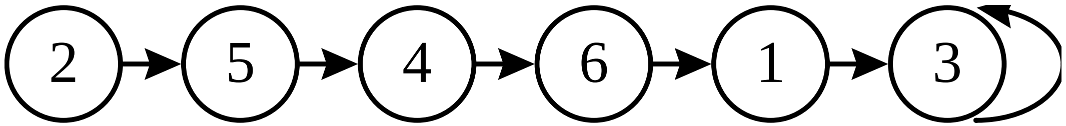
List

Petr Hanáček

No inserted slides

Cor 2007/8 Pre

Práce se seznamy

- Algoritmy pracují s vázaným seznamem
- Každý prvek má hodnotu $\underline{v}_i$ a ukazatel na následníka Succ[i]
- List: 

- Pole succ:

– Succ	3	5	3	6	4	1
– Index	1	2	3	4	5	6

Úloha: Predecessor computing

```
Algorithm:
        for i = 1 to n do in parallel
                Pred[Succ[i]] = i

Pozn.:
Je třeba zvlášť ošetřit první a poslední prvek seznamu
```

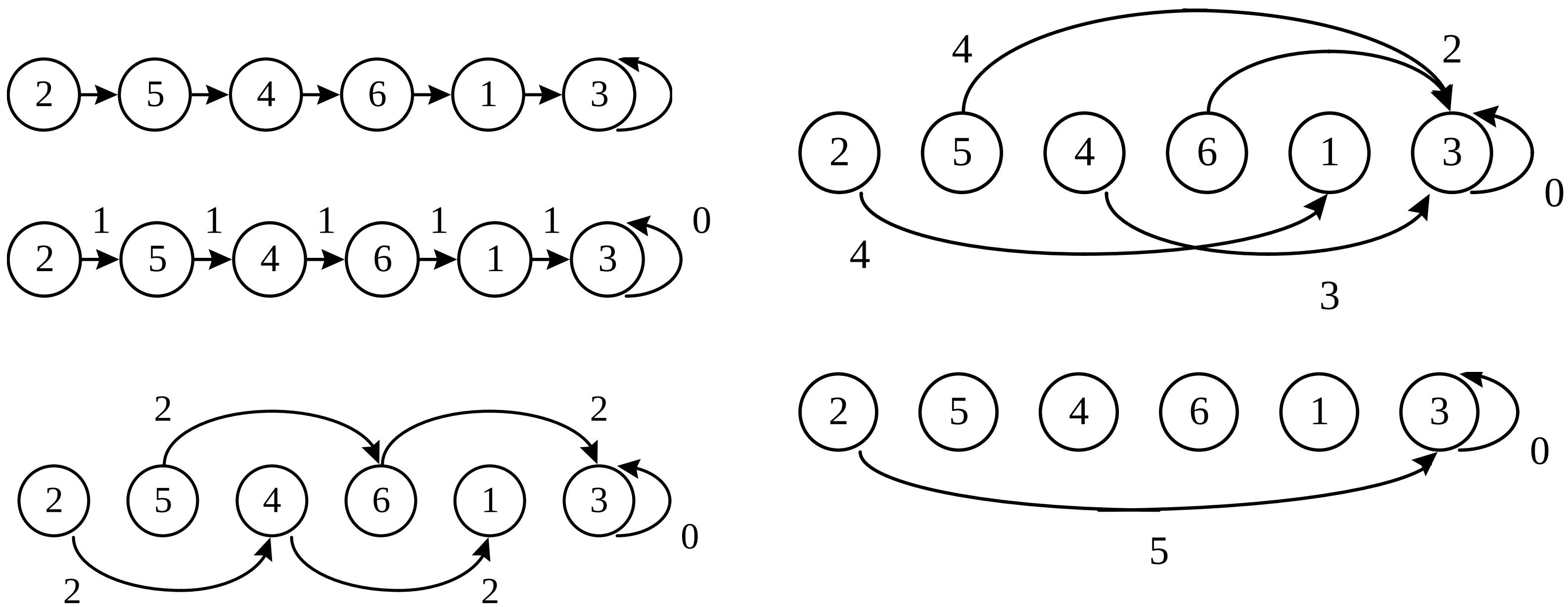
- Analýza
- $t(n) = O(c)$ $p(n) = n$
- $c(n) = O(n)$

```
for i = 1 to n do in parallel
        if i<>Succ[i] then
                Pred[Succ[i]] = i;
```

Úloha: List ranking

- Nalezni pořadí (rank) prvků seznamu (vzdálenost od konce seznamu)
- Sekvenční algoritmus má složitost O(n)
- Použitá technika je zdvojování cesty (path doubling)

```
Algorithm
Input: array Succ[1..n]
Output: array Rank[1..n]

for i=1 to n do in parallel
       if Succ[i]=i   then Rank[i] = 0
                      else Rank[i] = 1
       for k = 1 to log n do
              Rank[i] = Rank[i] + Rank[Succ[i]]
              Succ[i] = Succ[Succ[i]]
       end for
end for
```



Example:

	Succ	Rank
initially	3 5 3 6 4 1	1 1 0 1 1 1
step1	3 4 3 1 6 3	1 2 0 2 2 2
step2	3 1 3 3 3 3	1 4 0 3 4 2
step3	3 3 3 3 3 3	1 5 0 3 4 2
index	1 2 3 4 5 6	1 2 3 4 5 6

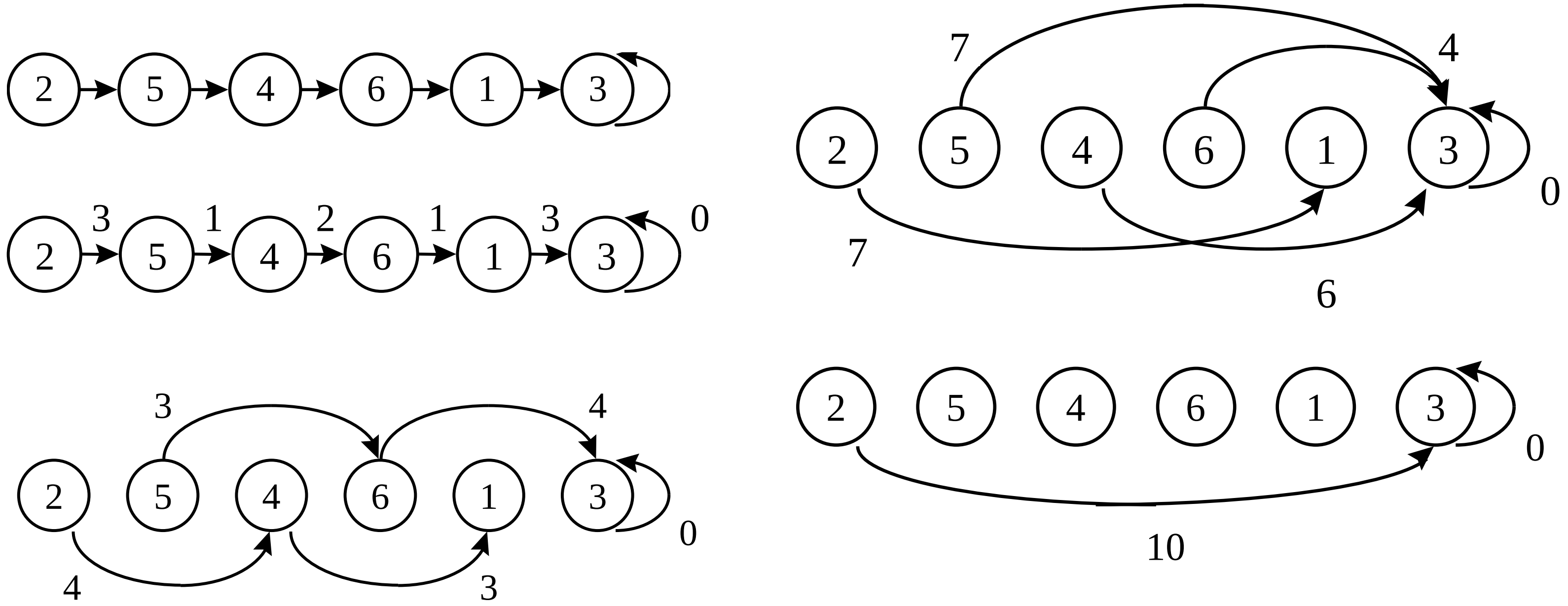
- Analýza
- $t(n) = O(\log n)$ $p(n) = n$
- $c(n) = O(n.\log n)$ – není optimální

Úloha: Parallel suffix sum

- Vypočítává součet suffixů (podobně jako suma prefixů) seznamu
- Suffix – podseznam mezi prvkem a koncem seznamu
- Vstupy: Pole hodnot V, binární asociativní operátor ⊕

```
Algorithm
V = [vn-1, ... v1,0]
for i = 1 to n do in parallel
        if Succ[i] = i then Val[i] = 0 /* neboli neutrální prvek operace ⊕ */
                       else Val[i] = vi
        for k = 1 to log n do
                Val[i] = Val[i] ⊕ Val[Succ[i]]
                Succ[i] = Succ[Succ[i]]
        end for
        if Val[last] <> 0 then Val[i] = Val[i] ⊕ Val[last]

end for
```

Příklad




```
Example:
                     Succ                  Suffix sum
initially        3 5 3 6 4 1           3   3   0   2   1   1
step1            3 4 3 1 6 3           3   4   0   3   3   4
step2            3 1 3 3 3 3           3   7   0   6   7   4
step3            3 3 3 3 3 3           3  10   0   6   7   4
index            1 2 3 4 5 6           1   2   3   4   5   6
```

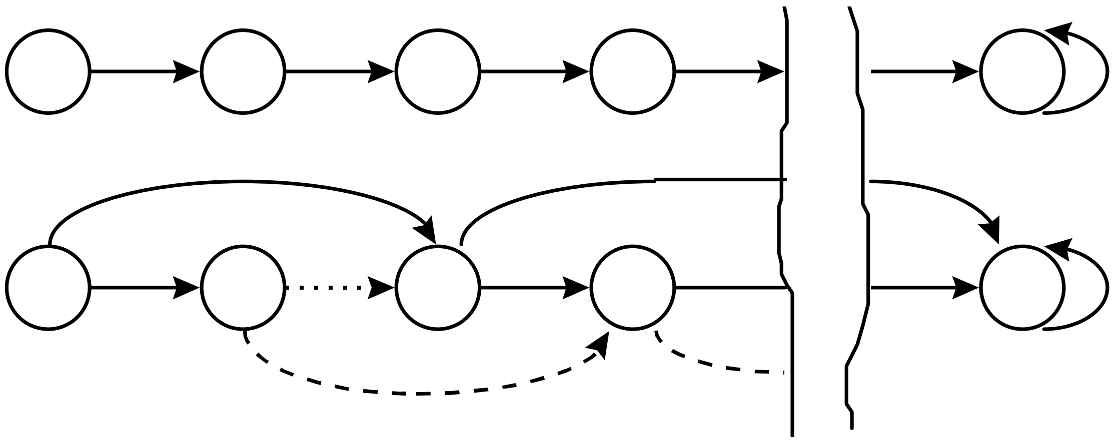
- *na konci musíme upravit hodnotu posledního prvku*

– *poslední prvek - pokud některý z procesorů už doskákal na konec, tak už nic nemusí číst, ostatní procesory ale ještě hodnotu požadují $\Rightarrow$ hodnota posledního prvku by se přičítala několikrát $\Rightarrow$ inicializujeme na 0 (neutrální prvek) a ten se přičítá*

- <u>Analýza</u>
- t(n) = O(log n)
- c(n) = O(n.log n)

List ranking revisited

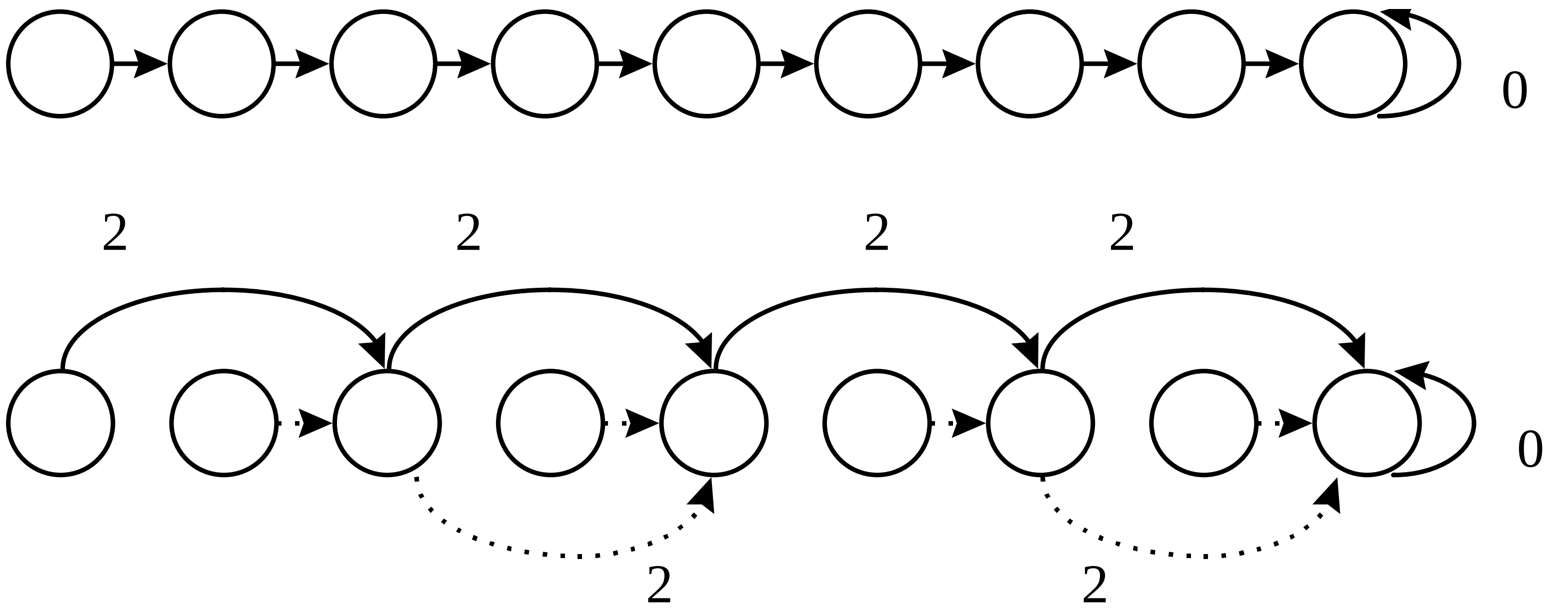
- Základní algoritmus list ranking *(Wyllie's algorithm)* nemá optimální cenu c(n) = O(n.log n)
- Příčina – je prováděna zbytečná práce



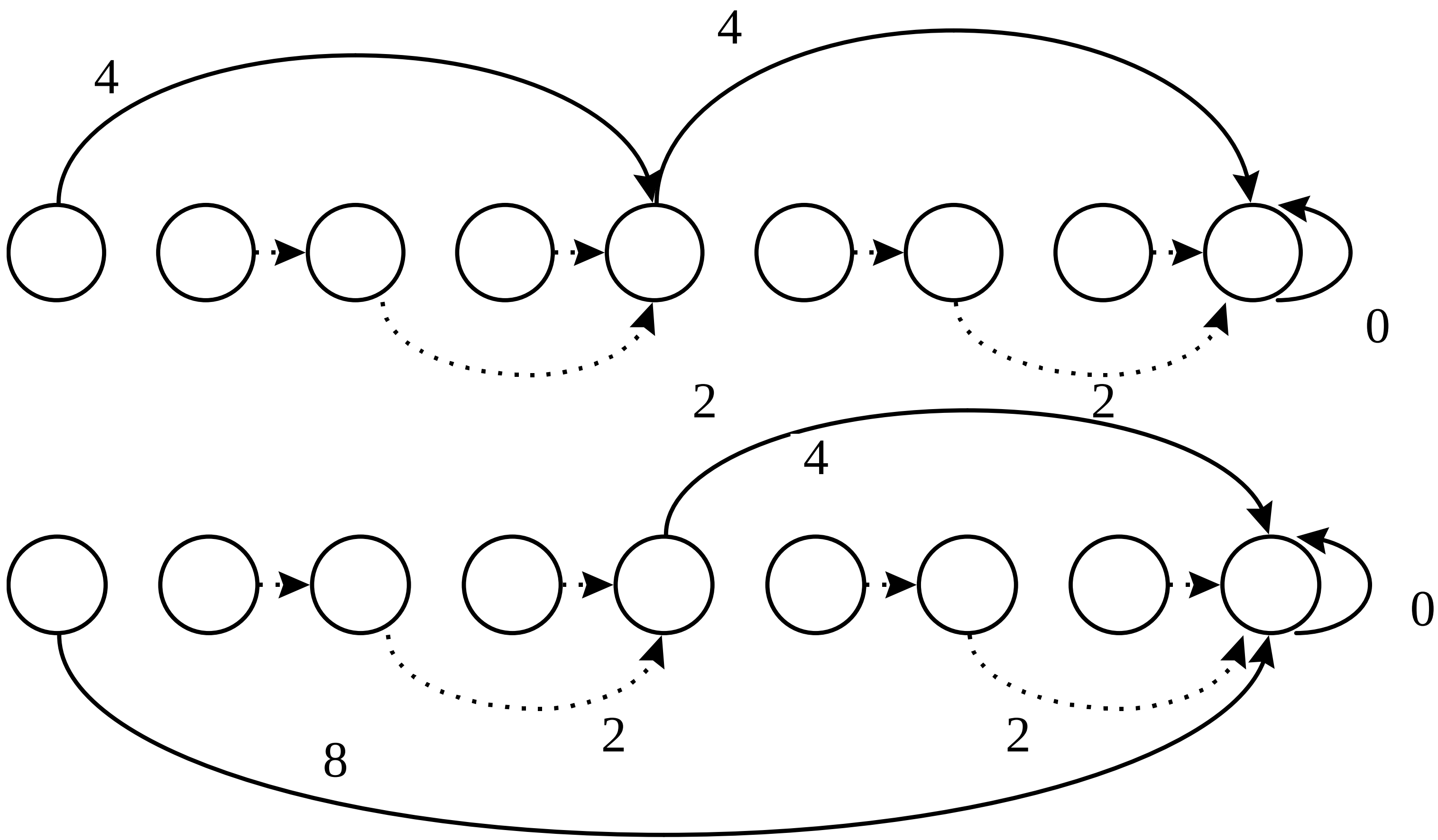
Řešení: v každém kole přestane pracovat polovina procesorů

Jumping phase 1

- *Jumping phase:*

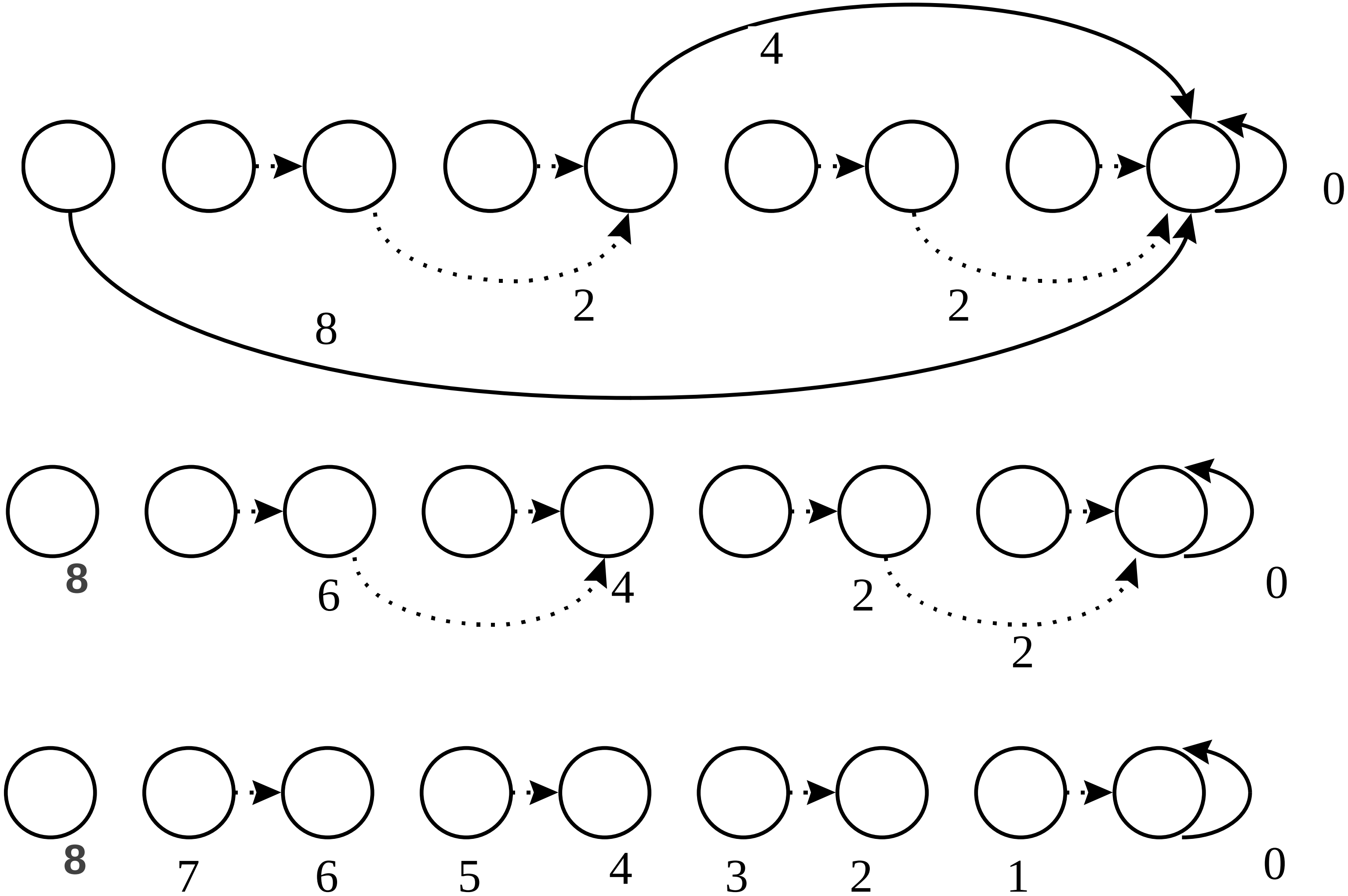


Jumping phase 1



- *Reconstruction phase:*
  - *...*

Reconstruction phase



Random mating

- Optimalizovaný algoritmus list ranking
- Každý procesor si hodí korunou a přiřadí si pohlaví (male/female)
- Pokud je procesor female a jeho následník je male, pak se prvek přeskočí (jump over) a procesor se uvolní

```
procedure RandomMate
  for i = 1 to n do in parallel
        rank[i] = 1
        active[i] = True   /*if True - processor is working, False - processor is waiting */
  end for
  t = 1  /* global variable */
  while succ[head] <> tail do in parallel
        if active[i] and succ[i] <> tail then
                sex[i] = Random(M, F)
                if sex[i] = F and sex[succ[i]] = M then
                        time[succ[i]] = t
                        active[succ[i]] = False
                        rank[i] = rank[i] + rank[succ[i]]
                        succ[i] = succ[succ[i]]
                end if
                t = t + 1
        end if
  end while
  /* end of jumping phase */
```

```
 /* reconstuction phase */    while t > 0 do in parallel
              if time[i] = t and succ[i] <> tail then
                             rank[i] = rank[i] + rank[succ[i]]
              end if
              t = t - 1
       end while
end
```

- <u>Analýza</u>
  - Pro velké <u>n</u> je velmi malá pravděpodobnost, že počet kol algoritmu bude větší než c.log n

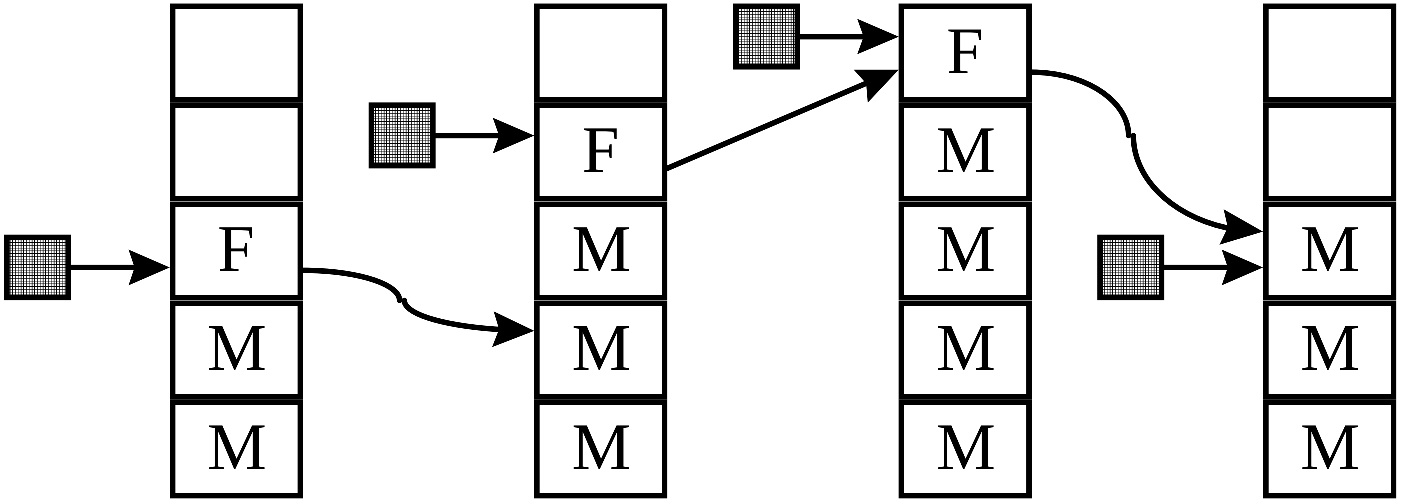
    t(n) = O(log n)
  - Cena

    c(n) = n.log n → není optimální
  - Cena (množství práce)

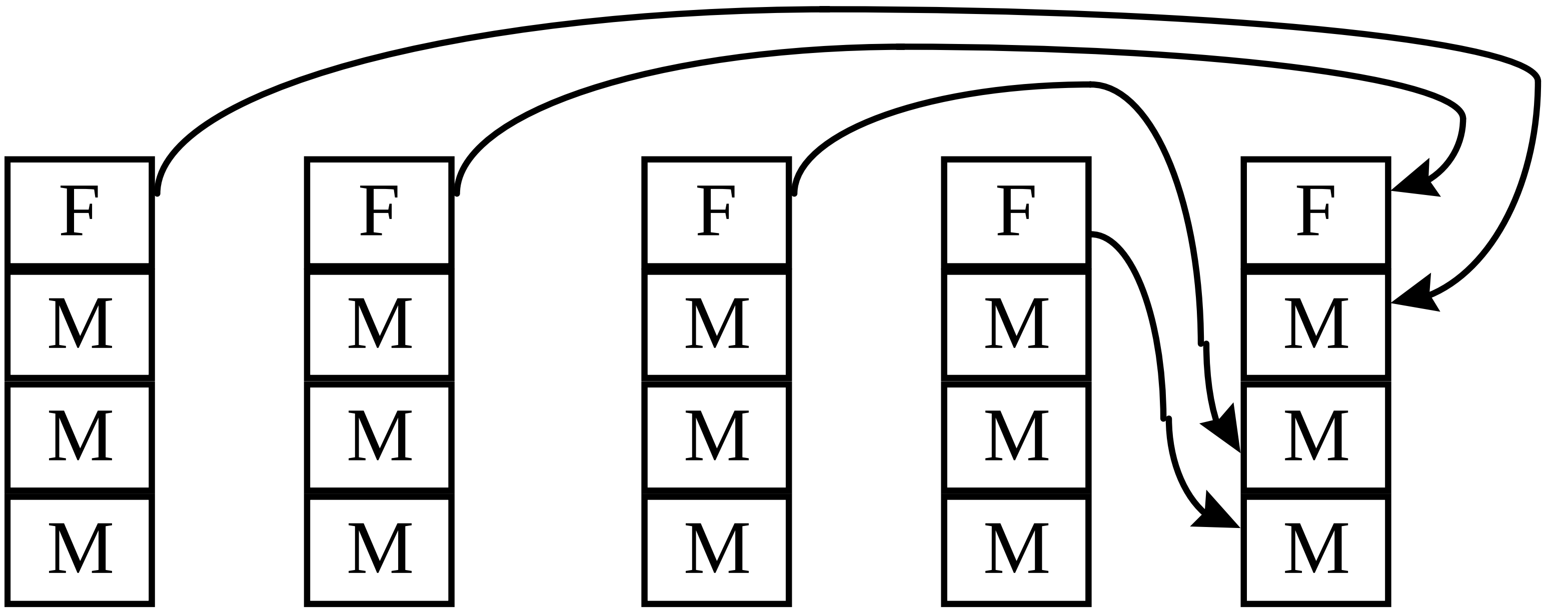
    $c(n) = n + 3/4n + (3/4)^2 n + ... = O(n)$ → optimální

Optimal list ranking

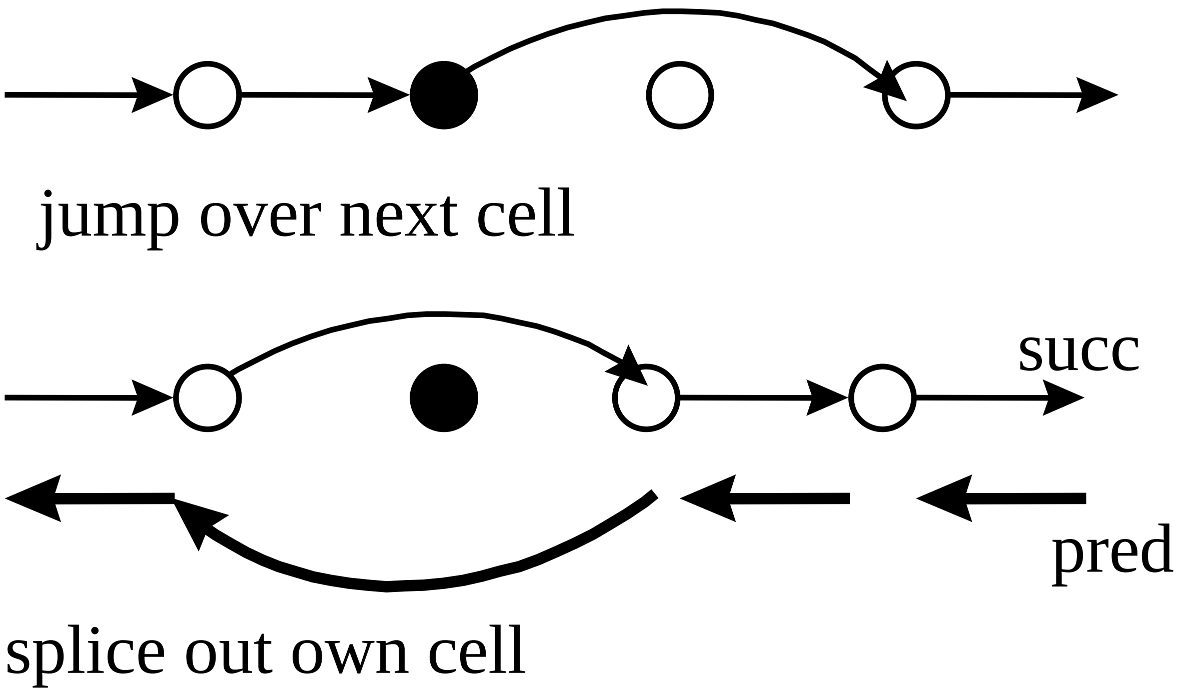
- Požadavek: pevný počet stále pracujících procesorů
- Simulace algoritmu Random Mate pomocí n/log n procesorů, každý procesor vykonává práci log n procesorů
- Každý procesor má zásobník prvků
- Každý procesor se pokouší přeskočit (jump over) následníka prvku na vrcholu zásobníku
- Pokud je vrchol zásobníku přeskočen, procesor se zabývá dalším prvkem na zásobníku



- <u>Problém</u>: algoritmus může být nevyvážený



- Řešení
  - Operace jump over se změní na splice out (vypletení)



- Prvky jsou v poli zásobníků q[i, j]
- Máme „makro“ označující vrcholy zásobníků top[i]
- Prvek je vypleten pokud je male na kterého ukazuje female

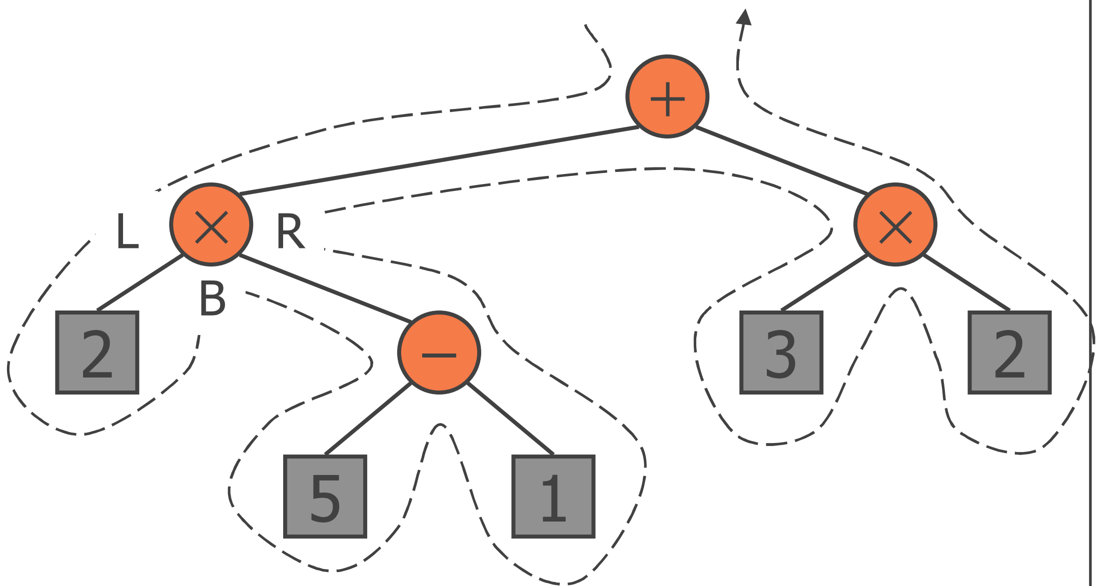
```
Algorithm
- optimal randomized list ranking

Procedure SpliceOut(i)
        rank[pred[i]] = rank[pred[i]] + rank[i]
        succ[pred[i]] = succ[i]
        if succ[i] <> tail then pred[succ[i]] = pred[i]
end proc
```

```
procedure OptimalRandomMate
  for i = 1 to P do in parallel                    { initialize }
    for j = 1 to log n do
         rank[i] = 1; sex[i] = F
    end for
  endofor
  sex[tail] = M
  t = 1                                            { global variable – time }
  for i = 1 to P do in parallel                    { pointer jumping - splicing out }
    while succ[head] <> tail do
         sex[top[i]]= Random(M, F)
         if sex[pred[top[i]]] = F and sex[top[i]]=M then
           SpliceOut(top[i])
           Decrease(top[i])
           splicetime[top[i]] = t
         endif
         t = t + 1
    endwhile
    while t >= 0 do
         if splicetime[top[i]] = t and succ[top[i]]<> tail do
           rank[top[i]] = rank[top[i]] + rank[succ[top[i]]]
           Increase(top[i])
         endif
         t = t - 1
    endwhile
  endfor
end proc
```

Tree

Euler tour traversal

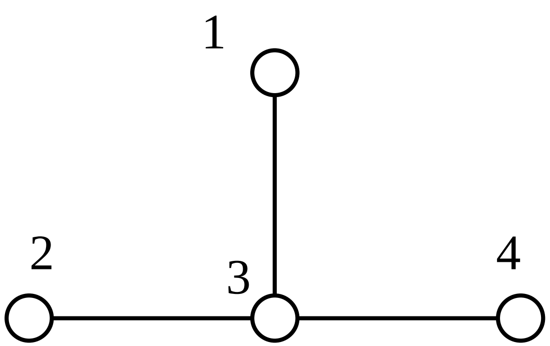
- Obecný průchod binárním stromem
- Její speciální případy jsou průchody preorder, postorder a inorder



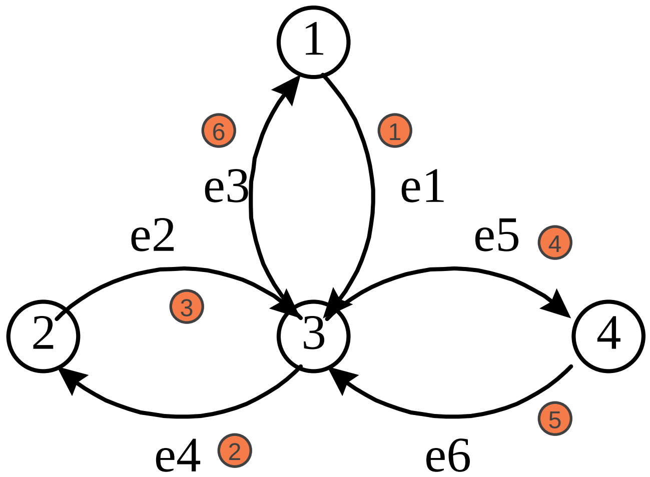
Euler tour

- T = (V, E)   - daný strom
- T' = (V, E')   - orientovaný graf získaný z T tak, že každá hrana (u, v) je nahrazena dvěma orientovanými hranami <u, v> a <v, u>
- T' je Elerovský graf – obsahuje orientovanou kružnici, která prochází každou hranou právě jednou

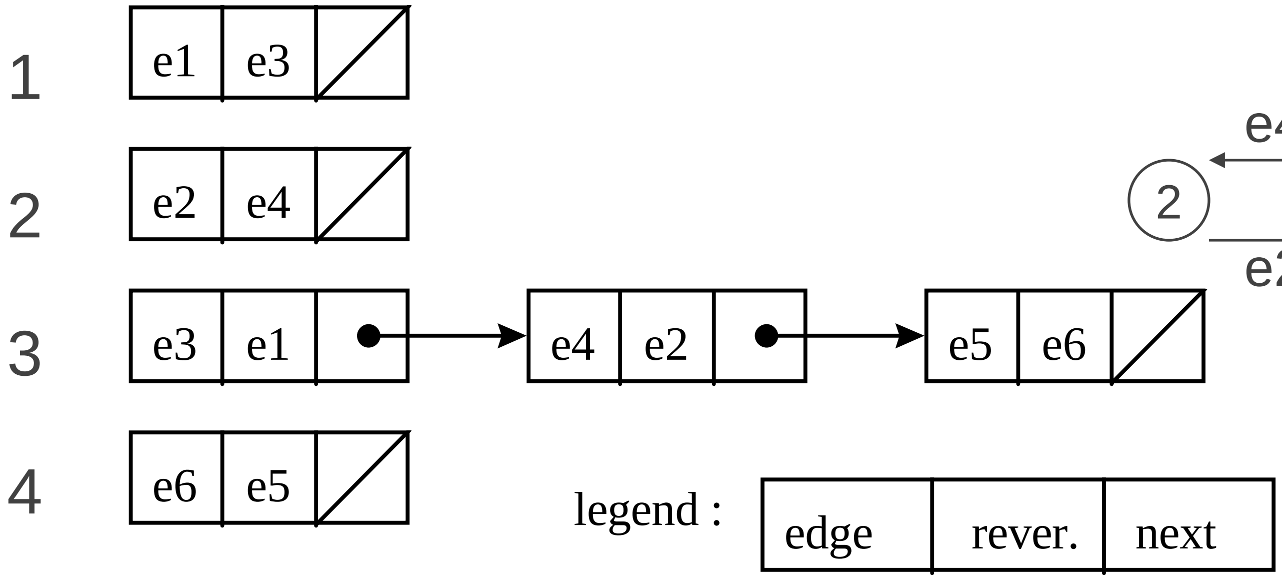
A Tree

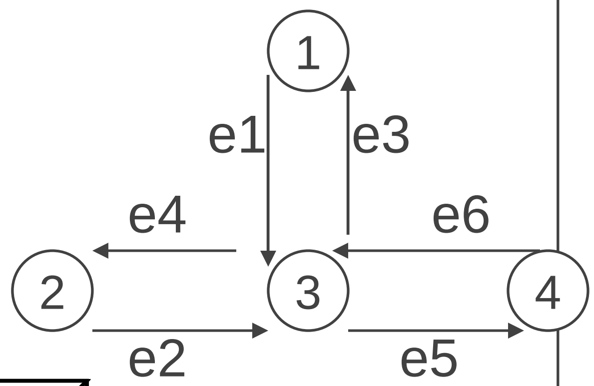


Eulerian graph T



- Eulerova kružnice je reprezentována funkcí následníka Etour která každé hraně e∈ E’ přiřazuje hranu Etour(e) ∈ E’ která následuje hranu e
- Reprezentace – seznam sousednosti (adjacency list)
- Vrchol





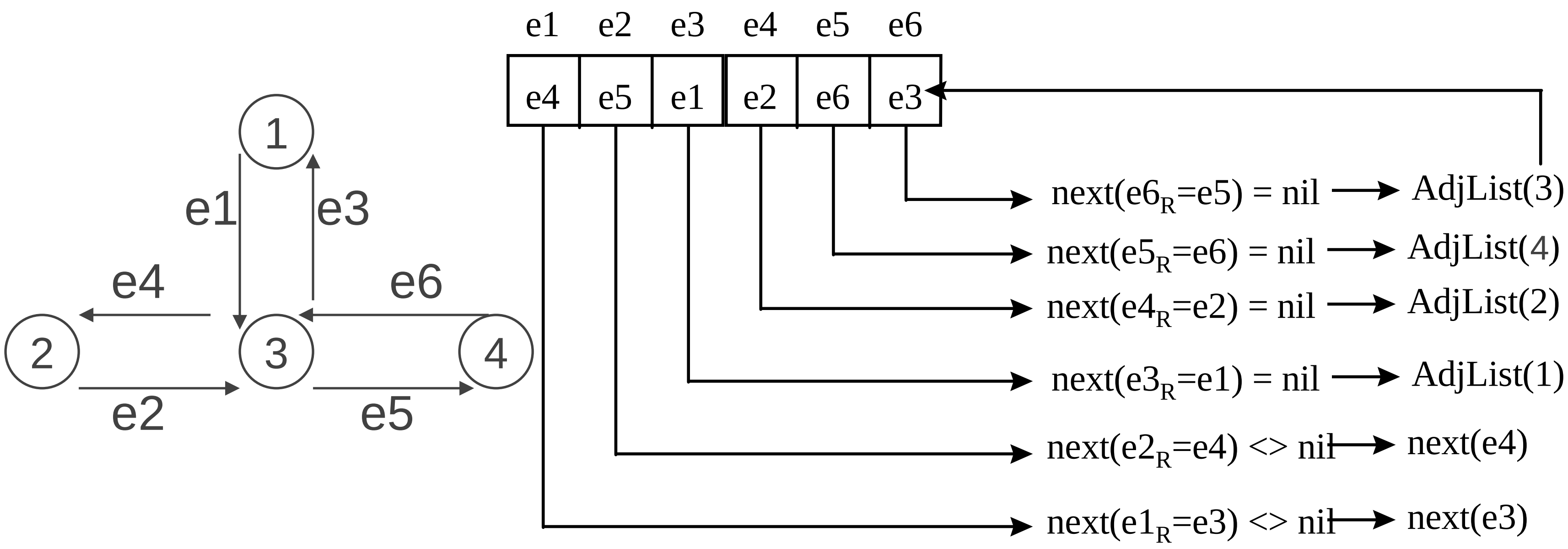
Pokud e=(u, v) je hrana e, pak R = (v, u) je reverzní hrana

Vytvoření Eulerovy cesty

- .

```
Algorithm
- constructing Euler tour
Input: adjacency list of T
Output: Array Etour with 2n-2 entries
        Etour(e) is a edge following e

for i = 1 to 2n-2 do in parallel      { e_i = (u, v) }
  if next(e_R) <> nil   then Etour(e) = next(e_R)
    else Etour(e) = AdjList(v)        { first item of adj. list of   vertex v }
  endif
endfor
```

Příklad



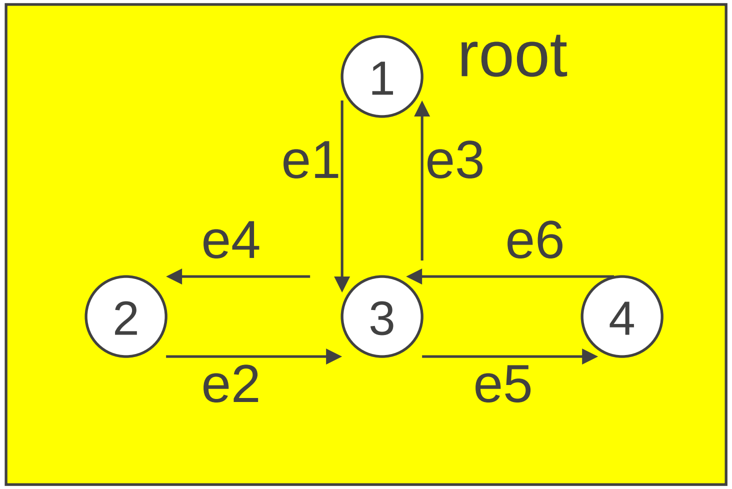
- Algoritmus nepoužívá kořen stromu
- Pokud použijeme kořen, pak cesta prochází stromem průchodem „depth-first search“

Zavedení kořene

- Pro operace nad stromem musíme být schopni pro každý vrchol v určit jeho rodiče parent(v)
- Proto musíme definovat kořen stromu, kterým bude vrchol r
- Zkonstruujeme Eulerovu cestu a přiřadíme pro hranu <u>e</u> vedoucí do kořene r
  - Etour(e) = e
- Jinými slovy přeřízneme Eulerovu cestu ve vrcholu r

Úkol: Spočíst pozici každé hrany

- <u>Algoritmus</u> – Eulerova cesta s pozocemi
  - – Vstup: binární strom (adjacency list)
  - – Výstup: pole Etour, pole Posn, kde Posn(e) je pozice hrany <u>e</u> v Eulerově cestě
    - » 1. Spočteme se pole Etour *O(c)*
    - » 2. Přiřadíme Etour(e) = e pro hranu <u>e</u> vedoucí do kořene *O(1)*
    - » 3. Rank = ListRanking(Etour) *O(log n)*
    - » 4. Spočteme paralelně posn(e) = 2n-2-Rank(e) *O(1)*



	e1	e2	e3	e4	e5	e6
E-tour	e4	e5	e3	e2	e6	e3
Rank	5	3	0	4	2	1
Posn = 6 - RANK	1	3	6	2	4	5

Úkol: Nalezení rodičů

Vstup: Eulerova cesta a speciální vrchol *root*

Výstup: Pro každý vrchol $v \neq root$, je jeho rodičem vrchol *parent*(v)

- Hrana - dopředná posn(e) < posn(e_R)
- - zpětná posn(e) > posn(e_R)
- Pokud (u, v) je dopředná hrana, pak u je rodičem v ve stromu

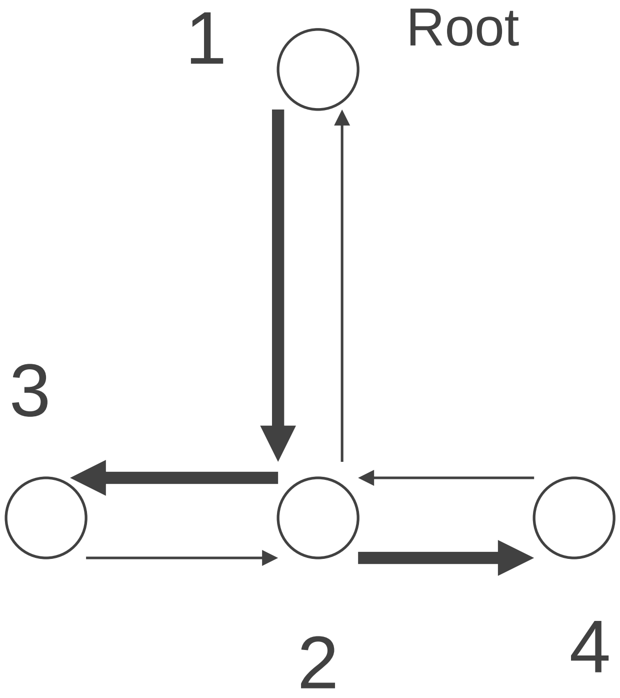
```
Algorithm
for each edge e = (u, v) do in parallel
  if posn(e) < posn(eR) then
    parent(v) := u;
  endif
  parent(root) := nil;
endfor
```

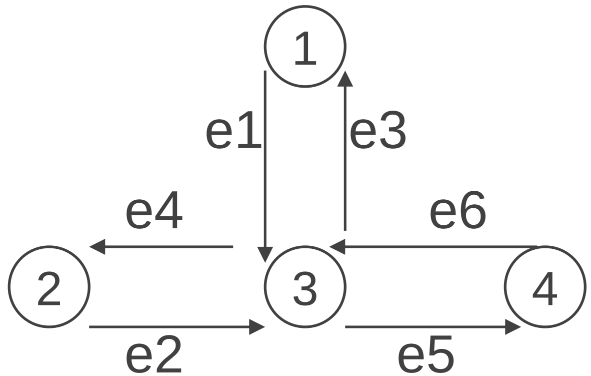
Obecný výpočet ve stromu

- Pro mnoho algoritmů nad stromem T je postup:
  - Vytvoříme Eulerovu cestu
  - Vytvoříme pole hodnot
  - Spočteme nad ním sumu suffixů
  - Provedeme korekci
- Můžeme tak např. spočíst:
  - Pořadí <u>postorder</u> vrcholu
  - Pořadí <u>preorder</u> vrcholu
  - Pořadí <u>inorder</u> vrcholu
  - <u>Počet následníků</u> vrcholu
  - <u>Úroveň</u> vrcholu

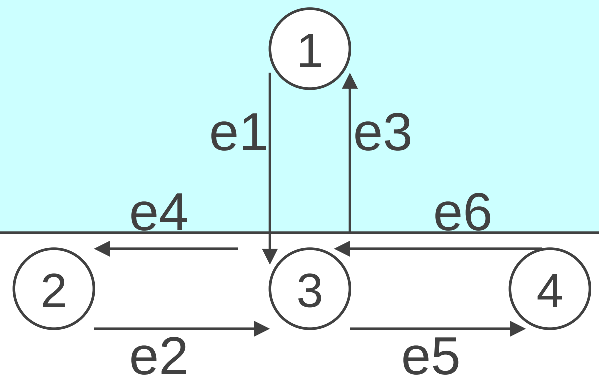
Úloha: Přiřazení pořadí preorder vrcholům

– Pořadí preorder vrcholu ve stromě je 1+počet dopředných hran, kterými jsme prošli po cestě k vrcholu
– (Preorder – navštiv nejdřív otce, pak oba syny)





```
Algorithm
1) for each e do in parallel
       if e is forward edge   then weight = 1
              else weight = 0
       endif
2) weight = SuffixSums(Etour, Weight)
3) for each e do in parallel
       if e=(u, v) is forward edge then
         preorder(v) = n - weight(e) + 1
       endif
   preorder(root) = 1
```



- Příklad

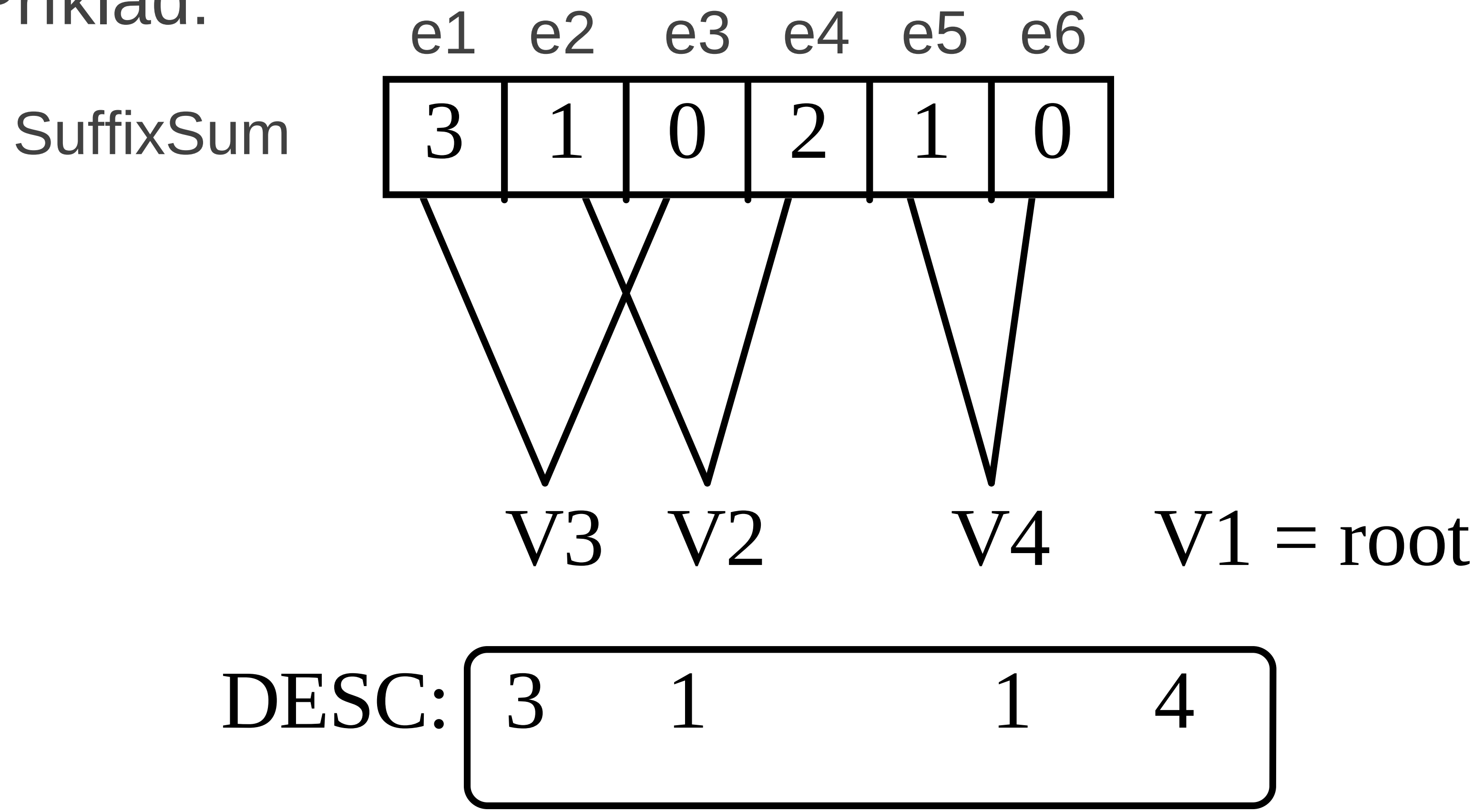
	e1	e2	e3	e4	e5	e6
Weight	1	0	0	1	1	0
pořadí v EC	1	3	6	2	4	5
Suffix Sums	3	1	0	2	1	0

Úkol: Počet následníků vrcholu v

- Počet dopředných hran v podstromu, který má kořen ve vrcholu v + 1 (aby byl započítán i vrchol)
- Počet dopředných hran v segmentu Eulerovy cesty, počínajícím i končícím ve v

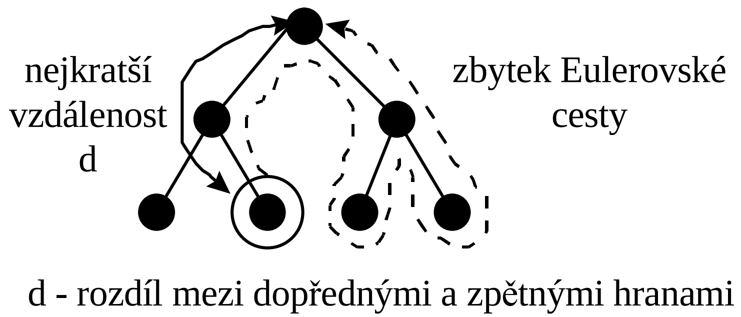
```
Algorithm - number of descendants
1) for each e do in parallel
       if e is forward edge  then weight = 1
               else weight = 0
       endif
2) weight = SuffixSums(Etour, Weight)
3) for each e do in parallel
       if e=(u, v) is forward edge then
         desc(v) = weight(u, v) - weight(v, u)
       endif
   desc(root) = n
```

- Příklad:



Úkol: Výpočet úrovně vrcholu

- Počet hran na cestě (nikoli Eulerově) z vrcholu v do kořene
- Rozdíl počtu zpětných a dopředných hran na zbytku Eulerovy cesty od vrcholu v do konce




```
Algorithm: - level of vertex
1) for each e do in parallel
        if e  is forward edge  then weight(e) = 1
                               else weight(e) = -1
        end if
   end for
2) weight = SuffixSums(Etour, weight)
3) for each e do in parallel
        if e = (u, v) is forward edge then level(v) = weight(e) + 1
        endif
   endfor
   level(root) = 0
```

Analýza předchozích algoritmů

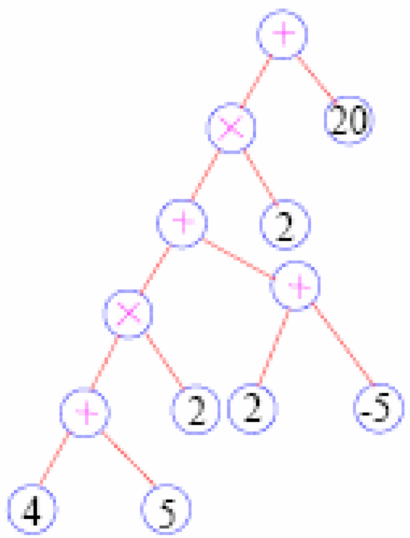
- 0) Spočtení Etour O(c)
- 1) Inicializace weight O(c)
- 2) Výpočet SuffixSums O(log n)
- 3) Korekce výsledku O(c)

- t(n) = O(log n)
- c(n) – závisí na implementaci SuffixSums

Tree Contraction

Tree Contraction

- Některé operace nad stromem se nedají provést efektivně pouze pomocí Eulerovy cesty
- Například paralelní vyhodnocení aritmetického výrazu uloženého v binárním stromu



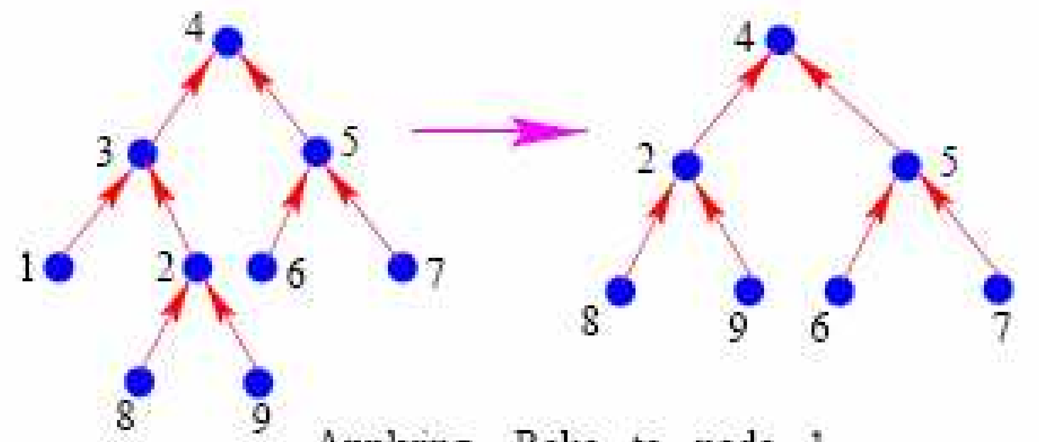
((4 + 5) * 2 + (-5 + 2)) * 2 + 20

Tree Contraction

- Každý list obsahuje operand a každý nelist obsahuje operátor, jako je například **+**,$*$
- Cílem je spočíst hodnotu výrazu v kořeni
- Technika **tree contraction** je systematický způsob jak zmenšovat strom až do velikosti jednoho vrcholu
- Opakovaně aplikujeme
  - Spojení listu s jeho rodičem
  - nebo
  - Spojení vrcholu stupně 2 s jeho rodičem

The RAKE operation

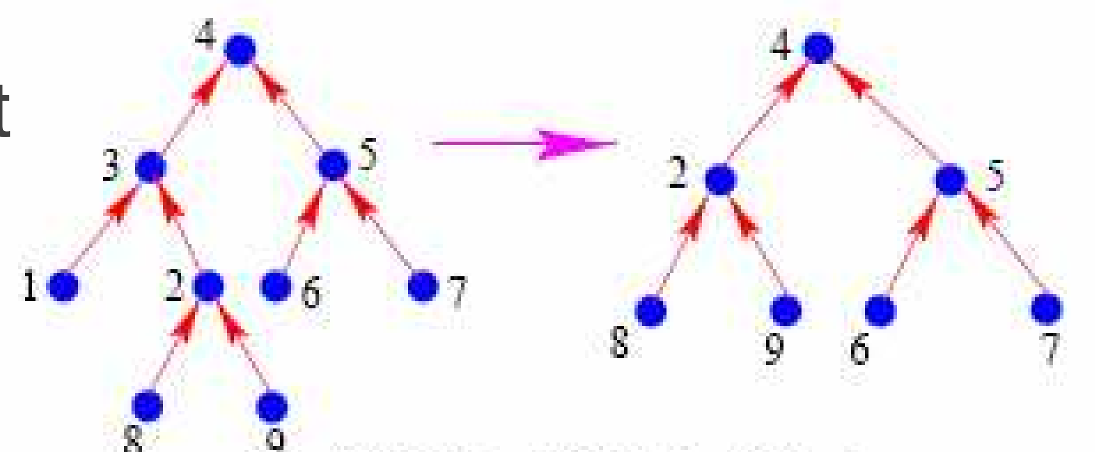
- $T = (V, E)$ je binární strom a pro každý vrchol v je $p(v)$ jeho rodič
- $sib(v)$ je synem $p(v)$
  - Bereme v úvahu pouze binární stromy
- Operace RAKE pro listový vrchol u takový, že $p(u) \neq r$ provede následující:
  - Odstraní u a $p(u)$ ze stromu T
  - Připojí podstrom $sib(u)$ to $p(p(u))$



Applying Rake to node 1

The RAKE operation

- V algoritmu tree contraction aplikujeme operace RAKE opakovaně, abychom pokaždé zmenšili velikost stromu
- Pro maximální rychlost ji chceme aplikovat na co nejvíce listů paralelně
  - Ale nelze aplikovat operaci RAKE na vrcholy, jejichž rodiče ve stromu sousedí
  - Například nelze provést paralelně operaci RAKE na vrcholy 1 a 8
- Musíme aplikovat operaci RAKE na nesousedící uzly



Applying Rake to node 1

The RAKE operation

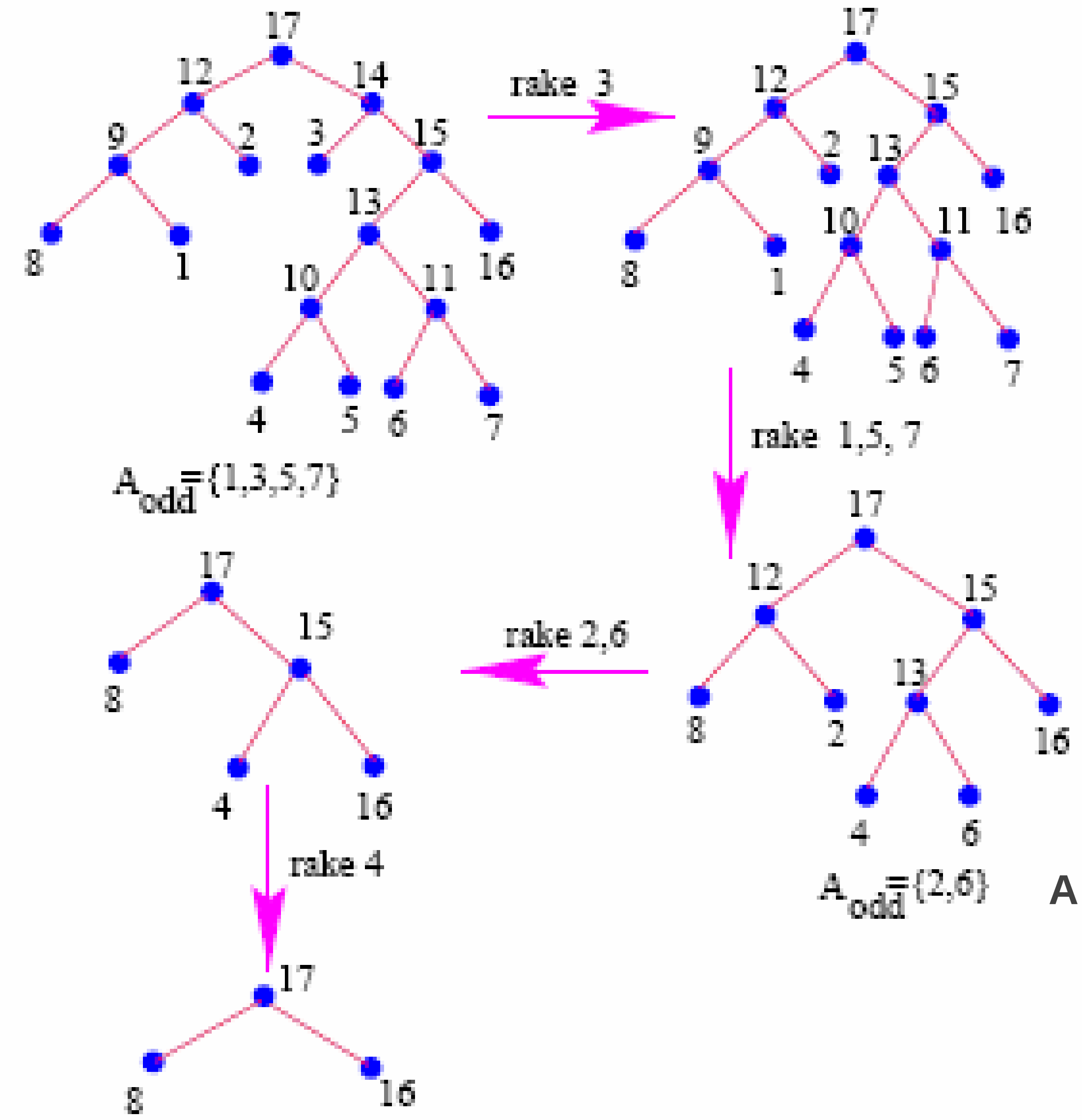
- Označíme listy jejich pořadím zleva doprava
- V Eulerově cestě se listy vyskytují také v pořadí zleva doprava
- Každé hraně $(v, p(v))$, kde v je listem, přiřadíme váhu 1
- Vyřadíme nejlevější a nejpravější list. Tyto listy budou dva synové kořene až se podaří strom zmenšit na strom se třemi vrcholy
- Nad výsledným seznamem provedeme sumu suffixů a získáme listy, očíslované zleva doprava

The RAKE operation

- Pak uložíme všech n listů do pole A
- A_{odd} obsahuje prvky pole A s lichými indexy
- A_{even} obsahuje prvky pole A se sudými indexy
- Pole A_{odd} a A_{even} vytvoříme v čase $O(1)$ za cenu $O(n)$

```
for i:=1 to log(n+1) do
   1. Apply the rake operation in parallel to all the elements
      of A_odd that are left children
   2. Apply the rake operation in parallel to the rest
      of the elements in A_odd
   3. A := A_even
endfor
```

Tree contraction



A = (2,4,6)

Vlastnosti algoritmu

- Když je operace RAKE aplikována paralelně na několik listů, rodiče těchto listů nejsou sousedi
  - Provádíme RAKE na každého druhého levého syna, pak na každého druhého pravého syna atd.
- Počet listů se po každé iteraci cyklu zmenší na polovinu
  - Liché listy jsou „shrabány“
- Proto je strom zcela zredukován v čase O(log n)

List coloring, Ruling set

Obarvení seznamu - List coloring

- k-obarvení seznamu je zobrazení
- $C: V \rightarrow \{0, 1, ..., k-1\}$
- pokud pro každý vrchol platí $C[x] \neq C[succ(x)]$

Úkol (zjednodušený): 2 log n coloring

- 2 log n - počet barev, kde n je počet prvků v seznamu
- Použijeme ID procesoru – rozbití symetrie

```
Algorithm
Input: linked list, array Index with indices of processors
Output: 2.log n  - coloring ⇒ array Color with colors of vertices

for every vertex v do in parallel
       k = least significant position in which
           Index[v] and Index[succ(v)] disagree
       Color[v] = 2.k + k-th bit of Index[v]
end for
```

Index	Index (binary)	k	Color
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	…	…
	3 2 1 0		

Ruling set (množina oddělovačů)

- Máme seznam s vrcholy

  V = {v1, v2, ..., vn}

- Pak podmnožina S množiny vrcholů V je <u>k-ruling set</u> pokud

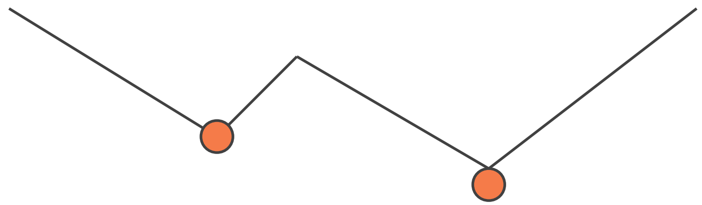
  1. Žádné dva vrcholy v S spolu nesousedí
  2. Vzdálenost mezi (nevybraným) vrcholem k následujícímu vybranému je nejvíce <u>k</u>

- $\underline{v_1}$ v_2 v_3 $\underline{v_4}$ v_5 $\underline{v_6}$ … $\Rightarrow$ 2-ruling set

Úkol: (zjednodušený)

- 2.k-ruling set from k-coloring

```
Algorithm
Input: k-coloring-array Color
Output: 2.k-ruling set - array Ruler

init Ruler for Falses
for each vertex v do in parallel
        if pred(v) ≠ head
           and Color[pred(v)]>Color[v]
           and Color[succ(v)]>Color[v]
                then Ruler[v] = True
endfor
Ruler[head] = True
```



Úkol: 2-ruling set

```
Algorithm
Input: linked list, array Index with indices of processors
Output: array Ruler - 2-ruling set

(1) Compute 2.log n coloring
(2) for Col = 0 to 2.log n do
        if Color[i] = Col
             then if not (Ruler[i] or Ruler[pred(i)]
                      or Ruler[succ(i)])
                       then Ruler[i] = True
             end if
        end if
```

- Vlastnosti
- 1. Nejsou vybrány dva sousední vrcholy
- 2. Každý nevybraný vrchol má vybraného souseda
- ad 1) Dva sousedé nemohou být vybráni ve stejné iteraci – sousedé nemohou mít stejnou barvu
- ad 2) Jediný důvod proč není vrchol vybrán je ten, že má vybraného souseda

KONEC