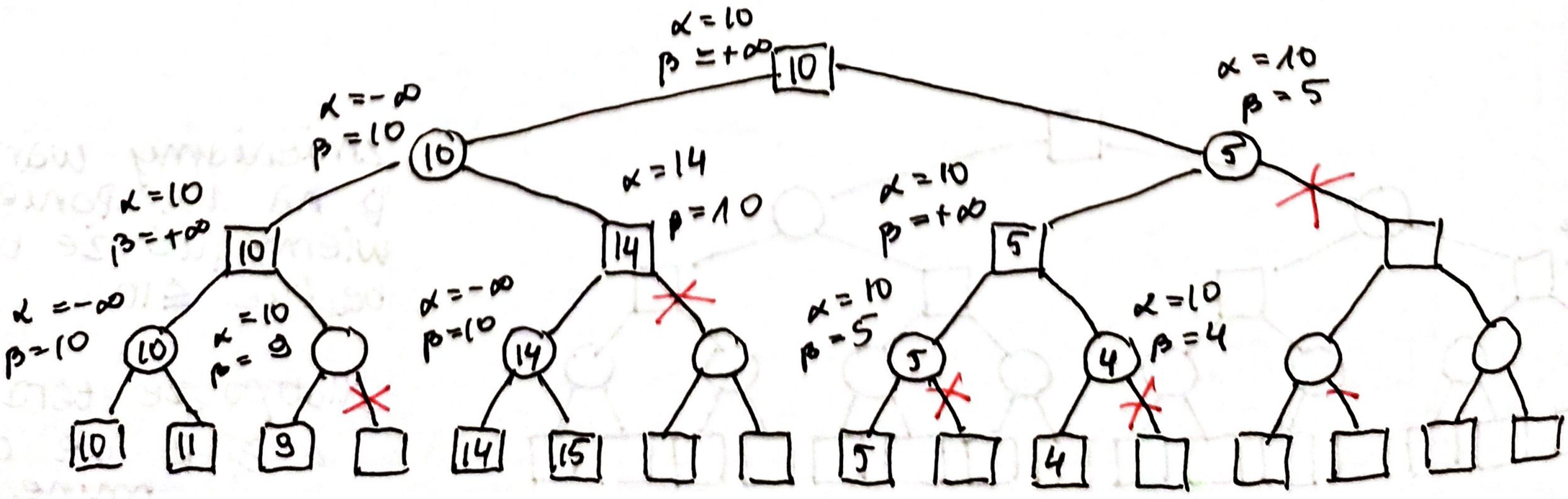


przypisujemy β wartość 9 (teraz β musi być ≤ 9



teraz mamy spełniony warunek α ≥ β we can prune

↓

nie przeglądamy podgałęzi

EFEKT KOŃCOWY ALGORYTMU