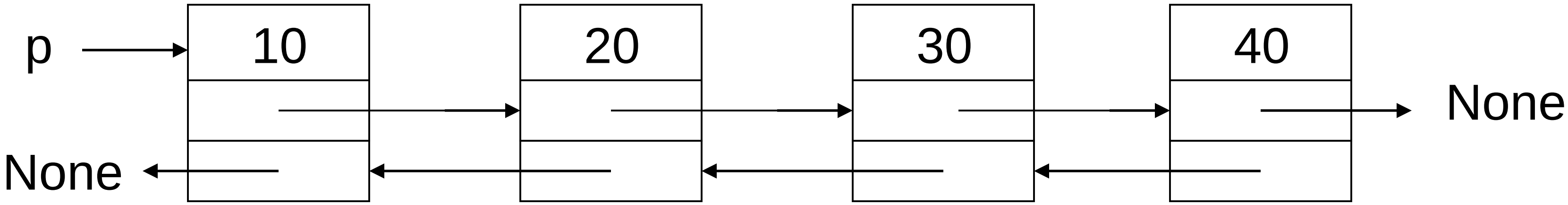
Druhy lineárních spojových seznamů

- obyčejný seznam *(dosud)*
- obousměrný seznam
- cyklický seznam
- seznam s hlavou

Obousměrný seznam

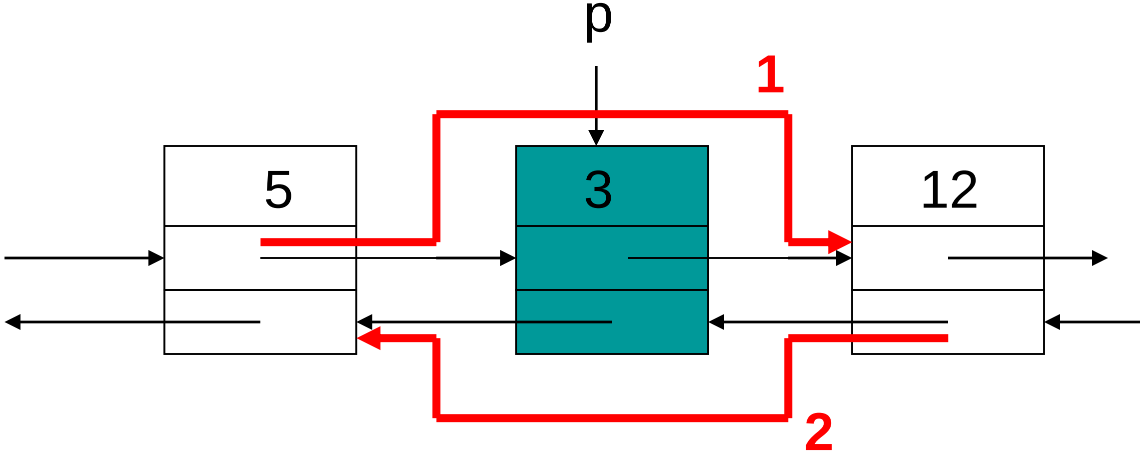
```
class Uzel:
    """uzel obousměrného spojového seznamu"""

    def __init__(self, x = None):
        self.info = x                  # uložená hodnota
        self.za = None                 # následník
        self.pred = None               # předchůdce
```



Vlastnosti
- paměťově náročnější než jednosměrný seznam
- umožňuje procházení seznamem střídavě oběma směry
- některé operace obtížnější (přepojuje se více ukazatelů), jiné naopak snadnější

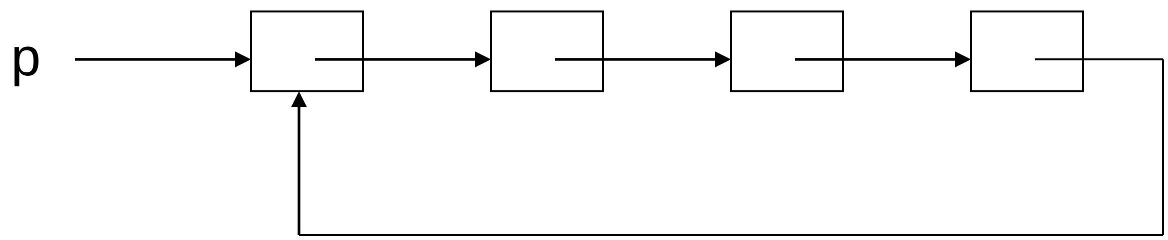
Příklad: vypuštění prvku z obousměrného LSS
- máme ukazatel p na rušený prvek (p není okrajový prvek)

```
p.pred.za = p.za;    {1}
p.za.pred = p.pred;  {2}
```



Cyklický seznam

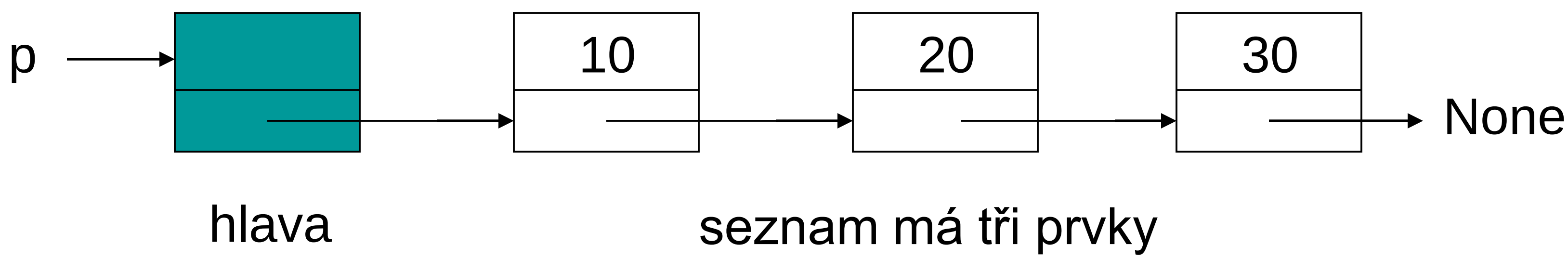
- poslední prvek seznamu neodkazuje na **`None`**, nýbrž na první prvek
- lze použít i pro obousměrné seznamy, potom navíc první prvek seznamu odkazuje svým ukazatelem `pred` na poslední prvek



Seznam s hlavou

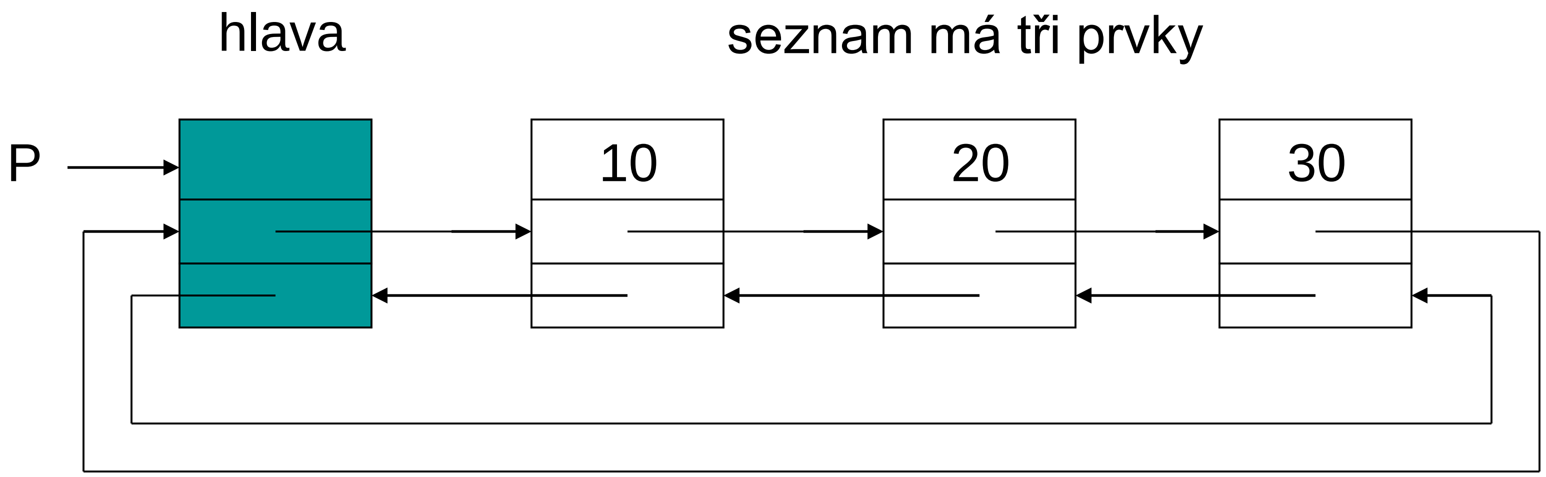
hlava = jeden prvek navíc umístěný na začátku seznamu, neobsahuje uloženou hodnotou
(příp. atribut `info` v hlavě lze využít pro jiné účely)

Použití:
prázdný seznam je tvořen samotnou hlavou (není to tedy jen **`None`** jako u obyčejných seznamů) $\rightarrow$ snáze se programují některé akce spočívající v přidávání resp. odebírání prvků ze seznamu (není třeba stále odlišovat zvláštní případy, kdy seznam je prázdný a kdy přidáváme/odebíráme na začátku seznamu).



Možné kombinace

- např. obousměrný cyklický lineární spojový seznam s hlavou



Dočasná hlava

Někdy pracujeme s obyčejnými spojovými seznamy, ale pro snazší realizaci požadované operace se vyplatí použít u výsledného seznamu dočasnou hlavu a před vrácením výsledku ji opět zrušit.

Příklad:
Uzly zadaného lineárního spojového seznamu celých čísel rozdělit do dvou seznamů takto:
- do jednoho dát sudé hodnoty (se zachování vzájemného pořadí)
- do druhého dát liché hodnoty (se zachování vzájemného pořadí).

Rekurze

- objekt je definován pomocí sebe sama

V programování ve dvou hladinách:

- **rekurzivní algoritmus** – řešení úlohy je definováno pomocí řešení menších instancí téhož problému (tzn. podúloh stejného charakteru)

- **rekurzivní volání funkce** – funkce volá sama sebe (přímo nebo případně nepřímo prostřednictvím jiných funkcí)

Většinou se rekurzivní algoritmy realizují pomocí rekurzivních volání, ale není to nezbytné:

- rekurzivní algoritmus lze realizovat bez rekurzivních volání (pomocí vlastního zásobníku na uložení rozpracovaných nedokončených podúloh)
$\rightarrow$ více práce pro programátora, program obvykle delší a méně přehledný, výpočet ale může být o něco efektivnější

- rekurzivní volání lze teoreticky použít i při realizaci nerekurzivních iteračních algoritmů (dokonce každý cyklus lze nahradit rekurzivní procedurou)
$\rightarrow$ většinou nevhodné, nečitelný a méně efektivní program

Ukončení rekurze

- rekurzivní volání funkce musí být vázáno na nějakou podmínku, která časem jistě přestane platit → jinak „zacyklení výpočtu“

- na rozdíl od nekonečného while-cyklu dojde v Pythonu k zastavení výpočtu po dosažení maximální povolené hloubky rekurze (1012)

```
def rekurze(p):
    print(p)
    rekurze(p+1)

rekurze(1)
```

Poznámka: programovací jazyky bez limitu na hloubku rekurze → běhová chyba přetečení zásobníku (stack overflow)

Průběh výpočtu při rekurzivním volání funkce

- najednou je rozpočítáno více exemplářů téže funkce

- všechny počítají podle téhož kódu

- každý exemplář má na volacím zásobníku svůj vlastní aktivační záznam s lokálními proměnnými, parametry a technickými údaji (kde je rozpočítán, návratová adresa)

- funkce nemá přístup k lokálním proměnným jiného rekurzivního exempláře

Příklad 3: výpis znaků ze vstupu pozpátku

```
def otoc():
    u = input("Znak: ")
    if u != " ":
        otoc()
    print(u)
```

Vstup:	*Výstup:*
A	<mezera>
B	C
C	B
<mezera>	A

Příklady jednoduchých rekurzivních funkcí

Palindrom – řetězec se čte stejně zleva i zprava (je symetrický)

```
def palindrom1(s):
    n = len(s)
    for i in range(n//2):
        if s[i] != s[n-i-1]:
            return False
    return True

def palindrom2(s):
    if len(s) <=1 :
        return True
    else:
        return s[0] == s[-1] and palindrom2(s[1:-1])
```

Eukleidův algoritmus (odčítací) realizovaný cyklem *(bylo dříve):*

```
def nsd(x, y):
    while x != y:
        if x > y:
            x -= y
        else:
            y -= x
    return x
```

Eukleidův algoritmus realizovaný rekurzivní funkcí
- přesně kopíruje rekurzivní vztah pro NSD,
na němž je Eukleidův algoritmus založen:

když X > Y NSD(X ,Y) = NSD(X-Y, Y)
když X < Y NSD(X ,Y) = NSD(X, Y-X)
když X = Y NSD(X ,Y) = X

```
def nsd(x, y):
    if x > y:
        return nsd(x-y, y)
    elif x < y:
        return nsd(x, y-x)
    else:
        return x
```

Faktoriál *n!* (součin čísel od 1 do *n*, pro *n* >= 0)

rekurzivní definice: $n! = 1$ pro $n = 0$

$n! = n \cdot (n-1)!$ pro $n > 0$

```
def faktorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f

def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n-1)
```

v obou případech časová složitost O(*n*)

Fibonacciho čísla

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$ pro $n > 1$

rekurzivní definice posloupnosti čísel
→ realizace rekurzivní funkcí přesně podle definice:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

funkce teoreticky správná, ale časová složitost exponenciální
→ pro n > cca 40 prakticky nepoužitelná
důvod: mnohokrát se opakovaně počítají stejné věci

Možnosti řešení:

1. rekurzivní algoritmus + pomocné pole velikosti *n*
   pro uložení již spočítaných funkčních hodnot

   → každé F_i se počítá jen jednou → časová složitost O(*n*)

   „chytrá rekurze“
   kešování hodnot (cache = mezipaměť)
   memoizace (memory = paměť)
   dynamické programování

2. počítat hodnoty iteračně odspodu – v pořadí F_1, F_2, … F_n
$\rightarrow$ časová složitost O(*n*), navíc stačí konstantní paměť

dynamické programování

```
def fib(n):
    if n == 0:
        return 0
    a = 0; b = 1
    while n > 1:
        a, b = b, a+b
        n -= 1
    return b
```

3. z rekurzivní definice odvodit explicitní vzorec a počítat podle něj

$$F_n = \frac{\sqrt{5}}{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

4. využití rychlého umocňování matice

Platí rovnost $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$

Tedy $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} F1 \\ F2 \end{pmatrix}$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} . \begin{pmatrix} F1 \\ F2 \end{pmatrix} = \begin{pmatrix} F2 \\ F3 \end{pmatrix}$$

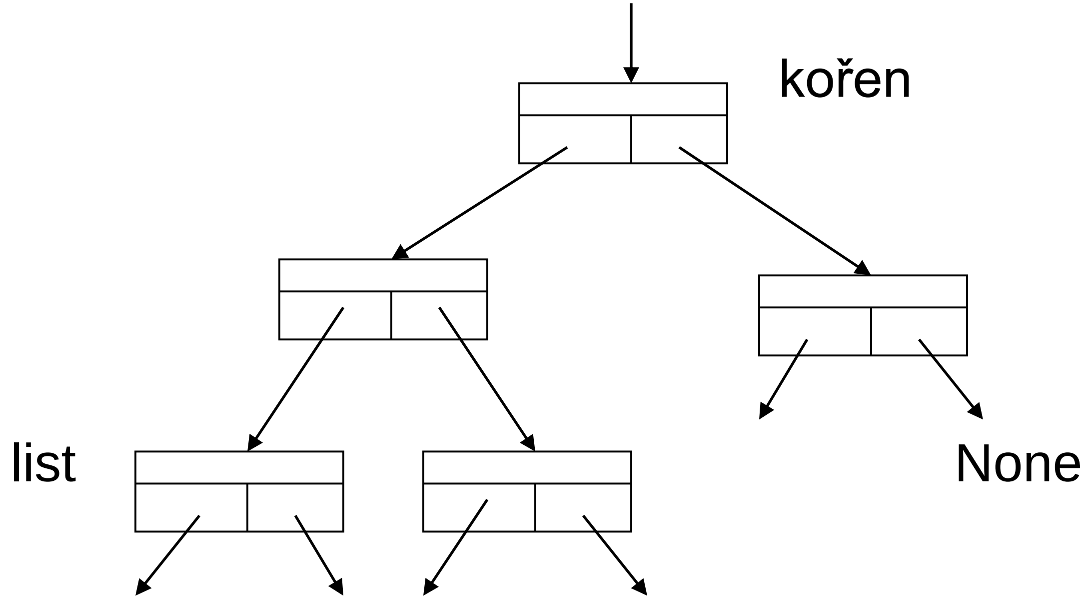
$$\cdots$$

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n . \begin{pmatrix} F0 \\ F1 \end{pmatrix} = \begin{pmatrix} Fn \\ Fn+1 \end{pmatrix}$$

Matici $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$ spočítáme rychlým umocňováním

→ časová složitost O(log *N)*

Binární strom



```
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None, l = None, r = None):
        self.info = x             # uložená hodnota
        self.levy = l             # levý syn
        self.pravy = r            # pravý syn
```

Příklady použití binárního stromu:

- halda *(bylo)*
- binární vyhledávací strom *(bude)*
- reprezentace aritmetického výrazu *(bude)*

Průchod binárním stromem (do hloubky)
- v každém vrcholu provést zvolenou akci
(např. vypsat hodnotu atributu “info”)

Varianty průchodu podle pořadí zpracování vrcholů:

PREORDER – nejprve zpracuje vrchol, pak jde postupně do obou jeho podstromů

INORDER – nejprve jde do levého podstromu, pak zpracuje vrchol, nakonec jde do pravého podstromu

POSTORDER – nejprve jde postupně do obou podstromů vrcholu, pak zpracuje vrchol samotný

```python
class Vrchol:
    """vrchol binárního stromu"""

    def __init__(self, x = None):
        self.info = x                  # uložená hodnota
        self.levy = None               # levý syn
        self.pravy = None              # pravý syn

    def preorder(self):
        """průchod stromem s kořenem v tomto vrcholu
           metodou preorder, vypisuje hodnoty všech
           vrcholů
        """
        print(self.info)
        if self.levy != None:
            self.levy.preorder()
        if self.pravy != None:
            self.pravy.preorder()
```

```python
def inorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou inorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.inorder()
    print(self.info)
    if self.pravy != None:
        self.pravy.inorder()

def postorder(self):
    """průchod stromem s kořenem v tomto vrcholu
       metodou postorder, vypisuje hodnoty všech
       vrcholů"""
    if self.levy != None:
        self.levy.postorder()
    if self.pravy != None:
        self.pravy.postorder()
    print(self.info)
```

Průchod do hloubky bez použití rekurze – pomocí zásobníku

```
DO_ZÁSOBNÍKU(Kořen)
dokud ZÁSOBNÍK není prázdný
   P = ZE_ZÁSOBNÍKU
   AKCE(P.info)
   jestliže P.pravy != None: DO_ZÁSOBNÍKU(P.pravy)
   jestliže P.levy != None: DO_ZÁSOBNÍKU(P.levy)
```

Průchod do šířky (po vrstvách) – pomocí fronty

```
DO_FRONTY(Kořen)
dokud FRONTA není prázdná
   P = Z_FRONTY
   AKCE(P.info)
   jestliže P.levy != None: DO_FRONTY(P.levy)
   jestliže P.pravy != None: DO_FRONTY(P.pravy)
```

Časová složitost všech uvedených metod průchodu:

O(N), kde N je počet vrcholů ve stromu

- neboť každý vrchol stromu je navštíven právě jednou
  a jeho zpracování má konstantní časovou složitost

Obecný strom

1. známe maximální stupeň větvení *M*

- podobná reprezentace jako u binárního stromu
- v každém uzlu je připraveno *M* odkazů na syny,
  z nich několik prvních je využito, ostatní mají hodnotu **`None`**
- v listu mají všechny odkazy na syny hodnotu **`None`**
- použitelné, pokud *M* předem známe a je dostatečně malé

2. obecné řešení

- v každém uzlu je uložen seznam odkazů na syny potřebné délky
- v listu je tento seznam prázdný
- *viz program na následující straně*

```python
class Vrchol:
    """vrchol obecného stromu"""

    def __init__(self, x = None):
        self.info = x                 # uložená hodnota
        self.synove = []              # seznam synů

    def pruchod(self):
        """
        průchod stromem s kořenem v tomto vrcholu
        metodou preorder, vypisuje hodnoty všech
        vrcholů
        """
        print(self.info)
        for x in self.synove:
            x.pruchod()
```

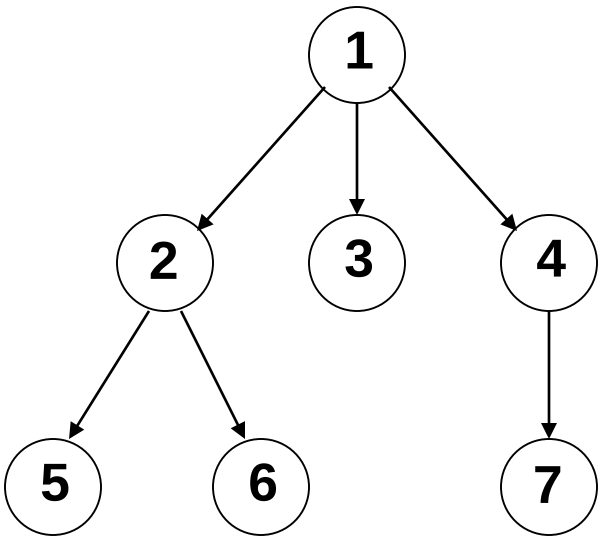
3. kanonická reprezentace

- reprezentace obecného stromu binárním stromem

```
class Vrchol:
    """vrchol stromu"""

    def __init__(self, x = None):
        self.info = x                # uložená hodnota
        self.syn = None              # nejstarší syn
        self.bratr = None            # mladší bratr
```

- každý uzel ukazuje jen na svého nejstaršího syna (položka „syn“)
- všichni synové téhož uzlu jsou navzájem propojeni pomocí odkazů „bratr“
- v listu má položka „syn“ hodnotu **None**

příklad stromu:



uložení v datové struktuře: