

## SESJA – TERMIN I

1. Omów i napisz w C++ algorytm **HEAP SORT**:

a) Procedura **przesiej(int t[], int i, int n)**,

```
void przesiej(double t[], int n, int i) {
    int k;
    double x = t[i]; // przesiewany element

    while(true) {
        k = 2 * i + 2; // prawy syn x

        if((k < n && t[k] > x && t[k] > t[k-1])
        || (-k < n && t[k] > x)) {
            t[i] = t[k];
        } else {
            t[i] = x;
            return;
        }
    }
}
```

b) Procedura **void buduj\_kopiec(int t[], int n)**,

```
void build_heap(double t[], int n) {
    for(int i = n/2; i >= 0; i--) {
        przesiej(t, n, i);
    }
}
```

c) Procedura **heap\_sort(int t[], int n)**,

```
void heap_sort(double t[], int n) {
    build_heap(t, n);

    while(--n) {
        std::swap(t[0], t[n]);
        przesiej(t, n, 0);
    }
}
```

d) Jaka jest złożoność każdej z powyższych procedur?

przesiej:  $O(n)$

buduj\_kopiec:  $O(\log n)$

heap\_sort:  $O(n \log n)$  |  $O(n^2)$

e) Uzasadnij swoje odpowiedzi z punktu d).

przesiej - dla każdego kroku potrzebna jest jedna pętla

*buduj\_kopiec - 1 węzeł ma wysokość  $\log_2(1)+1=1$ , 2 węzeł ma wysokość  $\log_2(2)+1 = 2$ , 4 zaś  $\log_2(4)+1 = 3$  itd., n węzeł ma wysokość  $\log_2(n)+1$ , co po dodaniu węzłów do drzewa sprawia, że średnia złożoność rośnie logarytmicznie*

**heap\_sort -**

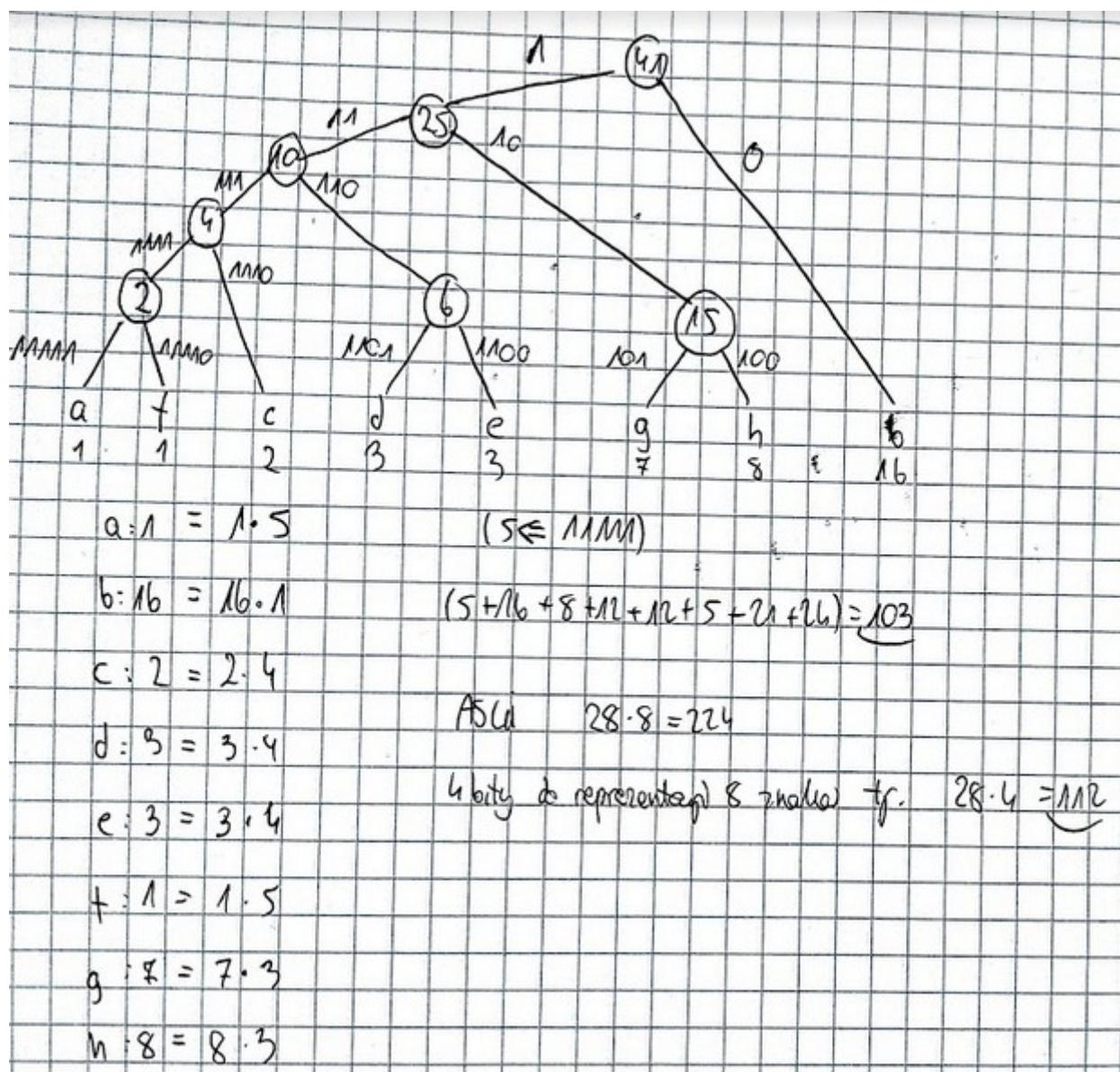
2.

a) Co to jest kod prefiksowy? Czy kody stałej długości są kodami prefiksowymi?

Jednoznacznie dekodowalny (możliwy do zapisu w postaci drzewa binarnego dla kodów dwójkowych) kod, w którym żadne ze słów nie jest przedrostkiem innego słowa.  
Kody stałej długości są kodami prefiksowymi.

b) Zasymuluj działanie algorytmu Huffmana dla następujących ilości liter w tekście:

**a:1 b:16 c:2 d:3 e:3 f:1 g:7 h:8.** Wypisz otrzymane kody.



- c) Oblicz o ile bitów zaoszczędzono korzystając z kodów Huffmana zamiast kodów stałej długości. Zaoszczędzono 9 bitów.
- 3.
- a) Jakie informacje zapamiętane są w węźle drzewa dwumianowego?
- *Lista dzieci*
  - *Lista kluczy*
  - *Ilość kluczy*
  - *Czy jest liściem (boolean)*
- b) Narysuj kopce dwumianowe o 3 i 7 węzłach i wykonaj na nich operację UNION.
- c) Dla powstałego kopca wykonaj GETMAX (załóż, że największy klucz jest w ostatnim drzewie).

4. Jaka jest średnia i pesymistyczna złożoność wyszukiwania klucza najlepszym znanim Ci algorytmem w **n-elementowej**:

- a) tablicy średnia:  $O(n)$  pesymistyczna:  $O(n)$
- b) tablicy posortowanej średnia:  $O(\log n)$  pesymistyczna:  $O(\log n)$
- c) tablicy z haszowaniem średnia:  $O(1)$  pesymistyczna:  $O(n)$
- d) drzewie BST średnia:  $O(\log n)$  pesymistyczna:  $O(n)$
- e) drzewie czerwono-czarnym średnia:  $O(\log n)$  pesymistyczna:  $O(\log n)$

5. Drzewo czerwono-czarne:

- a) struktura węzła i założenia (definicja),

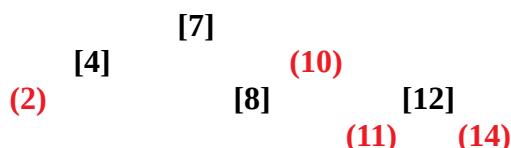
Każdy węzeł drzewa czerwono-czarnego zawiera atrybuty:

- Kolor: czerwony albo czarny
- Klucz
- Wskaźnik na prawego lub lewego syna
- Wskaźnik na ojca

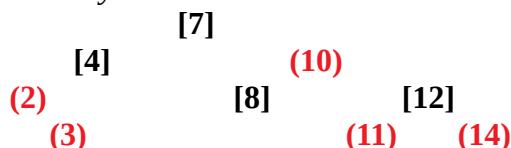
Drzewem czerwono-czarnym nazywamy drzewo przeszukiwań binarnych dla którego każdy węzeł posiada dodatkowy bit koloru – czarny lub czerwony – o następujących właściwościach:

- Każdy węzeł posiada kolor
- Korzeń jest koloru czarnego
- Każdy liść (NULL) jest czarny
- Dla czerwonego węzła dzieci są zawsze czarne
- Każda ścieżka z węzła do liścia ma zawsze taką samą liczbę czarnych węzłów

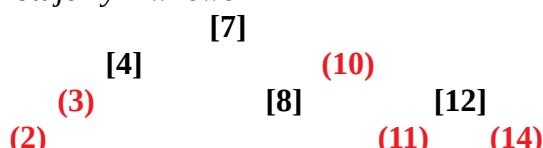
- b) wstaw 3 do poniższego drzewa czerwono-czarnego:



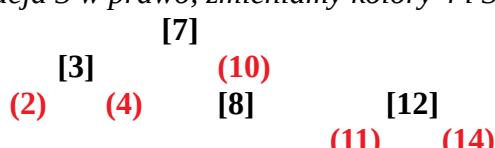
- wstawiamy 3



- rotujemy 2 w lewo

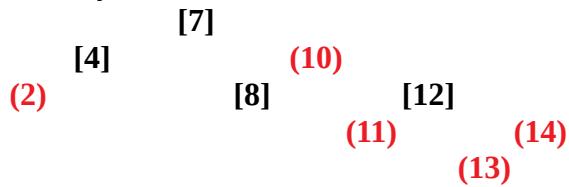


- rotacja 3 w prawo, zmieniamy kolory 4 i 3

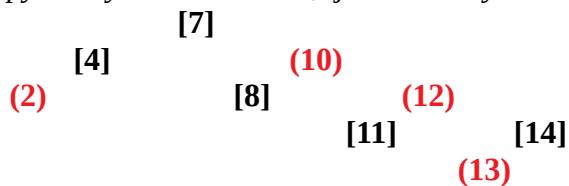


c) wstaw do powyższego drzewa 13

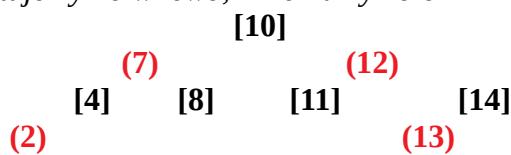
- wstawiamy 13



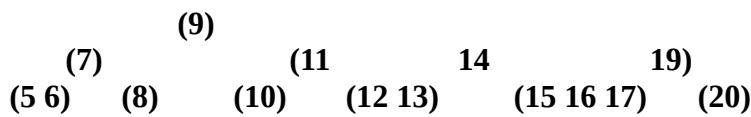
- wypychamy kolor z 11 i 14, tj. zmieniamy kolor 11, 14 i 12



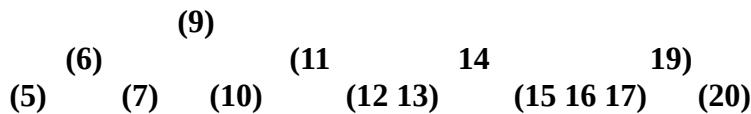
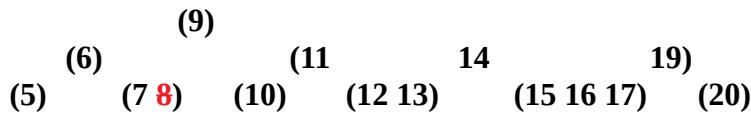
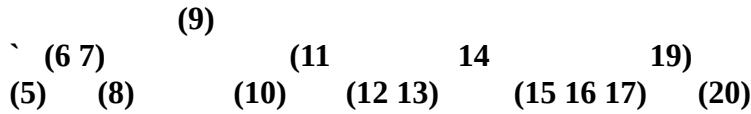
- rotujemy 10 w lewo, zmieniamy kolor 7 i 10



6. Podano na rysunku B-drzewo o  $t = 2$ :



- usuń z tego drzewa 8



- dodaj do niego 21

$$\begin{array}{ccccccccc} & & (9) & & & & & & \\ (7) & (8) & (10) & (11) & (12 \ 13) & 14 & (15 \ 16 \ 17) & 19) & (20 \ 21) \\ (5 \ 6) & & & & & & & & \end{array}$$

7.

a) Napisz twierdzenie o rekursji uniwersalnej

Twierdzenie matematyczne pozwalające na określenie ograniczenia asymptotycznego dla danej klasy funkcji zdefiniowanych.

b) Napisz (jak najściślej) definicję, co oznacza zapis  $f(n) = O(g(n))$

Funkcja  $f$  rośnie szybciej niż  $g$ .

c) Napisz (jak najściślej) definicję, co oznacza zapis  $f(n) = \Theta(g(n))$

Funkcje  $f$  i  $g$  rosną w tym samym tempie.

8. Zastosuj twierdzenie o rekurencji uniwersalnej do rozwiązywania następujących problemów:

Dla funkcji  $T(n)$  zadanej przez

$$T(n) = \begin{cases} \Theta(1), & \text{dla } n \in \{0, 1\} \\ a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n), & \text{dla } n > 1 \end{cases}$$

zachodzi:

- jeśli  $f(n) = O(n^{\log_b a - \epsilon})$  dla pewnego  $\epsilon > 0$ , to  $T(n) = \Theta(n^{\log_b a})$ ,
- jeśli  $f(n) = \Theta(n^{\log_b a})$ , to  $T(n) = \Theta(n^{\log_b a} \lg n)$ ,
- jeśli  $f(n) = \Omega(n^{\log_b a + \epsilon})$  dla pewnego  $\epsilon > 0$  oraz  $af(\frac{n}{b}) \leq cf(n)$  dla pewnego  $c < 1$  i prawie wszystkich  $n$ , to  $T(n) = \Theta(f(n))$ .

a)  $T(n) = 4T(n/2) + n^2$

$$a = 4 \ b = 2 \ f(n) = n^2$$

$$g(n) = n^{\log_2 4}$$

$f(n^2) = O(n^2)$  - rosną w tym samym tempie, przypadek 2

$$T(n) = \Theta(n^{\log_2 4} \lg n)$$

b)  $T(n) = 6T(n/3) + n^2$

$$a = 6 \ b = 3 \ f(n) = n^2$$

$$g(n) = n^{\log_3 6}$$

$f(n^2) = O(n^{\log_3 6})$  - rośnie szybciej, przypadek 1

$$T(n) = \Theta(n^{\log_3 6})$$

c)  $T(n) = 4T(n/4) + n$

$$a = 4 \ b = 4 \ f(n) = n$$

$$g(n) = n^{\log_4 4}$$

$f(n) = O(n)$  - rosną w tym samym tempie, przypadek 2

$$T(n) = \Theta(n^{\log_4 4} \lg n)$$

d)  $T(n) = 7T(n/2) + n^3$

$$a = 7 \ b = 2 \ f(n) = n^3$$

$$g(n) = n^{\log_2 7}$$

$f(n^3) = O(n^{\log_2 7})$  - *frośnie szybciej, przypadek 1*

$$T(n) = \Theta(n^{\log_2 7})$$

e)  $T(n) = 8T(n/2) + n^3$

$$a = 8 \ b = 2 \ f(n) = n^3$$

$$g(n) = n^{\log_2 8}$$

$f(n^3) = O(n^3)$  - *frośnie szybciej, przypadek 1*

$$T(n) = \Theta(n^{\log_2 8} \lg n)$$

f)  $T(n) = T(n/2) + \sqrt{n}$

$$a = 1 \ b = 2 \ f(n) = \sqrt{n}$$

$$g(n) = n^{\log_1 2}$$

$f(\sqrt{n}) = O(n^0)$  - *g jest stałe, zaś frośnie*

$$T(n) = \Theta(n^{\log_1 2} \lg n)$$

9.

- a) Napisz funkcję **int max(lnode\* l)** zwracającą największy klucz listy linkowanej o początku l, jeśli lnode jest zdefiniowana tak: **struct lnode{ int x; lnode\* next; };**

```
int max(lnode* l) {
    int max = INT_MIN;
    while(l != NULL) {
        if(max < l->x) {
            max = l->x;
        }
        l = l->next;
    }
    return max;
}
```

- b) Napisz funkcję **int max(node\* t)** zwracającą największy klucz drzewa BST o korzeniu t, jeśli node jest zdefiniowana tak: **struct node{ int x; node\* left; node\* right; };**

```
int max(node* t) {
    node* temp = t;
    while(temp->right != NULL) {
        temp = temp->right;
    }
    return (temp->x);
}
```

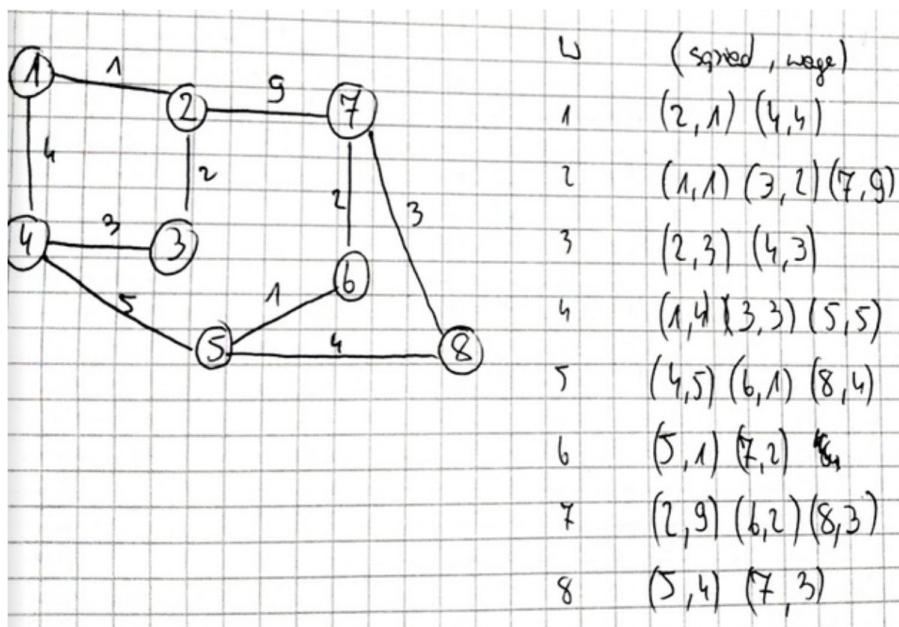
- c) Napisz funkcję **int max(node\* t, int limit)** zwracającą największy klucz drzewa BST mniejszy od liczby **limit**.

```
int max(node* t, int limit) {
    node* temp = t;
    while(temp->right != NULL || temp->right <= limit) {
        temp = temp->right;
    }
    return (temp->x);
}
```

10. Na podstawie grafu nieskierowanego zadanego jako następująca lista krawędzi:

(1,2):1 (2,3):2 (3,4):3 (1,4):4 (4,5):5 (5,6):1 (6,7):2 (7,8):3 (5,8):4 (2,7):9

- a) Wykonaj rysunek grafu i zapisz tablicę list sąsiedztwa. Wierzchołki na listach sąsiedztwa powinny być ustalone rosnąco wg numeru wierzchołka. Ta kolejność powinna być stosowana w symulacji algorytmów DFS, BFS i Dijkstry.



- b) Zapisz kolejność odwiedzania wierzchołków przez algorytm **DFS** startujący z wierzchołka 5.

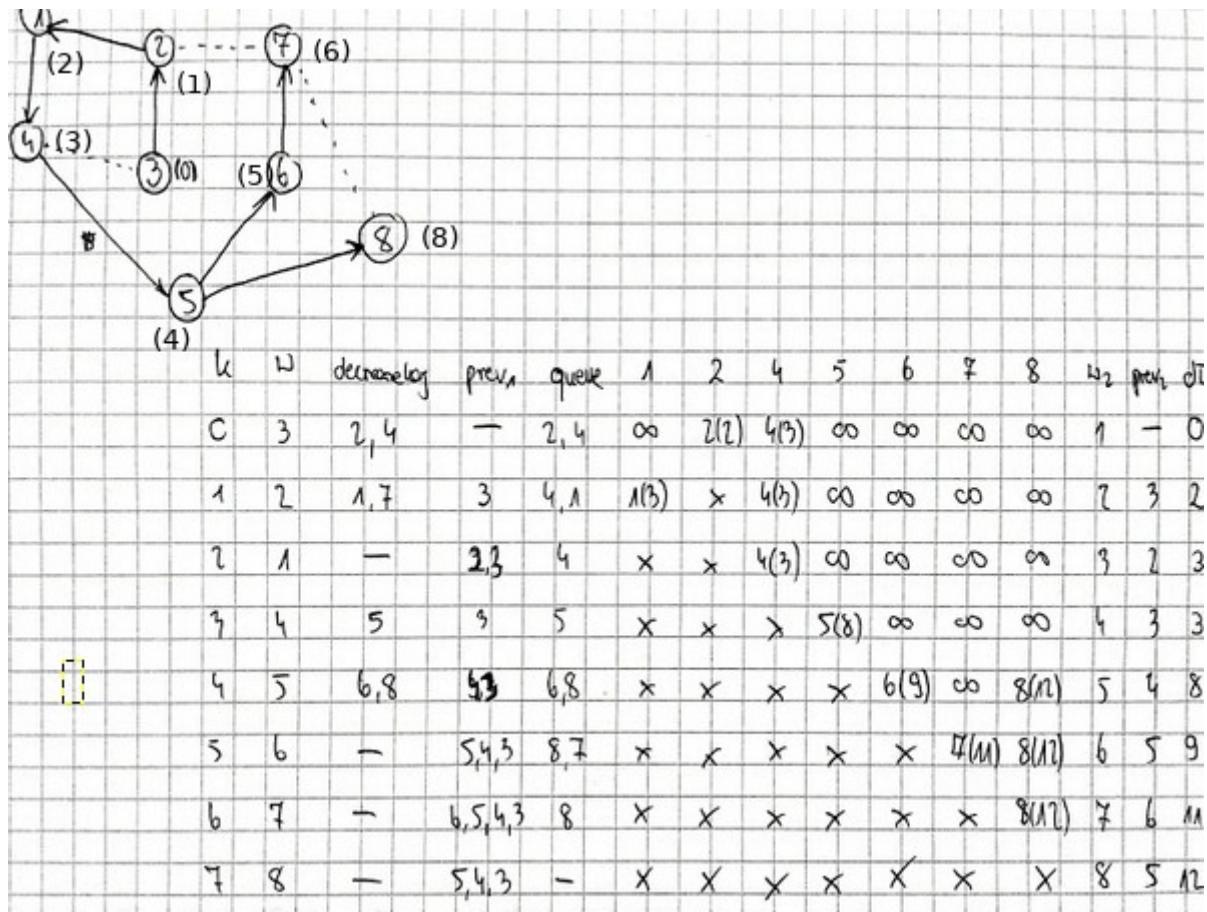
k	s	t
1	s	s
2	s6	s6
3	s67	s67
4	s678	s678
	s678	
5	s672	s6782
6	s6721	s67821
7	s67214	s678214
8	s672143	<u>s6782143</u>
	s6782143	

- c) Zapisz kolejność odwiedzania wierzchołków w algorytmie **BFS** startującym z tego samego wierzchołka.

k	q	t
1	s	s
2	s684	s684
3	s6847	s6847
4	<u>s684731</u>	s684731
5	<u>s6847312</u>	<u>s6847312</u>

11. Dla grafu z poprzedniego zadania zasymuluj działanie algorytmu **Dijkstry startując z wierzchołka 3**. Algorytm zilustruj grafem, w którym:

- Przy każdym wierzchołku będzie podany w nawiasie okrągły numer kroku algorytmu, w którym wierzchołek został odwiedzony
- Strzałkami ciągłymi oznaczone będą krawędzie należące do drzewa wynikowego
- Strzałkami przerywanymi oznaczone będą krawędzie, które w trakcie algorytmu wskazywały na poprzednika, jednak nie należą do drzewa wynikowego.



## SESJA – TERMIN II

1.

- a) Napisz w C++ procedurę **void insertion\_sort(double t[], int n)** (sortowanie przez wstawianie)

```
void insertion_sort(double t[], int n) {
    for(int k = 1; k > n; k++)
        for(int i = k; i > 0 && t[i-1]; i--)
            std::swap(t[i], t[i-1]);
}
```

- b) Jaka jest jego średnia złożoność :  $O(n^2)$   
 c) Jak faktyczna złożoność zależy od rozmiaru danych (n) oraz ilości inwersji danych wejściowych (m) :  $O(n*m)$

2.

- a) Napisz funkcję `int partition(double t[], int n)`

```
int partition(double t[], int n) {
    int k = -1;           // indeks pomocniczy
    double x = t[n/2];   // wybrany pivot

    for(;;) {
        // przesuwamy n w lewo od końca, dopóki n-ty element >pivot
        while(t[--n] > x);
        // -- w prawo od początku, -- k-ty element <pivot
        while(t[++k] < x);
        // zamieniamy n-ty i k-ty element jeśli k < n
        if(k < n) {
            std::swap(t[k], t[n]);
        } else
            return k; // zwracamy indeks pivota
    }
}
```

- b) Co jest wynikiem funkcji `partition`? Jaka jest jej złożoność?

Wynikiem funkcji `partition` jest tablica z elementami mniejszymi od pivota po lewej jego stronie, większymi po prawej.

Złożoność:  $O(n * \log(n))$

- c) Napisz funkcję `void quick_sort(double t[], int n)`

```
void quick_sort(double t[], int n) {
    if(n > 1) {
        int k = partition(t, n);
        quick_sort(t, k);
        quick_sort(t + k, n - k);
    }
}
```

- d) Jaka jest złożoność `quick_sort` (średnia i pesymistyczna)? Jaka jest maksymalna głębokość rekursji? Odpowiedź uzasadnij.

Złożoność średnia  $O(n * \log(n))$ , pesymistyczna:  $O(n^2)$

Maksymalna złożoność rekursji  $O(n)$ , Filip tak powiedział, a on nie może kłamać

- e) Podaj przykład danych, dla których algorytm nie jest stabilny i zasymuluj jego działanie dla tych danych

Algorytm jest niestabilny w przypadku posiadania dwóch elementów o tej samej wartości. Zostaną one zamienione miejscami. Przykład:

1 5 4 2 5 6    4 jest pivot-em  
1 5 3 5 2 4      >      1 2 3 5 5 4 (dochodzi do zamiany piątek)      >      1 2 3 4 5 5

### 3.

- a) Napisz w języku C++ procedurę **void counting\_sort(int n, int t[], int c = 1)**, która sortuje stabilnie w czasie liniowym tablicę t rozmiaru n względem cyfry jedności (gdy c = 1), dziesiątek (gdy c = 10) itd.

```
void counting_sort(int t[], int n, int c = 1) {
    int output[n];
    int i, count[10] = {};

    for(i = 0; i < n; i++)
        count[ (t[i] / c) % 10 ]++;

    for(i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for(i = n-1; i >= 0; i--) {
        output[count[ (t[i] / c) % 10 ] - 1] = t[i];
        count[ (t[i] / c) % 10 ]--;
    }

    for(i = 0; i < n; i++)
        t[i] = output[i];
}
```

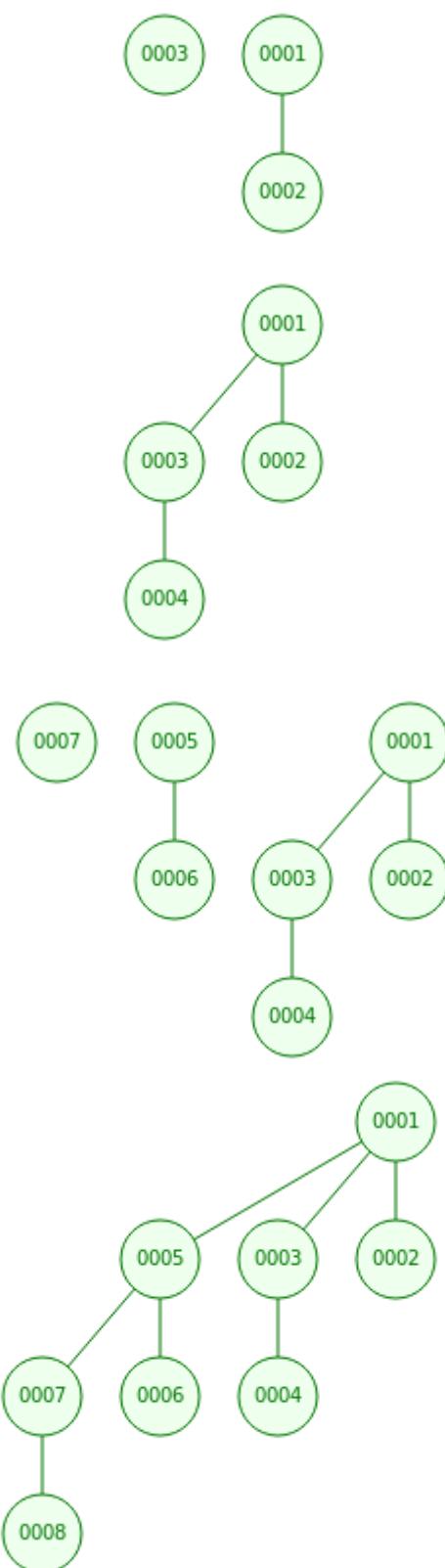
- b) Wykorzystaj ją do napisania procedury **void radix\_sort(int t[], int n)** zakładając, że w tablicy występują nieujemne liczby mniejsza od M.

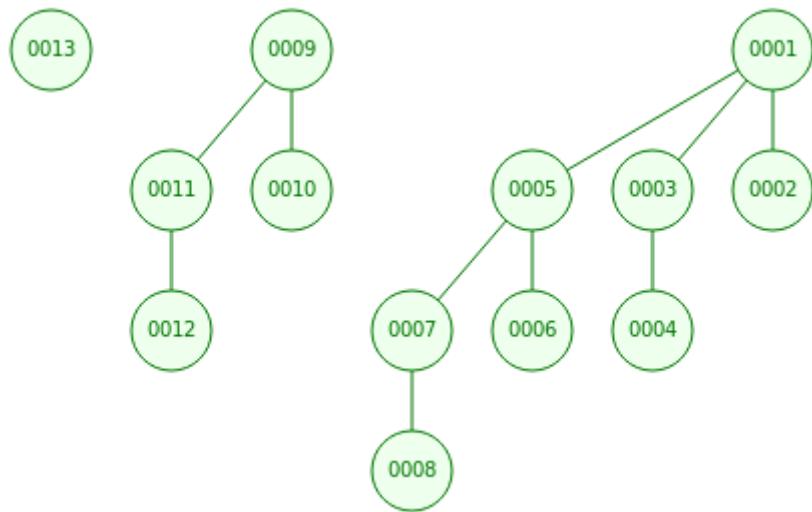
```
int radix_sort(int t[], int n) {
    int m = t[0];
    for(int i = 1; i < n; i++)
        if(t[i] > m)
            m = t[i];

    for(int c = 1; m/c > 0; c *= 10)
        counting_sort(t, n, c);
}
```

4.

- a) Jakie dane przechowywane są w każdym węźle kopca dwumianowego
- b) Narysuj drzewo dwumianowe rzędu 3, uwzględniając na rysunku wszystkie dane zawarte w każdym z węzłów
  
- c) Narysuj schematycznie kopiec dwumianowy zawierający liczby całkowite od 1 do 13 tak, aby klucz 13 znajdował się w ostatnim drzewie





- d) Wykonaj na tym drzewie operację GETMAX
- e) Wykonaj na tym drzewie operację INSERT(14)
- f) Wykonaj UNION tego kopca z kopcem zawierającym klucze od 14 do 20.

5. (?) Zastosuj twierdzenie o rekurencji uniwersalnej do rozwiązywania następujących problemów:

a)  $T(n) = 3T(n/2) + n^2$

$$a = 3 \quad b = 2 \quad f(n) = n^2$$

$$g(n) = n^{\log_2 3}$$

$$f(n^2) = O(n^{1.58}) \quad - f \text{ rośnie szybciej, opcja 1}$$

$$T(n) = \Theta(n^{\log_2 3})$$

b)  $T(n) = 4T(n/2) + n^2$

$$a = 4 \quad b = 2 \quad f(n) = n^2$$

$$g(n) = n^{\log_2 4}$$

$$f(n^2) = O(n^2) \quad - f \text{ i } g \text{ rosną z taką samą prędkością, opcja 2}$$

$$T(n) = \Theta(n^{\log_2 4} \lg n)$$

c)  $T(n) = 3T(n/3) + n^2$

$$a = 3 \quad b = 3 \quad f(n) = n^2$$

$$g(n) = n^{\log_3 3}$$

$$f(n^2) = O(n) \quad - f \text{ rośnie szybciej}$$

$$T(n) = \Theta(n^{\log_3 3})$$

d)  $T(n) = 7T(n/2) + n^2$

$$\begin{aligned} a &= 7 & b &= 2 & f(n) &= n^2 \\ g(n) &= n^{\log_2 7} \\ f(n^2) &= \Theta(n^{2.8}) & - f \text{ rośnie wolniej od } g \\ T(n) &= \Theta(f(n^2)) \end{aligned}$$

e)  $T(n) = 2T(n/4) + n^{0.5}$

$$\begin{aligned} a &= 2 & b &= 4 & f(n) &= n^{0.5} \\ f(g) &= n^{\log_2 4} \\ f(n^{0.5}) &= \Theta(n^2) & - f \text{ rośnie wolniej od } g \\ T(n) &= \Theta(f(n^{0.5})) \end{aligned}$$

6.

- a) Do tablicy z haszowaniem podwójnym i rozmiarze  $m = 16$  o funkcji haszującej:  
 $h_1(x) = x \bmod 16$  i  $h_2(x) = (2x^2 + 1) \bmod 16$  wyznacz ciąg kontrolny dla liczby 45.
- b) Czy można przyjąć  $h_2(x) = (3x^2 + 1) \bmod 15 + 1$ ? Odpowiedź uzasadnij.

7. Dane są definicje: struct lnode{int x; lnode\* next; }; oraz  
struct node{ int x; node\* left; node\* right; };

- a) Napisz nierekurencyjną funkcję int suma(lnode\* l) zwracającą sumę kluczy listy linkowej o początku 1

```
int suma(lnode* l) {
    int counter = 0;
    while(l->next != NULL) {
        counter += 1;
        l = l->next;
    }
    return counter;
}
```

- b) Napisz funkcję int suma(node\* t, int a, int b) zwracającą sumę kluczy drzewa BST należących do przedziału domkniętego  $[a,b]$ . Zadbaj by nie odwiedzała lewych poddrzew elementów mniejszych od a i prawych poddrzew elementów większych od b.

```

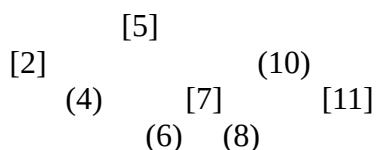
int suma(node* t, int a, int b) {
    if(!t) return 0;
    if(t->x == a && t->x == b) {
        return (t->x);
    }
    if(t->x <= b && t->x >= a) {
        return (t->x + suma(t->left, a, b) + suma(t->right, a, b));
    }
    else if(t->x < a)
        return suma(t->right, a, b);
    else return suma(t->left, a, b);
}

```

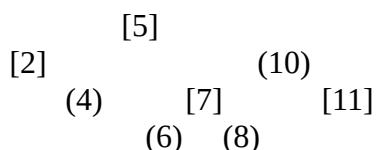
8.

- a) Podaj definicję B-drzewa (obowiązuje wersja z wykładu i Cormen)
- b) Narysuj minimalne poprawne B-drzewo o  $t = 4$ , które ma klucze na 3 poziomach.
- c) Usuń z niego klucz znajdujący się w korzeniu

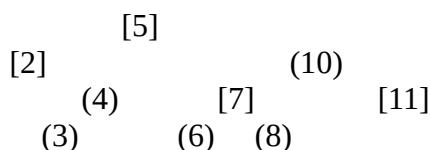
9. W poniższym drzewie czerwono-czarnym czarne węzły oznaczono nawiasem kwadratowym:



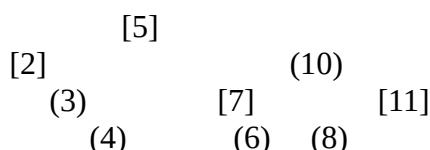
- a) wstaw do tego drzewa 3



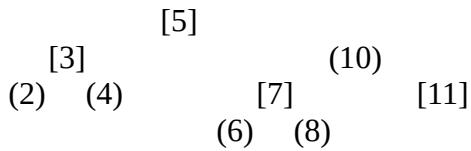
- wstawiamy 3



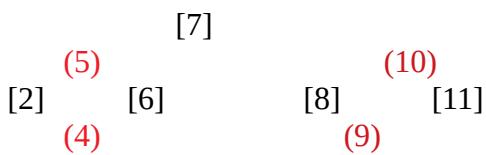
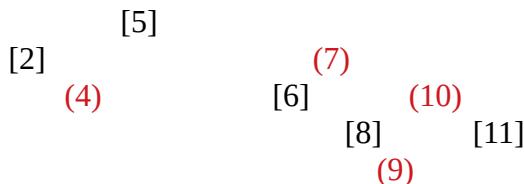
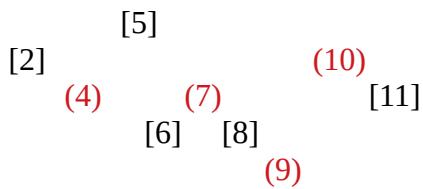
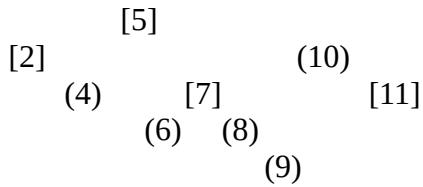
- rotacja w prawo 3 i 4



- rotacja w lewo 2, 3 i 4, przekolorowanie 2 i 3



b) wstaw do tego drzewa 9



10. Na podstawie grafu nieskierowanego zadanego jako następująca lista krawędzi:

(1,2):5 (1,7):6 (1,8):1 (2,3):1 (2,6):15 (2,8):4 (3,4):6 (3,5):2 (4,5):3 (5,6):11 (6,7):7  
 (6,8):3 (7,8):2

a) Wykonaj rysunek grafu i zapisz tablicę list sąsiedztwa. Wierzchołki na listach sąsiedztwa powinny być ustawione rosnąco wg numeru wierzchołka. Ta kolejność powinna być stosowana w symulacji algorytmów DFS, BFS i Dijkstry.

b) Zapisz kolejność odwiedzania wierzchołków przez algorytm **DFS** startujący z **wierzchołka 6**.

DFS		
u	s	t
1	b	b
2	b8	b8
3	b81	b81
4	b812	b812
5	b8123	b8123
6	b81235	b81235
7	b812354	b812354
8	b817	<u>b8123547</u>
9	XXXX	

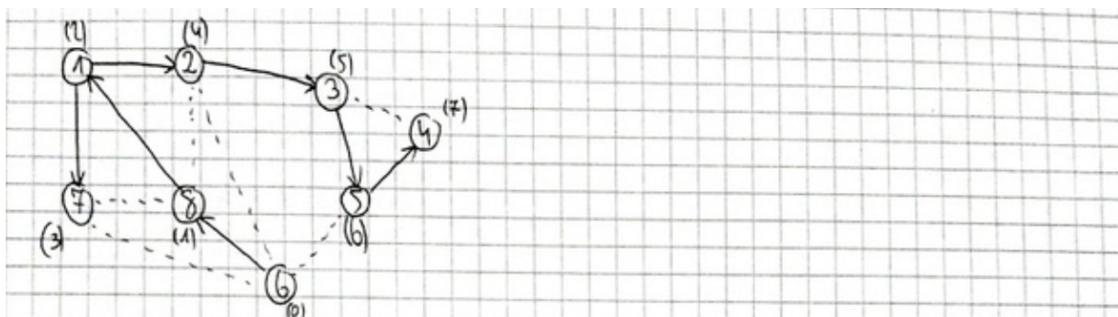
c) Zapisz kolejność odwiedzania wierzchołków w algorytmie **BFS** startującym z tego samego wierzchołka.

k	q	t
1	b	b
2	<u>b8752</u>	b8752
3	<u>b87521</u>	b87521
4	<u>b8752134</u>	<u>b8752134</u>
5	<u>b8752134</u>	

11. Dla grafu z poprzedniego zadania zasymuluj działanie algorytmu **Dijkstry startując z wierzchołka 6**. Algorytm zilustruj grafem, w którym:

- Przy każdym wierzchołku będzie podany w nawiasie okrągły numer kroku algorytmu, w którym wierzchołek został odwiedzony

- Strzałkami ciągłymi oznaczone będą krawędzie należące do drzewa wynikowego
- Strzałkami przerywanymi oznaczone będą krawędzie, które w trakcie algorytmu wskazywały na poprzednika, jednak nie należą do drzewa wynikowego.



k	↳	decrease[k]	prev	queue	1	2	3	4	5	7	8	↳	prev	d2	
0	6	8,7,5,2	—	8752	∞	2(15)	∞	∞	∞	8(1)	7(7)	8(5)	1	—	0
1	8	6	7,5,4,1	1(4)	2(15)	∞	∞	∞	∞	7(5)	4(5)	x	2	6	3
2	1	8,6	5,2,7	x	2(7)	∞	∞	∞	∞	7(1)	7(5)	x	3	8	4
3	7	8,6	2,5	x	2(7)	∞	∞	∞	∞	8(1)	x	x	4	1	5
4	2	8,6	5,3	x	x	3(8)	∞	∞	8(1)	x	x	5	7	3	6
5	3	7,8,6	5,4	x	x	x	4(13)	8(10)	x	x	6	2	8	7	10
6	5	3,2,8,6	4	x	x	x	4(13)	x	x	x	7	3	10	8	13
7	4	5,3,2,8,6	—	x	x	x	x	x	x	x	8	5	13	7	12