Dynamické datové struktury II.

Stromy. Binární vyhledávací strom. Základní operace nad stromy. DFS. BFS.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Strom

Patří mezi rekurzivní datové struktury.

Takové uspořádání dat, kdy každý prvek má nejvýše jednoho předchůdce a může mít více než jednoho následníka.

Jedna z nejčastěji používaných dynamických datových struktur.

Mnoho typů stromů: binární stromy, ternární stromy, kořenové stromy, volné stromy...

Neorientovaný graf

$$G = \langle H, U, \rho \rangle, \qquad H = \{h_i\}_{i=1}^{n_h}, U = \{u_i\}_{i=1}^{n_u}, \rho(h) = [u_1, u_2].$$

G stromem T, $T \subset G$, pak $\exists$ právě jedna cesta C

$$C(u, v) = \{h_1, ..., h_k\}, \qquad \rho(h_1) = [u, \cdot], \rho(h_1) = [\cdot, v], \qquad u, v \in U.$$

Pozn.: každá hrana h se v cestě C vyskytuje pouze 1x.

2. Typy uzlů

3 typy uzlů:

- *Listy:*
  Uzly bez následníka, u_l.
  Není k nim připojen žádný podstrom.
- *Kořen:*
  Uzel bez předchůdce = kořen.
  Existuje právě 1, u_0.
- *Vnitřní uzly:*
  Uzly, které nejsou listem ani kořenem.

Kořenové stromy:

Každý uzel je současně kořenem stromu a zároveň listem stromu vyšší úrovně.

Volné stromy:

Nedisponují kořenem, obecnější datové struktury (seznam).

3. Stromy, základní vztahy

Stupeň uzlu m:

Maximální počet potomků uzlu ($m = 2$, binární; $m = 3$, ternární).

Vztah počtu uzlů a hran

$$n_h = n_u - 1.$$

Výška x uzlu v (úroveň):

Délka cesty od kořene u_0 k uzlu u (počet uzlů)

$$x(u) = \|C(u_0, u)\|.$$

Výška stromu h:

Představuje max. délku cesty z kořene u_0 k některému listu u_l

$$h = \max_{\forall u_i T}(x(u_i)) = \max_{\forall u_l \in U} \|C(u_0, u_l)\|.$$

Ovlivňuje rychlost operací nad stromem (odpovídá hloubce rekurze).

Snaha vytvářet stromy s $\underline{h}$, cesty do všech uzlů stejně dlouhé: $\underline{x}(u_l) = \overline{x}(u_l)$.

4. Stromy, základní vztahy

Počtem uzlů n_u v úrovni x

$$n_u(x) \leq m^{x(u)-1}.$$

Vztah mezi počtem uzlů n_u a výškou stromu h:

$$\begin{aligned} n_u \leq m^0 + m^1 + ... + m^h = & \sum_{i=0}^{h} m^h, \\ = & a_1 \frac{q^k - 1}{q - 1}, \\ = & \frac{m^{h+1} - 1}{m - 1}. \end{aligned}$$

Binární strom: $n_u \leq 2^{h+1} - 1$.

Vztah mezi výškou h a počtem uzlů n_u:

$$\begin{aligned} h & \in & \left\langle \frac{\ln(n_u)}{\ln(m)}, n_u - 1 \right\rangle, \\ & = & \langle \log_m(n_u), n_u - 1 \rangle. \end{aligned}$$

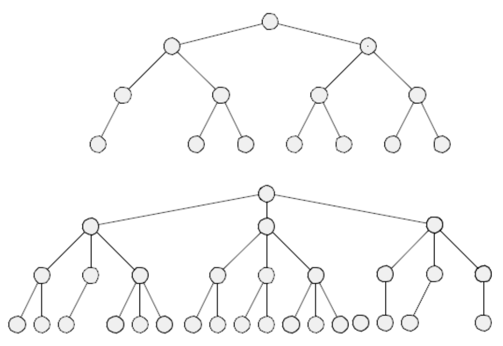
Pozor na stromy $h \gg \sup(n_u - 1)$, neefektivní!

5. Ukázky stromů pro různá *n*

Unární strom: $n = 1$ $\Rightarrow$seznam, fronta zásobník.

Binární strom: $n = 2$.

Ternární strom: $n = 3$.



6. Vyvážené a degenerované stromy

Vyvážený strom:

Každý uzel má podobný počet potomků nebo výšku.

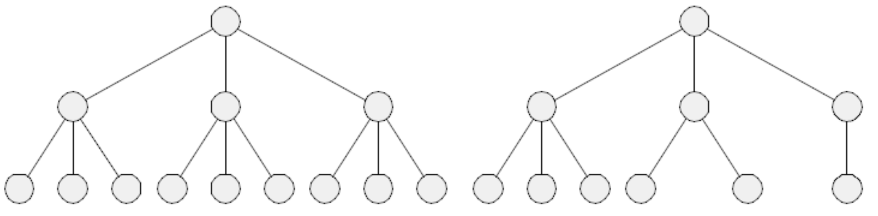
Pro konstrukci vyvážených stromů existuje řada různých algoritmů.

Využití při práci s daty, teorii grafů, indexování.

Nevyvážený (degenerovaný) strom:

Nerovnoměrné rozložení uzlů, větší výška.

Nebezpečí používání: značná hloubka rekurzivního volání, při provádění může dojít paměť.



7. Binární strom

Nejčastěji používané typy stromů v informatice, $m = 2$.

Každý uzel (rodič) má dva podstromy (potomky): levý podstrom a pravý podstrom.

Využití pro rychlé vyhledávání, indexace, komprese, ...

Vyvážené binární stromy:

Rovnoměrné rozložení uzlů, neobsahuje příliš dlouhé či krátké větve.

Všechny listy se nacházejí na přibližně stejné úrovni.

Hloubka vyváženého binárního stromu:

$$h = \log_2(n_u).$$

Důsledek:

Operace nad tímto stromem mají příznivou asymptotickou složitost.

Procházení, vyhledávání se stávají efektivní, méně efektivní přidávání či mazání uzlů.

8. Vyvážené binární stromy

Dokonale vyvážený binární strom:

Počet vrcholu v levém a pravém podstromu se liší maximálně o 1.

Obtížné na implementaci při přidávání a odebírání vrcholu.

Nutné jeho převažování (LL, LP, PL, PP rotace).

Absolutně vyvážený binární strom:

Výška levého podstromu se rovná výšce pravého podstromu nebo se liší právě o 1 (Adelson, Velskii, Landis)⇒AVL stromy.

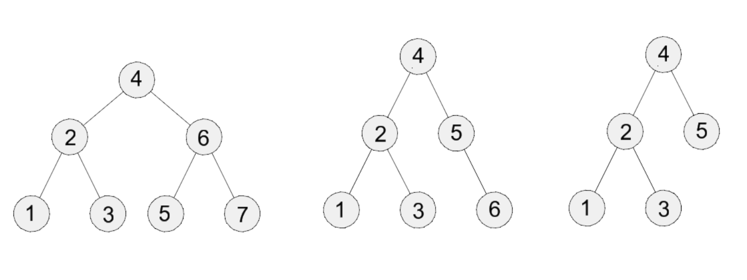
Snadnější zajištění vyváženosti než v předchozí variantě, používány v praxi.

$$h_{AVL} \leq 2 \cdot \log(n_u).$$

Převažování stromu $O(\log(n_u)$.

9. Ukázky binárních stromů

Dokonale vyvážený strom (1,2), absolutně vyvážený strom (3).

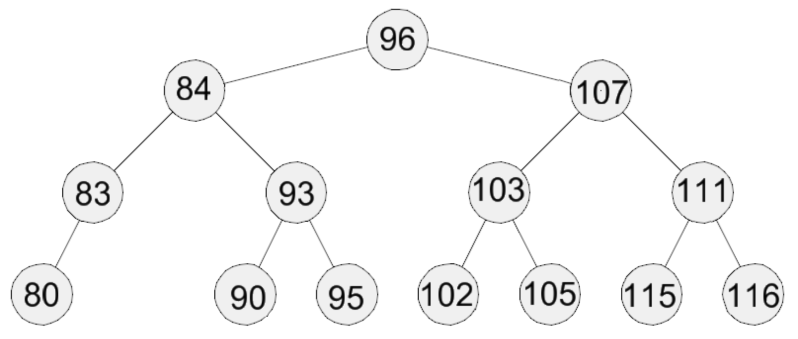


10. Binární vyhledávací strom (BST)

Pro každý uzel $u \in U$, a jeho potomky $u_{left}, u_{right} \in U$ platí:

$$u_{left} < u \wedge u_{right} > u, \qquad \forall u \in U.$$

Všechny prvky v levém podstromu jsou menší než u,
Všechny prvky v pravém podstromu větší než u.



11. Operace nad BST stromy

Základní operace:

- Přidání uzlu: $O(\log(N))$.
- Odstranění uzlu: $O(\log(N))$.
- Nalezení uzlu: $O(\log(N))$.
- Průchod stromem: $O(N)$.
- Smazání stromu: $O(N)$.

Pokud stromy nejsou vyvážené, složitost $O(\log(N)) \rightarrow O(N)$!!!

Operace probíhají ve směru shora-dolů:

- od kořene k listům (přidání uzlu),
- od listům ke kořeni (mazání).

12. Procházení BST stromu

3 základní metody procházení stromu, složitost $O(N)$:

1. Přímé zpracování: PREORDER.
2. Vnitřní zpracování: INORDER.
3. Zpětné zpracování: POSTORDER.

Přímé zpracování:

Nejprve zpracován kořen, poté L podstrom, nakonec P podstrom.

Vnitřní zpracování:

Nejprve zpracován L podstrom, poté kořen, nakonec P podstrom.

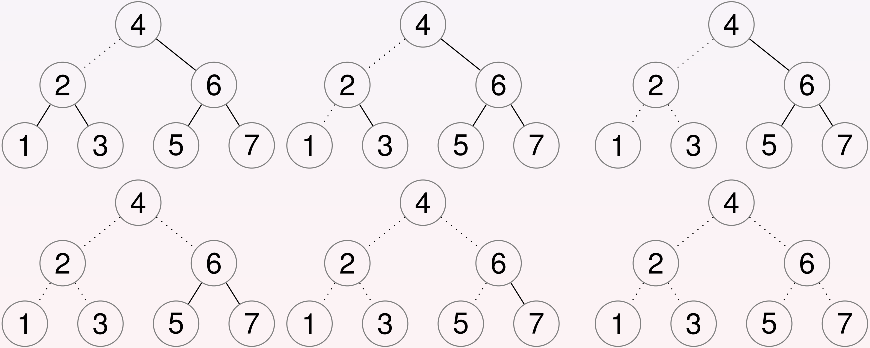
Zpětné zpracování:

Nejprve zpracován L podstrom, poté P podstrom, nakonec kořen.

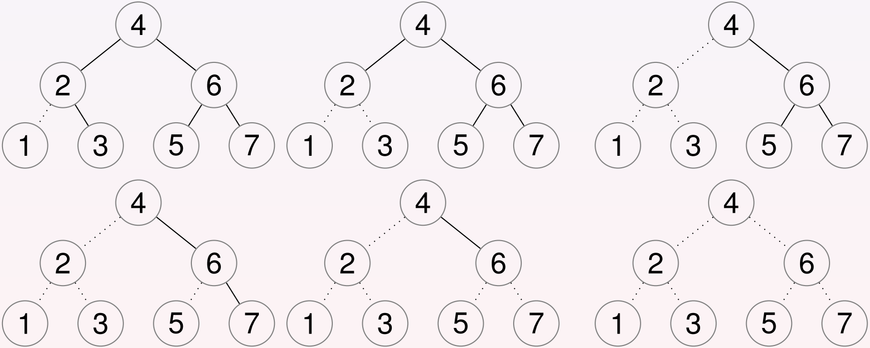
Princip zpracování stejný, liší se však pořadím prováděných operací.

Lze implementovat rekurzivně i nerekurzivně.

13. Procházení stromu PREORDER



14. Procházení stromu INORDER



15. Implementace třídy Uzel v BST stromu

Uzel nejprve vytvořen.

Na správné místo zařazen až následně.

```
class TNode:
    def __init__(self, data):
        self.left = None     #Left subtree
        self.right = None    #Right subtree
        self.data = data     #Data member
```

16. Vytvoření prázdného stromu, ukázka

Složitost $O(1)$.

Postup vytvoření prázdného stromu:

- Vytvoření odkazu na kořen.
- Inicializace odkazu na None: `root`=None.

```
class BST:
    def __init__(self):
        self.root = None
```

17. Procházení stromu PREORDER: rekurze

Záměna pořadí tisku: varianty INORDER, PREORDER, POSTORDER.

Rekurzivní varianta:

```
def preorder(self, u):
    if u==None:
        return;                      #Empty tree
    else:
        print(u.data)                #1. root
        self.preorder(u.left)        #2. left subtree
        self.preorder(u.right)       #3. right subtree
```

1. kořen, 2. levý podstrom, 3. pravý podstrom

18. Procházení stromu PREORDER: zásobník

Nekurzivní varianta s využitím zásobníku.

```
def preorder2(self, u):
    S = []                      #Stack as list
    S.append(u)
    while S:
        v = S.pop()
        print(v.data)           #1. root
        if v.left != None:
            S.append(v.left)    #2. left subtree
        if v.right != None:
            S.append(v.right)   #3. right subtree
```

19. Procházení stromu INORDER: rekurze

```
def inorder(self, u):
    if u==None:
        return;                  #Empty tree
    else:
        self.inorder(u.left)     #1. left subtree
        print(u.data)            #2. root
        self.inorder(u.right)    #3. right subtree
```

1. levý podstrom, 2. kořen, 3. pravý podstrom

20. Procházení stromu INORDER: zásobník

```
def inorder2(self, u):
    S = []                      #Stack as list
    S.append(u)
    while S:
        v = S.pop()
        if v.left != None:
            S.append(v.left)    #1. left subtree
        print(v.data)           #2. root
        if v.right != None:
            S.append(v.right)   #3. right subtree
```

21. Procházení stromu POSTORDER: rekurze

```
def postorder(self, u):
    if u==None:
        return;                          #Empty tree
    else:
        self.postorder(u.left)           #1. left subtree
        self.postorder(u.right)          #2. right subtree
        print(u.data)                    #3. root
```

1. levý podstrom, 2. pravý podstrom, 3. kořen

22. Procházení stromu POSTORDER: zásobník

```
def postorder2(self, u):
    S = []                          #Stack as list
    S.append(u)
    while S:
        v = S.pop()
        if v.left != None:
            S.append(v.left)   #1. left subtree
        if v.right != None:
            S.append(v.right)  #2. right subtree
        print(v.data)               #3. root
```

23. Přidání nového uzlu

Složitost $O(1.4\log(N))$.

Rekurzivní prohledávání stromu ve směru od kořene k listům.

Nalezení vhodného uzlu a zařazení položky.

Využijeme přístup PREORDER:

Nejprve zpracován kořen, poté levý podstrom, následně pravý podstrom)

Postup přidání nového uzlu:

1. Otestujeme, zda je přidávaný uzel `v` kořenem
2. Pokud ano (`v=None`), vytvořen nový uzel.
3. Jinak:
   - Je-li `data < v.data` (tj. kořen podstromu), přidáme `v` do levého podstromu
   - V opačném případě přidáme `v` do pravého podstromu
4. Vrátíme odkaz na nově vytvořený kořen.

24. Přidání nového uzlu: rekurze

```
def add (self, u, data):
    if u==None:                              #Empty tree
        u = TNode(data)
    else:
        if data < u.data:
            u.left = add(u.left, data)       #Add to the left subtree
        elif data > u.data:
            u.right = add(u.right, data)     #Add to the right subtree
        else:                                #Identical value
            return None
    return u                                 #Return reference to new roo

def addNode(self, data):
    self.root = self.add(self.root, data) #Start from root
    return self.root
```

25. Přidání nového uzlu: nerekurzivní řešení

```
def add(self, data):
    if self.root == None:        # Empty tree
        self.root = TNode(data)
        return
    u = self.root
    p = u
    while u != None:             #Find node and its p(u).
        p = u                    #Predeccessor of u
        if data < u.data:
            u = u.left           #Search the left subtree
        elif data > v.data:
            u = u.right          #Search the right subtree
        else:
            return None          #Already in tree
    v = TNode(data)
    if data < p.data:
        p.left = v               #Add to the left subtree
    else:
        p.right = v              #Add to the right subtree
   return v
```

26. Nalezení uzlu ve stromu

Složitost $C_{aver} = 1.4 \log(N)$.

1. Pokud je strom prázdný, ukončíme jeho prohledávání.
2. Jinak testujeme shodu uzlu a hledané položky.
   V případě shody ukončíme prohledávání.
3. Pokud hledaná položka menší než hodnota v L podstromu, hledáme v L podstromu.
4. Jinak prohledáme P podstrom.

Poznámka k implementaci:

Využití rekurze.

Pokud je uzel nalezen, vracíme odkaz na něj.

V opačném případě vracíme `None`.

27. Nalezení uzlu v BST stromu, ukázka

```
def find(self, data, u):
    if u == None:                  #We are in leaf
        return None
    if data == u.data:             #Matching node
        return u
    if data < u.data:              #Traverse left subt
        return self.find(data, u.left)
    else:                          #Traverse right sub
        return self.find(data, u.right)

def findNode(self, data:)
    return self.find(data, self.root)
```

Na tomto principu založeno tzv. *binární vyhledávání*.

28. Smazání stromu

Složitost $O(N)$.

Mazání probíhá od kořene k uzlům.

Nejprve mažeme levý podstrom.

Poté mažeme pravý podstrom.

Jako poslední mažeme kořen.

Postup mazání stromu:

1. Pokud je strom prázdný, ukončíme mazání.
2. Smažeme levý podstrom připojený k aktuálnímu uzlu `u`.
3. Smažeme pravý podstrom připojený k aktuálnímu uzlu `u`.
4. Smažeme uzel `u`.

29. Smazání stromu, ukázka

Využití dvojnásobné rekurze.

```
def clear(self, u):
    if u != None:
        self.clear(u.left)
        self.clear(u.right)
        u = None
```

Aplikace POSTORDER procházení.

30. Smazání uzlu

Složitost $O(1.4\log(N))$.
Cílem je odstranění uzlu zadaného hodnotou ze stromu.
Implementačně nejtěžší operace nad binárním stromem.

Při mazání uzlu dochází ke třem situacím:

- Mazaný uzel je list.
- Mazaný uzel má jednoho následníka.
- Mazaný uzel má dva následníky.

Varianty (1) (2) jednoduché, varianta (3) obtížnější.

Postup mazání uzlu:

1. Test, zda se ve stromu vyskytuje uzel se zadaným klíčem.
2. Pokud se uzel ve stromu nevyskytuje, proces mazání je ukončen.
3. V opačném případě mohou nastat 3 situace (viz výše).
   Detekce, o jakou variantu se jedná.
4. Smazání uzlu.
5. Rekonstrukce sousedských vztahů (odkazy na L a P podstrom).

31. Upravené prohledávání stromu

Při hledání si zapamatujeme předchůdce p nalezeného uzlu u.

Uzlu p následně upravíme odkaz na levý či pravý uzel (následník mazaného).

```
def find2(self, data, u, p):
    if u == None:                           #We are in leaf
        return None
    if data == u.data:                      #Matching node
        return u
    p = u                                   #Store the predecessor
    if data < u.data:                       #Traverse left subtree
        return self.find2(data, u.left, p)
    else:                                   #Traverse right subtree
        return self.find2(data, u.right, p)

def findNode2(self, data:)
    p = None
    return self.find2(data, self.root, p)   #Start from root
```

31. Prohledávání stromu, nerekurzivní

Při hledání si zapamatujeme předchůdce `p` nalezeného uzlu `u`.

Uzlu `p` následně upravíme odkaz na levý či pravý uzel (následník mazaného).

```
def findNode(self, data):
    res = []
    if self.root == None:   #Empty tree
        return None
    u = self.root           #Intialize, p(u)
    p = None
    while u != None:
        if data < u.data:   #Traverse left sub-tree
            p = u
            u = u.left
        elif data > u.data: #Traverse right sub-tree
            p = u
            u = u.right
        else:               #Node has been found
            res.append(p)   #Add predecessor
            res.append(u)   #Add node
            return res
    return None
```

32. Smazání uzlu, ukázka

Metoda rozlišuje 3 situace:

List, 1 následník, 2 následníci

```
def delete(self, data):
    p = None
    u = self.find(data, self.root, p)
    #Delete leaf
    if u.left == None and u.right == None:
        self.delLeaf(p, u)
    #Delete 1 subtree
    if u.left == None or u.right == None:
        self.del1Subtree(p, u)
    #Delete 2 subtrees
    else:
        self.del2Subtrees(u)
```

33. Smazání listu

Metodě předáváme jako parametr předchůdce `p` uzlu `u`.

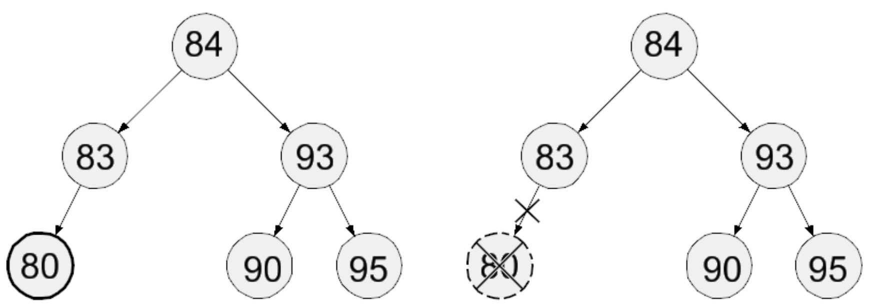
Při mazání listu `u` mohou nastat dva případy:

- *Mazaný list* `u` *je kořenem.*
  Předchůdce listu `u` má adresu `None`.
  Uzel `u` smažeme.
- *Mazaný list* `u` *není kořenem.*
  Předchůdce uzlu `u` nemá adresu `None`.
  Otestujeme, zda na `u` ukazuje levý nebo pravý podstrom.
  Nastavíme `p.levy=None` nebo `p.pravy=None`.

Nejjednodušší varianta, obejde se bez použití rekurze.

34. Mazání uzlu (list)

Odstranění uzlu a přesměrování odkazu z jeho předchůdce na null.



35. Smazání listu, ukázka

```
def delLeaf(p, u):
    if p == None:           #Tree contains only root
        self.root = None
    else:                   #Current tree
        if p.left == u:     #Left successor of p is u
            p.left = None
        else:               #Right successor of p is u
            p.right = None
```

36. Smazání uzlu s jedním potomkem

Metodě předáváme jako parametry uzel `u` a jeho předchůdce `p`.

Zavádíme pomocnou uzel `w`, následník `u`.

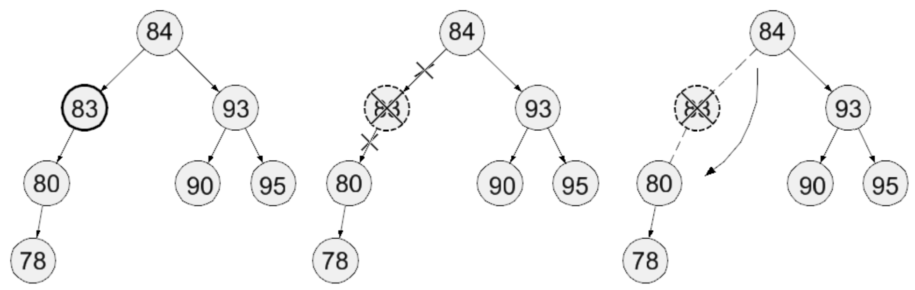
Postup bez použití rekurze.

Postup mazání uzlu:

1. Rozhodneme, zda následník `w` mazaného uzlu `u` je v L/P podstromu:
   Pokud `u.left!=None`, pak `w=u.left`.
   Jinak `w=u.right`.
2. Otestujeme, zda je mazaný prvek `u` kořenem:
   - Pokud ano: bude `w` novým kořenem.
   - Jinak zjistíme, zda je `u` levým či pravým následníkem `p`.
   - Nastavíme uzlu `p` jako následníka `w`.
3. Smazání uzlu `u`.

37. Mazání uzlu (jeden následník)

Přesměrování odkazu z předchůdce na následníka mazaného uzlu s následným smazáním uzlu.



38. Smazání uzlu s jedním potomkem, ukázka

```
def del1Subtree(p, u):
    w = None
    if u.left != None:          #Is w right successor?
        w = u.left
    else:                       #Is w left successor?
        w = u.right
    if p == None:               #Node u is the root
        self.root = w           #Node w is new root
    else:
        if p.left != u:         #Node u is left succ. of p
            p.left = w          #Link p and w
        else:                   #Node u is right succ of p
            p.right = w         #Link p a and w
```

39. Mazání uzlu (2 následníci)

Implementačně nejtěžší varianta.

Který uzel bude novým rodičovským uzlem po smazání rodiče?

Vede k přeuspořádání stromu.

Dvě možnosti:

Metoda PL: pracuje s největší *nižší* hodnotou, než je hodnota v mazaném uzlu (list).

Metoda LP: pracuje s nejmenší *vyšší* hodnotou, než je hodnota v mazaném uzlu (list).

Postup mazání uzlu tvořen 3 kroky:

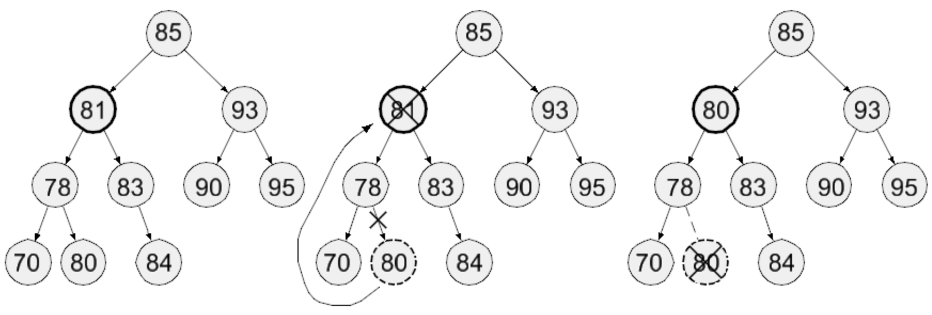
1. Nalezení nejpravějšího uzlu `n` v levém podstromu (je to list). Jde o uzel s největší hodnotou menší než hodnota rušeného uzlu (PL). [Nebo: Nalezení nejlevějšího uzlu `n` v pravém podstromu. (LP).]
2. Hodnotu `n` přesuneme do rušeného uzlu.
3. Zrušíme uzel `n`.

40. Mazání uzlu (2 následníci), varianta PL

Nalezení nejpravějšího prvku v levém podstromu (list).

Prohození pozice s mazaným prvkem.

Zrušení listu.

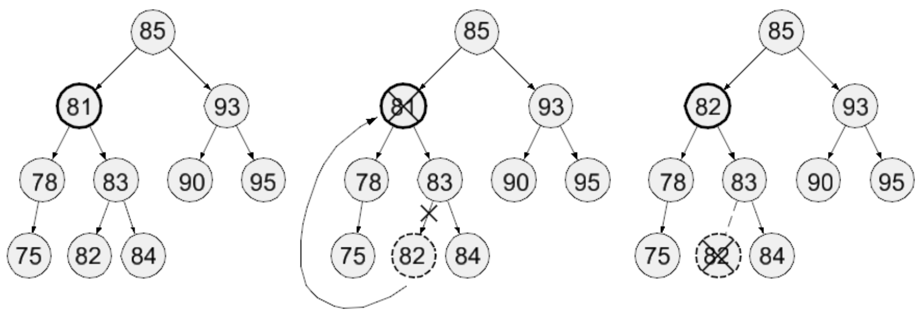


41. Mazání uzlu (2 následníci), varianta LP

Nalezení nejlevějšího prvku v pravém podstromu (list).

Prohození pozice s mazaným prvkem.

Zrušení listu.



42. Varianta PL

Postup mazání uzlu:

1. Nalezení uzlu `w` v levém podstromu, který je následníkem `u`.
2. Nalezení nejpravějšího `w` uzlu a jeho předchůdce `p`:
   - Inicializace `p=u`.
   - Dokud existuje pravý podstrom (`w.right!=None`), tak: (1) zapamatuj si uzel `w`: `p=w` (2) inkrementuj `w`: `w=w.right`.
3. Prohození obsahu uzlů `u` a `w`.
4. Smazání uzlu `w`, 2 varianty:
   - `w` je list: smaž list (`w`).
   - `w` není list: smaž uzel s jedním následníkem (`w`).

43. Varianta PL: nerekurzivní varianta

```
def del2Subtrees(self, u):
    w = u.left                         #Left successor
    p = u
    while w.right != None:              #Right-most node in left subtr
        p = w
        w = w.right
    u.data = w.data                     #Swap u <=> v
    if w.left != None or w.right != None:
        self.del1Subtree(p, w)          #Delete successor u + subtree
    else:
        self.delLeaf(p, w)              #Delete successor u
```

44. Prohledávání stromu

Strom lze procházet dvěma způsoby:

- *Prohledávání do hloubky (Depth First Search).*
  Postup stromem dolů tak dlouho, dokud je to možné.
  Jakmile narazí na list, couvá a do první neprozkoumané větve a postup opakuje.
  Analogie procházení bludištěm.
  Implementace zásobníkem.
- *Prohledávání do šířky (Breadth First Search).*
  Postup stromem do stran od kořene.
  Přechod na následující úroveň až po zpracování předchozí.
  Implementace frontou.

Oproti obecným grafům odpadá nutnost značkovat vrcholy⇒neexistence cyklu.

Základní paradigmata používaná při řešení problémů, prohledávání stavového prostoru.

Projdeme -li celý strom, nemůžeme minout řešení ⇒ pozor, vede k exponenciálnímu nárůstu složitosti.

45. Procházení stromu do hloubky

DFS, tzv. backtracking.

Postup od kořene stromem směrem dolů.

V každém uzlu postupujeme vlevo.

Zpracováváme levý podstrom, dokud nedorazíme do listu.

Poté návrat do nejbližšího uzlu vyšší úrovně a pokračujeme pravým stromem.

Pokud prohledány oba podstromy: Návrat do nejbližšího uzlu vyšší úrovně.

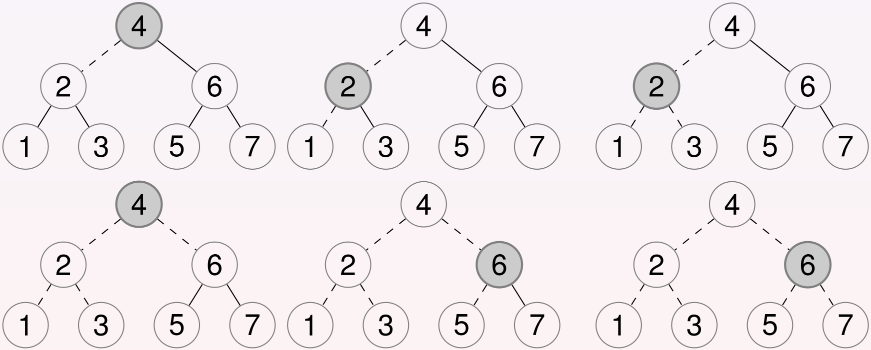
Každý uzel navštíven celkem 3x:

- od předchůdce,
- zpět z levého podstromu,
- zpět z pravého podstromu.

Tento průchod binárním stromem odpovídá metodě preorder.

Implementace s využitím zásobníku.

46. Ukázka procházení stromu DFS



47. Implementace DFS: zásobník

Metoda bez použití značkování.

```
def dfs(self):
    S = []
    S.append(self.root)
    while S:
        u = S.pop()
        if u.left != None:
            S.append(u.left)
        if u.right != None:
            S.append(u.right)
```

48. Procházení stromu do šířky

BFS.

Postup od kořene stromu k listům po jednotlivých úrovních.

Dokud nejsou zpracovány všechny uzly jedné úrovně, nepřesouváme se na úroveň další.

Každá úroveň procházena postupně ve směru od nejlevějšího uzlu k nejpravějšímu.

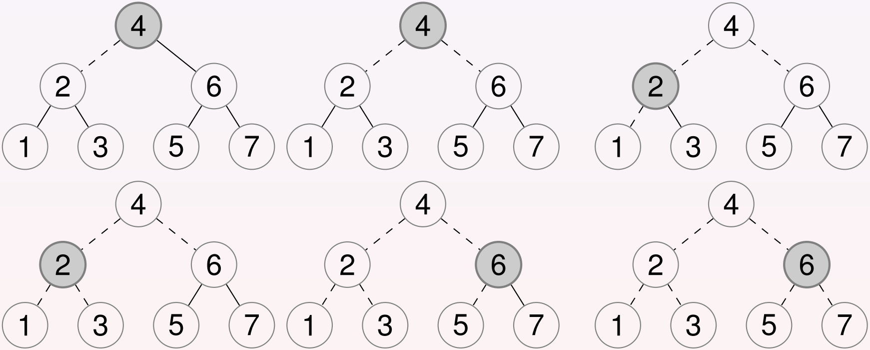
Na rozdíl od DFS se stromem nevracíme zpět.

Implementace velmi podobná předchozí.

Místo zásobníku použita fronta.

Uzly opět není nutné značkovat, graf neobsahuje cyklus.

49. Ukázka procházení stromu do šířky



50. Implementace BFS: fronta

Metoda bez použití značkování.

Záměna zásobníku za frontu.

```
def bfs(self):
    Q = queue.Queue()
    Q.put(self.root)
    while S:
        u = Q.get()
        if u.left != None:
            Q.put(u.left)
        if u.right != None:
            Q.put(u.right)
```

51. Vyvážené stromy

BST stromy mají dobré výkonnostní charakteristiky, pozor na Worst case (degenerované stromy).

$$\begin{aligned} O(N) &\Rightarrow O(N^2)\text{: tvorba stromu} \\ O(1.4\log(N)) &\Rightarrow O(N)\text{: hledání} \end{aligned}$$

Způsobeny nevhodnou konfigurací vstupních množin:

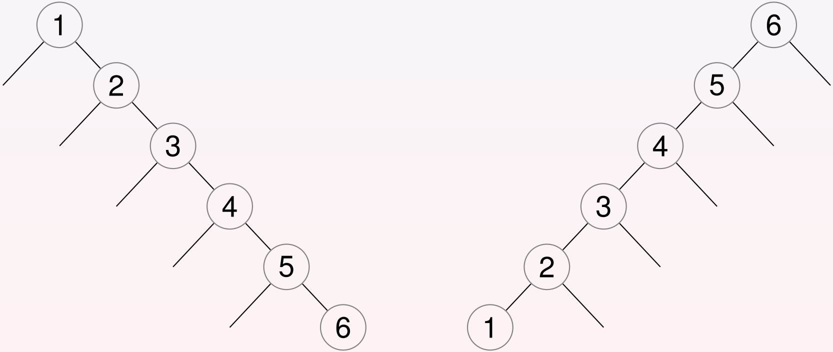
- setříděné množiny,
- reverzně setříděné množiny,
- Střídání velkých a malých hodnot.

Možnosti řešení:

- randomizace vstupních dat ⇒randomizované BST.
- průběžná optimalizace tvaru stromu ⇒ převažování.

52. Degenerovaný strom, setříděná množina

Setříděná množina $X_1 = \{1, 2, 3, 4, 5, 6\}$, reverzně setříděná množina $X_2 = \{6, 5, 4, 3, 2, 1\}$.



53. Randomizované BST stromy

Zavedení prvku náhodnosti při konstrukci stromu.

Výhodou jednoduchost implementace a rychlost.

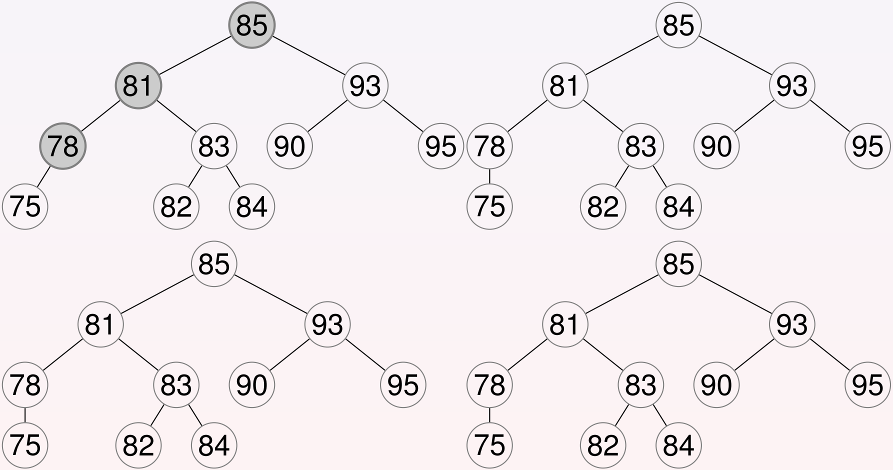
2 metody:

1. *Permutace vstupní množiny*
   Vytvoření počáteční permutace.
   Snaha převést uspořádánou vstupní množinu na randomizovanou.
   "Proházení" prvků ve vstupní množině.
2. *Randomizace vstupu*
   Vytvoření počáteční permutace.
   Výběr "náhodného" uzlu s cílem zamezit vzniku degenerovaného stromu.

Pro specifická data může vést k příliš dlouhým stromům.

Řešení: použití AVL stromů.

54. Ukázka randomizovaného vkládání: x=80



Při přidávání uzlu nutno převažovat: implementačně náročné.

55. AVL strom

Absolutně vyvážený strom.

Výšky L a P podstromu se liší nejvýše o 1.

Délka nepřesáhne 1.4 násobek dokonale vyváženého stromu.

Při přidávání uzlu nutno převažovat: implementačně náročné.

Faktor *BF* vyváženosti AVL stromu

$$BF = h_L - h_P.$$

Pro $f = -1, 0, 1$ netřeba AVL převážit.

Pro $f = \pm 2$ nutné převážení stromu, složitost $O(1)$:

- jednoduché rotace: LL, PP.
- složené rotace: LP, PR.

Příklady:

$h_L = h_P$: po přidání strom zůstává vyvážený: $h_L = h_P + 1$.

$h_L = h_P - 1$: Po přidání strom zůstává vyvážený: $h_L = v_P$.

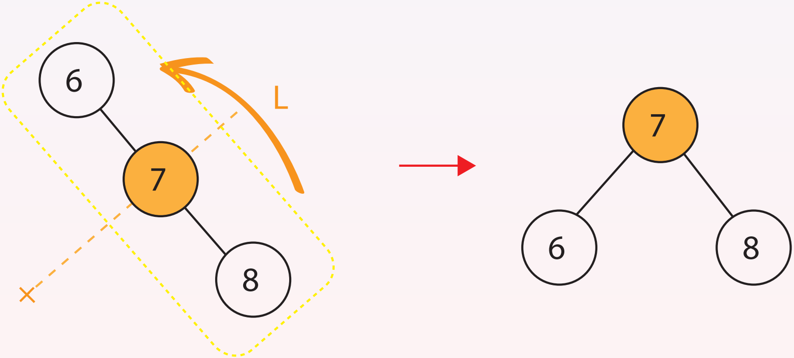
$h_L = h_P + 1$: Po přidání strom není vyvážený: $h_L = h_P + 2$.

56. Jednoduchá rotace: PP

Při prodloužení pravého podstromu pravého podstromu.

Překořenění: prostřední uzel se stává kořenem.

Rotace ve směru CCW: směrem vpravo, L-rotace



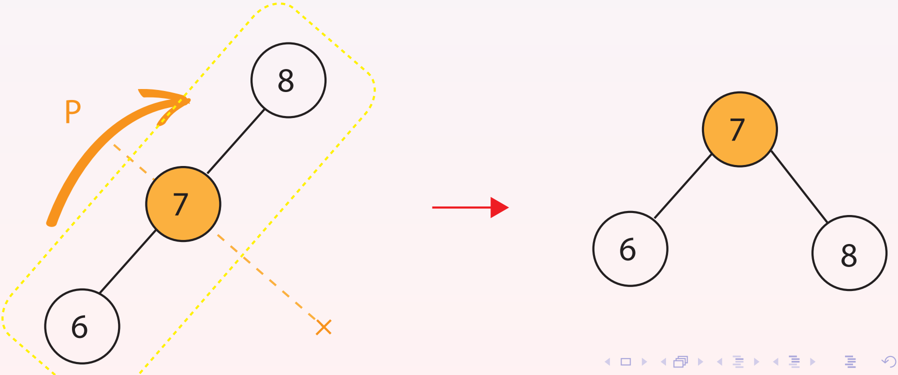
57. Jednoduchá rotace: LL

Při prodloužení levého podstromu levého podstromu.

Zrcadlová symetrie vůči PP.

Překořenění: prostřední uzel se stává kořenem.

Rotace ve směru CW: P-rotace

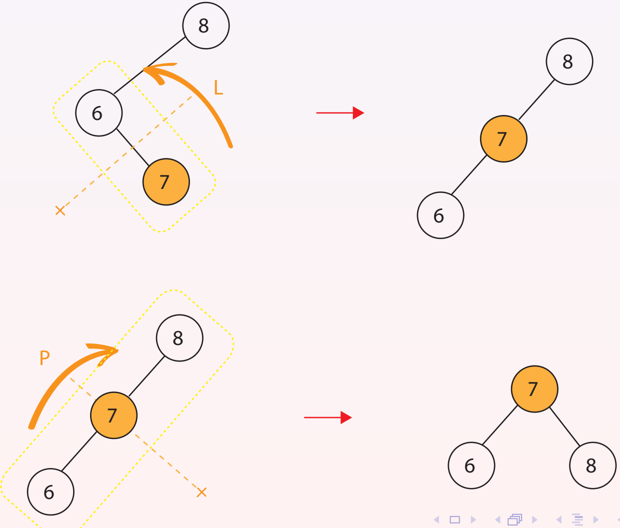


58. Složená rotace: LP

V levém podstromu prodloužen pravý podstrom.

Složená rotace: Nejprve L-rotace, poté P-rotace.

L-rotace: transformace na tvar LL.

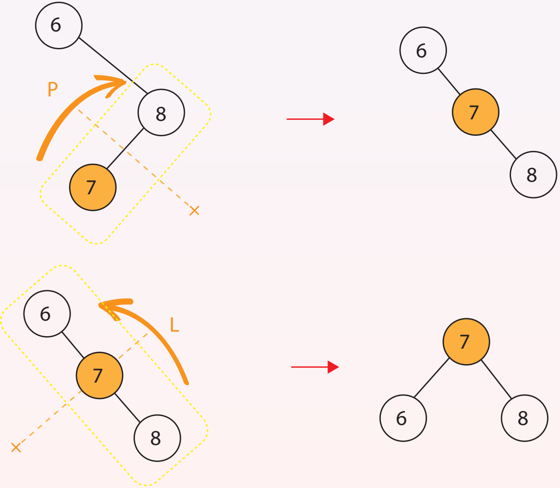


59. Složená rotace: PL

V pravém podstromu prodloužen levý podstrom.

Složená rotace: Nejprve P-rotace, poté L-rotace.

P-rotace: transformace na tvar PP.



60. Komplikovanější LL

Pokud částečně zaplněný P-podstrom, složitější rotace.

Probíhá nad 4 uzly.