Algorytmy i struktury danych

Norbert Jankowski

Katedra Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika

www.is.umk.pl/~norbert/asd

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany ze środków
Uni Europejskiej w ramach
Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Literatura

A. V. Aho, J. E. Hopcroft, J. D. Ullman.
Projektowanie i analiza algorytmów.
Helion, Warszawa, 2003.

A. V. Aho, J. E. Hopcroft, J. D. Ullman.
Projektowanie i analiza algorytmów.
Państwowe Wydawnictwa Naukowe, Warszawa, 1983.

Niklaus Wirth.
Algorytmy + Struktury Danych = Programy.
Wydawnictwa Naukowo-Techniczne, Warszawa, wydanie 2, 1989.

T. H. Cormen, C. E. Leiserson, R. L Rivest.
Wprowadzenie do algorytmów.
Wydawnictwa Naukowo-Techniczne, Warszawa, 1997.

L. Banachowski, K. Diks, W. Rytter.
Algorytmy i struktury danych.
Wydawnictwa Naukowo-Techniczne, Warszawa, 1996.

M. M. Sysło, N. Deo, J. Kowalik.
Algorytmy optymalizacji dyskretnej.
Państwowe Wydawnictwa Naukowe, Warszawa, 1993.

D. Harel.
Rzecz o istocie informatyki. Algorytmika.
Wydawnictwa Naukowo-Techniczne, Warszawa, 1992.

L. Banachowski, A. Kreczmar, W. Rytter.
Analiza algorytmów i struktur danych.
Wydawnictwa Naukowo-Techniczne, Warszawa, 1989.

L. Banachowski, A. Kreczmar.

Elementy analizy algorytmów.

Wydawnictwa Naukowo-Techniczne, Warszawa, 1982.

D. E. Knuth.

Sztuka Programowania, wolumen I–III.

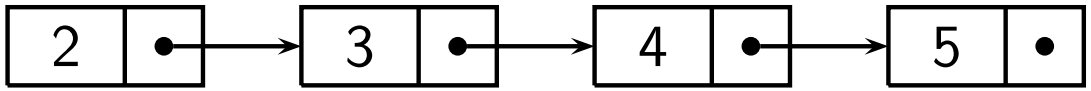
Wydawnictwa Naukowo-Techniczne, 2002.

Dynamiczne struktury danych

- Listy
- FIFO
- Stos
- Kolejki
- Listy dwukierunkowe
- Listy cykliczne
- Drzewa
- Drzewa binarne, z porządkiem, dwukierunkowe
- Drzewa zrównoważone (później ...)
- Grafy i ich reprezentacje (macierz sąsiedztwa, listy sąsiedztwa, tablica sąsiedztwa)

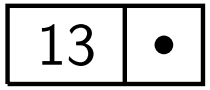
Struktura listy

```
1 struct el {
2     int klucz;
3     struct el *nast;
4 };
5
6 typedef struct el elListy;
7 typedef elListy *lista;
```



Stworzenie elementu listy

```
1 lista  l=0;        // bądź: elListy *l;
2
3 l = malloc(sizeof( elListy ));
4 l->klucz = 13;
5 l->nast = 0;
```

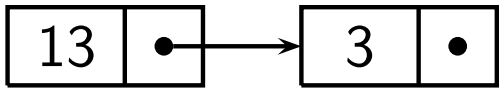


Usunięcie elementu listy

```
1 free(l);
2 l=0;
```

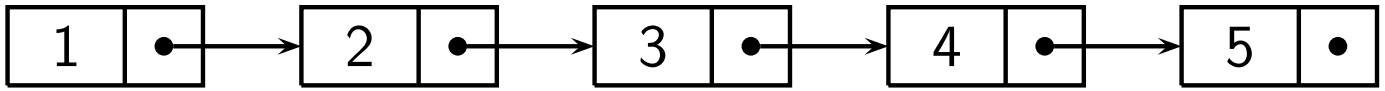
Połączenie dwóch elementów *p* i *q* w listę

```
elListy *p=0, *q=0;

p = malloc(sizeof( elListy ));
p->klucz = 3;
p->nast = 0;
q = malloc(sizeof( elListy ));
q->klucz = 13;
q->nast = p;
```



Stworzenie listy elementów: $1, 2, 3, 4, \ldots, n$

```
elListy *l=0, *p=0;
int n=5;
while(n--){
    p = malloc(sizeof( elListy ));
    p->klucz = n+1;
    p->nast = l;
    l = p;
}
```

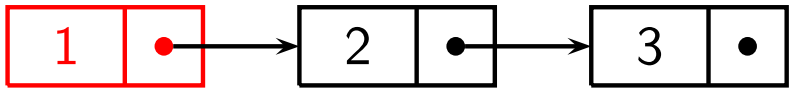


Wyświetlenie listy

```
1  void WyswietlListe( lista _lista )
2  {
3      lista l = _lista;
4      while (l){
5          printf ("%d-", l->klucz);
6          l = l->nast;
7      }
8      printf ("|\n");
9  }
10
11 int main(int argc, char* argv[])
12 {
13     lista _l = 0;
14     ...
15     WyswietlListe(_l);
16 }
```

Dodanie elementu na początek listy

```
1  void DNPL(lista *l, int i) // l wsk. na listę!
2  {
3      lista p = (lista)malloc(sizeof(elListy));
4      p->klucz = i;
5      p->nast = *l;
6      *l = p;
7  }
8  int main(int argc, char* argv[]) {
9      lista _l = 0;
10     DNPL(&_l, 3);
11     DNPL(&_l, 2);
12     DNPL(&_l, 1);
13     DNPL(&_l->nast, 0);
14     WyswietlListe(_l);
15 }
```



Dodanie elementu na koniec listy

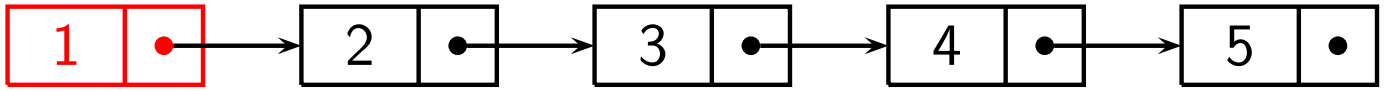
```
1  void DNKL(lista *l, int i)
2  {
3      lista p = (lista)malloc(sizeof(elListy));
4      p->klucz = i;
5      p->nast = 0;
6      while ((*l))
7          l = &(*l)->nast;
8      *l = p;
9  }
10 int main(int argc, char* argv[])
11 {
12     lista _l = 0;
13     DNKL(&_l, 1);
14     DNKL(&_l, 2);
15     DNKL(&_l, 3);
16     WyswietlListe(_l);
17 }
```

Dodanie elementu na koniec listy inaczej...

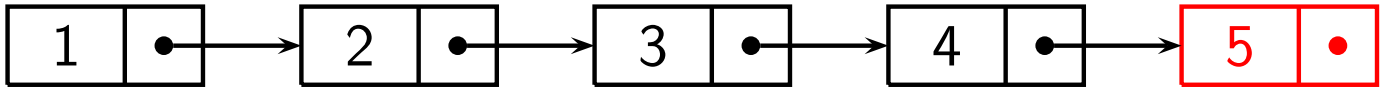
```
1 void DNKL2(lista *l, int i)
2 {
3   while (*l)
4     l = &(*l)->nast;
5   DNPL(l, i);
6 }
```

Usunięcie pierwszego elementu listy *l*

```
1  void UPeL(lista *l){ // wsk. na listę!
2      lista p;
3      if(*l != 0){
4          p = *l;
5          *l = (*l)->nast;
6          free(p);
7      }
8  }
9  int main(int argc, char* argv[])
10 {
11     ...
12     UPeL(&_l);
13 }
```



Usunięcie ostatniego elementu listy




```
1 void UOeL(lista *l){ // wsk. na listę!
2       if(*l == 0) return;
3       while((*l)->nast)
4             l = &(*l)->nast;
5       free(*l);
6       *l=0;          // nowy koniec listy!
7 }
```

Zwolnienie całej listy


```
1  void ZwolnijListe ( lista *l)
2  {
3     lista p = *l;
4    while (p)
5    {
6       *l = (*l)->nast;
7       free(p);
8       p = *l;
9    }
10 }
```

Zwolnienie całej listy inna wersja

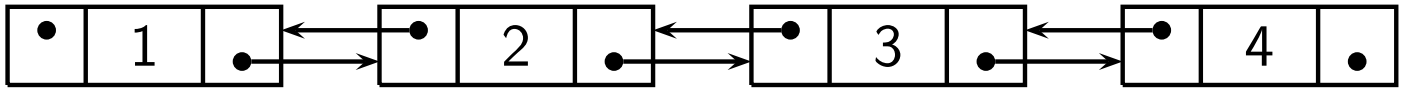
```
1  void ZwolnijListe2 ( lista *l)
2  {
3     lista p = *l, p2;
4    *l = 0;
5    while (p)
6    {
7       p2 = p;
8       p = p->nast;
9       free(p2);
10   }
11 }
```

Stos

- dodawanie
- ściąganie

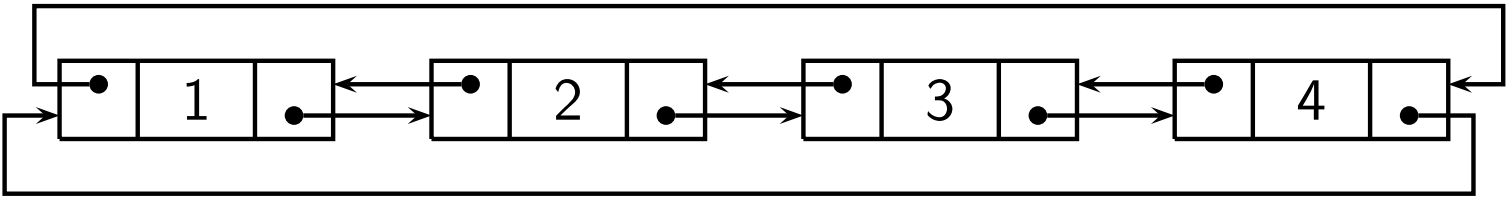
```
1  void StosDodaj(lista *l, int i)
2  {
3    DNPL(l, i);
4  }
5  int StosSciagnij(lista *l) {
6    lista p = *l;
7    int i;
8    if (p == 0) return INT_MAX;
9    i = p->klucz;
10   *l = (*l)->nast;
11   free(p);
12   return i;
13 }
```

Lista dwukierunkowa



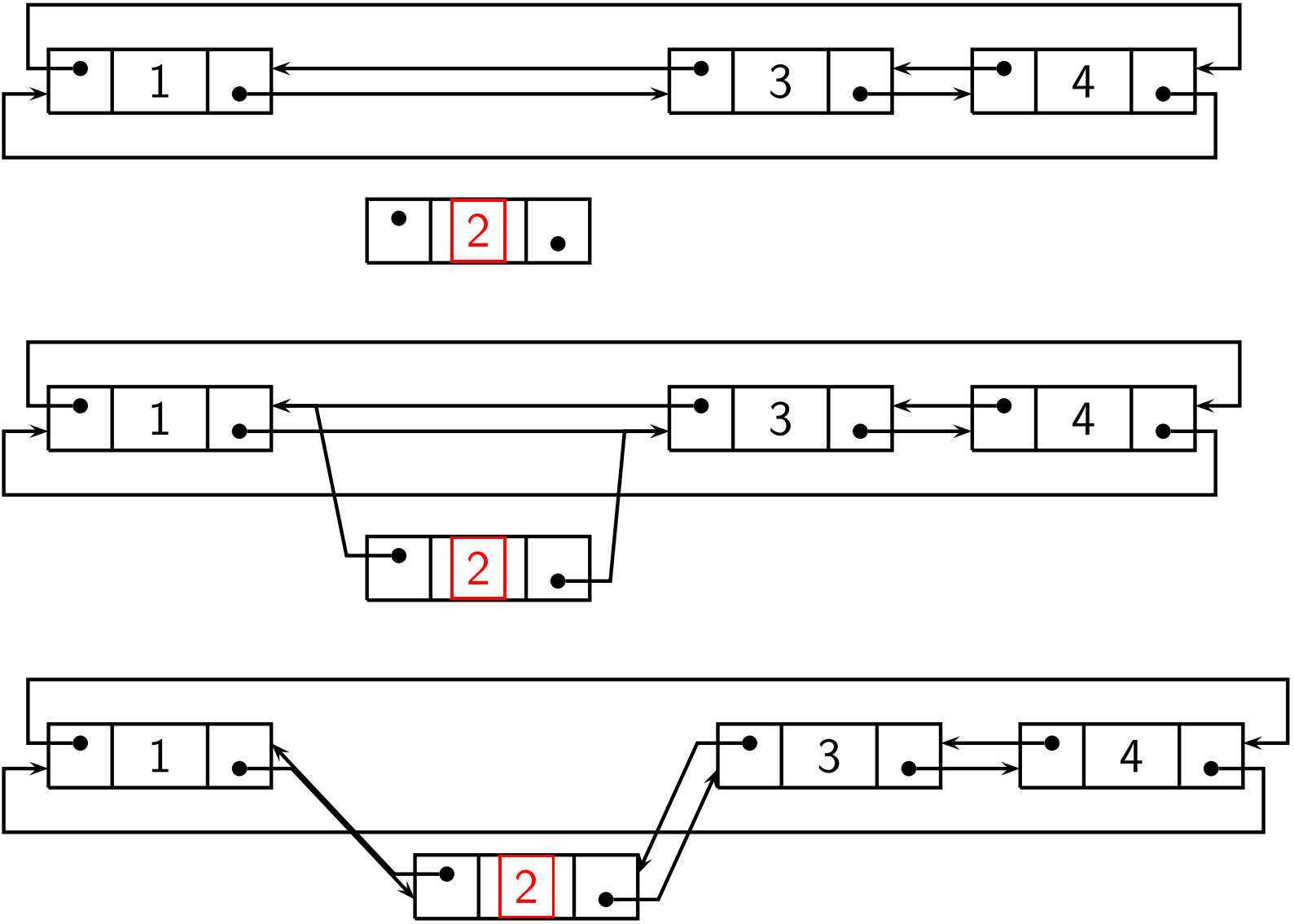

```
1 struct elC
2 {
3   int klucz;
4   struct elC *nast;
5   struct elC *pop;
6 };
7 typedef struct elC elListyC;
8 typedef elListyC* listaC;
```

Lista dwukierunkowa cykliczna



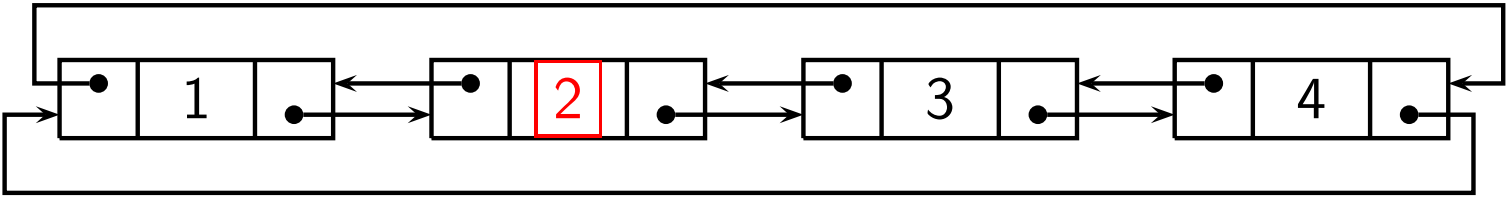

```
1 struct elC
2 {
3   int klucz;
4   struct elC *nast;
5   struct elC *pop;
6 };
7 typedef struct elC elListyC;
8 typedef elListyC* listaC;
```

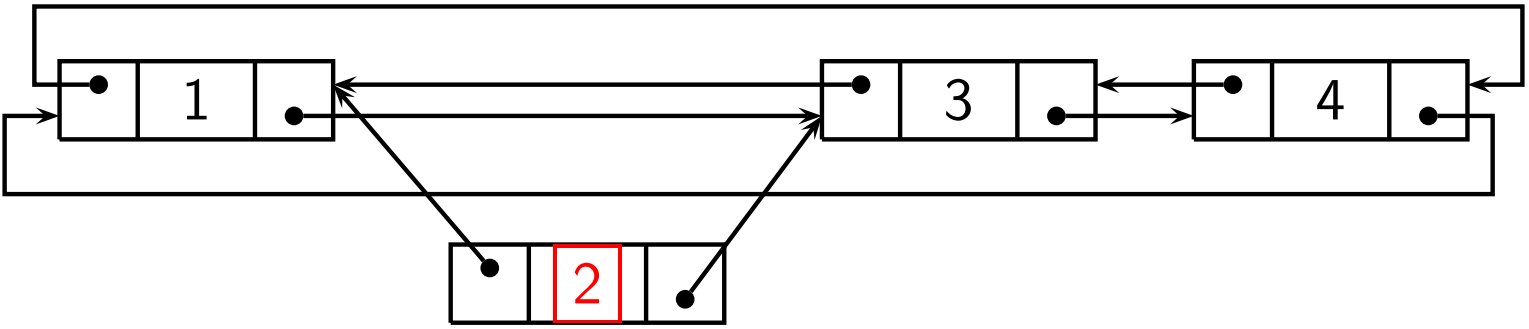
Dodanie elementu do l. cyklicznej

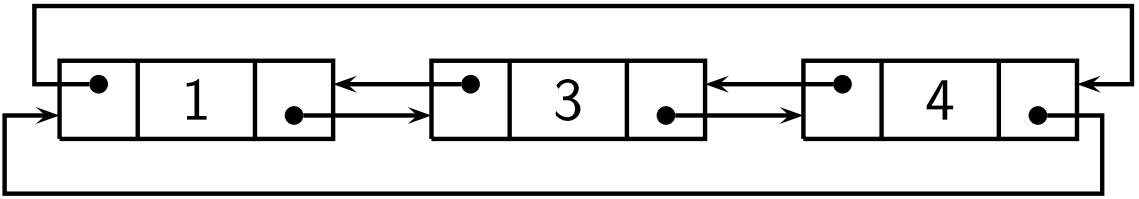


```
1  void DodajListaC(listaC *_lista, int i)
2  {
3    listaC p, n;
4    p = (listaC)malloc(sizeof(elListyC));
5    p->klucz = i;
6    if (*_lista == 0){
7      *_lista = p;
8      p->nast = p;
9      p->pop = p;
10   }
11   else{
12     n = (*_lista)->nast;
13     p->pop = *_lista;
14     p->nast = n;
15     (*_lista)->nast = p;
16     n->pop = p;
17   }
18 }
```

Usunięcie elementu z l. cyklicznej



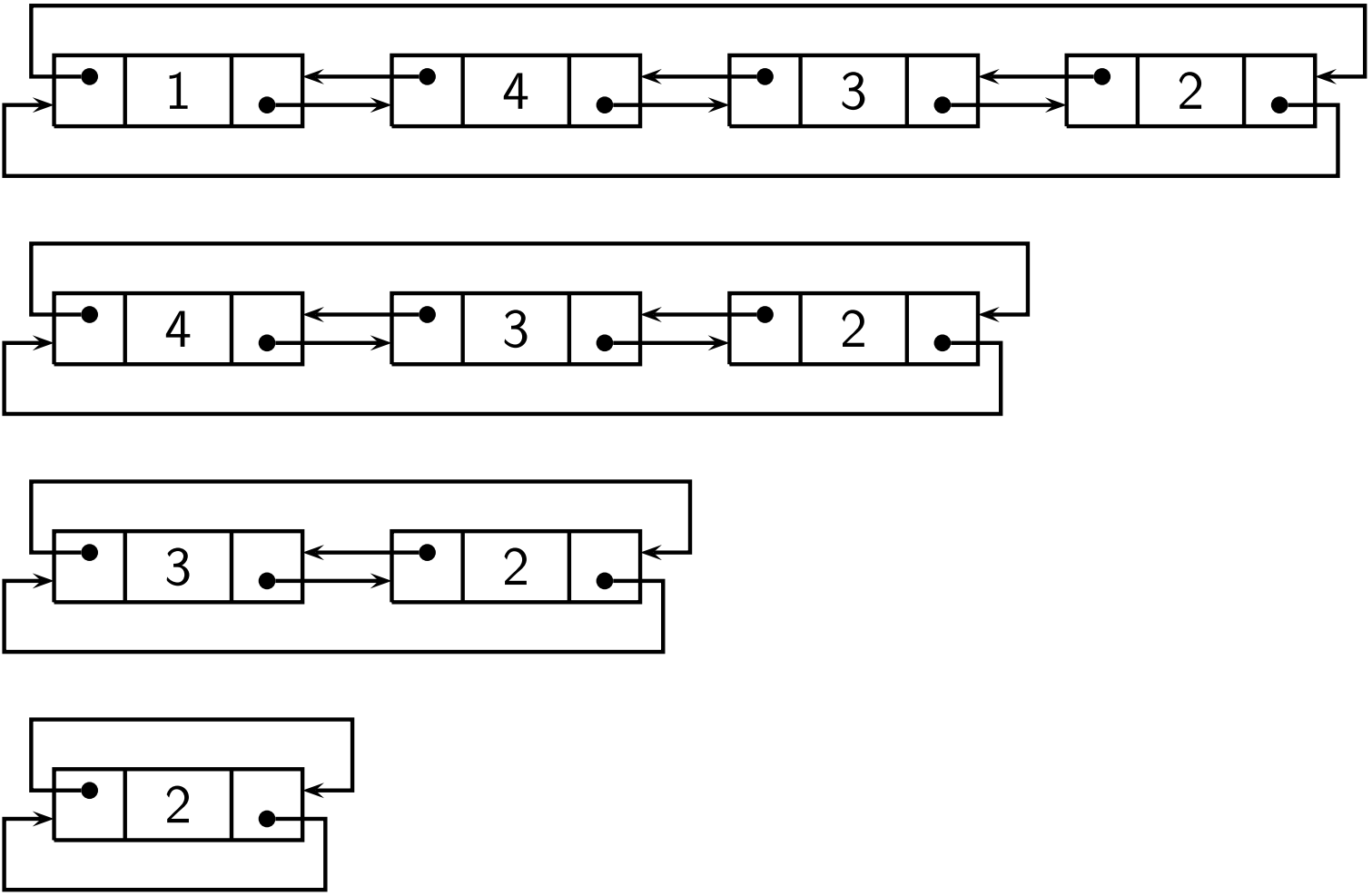




```
1  void UsunListaC(listaC *_lista)
2  {
3     listaC p;
4     if (*_lista == 0) return;
5     if (*_lista == (*_lista)->pop){
6        free(*_lista);
7        *_lista = 0;
8     }
9     else{
10       p = *_lista;
11       p->pop->nast = p->nast;
12       p->nast->pop = p->pop;
13       *_lista = p->nast;
14       free(p);
15    }
16 }
```

Dodanie elementu do l. cyklicznej

```
DodajListaC(l,1);
DodajListaC(l,2);
DodajListaC(l,3);
DodajListaC(l,4);
    DispListCTeX(l,'a');
UsunListaC(l);
    DispListCTeX(l,'a');
UsunListaC(l);
    DispListCTeX(l,'a');
UsunListaC(l);
    DispListCTeX(l,'a');
UsunListaC(l);
    DispListCTeX(l,'a');
```


1
4
3
2
4
3
2
3
2
2

Wypisuje elementy l. cyklicznej

```
1  void WyswietlListe( listaC _lista )
2  {
3     listaC l = _lista;
4     if (l != 0)
5       do{
6          printf ("%d-", l->klucz);
7          l = l->nast;
8       } while (l != _lista );
9     printf ("|\n");
10 }
```

Usuwanie l. cyklicznej

```
1 void ZwolnijListe(listaC *_lista)
2 {
3   while (*_lista)
4     UsunListaC(_lista);
5 }
```

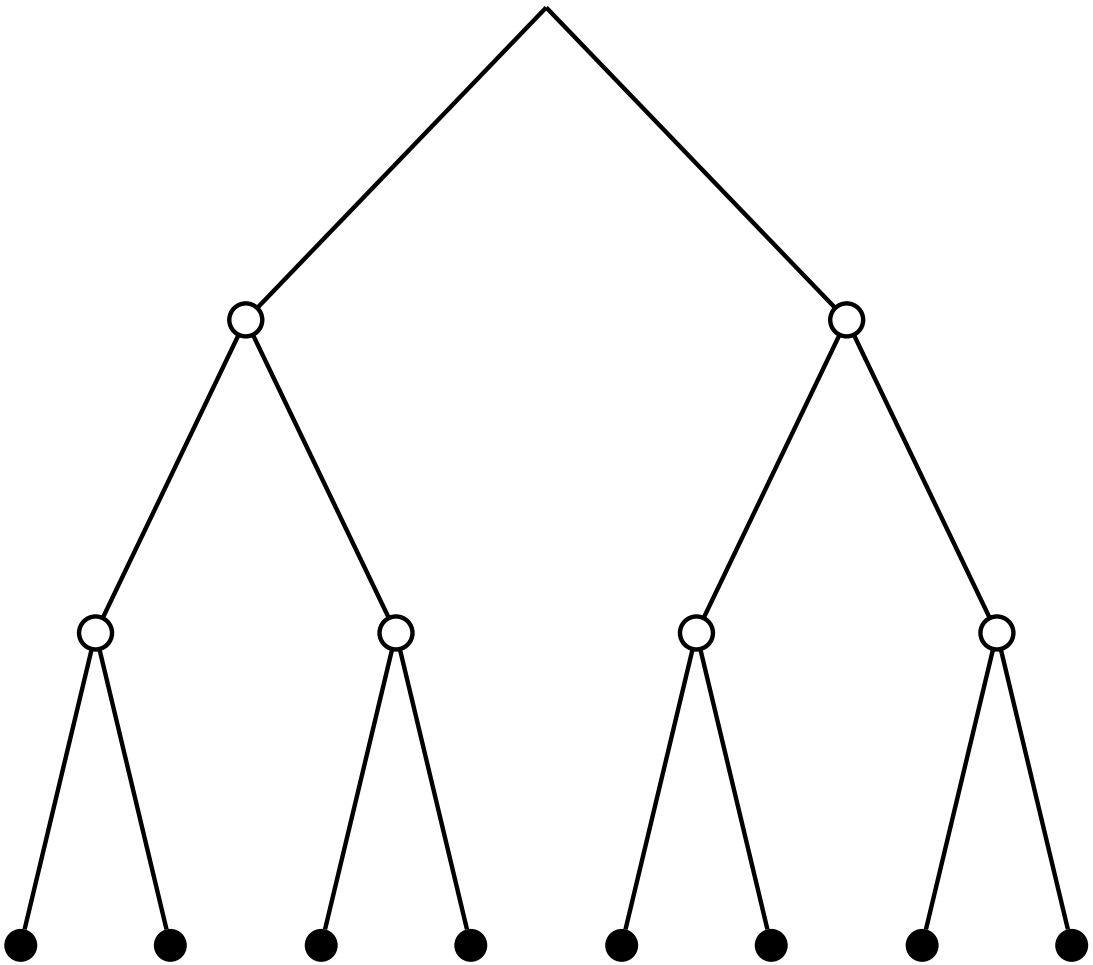
FIFO,LIFO/Stos, kolejki priorytetowe

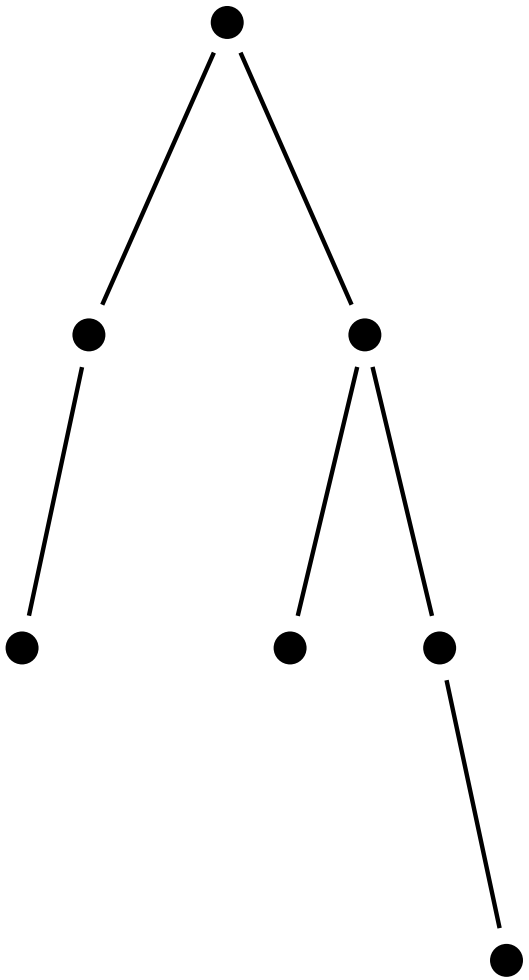
- Koncepcja wartownika dla list
- FIFO — fist in first out
  - wystarczy dobrze używać listy ...
  - dodaj do kolejki — $O(1)$
  - zdjąć z kolejki — $O(1)$
  - dodajemy na koniec kolejki, zdejmujemy element-głowę
- Stos = LIFO — last in first out
  - wystarczy dobrze używać listy ...
  - dołóż na stos — $O(1)$
  - zdjąć ze stosu — $O(1)$
  - dodawanie/zdejmowanie na/z początek/-ku listy
- kolejka priorytetowa
  - pola węzła: *klucz*, *priorytet* i *nast*!
  - wstawianie zgodnie z priorytetem
  - zdejmowanie elementu o najwyższym priorytecie
  - zmiana priorytetu, usuwanie elementu

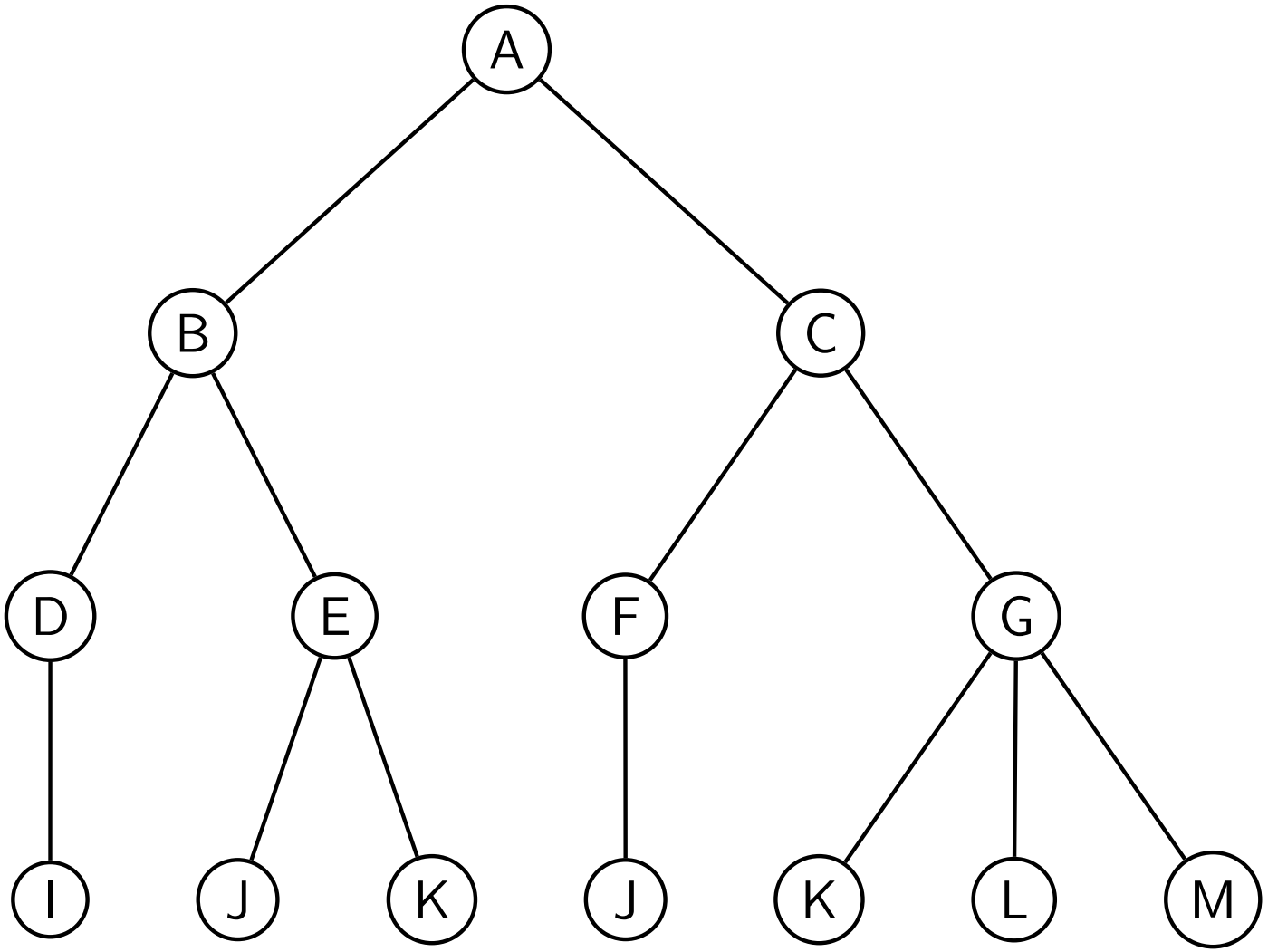
Zadanie:

- Zaimplementuj rekurencyjne wersje omówionych funkcji.
- Zaimplementuj łączenie dwóch list.
- Zaimplementuj odwracanie kolejności elementów na liście.
- Zaimplementuj szukanie określonej wartości w liście.
- Zaimplementuj usuwanie określonej wartości w liście.
- Zaimplementuj operacje na FIFO
- Zaimplementować dla listy cyklicznej:
  łączenie dwóch list, szukanie na liście określonej wartości.
- Wykorzystać stos do wyznaczania wyrażeń zapisanych w odwrotnej notacji polskiej (ONP).
  `3 4 + 2 *`
  `2 2 + 2 + 10 *`

Drzewa



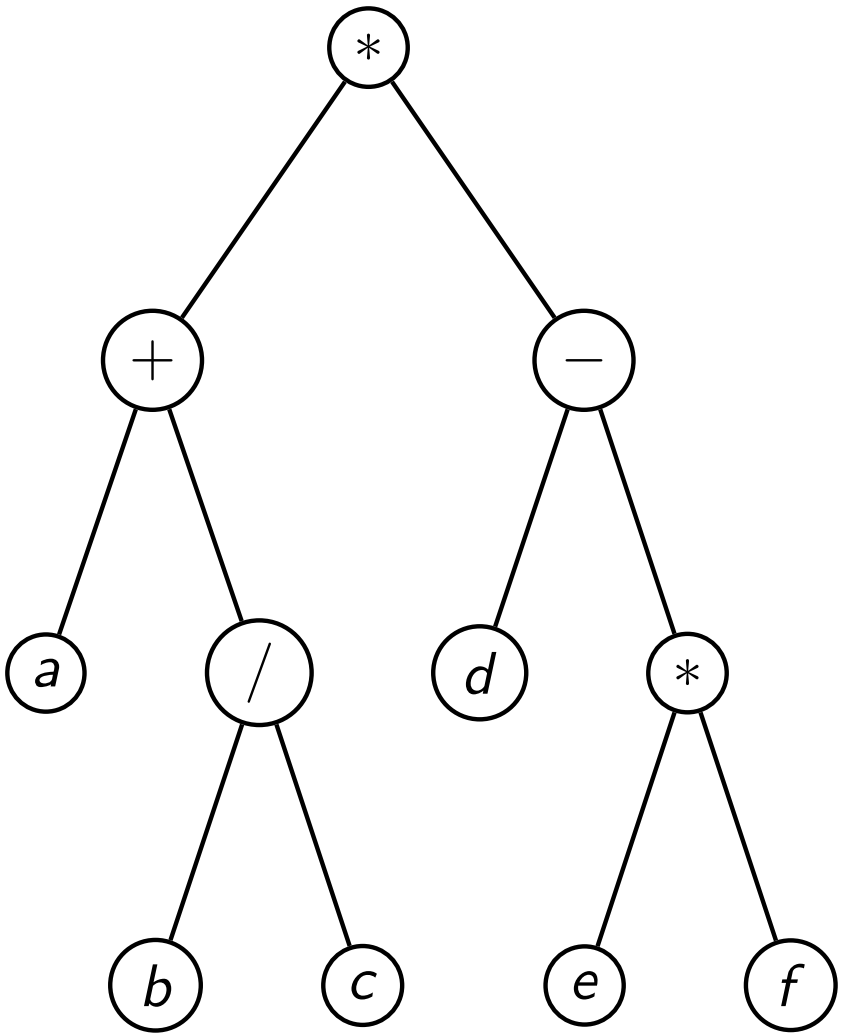



A
B
C
D
E
F
G
I
J
K
J
K
L
M

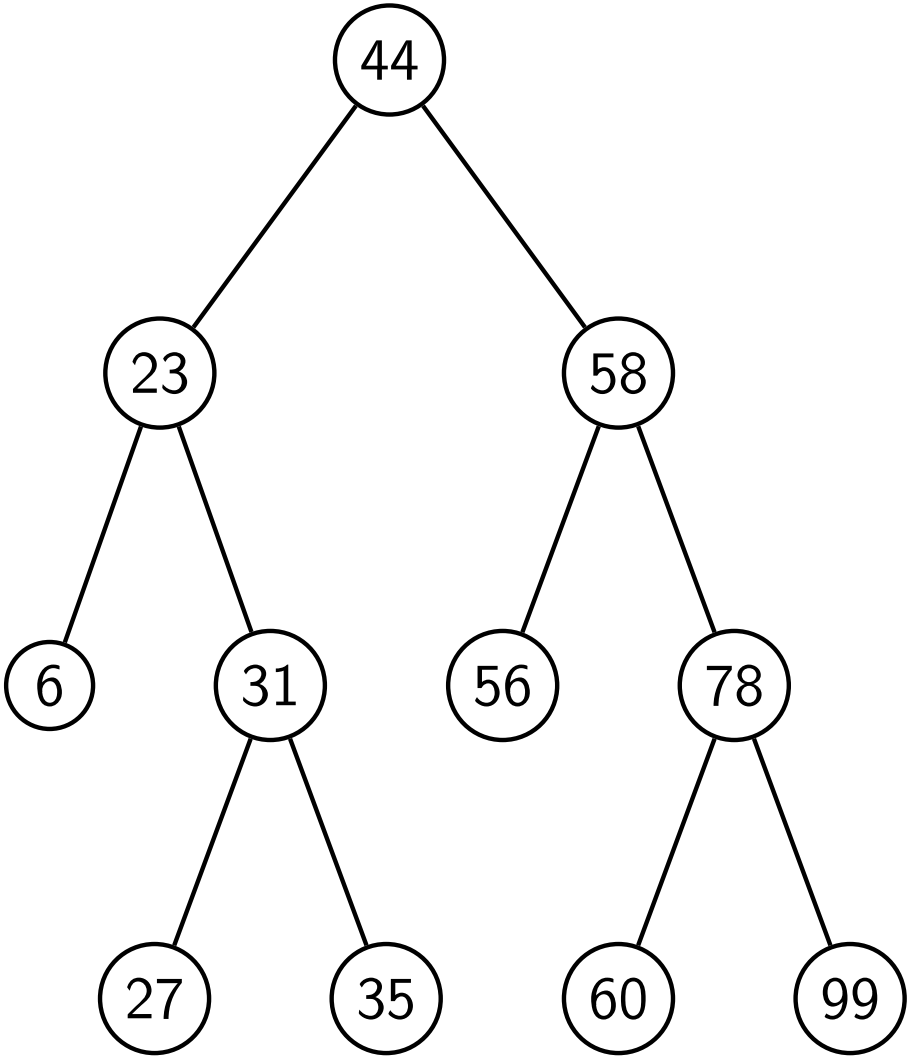
$(a + b/c) * (d - e * f)$

$a\ b\ c\ /\ +\ d\ e\ f\ *\ -\ *$

$*\ +\ a\ /\ b\ c\ -\ d\ *\ e\ f$



Drzewo z porządkiem



Reprezentacja drzewa

Drzewo binarne

```
1 struct elDrzewaB
2 {
3   int klucz;
4   struct elDrzewaB *lewy;
5   struct elDrzewaB *prawy;
6 };
7 typedef struct elDrzewaB wDrzewaB;
8 typedef wDrzewaB* drzewo;
```

Drzewo o dowolnej liczbie gałęzi

```
1  // el. listy drzew
2  struct elDM
3  {
4    struct elDrzewaM *klucz;
5    struct elDM *nast;
6  };
7  typedef struct elDM* listaDrzewM;
8  // węzeł drzewa z wieloma dziećmi
9  struct elDrzewaM
10 {
11   int klucz;
12   listaDrzewM dzieci;
13 };
14 typedef struct elDrzewaM *drzewoM;
```

Węzły z info o przodku

```
1 struct elDrzewaB
2 {
3   int klucz;
4   struct elDrzewaB *lewy;
5   struct elDrzewaB *prawy;
6   struct elDrzewaB *ojciec;
7 };
```

Drzewo – słownik z porządkiem

```
1 struct elDrzewaB
2 {
3   int klucz;
4   int licznik;
5   struct elDrzewaB *lewy;
6   struct elDrzewaB *prawy;
7   struct elDrzewaB *ojciec;
8 };
```

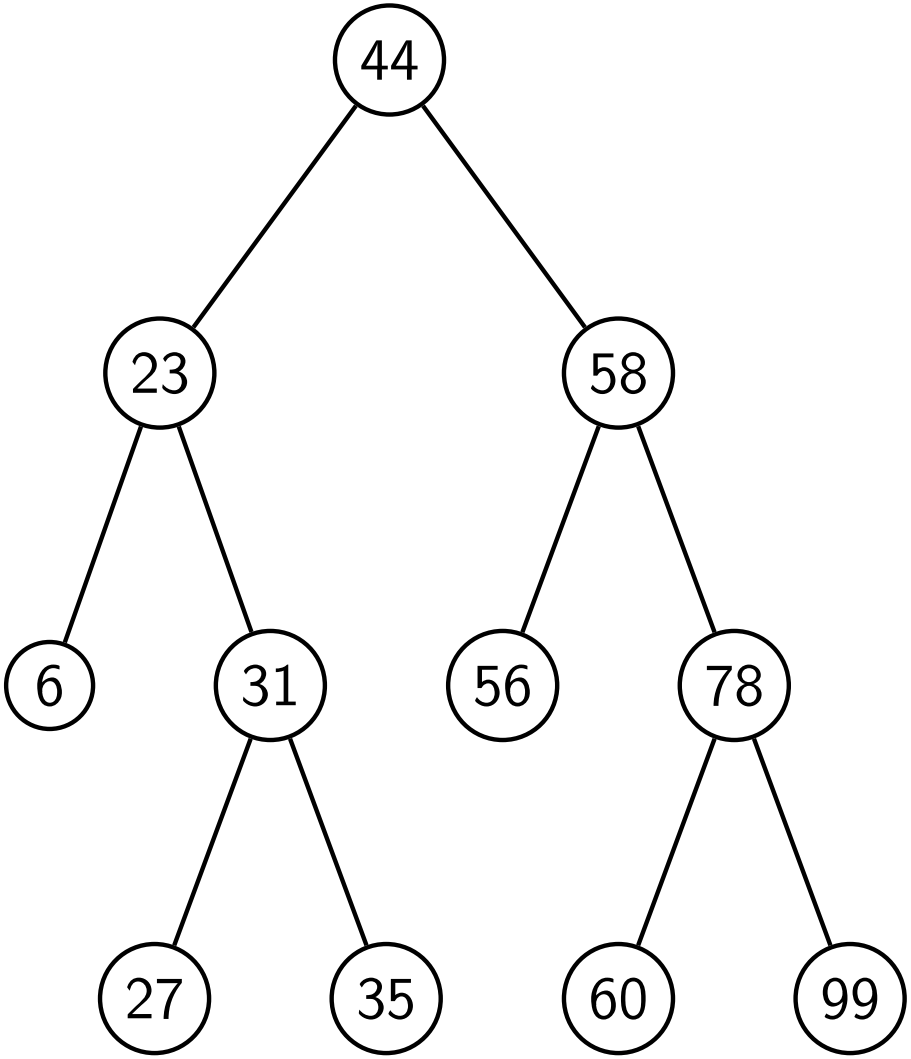
Podstawowe operacje słownikowe

- Dopisz
- Znajdź
- Usuń
- Maximum, Minimum,
- Poprzednik, Następnik

Drukowanie drzewa

```
1  void DrukujDrzewo0(drzewo d, int glebokosc)
2  {
3     if (d == 0) return;
4     DrukujDrzewo0(d->lewy, glebokosc + 1);
5     for (int i = 0; i < glebokosc; i++)
6        putchar(' ');
7     printf ("%d\n", d->klucz);
8     DrukujDrzewo0(d->prawy, glebokosc + 1);
9  }
10 void DrukujDrzewo(drzewo d)
11 {
12    DrukujDrzewo0(d, 0);
13    putchar('\n');
14 }
```

Drzewo z porządkiem



Dodawanie do drzewa

```
1  void DodajD(drzewo *d, int klucz){
2    if (*d == 0){
3      *d = (drzewo)malloc(sizeof(wDrzewaB));
4      (*d)->klucz = klucz;
5      (*d)->licznik = 1;
6      (*d)->lewy = (*d)->prawy = 0;
7    }
8    else if (klucz < (*d)->klucz)
9      DodajD(&(*d)->lewy, klucz);
10   else if (klucz >(*d)->klucz)
11     DodajD(&(*d)->prawy, klucz);
12   else
13     (*d)->licznik++;
14 }
15 int main(int argc, char* argv []){
16   drzewo _d = 0, *w;
17   DodajD(&_d, 44);
18   DrukujDrzewo(_d);
19 }
```

Szukanie w drzewie

```
drzewo* ZnajdzD(drzewo *d, int klucz)
{
  if (*d == 0) return 0;
  if (klucz < (*d)->klucz)
    return ZnajdzD(&(*d)->lewy, klucz);
  else if (klucz >(*d)->klucz)
    return ZnajdzD(&(*d)->prawy, klucz);
  else
    return d;
}

int main(int argc, char* argv[]){
  drzewo _d = 0, *w;
  DodajD(&_d, 44); ...
  w = ZnajdzD(&_d, 33);
  if (w != 0)
    printf ("%d\n", (*w)->klucz);
}
```

Zwalnianie drzewa

```
1 void ZwolnijD(drzewo *d)
2 {
3     if (*d == 0) return;
4     ZwolnijD(&(*d)->lewy);
5     ZwolnijD(&(*d)->prawy);
6     free(*d);
7     *d = 0;
8 }
```

Minimum i maksimum drzewa

```
drzewo* MaxD(drzewo *d)
{
  if (*d == 0) return 0;
  while ((*d)->prawy)
    d = &(*d)->prawy;
  return d;
}
drzewo* MinD(drzewo *d)
{
  if (*d == 0) return 0;
  while ((*d)->lewy)
    d = &(*d)->lewy;
  return d;
}
```

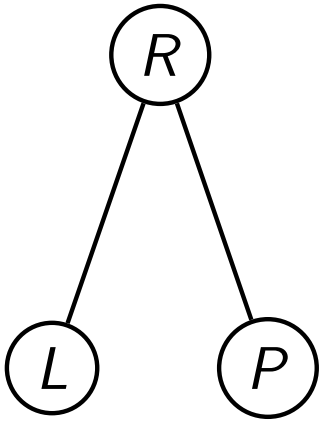
Usuwanie z drzewa

```
1  void UsunD(drzewo *d, int klucz)
2  {
3    drzewo *do_usuniecia, us;
4    if (*d == 0) return;
5    d = ZnajdzD(d, klucz);
6    if (d == 0) return;
7    if ((*d)->licznik > 1)
8    {
9      (*d)->licznik--;
10     return;
11   }
12   ...
```

```
13    ...
14    if ((*d)->lewy == 0 || (*d)->prawy == 0)
15      do_usuniecia = d;
16    else
17    {
18      do_usuniecia = MaxD(&(*d)->lewy);
19      (*d)->klucz = (*do_usuniecia)->klucz;
20      (*d)->licznik = (*do_usuniecia)->licznik;
21    }
22    us = *do_usuniecia;
23    if ((*do_usuniecia)->lewy == 0)
24      *do_usuniecia = (*do_usuniecia)->prawy;
25    else
26      *do_usuniecia = (*do_usuniecia)->lewy;
27    free(us);
28  }
```

Zadania

- Zaimplementuj inne funkcje drzewa/słownika w j. C.
- Napisać algorytm, który dla N liczb wejściowych zbuduje drzewo możliwie dokładnie wyważone.
- Napisz funkcje wyświetlające drzewo w układach (RLP – wzdłużny), (LRP – poprzeczny), (LPR – wsteczny).



- Zaproponować algorytm wyliczania wszystkich możliwych drzew-wyrażeń z zastosowaniem nawiasów dla dowolnego wyrażenia postaci:

$$L_1 \; op_1 \; L_2 \; op_2 \; \ldots \; op_{n-1} \; L_n,$$

np. dla:

$$a + b/c * d - e * f$$

- j.w., lecz tylko tych wyrażeń, które spełniają równość:

$$L_1 \; op_1 \; L_2 \; op_2 \; \ldots \; op_{n-1} \; L_n = p,$$

np.:

$$6 * 8 + 20 : 4 - 2 = 58$$

$$3248 : 16 - 3 * 315 - 156 * 2 = 600$$

- Napisz funkcję szukającą węzeł o wskazanym kluczu w drzewie bez porządku (rekurencyjnie).
- j.w. lecz nierekurencyjnie, ale wykorzystując strukturę węzła ze wskaźnikiem na ojca.

- Napisz procedury wyznaczające: wartość minimalną, maksymalną, poprzednią i następną dla drzew binarnych z porządkiem.

Grafy

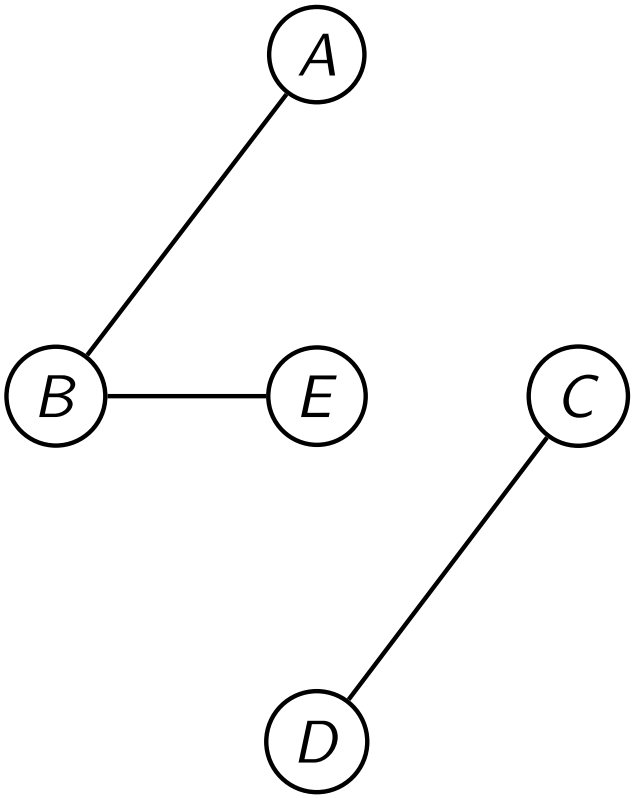
$$G = (V, E)$$

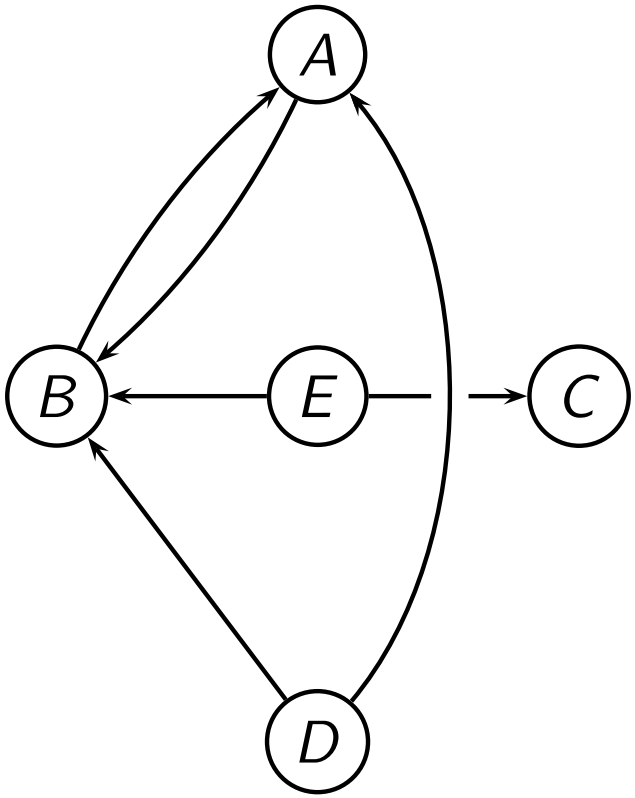
V — zbiór wierzchołków

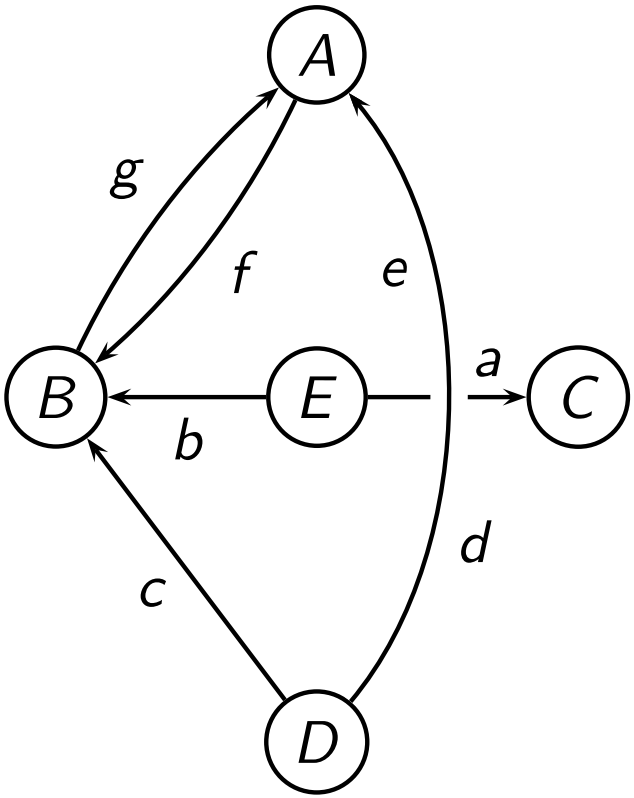
E — zbiór krawędzi (elementami są pary wierzchołków)

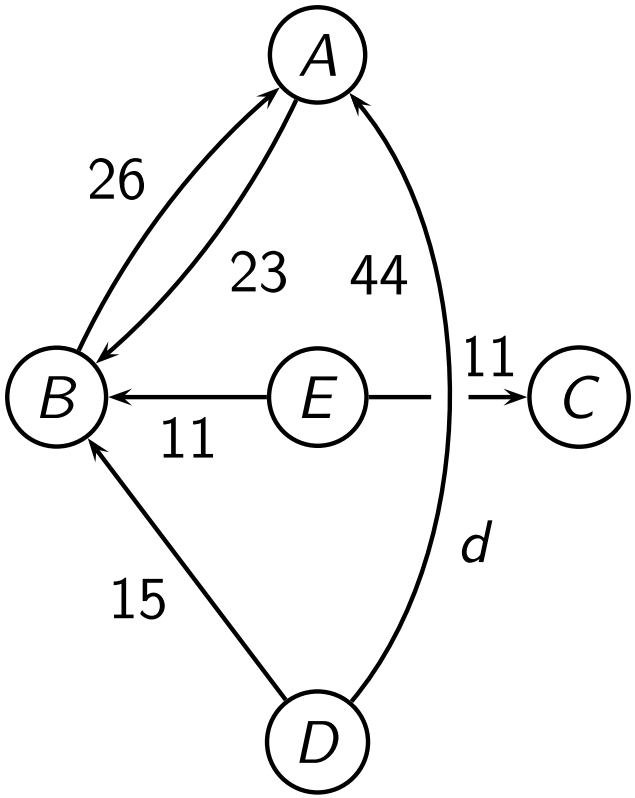
Grafy:

- Zorientowane — krawędzie są parami uporządkowane (v, w), tj, istnieje połączenie v z w.
- Niezorientowane — krawędź (v, w) oznacza połączenie v z w i w z v


A
B
E
C
D


A
B
E
C
D

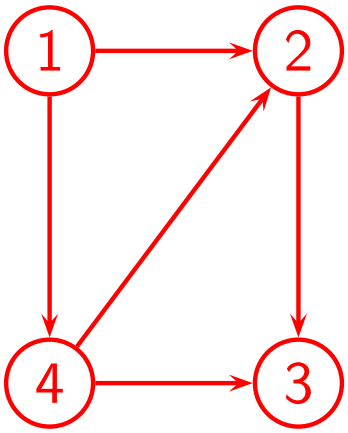




$G = (V, E)$

$V = \{v_i \mid i = 1, \ldots, m\}$ $\quad$ $E = \{k_i = (\langle v, w \rangle, d_k) \mid k = 1, \ldots, n\}$

Reprezentacje grafów



- Macierz sąsiedztwa, $M[i,j] = 0/1$:

$$\begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{matrix}$$

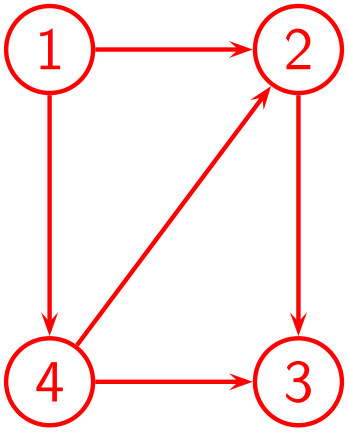
- Listy sąsiedztwa:

W1: [2 | •] → [4 | •]

W2: [3 | •]

W3: ⋮

W4: [2 | •] → [3 | •]



- Reprezentacja tablicowa

1		5
2		7
3		0
4		8
5	2	6
6	4	0
7	3	0
8	2	9
9	3	0

Terminologia około-grafowa

- v jest sąsiadem w, gdy istnieje krawędź (v, w)
- Stopień wierzchołka: liczba sąsiadów
- Droga/ścieżka: ciąg krawędzi – $(v_1, v_2), (v_2, v3), \ldots, (v_{k-1}, v_k)$
- Cykl: droga z pewnego wierzchołka v do v (musi składać się przynajmniej z jednej krawędzi!)
- Graf spójny: istnieje ścieżka pomiędzy każdą parą wierzchołków
- Drzewo: graf spójny bez cykli!
- Klika grafu $G = (V, E)$: podzbiór $V' \subseteq V$ taki, że dla każdej pary wierzchołków $p, q \in V'$ istnieje krawędź $(p, q) \in E$.

Złożoność

Złożoność — *rozmiar* zadania

Złożoność:

- Czasowa
- Pamięciowa

Złożoność czasowa: czas pracy programu wyrażony jako funkcja rozmiaru zadania.

Złożoność pamięciowa: zajętość pamięci przez program wyrażony jako funkcja rozmiaru zadania.

Definicja

Złożoność czasowa programu wykorzystującego $g(n)$ jednostek czasowych jest $O(f(n))$, jeśli istnieje stała c i

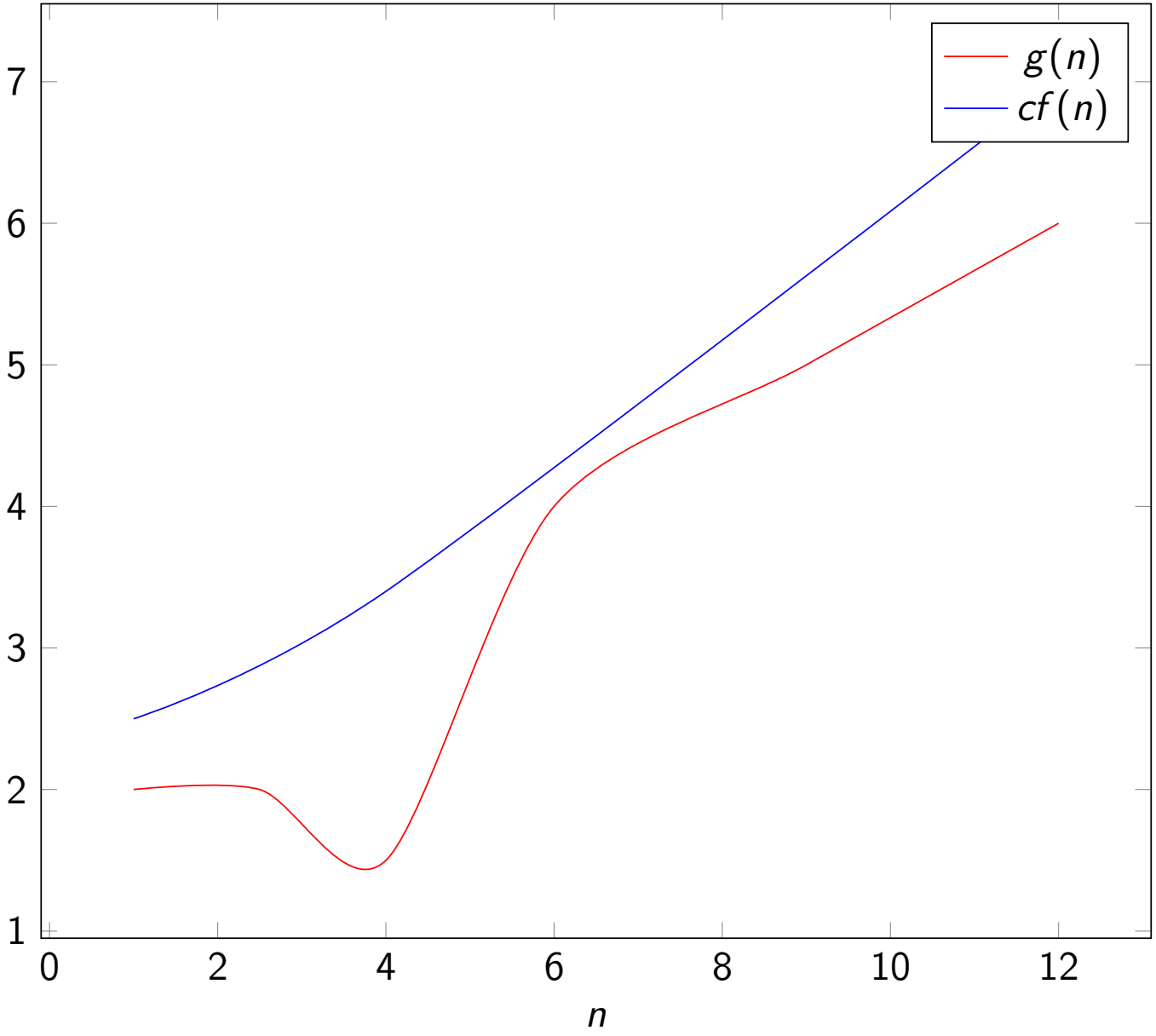
$$g(n) \leqslant cf(n),$$

dla prawie wszystkich n (wszystkich z wyjątkiem skończonego zbioru wartości n).

$O(f(n))$ jest rodziną funkcji spełniających powyższą definicję.

$g(n) = 3n^2 \quad \Rightarrow \quad O(n^2)$
$g(n) = .01n \quad \Rightarrow \quad O(n)$
$g(n) = n + 44n^3 + n^{1/2} \quad \Rightarrow \quad O(n^3)$

$O(f(n))$	ma złożoność $f(n)$	jest rzędu $f(n)$


$g(n)$
$cf(n)$
7
6
5
4
3
2
1
0
2
4
6
8
10
12
n

$g(n) = 1024 \quad \Rightarrow \quad O(1)$
$g(n) = 0.25 \quad \Rightarrow \quad O(1)$
$g(n) = n + \frac{1}{n} \quad \Rightarrow \quad O(n)$
$g(n) = n + \log n \quad \Rightarrow \quad O(n)$

Prawidłowy jest też zapis:
$\frac{1}{n} = O(1)$ (odpowiednik matematycznego $\in$ w: $\frac{1}{n} \in O(1)$)
$5n = O(n)$
$5n = O(n^2)$

$5n + O(n) = O(n)$
$5n + O(n) = O(n^2)$
$5n + O(n^2) \neq O(n)$

$n^4 \neq \frac{1}{4}n^4 + O(n^3)$

$g(n) = n^3 + 5n^2 - 22n = O(n^3)$

$n + 1/\log n = O(n)$

$n^2 + O(n) = O(n^2)$
$n^2 + c \cdot O(n) = O(n^2)$

$n^2 + O(n \log n) = O(n^2)$
$\frac{n(n+1)}{2} = O(n^2)$
$\frac{n(n+1)}{2} - 0.5n^2 = O(n)$
$\frac{n(n+1)}{2} - 0.3n^2 = O(n^2)$
$n + 2 \cdot O(n^2) = O(n^2)$
$n + \log n \cdot O(n^2) = O(n^2 \log n)$
$\frac{\log n}{n} = O(1)$

$$g(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 = O(n^k) \qquad \text{dla } a_k > 0$$

$33 \cdot 2^n = O(2^n)$
$2^{n+2} + n^6 = O(2^n)$

$2^n = O(3^n)$
$3^n \neq O(2^n)$
$\sqrt{n} = O(\sqrt{n})$
$\sqrt{n} = O(n)$
$\sqrt[3]{n} = O(\sqrt{n})$

Granice rozmiaru zadania vs. złożoność

Złożoność	Max roz. zadania		
	$1s$	$1min$	$1h$
n	1000	$6 \cdot 10^4$	$3.6 \cdot 10^6$
$n \log n$	140	4893	$2.0 \cdot 10^5$
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Wpływ 10-krotnego przyśpieszenia komputera na maksymalny rozmiar zadania

Złożoność	$R.z.przed$	$R.z.po$
n	s_1	$10s_1$
$n \log n$	s_2	$10s_2$(dla dużych s_2)
n^2	s_3	$3.16s_3$
n^3	s_4	$2.15s_4$
2^n	s_5	$s_5 + 3.3$

Złożoność c.d.

Złożoność pesymistyczna (z. czasowa) — z. najgorszego przypadku.

Złożoność średnia — średnia z. po wszystkich danych rozmiaru n.

Z. pesymistyczna nierzadko równa jest z. średniej.

O, Θ i Ω

O — asymptotyczna granica górna

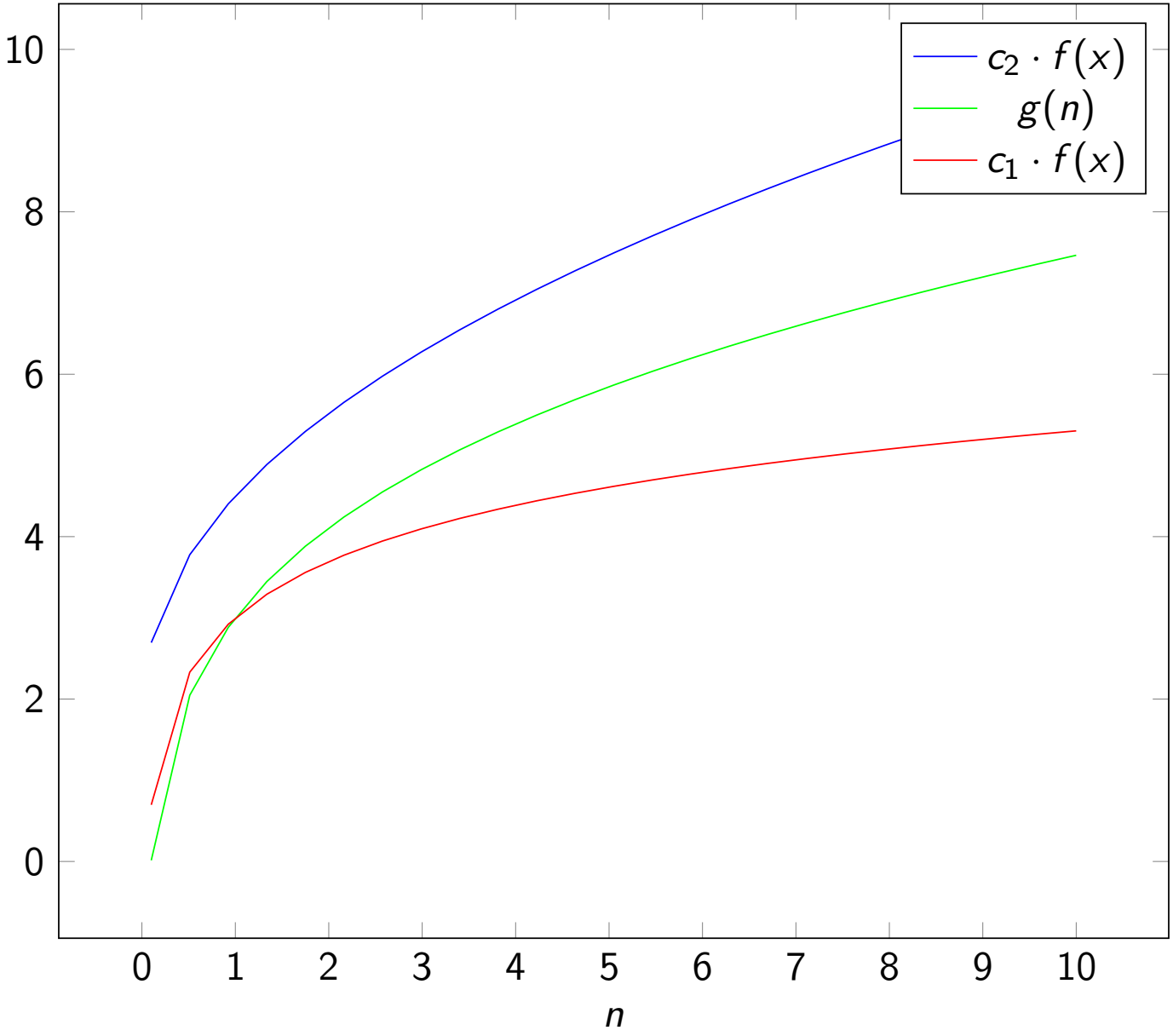
$$O(f(n)) = \{g(n) : \exists_{c>0,n_0>0}\ \forall_{n\geqslant n_0} \quad 0 \leqslant g(n) \leqslant cf(n)\}$$

Ω — asymptotyczna granica dolna

$$\Omega(f(n)) = \{g(n) : \exists_{c>0,n_0>0}\ \forall_{n\geqslant n_0} \quad 0 \leqslant cf(n) \leqslant g(n)\}$$

Θ — asymptotycznie dokładne oszacowanie

$$\Theta(f(n)) = \{g(n) : \exists_{c_1>0,c_2>0,n_0>0}\ \forall_{n\geqslant n_0} \quad 0 \leqslant c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)\}$$


10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
9
10
n
$c_2 \cdot f(x)$
$g(n)$
$c_1 \cdot f(x)$

Klasy $\mathcal{P}$ i $\mathcal{NP}$

Klasa $\mathcal{P}$ gdy algorytm ma złożoność wielomianową.

Klasa $\mathcal{NP}$ gdy algorytm nie ma złożoności wielomianowej.

Kryteria kosztów

- Jednorodne kryterium kosztów
  zakłada się, że jedna instrukcja wykonywana jest w jednej jednostce czasu
  zakłada się, że każdy rejestr wymaga jednej jednostki pamięci

$$l(i) = 1$$

- Logarytmiczne kryterium kosztów
  [uwzględnienie długości słowa komputera]

$$l(i) = \begin{cases} \lfloor \log |i| \rfloor + 1 & i > 0 \\ 1 & i = 0 \end{cases}$$

Złożoność Kołmogorowa i Levina

Złożoność Kołmogorowa

$$K_U(x) = \min_p \{ \ l(p) : \ \text{program } p \text{ prints } x\}. \tag{1}$$

Złożoność Levina

$$Kt_U(x) = \min_p \{ \ l(p) + \log t(p,x) : \ \text{program } p \\ \text{prints } x \text{ in } t(p,x)\}. \tag{2}$$

Warunkowa złożoność Levina

$$Kt'_U(w|x,\phi) = \min_p \{ l(p) + \log t(p,x) : \ \text{program } p \\ \text{prints } w \text{ and test } \phi(w) = x \text{ in } t(p,x)\}. \tag{3}$$

Zadania

- Wyznaczyć złożoność różnych funkcji dotyczących operowania na listach i drzewach.
- Jaka jest złożoność wyznaczania wektora o maksymalnej normie ze zbioru $\mathcal{S} = \{\mathbf{x}_i : i = 1, \ldots, n,\ \mathbf{x} \in \mathcal{R}^d\}$?

Zadanie sortowania

Wejście: zbiór liczb

$$a_1, \ldots, a_n$$

Wyjście: ciąg liczb spełniający

$$a_{i_1} \leqslant \ldots \leqslant a_{i_n}$$

Algorytmy sortowania i ich złożoności

Trudno gorzej niż $O(n^2)$

Aż do ... $O(n)$!!!

Sortowanie za pomocą: tablic, drzew, kolejek

$O(n^2)$ — przykłady

- Sortowanie przez proste wstawianie
- Sortowanie przez wstawianie połówkowe (ZADANIE)
- Sortowanie przez proste wybieranie
- Sortowanie bąbelkowe

```
1  void ProsteWstawianie(int a[], int n)
2  {
3    int i, j, x;
4    for (i = 1; i < n; i++)
5    {
6      if (a[i] < a[0]){ x = a[0]; a[0] = a[i]; }
7      else x = a[i];
8      for (j=i-1; x < a[j]; j--)
9        a[j + 1] = a[j];
10     a[j + 1] = x;
11   }
12 }
```

Wynik pośredni dla *i*-tej iteracji pętli *for*:

$a_1, \ldots, a_{i-1}$ (posortowane) $\quad a_i, \ldots, a_n$ (nieposortowane)

```
1  void ProsteWybieranie(int a[], int n)
2  {
3    int i, j, k, x;
4    for (i = 0; i<n - 1; i++)
5    {
6      k = i; x = a[i];
7      for (j = i + 1; j < n; j++)
8        if (a[j] < x) {
9          k = j; x = a[j];
10       }
11     a[k] = a[i]; a[i] = x;
12   }
13 }
```

Najmniejszy na początek, na drugą pozycję najmniejszy z pozostałych, itd.

```
1  void SortowanieBabelkowe(int a[], int n)
2  {
3    int i, j, x;
4    for (i = 1; i<n; i++)
5    {
6      for (j = n - 1; j >= i; j--)
7        if (a[j - 1] > a[j]) {
8          x = a[j - 1]; a[j - 1] = a[j]; a[j] = x;
9        }
10   }
11 }
```

Duże bąbelki szybciej wychodzą na powierzchnię ...

Algorytmy sortujące w $O(n \log n)$

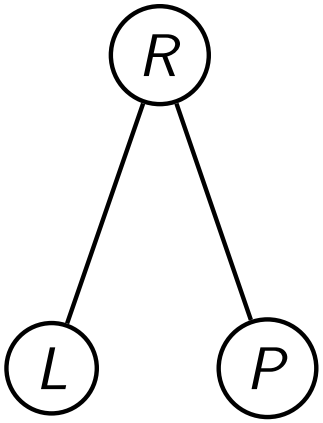
Złożoność średniego przypadku, a pesymistyczna złożoność

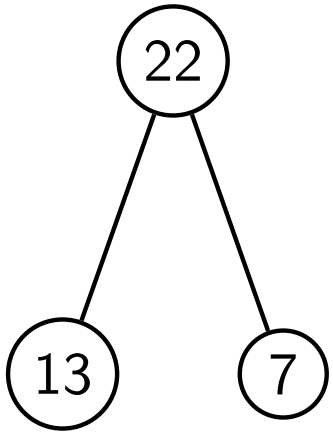
- Sortowanie przez kopcowanie
- Szybkie sortowanie
- Drzewa z porządkiem
- Sortowanie przez łączenie (ZADANIE)

Sortowanie przez kopcowanie

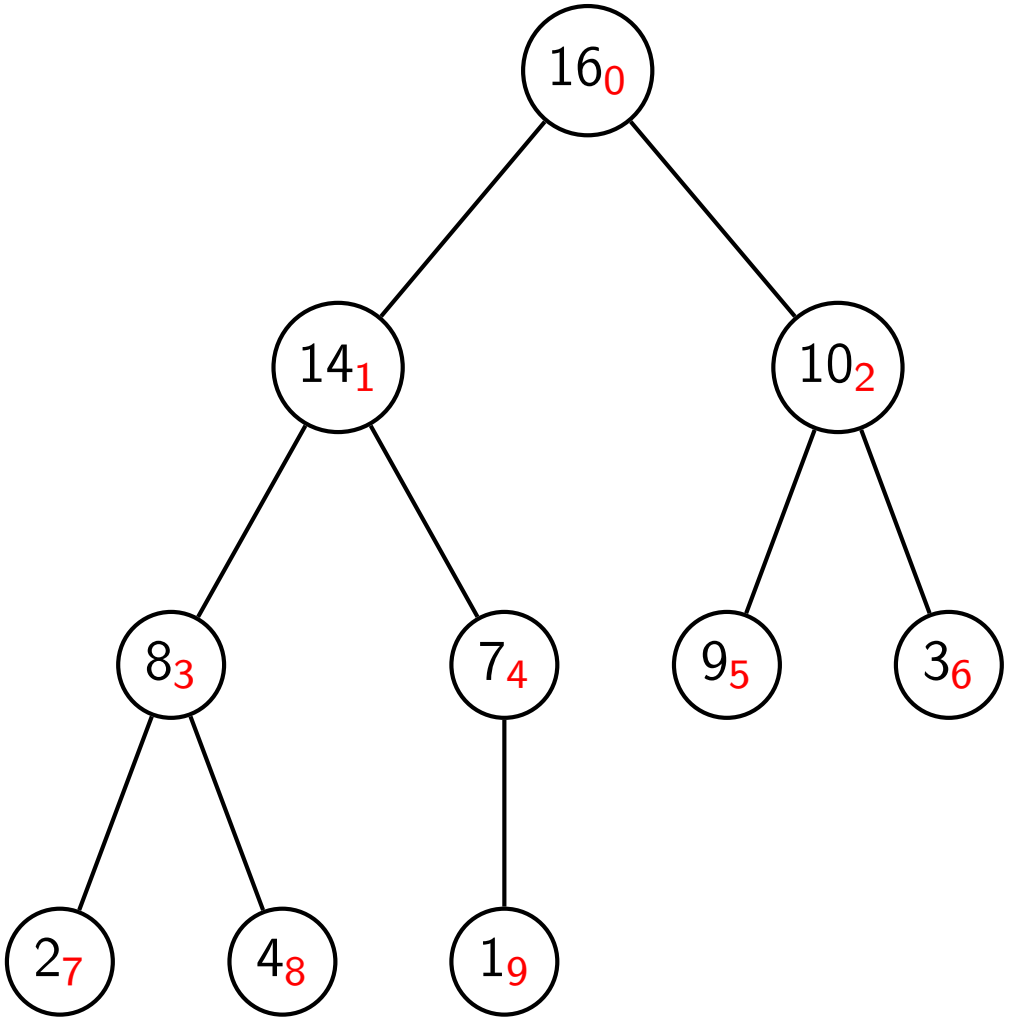
Algorytm *in-situ*, jak poprzednie, ale o złożoności $O(n \log n)$.

Warunek kopca: $R \geqslant L$ i $R \geqslant P$





Kopiec



Kopiec c.d

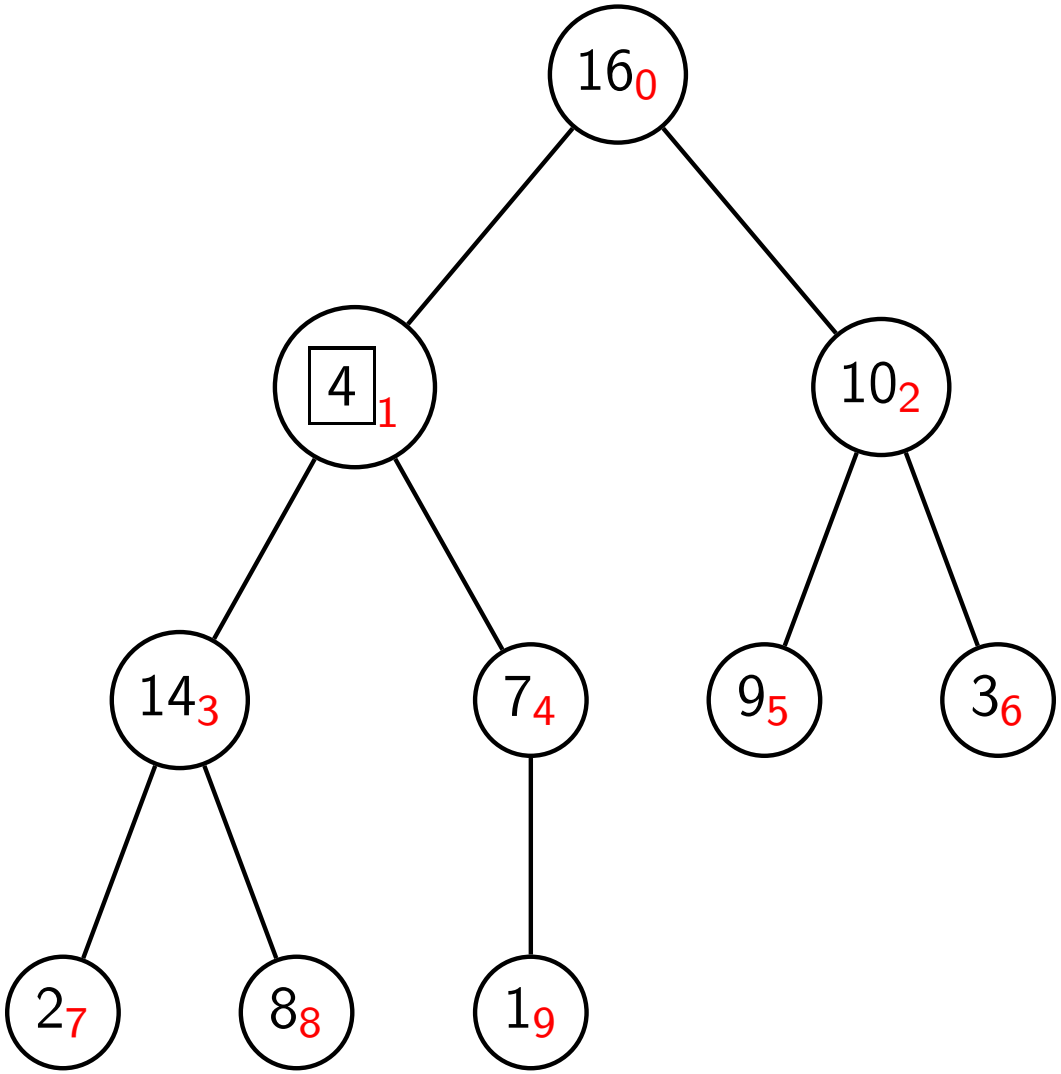
Reprezentacja: drzewo lub tablica

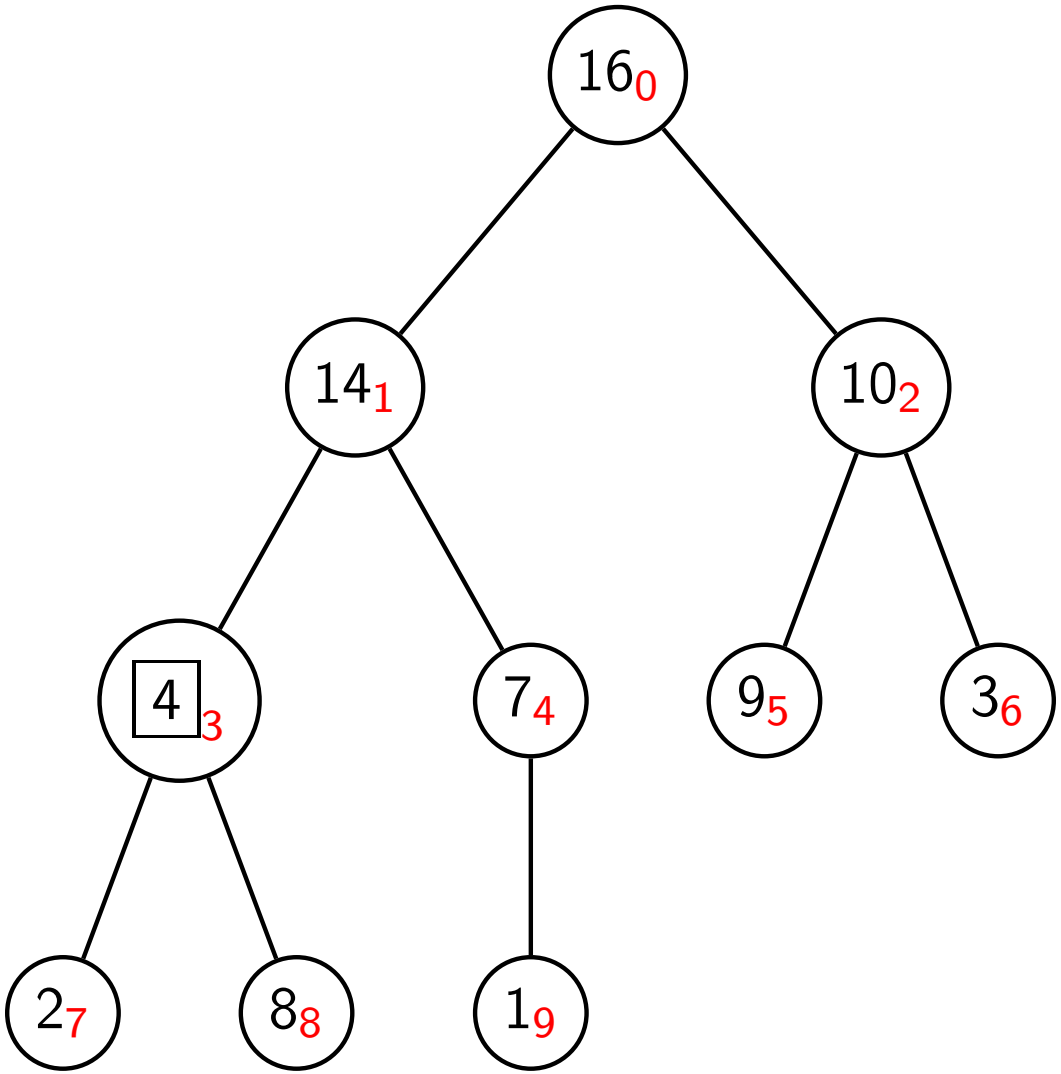
$$PARENT(i) = (i - 1)/2$$

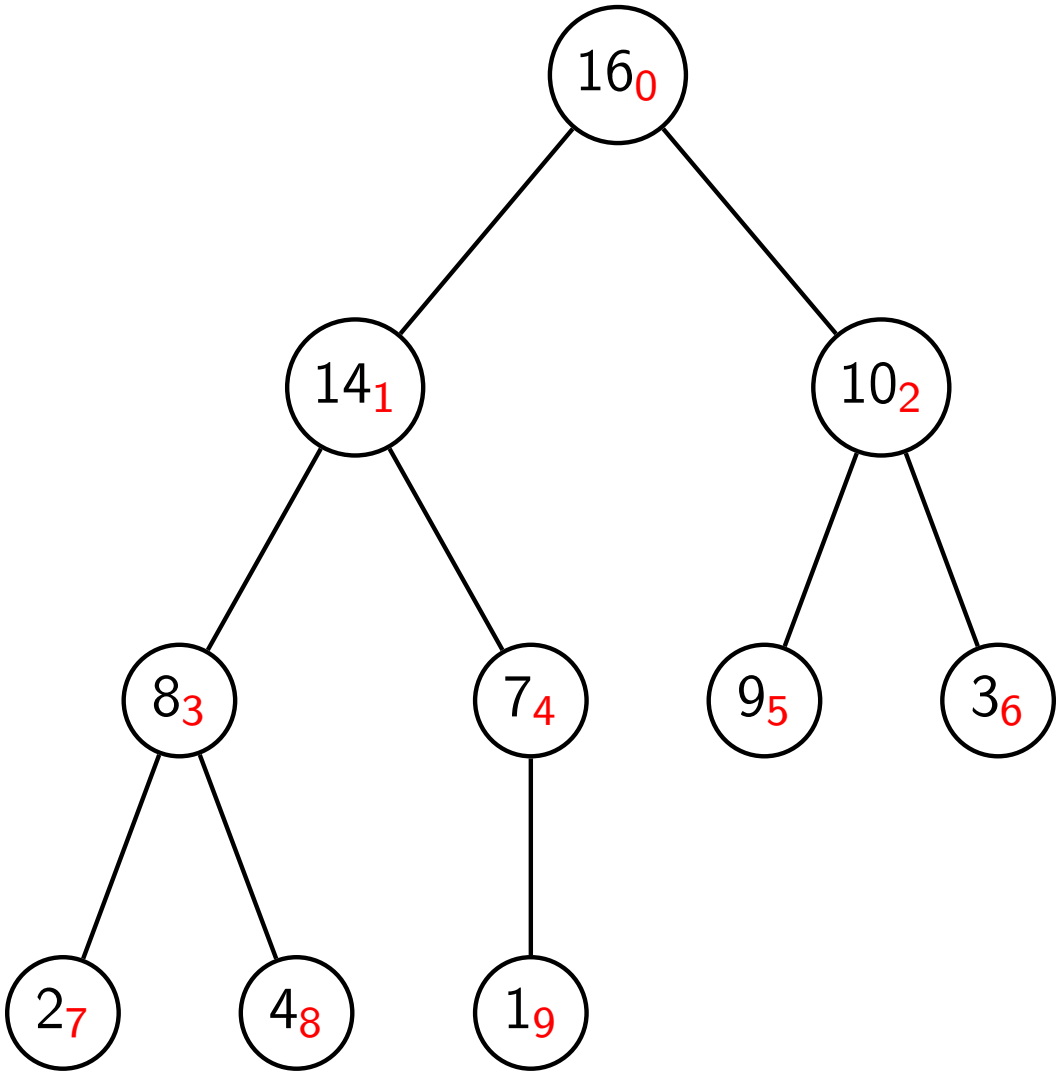
$$LEFT(i) = 2i + 1$$

$$RIGHT(i) = 2i + 2$$

Heapify — $O(\log n)$




16_0
14_1
10_2
4_3
7_4
9_5
3_6
2_7
8_8
1_9

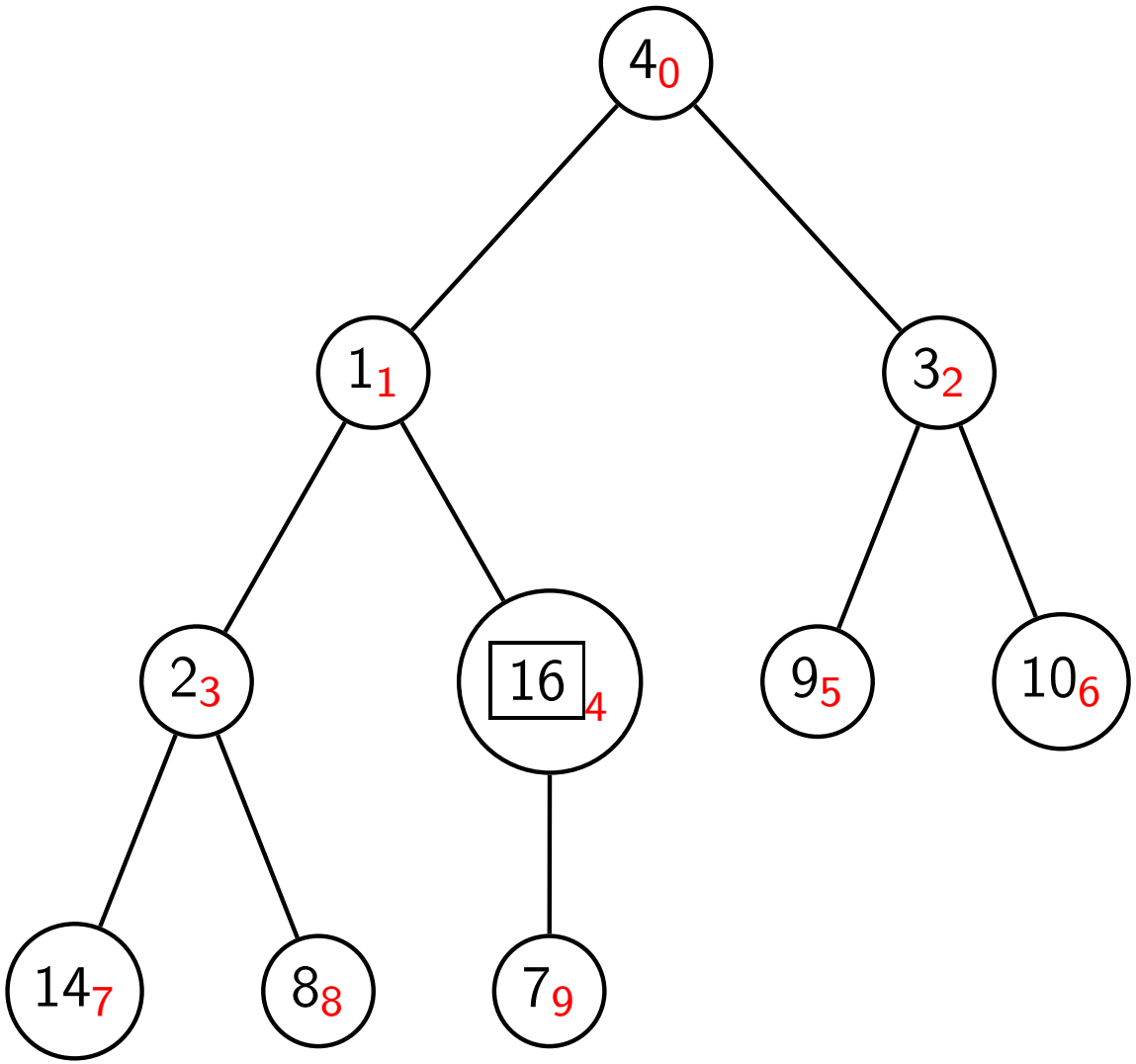

16_0
14_1
10_2
8_3
7_4
9_5
3_6
2_7
4_8
1_9

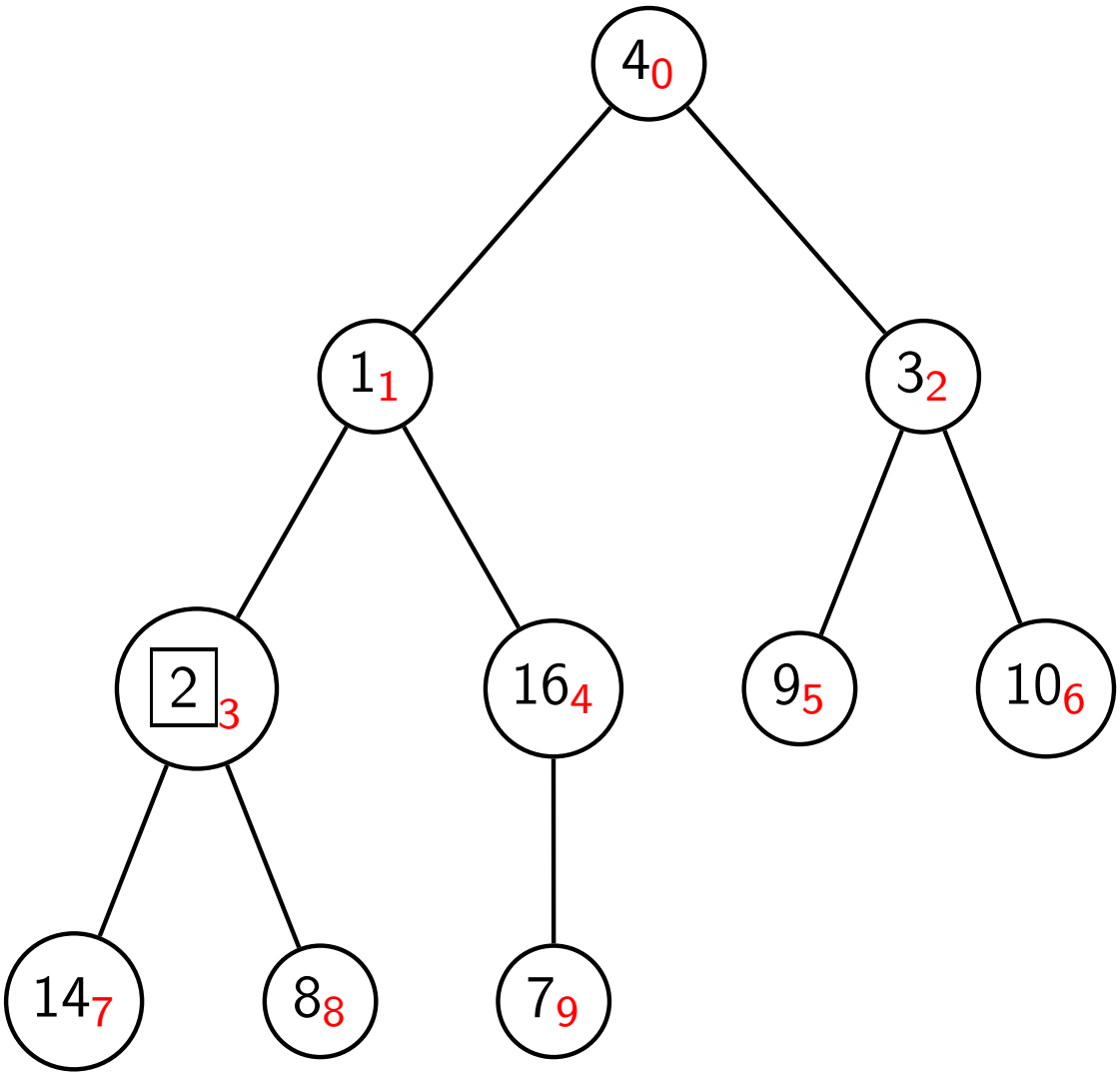
```
1  #define LEFT(i) (2*i + 1)
2  #define RIGHT(i) (2*i + 2)
3  void Heapify(int A[], int i, int heapSize)
4  { int l, r, x, largest;
5    l=LEFT(i); r=RIGHT(i);
6    if (l < heapSize && A[l] > A[i])
7      largest = l;
8    else largest = i;
9    if (r < heapSize && A[r] > A[largest])
10     largest =r;
11   if (largest != i) {
12     x = A[i]; A[i] = A[largest]; A[largest] = x;
13     Heapify(A, largest, heapSize);
14   }
15 }
```

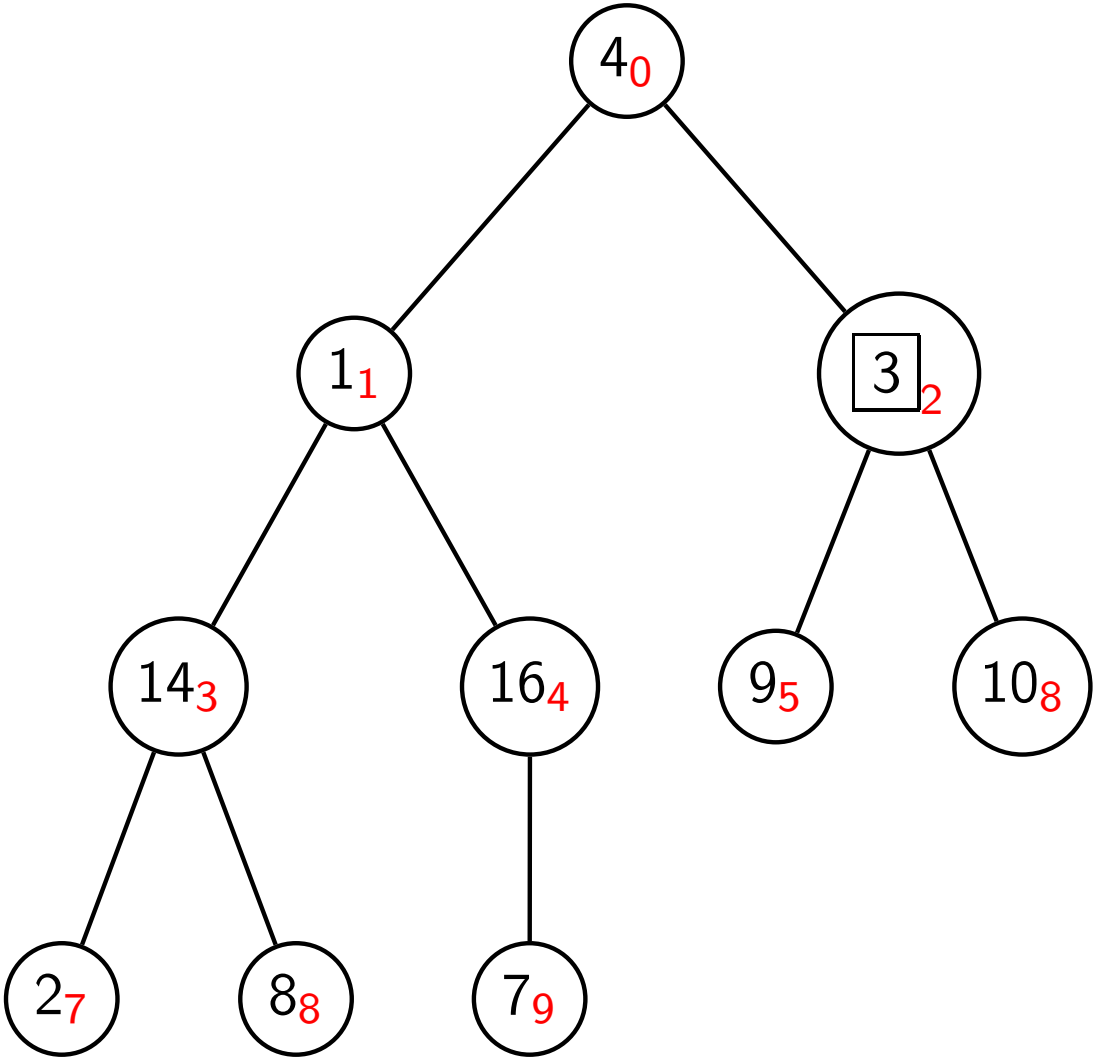
Budowanie kopca

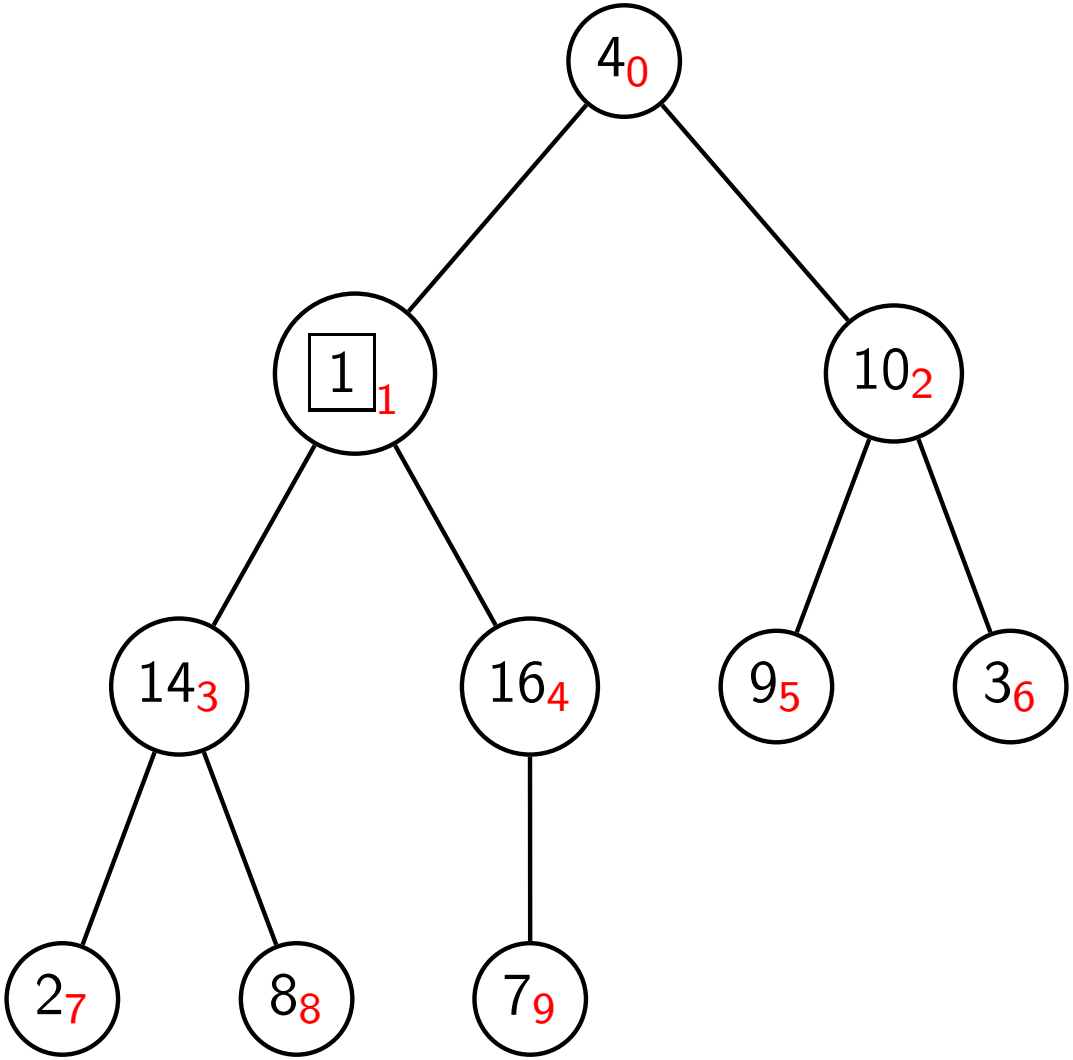
```
1 void BuildHeap(int A[], int n)
2 {
3   int i;
4   for (i = (n - 1) / 2; i >= 0; i--)
5     Heapify(A, i, n);
6 }
```

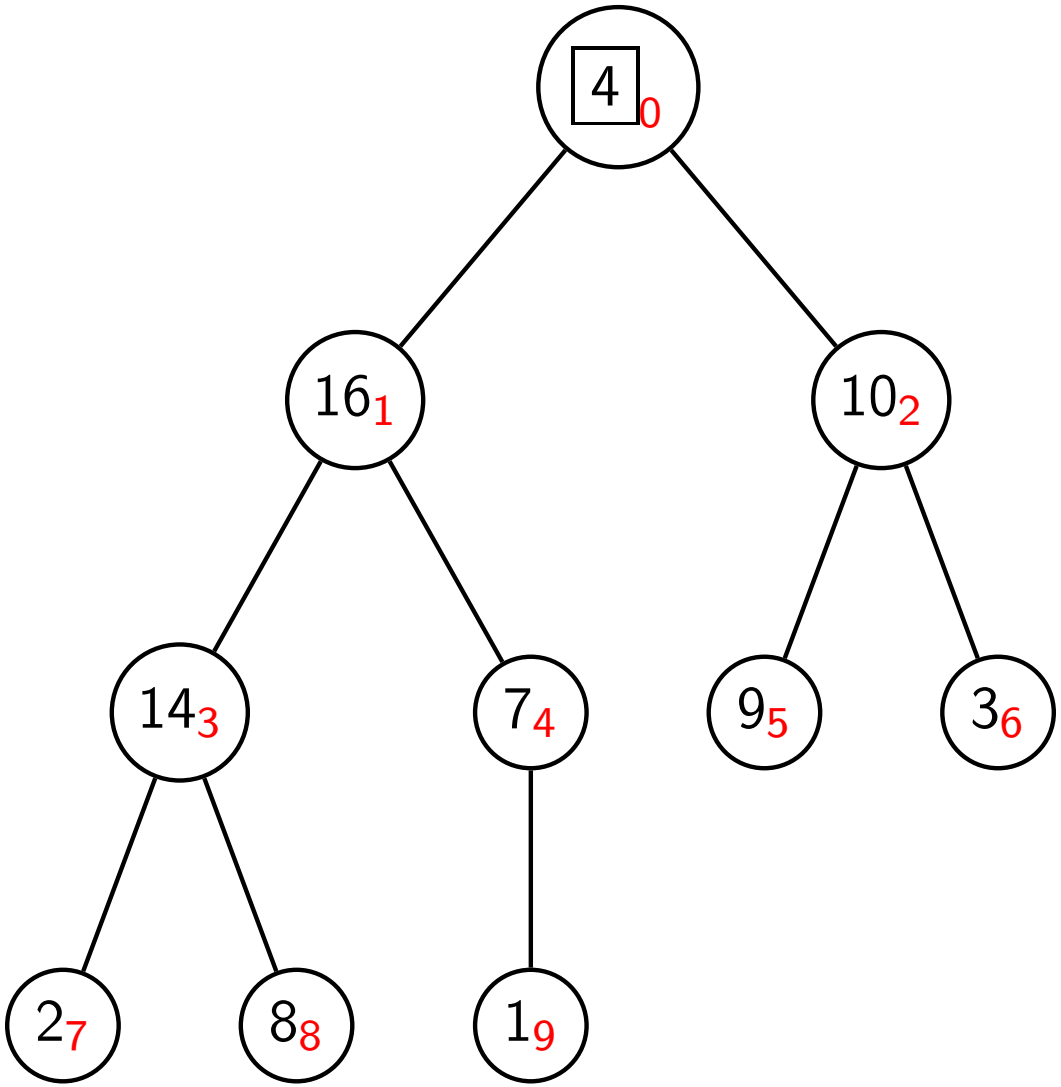
kopce, kopce kopców, ...

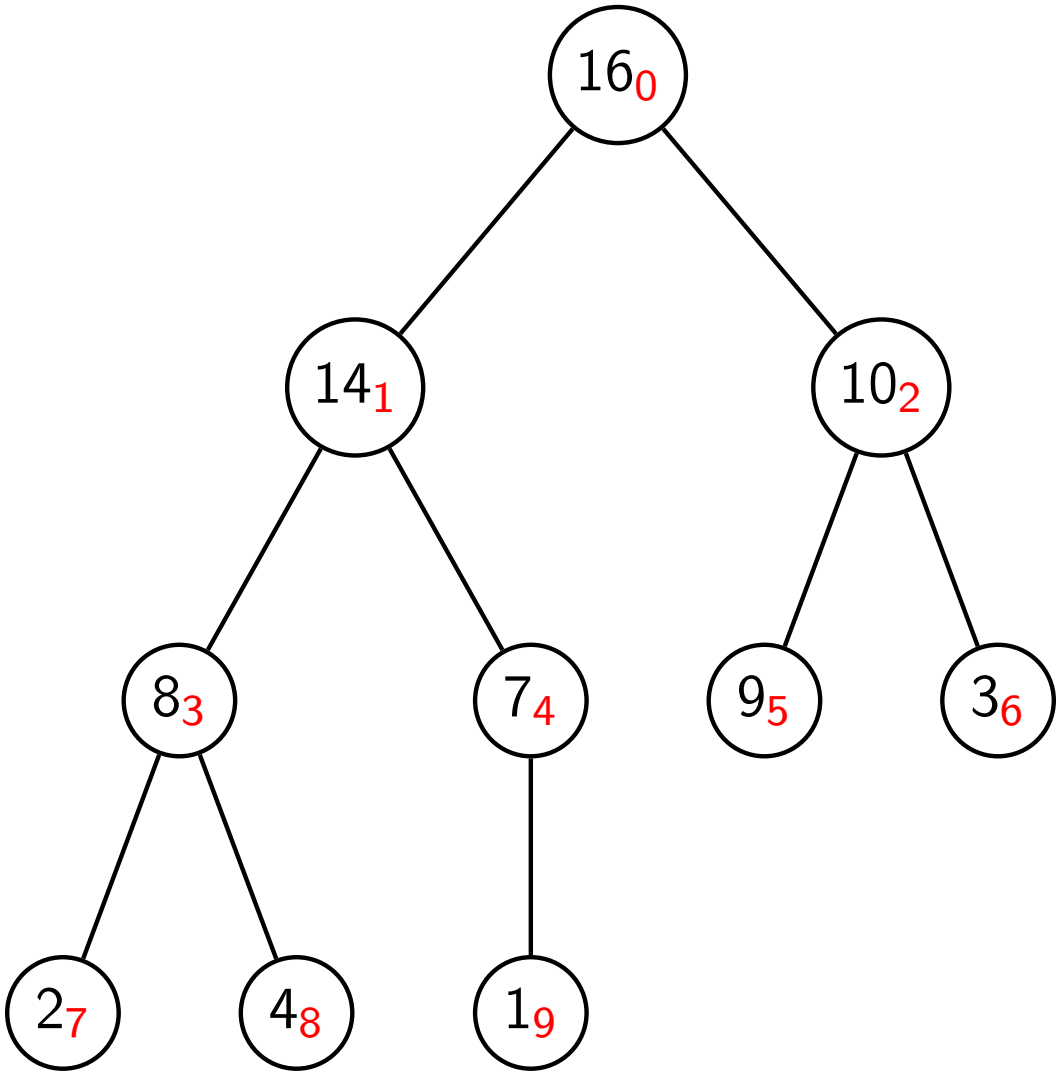

4 0
1 1
3 2
2 3
16 4
9 5
10 6
14 7
8 8
7 9


4 0
1 1
3 2
2 3
16 4
9 5
10 6
14 7
8 8
7 9


4_0
1_1
$\boxed{3}_2$
14_3
16_4
9_5
10_8
2_7
8_8
7_9


4 0
1 1
10 2
14 3
16 4
9 5
3 6
2 7
8 8
7 9


4 0
16 1
10 2
14 3
7 4
9 5
3 6
2 7
8 8
1 9


16_0
14_1
10_2
8_3
7_4
9_5
3_6
2_7
4_8
1_9

Złożoność procedury BuildHeap(A)

- Heapify — $O(h)$ — h wysokość kopca
- n elementowy kopiec ma co najwyżej $\lceil n/2^{h+1} \rceil$ pod-kopców wysokości h
- Całość:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

mamy jednak:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$

ostatecznie:

$$= O(n)$$

Sortowanie przez kopcowanie

- Zbudowanie kopca
- Korzeń = max
- Usuniecie korzenia (na koniec ...) i powtarzanie ...
  [zamiana max i el. ostatnim, i skrócenie tablicy A]

```
1  void HeapSort(int A[], int n)
2  {
3    int i, x, heapSize = n;
4    BuildHeap(A, n);
5    for (i = n - 1; i>0; i--)
6    {
7      x = A[0]; A[0] = A[i]; A[i] = x;
8      Heapify(A, 0, --heapSize);
9    }
10 }
```

Ostateczna złożoność: $O(n \log n)$

Zadania

- Prześledzić na kilku około 10 elementowych tablicach omówione algorytmy sortowania.
- Opracuj algorytm sortowania przez wstawianie połówkowe.
- Opracuj algorytm sortowania przez łączenie.
- Zaproponować wersję iteracyjną rekurencyjnej procedury Heapify.
- Analiza złożoności algorytmów (porównaj liczby porównań, przestawień, poszczególnych pod-etapów różnych algorytmów).
- Prześledź zachowanie się algorytmów sortowania dla najlepszych i najgorszych przypadków.
- Co można powiedzieć o złożonościach O, Ω, Θ dla algorytmów sortowania przez przestawianie, wybór i sortowanie bąbelkowe?

Algorytm szybkiego sortowania (Quicksort)

Dziel i zwyciężaj!

- Dziel na dwie części A i B.
  A ma elementy nie większe od x,
  B ma większe od x.
- Zwyciężaj: sortuj A i B.
- Łączenie A i B.

```
1 void QSort(int A[], int p, int r)
2 {
3     int q;
4     if(p < r) {
5         q=Partition(A,p,r);
6         QSort(A, p,q);
7         QSort(A, q+1,r);
8     }
9 }
```

$$\boxed{A_p, \ldots, A_q \;\Big|\; \leqslant \;\Big|\; A_{q+1}, \ldots, A_r}$$

$$\Downarrow$$

$$\boxed{\boxed{A_p, \ldots, A_q} \;\Big|\; \leqslant \;\Big|\; \boxed{A_{q+1}, \ldots, A_r}} \;\Big|\; \leqslant \;\Big|\; \boxed{\boxed{A_p, \ldots, A_q} \;\Big|\; \leqslant \;\Big|\; \boxed{A_{q+1}, \ldots, A_r}}$$

$$\Downarrow \ldots$$

$$\boxed{A_p \leqslant A_{p+1} \leqslant A_{p+2} \leqslant \ldots \leqslant A_{r-1} \leqslant A_r}$$

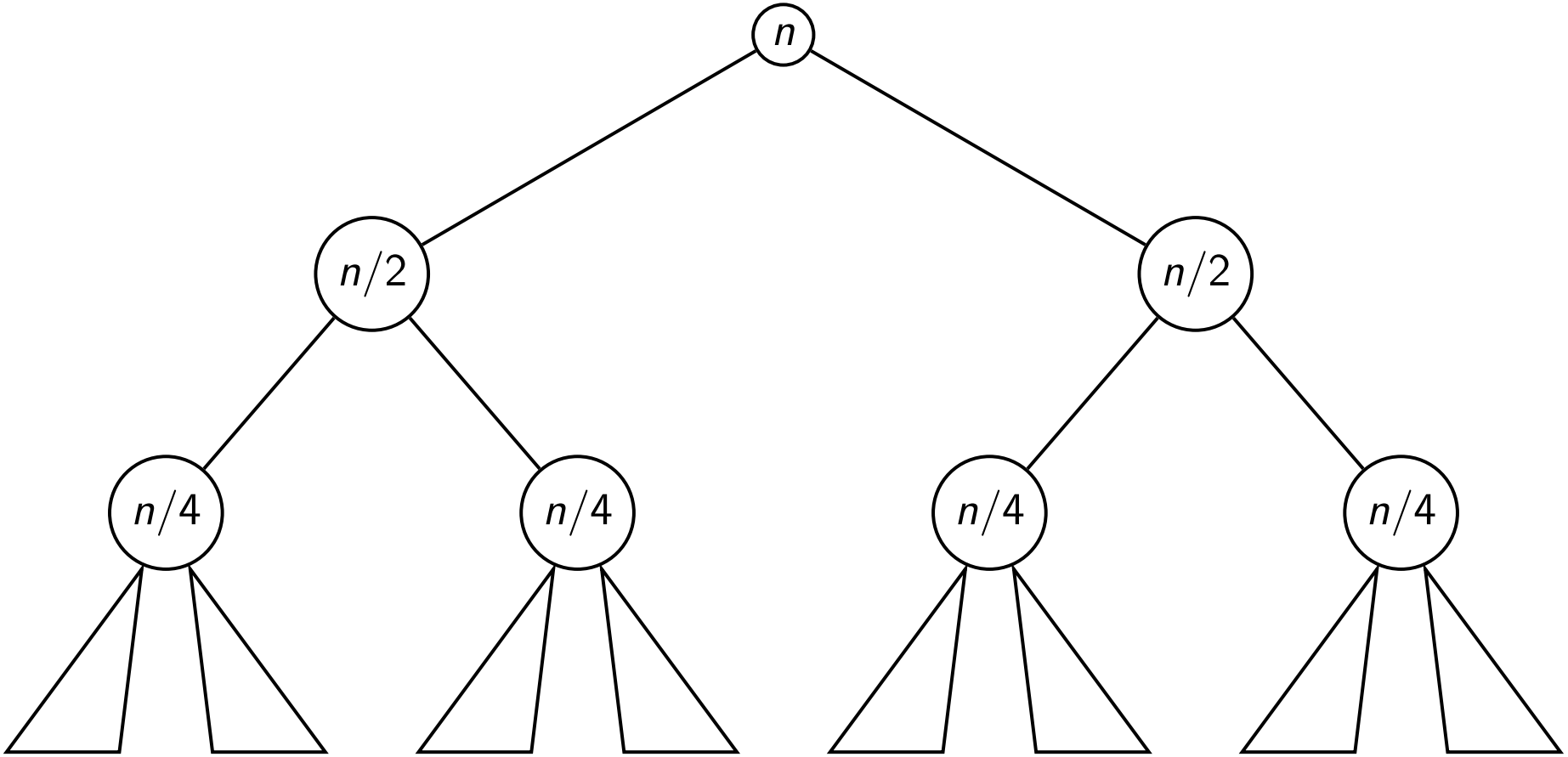
```
int Partition (int A[], int p, int r)
{
  int _r = rand()%(r-p+1)+p, x, i, j;
  x = A[p]; A[p] = A[_r]; A[_r] = x;
  x = A[p];
  i = p-1; j = r+1;
  do {
    do { j--; } while (A[j]>x);
    do { i++; } while (A[i]<x);
    if (i < j) { y = A[i]; A[i] = A[j]; A[j] = y; }
  } while (i < j);
  return j;
}
```

$$A[p,i] \quad \leqslant \quad x \quad \leqslant \quad A[j,r]$$

Złożoność szybkiego sortowania

- Złożoność funkcji *Partition*: $\Theta(n)$
- Najlepszy przypadek: $\Theta(n \log n)$
- Najgorszy przypadek: $\Theta(n^2)$
- Średni przypadek: $\Theta(n \log n)$

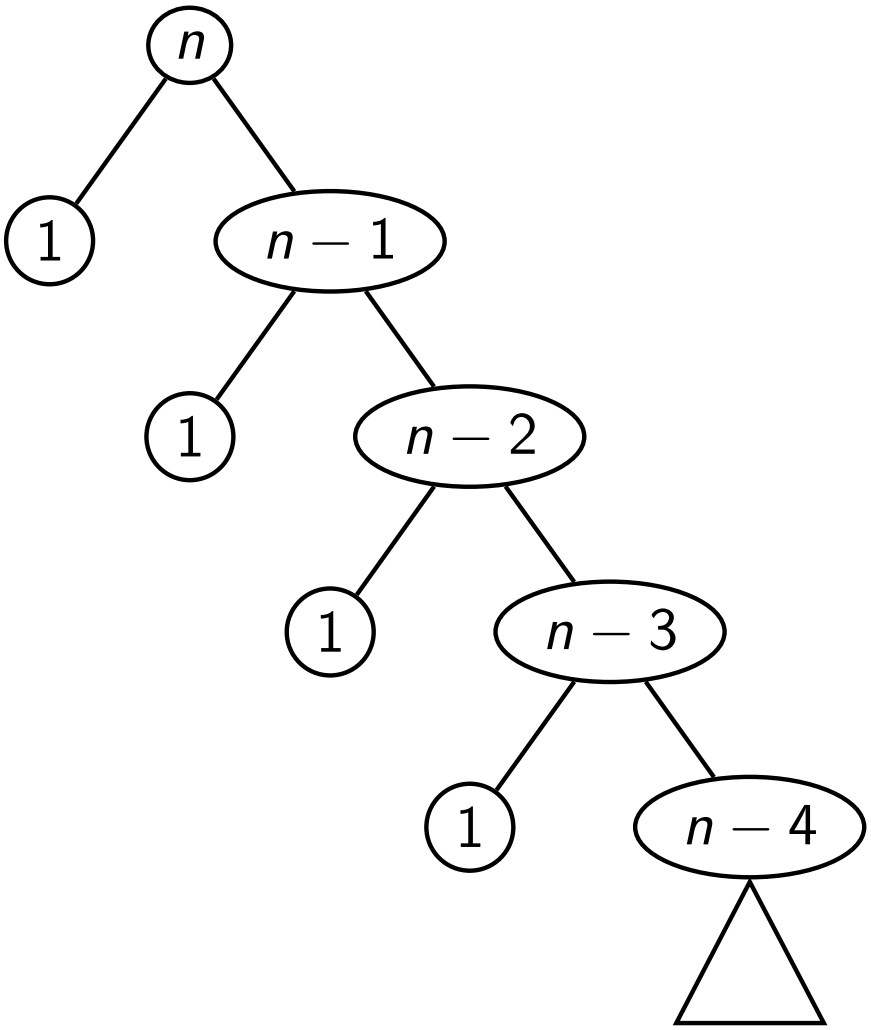
Najlepszy przypadek: $\Theta(n \log n)$



Ponieważ:

- Koszt całościowy każdego z poziomów: $\Theta(n)$
- Liczba poziomów: $\lceil \log n \rceil$
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

Najgorszy przypadek: $\Theta(n^2)$



Ponieważ:

- Koszt całościowy każdego z poziomów: $\Theta(n)$
- Liczba poziomów: n
- $T(n) = T(n-1) + \Theta(n) = \sum_{i=1}^{n} \Theta(i)$
  $T(n) = \Theta\left(\sum_{i=1}^{n} i\right)$
  $T(n) = \Theta(n^2)$

Średni przypadek: $\Theta(n \log n)$

- Niech $T(n)$ będzie średnim czasem sortowania przez QS
- Wtedy czasy wywołań rekurencyjnych to $T(i)$ i $T(n-i)$
- Każde i jest jednakowo prawdopodobne — wybrane *losowo*
- Wykonanie *Partition* kosztuje $\Theta(n)$

Mamy więc:

$$T(n) = \frac{1}{n} \sum_{i=1}^{n-1} [T(i) + T(n-i)] + \Theta(n)$$

jako że każde $T(j)$ w $\sum$ występuje 2 razy:

$$T(n) = \frac{2}{n} \sum_{i=1}^{n-1} T(i) + \Theta(n)$$

Przez indukcję chcemy pokazać:

$$T(n) \leqslant an \log n + b + \Theta(n) \qquad a, b > 0$$

Dla $n = 1$ — prosto znaleźć odpowiednie a, b!

Dla $n > 1$

$$T(n) \leqslant \frac{2}{n} \sum_{k=1}^{n-1} (ak \log k + b) + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k \log k + 2b \frac{n-1}{n} + \Theta(n)$$

Skorzystamy z:

$$\sum_{k=1}^{n-1} k \log k \leqslant \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Mamy więc:

$$T(n) \leqslant \frac{2a}{n} \left[\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right] + 2b \frac{n-1}{n} + \Theta(n)$$

$$\leqslant an \log n - \frac{a}{4} n + 2b + \Theta(n)$$

$$= an \log n + b + \left[-\frac{an}{4} + b + \Theta(n) \right]$$

$$\leqslant an \log n + b \qquad \text{dla dużego } a$$

Wróćmy do:

$$\sum_{k=1}^{n-1} k \log k \leqslant \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

Wiemy, że:

$$\log \lceil n/2 \rceil \sum_{k=1}^{\lceil n/2 \rceil - 1} k \geqslant \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k$$

$$\log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \geqslant \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

$$\sum_{k=1}^{n-1} k \log k \leqslant (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$= \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\leqslant \frac{1}{2} n(n-1) \log n - \frac{1}{2} \frac{n}{2} \left(\frac{n}{2} - 1 \right)$$

$$\leqslant \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Algorytmy sortowania $O(n)$

- Sortowanie przez zliczanie
- Sortowanie kubełkowe

Sortowanie przez zliczanie

Założenia:

$$A_i \in \mathcal{N}$$

$$1 \leqslant A_i \leqslant k$$

IDEA: Jeśli wiemy ile jest elementów mniejszych od pewnego A_i, wiemy gdzie (na której pozycji) powinien być umieszczony element A_i w tablicy posortowanej:

$$|\{k : A_k < A_i,\ A_k \in \mathcal{A}\}| = ???$$

```
1                                   // A-wejście, B-wyjście
2  void SortZliczanieProste(int A[], int B[], int n, int k) {
3      int i, j=0, *C=malloc(k*sizeof(int));
4      for(i=0; i<k; i++) C[i]=0;
5      for(i=0; i<n; i++)
6          C[A[i]]++;                          // zliczanie
7      for(i=0; i<k; i++)
8          while(C[i]--)
9              B[j++]=i;
10     free(C);
11 }
```

```
1                                   // A-wejście, B-wyjście
2  void SortZliczanie (int A[], int B[], int n, int k) {
3      int i, *C=malloc(k*sizeof(int));
4      for(i=0; i<k; i++) C[i]=0;
5      for(i=0; i<n; i++)
6          C[A[i]]++;                  // zliczanie
7      for(i=1; i<k; i++)
8          C[i]+=C[i-1];               // ile <= od i
9      for(i=n-1; i>=0; i--)
10         B[--C[A[i]]]=A[i];              // w odpowiednie B[.] A[i]
11     free(C);
12 }
```

A:

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

C:

0	1	2	3	4	5
2	0	2	3	0	1

`C[A[i]]++`

C:

0	1	2	3	4	5
2	2	4	7	7	8

`C[i]+=C[i-1]`

B:

0	1	2	3	4	5	6	7
						3	

B:

0	1	2	3	4	5	6	7
	0					3	

B:

0	1	2	3	4	5	6	7
0	0	2	2	3	3	3	5

`B[C[A[i]]]:=A[i]`

Złożoność sortowania przez zliczanie

Wiersze 3–4: $O(k)$

Wiersze 5–6: $O(n)$

Wiersze 7–8: $O(k)$

Wiersze 9–10: $O(n)$

Razem $O(k) + O(n) = O(n + k)$

$= O(n)$, gdy $k = O(n)$

Stabilny = liczby o tej samej wartości w tej samej kolejności w A i w B!

Algorytm sortowania kubełkowego

Średni czas: $O(n)$

Założenia:

- $A_i \in [0, 1) \quad i = 1, \ldots, n$
- A_i mają rozkład jednostajny(!)

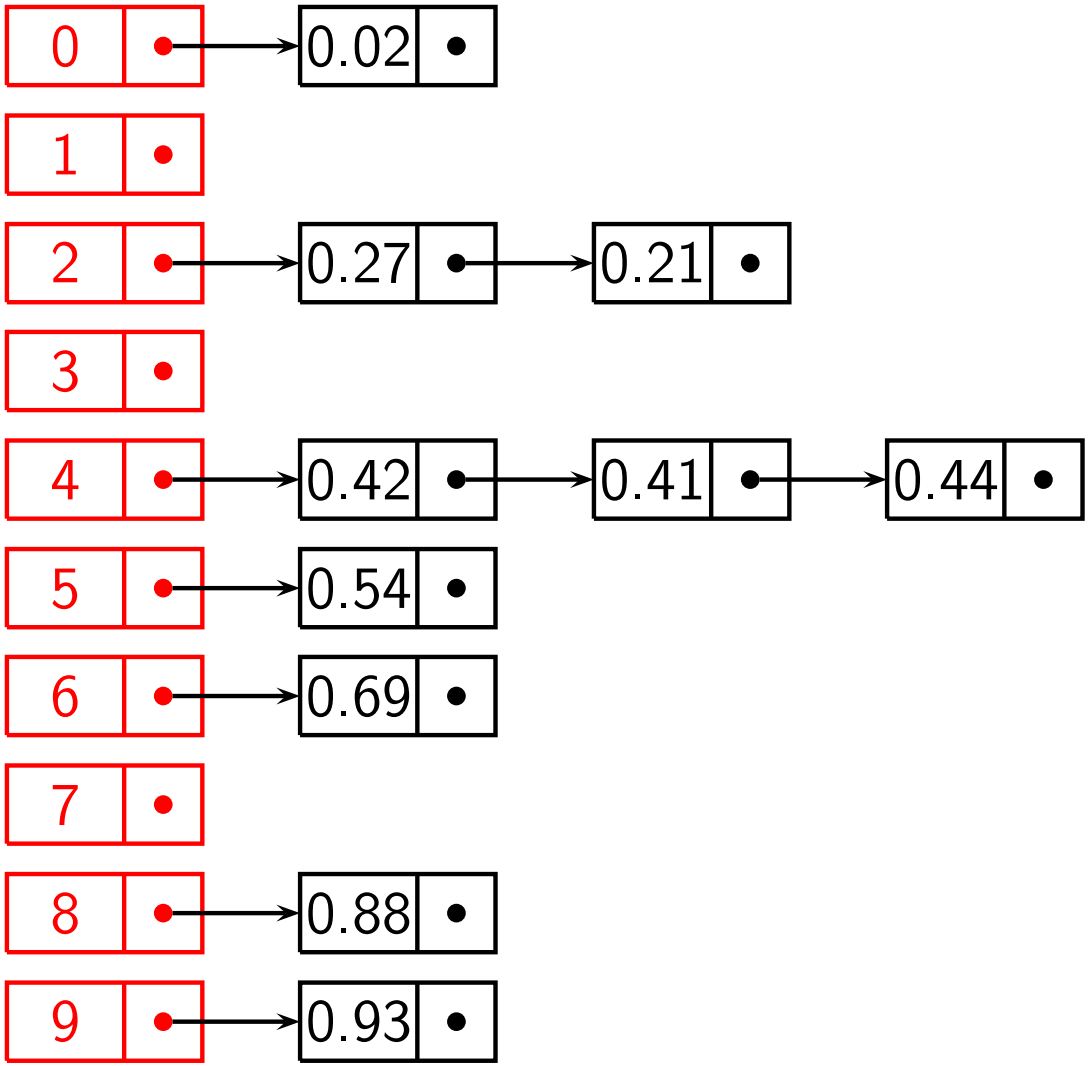
IDEA:

Kubełki = dzielimy $[0, 1)$ na n podprzedziałów.

A

1	0.21
2	0.93
3	0.02
4	0.27
5	0.44
6	0.41
7	0.88
8	0.54
9	0.69
10	0.42

B



```
1 void SortKubelkowe(int A[], int n)
2 {
3     int i;
4     for(i=0; i<n; i++)
5         wstaw A[i] do listy B[⌊nA[i]⌋] z porządkiem;
6     połącz listy B[0], ..., B[n−1];
7     wynik umieść w A;
8 }
```

Oczekiwana złożoność sortowania kubełkowego

w. 3–5 $O(n)$

w. 6–7 ???

n_i — zmienna losowa oznaczająca liczbę elementów w $B[i]$.

Sortowanie z w. 5 i-tego kubełka działa w $O(n_i^2)$.

Czas oczekiwany posortowania $B[i]$ wynosi:

$$E[O(n_i^2)] = O(E[n_i^2])$$

Wtedy oczekiwany czas wykonania w. 6–7:

$$\sum_{i=0}^{n-1} O(E[n_i^2]) = O\left(\sum_{i=0}^{n-1} E[n_i^2]\right)$$

Prawdopodobieństwo p, że element trafi do $B[i]$ wynosi $1/n$ ($1/n$ rozmiar przedziału $B[i]$).

Zmienna n_i ma rozkład dwumianowy.

$P(n_i = k) = b(k; n, p)$

$E[n_i] = np = 1$

$Var[n_i] = np(1-p) = 1 - 1/n$

Wiemy, że $Var[n_i] = E[n_i^2] - E^2[n_i]$

$E[n_i^2] = Var[n_i] + E^2[n_i] = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n} = \Theta(1)$

$$O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) = n\Theta(1) = \Theta(n)$$

Zadania

- Zaimplementuj 3 różne algorytmy szybkiego sortowania różniące się sposobem wyboru elementu podziału
  - pierwszy element ($A[p]$),
  - losowy element,
  - mediana z 3 losowo wybranych elementów.

  Porównaj zachowania się algorytmów dla losowych ciągów, jak i uporządkowanych (częściowo też).
- Jak usprawnić algorytm sortowania kubełkowego, aby jego pesymistyczny czas wynosił $O(n \log n)$?

Statystyka pozycyjna
Wybór k-tego najmniejszego elementu

Mediana

$k = \frac{n}{2}$ — mediana

Rozwiązanie

- Sortowanie — drogie... — $O(n \log n)$
- *Dziel i zwyciężaj* — $O(n)$

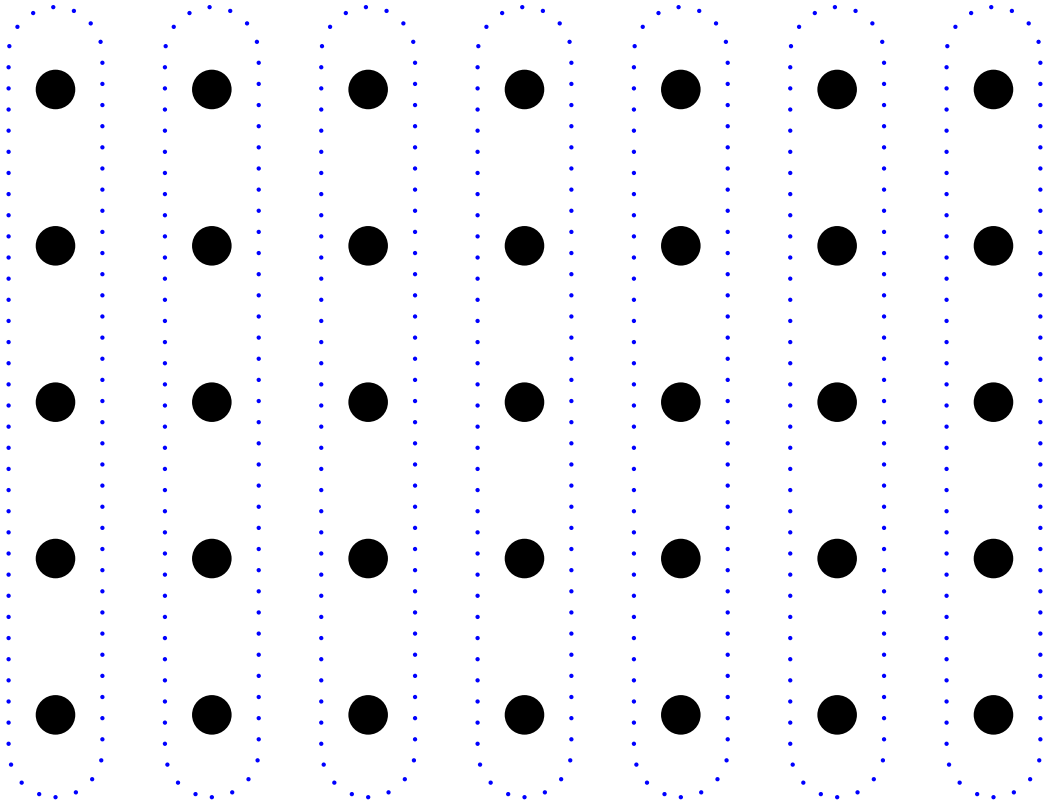
Algorytm Hoare'a

Idea:

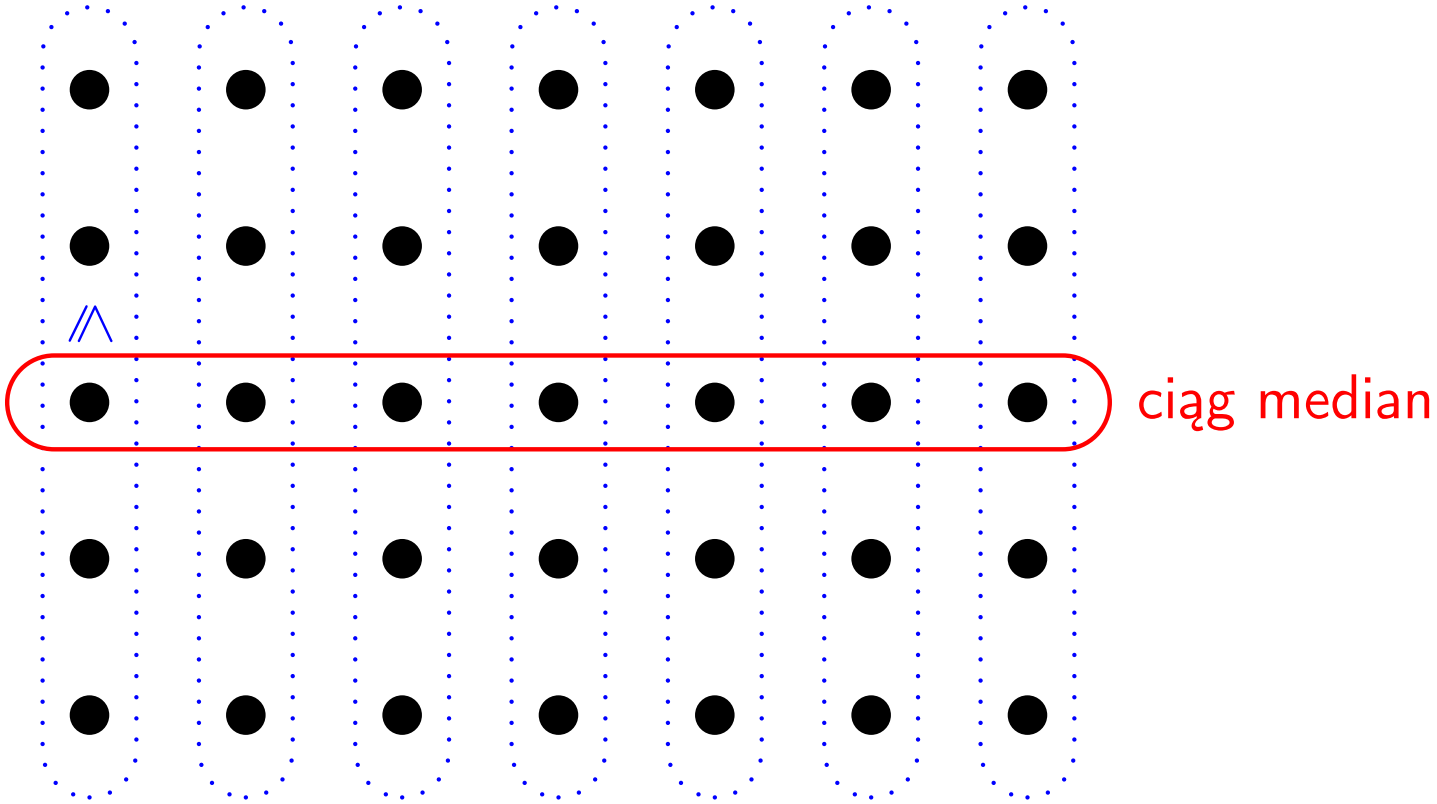
- Podziel $\mathcal{S}$ na trzy zbiory $\mathcal{S}_1$ ($< m$), $\mathcal{S}_2$ ($= m$), $\mathcal{S}_3$ ($> m$).
- Aby czas był liniowy długości $\mathcal{S}_1$ i $\mathcal{S}_3$ muszą być ułamkami długości $\mathcal{S}$.
- Powtarzamy rekurencyjnie...

Jak znaleźć m, dla którego liczba wartości mniejszych i większych będzie ułamkiem $\mathcal{S}$?

- Podzielić $\mathcal{S}$ na $\lfloor \frac{n}{5} \rfloor$ zbiorów.
- Posortować podzbiory (ile to kosztuje?).



- Niech M będzie ciągiem median posortowanych ciągów (M ma $\lfloor n/5 \rfloor$ elementów).
- Znajdź medianę m ciągu M ($T(n/5)$ szybciej niż $T(n)\ldots$)
- Podziel $\mathcal{S}$ na trzy zbiory $\mathcal{S}_1$ ($< m$), $\mathcal{S}_2$ ($= m$), $\mathcal{S}_3$ ($> m$).



Gdzie jest k-ty element?
W $\mathcal{S}_1$, $\mathcal{S}_2$ czy $\mathcal{S}_3$?

- Jeśli $|\mathcal{S}_1| \geqslant k$ — $\mathcal{S}_1$ ma k-ty element, w p.p.
- Jeśli $|\mathcal{S}_1| + |\mathcal{S}_2| \geqslant k$ — $\mathcal{S}_2$ ma k-ty element $= m$!
- w p.p. k-ty element jest w $\mathcal{S}_3$, jako $k - |\mathcal{S}_1| - |\mathcal{S}_2|$ element w $\mathcal{S}_3$.

```
Select(k,S) {
  if (|S| < 50) {
      sortuj S;
      return k-ty najmniejszy;
  }
  else {
    podziel S na ⌊|S|/5⌋ możliwie równych ciągów;
    sortuj każdy taki ciąg;
    niech M będzie ciągiem median powyższych podciągów;
    m=Select(⌈|M|/2⌉, M);
    Podziel 𝒮 na 𝒮_1(< m), 𝒮_2(= m), 𝒮_3(> m);
    if (|𝒮_1| >= k) return Select(k,𝒮_1);
    else
    if (|𝒮_1| + |𝒮_2| >=k) return m;
    else return Select(k−|𝒮_1| − |𝒮_2|, 𝒮_3);
  }
}
```

Analiza algorytmu

Załóżmy, że dla ciągu długości n algorytm wykonuje $T(n)$ operacji.

w. 3 — O(1)

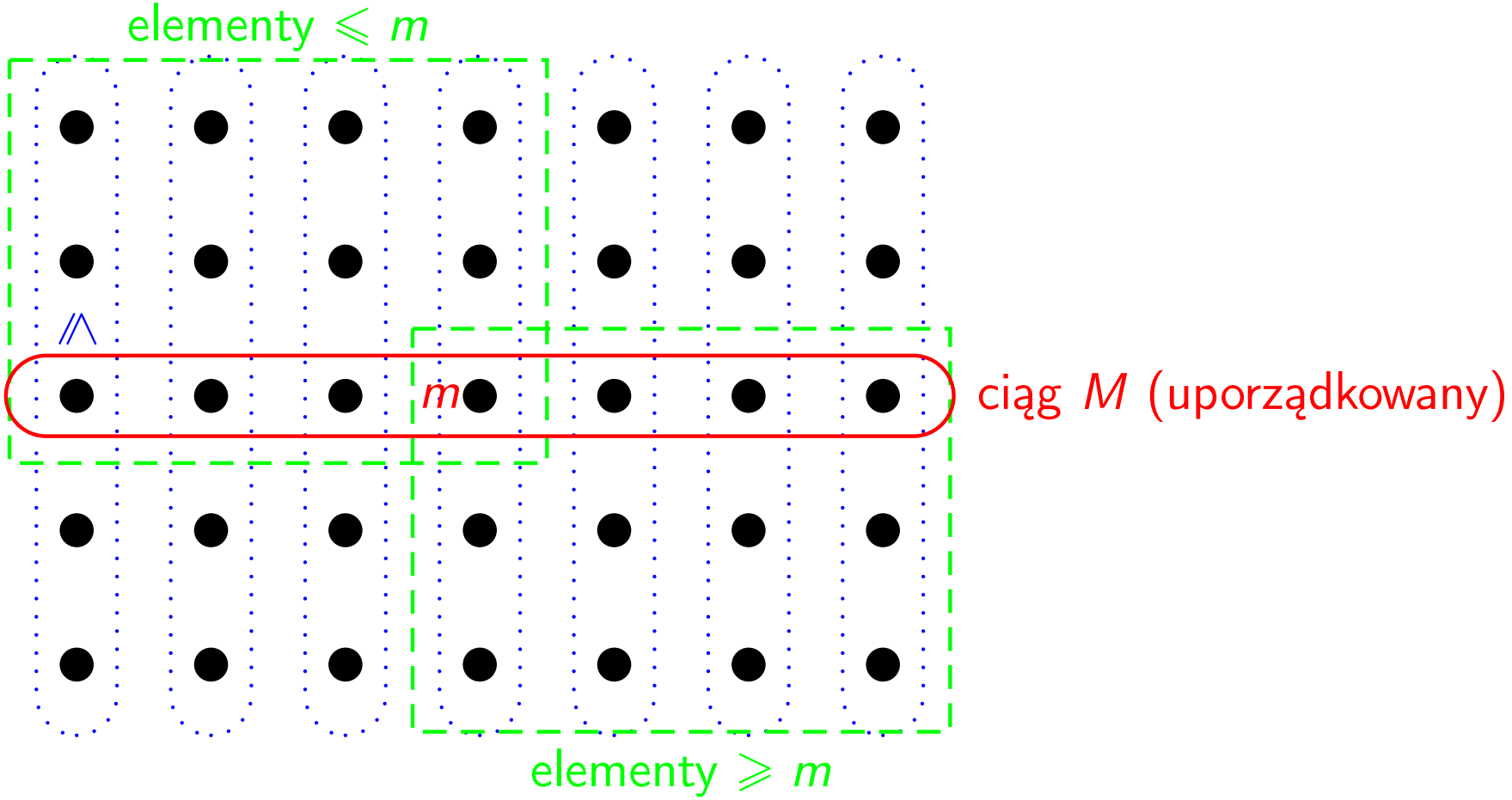
w. 8 — O(n)

w. 9 — n/5

w. 10 — $T(n/5)$

w. 11 — O(n)

w. 12 i 15 — $T(|\mathcal{S}_1|)$ i $T(|\mathcal{S}_3|)$ odpowiednio (wykonywane jest tylko jedno z nich!)


elementy ⩽ m
m
ciąg M (uporządkowany)
elementy ⩾ m

- M ma co najwyżej $\lfloor n/10 \rfloor$ elementów $\geqslant m$
- Wtedy $\mathcal{S}_1$ ma co najwyżej $n - 3\lfloor n/10 \rfloor$ elementów.
  Dla $n \geqslant 50$ $\quad |\mathcal{S}_1| < 3n/4$.
- Podobnie dla $\mathcal{S}_3$.
- Czyli w. 12 i 15 wymagają nie więcej niż $T(3n/4)$

Mamy więc:

$$\begin{array}{ll} T(n) \leqslant cn & \text{dla } n < 50 \\ T(n) \leqslant T(n/5) + T(3n/4) + cn & \text{dla } n \geqslant 50 \end{array}$$

Można (przez indukcję) pokazać, że $T(n) \leqslant 20cn$.

Algorytm wyboru II

```
Select2(k,S) {
  if (|S|=1) return S[1];
  else
  {
    m = S[Random(1,n)];
    Podziel 𝒮 na 𝒮₁(< m), 𝒮₂(= m), 𝒮₃(> m);
    if (|𝒮₁| >= k) return Select2(k,𝒮₁);
    else
    if (|𝒮₁| + |𝒮₂| >= k) return m;
    else return Select2(k−|𝒮₁| − |𝒮₂|,𝒮₃);
  }
}
```

Złożoność średniego przypadku algorytmu Select2

- Każdy indeks i wybranego elementu m w w.5 (po wykonaniu partition w.6) jest tak samo prawdopodobny.
- Jeśli $i > k$ to wykonywana jest funkcja *Select* dla ciągu $i - 1$ elementowego.
- Jeśli $i < k$ to dla $n - i$.

Stąd oczekiwany koszt w.7 lub w.10 — uśredniamy po różnych możliwych wartościach i dla zadanego k:

$$\frac{1}{n}\left[\sum_{i=1}^{k-1} T(n-i) + \sum_{i=k+1}^{n} T(i-1)\right] = \frac{1}{n}\left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i)\right]$$

Wtedy dla $T(n)$ mamy:

$$T(n) \leqslant cn + \max_k \left\{ \frac{1}{n} \left[\sum_{i=n-k+1}^{n-1} T(i) + \sum_{i=k}^{n-1} T(i) \right] \right\}$$

Przez indukcję otrzymujemy:

$$T(n) \leqslant 4cn.$$

Zadania

- Zapisz ostateczną postać wiersza 11 funkcji *Select*.
- Przeanalizuj działanie funkcji *Select* dla wybranego ciągu długości około 8 elementów.
- W wierszu 7 na stronie 127 dzielimy S na podciągi długości 5. Jak algorytm zachowa się dla innych liczb (3,4,7,11,etc.)?
- Co możesz powiedzieć o algorytmie szybkiego sortowania w którym procedura *Partition* dzieliła by zbiór względem mediany? Co dzieje się ze złożonością średniego przypadku, a co ze złożonością najgorszego przypadku?
- Udowodnić przez indukcję, że dla algorytmu *Select* mamy: $T(n) \leqslant 20cn$.
- Udowodnić przez indukcję, że dla algorytmu *Select2* mamy: $T(n) \leqslant 4cn$.

Mnożenie macierzy metodą Strassena

Zwyczajne mnożenie — $O(n^3)$ — $C_{ij} = \sum_k A_{ik} B_{kj}$

Można tak:

$$\begin{bmatrix} A_{11} A_{12} \\ A_{21} A_{22} \end{bmatrix} \begin{bmatrix} B_{11} B_{12} \\ B_{21} B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} C_{12} \\ C_{21} C_{22} \end{bmatrix}$$

gdzie

$$\begin{aligned} C_{11} &= A_{11} B_{11} + A_{12} B_{21} \\ C_{12} &= A_{11} B_{12} + A_{12} B_{22} \\ C_{21} &= A_{21} B_{11} + A_{22} B_{21} \\ C_{22} &= A_{21} B_{12} + A_{22} B_{22} \end{aligned}$$

Ogólny koszt:

$$T(n) = mT\left(\frac{n}{2}\right) + \frac{an^2}{4}$$

Wtedy

$$T(n) \leqslant kn^{\log m}$$

k – stała

Strassen — $m = 7$**!**

7 mnożeń i 18 dodawań

Mamy wtedy algorytm mnożenia o złożoności $O(n^{\log 7}) = O(n^{2.81})$.

$$
\begin{aligned}
m_1 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
m_2 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
m_3 &= (A_{11} - A_{21})(B_{11} + B_{12}) \\
m_4 &= (A_{11} + A_{12})B_{22} \\
m_5 &= A_{11}(B_{12} - B_{22}) \\
m_6 &= A_{22}(B_{21} - B_{11}) \\
m_7 &= (A_{21} + A_{22})B_{11}
\end{aligned}
$$

wtedy

$$
\begin{aligned}
C_{11} &= m_1 + m_2 - m_4 + m_6 \\
C_{12} &= m_4 + m_5 \\
C_{21} &= m_6 + m_7 \\
C_{22} &= m_2 + m_3 + m_5 - m_7
\end{aligned}
$$

$$n = 10^3 \qquad n^3 = 10^9 \qquad n^{2.81} = 2.7 \cdot 10^8$$

$$n = 10^6 \qquad n^3 = 10^{18} \qquad n^{2.81} = 7.2 \cdot 10^{16}$$

Programowanie dynamiczne

- Oblicza rozwiązania wszystkich podzadań.
- Kolejność rozwiązywania zadań od małych do dużych.
- Rozwiązania małych zadań służą pomocą w rozwiązywaniu większych zadań.
- Rozwiązuje w czasie wielomianowym, to co wydaje się *bardziej* trudne.

Mnożenie ciągu macierzy

$$M = M_1 \times M_2 \times \ldots M_n$$

Niech M_i maja rozmiar $r_{i-1} \times r_i$.

Wtedy koszt mnożenia $M_i \times M_{i+1}$ wynosi $r_{i-1} \cdot r_i \cdot r_{i+1}$.

$$\begin{array}{ccccccc} M = & M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ & [10 \times 20] & & [20 \times 50] & & [50 \times 1] & & [1 \times 100] \end{array}$$

Koszty?

$M_1 \times (M_2 \times (M_3 \times M_4))$ — 125.000

$(M_1 \times (M_2 \times M_3)) \times M_4$ — 2.200

Liczba wszystkich możliwych nawiasowań

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

$P(n) = C(n-1)$

$C(n)$ — tworzy ciąg liczb Catalana:

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})$$

Optymalne nawiasowanie

Załóżmy, że ciąg $M = M_1, M_2, \ldots, M_n$

dzielony jest na

$L = M_1, M_2, \ldots, M_k$ i $P = M_{k+1}, M_{k+2}, \ldots, M_n$ $(1 \leqslant k < n)$

$$(L) \times (P)$$

w wyniku pewnego nawiasowania.

Najpierw następuje mnożenie ciągu L i P, a następnie mnożenie $M_{1\ldots k} \times M_{k+1\ldots n}$ (tj. produktów L i P).

Jeśli nawiasowanie ciągu M jest optymalne, to również nawiasowanie L i P jest optymalne.

Niech $m[i,j]$ będzie kosztem wymnożenia ciągu macierzy $M_i \times \ldots \times M_j$ (dla optymalnego nawiasowania).

$m[i,i] = 0 \quad i = 1, \ldots, n$

Dla pewnego k ($i \leqslant k < j$) koszt:

$m[i,j] = m[i,k] + m[k+1,j] + r_{i-1} r_k r_j \quad i < j$

$m[i,j] = \text{koszt}(L = M_i \times \ldots \times M_k) + \text{koszt}(P = M_{k+1} \times \ldots \times M_j) + \text{koszt}(L \times P)$

Można wybrać k tak, aby koszt był jak najmniejszy:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leqslant k < j} \left[m[i,k] + m[k+1,j] + r_{i-1} r_k r_j \right] & i < j \end{cases}$$

Zobaczmy, że $m[2,6]$ wymaga $m[2,2], m[2,3], m[3,4], m[2,4], m[3,6]$, etc.

```
MatrixChainOrder(n,r,m,s)
{
  for(i=1; i<=n; i++)
    m[i,i]=0;
  for(l=1; l<=n-1; l++)
    for  i=1; i<=n-l; i++)
    {
      j=i+l;
      m[i,j]= min_{i≤k<j} (m[i,k] + m[k+1,j] + r[i-1] * r[k] * r[j])
      s[i,j]= arg min_{i≤k<j} (m[i,k] + m[k+1,j] + r[i-1] * r[k] * r[j])
      }
}
```

$m[1,2], m[2,3], m[3,4] \ldots, m[1,3], m[2,4], m[3,5], \ldots, m[1,4], m[2,5], \ldots$

$m[1,n]$ — koszt całkowity $\quad$ $s[1,n]$ — główne nawiasowanie

$s[1, s[1,n]]$ i $s[s[1,n]+1, n]$ — kolejne nawiasy od zewnątrz

$$((M_1 \times \ldots \times M_{s[1,s[1,n]]}) \times (M_{s[1,s[1,n]]+1} \times \ldots \times M_{s[1,n]}))$$

$$\times$$

$$((M_{s[1,n]+1} \times \ldots \times M_{s[s[1,n]+1,n]}) \times (M_{s[s[1,n]+1,n]+1} \times \ldots \times M_n))$$

Złożoność: $O(n^3)$

w. 3–4 — $O(n)$

w. 9–10 — $O(n)$ (i w jednej pętli!)

w. 5–11 — $O(n^3)$

```
MatrixChainMultiply(M,s,i,j)
{
    if (j>i) {
      X=MatrixChainMultiply(M,s,i,s[i,j]);
      Y=MatrixChainMultiply(M,s,s[i,j]+1,j);
      return MatrixMultiply(X,Y);
    }
    else
        return M_i;
}
```

M — ciąg macierzy M_i

Wywołanie: MatrixChainMultiply(M,s,1,n)

Zadania

- Wyznaczyć macierze m i s (kosztów i podziałów) dla ciągu rozmiarów macierzy: $6, 12, 3, 5, 20, 4, 4, 12$.
- Domknięcie grafu $G = (V, E)$. Domknięciem grafu G jest taki graf $G' = (V, E')$, w którym wierzchołki i i j są połączone krawędzią, jeśli tylko istnieje ścieżka z i do j w grafie G. Czyli $E' = \{(i, j) : \text{istanieje ścieżka z } i \text{ do } j \text{ w } G\}$.
  Zaproponuj algorytm programowania dynamicznego dla tak zdefiniowanego problemu.
- Wykaż, że pełne nawiasowanie n-elementowego wyrażenia zawiera dokładnie $n - 1$ par nawiasów.

Programowanie dynamiczne II

Największy wspólny podciąg (NWP)

Podciąg ciągu X: Ciąg X z usuniętymi(!) niektórymi elementami.
$X = \langle x_1, x_2, \ldots, x_m \rangle$ i $Y = \langle y_1, y_2, \ldots, y_n \rangle$

np.: $X = \langle A, B, C, B, D, A \rangle$ i $Y = \langle B, D, C, A, B, A \rangle$

Cel: Znaleźć najdłuższy ciąg Z, który jest podciągiem X i Y.

$Z = \langle z_1, z_2, \ldots, z_k \rangle$

odp.: $Z = \langle B, C, B, A \rangle$

Naiwnie: wygenerować wszystkie podciągi X i sprawdzić (dla każdego) czy jest podciągiem Y. Zapamiętać najdłuższy.

Koszt: 2^m — okropnie!

$[X_i = \langle x_1, x_2, \ldots, x_i \rangle]$ — prefiks

Jeśli Z jest NWP, to:

- Jeśli $x_m = y_n$, to $z_k = x_m = y_n$, a Z_{k-1} jest NWP X_{m-1} i Y_{n-1}.
- Jeśli $x_m \neq y_n$ i $z_k \neq x_m$, to Z jest NWP X_{m-1} i Y.
- Jeśli $x_m \neq y_n$ i $z_k \neq y_n$, to Z jest NWP X i Y_{n-1}.

Czyli

- $x_m = y_n$ to $Z = \mathrm{NWP}(X_{m-1},\ Y_{n-1}) + x_m$
- $x_m \neq y_n$ to Z dłuższy z
  - $\mathrm{NWP}(X_{m-1},\ Y)$
  - $\mathrm{NWP}(X,\ Y_{n-1})$

$\mathrm{NWP}(X_{m-1},\ Y)$ i $\mathrm{NWP}(X,\ Y_{n-1})$ mają wspólny $\mathrm{NWP}(X_{m-1},\ Y_{n-1})$.

$c[i,j]$ — długość NWP X_i i Y_j.

Jeśli $i = 0$ lub $j = 0$ to $c[i,j] = 0$.

$$c[i,j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ c[i-1,j-1]+1 & i,j > 0 \wedge x_i = y_j \\ \max[c[i,j-1], c[i-1,j]] & i,j > 0 \wedge x_i \neq y_j \end{cases}$$

- Warto zauważyć, że macierz C ma rozmiary $m \times n$, czyli łączna liczba podproblemów jest stosunkowo niska (nie wykładnicza!!!).
- Wyznaczanie $c[i,j]$ będzie przebiegało po wierszach, od lewej do prawej strony.

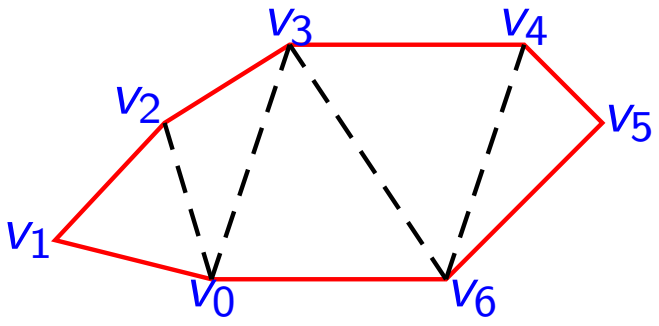
```
LCS_length(X,Y) {
    m=length(X); n=length(Y);
    for(i=1; i<=m; i++) c[i,0]=0;
    for(i=1; i<=n; i++) c[0,i]:=0;

    for(i=1; i<=m; i++)
        for(j=1; j<=n; j ++)
            if(x[i] == y[j]) {
                c[i,j]=c[i-1,j-1]+1;   b[i,j]=↖;
            }
            else if(c[i-1,j] >= c[i,j-1]) {
                c[i,j]=c[i-1,j];    b[i,j]=↑;
            }
            else {
                c[i,j]=c[i,j-1];    b[i,j]=←;
            }
}
```

i	j	0	1	2	3	4	5	6
		y_j	B	d	C	a	B	A
0	x_i	0	0	0	0	0	0	0
1	a	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
2	B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
3	C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
4	B	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
5	d	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
6	A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
7	b	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

```
Print_LCS(b,X,Y,i,j)
{
    if (i==0 || j==0) return;
    else
    if (b[i,j] == ↖) {
            Print_LCS(b,X,Y,i−1,j−1);
            print x[i];
    } else
    if (b[i,j] == ↑)
        Print_LCS(b,X,Y,i−1,j);
    else
        Print_LCS(b,X,Y,i,j−1);
}
```

Złożoność: $O(n + m)$

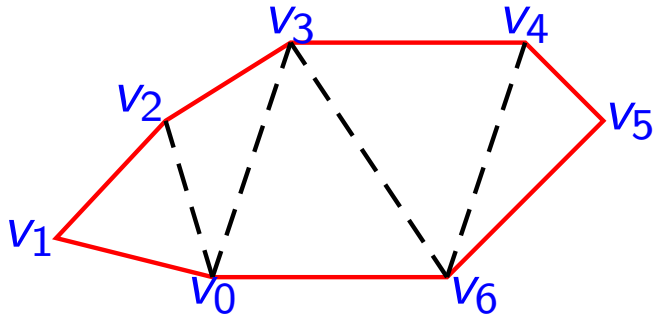
Triangularyzacja wielokąta (wypukłego) — PD III

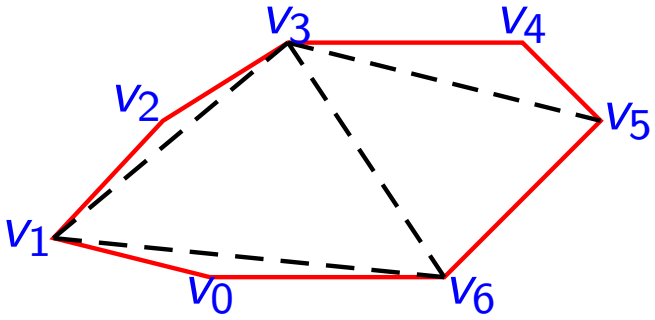


- Podział wielokąta na trójkąty.
- $n-2$ trójkąty i $n-3$ przekątne.
- Podziałów różnych wiele...
- Optymalna triangularyzacja:
  minimalizująca sumę pewnych funkcji wag trójkątów, np.:

$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

Nawiasowanie a triangularyzacja



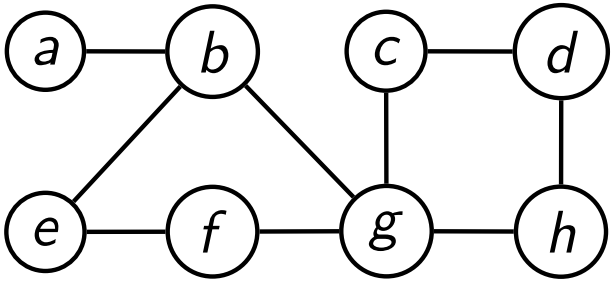


- Takie samo poszukiwanie optymalnego rozwiązania:
  Jeśli cała triangularyzacja optymalna, to dwie pod-triangularyzacje również!
- $t[i,j]$ — koszt triangularyzacji wielokąta $\langle v_{i-1}, v_i, \ldots, v_j \rangle$

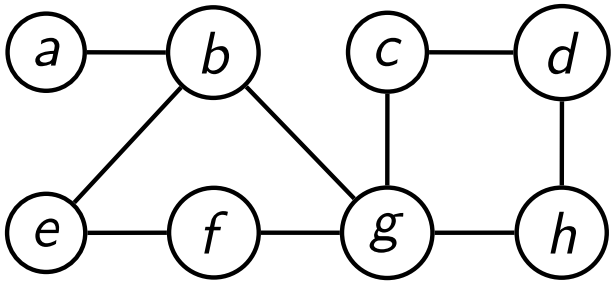
$$t[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leqslant k < j} \left[t[i,k] + t[k+1,j] + w(v_{i-1}, v_k, v_j) \right] & i < j \end{cases}$$

Przeszukiwanie grafu wszerz

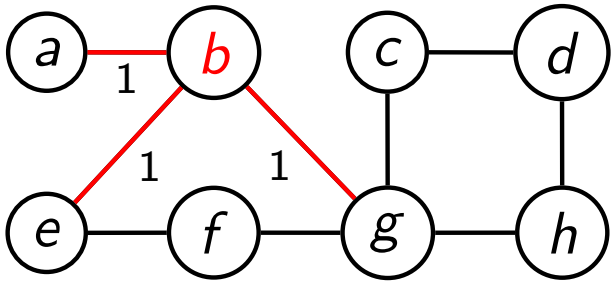
Cel: Odwiedzić wszystkie wierzchołki grafu $G = (V, E)$, do których daje się dojść z wybranego wierzchołka v tak, aby wierzchołki bliższe v były odwiedzane przed odleglejszymi (odległość $d(v, w)$ jest równa liczbie krawędzi minimalnej ścieżki z v do danego wierzchołka w, brak ścieżki oznacza odległość ∞).



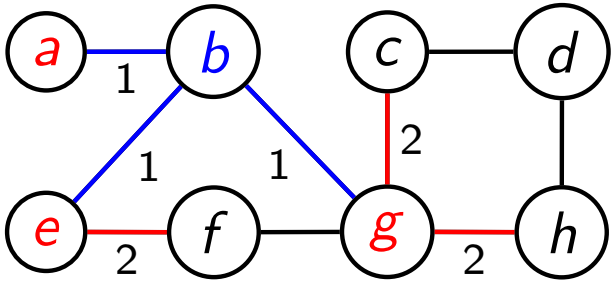
1


a
b
c
d
e
f
g
h

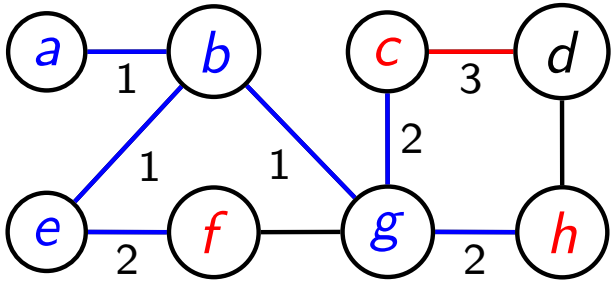
2


a
b
c
d
e
f
g
h
1
1
1

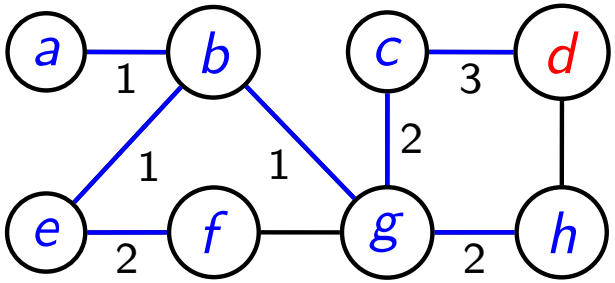
3


a
b
c
d
e
f
g
h
1
1
1
2
2
2

4


a
b
c
d
e
f
g
h
1
1
1
2
3
2
2

5


a
b
c
d
e
f
g
h
1
1
1
2
3
2
2

d[u] — odległość z s do u

p[u] — poprzednik u w drzewie przeszukiwania wszerz

S[u] — lista sąsiadów wierzchołka u

Q — kolejka FIFO

kolor[u] — kolor wierzchołka u. biały = wierzchołek nieodwiedzony, czerwony = odwiedzany, niebieski = odwiedzony

```
BFS(G,s)
{
    foreach(u ∈ V \ {s})
    {
        kolor[u]=biały;
        d[u]=∞;
        p[u]=NULL;
    }
    kolor[s]=czerwony;
```

```
    d[s]=0;
    p[s]=NULL;
    Q={s};
    while(Q != ∅)
    {
        u=HEAD(Q);
        foreach(v ∈ S[u])
            if ( kolor [v]==biały) {
                kolor [v]=czerwony
                d[v]=d[u]+1;
                p[v]=u;
                Q=Q ∪ v;
            }
        Q=Q \ u;
        kolor [u]= niebieski ;
    }
}
```

Złożoność BFS: $O(|V| + |E|)$

w. 3–8 — $O(|V|)$

w. 13–25 — $O(|E|)$

Przeszukiwanie wszerz – drzewo przeszukiwań

Najkrótsze ścieżki

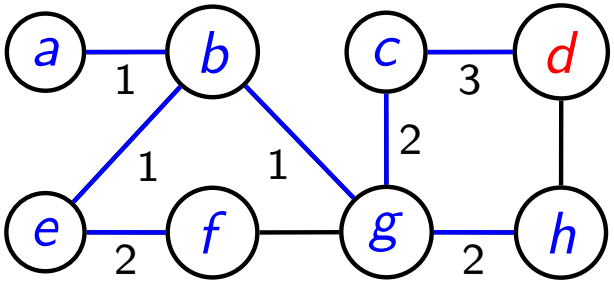
d[v] zawiera informację o najkrótszej ścieżce z s do v.

Podgraf poprzedników $G_p = (V_p, E_p)$

$V_p = \{v \in V : p[v] \neq NULL\} \cup \{s\}$

$E_p = \{(p[v], v) \in E : v \in V_p \setminus \{s\}\}$

Korzystając z $p[.]$ można wypisać całą ścieżkę pomiędzy s, a wybranym wierzchołkiem v.



```
PrintPath(G,s,v) {
    if (v==s)
        write(s);
    else
    if (p[v]==NULL)
        printf ("nie ma ścieżki z s do v");
    else
    {
        PrintPath(G,s,p[v]);
        printf (v);
    }
}
```

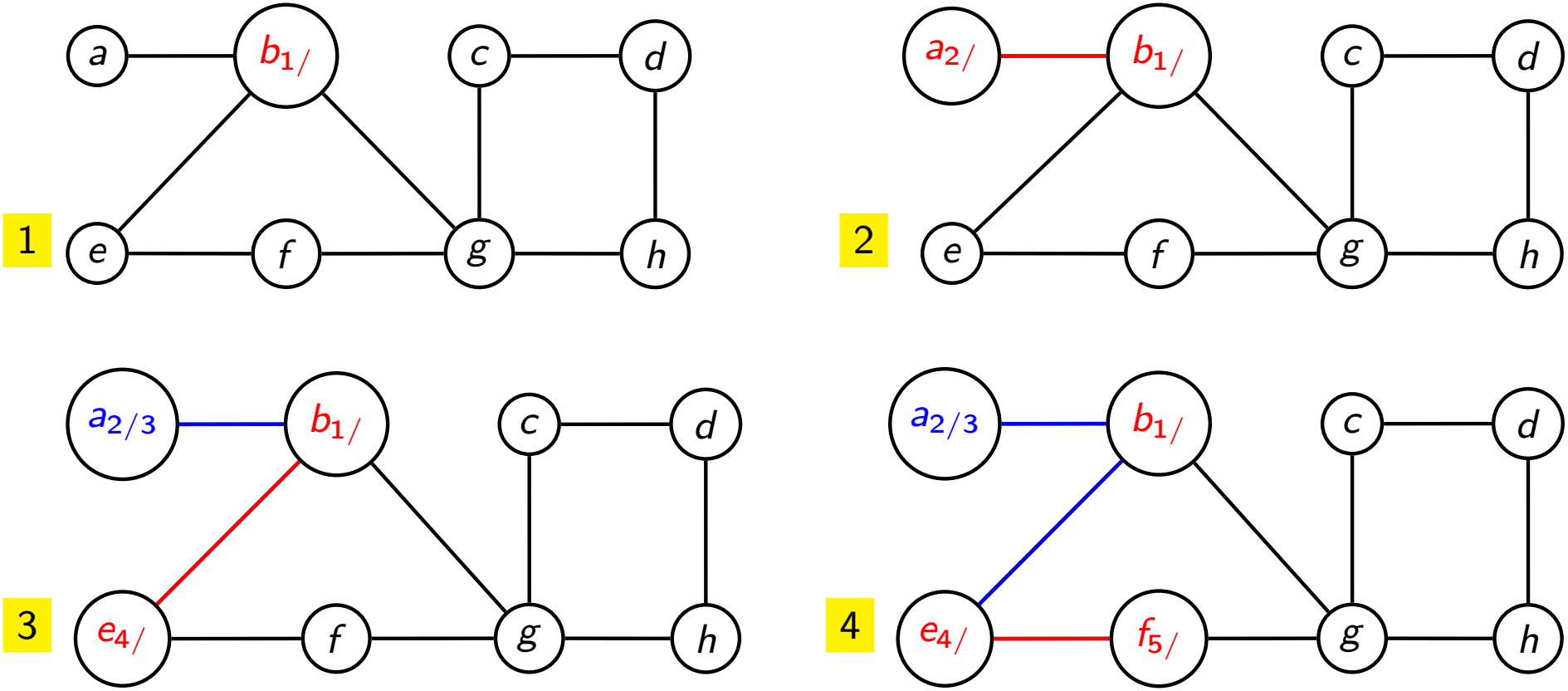
Przeszukiwanie w głąb

Cel: Przejrzenie wierzchołków grafu. W pierwszej kolejności przeszukuje się pierwszą ścieżkę do nieodwiedzonego jeszcze wierzchołka, od którego jeśli tylko również da się przejść jeszcze *głębiej* to jest to wykonywane, itd. Gdy w końcu nie da się kontynuować takiej strategii następuje powrót do wierzchołka, w którym zostały jeszcze ścieżki do nieodwiedzonych wierzchołków, z których algorytm znów będzie próbował przeszukiwać w głąb. Całość jest tak długo powtarzana, aż każdy wierzchołek grafu zostanie przejrzany.

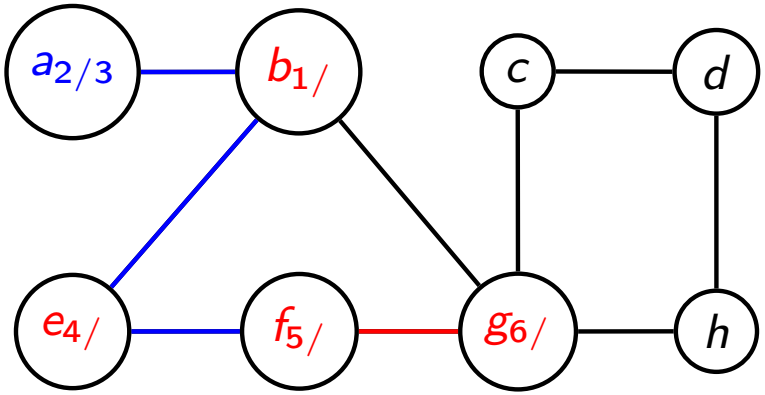
Taka strategia postępowania powoduje powstanie pojedynczego drzewa, bądź lasu drzew.

$E_p = \{(p[v], v) : v \in V \ \wedge \ p[v] \neq NULL\}$

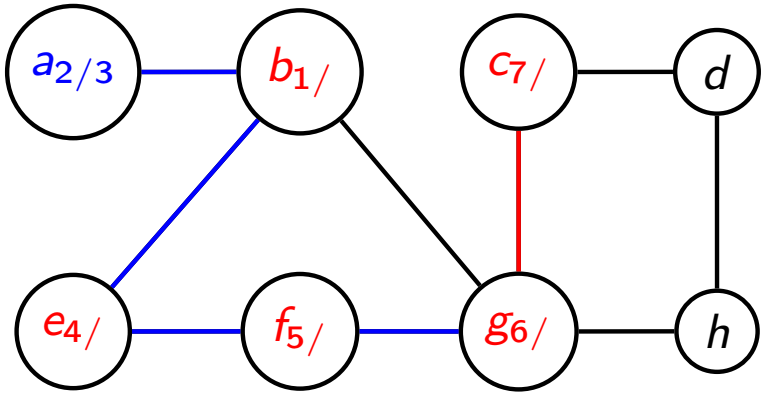
Kolor wierzchołka – jak w algorytmie
Kolor krawędzi — niebieski (przebyta droga), czerwony (kierunek kolejnego wierzchołka)



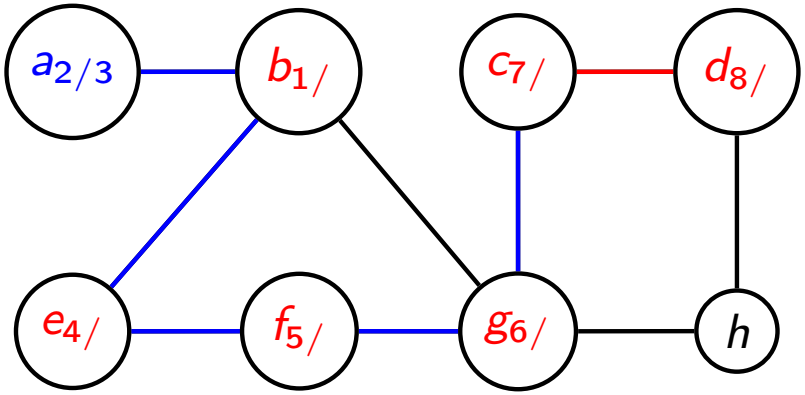
5


$a_{2/3}$
$b_{1/}$
c
d
$e_{4/}$
$f_{5/}$
$g_{6/}$
h

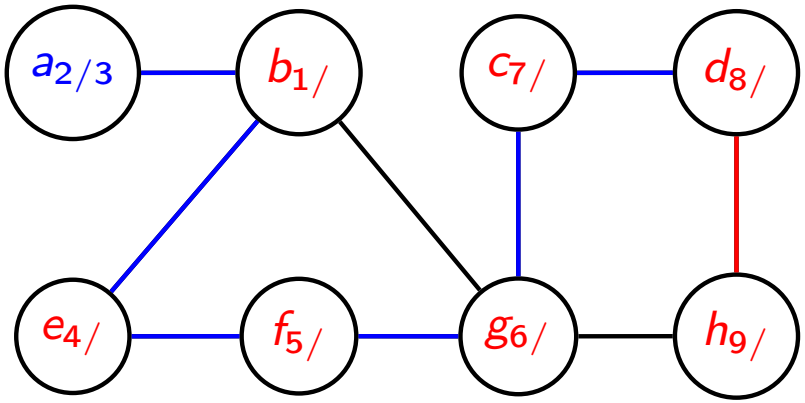
6


$a_{2/3}$
$b_{1/}$
$c_{7/}$
d
$e_{4/}$
$f_{5/}$
$g_{6/}$
h

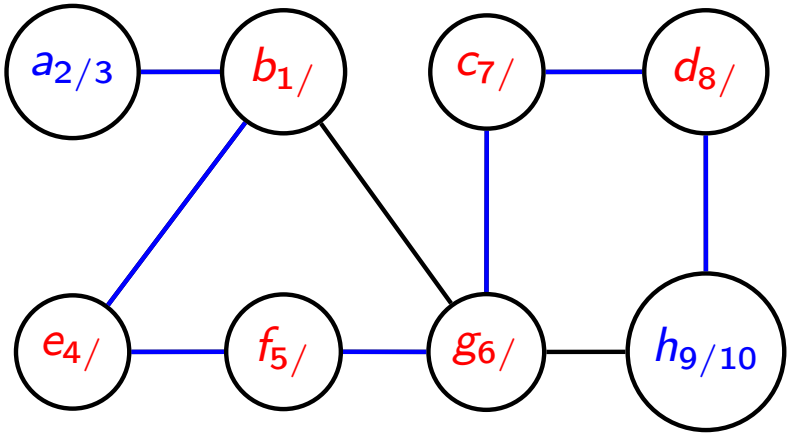
7


$a_{2/3}$
$b_{1/}$
$c_{7/}$
$d_{8/}$
$e_{4/}$
$f_{5/}$
$g_{6/}$
h

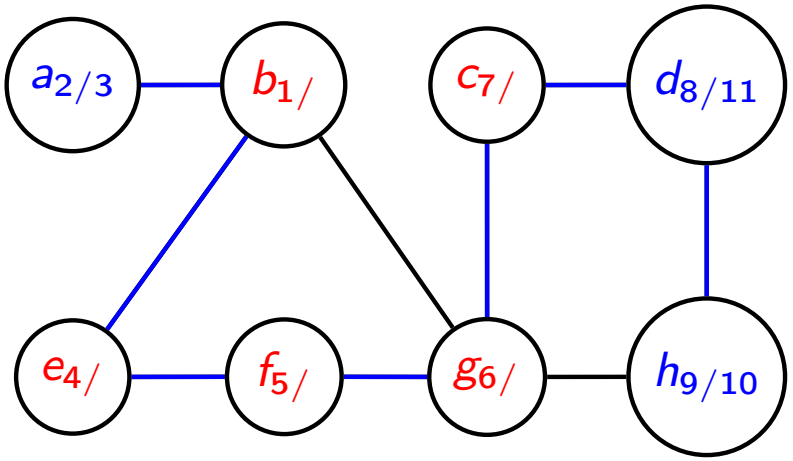
8


$a_{2/3}$
$b_{1/}$
$c_{7/}$
$d_{8/}$
$e_{4/}$
$f_{5/}$
$g_{6/}$
$h_{9/}$

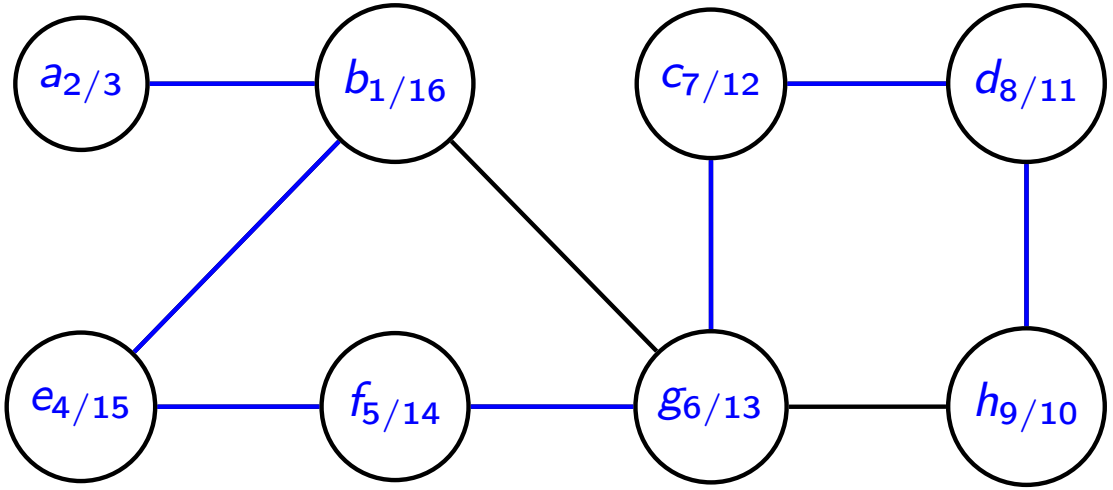
9


$a_{2/3}$
$b_{1/}$
$c_{7/}$
$d_{8/}$
$e_{4/}$
$f_{5/}$
$g_{6/}$
$h_{9/10}$

10

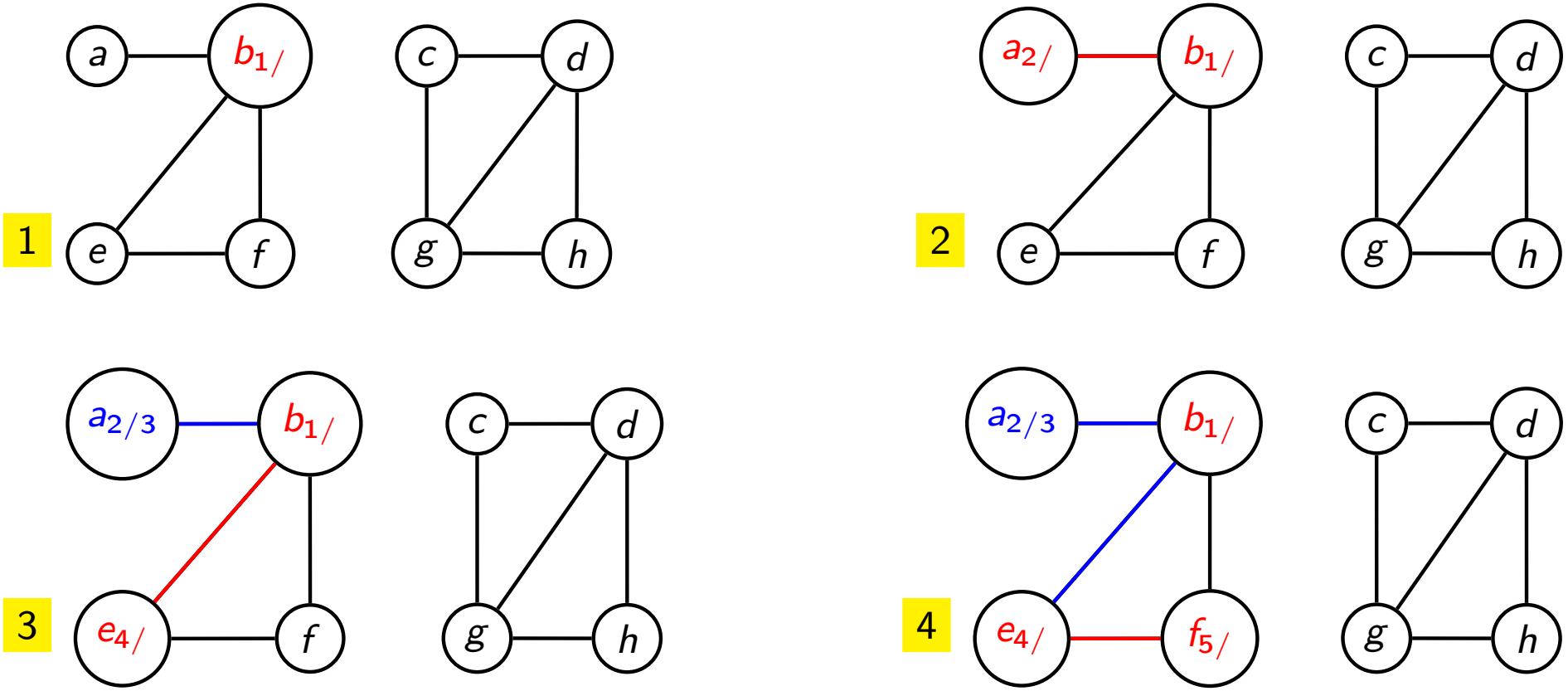

$a_{2/3}$
$b_{1/}$
$c_{7/}$
$d_{8/11}$
$e_{4/}$
$f_{5/}$
$g_{6/}$
$h_{9/10}$

...

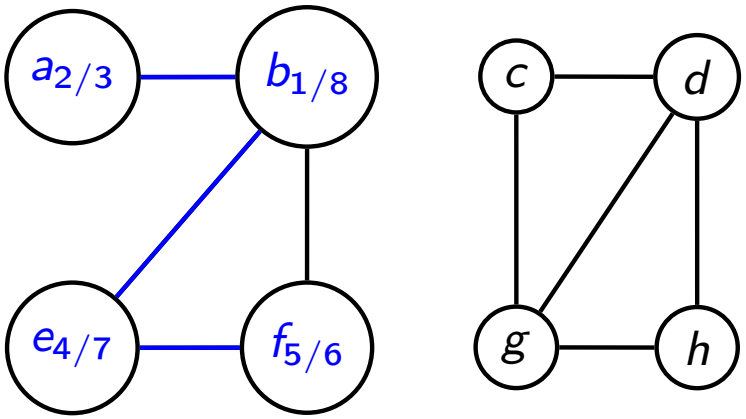

$a_{2/3}$
$b_{1/16}$
$c_{7/12}$
$d_{8/11}$
$e_{4/15}$
$f_{5/14}$
$g_{6/13}$
$h_{9/10}$

```
DFS(G) {
    foreach(u ∈ V)
    {
        kolor[u]=biały;
        p[u]=NULL;
    }
    time=0
    foreach(u ∈ V)
        if ( kolor[u]==biały)
            DFSV(G,u);
}
```

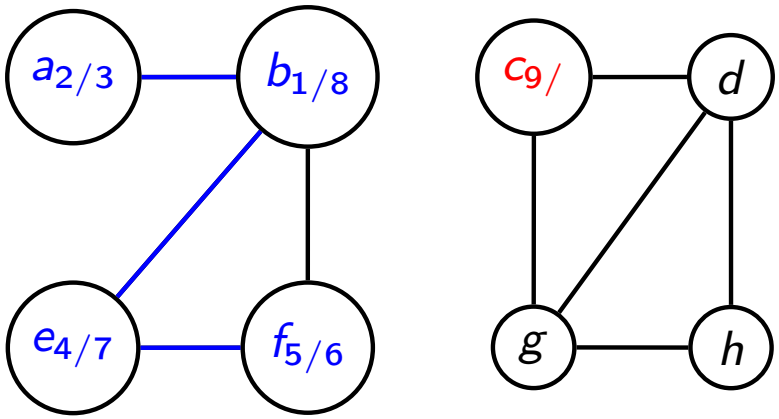
```
DFSV(G,u) {
    kolor[u]=czerwony;
    time=time+1;
    d[u]=time;
    foreach(v ∈ S[u])
        if ( kolor[v]==biały)
        {
            p[v]=u;
            DFSV(G,v);
        }
    kolor[u]=niebieski;
    time=time+1;
    f[u]=time;
}
```


1
a
$b_{1/}$
c
d
e
f
g
h
2
$a_{2/}$
$b_{1/}$
c
d
e
f
g
h
3
$a_{2/3}$
$b_{1/}$
c
d
$e_{4/}$
f
g
h
4
$a_{2/3}$
$b_{1/}$
c
d
$e_{4/}$
$f_{5/}$
g
h

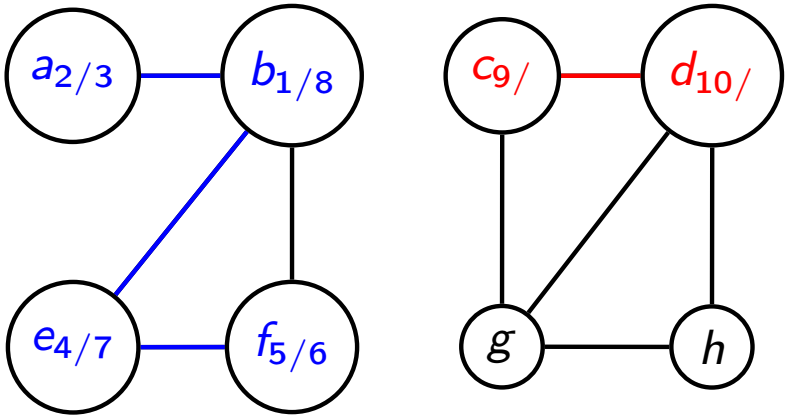
5–8


$a_{2/3}$
$b_{1/8}$
c
d
$e_{4/7}$
$f_{5/6}$
g
h

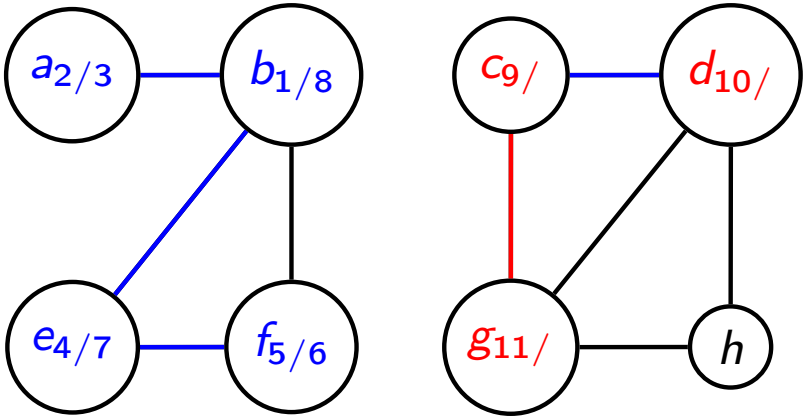
9


$a_{2/3}$
$b_{1/8}$
$c_{9/}$
d
$e_{4/7}$
$f_{5/6}$
g
h

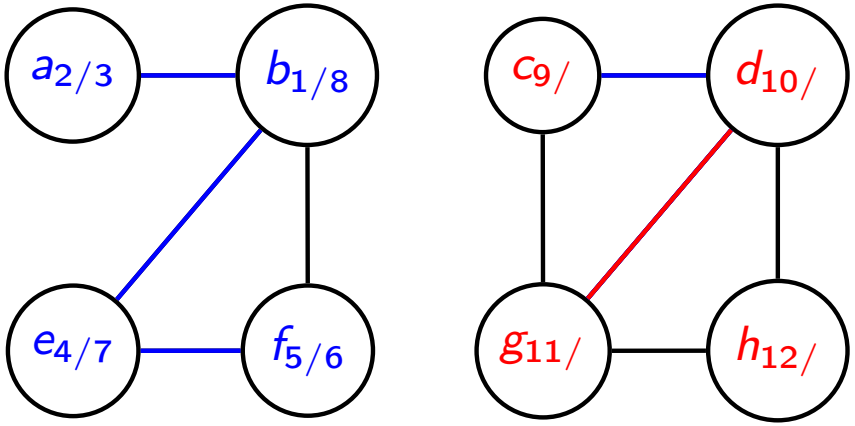
10


$a_{2/3}$
$b_{1/8}$
$c_{9/}$
$d_{10/}$
$e_{4/7}$
$f_{5/6}$
g
h

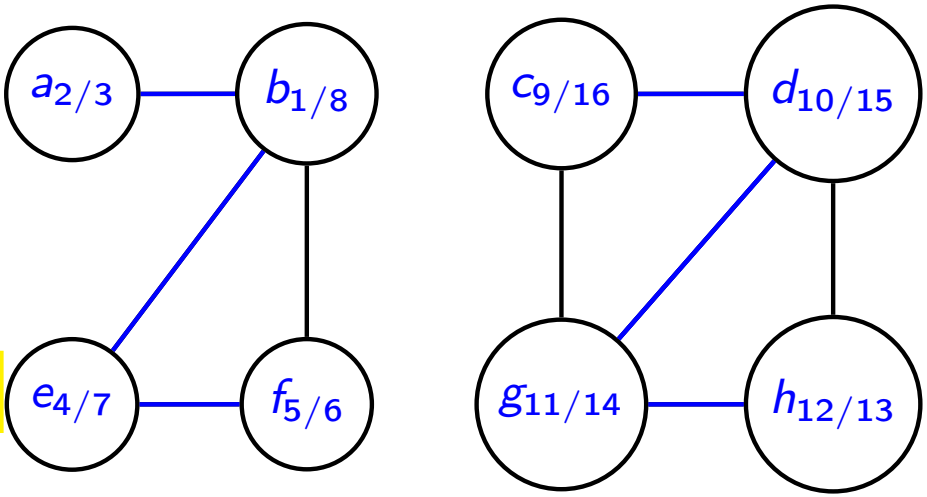
11


$a_{2/3}$
$b_{1/8}$
$c_{9/}$
$d_{10/}$
$e_{4/7}$
$f_{5/6}$
$g_{11/}$
h

12


$a_{2/3}$
$b_{1/8}$
$c_{9/}$
$d_{10/}$
$e_{4/7}$
$f_{5/6}$
$g_{11/}$
$h_{12/}$

13/16


$a_{2/3}$
$b_{1/8}$
$c_{9/16}$
$d_{10/15}$
$e_{4/7}$
$f_{5/6}$
$g_{11/14}$
$h_{12/13}$

Złożoność DFS: $\Theta(V + E)$

w. 2–5 z DFS — $O(V)$ Procedura DFSV, jak i w procedurze DFS, tak i w DFSV, jest wywoływana łącznie $|V|$ ponieważ jest wywoływana tylko dla białych wierzchołków, a każdy biały tuż po wywołaniu przestaje być biały. Na początku wszystkie wierzchołki są białe co oznacza, że procedura DFS musi być wywołana dokładnie $|V|$ razy.

w. 3–6 z DFSV tworzą pętlę, która wykonuje się $|S[u]|$ razy.

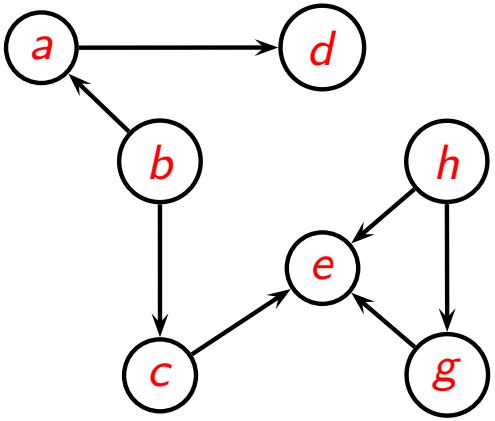
czyli:
$\sum_{u \in V} |S[u]| = \Theta(E)$

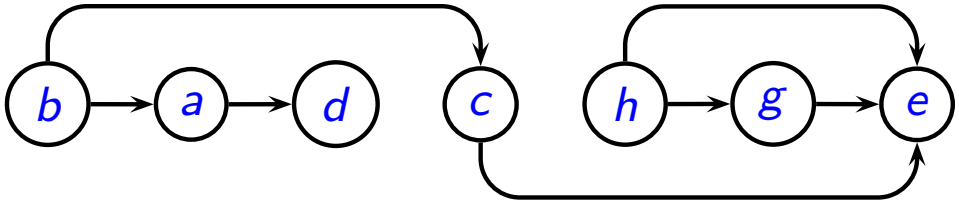
Zadania

- Zmodyfikuj tak algorytm przeszukiwania wszerz, aby znajdował najkrótszą ścieżkę pomiędzy wybranymi wierzchołkami v i w.
- Procedurę przeszukiwania w głąb można wykorzystać do sprawdzania spójności grafu. Zmodyfikuj tak tą procedurę aby etykiety wierzchołków odpowiadały pewnemu ponumerowaniu składowych spójności grafu (wierzchołki mają tą sama etykietę, gdy należą do tej samej składowej, różne gdy do różnych).
- Graf skierowany G jest pojedynczo spójny, gdy istnienie ścieżki pomiędzy parą wierzchołków v i w implikuje, że istnieje tylko jedna ścieżka pomiędzy v i w. Zbuduj algorytm, który sprawdza czy dany graf skierowany G jest pojedynczo spójny.

Sortowanie topologiczne

- Elementy zbioru $S = \{v_1, \ldots, v_n\}$ mają określony (tylko) częściowy porządek, tzn. dla niektórych par $\langle v_i, v_j \rangle$ określony jest porządek, np. $v_i < v_j$.
- Niech $G = (V, E)$ graf skierowany z $V = S$ i $E = \{\langle v_i, v_j \rangle : \text{zbiór par z określonym porządkiem}\}$.
- Zadanie: ułożyć elementy v_i w takiej kolejności, aby nie było konfliktów.





- Możliwe, gdy graf G jest grafem skierowanym acyklicznym.

Rozwiązanie:

Wykonać przeglądanie grafu w głąb DFS(G) wstawiając każdy wierzchołek na początek listy jeśli tylko zostanie przetworzony (zmiana na kolor niebieski). Taka lista spełnia warunki sortowania topologicznego.

Złożoność: Także $\Theta(|V| + |E|)$.

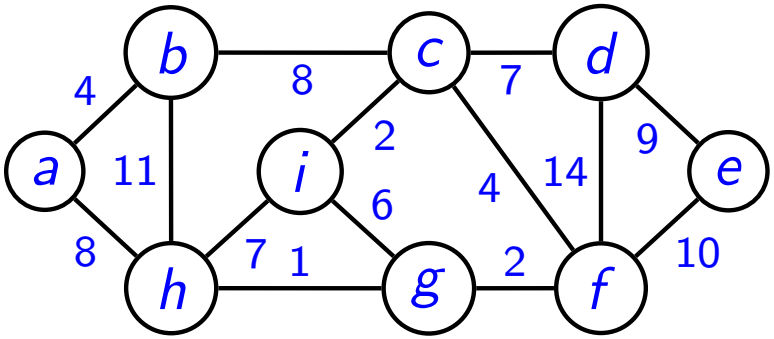
Zadania

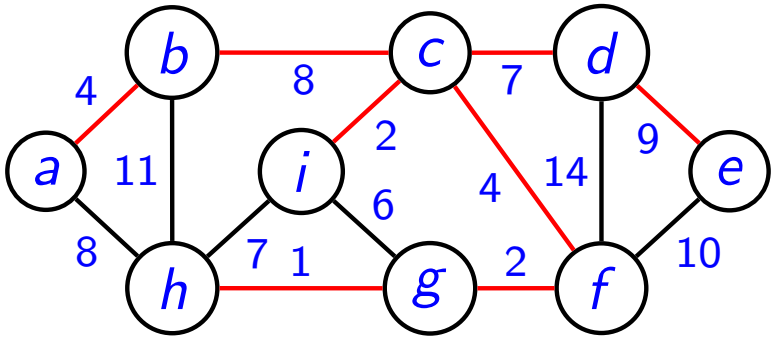
- Dla przykładowego acyklicznego grafu skierowanego prześledź poszczególne kroki algorytmu.
- Podaj algorytm sprawdzania czy graf nieskierowany ma cykl. Algorytm powinien działać w $O(V)$.

Minimalne drzewa rozpinające

Problem Mamy spójny graf G, szukamy takiego drzewa w grafie G, które zawiera wszystkie wierzchołki z G, a suma długości krawędzi T jest możliwie najmniejsza:

$$T = \arg\min_{T' \subseteq G} \sum_{(u,v) \in T'} w(u,v) \qquad T' \text{– drzewo rozpinające}$$





Ogólny algorytm budowania drzewa rozpinającego

```
DRozp(G,w) {
    A=∅;
    while(! A jest drzewem rozpinającym)
    {
        znajdź dobrą krawędź (u, v) dla A;
        A = A ∪{(u, v)};
    }
    return A;
}
```

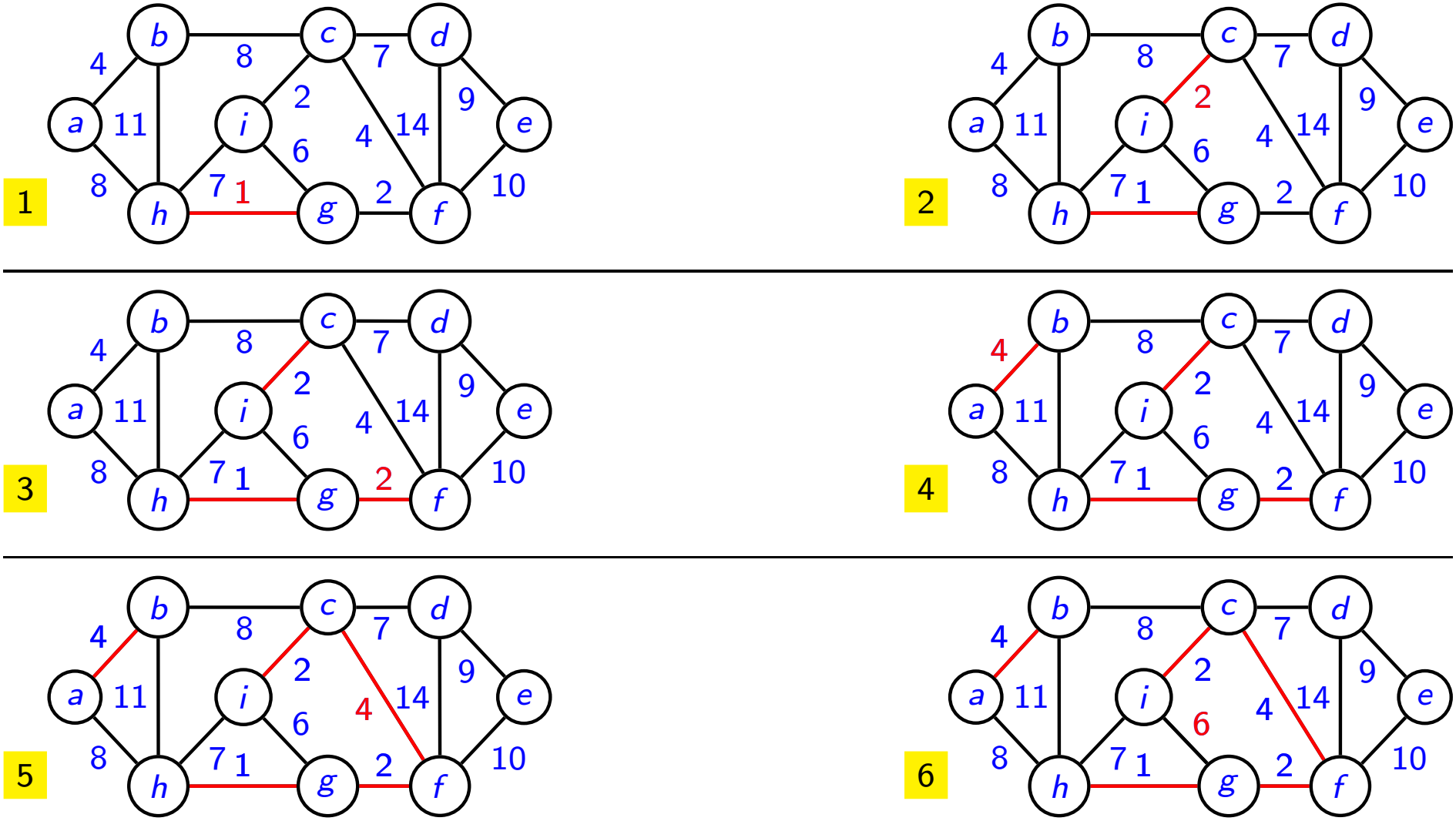
Krawędź (u, v) jest *dobra*, gdy $A \cup \{(u, v)\}$ jest podzbiorem minimalnego drzewa rozpinającego.

Strategie zachłanne: algorytm Kruskala i Prima

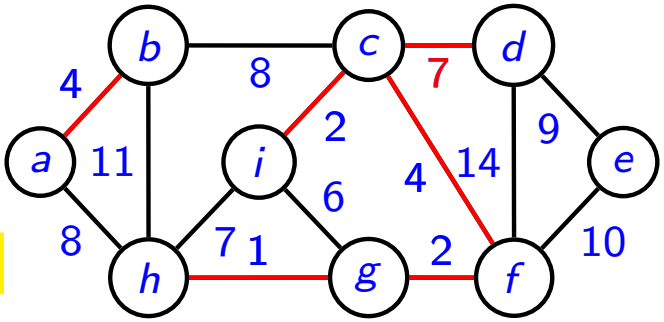
- Algorytm zachłanny: w każdym kroku wybiera optymalny ruch (ze względu na pewną funkcję oceny).
- Strategia zachłanna nie zawsze jest najlepsza (znajdująca optymalne rozwiązanie)!
- Zachłanność w budowaniu drzewa rozpinającego wyznacza rozwiązanie optymalne.

Algorytm Kruskala

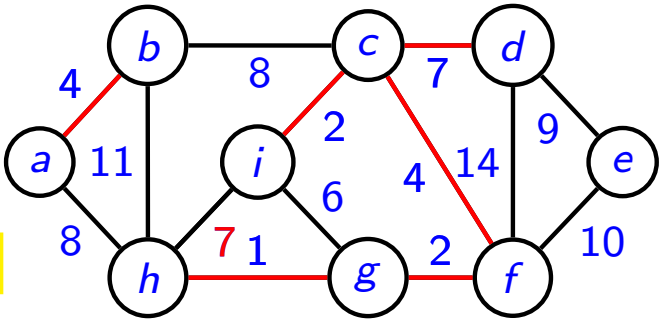
- Na początku każdy wierzchołek należy do innego zbioru (ponumerowane ...).
- Inicjujemy drzewo A jako graf $A = (V, \emptyset)$.
- Sortujemy krawędzie w E względem wag $w(u, v)$
- Dla każdej krawędzi (u, v) zgodnie z posortowaniem E sprawdzamy czy wierzchołki u i v należą do tych samych spójnych składowych (tych samych zbiorów).
- Jeśli nie, dodajemy krawędź (u, v) do A.
- Całość powtarzamy, aż wyczerpiemy wszystkie krawędzie z E.


1
2
3
4
5
6

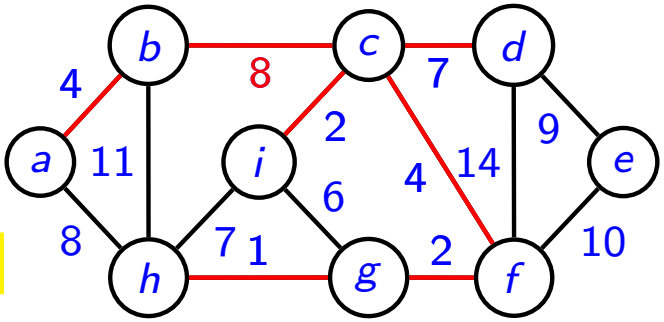
7


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
14
6
4
8
7
1
2
10

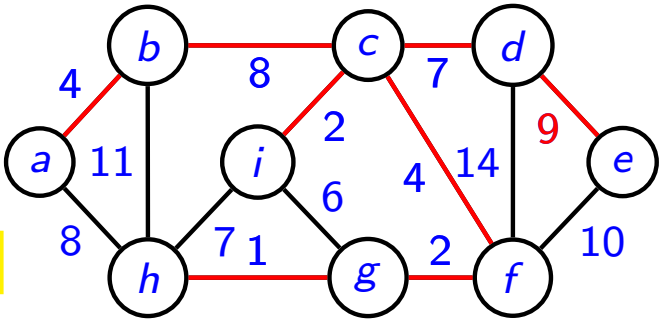
8


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
14
6
4
8
7
1
2
10

9


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
14
6
4
8
7
1
2
10

10 & . . .


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
14
6
4
8
7
1
2
10

```
Kruskal(G(V,E),w) {
    A=∅; i=1;
    foreach(v ∈ V) {
        set_id[v]=i;  i++;
    }
    sort(E,w);                          //sortowanie krawędzi względem wag
    foreach((u, v) ∈ E)          //pobieranie zgodnie z posortowaniem!
        if(Find_set_id(u) != Find_set_id(v))
        {
            A = A ∪{(u, v)};
            Union(u,v);                 //łączenie zbiorów
        }
    return A;
}
```

Złożoność algorytmu Kruskala: $O(E \log E)$

w.3–5 — $O(V)$

w.6 — $O(E \log E)$

w.7 — pętla wykonywana jest $|E|$ razy
$|E|$ krotny koszt wykonania operacji $Find\_set\_id(u)$ to $O(E\alpha(E, V))$.
Podobnie dla $Union(u, v)$, mamy $E\alpha(E, V))$.

Czyli ostatecznie $O(E \log E) = O(E \log V)$

$$\begin{array}{ll} F(1,j) = 2^j & j \geqslant 1 \\ F(i,1) = F(i-1,2) & i \geqslant 2 \\ F(i,j) = F(i-1, F(i,j-1)) & i,j \geqslant 2 \end{array}$$

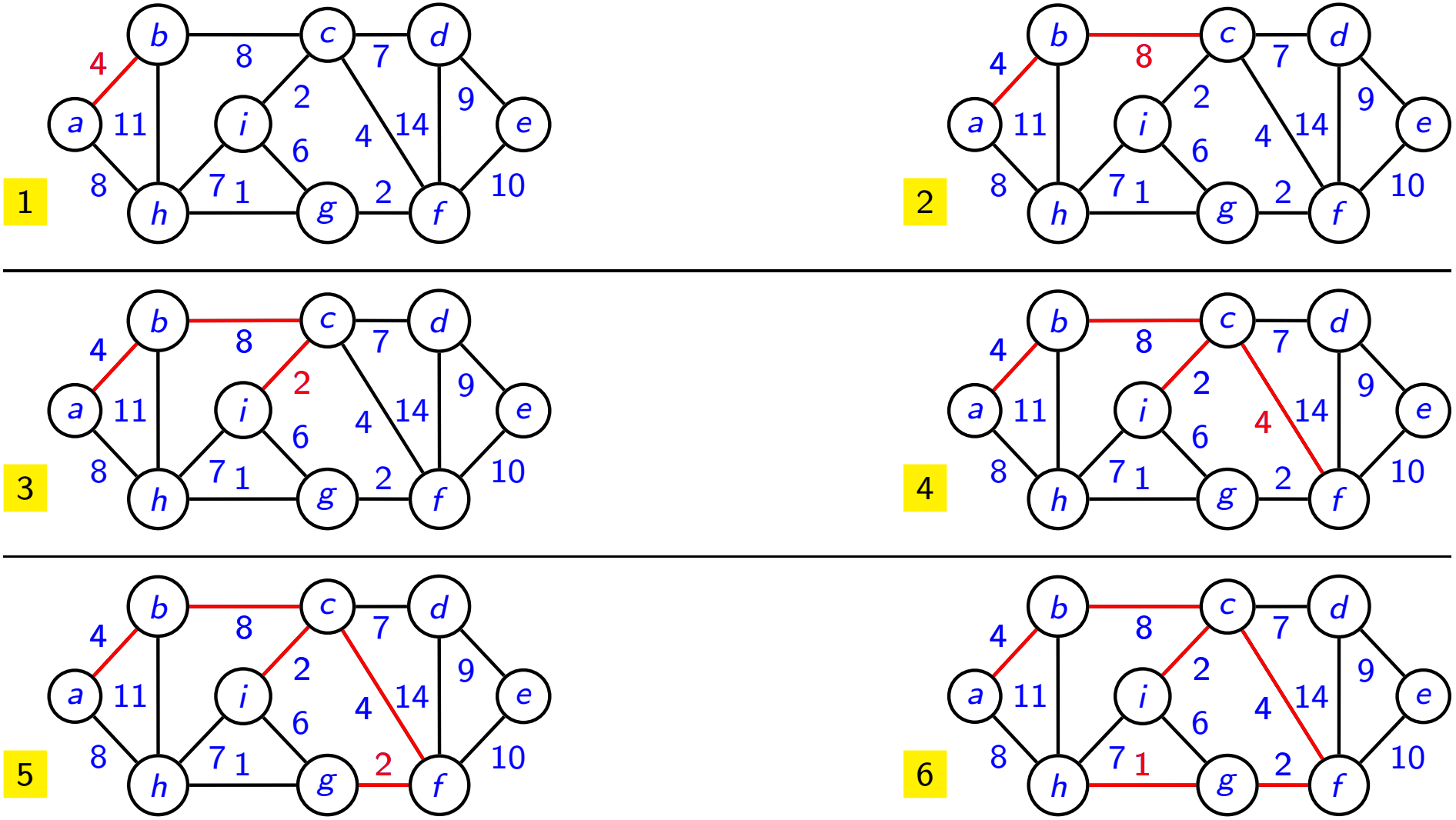
$\alpha(m, n) = \min\{i \geqslant 1 : F(i, \lfloor m/n \rfloor) > \log n\}$

$F(1,1) = 2 \quad F(2,1) = 4 \quad F(3,1) = 16$

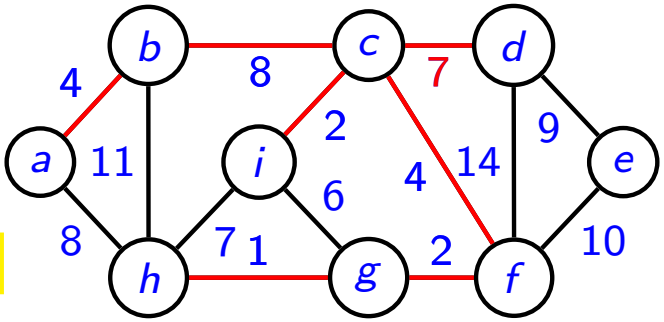
$F(4,1) = 65536 \quad F(5,1) = 2^{65536}$

Algorytm Prima

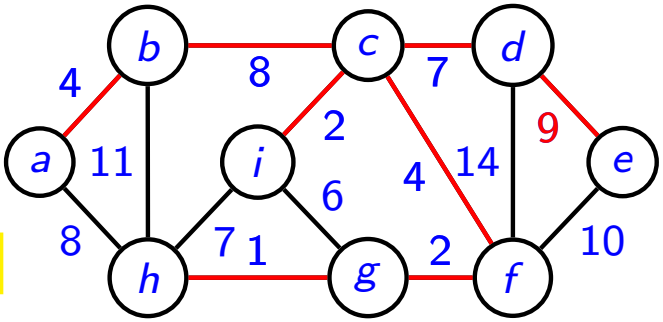
- Algorytm startuje ze wskazanego wierzchołka r.
- Pierwsza dodana krawędź do A, to najtańsza krawędź wychodząca z r.
- W kolejnych krokach celem jest dodanie najtańszej krawędzi wychodzącej z A! Ale takiej, która nie zmieni A w graf z cyklem.
- Algorytm kończy, gdy każdy wierzchołek znajdzie się w A


1
2
3
4
5
6
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
6
4
14
8
7
1
2
10

7


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
6
4
14
8
7
1
2
10

8


a
b
c
d
e
f
g
h
i
4
8
7
11
2
9
6
4
14
8
7
1
2
10

```
Prim(G(V,E),w,r) {
    Q=V;        //Q kolejka priorytetowa względem key[.]
    foreach(u ∈ Q)
        key[u]=∞
    key[r]=0;
    p[r]=NULL;
    while(Q != ∅) {
        u=ExtractMin(Q);
        A = A ∪ (u, p[u]);
        foreach(v ∈ S[u])
            if (v ∈ Q && w(u,v)<key[v]) {
                p[v]=u;
                key[v]=w(u,v);
                ChangePriority(Q,v,key[v]);
            }
    }
}
```

Wynikowe drzewo rozpinające znajduje się w tablicy p. Mamy tam opis $|V| - 1$ wybranych krawędzi.

Złożoność algorytmu Prima: $O(E + V \log V)$

w. 3–4 — $O(V)$

w. 7 — pętla wykonywana $|V|$ razy

w. 8 — $O(\log V)$ — dla Q w postaci kopca

Czyli w.7 $\cup$ w.8 — $O(V \log V)$

w. 10–14 pętla wraz z w. 7. powoduje przejrzenie wszystkich krawędzi — $|E|$.

w. 14
a) przy kopcu zwykłym — $O(\log V)$
b) przy wykorzystaniu kopców Fibonacciego — $O(1)$

Ostatecznie:

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

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

Zadania

- Dla przykładowych grafów wyznaczyć drzewa rozpinające metodą Kruskala i Primy.
- Zastanowić się nad szczegółami implementacji algorytmu Primy z użyciem zwykłych kopców (takich jak w sortowaniu przez kopcowanie).
- Jakie krawędzie grafu muszą być częścią każdego drzewa rozpinającego, a jakie nie muszą?

Najkrótsze ścieżki w grafach z wagami

- Koszt najkrótszej ścieżki z u do v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : p \text{ ścieżka z } u \text{ do } v\} & \exists_p \\ \infty & \neg\exists_p \end{cases}$$

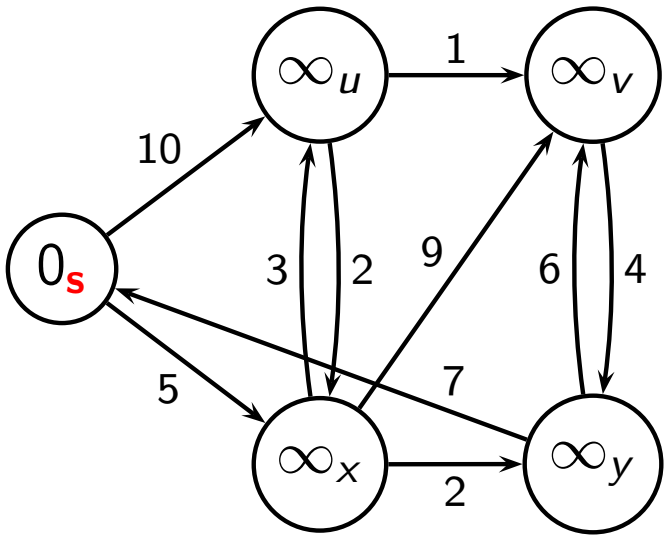
- Problemy:
  1. Najkrótsze ścieżki z jednym źródłem s
  2. Najkrótsze ścieżki z jednym wierzchołkiem docelowym (można problem odwrócić...)
  3. Najkrótsza ścieżka między parą wierzchołków (z u do v).
  4. Najkrótsze ścieżki pomiędzy wszystkimi parami wierzchołków

Procedury wstępne i oznaczenia

d[u] — odległość od źródła s

p[u] — przodek wierzchołka u (NULL = brak przodka)

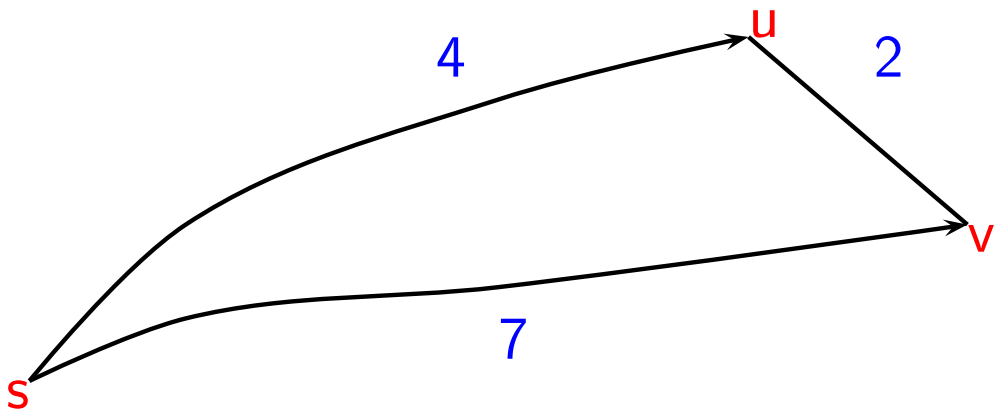
Inicjalizacja grafu

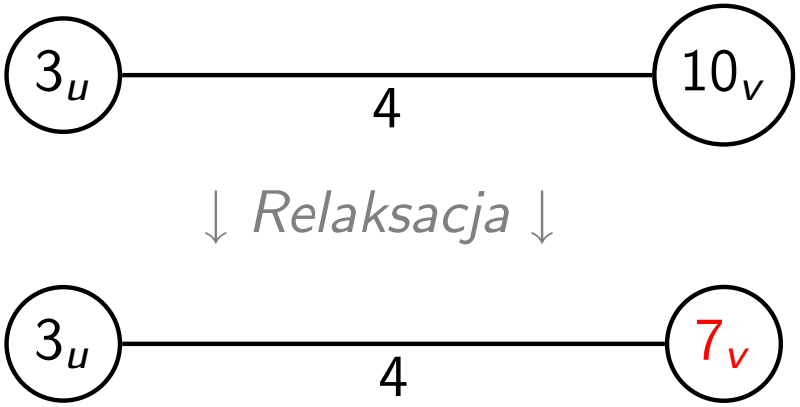


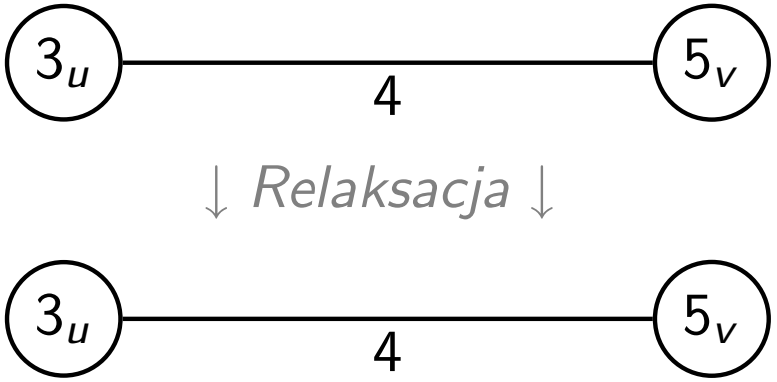
```
Init_Single_Source(G,s)
    foreach(v ∈ V)
    {
        d[v]=∞;
        p[v]=NULL;
    }
    d[s]=0;
}
```

Relaksacja

- Sukcesywnie każdy z algorytmów doszukuje się coraz lepszych rozwiązań poprzez penetracje odpowiednich ścieżek.
- Uaktualnianie najlepszych rozwiązań, to element każdego z algorytmów.



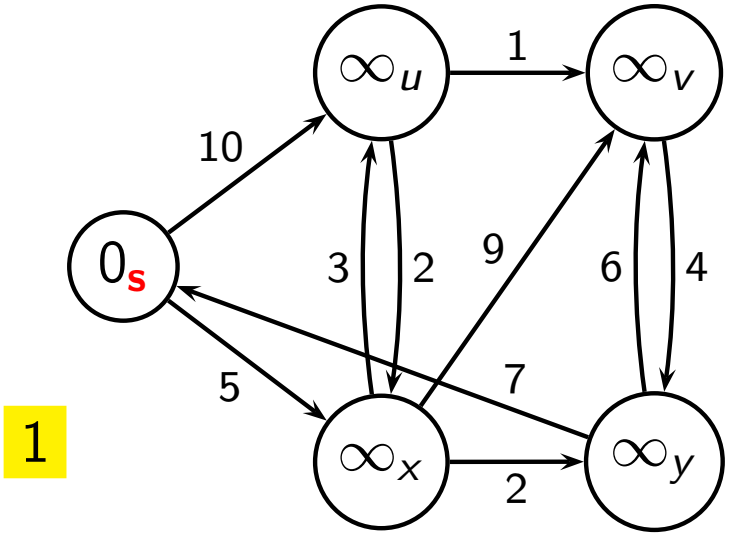


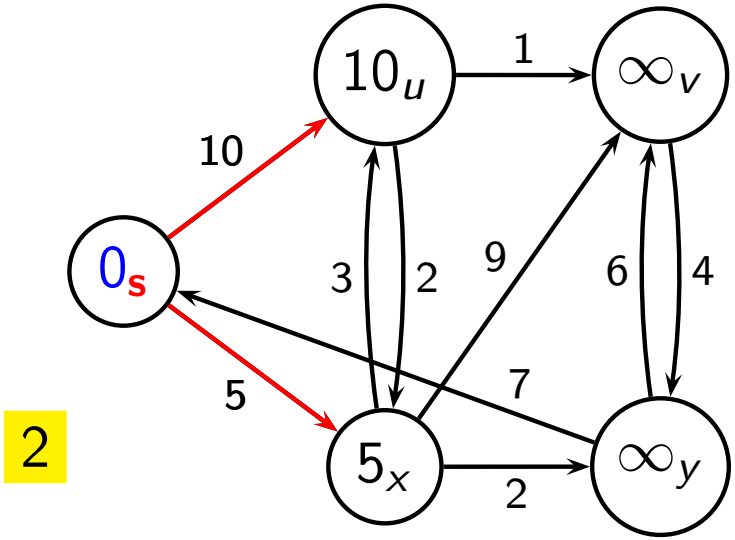


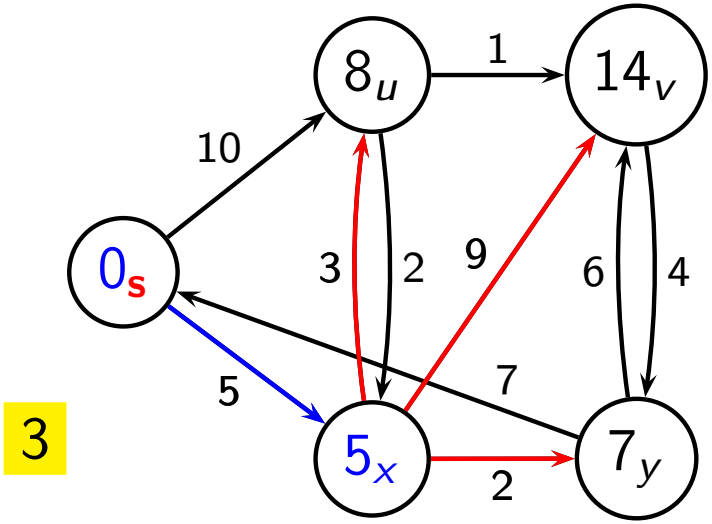

```
Relax(u,v,w) {
    if (d[v] > d[u]+w(u,v))
    {
        d[v]=d[u]+w(u,v);
        p[v]=u;
    }
}
```

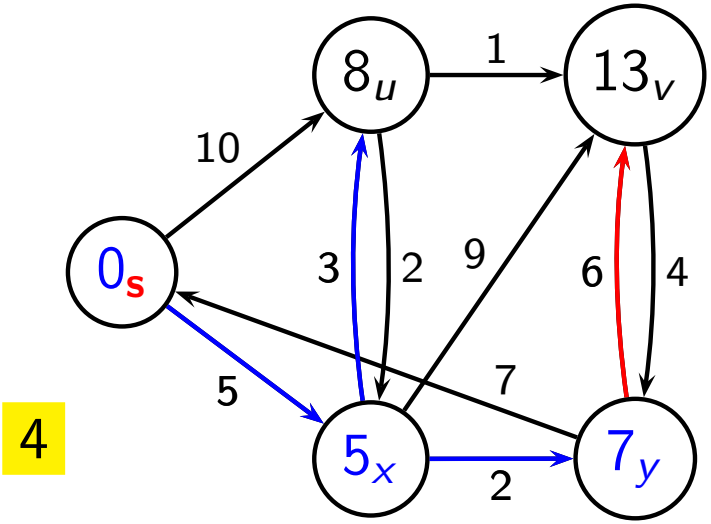
Algorytm Dijkstry

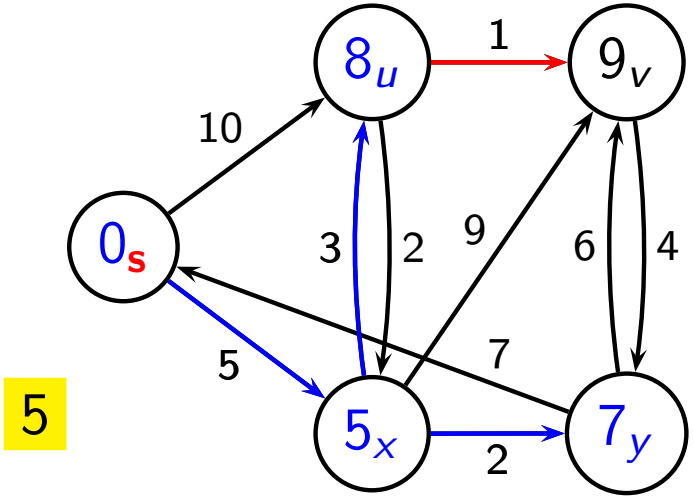
- Założenie: $w(u, v) \geqslant 0$
- Analizuje wszystkie wierzchołki z V w kolejności odległości od s
- Dla każdej krawędzi z bieżącego wierzchołka wykonuje się relaksację.

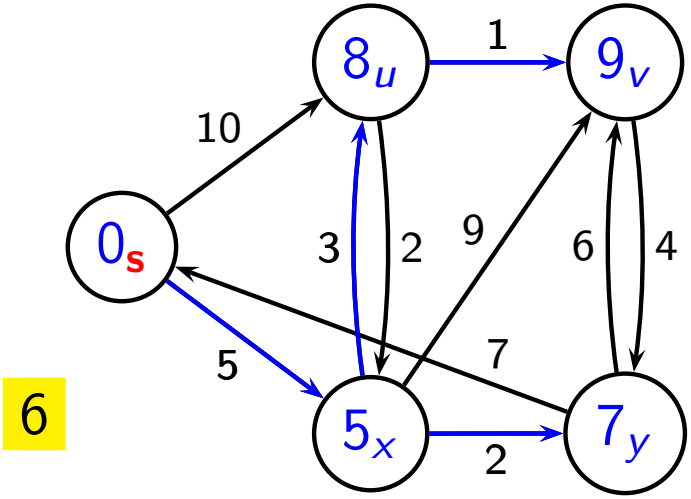





3
0_s
8_u
14_v
5_x
7_y
10
1
3
2
9
6
4
5
7
2


4
0_s
8_u
13_v
5_x
7_y
10
1
3
2
9
6
4
5
7
2


5
0_s
8_u
9_v
5_x
7_y
10
1
3
2
9
6
4
5
7
2


6
0_s
8_u
9_v
5_x
7_y
10
1
3
2
9
6
4
5
7
2

```
Dijkstra (G(V,E),w,s) {
    Init_Single_Source(G,s);
    Q=V;                //Q kolejka priorytetowa względem d[.]
    while(Q ≠ ∅)
    {
        u=ExtractMin(Q);
        foreach(v ∈ S[u])
            Relax(u,v,w);       // relaksacja zmienia kolejność w Q!
    }
}
```

Złożoność

Zależna od implementacji!

- Q — prosta reprezentacja tablicowa (lub listowa)
  w. 5 petla wykonuje się $|V|$ razy.
  w. 7 — $O(V)$
  w. 9–10 — $O(S[u])$

  co daje: $O(V^2 + E)$
- Q — reprezentacja kopcem binarnym
  w. 7 — $O(\log V)$
  w. 9–10 — $O(S[u] \log V)$

  co daje: $O((V + E) \log V)$
- Q — reprezentacja kopcem fibonacciego
  w. 7 — $O(\log V)$
  w. 9–10 — $O(S[u])$

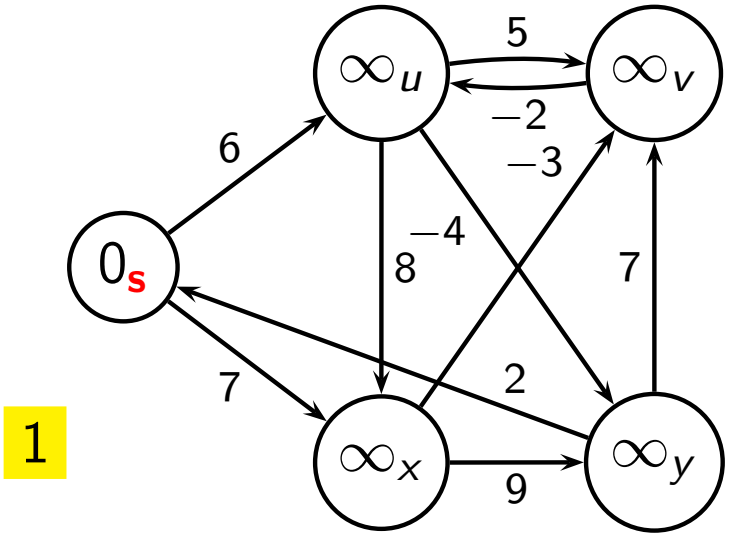
  co daje: $O(V \log V + E)$

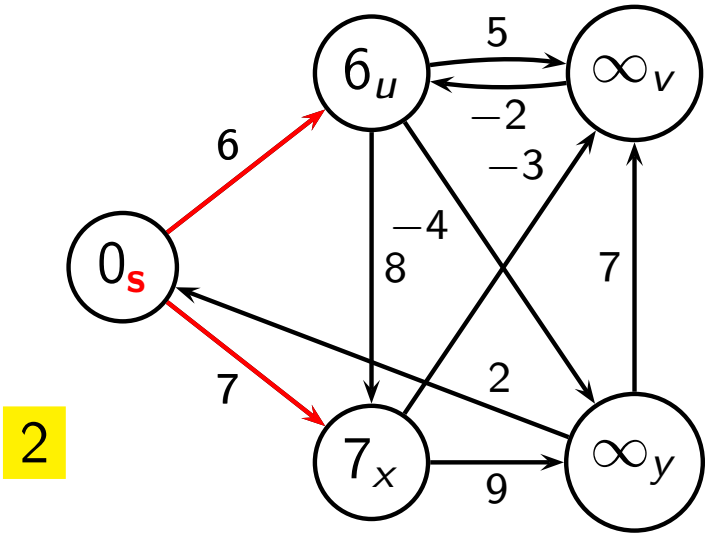
Zadania

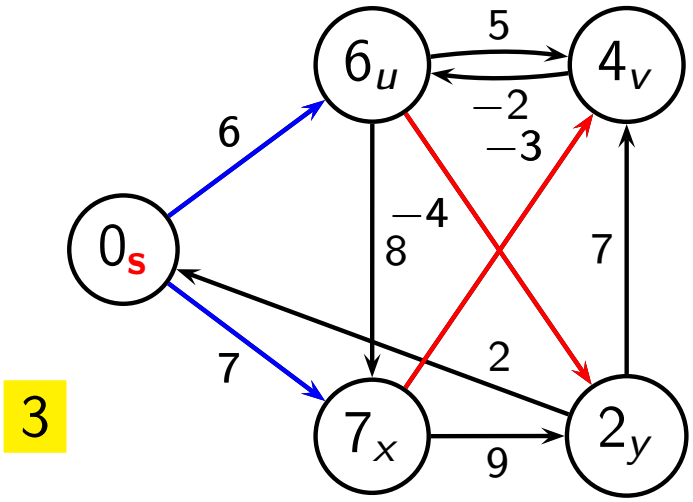
- Pokaż złe działanie algorytmu Dijkstry dla pewnych grafów z krawędziami o ujemnych wagach.
- Czy można wykorzystać algorytm gdy poszukujemy tylko najkrótszej ścieżki pomiędzy pewnymi wierzchołkami s i q?
- Przeanalizować szczegóły implementacji.

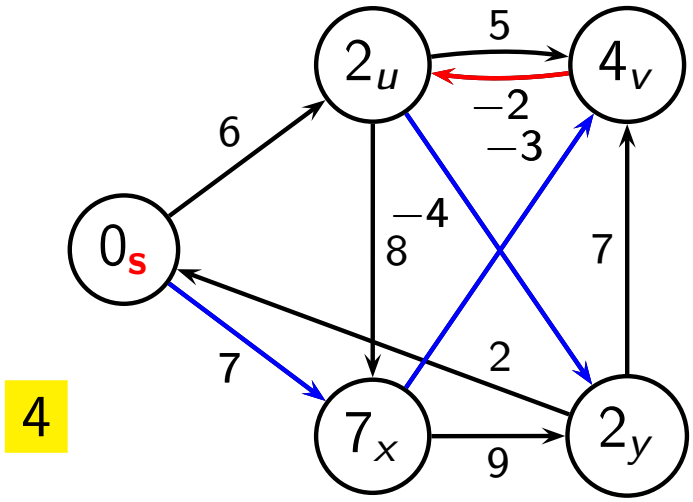
Algorytm Bellmana-Forda

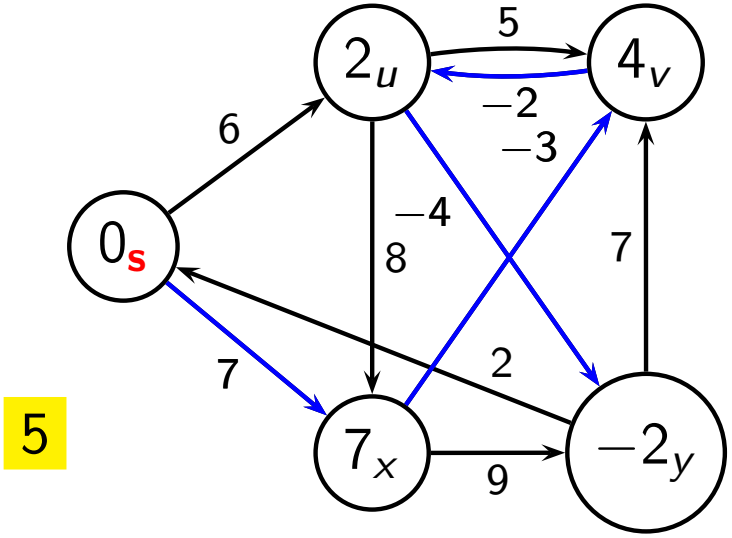
- Wagi krawędzi grafu dla tego algorytmu mogą być ujemne
- Jednak, aby algorytm zakończył się sukcesem (zbudował najkrótsze ścieżki) graf nie może zawierać ujemnych cykli (w takim przypadku algorytm zwraca FALSE)
- Obsługa ujemnych wag krawędzi kosztuje ...
- Idea: wielokrotne ($|V| - 1$) powtarzanie relaksacji dla wszystkich krawędzi.


1
0_s
∞_u
∞_v
∞_x
∞_y
6
7
5
−2
−3
−4
8
7
2
9


2
0_s
6_u
∞_v
7_x
∞_y
6
7
5
−2
−3
−4
8
7
2
9


3
0_s
6_u
4_v
7_x
2_y
6
7
5
−2
−3
−4
8
7
2
9


4
0_s
2_u
4_v
7_x
2_y
6
7
5
−2
−3
−4
8
7
2
9


5
2_u
4_v
-2
-3
6
-4
0_s
8
7
2
7
5
7_x
-2_y
9

```
Bellman_Ford(G(V,E),w,s) {
    Init_Single_Source(G,s);
    for(i=1; i<=|V| − 1; i++)
        foreach((u, v) ∈ E)
            Relax(u,v,w);
    foreach((u, v) ∈ E)
        if(d[v] > d[u]+w(u,v))
            return FALSE;
    return TRUE;
}
```

Złożoność: $O(VE)$

w. 2 — $\Theta(V)$

w. 3 petla wykonywana $|V| - 1$ razy

w. 4–5 — $O(E)$

w. 6–8 — $O(E)$

Co daje: $O(VE)$

Można nieco usprawnić. Gdy w.7 będzie wykonywany jeśli tylko, któraś z relaksacji ostatniego przebiegu pętli z w.4 powiodła się.

Zadania

Zmodyfikuj algorytm tak, aby dla wszystkich wierzchołków v leżącej na ujemnym cyklu wartość $d[v] = -\infty$.

Algorytm Floyda-Warshalla Najkrótsze ścieżki pomiędzy wszystkimi wierzchołkami grafu

- Raz jeszcze programowanie dynamiczne!
- d_{ij}^k — długość najkrótszej ścieżki z i do j, której wierzchołki wewnętrzne ścieżki należą do zbioru $\{1, 2, \ldots, k\}$.

$$d_{ij}^k = \begin{cases} w_{ij} & k = 0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & k \geqslant 1 \end{cases}$$

Ścieżka: $u_i, v_1, v_2, \ldots, v_{k-1}, v_k, u_j$

```
Floyd_Warshall(W) {
    D^0=W;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(i==j || w[i,j]==∞) p[i,j]=NULL;
            else if(i!=j && w[i,j]<∞) p[i,j]=i;
    for(k=1;k<=n;k++)
        for(i=1;i<=n; i++)
            for(j=1;j<=n;j++)
                if(d_ij^{k-1} ⩽ d_ik^{k-1} + d_kj^{k-1}) {
                    d_ij^k = d_ij^{k-1};        p_ij^k = p_ij^{k-1};
                } else {
                    d_ij^k = d_ik^{k-1} + d_kj^{k-1}; p_ij^k = p_kj^{k-1}; }
    return D^n;
}
```

p_{ij} — poprzednik wierzchołka j na najkrótszej ścieżce z i.

Złożoność: $\Theta(n^3)$

Zadania

- Ile macierzy D potrzeba rzeczywiście aby algorytm Floyda-Warshalla mógł działać poprawnie?
- Dlaczego algorytm Floyda-Warshalla znajduje najkrótsze ścieżki?
- Napisz procedurę wypisującą ścieżkę z wierzchołka i do wierzchołka j.

Drzewa czerwono–czarne

- Przeszukiwanie binarne tablicy uporządkowanej:
  - Podziel na pół.
  - Szukaj w lewej bądź prawej części w zależności od wartości klucza.
- Drzewa przeszukiwań binarnych
  - Drzewo binarne.
  - Operacje: search, predecessor, successor, minimum, maximum, insert, delete.
- Drzewa zrównoważone: utrzymywanie zbliżonej wysokości poddrzew na każdym poziomie.

Zrównoważenie DCzCz:

Każda ścieżka (z korzenia do liścia) jest co najwyżej dwa razy dłuższa niż każda inna.

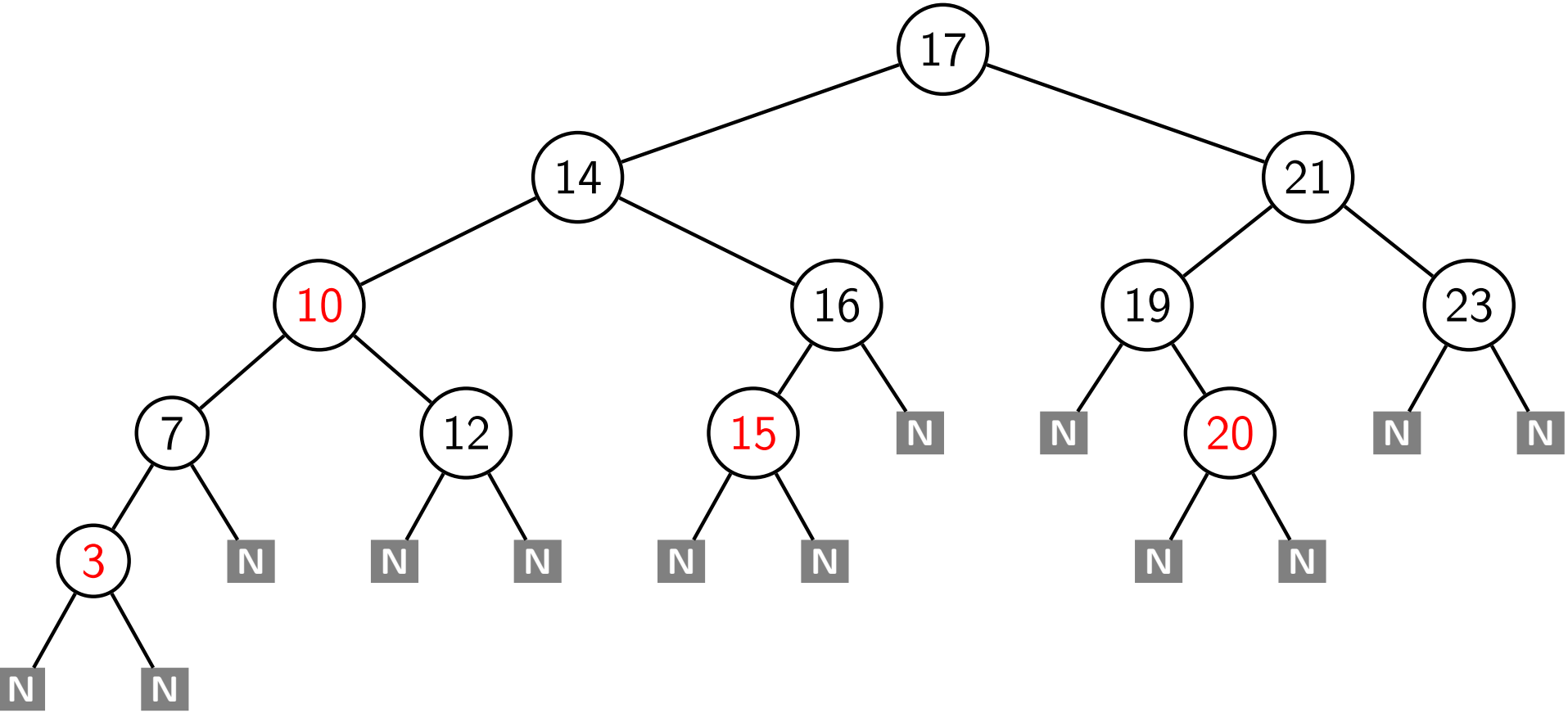
Warunki nałożone na DCzCz:

- Węzeł jest czerwony albo czarny.
- Każdy liść jest czarny.
- Jeśli węzeł jest czerwony, to synowie są czarni.
- Wszystkie ścieżki z ustalonego węzła do liści maja tyle samo węzłów czarnych.

Pola węzła: left, right, p (ojciec), color (RED, BLACK), key

DCzCz — NULL=wartownik!

Przykład DCzCz



Max wysokość drzewa CzCz: $2\log(n+1)$

Koszty operacji na drzewie CzCZ

search, predecessor, successor, minimum, maximum, insert, delete

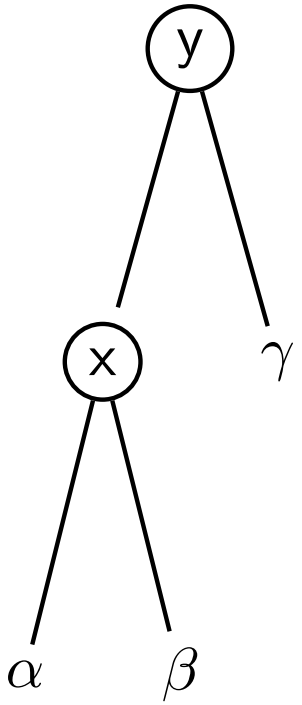
$$O(\log n)$$

Operacje na drzewie mogą być wykonywane naprzemiennie!

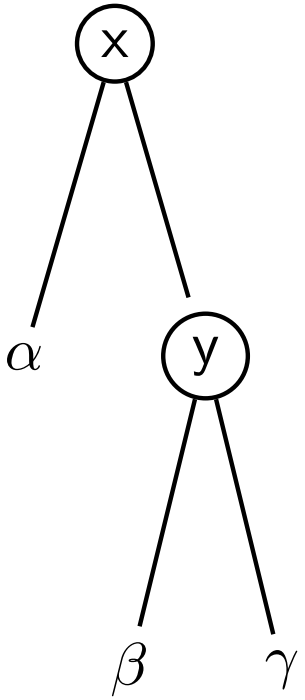
Operacje rotacji

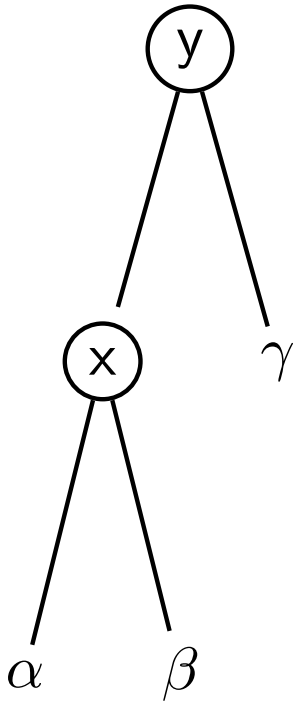
Cel: Zmiana zrównoważenia przy jednoczesnym utrzymaniu porządku.

RightRotate(T,y) $\rightarrow$

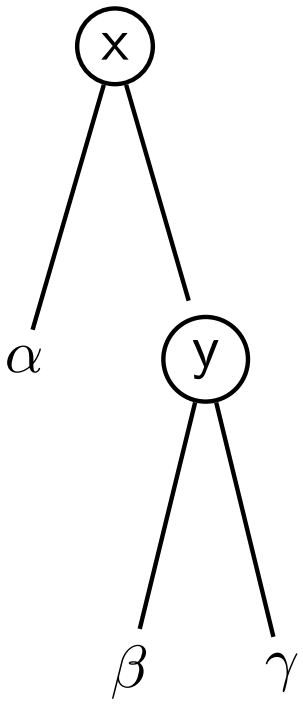


$\leftarrow$ LeftRotate(T,x)





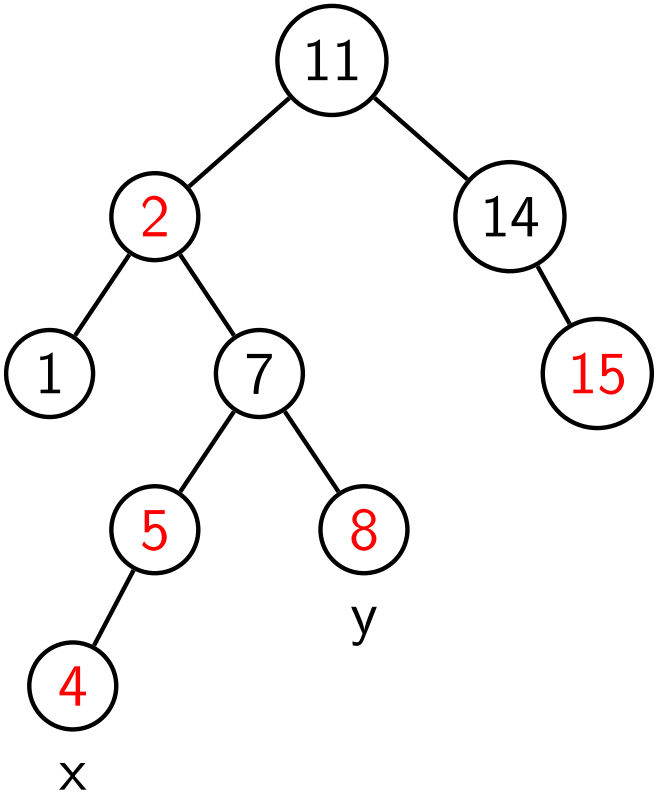

```
LeftRotate(T,x) {
    y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->p = x;
    y->p = x->p;
    if (x->p == NULL)
        root = y;
    else
    if (x == x->p->left)
        x->p->left = y;
    else
        x->p->right = y;
    y->left = x;
    x->p = y;
}
```

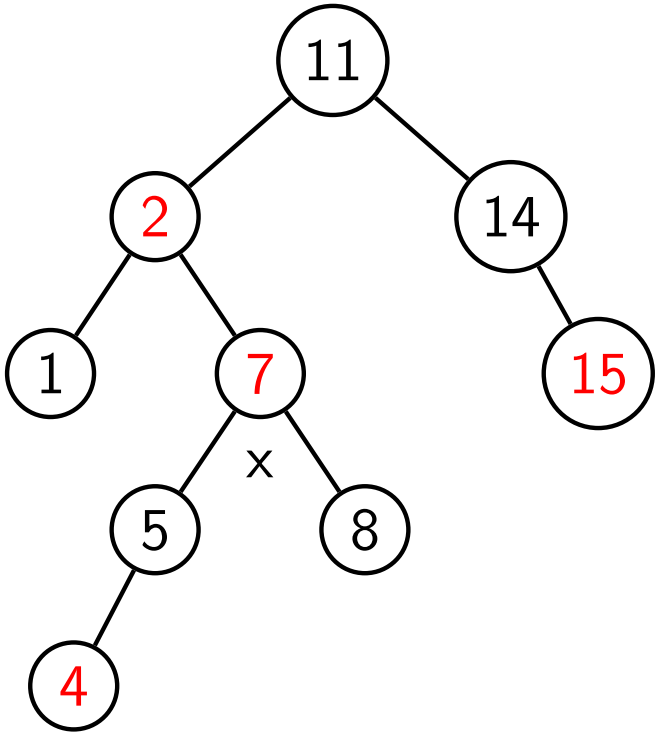


Wstawianie do DCzCz

```
RBInsert(T,x) {
 TreeInsert (T,x);
x->color = RED;
while(x!=root &&
   x->p->color == RED)
    if (x->p == x->p->p->left)
    {
       y = x->p->p->right;
       if (y->color == RED)
       {
          x->p->color = BLACK;
          y->color = BLACK;
          x->p->p->color = RED;
          x = x->p->p;
       }
       else {
          if (x == x->p->right)
          {
             x = x->p;
             LeftRotate(T,x);
          }
          x->p->color = BLACK;
          x->p->p->color = RED;
          RightRotate(T, x->p->p)
       }
    }
    else {
       XYZ (Viceversa(left-right))
    }
root->color = BLACK;
}
```

P1 w. 9–14 **y==czerwony**

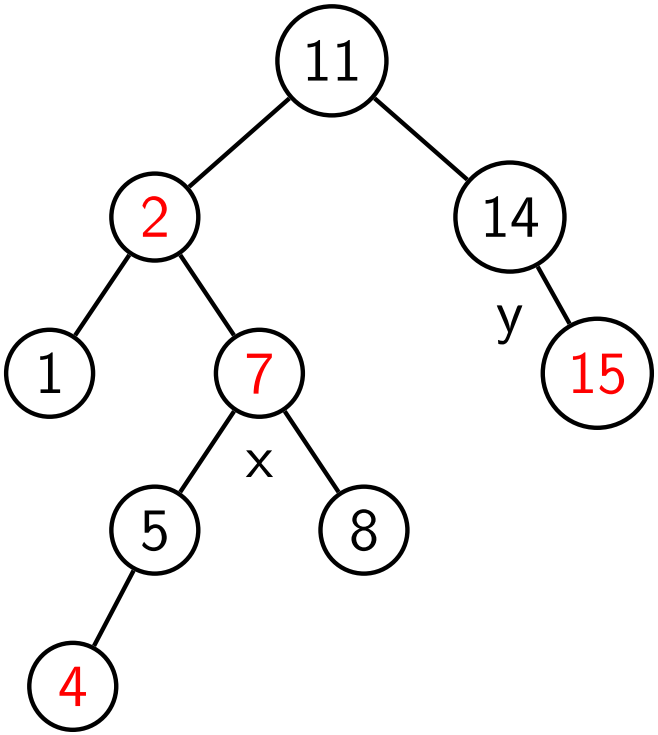


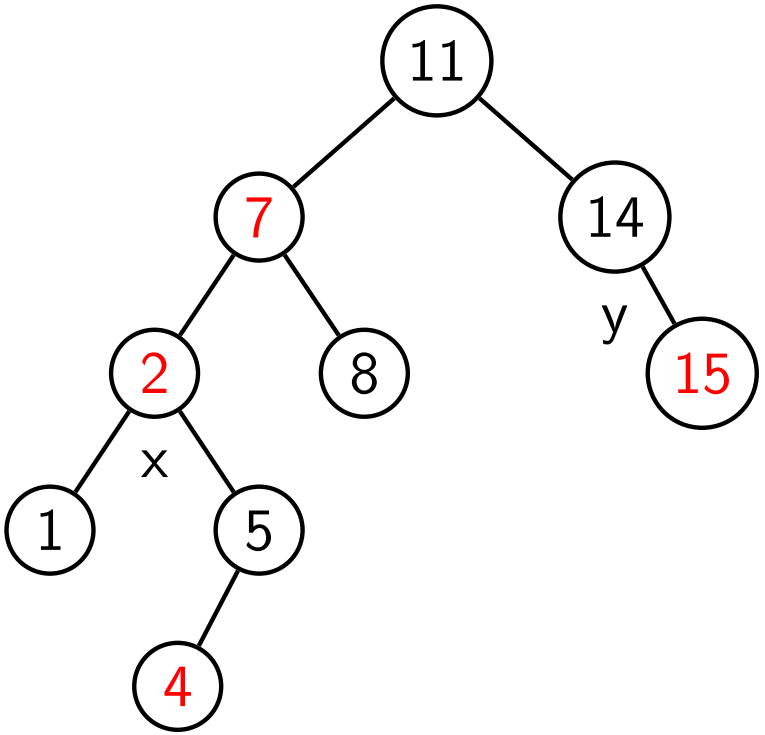


ojciec = czarny, wujek = czarny, x o 2 do góry

P1 $\rightarrow$ [P1,P2,P3,$\emptyset$]

P2 w. 17–20 y==czarny, x jest prawym dzieckiem

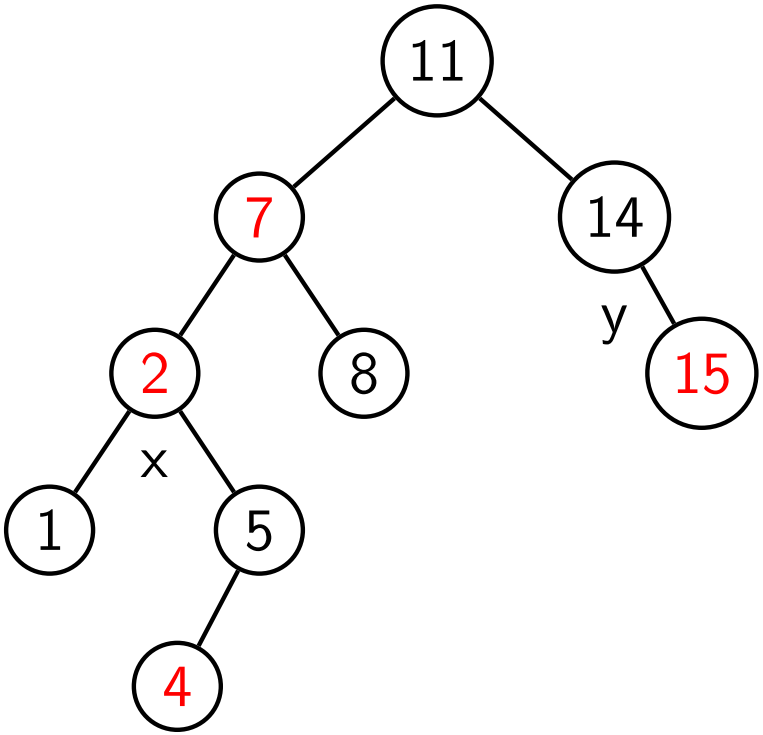


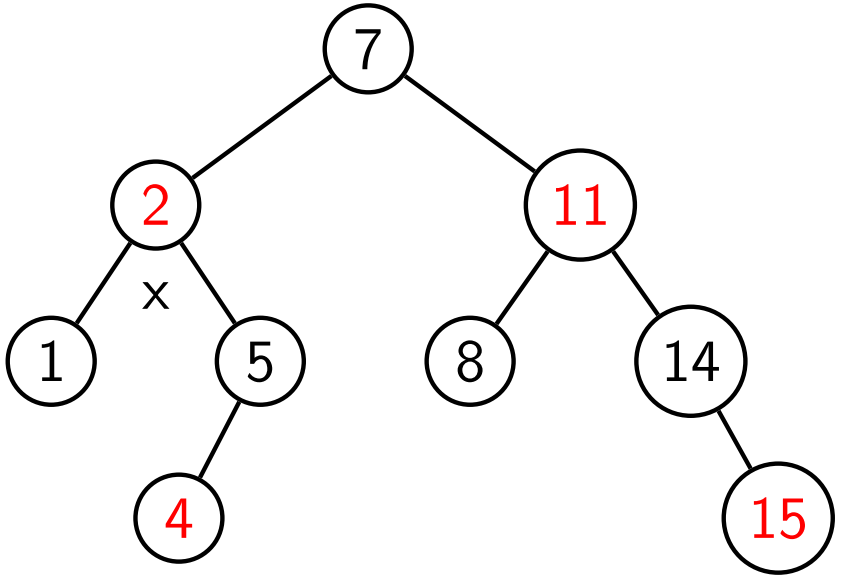


x o 1 do góry, rotacja left x

P2 → [P3]

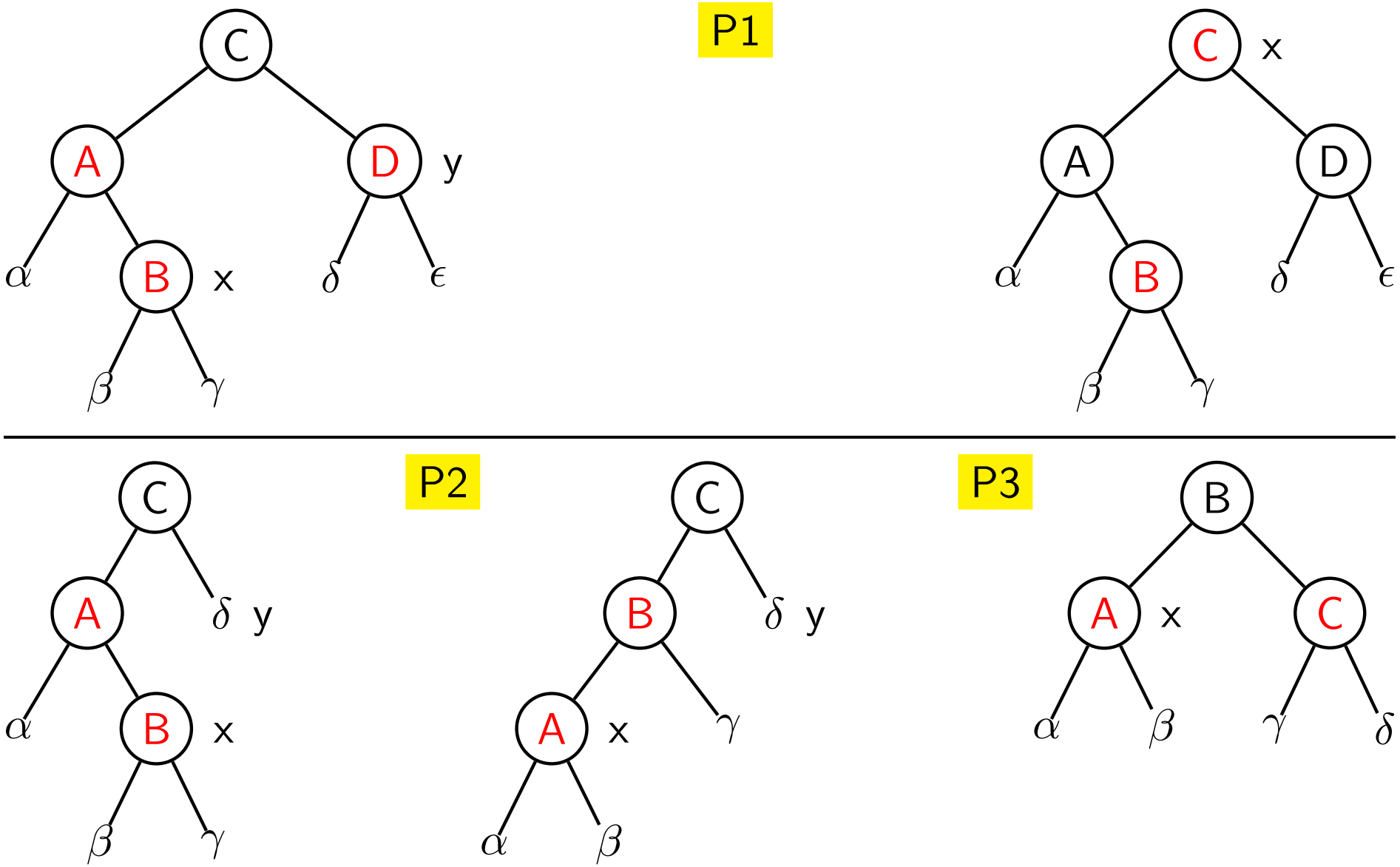
P3 w. 22–24 y==czarny, x jest lewym dzieckiem





zamiana kolorów ojciec–dziadek, rotacje right dziadek, koniec

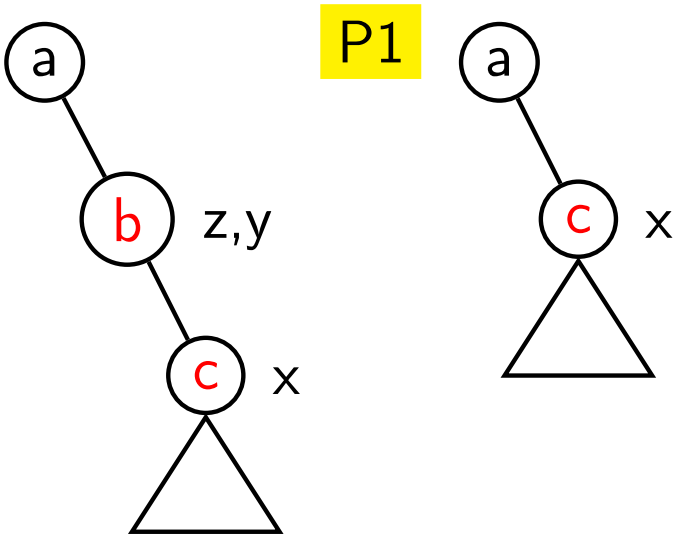
P3 $\rightarrow$ [$\emptyset$]

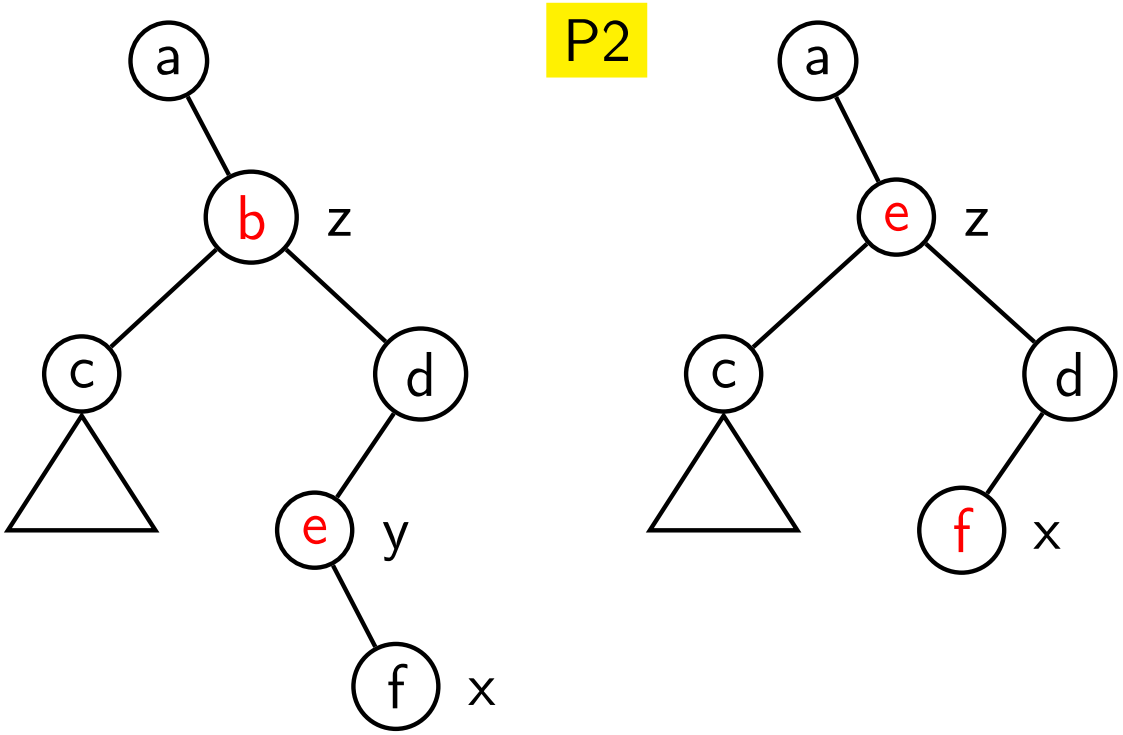

P1
C
A
D
y
B
x
α
β
γ
δ
ϵ
C
x
A
D
B
α
β
γ
δ
ϵ
P2
C
A
δ y
B
x
α
β
γ
C
B
δ y
A
x
γ
α
β
P3
B
A
x
C
α
β
γ
δ

```
XYZ() {
  y = x->p->p->left;
  if (y->color == RED)
  {
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
  }
  else {
    if (x == x->p->left)
    {
      x = x->p;
      RightRotate(T,x);
    }
    x->p->color = BLACK;
    x->p->p->color = RED;
    LeftRotate(T, x->p->p)
  }
}
```

Usuwanie z DCzCz

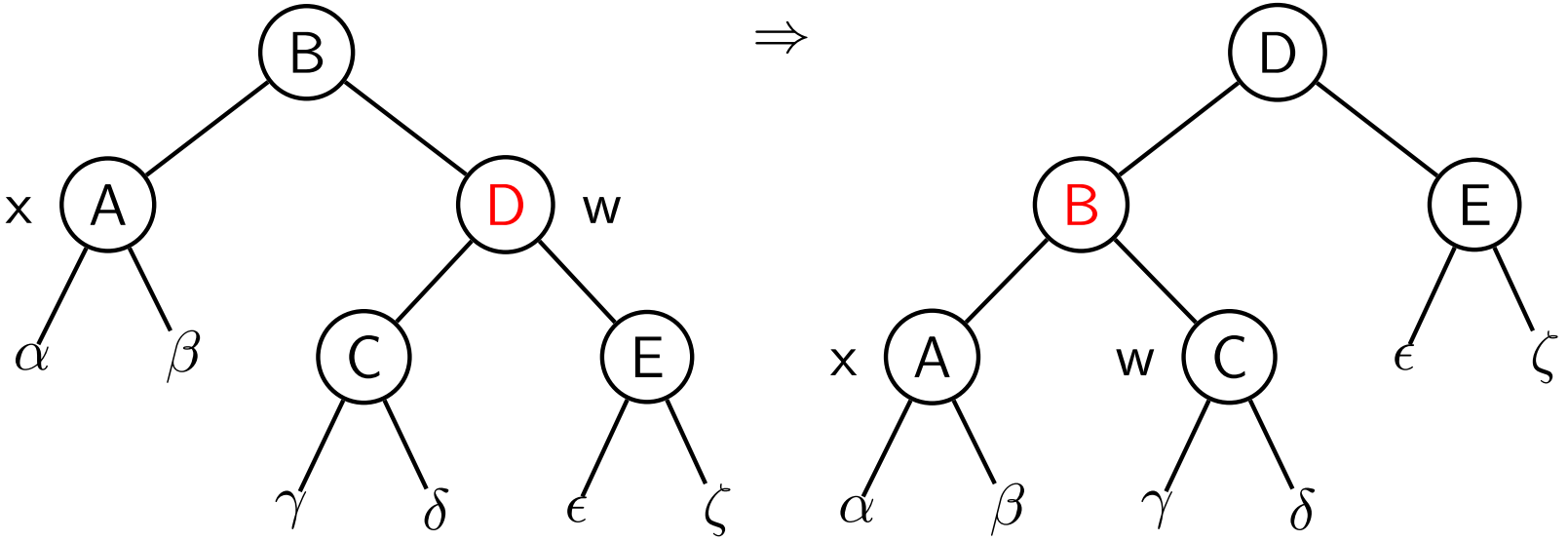
```
RBDelete(T,z) {
    if (z->left == NULL ||
      z->right == NULL)
        y = z;
    else y=Successor(z);
    if (y->left != NULL)
        x = y->left;
    else
        x = y->right;
    x->p = y->p;
    if (y->p == NULL)
        root = x;
    else
    if (y == y->p->left)
        y->p->left = x;
    else
        y->p->right = x;
    if (y!=z)
        z->key = y->key;
    if (y->color == BLACK)
        RBDeleteFixup(T,x);
}
```


a
b
z,y
c
x
P1
a
c
x


a
b
z
c
d
e
y
f
x
P2
a
e
z
c
d
f
x

```
RBDeleteFixup(T,x) {
while(x != root &&
      x->color==BLACK)
 if (x == x->p->left)
 {
      w = x->p->right;
      if (w->color == RED)
      {
           w->color = BLACK;
           w->p->color = RED;
           LeftRotate(T,x->p);
           w = x->p->right;
      }
      if (w->left->color==BLACK &&
          w->right->color==BLACK)
      {
           w->color = RED;
           x = x->p;
      }
      else
      {
           if (w->right->color == BLACK)
           {
                w->left->color = BLACK;
                w->color = RED;
                RightRotate(T,w);
                w = x->p->right;
           }
           w->color = x->p->color;
           x->p->color = BLACK;
           w->right->color = BLACK;
           LeftRotate(T,x->p);
           x = root;
      }
 }
 else
      ViceVersa  ( left <->right)
x->color=BLACK;
}
```

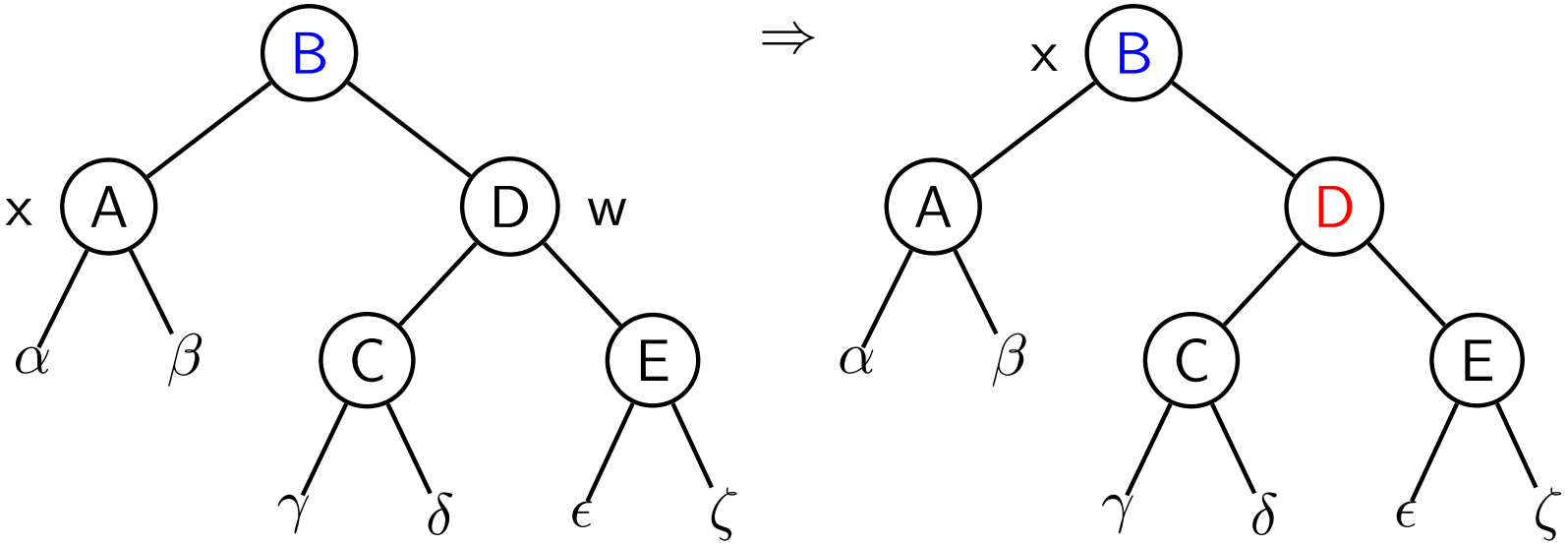
P1: w. 9–13 D==czerwony



P1 → [P2,P3,P4]

B↔D, rotacja left B

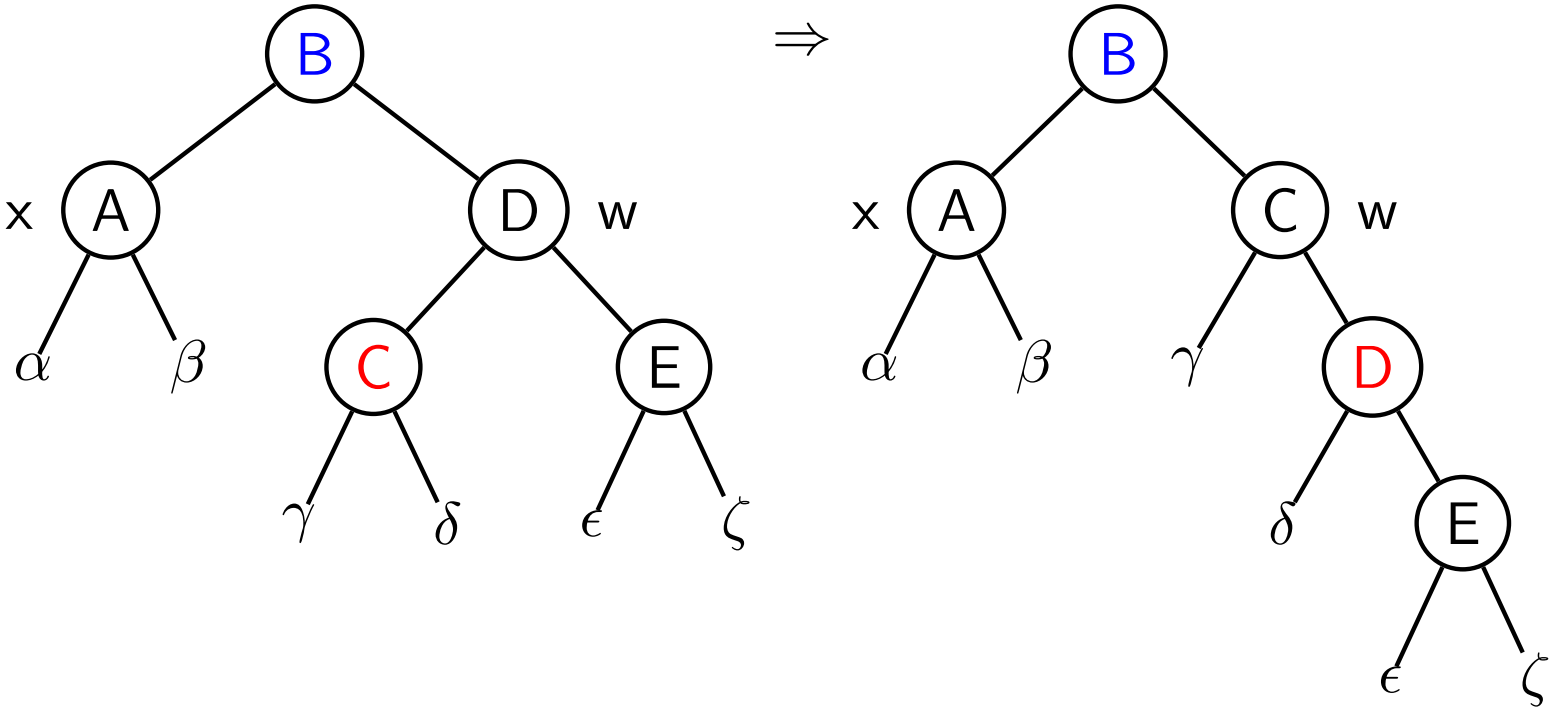
P2 w. 18–19 D,C,E==czarne



P2 → [P1,P2,P3,P4,∅]

D=czerwony, przesuń x o 1 do góry

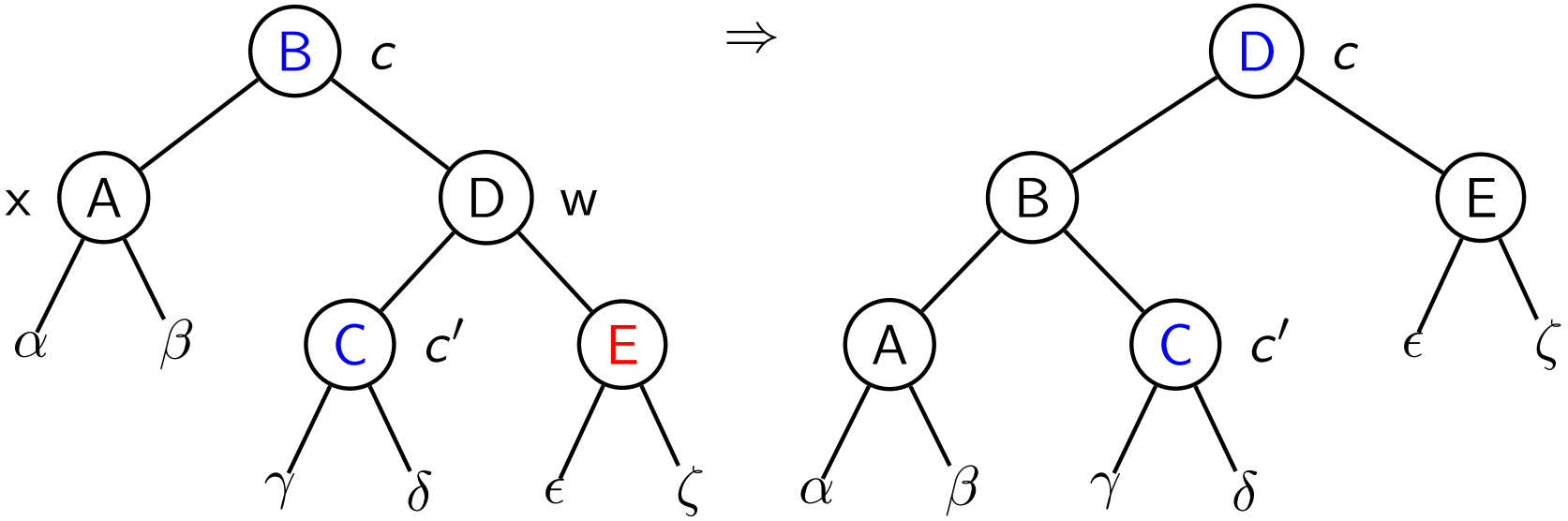
P3 w. 24–27 D,E==czarne, C==czerwony



P3 → [P4]

C↔D, rotacja right D, w do C

P4 w. 29–33 D==czarny, E==czerwony



nowy $x = root$

P4 $\rightarrow [\emptyset]$ B↔D, E=czarny, x- -, rotacja left B

dodatkowa jednostka czarno została wchłonięta

Zadania

- Zapisz procedury: search(wezel, klucz), predecessor(wezel), successor(wezel), minimum(wezel), maximum(wezel).
- Pokaż, że w drzewie czerwono–czarnym każda ścieżka z korzenia do liścia jest co najwyżej dwa razy dłuższa niż każda inna.
- Prześledź działanie algorytmu wstawiania elementów do drzewa CzCz na przykładzie ciągu liczb: $33, 28, 19, 10, 6, 3, 17$.
- Prześledź usuwanie elementów z drzewa skonstruowanego w poprzednim zadaniu.