

# Státnicový okruh 3: Algoritmizace a programování

26. dubna 2022

## **Obsah**

# 1

**Lineární datové struktury:** seznam, zásobník, fronta. **Úloha třídění a rozdělení třídicích algoritmů.** Metody třídění porovnáváním: insert sort, Select sort, Bubble sort, Quick sort, Merge sort, Heap sort. Další metody třídění: Counting sort, Radix Sort, Bucket sort, vnější třídění. Složitosti třídicích algoritmů. Pořádkové statistiky.

## 1.1 Lineární datové struktury: seznam, zásobník, fronta

Linearni datove struktury se vyznacují tim, ze jednotlive prvky mnoziny dat jsou v nich ulozeny postupne za sebou.

### 1.1.1 Lineární seznam

je dynamická (je vytvářen postupně - paměť pro něj není přidělena na jednou jako u pole) datová struktura, vzdáleně podobná poli, obsahující jednu a více datových položek (struktur) stejného typu (uzly), které jsou navzájem lineárně provázany vzájemnými odkazy pomocí ukazatelů nebo referencí. Aby byl seznam lineární, nesmí existovat cykly ve vzájemných odkazech. Zavádí se také ukazatel nebo reference na aktuální (vybraný) prvek seznamu. Na konci (a začátku) seznamu musí být definována zarážka označující konec seznamu. V **jednosměrném seznamu** odkazuje každá položka na položku následující a v **obousměrném seznamu** odkazuje položka na následující i předcházející položky. Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o **kruhový seznam**.

**operace nad seznamem:**

- **Vyhledání prvku:** Sekvenční přístup = začneme od začátku seznamu a postupně procházíme, dokud prvek nenalezneme (nebo nedojdeme na konec seznamu). Časová složitost  $\theta(n)$ .
- **Přidání na začátek:** Vytvoří se nový uzel, kam se uloží hodnota a nastaví se ukazatel na původní první prvek. Časová složitost  $\theta(1)$ .
- **Přidání uvnitř seznamu:** Vyhledáme pozici, kam chceme přidat, vytvoříme nový uzel, předchozí ukazatel přemapujeme na nový uzel a do nového uzlu nastavíme ukazatel následujícího prvku seznamu. Časová složitost  $\theta(n)$ .
- **Odstranění uzlu:** Pokud odstraňujeme první prvek, stačí nastavit začátek seznamu na následující. Při odstranění uvnitř seznamu se ukazatel předchozího nastaví na následující prvek za rušeným uzlem. Časová složitost  $\theta(n)$ .

Práce se seznamem je komplikovanější než s polem. Musíme vytvářet uzly, nastavovat hodnoty jejich ukazatelů atd. Navíc nemáme přímý přístup k jednotlivým datovým prvkům uloženým v seznamu.

### 1.1.2 Zásobník

Významná hojně používaná datová struktura typu LIFO (last in first out). **Dno zásobníku** je fixně stanovený konec struktury, začátek struktury označován jako **vrchol zásobníku**.

**operace na zásobníku:**

- **Push:** Vložení prvků na vrchol zásobníku.

- **Pop**: Odebrání prvku z vrcholu zásobníku.
- **Top**: Dotaz na vrchol zásobníku.
- **Is\_empty**: True, pokud je dno totožné s vrcholem.

Prkvy kromě vrcholu zásobníku nejsou přímo přístupné. K implementaci je možné využít pole (začátek pole - dno zásobníku) nebo seznam (konec seznamu - dno zásobníku).

### 1.1.3 Fronta

Datová struktura typu FIFO (first in first out).

#### operace na frontě:

- **Add** = enqueue: Vložení prvku na konec fronty.
- **Remove** = dequeue: Odebrání prvku ze začátku fronty.

K implementaci pole nebo seznamem.

**Modifikací je prioritní fronta:** K jeho prvkům se na rozdíl od prvků obyčejné fronty váže ještě priorita: Pokud mají prkny stejnou prioritu, opouští frontu v pořadí, v jakém do ní byly vloženy, ale prvek s vyšší prioritou prkny s nižší prioritou předběhne a jde na výstup dříve.

#### operace na prioritní frontě:

- **zařazení do fronty s udanou prioritou**: přijímá jako vstup prvek a jeho prioritu a prvek s jeho prioritou zařadí do fronty
- **odeber nejstarší z prvků s nejvyšší prioritou**: odstraní z fronty ten z prvků s nejvyšší prioritou, který je tam nejdéle, a vrátí ho jako svůj výstup

Někdy jsou implementovány i další funkce, například možnost zjistit prvek s nejvyšší prioritou bez toho, že by byl odstraněn.

## 1.2 Úloha třídění a rozdělení třídících algoritmů

Algoritmy třídění jsou založeny na dvou základních operacích - **srovnání** a **přesuny**. Podle toho, jak tyto přesuny probíhají, lze algoritmy rozdělit:

- třídění vkládáním (insert)
- třídění výměnou (bubble)
- třídění výběrem (select)

### 1.2.1 Úloha třídění

Potřeba setřídit údaje se v praxi objevuje velmi často. Proto algoritmy pro třídění patří do skupiny základních, velmi používaných algoritmů. Úloha třídění spočívá v seřazení prvků datové množiny vzestupně podle velikosti jejich zvoleného třídícího klíče.

Mějme n datových prvků  $a_1, a_2, \dots, a_n$  a nechť jejich třídící klíč je takového datového typu, že má definovanou operaci srovnání dle velikosti. Cílem je prvky seřadit do posloupnosti  $a_{i_1}, a_{i_2}, \dots, a_{i_n}$  tak, že platí  $a_{i_1}.key \leq a_{i_2}.key \leq \dots \leq a_{i_n}.key$ . Můžeme mít také opačné setřídění - od největší hodnoty po nejmenší.

### 1.2.2 Rozdělení třídících algoritmů

Třídící algoritmy lze rozdělit do dvou skupin:

- Algoritmy vnitřního třízení
- Algoritmy vnějšího třízení

**Algoritmy vnitřního třízení** Základní vlastností je, že všechny tříděné prvky jsou během třídění uloženy ve vnitřní paměti počítače. Výhodou pak je, že všechny operace nad prvky pracují jen v rámci vnitřní paměti. Vhodné ke třídění malého množství prvků, které se vejdu do vnitřní paměti.

- Hlavní třídící metody
  - insert sort
  - select sort
  - bubble sort
- účinnější metody
  - quick sort
  - merge sort
  - shell sort
  - heap sort
- specifické metody
  - radix sort
  - bucket sort

**Algoritmy vnějšího třízení** Tříděné prvky jsou uloženy v souborech ve vnější paměti. Průběžně je část prvků načtena do vnitřní paměti, setřízena a vrácena zpět do vnější paměti. Obecně je práce s vnější pamětí pomalejší. Užití při velkém množství dat, kdy není možné prvky uložit do vnitřní paměti.

- třídění se stejným počtem vstup-výstupních souborů
- vnější třízení s využitím vnitřního
- polyfázové třídění

## 1.3 Metody třídění porovnáváním: insert sort, Select sort, Bubble sort, Quick sort, Merge sort, Heap sort

### 1.3.1 Insert sort

Pole, ve kterém jsou uloženy prvky, je v průběhu třídění rozděleno na dvě části - setříděnou část (ta je první) a nesetříděnou část. V každém kroku vezmeme první prvek v nesetříděné části, projdeme setříděnou část, najdeme v ní místo vložení a na toto místo prvek vložíme. Vložení se provede tak, že posuneme všechny prvky počínaje místem vložení až po konec setříděné části o jednu pozici dozadu, aby se uvolnilo místo pro vkládaný prvek, a na uvolněnou pozici nový prvek vložíme. Hledání pozice vložení lze udělat postupným procházením od začátku setříděné části a srovnáváním vkládaného prvku s prvky setříděné části. Výhodnější ale v tomto případě je zvolit opačný směr, začít procházení od konce setříděné části, neboť můžeme přitom souběžně dělat i posouvání prvků setříděné části o jednu pozici dozadu.

#### 1. Počáteční krok

Na začátku vytvoříme počáteční rozdělení pole na setříděnou a nesetříděnou část. Setříděnou částí bude první prvek v poli, nesetříděnou částí bude zbývajících  $n-1$  prvků.

#### 2. Průběžný krok

Nechť setříděná část je nyní tvořena k prvky a za ní je zbývajících  $n-k$  prvků, které tvoří nesetříděnou část. Vezmeme první prvek z nesetříděné části ( $a_k$ ), uložíme ho do pomocné proměnné, označme ji  $x$ , a pozici tohoto prvku považujeme za volnou. Nyní od zadu procházíme prvky v setříděné části a postupně pro indexy  $r = k-1, k-2, \dots$  srovnáváme, zda mezi prvkem  $a_r$  a uloženým prvkem  $x$  platí  $a_r > x$ . Pokud ano, posuneme prvek  $a_r$  o jednu pozici dozadu (tato pozice za prvkem je v tomto okamžiku volná) a jeho původní pozice se nyní stane novou volnou pozicí. Celý proces srovnání skončí v okamžiku, kdy buďto pro některý prvek platí  $a_r \geq x$  anebo jsme celou setříděnou část již prošli (a tato je rovněž i posunuta dozadu). Pozice vložení ve chvíli, kdy ukončíme srovnávání, odpovídá stávající volné pozici. Na ni uložíme prvek obsažený v proměnné  $x$ . Po každém provedení průběžného kroku se zvětší velikost setříděné části o jeden prvek, až nakonec setříděná část bude zahrnovat všechny prvky.

---

#### Algorithm 1 Pseudokód

---

```
procedure INSERTIONSORT(A, n)
    for  $i \leftarrow 1$  to  $n - 1$  do
         $x \leftarrow A[i]$ 
         $j \leftarrow i - 1$ 
        while  $j \geq 0$  and  $A[j] > x$  do
             $A[j + 1] \leftarrow A[j]$ 
             $j \leftarrow j - 1$ 
         $A[j + 1] \leftarrow x$ 
```

---

**Příklad.** Mějme seřidit přirozená čísla

7 1 2 8 4 5 3 9

Po prvním kroku bude pole rozděleno na dvě části:

7 1 2 8 4 5 3 9

První prvek v nesetříděné části je číslo 1. Uložíme ho do proměnné  $x$  a jeho místo je nyní volné

7 • 2 8 4 5 3 9     $x=1$

Srovnáme 7 s  $x$  : posunutí 7 doprava:

• 7 2 8 4 5 3 9     $x=1$

Už není co srovnávat, vložíme  $x$  na volnou pozici a posuneme hranici setříděné části:

1 7 2 8 4 5 3 9

Další krok - uložíme první prvek 2 do proměnné  $x$  a jeho místo je volné:

1 7 • 8 4 5 3 9     $x=2$

Srovnáme 7 s  $x$  : posunutí 7 doprava:

1 • 7 8 4 5 3 9     $x=2$

Srovnáme 1 s  $x$  : vložení  $x$  a posunutí hranice setříděné části:

1 2 7 8 4 5 3 9

Další krok - uložíme první prvek 8 do proměnné  $x$  a jeho místo je volné:

1 2 7 • 4 5 3 9     $x=8$

Srovnáme 7 s  $x$  : vložení  $x$  a posunutí hranice setříděné části:

1 2 7 8 4 5 3 9

Atd.

### 1.3.2 Select sort

Část obsahující již setříděné prvky je první a část s doposud nesetříděnými prvky je za ní. V nesetříděné části se v každém kroku najde nejménší prvek a ten se přesune na konec již setříděné části.

#### 1. Počáteční krok

Na začátku celé pole, tj. všech n prvků tvoří nesetříděnou část.

#### 2. Průběžný krok

Nechť setříděná část, která je na začátku, má n-k prvků a setříděná část za ní má k prvků. V nesetříděné části vybereme nejménší její prvek. Postupujeme tak, že si na začátku zapamatujeme pozici prvního prvku nesetříděné části. Nyní postupně procházíme prvky nesetříděné části počínaje od jejího druhého prvku a každý z nich srovnáváme s prvkem na zapamatované pozici. Jestliže srovnávaný prvek je menší, jeho pozice se stane novou zapamatovanou pozicí. Je zřejmé, že na konci je na zapamatované pozici nejménší prvek. Ten musíme přesunout na konec setříděné části - vyměníme ho s prvním prvkem v nesetříděné části. Po každém provedení průběžného kroku se zvětší velikost setříděné části o jeden prvek. V okamžiku, kdy setříděná část bude obsahovat n-1 prvků, je třídění ukončeno, protože v nesetříděné části tímto zůstane už jen největší prvek, který je na konci a tudíž na své cílové pozici.

---

#### Algorithm 2 Pseudokód

---

```
procedure SELECTIONSORT(A, n)
    for i ← 0 to n – 2 do
        m ← i
        for j ← i + 1 to n – 1 do
            if A[j] < A[m] then
                m ← j
            exch (A[i], A[m])
```

---

**Příklad.** Mějme setřídit přirozená čísla

7 1 2 8 4 5 3 9

Zapamatujeme si pozici prvku 7

7 1 2 8 4 5 3 9

Srovnáme 7 s 1 ☐ zapamatujeme si pozici prvku 1

7 1 2 8 4 5 3 9

Srovnáme 1 s 2, 8, 4, 5, 3, 9

Zapamatovaná je pozice prvku 1, ten vyměníme s prvním prvkem 7

1 7 2 8 4 5 3 9

Další krok – zapamatujeme si pozici prvku 7

1 7 2 8 4 5 3 9

Srovnáme 7 s 2 ☐ zapamatujeme si pozici prvku 2

1 7 2 8 4 5 3 9

Srovnáme 2 s 8, 4, 5, 3, 9

Zapamatovaná je pozice prvku 2, ten vyměníme s prvkem 7

1 2 7 8 4 5 3 9

Další krok – zapamatujeme si pozici prvku 7

1 2 7 8 4 5 3 9

Srovnáme 7 s 8, 4 ☐ zapamatujeme si pozici prvku 4

1 2 7 8 4 5 3 9

Srovnáme 4 s 5, 3 ☐ zapamatujeme si pozici prvku 3

1 2 7 8 4 5 3 9

Srovnáme 3 s 9

Zapamatovaná je pozice prvku 3, ten vyměníme s prvkem 7

1 2 3 8 4 5 7 9

Atd.

### 1.3.3 Bubble sort

Při třídění přímou výměnou procházíme postupně pole s tříděnými prvky směrem od začátku k jeho konci a srovnáváme sousední dvojice, zda prvky v nich jsou podle velikosti ve správném pořadí. Jestliže ne, tedy větší prvek je před menším, provedeme jejich vzájemnou výměnu. V dalším kroku celý postup zopakujeme, ale už bez posledního prvku, tedy jen s prvními n-1 prvky. Při něm se obdobným způsobem dostane na své cílové místo předposlední prvek setříděné posloupnosti. V následujícím kroku už budeme procházet jen n-2 prvky atd. Je zřejmé, že každým průchodem se délka procházené části sníží o jeden prvek, až nakonec v posledním průchodu bude procházená část mít jen dva prvky a po jejich srovnání a případné výměně je třídění dokončeno.

---

#### Algorithm 3 Pseudokód

---

```
procedure BUBBLESORT(A, N)
    for i ← n - 2 downto 0 do
        for j ← 0 to i do
            if A[j] > A[j + 1] then
                exch (A[i], A[m])
```

---

**Příklad.** Budeme opět třídit posloupnost

7 1 2 8 4 5 3 9

Srovnáme první dva sousední prvky 7 a 1

7 1 2 8 4 5 3 9   □ výměna

Srovnáme další dva sousední prvky 7 a 2

1 7 2 8 4 5 3 9   □ výměna

Srovnáme další dva sousední prvky 7 a 8

1 2 7 8 4 5 3 9

Srovnáme další dva sousední prvky 8 a 4

1 2 7 8 4 5 3 9   □ výměna

Srovnáme další dva sousední prvky 8 a 5

1 2 7 4 8 5 3 9   □ výměna

Srovnáme další dva sousední prvky 8 a 3

1 2 7 4 5 8 3 9   □ výměna

Srovnáme poslední dva sousední prvky 8 a 9

1 2 7 4 5 3 8 9

V dalším průchodu budeme srovnávat už jen 7 prvních prvků:

1 2 7 4 5 3 8 9

Srovnáme 1 s 2

Srovnáme 2 s 7

Srovnáme 7 s 4

1 2 7 4 5 3 8 9   □ výměna

Srovnáme 7 s 5

1 2 4 7 5 3 8 9   □ výměna

Srovnáme 7 s 3

1 2 4 5 7 3 8 9   □ výměna

Srovnáme 7 s 8

1 2 4 5 3 7 8 9

Atd.

### 1.3.4 Quick sort

Quicksort je efektivní metoda třídění výměnou. Princip metody spočívá v tom, že v poli zvolíme určitý prvek (bývá označován jako pivot), označme ho  $A_p$ , a postupnými výměnami rozdělíme pole na dvě části. V první části (označme ji L) budou prvky, které nejsou větší než zvolený prvek  $A_p$ , a v druhé části (R) budou prvky, které nejsou menší než  $A_p$ . Postupným rekurzivním opakováním tohoto postupu se tříděné části stávají postupně stávají menší, až nakonec nastane stav, že každá část obsahuje jen jeden prvek, čímž třídění skončí.

Klíčovou záležitostí je volba pivota  $A_p$ . Optimální by bylo zvolit ho tak, aby vzniklé části L a R měly stejnou velikost. To by znamenalo najít prvek, jehož hodnota je uprostřed hodnot všech prvků (medián). Nalezení takového prvku je ale tak časově náročné, že by to natolik zhoršilo účinnost třídění, že by se třídění stalo neefektivní. Proto se volí mnohem jednodušší způsob – jako pivot se vezme prvek, který je na určité pozici v tříděné části pole.

**Pivot je poslední prvek** Popis algoritmu:

A – je pole s tříděnými prvky, počet prvků v poli je n

l,r – jsou indexy počátku a konce části pole, která je právě tříděna

i,j – jsou průběžné indexy používané pro procházení polem ve směrech odpředu a odzadu

Popis algoritmu:

1. Počáteční krok

Na počátku právě tříděnou částí bude celé pole, tj. nastavíme  $l = 0$ ,  $r = n-1$ .

2. Třídící krok

Do proměnné  $x$  uložíme pivota  $x = A[r]$  a průběžné indexy nastavíme před začátek právě tříděné části a na konec tříděné části  $i = l-1$ ,  $j = r$ .

3. Výměny

Začneme inkrementovat index  $i$  tak dlouho, až najdeme prvek, pro který platí  $a_i \geq x$ . Následně začneme dekrementovat index  $j$  tak dlouho, až buďto najdeme prvek, pro který platí  $a_j \leq x$ , anebo nastane  $i \geq j$ .

Pokud  $i < j$ , uděláme výměnu prvků  $A_i$  a  $A_j$  a pokračujeme krokem 3.

Je-li  $i \geq j$ , ukončíme výměny a přejdeme ke kroku 4.

4. Prvky  $A$  i a  $A$   $r$  mezi sebou vyměníme.

Je-li  $l < i - 1$ , aplikujeme krok 2. na část pole s koncovými indexy  $l$  a  $i - 1$ .

Je-li  $i + 1 < r$ , aplikujeme krok 2. na část pole s koncovými indexy  $i + 1$  a  $r$ .

**Pivot je prvek uprostřed** Popis algoritmu:

1. Počáteční krok

Na počátku aktuální tříděnou částí bude celé pole, tj. nastavíme  $l = 0$ ,  $r = n-1$ .

2. Třídící krok

Do proměnné  $x$  uložíme střední prvek  $x = A[\frac{l+r}{2}]$  a průběžné indexy nastavíme na začátek a konec právě tříděné části  $i = l$ ,  $j = r$ .

Nyní procházíme tříděnou část směrem dozadu - zvětšujeme index  $i$  tak dlouho, až najdeme prvek, pro který platí  $A_i \geq x$ .

Následně procházíme tříděnou část směrem dopředu - snižujeme index  $j$ , tak dlouho, až najdeme prvek, pro který platí  $A_j \leq x$ .

Prvky  $A_i$  a  $A_j$  mezi sebou vyměníme a následně  $i++$  a  $j-$ .

Tento proces provádíme tak dlouho, dokud nenastane  $i > j$ . V tomto okamžiku je buďto  $i=j+1$  anebo je  $i=j+2$ .

Má-li část L více než jeden prvek ( $l < j$ ), pak rekurzivně provedeme třídící krok 2. na této části pole - její počáteční a koncový index je dán současnými hodnotami proměnných  $l,j$ .

Má-li část R více než jeden prvek ( $i < r$ ), pak rekurzivně provedeme třídící krok 2. na této části pole - její počáteční a koncový index je dán současnými hodnotami proměnných  $i,r$ .

---

**Algorithm 4** Pseudokód

---

```
procedure QS(A, l, r)
    x ← A[l + r/2]
    i ← l
    j ← r
    do
        while A[i] < x do
            i ← i + 1
        while x < A[j] do
            j ← j - 1
        if i < j then
            break
        exch(A[i], A[j])
        i ← i + 1
        j ← j - 1
    while i ≤ j
    if l < j then
        QS(A, l, j)
    if i < r then
        QS(A, i, r)
```

---

### 1.3.5 Heap sort

Halda je určitý typ binárního stromu. V každém jeho uzlu je jeden tříděný prvek. Dále mezi prvkem v uzlu a prvky uloženými v jeho následnících (má-li nějaké) platí vztah, že prvek v uzlu má z nich největší (maximální) hodnotu.

Popis algoritmu:

Proces třídění má dvě fáze – vytvoření haldy a vlastní třídění.

**Vytvoření haldy** V první fázi vytvoříme haldu. Halda je vyvážený binární strom. V něm jsou všechny úroveně zcela zaplněné až na poslední úroveň, která jediná může být neúplná.

- Uzly v úrovni k mají indexy v rozmezí  $2^k - 1 \dots 2^{(k+1)}$ .
- Následníci (existují-li) uzlu s indexem i mají indexy  $2*i + 1$  a  $2*i + 2$ , tj. následníci uzlu  $u_i$  jsou uzly  $u_{2*i+1}$  a  $u_{2*i+2}$ .

Nejdříve si z počtu tříděných prvků n vypočítáme výšku haldy, označme ji h:

$$h = \lfloor \log_2(n) \rfloor.$$

Sestavíme si vyvážený binární strom této výšky s n uzly a zaplníme ho tříděnými prvky, tj. do každého uzlu dáme jeden tříděný prvek.

Následně budeme odspodu procházet nelistové uzly a budeme ověřovat, zda splňují podmínku haldy vzhledem ke svým následníkům. U každého z těchto uzlů ověříme, zda prvek v něm uložený není menší než prvek v některém z jeho následníků. Pokud ano, uděláme výměnu.

Jestliže došlo k výměně, je nyní v příslušném následníku jiný prvek, čímž může u něho dojít k narušení podmínky haldy vzhledem k prvkům v jeho následnících. Musíme tedy obdobné srovnání (a případnou výměnu) nyní provést pro tohoto následníka. Takto budeme postupovat směrem dolů tak dlouho, dokud nedojdeme buďto k uzlu, u kterého výměna není nutná, anebo k uzlu, který už žádné následníky nemá (listu).

**Vlastní třídění** Z vlastností haldy plyne, že v jejím kořenu je největší prvek. Ten vyměníme s prvkem v posledním listu haldy. O tento list následně haldu zkrátíme a pro nový prvek v kořenu ověříme, zda

splňuje vlastnost haldy vzhledem ke svým následníkům. Pokud ne, vyměňujeme prvky mezi uzly a jejich následníky tak dlouho, až všechny splňují vlastnost haldy. Jde o stejný postup, jakým jsme ve fázi vytváření haldy prováděli výměny prvků mezi nelistovými uzly a jejich následníky. Je zřejmé, že každým krokem se halda zkrátí o jeden uzel. Až nakonec obsahuje jen jeden uzel, čímž proces třídění končí.

### 1.3.6 Merge sort

Metoda je založena na dvoucestném slučování (sléváním), což je proces, který ze dvou setříděných posloupností vytvoří jednu (delší) posloupnost.

A – je pole, ve kterém jsou za sebou dvě setříděné posloupnosti prvků.

p,q,r – jsou indexy označující, kde uvedené posloupnosti začínají a končí.

i,j – jsou průběžné indexy používané pro procházení oběma setříděnými posloupnostmi

B – je pole, ve kterém je vytvářena setříděna posloupnost.

k – je průběžný index pro vytváření výstupní posloupnosti

**Algoritmus slučování:**

1. Počáteční krok

Indexy nastavíme na začátek setříděných posloupností a na začátek pole B. i=p, j=q+1, k=0.

2. Krok slučování

Srovnáme prvky  $A_i$  a  $A_j$ . Jestliže je  $A_i \leq A_j$ , přesuneme  $A_i$  do prvku  $B_k$  a zvýšíme indexy i a k. V opačné případě (je-li  $A_i > A_j$ ), přesuneme  $A_j$  do prvku  $B_k$  a zvýšíme indexy j a k.

3. Krok 2. děláme tak dlouho, dokud nedojdeme na konec některé ze tříděných posloupností.

Jestliže je  $i > q$  (došli jsme na konec první posloupnosti), přesuneme zbývající část 2. posloupnosti do pole B za již přesunuté prvky.

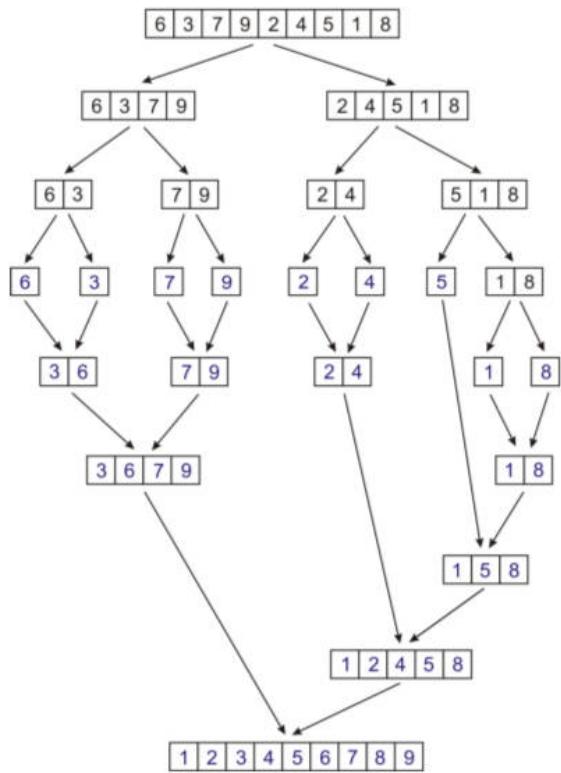
Jestliže je  $j > r$  (došli jsme na konec druhé posloupnosti), přesuneme do pole B zbývající část 1. posloupnosti.

Nakonec setříděnou posloupnost přesuneme z pole B do pole A na místo původních slučovaných posloupností.

**Algoritmus třídění slučování:**

- Každou nesetříděnou část rozdělíme na dvě části, které mají stejný počet prvků, je-li sudý počet prvků, nebo na části lišící se o jeden prvek u lichého počtu prvků.
- Je-li první část nesetříděná (obsahuje více než jeden prvek), setřídíme ji stejnou metodu – dělíme ji opět na dvě části atd.
- Je-li druhá část nesetříděná (obsahuje více než jeden prvek), setřídíme ji stejnou metodu – dělíme ji opět na dvě části atd.
- Po případném setřídění obou částí je sloučíme v jednou setříděnou část.

Je zřejmé, že algoritmu je založen na rozdělování až do setříděných částí (částí obsahující jeden prvek) a slučování již setříděných částí.



## 1.4 Další metody třídění: Counting sort, Radix Sort, Bucket sort, vnější třídění

### 1.4.1 Bucket sort

Přihrádkové třídění je třídící algoritmus, který tříděné prvky rozdělí do zvoleného počtu přihrádek. Každá přihrádka je následně tříděna samostatně buďto jiným třídícím algoritmem nebo opět přihrádkovým tříděním. Tato metoda je použitelná, je-li třídící klíč celé číslo. Rovněž se dá použít pro třídění řetězců. Mějme posloupnost celých čísel, jejichž hodnoty jsou v rozsahu:  $0 \dots m$ .

Zvolme, kolik různých hodnot bude jedné přihrádce:  $c$ .

Vypočítáme počet přihrádek  $k = \lceil \frac{m+1}{c} \rceil$

**Popis algoritmu:** Přihrádkové třídění pro přihrádky používá pole. Proces třídění má dvě fáze – vytvoření přihrádek a vlastní umístování prvků do přihrádek. Pro stanovení, kde jsou v cílovém poli jednotlivé přihrádky umístěny se používá další, pomocné pole.

$A$  – je pole, ve kterém jsou vstupní tříděná čísla.

$n$  – je počet tříděných čísel.  $B$  – je pole, ve kterém budou přihrádky.

$C$  – je pracovní pole pro zjištění, kde začínají jednotlivé přihrádky, a dále při umístování čísel do přihrádek se používá pro zjištění, na které místo v přihrádce právě umístované číslo dát. Počet prvků v poli je  $k+1$  (o 1 větší, než je počet přihrádek).

Přihrádky jsou číslovány od 0 po  $k-1$ .

### Vytvoření přihrádek

- Pole  $C$  vynulujeme.

- Pro každé číslo v poli A zjistíme, do které patří přihrádky, a zvýšíme o 1 hodnotu prvku pole C, který má index o 1 vyšší, než je zjištěné číslo přihrádky. Po ukončení tohoto kroku budeme v  $C_1$  mít počet čísel z pole A, která patří do přihrádky 0, v  $C_2$  počet čísel patřících do přihrádky 1, … v C k počet čísel patřících do přihrádky k-1.
- Nyní zjistíme začátky přihrádek v poli. Přičteme  $C_1$  k  $C_2$ ,  $C_2$  k  $C_3$ , ...,  $C_{k-2}$  k  $C_{k-1}$ . Nyní v  $C_0$  bude index, kde v poli B začíná přihrádka 0, v  $C_1$  začátek přihrádky 1, ..., v  $C_{k-1}$  začátek přihrádky k-1.

## Umisťování do přihrádek

- Pro každé číslo v poli A zjistíme číslo j přihrádky, do které číslo. V prvku pole C s indexem j zjistíme index prvního volného prvku přihrádky j v poli B. Na toto místo prvek z pole A umístíme a následně zvýšíme o 1 prvek pole C s indexem j – posuneme se v přihrádce j na další volné místo. Z uvedeného popisu je zřejmé, že pole C nám ukazuje v každém okamžiku index prvního volného prvku v jednotlivých přihrádkách (pokud daná přihrádka už není zcela zaplněna).
- Nakonec setříděnou posloupnost případně přesuneme z pole B do pole A na místo původních setříděný posloupnosti.

Pokud je počet různých hodnot v přihrádce větší než 1, pole se vhodnou metodou dotřídí – zpravidla přímou metodou třídění vkládáním.

### 1.4.2 Counting sort

Je speciální případ přihrádkového třídění, kdy každá přihrádka obsahuje čísla jen s jednou hodnotou (v předchozím popisu by to byl parametr c=1). Výsledkem je tím úplné setřídění (není zapotřebí následné dotřídění vkládáním).

### 1.4.3 Radix sort

Číslicové třídění třídí postupně číslice v jednotlivých řádech tříděných čísel přihrádkovým tříděním. Podle toho, v jakém pořadí jsou jednotlivé řády tříděny, máme dva způsoby číslicového třídění:

- **LSD (Least Significant Digit)** - číslice jsou tříděny od nejméně významného rádu po nejvíce významný (zprava-doleva).
- **MSD (Most Significant Digit)** - číslice jsou tříděny od nejvíce významného rádu po nejméně významný (zleva-doprava).

Z obou metod je třídění LSD poněkud efektivnější (rychlější) a navíc jeho implementace je výrazně jednodušší.

**Číslicové třídění LSD** Číslicovým tříděním můžeme třídit čísla, která jsou v libovolné číselné soustavě. Při realizaci číslicového třídění vyjdeme z číselné soustavy, ve které jsou uložena celá čísla v paměti počítače, což je dvojková soustava. Třídění každé číslice by mělo dvě přihrádky. Počet opakování by závisel na délce čísla. Pokud bychom vzali nejběžnější 32-bitová čísla, znamenalo by to, že by se přihrádkové třídění muselo udělat 32x. Při každém přihrádkovém třídění se všechna tříděná čísla přesunou z jednoho pole do druhého. To by znamenalo celkem 32 přesunů všech tříděných čísel, což je mnoho a třídění by se tolika přesuny zpomalilo. Bylo by efektivnější použít pro třídění číselnou soustavu s vyšším základem takovou, na kterou se binární číslo snadno převede. To jsou soustavy, jejichž základ je mocnina čísla 2. Čím je menší počet přihrádek, tím menší čas je zapotřebí na jejich přípravu. Čím je menší počet přihrádkových třídění, tím menší čas je zapotřebí pro jejich provedení. Zvolíme kompromis mezi těmito dvěma vzájemně si odpovídajícími požadavky – základ soustavy zvolíme 256.

#### 1.4.4 Vnější třídění

Na rozdíl od vnitřního třídění, kdy všechny tříděné prvky jsou ve vnitřní paměti, při vnějším třídění jsou tříděné prvky uloženy v souborech na vnější paměti (pevném disku). Třídění probíhá tím způsobem, že tříděné prvky jsou ze souborů po určitých částech přenášeny do vnitřní paměti, zde jsou zatřídovány a pak opět ukládány do souborů. Protože operace čtení ze souborů a zápis do souborů jsou výrazně pomalejší, než když přesuny prvků probíhají výlučně ve vnitřní paměti, je vnější třídění obecně pomalejší než vnitřní třídění a použijeme ho jen v případě, kdy vnitřní třídění nelze použít, protože tříděných prvků je tak velký objem, že se všechny do paměti najednou nevejdou.

**Třídění se stejným počtem vstupních a výstupních souborů** je nejjednodušší varianta vnějšího třídění. Označme si celkový počet souborů použitých při třídění  $r$ . Protože vstupních i výstupních souborů je stejný počet, musí to být sudé číslo. Minimální počet vstupních souborů, aby mohla probíhat operace zatřídování, je 2, čímž celkový nejmenší možný počet souborů je 4. Počet vstupních souborů je  $v = \frac{r}{2}$ . Vnější třídění je založeno na operaci zatřídování. Při ní je z více setříděných sekvencí vytvářena jedna delší setříděná sekvence. Setříděným sekvencím budeme říkat běhy. Na začátku třídění běhy mají délku 1, jsou tvořeny jedním prvkem. Opakováním procesem zatřídování (slučováním) jsou z nich vytvářeny stále delší a delší běhy, až se dosáhne stavu, že je vytvořen běh tak dlouhý, že obsahuje všechny tříděné prvky. Tím třídění končí.

Popis algoritmu:

##### 1. fáze rozdělování

První fáze je poměrně rychlá. Jejím cílem je pravidelně rozdělit tříděné prvky do v souborů, abychom mohli zahájit vlastní třídící proces. Vezmeme ze stanoveného počtu  $r$  souborů polovinu ( $v$ ) a otevřeme je pro výstup (zápis). Do nich pravidelně rozdělujeme tříděné prvky. První prvek zapíšeme do prvního souboru, druhý prvek do druhého souboru, ..., až  $v$ -tý prvek do  $v$ -tého souboru, následující prvek opět do prvního souboru atd. Po dokončení rozdělování soubory výstupní soubory uzavřeme a následně je otevřeme jako vstupní (pro čtení) a zbyvající polovinu souborů otevřeme jako výstupní. Délku běhů nastavíme na 1.

##### 2. fáze zatřídování

Ze všech v vstupních souborů načteme jejich první běh, zatříděním z něho vytvoříme jeden delší výstupní běh a ten uložíme do prvního výstupního souboru. Pak ze vstupních souborů načteme druhý běh, z něho opět vytvoříme výstupní běh a uložíme ho do druhého výstupního souboru. A tak pokračujeme až  $v$ -tý vytvořený výstupní běh uložíme do  $v$ -tého souboru. Následující ( $v+1$ )-tý vytvořený výstupní běh uložíme opět do prvního souboru, ( $v+2$ )-tý vytvořený výstupní běh uložíme opět do druhého souboru atd. Tj. vytvářené výstupní běhy pravidelně rozdělujeme do v výstupních souborů.

V okamžiku, kdy všechny běhy ze vstupních souborů jsou přečteny, výstupní soubory uzavřeme a otevřeme je jako vstupní a předchozí vstupní soubory uzavřeme a otevřeme je nyní jako výstupní. A znova opakujeme proces zatřídování. Tento postup probíhá tak dlouho, dokud vytvořený výstupní běh není tak dlouhý, že obsahuje všechny tříděné prvky.

**Algoritmus zatřídování** K zatřídování potřebujeme kolik proměnných (míst ve vnitřní paměti), kolik je vstupních souborů, tedy  $v$ . Tyto proměnné si označme  $x_1, x_2, \dots, x_v$ .

1. Do každé z proměnných  $x_1, x_2, \dots, x_v$  načteme první prvek z v vstupních běhů.
2. Vybereme nejmenší z prvků v  $x_1, x_2, \dots, x_v$ . Nechť tento prvek je třeba v proměnné  $x_i$ . Prvek zapíšeme do výstupního souboru, do kterého je právě ukládán výstupní běh, a do proměnné  $x_i$  načteme další prvek z  $i$ -tého vstupního běhu (zbývá-li v něm ještě nějaký). Tento proces pokračuje tak dlouho, dokud všechny prvky z proměnných nejsou zapsány do výstupního běhu a dokud všechny prvky ze současných v vstupních běhů nejsou vyčerpány.

**Vnější třídění s využitím vnitřního třídění** Předpokládejme, že bychom vytvářeli běhy délky  $m$ . Pro poslední k-tý běh  $m * v^k = n$ , odtud  $k = \lceil \log_v(n) - \log_v(m) \rceil$ . Z toho je zřejmé, že čím bude délka běhů  $m$  vytvářených vnitřním tříděním ve fázi rozdělování větší, tím více ušetříme průchodů zatřídování, což se dalo i očekávat.

I když pro vnitřní třídění máme nejrychlejší metodu Quicksort, vnější třídění používá určitou variantu třídění haldou, která umožňuje při dané používané velikosti paměti pro vnitřní třídění běhů vytvářet běhy typicky delší, než by poskytla metoda Quicksort.

Odlišnosti haldy při vnějším třídění oproti haldě používané při vnitřním třídění:

- Prvky jsou v haldě uspořádány obráceně – v kořenu je minimální (nejmenší prvek).
- Halda je rozdělena na dvě části – dolní část a horní část. V horní části jsou prvky právě vytvářeného běhu. V dolní části jsou prvky, které nelze zařadit do současného běhu a budou součástí až následujícího běhu.

## Popis algoritmu

### 1. Počáteční vytvoření haldy

Předpokládejme, že halda má  $m$  uzlů. Nejprve načteme prvky ze vstupního souboru do listových uzlů haldy  $u_{m-1}, u_{m-2} \dots u_{\frac{m}{2}}$ . Následně bereme nelistové uzly haldy v pořadí  $u_{\frac{m}{2}-1}, u_{\frac{m}{2}-2} \dots u_1, u_0$ . Pro každý nelistový uzel  $u_i$  načteme prvek ze vstupního souboru. Zjistíme jeho velikost a před jeho zařazením do haldy uděláme nejprve případné přesuny prvků, které jsou v uzlech pod uzlem  $u_i$ , aby po zařazení načteného prvku do haldy byla splněna podmínka uspořádání prvků v haldě. Po zaplnění haldy, celou haldu označíme jako horní část haldy (dolní část haldy je prázdná).

### 2. Vytvoření běhu

Přepokládejme, že aktuální horní část haldy je tvořena uzly  $u_0, u_1, \dots u_{s-1}, u_s$  a dolní část haldy je tvořena následujícími uzly  $u_{s+1}, u_{s+2}, \dots u_{m-2}, u_{m-1}$ . Z kořene haldy (uzlu  $u_0$ ) odebereme v něm uložený prvek  $x$  a zapíšeme ho do výstupního běhu. Následně přečteme ze vstupního souboru další prvek  $y$  a zařadíme ho do haldy. Zde mohou nastat dva případy:

- Je-li  $x \leq y$ , můžeme prvek  $y$  začlenit do právě vytvářeného běhu. Ověříme hodnotu prvků  $y$  a uděláme nejprve případné přesuny prvků v uzlech pod kořenem, aby po vložení prvků  $y$  do právě volného uzlu byla splněna podmínka uspořádání haldy.
- Je-li  $y < x$ , nelze prvek  $y$  již začlenit do právě vytvářeného běhu. Horní část haldy zkrátíme o jeden uzel (její poslední uzel  $u_s$  nyní bude patřit do dolní části haldy), odebereme prvek  $w$  z uzlu  $u_s$  a prvek  $w$  vložíme do horní části haldy (po případných přesunech prvků v uzlech pod volným kořenem). Tím uzel  $u_s$  patřící nyní do dolní části haldy uvolníme. Načtený prvek  $y$  zařadíme do dolní části haldy (opět po případných přesunech prvků v uzlech pod volným uzlem  $u_s$ ).

Je zřejmé, že tímto procesem se při vytváření výstupního běhu horní část haldy postupně zkracuje, až dojde k jejímu úplnému vyčerpání. V tomto okamžiku ukončíme vytváření aktuálního běhu, dolní část haldy (obsahující nyní celou haldu) označíme jako horní část, čímž dolní část bude opět prázdná. A začneme vytvářet další běh.

### 3. Vytvoření posledního běhu

V okamžiku, kdy už jsou načteny všechny prvky ze vstupního souboru, jen dokončíme vytváření posledního běhu odebíráním zbývajících prvků z horní části haldy.

**Polyfázové třídění** Je zřejmé, že počet potřebných průchodů zatřídování mimo jiné závisí na základu logaritmumu  $v$ , tj. počtu vstupních souborů. Čím větší tento bude, tím méně průchodů bude potřeba. Můžeme sice počet vstupních souborů zvýšit zvolením většího počtu celkově používaných souborů, ale na druhé straně příliš mnoho používaných souborů má negativní vliv na výkonnost systému. Spíše se nabízí myšlenka celkový počet  $r$  používaných souborů využít efektivněji a to tak, že místo rozdelení  $\frac{r}{2}$  vstupních souborů  $+ \frac{r}{2}$  výstupních souborů použijeme rozdelení  $r-1$  vstupních souborů  $+ 1$  výstupní soubor.

Při tomto rozdelení musí mít vstupní soubory rozdílné počty běhů, aby jeden z nich se vyčerpal dříve než ostatní. V tomto okamžiku se vyčerpaný vstupní soubor uzavře a stane se nyní novým výstupním souborem a dosavadní výstupní soubor se rovněž uzavře a přidá se ke vstupním souborům. Aby tento princip fungoval po celou dobu třídění, musí ve fázi rozdělování být vstupní běhy rozděleny do r-1 vytvářených souborů ve specifických počtech.

## 1.5 Složitosti třídících algoritmů

### 1.5.1 Insert sort

Vezměme nejprve operaci srovnání. Nechť počet prvků v setříděné části je k. Abychom zjistili místo vložení, uděláme 1 až k srovnání. Jedno srovnání v případě, kdy se prvek vkládá hned na konec setříděné části, k srovnání v případě, kdy prvek patří na první nebo druhou pozici v setříděné části. Odtud pro jeden krok dostáváme průměrný počet srovnání  $\frac{1+k}{2}$  a maximální počet srovnání k. Vezmeme-li v úvahu, že délky setříděných částí v jednotlivých krocích jsou k = 1, 2, ..., n-1, dostaneme celkové počty srovnání průměrný  $\frac{2+3+\dots+n}{2} = \frac{1+2+\dots+n-1}{2} = \frac{\frac{n(n-1)}{2}-1}{2} = \frac{n^2+n-2}{4}$  a maximální  $1+2+\dots+(n-1) = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ . Z toho plyne, že operace srovnání má kvadratickou složitost bez ohledu na to, zda uvažujeme průměrný nebo maximální počet srovnání.

Uvažujme operaci přesunu prvku o jednu pozici dozadu. Těch v jednom kroku proběhne 0 až k podle toho, na které místo je prvek vkládán. Odtud pro jeden krok dostáváme průměrný počet přesunů  $\frac{0+k}{2}$  a maximální počet přesunů k. Pro celé třídění průměrný počet  $\frac{1+2+\dots+(n-1)}{2} = \frac{\frac{(n-1)n}{2}}{2} = \frac{n^2-n}{2}$  a maximální počet  $1+2+\dots+(n-1) = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ . **Výsledná časová složitost je  $\theta(n^2)$ .**

### 1.5.2 Select sort

Vezměme nejprve operaci výběru. Nechť počet prvků v nesetříděné části je k. Abychom našli její nejmenší prvek, potřebujeme k tomu k-1 srovnání. Neboť počínaje druhým prvkem v nesetříděné části postupně srovnáváme všechny její prvky až po poslední prvek, vždy s právě zapamatovaným prvkem. Délky nesetříděných částí jsou v jednotlivých krocích n, n-1, n-2, ..., 2. Odtud celkový počet srovnání  $(n-1)+(n-2)+\dots+1 = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ . Nyní uvažujme operaci výměny vybraného, nejmenšího prvku s prvním prvkem nesetříděné části. Ta v každém kroku proběhne jen jednou. Třídících kroků je n-1 a odtud dostáváme celkový počet výměn n - 1. Největší časovou složitost má operace výběru. **Výsledná časová složitost je  $\theta(n^2)$ .**

### 1.5.3 Bubble sort

Vezměme opět nejprve operaci srovnání. V prvním průchodu se prochází všech n prvků, což reprezentuje srovnání n-1 sousedních dvojic. V druhém průchodu se prochází n-1 prvků, tedy n-2 srovnávaných dvojic. V posledním průchodu má procházená část dva prvky, tedy se provede jen jedno srovnání. Odtud celkový počet srovnání  $(n-1)+(n-2)+\dots+1 = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ . Nyní uvažujme operaci výměn. Výměn se provede maximálně kolik, kolik je srovnání. Minimální možný počet výměn je žádná výměna (prvky jsou na začátku uspořádány tak, že jsou setříděné). Dostáváme počty přesunů průměrný  $\frac{3(n^2-n)}{4}$  a maximální  $\frac{3(n^2-n)}{2}$ . **Výsledná časová složitost je  $\theta(n^2)$ .**

### 1.5.4 Quick sort

Složitost závisí na tom, v jakém poměru se tříděná část rozdělí na nové části L a R. Nejhorší případ nastane, když jedna z těchto částí obsahuje jen jeden prvek a ta druhá zbývající prvky. Pak by následující třídící krok proběhl vždy jen pro tu větší část a ta by měla vždy o jeden prvek méně, než tomu bylo v

předchozím kroku. Počet srovnání je maximálně o 1 větší, než je počet prvků v procházené části. Tento největší počet srovnání nastane v případě, kdy procházená část už je setříděná.

Pro počet výměn nám stačí skutečnost, že jich je nejvýše tolik, kolik je srovnání. Celkově je z toho vidět, že složitost operace srovnání a tím i celé metody je v tomto nejhorším případě kvadratická.

Naopak optimální případ nastane, když procházená část se vždy rozdělí na dvě stejné části. Počet třídících chodů logaritmicky závisí na počtu tříděných prvků. Zbývá stanovit složitost operace srovnání v jednom třídícím průchodu. Ta je závislá na tom, na kolik částí je tříděné pole rozděleno. Máme-li v daném třídícím chodu k části, pak v každé z nich bude  $\frac{n}{k}$  prvků. Počet srovnání v každé části je nejvýše o 1 větší než je počet prvků v ní. V prvním třídícím průchodu chodu je počet částí 1, v posledním, kdy části mají délku 2, je počet  $\frac{n}{2}$ . Dosazením zjistíme, že počet srovnání v jednom třídícím průchodu se pohybuje od  $n+1$  v prvním třídícím průchodu až po  $\frac{3n}{2}$  v posledním průchodu. Tedy ve všech průchodech je složitost lineární. **Výsledná časová složitost je  $\theta(n \times \ln(n))$ .**

### 1.5.5 Heap sort

Vezměme nejprve složitost vytvoření haldy. Při něm procházíme jednotlivé jeho nelistové uzly, srovnáváme prvky v něm uložené s následníky a případně provádíme přesuny. Srovnání je v každém uzlu nutné provést dvě - se dvěma následníky (má-li uzel oba následníky). Následně proběhne případný přesun. Vezměme ten nejhorší případ, kdy srovnání a přesuny proběhnou od nejhornějšího uzlu (kořene) až po ten nejspodnější nelistový uzel. Jde celkem o  $3 * h$  operací (3 operace v každém uzlu: 2 srovnání + 1 přesun), kde  $h$  je výška stromu. Výška binárního vyváženého stromu logaritmicky závisí na počtu uzlů (prvků). Tedy pro každý probíraný nelistový jeden uzel má v nejhorším případě (a i v průměrném případě) logaritmickou složitost. Vynásobením počtem uvažovaných nelistových uzlů dostaváme složitost  $\theta(n \times \ln(n))$ .

Složitost fáze vlastního třízení je hlavně určena složitostí operací srovnání a případných výměn, které následují po vzájemně výměně prvků mezi kořenem a posledním listem ve stávající haldě a zkrácením haldy o tento list. Opět to vyžaduje 2 srovnání a 1 přesun na každý nelistový uzel (nebo 1 srovnání, pokud uzel má jen jednoho následníka). Tedy maximálně  $3 * h$  operací (tentototo počet typicky klesá, jak se halda postupně zkraje). Opět logaritmická složitost pro každý nový prvek v kořenu haldy. Vynásobíme-li ji celkovým počtem prvků, dostaváme zase složitost  $\theta(n \times \ln(n))$ . **Výsledná časová složitost je  $\theta(n \times \ln(n))$ .**

### 1.5.6 Merge sort

Nejprve odvodíme časovou složitost slučování. Nechť počet prvků v první slučované posloupnosti je  $m_1$  a v druhé  $m_2$ . V průběhu slučování se dělají srovnání, jejich počet je  $\min(m_1, m_2) \dots m_1 + m_2 - 1$ . Dále probíhají přesuny. Každý prvek se nejprve přesune do pole B a na konci se vrátí do pole A. Celkem je přesunů  $2 * (m_1 + m_2)$ . Složitost slučování je lineární  $\theta(m_1 + m_2)$ .

Nechť n je počet tříděných prvků. Postupných rozdělování se rozdělí na n jednoprvkových částí. Ty se následně slučují. Pro jednoduchost předpokládejme, že obě slučované části mají stejnou délku. Počet operací (srovnání + přesunů) je celkově  $k * 3n$ . Zřejmě mezi číslem kroku i a délkom slučovaných částí v kroku i platí vztah délka slučované části je  $2^{i-1}$ . Pro poslední krok  $\frac{n}{2} = 2^{k-1}$ . Odtud  $k = \log_2(n)$ . **Výsledná časová složitost je  $\theta(n \times \ln(n))$ .**

### 1.5.7 Bucket sort

Nejprve odvodíme časovou složitost operací s tříděnými čísly. V průběhu třídění je pro každé číslo 2x počítáno, do které patří příhrádky, a 1x je číslo přesunuto do jiného pole. Počet operací pro každé číslo je konstantní a složitost těchto operací je  $\theta(n)$ .

Dále zde máme samostatné operace s příhrádkami. Příhrádky jsou na začátku vynulovány a dále je pro každou příhrádku počítán její začátek (přičtením začátku předchozí příhrádky k počtu prvků dané příhrádky). Složitost těchto operací je  $\theta(k)$ .

**Výsledná časová složitost je  $\theta(n + k)$ .**

### 1.5.8 Radix sort

Jde o příhrádkové třídění s konstantním počtem opakování. **Výsledná časová složitost je  $\theta(n + zkladselnsoustavy)$ .**

### 1.5.9 Vnější třídění

**Třídění se stejným počtem vstupních a výstupních souborů** Vezměme nejprve, kolik operací zahrnuje operace zatřídování pro jeden prvek:

- jednu operaci čtení, kdy je prvek načten ze vstupního běhu do paměti.
- $v-1$  operací srovnání, kdy je tento prvek vybrán jako nejmenší (případně těchto operací může být méně, když už některé vstupní běhy jsou vyčerpány a příslušné proměnné těchto běhů už žádný prvek neobsahují), případně žádnou operaci, když už zbyla část je jednoho běhu a tato je kopírována do výstupního běhu.
- jednu operaci zápisu, kdy je prvek zapsán do souboru, do něhož je právě výstupní běh ukládán.

Uvážíme-li, že máme  $n$  tříděných prvků, pak pro jeden průchod zatřídování dostáváme  $n$  operaci čtení + nejvýše  $n * (v - 1)$  operací srovnání +  $n$  operací zápisu. Protože v je zvolená konstanta, jejíž hodnota je velmi malá vzhledem k počtu tříděných prvků  $n$ , je složitost jednoho průchodu zatřídování lineární. Zbývá stanovit, kolik průchodů zatřídování proběhne. V prvním průchodu zatřídování, kdy délka vstupních běhů je 1, vzniknou výstupní běhy délky  $v$ . V druhém průchodu zatřídování, kdy délka vstupních běhů je  $v$ , vzniknou výstupní běhy délky  $v * v$ . Atd. Třídění končí, když výstupní běh obsahuje všechny tříděných prvků. Tedy pro číslo  $k$  posledního průchodu dostáváme vztah  $v^k = n$ . A odtud použitím logaritmu  $k = \lceil \log_v(n) \rceil$ .

Počet průchodů logaritmicky závisí na počtu tříděných prvků. Když to vynásobíme lineární složitostí jednoho průchodu, dostáváme, že **složitost** popsané metody vnějšího třídění je  $\theta(n \times \log(n))$ .

## 1.6 Pořádkové statistiky

K-tá statistika pořadí  $z$   $n$  prvků je  $k$ -tý nejmenší prvek.

Minimum  $z$   $n$  prvků je 1. statistika pořadí. Maximum  $z$   $n$  prvků je  $n$ -tá statistika pořadí.

Medián je prvek uprostřed. Je-li počet prvků  $n$  liché číslo, medián je statistika pořadí stupně  $\frac{n+1}{2}$ . Je-li počet prvků sudé číslo máme dva mediány – dolní medián a horní medián. Dolní medián je statistika pořadí stupně  $\frac{n}{2}$  a horní medián je statistika pořadí stupně  $\frac{n}{2} + 1$ . Pokud neuvedeme v případě lichého počtu prvků, zda je o dolní nebo horní medián, budeme mít na mysli dolní medián.

**Úloha výběru** Je dána množina  $A$  obsahující  $n$  vzájemně odlišných prvků (nejsou v ní žádné dva prvky stejné).

Úloha výběru: Pro zadané  $k$  ( $1 \leq k \leq n$ ) najít v množině  $A$   $k$ -tou statistiku pořadí. Nebo-li najít v množině  $A$  prvek, který je větší než právě  $k-1$  prvků z množiny  $A$ .

Jednoduché řešení: Množinu  $A$  setřídíme a hledaná  $k$ -tá statistika pořadí je  $k$ -tý v prvek v setříděné posloupnosti. Tímto způsobem se dá tato úloha vyřešit se složitostí  $\theta(n \times \ln(n))$ .

**Řešení úlohy výběru v lineárním čase** Úlohu lze řešit obdobnou metodou, jakou používá třídění Quicksort. Budeme procházenou částí pole s prvky, ve kterých hledáme statistiku pořadí daného stupně dělit na dvě části. Po rozdělení vybereme pro další dělení tu část, která obsahuje prvek s požadovaným stupněm pořadí.

**Složitost metody** závisí na tom, v jakém poměru se tříděná část rozdělí na nové části L a R. Nejhorší případ nastane, když jedna z těchto částí obsahuje jen jeden prvek a ta druhá zbývající prvky. Pak by následující třídící krok proběhl vždy jen pro tu větší část a ta by měla vždy o jeden prvek méně, než tomu bylo v předchozím kroku. Tedy třídící kroky by proběhly postupně pro části s počty prvků n, n-1, ..., 2. Tím bychom stejně jako u třídění Quicksort dostali kvadratickou složitost.

Naopak optimální případ nastane, když procházená část se vždy rozdělí na dvě stejné části (při lichém počtu prvků na dvě části lišící se o jeden prvek). Možnost, že dělení také může být na dvě části a střední prvek, pro jednoduchost opomíneme. Počet třídících chodů logaritmicky závisí na počtu tříděných prvků, jak bylo odvozeno u metody Quicksort. Počet operací srovnání v jednom průchodu dělení na dvě části je podle úvahy uvedené u třídění Quicksort nejvyšše o 1 více, než je délka částí. Sečtením přes všechny průchody dostaneme celkový počet operací srovnání. **Výsledná složitost je typicky  $\theta(n)$ .**

## 2

Grafy, stromy, základní pojmy a tvrzení. Vyhledávání a rozdelení vyhledávacích algoritmů. Vyhledávání v lineárních datových strukturách. Binární vyhledávací stromy, průchod a vyhledávání. Red-black stromy, AVL-stromy, B-stromy a jejich struktura, operace vyhledání, vložení a zrušení prvku. Hashování: hashovací funkce, organizace tabulek a způsoby řešení konfliktů.

### 2.1 Grafy

Grafický způsob vyjádření vztahů mezi nějakými objekty. Objekty jsou v grafu reprezentovány uzly. Vztahy jsou v grafu reprezentovány hranami. Způsob kreslení volíme především tak, aby graf byl přehledný. Hrana vždy začíná a končí v nějakém uzlu. Většinou jsou koncové uzly hrany různé, ale může to být i stejný uzel, pak takové hrany říkáme **smyčka**. Dva uzly mohou být spojeny více hranami – takovým hranám říkáme **násobné**. Graf, ve kterém mezi některými uzly je více hran, nazýváme **multigraf**.

Hrany v grafu mohou být:

- **Neorientované** reprezentují symetrické vztahy mezi uzly. Například v situaci, kdy Petr a Jana jsou bratr a sestra a hrana nám bude v grafu vyjadřovat tento vztah, že jsou sourozenci, tato hrana bude neorientovaná.
- **Orientované** reprezentují jednosměrné, nesymetrické vztahy mezi uzly. Orientace hrany je vyznačena šipkou na jednom jejím konci. Například je-li Eva matkou Petra a Jany a hrany nám budou vyjadřovat vztah, že Eva je jejich rodičem, bude tyto hrany orientované.

Podle typu hran dělíme grafy na:

- **Neorientované** – všechny jejich hrany jsou neorientované
- **Orientované** – všechny jejich hrany jsou orientované
- **Smíšené** – obsahují neorientované i orientované hrany

**Definice 1.** *Graf je dvojice  $G = (U, H)$ , kde*

$$U \text{ je množina uzlů } U = \{u_1, u_2, \dots, u_m\}$$

$$H \text{ je množina hran } H = \{h_1, h_2, \dots, h_n\}$$

U neorientovaného grafu jsou hrany určeny neuspořádanými dvojicemi svých koncových uzlů  $h_i = \{u_j, u_k\}$ , kde  $u_j, u_k \in U$

U orientovaného grafu jsou hrany určeny uspořádanými dvojicemi svých koncových uzlů  $h_i = \langle u_j, u_k \rangle$ , kde  $u_j, u_k \in U$

Počet hran se zančí  $|H|$ . Počet uzlů  $|U|$ .

**Definice 2.** *Stupeň uzlu je počet hran, které jsou s tímto uzel spojeny - mají tento uzel jako koncový. Pro skutečnost, že hrana má daný uzel jako koncový, používáme termín, že hrana s tímto uzlem **inciduje**.*

U orientovaného grafu navíc rozeznáváme výstupní stupeň uzlu, což je počet hran, které z něho vychází, označujeme ho  $d^-(u)$ , a vstupní stupeň uzlu, což je počet hran, které do něho vchází, označuje ho  $d^+(u)$ . Zřejmě pro stupeň uzlu v orientovaném grafu platí  $d(u) = d^-(u) + d^+(u)$ .

**Definice 3.** V teorii grafů se používají termíny:

- Pro uzel, jehož stupeň je nulový, používáme název diskrétní uzel.
- Graf, jehož všechny uzly jsou diskrétní, nazveme diskrétní graf.
- Dva uzly, jež jsou spojeny hranou, nazveme sousedními uzly.
- Pro graf, jehož všechny uzly jsou sousední (má při daném počtu uzlů maximální počet hran), se používá označení úplný graf.

**Definice 4.** Nechť je dán graf  $G = (U, H)$ . Pak graf  $G_1 = (U_1, H_1)$  takový, že  $U_1 \subset U$  a  $H_1 \subset H$ , nazveme podgrafem grafu  $G$ .

### 2.1.1 Reprezentace grafu v programech

Graf si můžeme v programu reprezentovat různými způsoby. Můžeme si ho například uložit jako matici sousednosti. Ovšem u rozsáhlejších grafů s větším počtem uzlů je tato matice značně velká a navíc její použití v algoritmech je poměrně neefektivní, neboť v ní musíme pracně hledat.

### 2.1.2 Reprezentace pomocí polí

Struktura grafu je uložena ve dvou polích. První pole má stejný počet prvků, jako je počet uzlů v grafu. Každému uzlu odpovídá jeden prvek pole. V něm je uložena hodnota indexu, od kterého v druhém poli začíná seznam uzlů, jež jsou sousedé tohoto uzlu.

### 2.1.3 Reprezentace dynamickou datovou strukturou

Další možnost je uzel grafu reprezentovat jako strukturovaný datový typ. Každý uzel obsahuje pole (seznam) ukazatelů na sousední uzly.

### 2.1.4 Izomorfismus grafů

**Definice 5.** Neorientované grafy  $G_1 = (U_1, H_1)$  a  $G_2 = (U_2, H_2)$  jsou izomorfní právě když existuje bijektivní zobrazení  $h : U_1 \rightarrow U_2$ , pro které platí  $\{u, v\} \in H_1$  právě když  $\{h(u), h(v)\} \in H_2$ .

**Definice 6.** Orientované grafy  $G_1 = (U_1, H_1)$  a  $G_2 = (U_2, H_2)$  jsou izomorfní právě když existuje bijektivní zobrazení  $h : U_1 \rightarrow U_2$ , pro které platí  $\langle u, v \rangle \in H_1$  právě když  $\langle h(u), h(v) \rangle \in H_2$ .

[http://upload.wikimedia.org/wikipedia/commons/9/9a/Graph\\_isomorphism\\_a.svg](http://upload.wikimedia.org/wikipedia/commons/9/9a/Graph_isomorphism_a.svg)

[http://upload.wikimedia.org/wikipedia/commons/8/84/Graph\\_isomorphism\\_b.svg](http://upload.wikimedia.org/wikipedia/commons/8/84/Graph_isomorphism_b.svg)

Je zřejmé, že izomorfní grafy musí mít stejný počet uzlů, stejný počet hran, stejný počet uzlů daného stupně atd.

### 2.1.5 Souvislost grafu

**Definice 7.** Nechť je dán graf  $G = (U, H)$  a dva jeho uzly  $u$  a  $v$ . Sledem mezi uzly  $u$  a  $v$  nazveme posloupnost uzlů a hran  $u, h_{i1}, u_{i1}, h_{i2}, u_{i2}, \dots, u_{ik-1}, h_{ik}, v$  pro kterou platí  $h_{ir} = \{u_{ir-1}, u_{ir}\}$  pro  $r = 1, \dots, k$

Tedy sled je na sebe navazující posloupnost hran, kdy vždy dvě za sebou následující hrany ve sledu mají společný koncový uzel, který je ve sledu uveden mezi nimi.

Je-li  $u=v$  (počáteční a koncový uzel sledu je stejný), jde o **uzavřený sled**.

**Tah** mezi uzly  $u$  a  $v$  je sled mezi těmito dvěma uzly, ve kterém se žádná hrana nevyskytuje vícekrát.

**Cesta** mezi uzly  $u$  a  $v$  je tah mezi těmito dvěma uzly, ve kterém se žádný jeho vnitřní uzel nevyskytuje vícekrát.

Uzavřená cesta (je-li  $u=v$ ) je označována jako **kružnice grafu**.

**Orientovaný sled** - všechny jeho hrany mají stejnou orientaci – od počátečního uzlu  $u$  ke koncovému uzlu  $v$ .

Dále jsou to pojmy orientovaný tah, orientovaná cesta a cyklus. Cyklus je označení pro orientovanou kružnicí.

**Definice 8.** *Graf, mezi jehož každými dvěma uzly existuje cesta, nazveme souvislým grafem.*

**Definice 9.** *Komponentou grafu nazveme každý jeho maximální souvislý podgraf. Přitom souvislý podgraf daného grafu považujeme za maximální, jestliže ho už nelze zvětšit přidáním dalších hran, či uzlů daného výchozího grafu tak, aby podgraf byl stále souvislý. Tedy není vlastním podgrafem jiného souvislého podgrafa.*

**Věta 1.** *Nechť souvislý graf s  $m$  uzly má  $p$  komponent. Pak má nejméně  $m-p$  hran.*

U orientovaného grafu rozeznáváme dva stupně souvislosti: souvislý a silně souvislý graf.

**Definice 10.** *Orientovaný graf je souvislý, jestliže mezi každými jeho dvěma uzly  $u$  a  $v$  existuje orientovaná cesta buďto z uzlu  $u$  do uzlu  $v$  nebo z uzlu  $v$  do uzlu  $u$ . Orientovaný graf je silně souvislý, jestliže mezi každými jeho dvěma uzly  $u$  a  $v$  existuje orientovaná cesta z uzlu  $u$  do uzlu  $v$  a rovněž opačná cesta z uzlu  $v$  do uzlu  $u$ .*

## 2.1.6 Minimální kostry grafu

Kostra je faktor grafu, který má stejný počet komponent a neobsahuje kružnice. Připomeňme, že faktor grafu je jeho podgraf, který má stejnou množinu uzlů. U hranově ohodnocených grafů se v praxi vyskytuje úloha nalezení minimální kostry.

## 2.1.7 Vzdálenosti v grafu

Mějme souvislý graf  $G = (U, H)$ , jehož hranы jsou ohodnoceny nezápornými reálnými čísly. Čísla, jimž jsou jednotlivé hranы ohodnoceny, budeme nazývat délkami těchto hran. Délkou cesty budeme nazývat součet délek všech hran obsažených na této cestě.

**Definice 11.** *Vzdáleností  $d(u,v)$  dvou uzlů  $u$  a  $v$  souvislého grafu  $G$  nazveme délku nejkratší z cest mezi oběma uzly  $u$  a  $v$ .*

**Věta 2.** *Pro libovolné tři uzly  $u$ ,  $v$  a  $w$  platí:*

1.  $d(u,v) \geq 0$ ,  $\text{prv} d(u,v) = 0 \text{ prvk} dy u = v$
2.  $d(u,v) = d(v,u)$
3.  $d(u,v) \leq d(u,w) + d(w,v)$

## 2.2 Stromy

Strom je velmi využívaná datová struktura.

### 2.2.1 Prvky stromu

- **Kořen stromu** Nejvyšší uzel stromu, jediný uzel bez rodiče, v každém stromu právě jeden kořen.
- **Vnitřní uzly** Uzel, který není koncový.
- **Koncový uzel = list** Prvek, který nemá žádného potomka.

Má-li strom pouze jeden prvek, je to kořen a list zároveň. Uzly jsou navzájem spojeny hranami. Neexistuje osamocený uzel, ke kterému by nevedla žádná hrana (s výjimkou stromu s pouze jedním uzlem). Pokud jsou hrany orientované, nazývají se uzly připojené k jednomu uzlu jako **potomci uzlu**, nadřazený uzel je potom **rodičovský uzel**. Uzel může mít pouze jednoho rodiče, ale více potomků. Počet všech potomků nějakého uzlu se nazývá **stupeň uzlu**.

**Definice 12.** *Listy stromu jsou uzly se stupněm 1. Uzly, které mají větší stupeň než je 1, jsou vnitřní uzly stromu.*

**Podstrom** je část stromové datové struktury tvořené jedním uzlem (kořenem podstromu) a všemi jeho potomky. Každý uzel ve stromu může tvořit kořen podstromu.

**Další pojmy:**

- **Cesta** k nějakému uzlu je definována jako posloupnost všech uzlů od kořene k uzlu.
- **Délka cesty** je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti-1.
- **Hloubka uzlu** je definována jako délka cesty od kořene k uzlu. Prvky se stejnou hloubkou jsou na téže úrovni.
- **Výška stromu** je rovna hodnotě maximální hloubky uzlu.
- **Šířka stromu** je počet uzlů na stejně úrovni.

Strom má **nejmenší výšku** právě tehdy, když na všech úrovních (s možnou výjimkou té poslední) má tato struktura plný počet uzlů. Úroveň všech listů je stejná nebo se liší maximálně o 1.  
Uspořádané vs. neuspořádané stromy.

### 2.2.2 procházení stromu do hloubky

začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větví (tj. v žádném podstromu) již není následník.

Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:

- Preorder: proved akci, projdi levý podstrom, projdi pravý podstrom.
- Inorder: projdi levý podstrom, proved akci, projdi pravý podstrom.
- Postorder: projdi levý podstrom, projdi pravý podstrom, proved akci.

V teorii grafů strom = acyklický graf.

### 2.2.3 Operace nad stromy

- Počet všech prvků
- Hledání prvků
- Přidání nového prvku na určitou pozici ve stromu
- Smazání prvku
- Vyjmutí celé části stromu = prořezávání
- Přidání celé části do stromu = roubování
- Hledání kořene pro každý uzel
- Výška (hloubka) stromu

## 2.3 Vyhledávání a rozdelení vyhledávacích algoritmů

Vyhledávání je další velmi důležitou a často se vyskytující úlohou. Při ní máme zadanou nějakou množinu (multimnožinu) prvků a cílem je nalézt mezi nimi takový prvek, který má danou hodnotu vyhledávacího klíče, anebo případně zjistit, že takový prvek mezi nimi není.

### Metody vyhledávání

- Vyhledávání v lineárních datových strukturách  
Jeden z nejjednodušších typů vyhledávání. Pole nebo seznam. Sekvenční vs. binární.
- Vyhledávací stromy  
Binární vyhledávací stromy, AVL-stromy (vyvážený binární strom), B-stromy (uzel má víc, než 2 následníky), 2-3-4-stromy (vylepšená verze B-stromů, efektivní vyvažování), red-black stromy.
- Číslicové vyhledávání  
Lze použít pro prvky, které jsou reprezentovány v binárním tvaru (celá čísla, řetězce)
- Hašovací tabulky  
Pokud je vyžadováno časté vyhledávání dat velmi rychle. Méně častá změna struktury dat.

## 2.4 Vyhledávání v lineárních datových strukturách

Mezi nejjednodušší případy vyhledávání patří vyhledávání v lineární datové struktuře, tj. v poli nebo v seznamu. Přepokládáme přitom, že prvky jsou v ní uloženy v libovolném pořadí (nesetříděné). Není zde jiný způsob, než prvky postupně procházet (zpravidla od začátku směrem ke konci) a každý srovnat s hledanou hodnotou. Počet srovnání se přitom pohybuje od 1, jestliže hledaný prvek je hned první, po  $n$ , jestliže hledaný prvek je až poslední anebo hledaný prvek mezi prohledávanými prvky není obsažen (nebyl nalezen). Tedy průměrný počet srovnání (je-li prvek nalezen) je  $\frac{1+n}{2}$ . Maximální počet srovnání je  $n$ .

Sekvenční vyhledávání v lineární datové struktuře má časovou složitost  $\theta(n)$

#### 2.4.1 Binární vyhledávání v setříděném poli

V případě pole je pro vyhledávání mnohem příznivější případ, když prvky jsou v něm uspořádány (seřazeny) dle velikosti vyhledávacího klíče. Zde se dá použít algoritmus binárního vyhledávání, často také nazývaný vyhledávání půlením intervalu.

Popis algoritmu

Vezmeme prvek, který je v poli uprostřed (je-li počet prvků sudý, jsou uprostřed dva prvky - zde vezmeme jeden z nich, při implementaci metody to zpravidla bývá ten levý), označme jeho index s.

Následně provedeme srovnání hledané x hodnoty s hodnotou středního prvku  $a_s$  :

- Nejprve srovnáme, zda je  $x < a_s$   
Pokud ano, pak zřejmě hledaný prvek, pokud v poli vůbec je, musí být v části L, jež je nalevo od středního prvku a s . Je-li část L neprázdná (obsahuje aspoň jeden prvek), rekurzivně na ni provedeme stejný postup. Je-li už prázdná, vyhledávání neúspěšně končí. Hledaný prvek není v poli obsažen.
- Pokud neplatí  $x < a_s$ , uděláme další srovnání. Srovnáme, zda je  $x > a_s$   
Pokud ano, musíme v dalším kroku hledání pokračovat v části P, jež je napravo od středního prvku a s . Je-li část P neprázdná (obsahuje aspoň jeden prvek), rekurzivně na ni provedeme stejný postup. Je-li už prázdná, vyhledávání neúspěšně končí.
- Pokud není ani  $x > a_s$ , zbývá už jen možnost, že platí  $x = a_s$ , čímž vyhledávání končí, neboť prvek  $a_s$  je hledaným prvkem.

Maximální počet kroků logaritmicky závisí na počtu prvků v prohledávané posloupnosti. V každém kroku provádíme nejvýše dvě operace srovnání. První operací zjistíme, zda hledaný prvek je menší než střední prvek. Pokud ano, pokračujeme v hledání v části nalevo. Pokud ne, druhou operací srovnání zjistíme, zda hledaný je větší než střední prvek, čímž rozhodneme, zda pokračovat v hledání v části napravo anebo už jsme hledaný prvek našli.

Složitost binárního vyhledávání je  $\theta(\ln(n))$

### 2.5 Binární vyhledávací stromy

Binární vyhledávání má velmi příznivou časovou složitost. Problém ovšem nastane, když se tato množina v průběhu času mění, tj. jsou k ní přidávány nové prvky nebo z ní jsou naopak některé prvky odebrány. Vkládání prvků doprostřed pole nebo jejich odebrání zprostředka pole je poměrně neefektivní operace, neboť je spojena s přesuny poměrně značné části prvků v poli. Pro takovéto případy je výhodnější použít vyhledávací stromy.

**Binární vyhledávací stromy** jsou binární stromy s vlastnostmi:

- V každém uzlu stromu je uložen jeden datový prvek.
- Každý uzel má nanejvýš dva potomky - levého a pravého.
- Pro každý uzel u a prvek v něm uložený c platí, že prvky uložené v levém podstromu uzlu u (má-li ulevý podstrom) jsou menší než prvek c a prvky uložené v pravém podstromu uzlu u (má-li u pravý podstrom) jsou větší než prvek c.

#### 2.5.1 Vyhledání prvku

##### 1. Počáteční krok

Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat u. Na začátku jím bude kořen stromu. Hledaná hodnota nechť je x.

## 2. Průběžný krok

Vezmeme prvek obsažený v aktuálním uzlu  $u$ , označme ho  $c$ , a provedeme jeho srovnání s hledanou hodnotou  $x$ :

- Nejprve srovnáme, zda je  $x < c$ :  
Pokud ano, pak je nutné v hledání pokračovat v levém podstromu. Jako nový aktuální uzel u položíme levého následníka současného aktuálního uzlu a znova provedeme krok 2.  
Pokud současný aktuální uzel levého následníka nemá, vyhledávání končí - hledaný prvek není ve stromu obsažen.
- Pokud není  $x < c$ , srovnáme, zda je  $x > c$ :  
Pokud ano, je nutné v hledání pokračovat v pravém podstromu. Jako nový aktuální uzel u položíme pravého následníka současného aktuálního uzlu a opět provedeme krok 2.  
Pokud uzel pravého následníka nemá, vyhledávání končí, hledaný prvek není ve stromu obsažen.
- Pokud není  $x > c$ , zbývá už jen případ, že platí  $x = c$ , čímž jsme u konce hledání, neboť prvek  $c$  obsažený v současném aktuálním uzlu  $u$  je tím hledaným prvkem.

Časová složitost  $\theta(h)$ , kde  $h$  je výška vyhledávacího stromu.

### 2.5.2 Přidání prvku

Operace přidání prvku do binárního vyhledávacího stromu znamená na příslušném místě přidat do stromu uzel, do kterého nový prvek vložíme. Označme přidávaný prvek  $x$ . Provedeme jeho vyhledání ve stromu. Použijeme k tomu již popsaný algoritmus vyhledávání. Ten může skončit třemi způsoby:

- Prvek  $x$  byl ve stromu nalezen. Tím přidávání končí, neboť prvek  $x$  už je ve stromu obsažen a u vyhledávacích stromů se nepředpokládá vícenásobný výskyt stejného prvku.
- Vyhledávání skončilo v uzlu  $u$  s prvkem  $c$ , přičemž  $x < c$  a přitom uzel  $u$  už nemá levého následníka. V tom případě přidáme ke stromu nový uzel jako levého následníka uzlu  $u$  a do něho nový prvek  $x$  vložíme.
- Vyhledávání skončilo v uzlu  $u$  s prvkem  $c$ , přičemž  $x > c$  a přitom uzel  $u$  už nemá pravého následníka. V tom případě přidáme ke stromu pravého následníka uzlu  $u$ , do kterého nový prvek  $x$  vložíme.

Časová složitost  $\theta(h)$ , kde  $h$  je výška vyhledávacího stromu.

### 2.5.3 Odebrání prvku

Operace odebrání prvku z binárního stromu znamená na příslušném místě zrušení uzlu ve stromu. Označme odebíraný prvek  $x$ . Vyhledáme prvek  $x$  ve stromu. Vyhledání může skončit třemi způsoby:

- Prvek  $x$  nebyl ve stromu nalezen – není co odebrat.
- Prvek byl nalezen v uzlu  $v$ , který má nejvýše jednoho následníka. Tento uzel zrušíme.
- Prvek byl nalezen v uzlu  $v$ , který má dva následníky. V tomto případě do uzlu  $v$  přesuneme buďto nejpravější (největší) prvek z jeho levého podstromu anebo nejlevější (nejmenší) prvek z jeho pravého podstromu a uzel, z kterého byl prvek přesunut, zrušíme.

Časová složitost  $\theta(h)$ , kde  $h$  je výška vyhledávacího stromu.

## 2.6 Red-black stromy

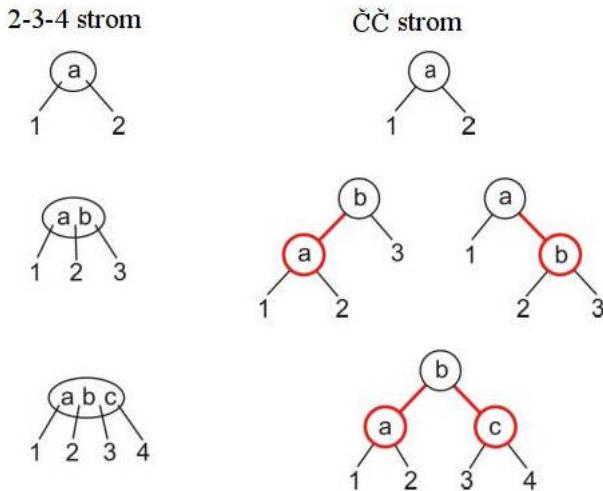
Červeno-černé stromy jsou vyvážené binární vyhledávací stromy. Mají obdobnou strukturu jako B-stromy řádu 4.

**B-stromy řádu 4** jsou vyvážené vyhledávací stromy s vlastnostmi:

- V každém uzlu mohou být uloženy 1-3 datové prvky.
- Každý uzel je list nebo má o 1 následníka více, než je počet prvků uložených v uzlu.
- Pro každý prvek uložený ve stromu platí pravidlo, že prvky uložené ve stejném uzlu vlevo od něho jsou menší a prvky uložené vpravo od něho jsou větší. Totéž platí pro podstromy – prvky v podstromu, který je vlevo, jsou menší a prvky v podstromu vpravo jsou větší.
- Všechny listy mají stejnou vzdálenost od kořene (strom je vyvážený).

### 2.6.1 Vytvoření RB stromu z B-stromu

Jednotlivé uzly B-stromu nahradíme 1-3 binárními uzly. Horní nahrazující uzel bude černý, uzly pod ním budou červené uzly. Horní uzel bude s uzly pod ním spojen červenými hranami (viz obrázek).



Obrázek 1: Vytvoření RB stromu z B-stromu

### 2.6.2 Vlastnosti RB stromu

- Kořenový uzel je vždy černý. Barva ostatních uzlů je dána barvou hrany, kterou je uzel spojen s předchůdcem.
- Mezi kořenem a libovolným listovým uzlem je stejný počet černých hran (a tím i černých uzlů).
- Ve stromu nikdy nenásledují dvě červené hrany po sobě (a tím i nikdy nenásledují dva červené uzly po sobě).
- Nechť mezi kořenem a listem je  $m$  černých hran. Pak mezi kořenem a libovolným listem je nejvýše  $m+1$  červených hran.

### 2.6.3 Přidání prvku

Přidání probíhá standardním způsobem jako v běžném binárním vyhledávacím stromu.

Označení: x - přidávaný prvek

Vyhledáme prvek x ve stromu. Vyhledání může skončit třemi způsoby:

- Prvek x byl ve stromu nalezen – nelze ho znovu přidat a přidávání tím končí.
- Vyhledávání skončilo v uzlu u, ve kterém je uložen prvek c, přičemž  $x < c$  a uzel u nemá levého následovníka. Vytvoříme nový uzel jako levého následovníka uzlu u a do něho dáme přidávaný prvek x.
- Vyhledávání skončilo v uzlu u s prvkem c, přičemž  $x > c$  a uzel u nemá pravého následovníka. Vytvoříme pravého následovníka uzlu u a do něho dáme přidávaný prvek x.

Přidávání je tedy realizováno vytvořením nového uzlu a jeho spojením hranou s uzlem u, ve kterém skončilo vyhledávání. Vytvořený uzel bude novým listem stromu. Aby zůstala zachována podmínka, že mezi kořenem a libovolným listem byl stejný počet černých hran, musíme nový uzel spojit s uzlem u červenou hranou a nový uzel bude červený uzel. Pokud uzel u je černý uzel, operace přidání je ukončena. Jestliže ale uzel u je červený uzel, jsou nyní ve stromu dvě červené hranы (dva červené uzly) po sobě.

**Odstranění dvou červených uzlů po sobě** rotace x výměna barev.

- Horní z dvojice po sobě následujících červených uzlů nemá červeného sourozence – jednoduchá nebo dvojitá rotace.
- Horní z dvojice červených uzlů má červeného sourozence. Změníme jejich obarvení na černou barvu a barvu jejich předchůdce, pokud to není kořen, změníme na červenou barvu.

### 2.6.4 Odebrání prvku

Označení: x - odebrávaný prvek

1. Vyhledáme prvek x ve stromu. Vyhledání může skončit třemi způsoby:

- Prvek x nebyl ve stromu nalezen – není co odebrat.
- Prvek byl nalezen v uzlu v, který má nejvýše jednoho následníka. Tento uzel zrušíme.
- Prvek byl nalezen v uzlu v, který má dva následníky. V tomto případě do uzlu v přesuneme buďto nejpravější (největší) prvek z jeho levého podstromu anebo nejlevější (nejmenší) prvek z jeho pravého podstromu a uzel, z kterého byl prvek přesunut, zrušíme.

2. Další postup závisí na tom, jakou barvu má rušený uzel.

- Rušený uzel má červenou barvu. V tomto případě zrušení uzlu neovlivní počet černých uzlů (a černých hran) a odebrání je ukončeno.
- Rušený uzel má černou barvu a má červeného následníka. Následníka přebarvíme na černou barvu. Tím počet černých uzlů (a černých hran) zůstane zachován a odebrání je ukončeno.
- Rušený uzel v má černou barvu a nemá červeného následníka. Uzel v obarvíme jako dvojitý černý. Toto obarvení vyjadřuje, že zrušením uzlu by nastal deficit černé barvy na cestách od kořenu k listům, na kterých tento uzel leží.

Další postup je transformacemi dosáhnout, aby ve stromu nebyl žádný uzel s dvojitým černým obarvením. Přitom u uzlu, který chceme zrušit, lze při odstranění jeho dvojitého černého obarvení ho ze stromu odstranit.

**Odstranění dvojitěho černého obarvení uzlu** rotace x výměna barev.

- Sourozenec uzlu s dvojitým černým obarvením je černý uzel a tento má přitom aspoň jednoho červeného následníka – jednoduchá nebo dvojitá rotace (spojená s určitým přebarvením uzlů).
- Sourozenec uzlu v s dvojitým černým obarvením je černý uzel a tento přitom nemá žádného červeného následníka. Jedno černé obarvení od obou uzlů (uzlu v a jeho sourozence) odebereme a k předchůdci těchto uzlů naopak jedno černé obarvení přidáme. Uzel v bude tímto mít jedno černé obarvení a jeho sourozenec bude červeně obarven.
  - Předchůdce uzlu v je červený uzel – nyní bude černý uzel.
  - Předchůdce uzlu v je černý uzel. Pokud to není kořen, bude mít nyní dvojité černé obarvení.
- Sourozenec uzlu v s dvojitým černým obarvením je červený uzel – jednoduchá rotace.

**Časová složitost operací** je závislá na výšce červeno-černého stromu. Odtud  $\theta(\ln(n))$ .

## 2.7 AVL stromy

Jsou určitým způsobem vyvážené binární vyhledávací stromy. Vyváženosť v AVL stromu je zajištěna podmírkou, že pro každý uzel u stromu musí platit, že rozdíl mezi výškou jeho levého podstromu a výškou jeho pravého podstromu je nejvýše 1. Přitom výškou podstromu zde rozumíme maximum ze vzdáleností od uzlu u k uzlům daného podstromu, tedy vzdálenost mezi uzlem u a uzly, které jsou úplně naspodu podstromu. Pokud uzel nemá daného následníka (levého, pravého), je výška tohoto podstromu 0. Při operaci přidání prvku do stromu je přidán nový uzel, při operaci odebrání prvku ze stromu je naopak uzel zrušen. Tyto operaci v příslušném místě, kde byl přidán nebo zrušen uzel, mění tvar stromu a mohou vést k porušení uvedené podmínky vyvážení u některého uzlu stromu. Proto je po operaci přidání nebo odebrání nutné ověřit, zda k této situaci nedošlo. Aby toto ověřování proběhlo efektivně, je v každém uzlu proměnná  $b$  ( $b=balance$ ), která je označovaná jako faktor vyvážení uzlu a která obsahuje informaci o jeho stávajícím vyvážení. Uzel je vyvážený, jestliže jeho faktor vyvážení je 1, 0 nebo -1. Při přidání prvku do stromu nebo odebrání prvku ze stromu se v důsledku změny tvaru stromu může v některém uzlu jeho faktor vyvážení změnit na 2 nebo -2. V tom případě je zapotřebí udělat transformaci, která obnoví vyvážení daného uzlu.

### 2.7.1 Přidání prvku

Operace přidání prvku do AVL stromu znamená na příslušném místě přidání uzlu do stromu, do kterého nový prvek vložíme, a následně aktualizujeme v příslušných částech stromu údaje o vyvážení v uzlech a ověřujeme, zda v některém uzlu nedošlo k narušení vyvážení stromu, a pokud ano, uzel vhodnou transformací vyvážíme.

Označení: x - přidávaný prvek

1. Vyhledáme prvek x ve stromu. Vyhledání může skončit třemi způsoby:

- Prvek x byl ve stromu nalezen. Tím přidávání končí, neboť prvek x už je ve stromu obsažen a nelze ho přidat.
- Vyhledání skončilo v uzlu u, ve kterém je uložen prvek c, přičemž  $x < c$  a přitom uzel u už nemá levého následníka. V tom případě přidáme ke stromu nový uzel v jako levého následníka uzlu u a do něho nový prvek x vložíme.
- Vyhledání skončilo v uzlu u s prvkem c, přičemž  $x > c$  a přitom uzel u už nemá pravého následníka. V tom případě přidáme ke stromu nový uzel v jako pravého následníka uzlu u a do něho nový prvek x vložíme.

2. Po přidání uzlu procházíme uzly stromu na cestě od přidaného uzlu v směrem ke kořenu. Pro každý procházený uzel u na cestě aktualizujeme jeho faktor vyvážení.

- Pokud přidaný prvek je v levém podstromu aktuálního uzlu u, jeho faktor vyvážení zvýšíme o 1.

Po aktualizaci faktoru vyvážení mohou nastat případy:

- Faktor vyvážení uzlu u je nyní 0. V tomto případě přidání prvku x nemá vliv na faktory vyvážení dalších uzlů na cestě ke kořenu a operace přidání tím skončila.
- Faktor vyvážení uzlu u je nyní 1. Je-li uzel u již kořen, operace přidání skončila. Jinak aktuální uzlem u učiníme předchůdce současného uzlu u a opět krok 2.
- Faktor vyvážení uzlu u je nyní 2. Uzel u je nevyvážený - vyvážíme ho vhodnou transformací (RR, RL). Po vyvážení:
  - \* Je-li uzel u již kořen, operace přidání skončila.
  - \* Je-li po vyvážení hodnota faktoru vyvážení uzlu u rovna 0, operace přidání je ukončena (maximum z výšek podstromů uzlu u se transformací snížilo o 1, tudíž už to nemá vliv na faktor vyvážení předchůdce uzlu u). Jinak přejdeme k předchůdci uzlu u a opět krok 2.

- Pokud přidaný prvek je v pravém podstromu aktuálního uzlu u, jeho faktor vyvážení snížíme o 1.

Po aktualizaci faktoru vyvážení mohou nastat případy:

- Faktor vyvážení uzlu u je nyní 0. V tomto případě přidání prvku x nemá vliv na faktory vyvážení dalších uzlů na cestě ke kořenu a operace přidání tím skončila.
- Faktor vyvážení uzlu u je nyní -1. Je-li uzel u již kořen, operace přidání skončila. Jinak aktuální uzlem u učiníme předchůdce současného uzlu u a opět krok 2.
- Faktor vyvážení uzlu u je nyní -2. Uzel u je nevyvážený - vyvážíme ho vhodnou transformací (LL, LR). Po vyvážení:
  - \* Je-li uzel u již kořen, operace přidání skončila.
  - \* Je-li po vyvážení hodnota faktoru vyvážení uzlu u rovna 0, operace přidání je ukončena (maximum z výšek podstromů uzlu u se transformací snížilo o 1, tudíž už to nemá vliv na faktor vyvážení předchůdce uzlu u). Jinak přejdeme k předchůdci uzlu u a opět krok 2.

### 2.7.2 Odebrání prvku

Operace odebrání prvku z AVL stromu znamená na příslušném místě zrušení uzlu ve stromu. Následně aktualizujeme v příslušných částech stromu údaje o vyvážení v uzlech a ověřujeme, zda v některém uzlu nedošlo k narušení vyvážení stromu, a pokud ano, uzel vhodnou transformací vyvážíme.

Označení: x - odebíraný prvek

1. Vyhledáme prvek x ve stromu. Vyhledání může skončit třemi způsoby:

- Prvek x nebyl ve stromu nalezen – není co odebrat.
- Prvek byl nalezen v uzlu v, který má nejvýše jednoho následníka. Tento uzel zrušíme.
- Prvek byl nalezen v uzlu v, který má dva následníky. V tomto případě do uzlu v přesuneme buďto nejpravější (největší) prvek z jeho levého podstromu anebo nejlevější (nejmenší) prvek z jeho pravého podstromu a uzel, z kterého byl prvek přesunut, zrušíme.

2. Po zrušení uzlu procházíme uzly stromu na cestě od zrušeného uzlu směrem ke kořenu. Pro každý procházený uzel u na cestě aktualizujeme jeho hodnotu vyvážení b.

- Pokud zrušený uzel byl v levém podstromu aktuálního uzlu u, jeho faktor vyvážení snížíme o 1.

Po aktualizaci faktoru vyvážení mohou nastat případy:

- Faktor vyvážení uzlu u je nyní -1. V tomto případě zrušení uzlu nemá vliv na faktory vyvážení dalších uzlů na cestě ke kořenu a operace odebrání tím skončila.
- Faktor vyvážení uzlu u je nyní 0. Je-li uzel u již kořen, operace odebrání skončila. Jinak aktuální uzlem u učiníme předchůdce současného uzlu u a opět krok 2.
- Faktor vyvážení uzlu u je nyní -2. Uzel u je nevyvážený - vyvážíme ho vhodnou transformací (LL, LR). Po vyvážení:
  - \* Je-li uzel u již kořen, operace odebrání skončila.
  - \* Je-li po vyvážení faktor vyvážení uzlu u roven -1, operace přidání je ukončena (maximum z výšek podstromů uzlu u se transformací zvýšilo o 1, tudíž už to nemá vliv na faktor vyvážení předchůdce uzlu u). Jinak přejdeme k předchůdci uzlu u a opět krok 2.
- Pokud zrušený uzel je v pravém podstromu aktuálního uzlu u, jeho faktor vyvážení zvýšíme o 1.  
Po aktualizaci faktoru vyvážení mohou nastat případy:
  - Faktor vyvážení uzlu u je nyní 1. V tomto případě zrušení uzlu nemá vliv na faktory vyvážení dalších uzlů na cestě ke kořenu a operace přidání tím skončila.
  - Faktor vyvážení uzlu u je nyní 0. Je-li uzel u již kořen, operace odebrání skončila. Jinak aktuální uzlem u učiníme předchůdce současného uzlu u a opět krok 2.
  - Faktor vyvážení uzlu u je nyní 2. Uzel u je nevyvážený - vyvážíme ho vhodnou transformací (RR, RL). Po vyvážení:
    - \* Je-li uzel u již kořen, operace přidání skončila.
    - \* Je-li po vyvážení faktor vyvážení uzlu u roven 1, operace přidání je ukončena (maximum z výšek podstromů uzlu u se transformací zvýšilo o 1, tudíž už to nemá vliv na faktor vyvážení předchůdce uzlu u). Jinak přejdeme k předchůdci uzlu u a opět krok 2.

**Časová složitost operací**  $\theta(\log(n))$ .

## 2.8 B-stromy

B-stromy jsou velmi významným typem vyhledávacích stromů. Mají v uzlech uloženo více prvků. Struktura B-stromů je definována následujícími vlastnostmi:

- Kapacita uzlu (počet prvků, který lze do uzlu uložit) je u všech uzlů stromu stejná a volíme ji před začátkem vytváření stromu, kapacitu označme  $r$  ( $r \geq 2$ ) Protože v B-stromech má významnou úlohu polovina z tohoto počtu, zavedeme si pro ni samostatné označení  $p = \frac{r}{2}$  pro r sudé a  $p = \frac{r-1}{2}$  pro r liché.
- Důležité pro efektivní využití uzlů je jejich zaplnění. Všechny uzly vyjma kořene musí být aspoň z poloviny zaplněny prvky pro r sudé nebo p prvky pro r liché, tedy počet prvků v uzlech uložených musí být v rozmezí p až r prvků. Jedině u kořene stačí, aby obsahoval aspoň jeden prvek, tedy jeho zaplnění je v rozmezí 1 až r prvků.
- Prvky uložené v uzlu jsou v něm seřazeny vzestupně dle velikosti.
- Uzel je buďto list anebo má o jednoho následníka více, než je počet prvků v něm uložený. Přitom pro prvky v jednotlivých podstromech, které těmito následníky začínají, platí:
  - Pro každý prvek  $d$  v podstromu začínajícího uzlem  $nsl_0$  platí  $d < c_1$ .
  - Pro každý prvek  $d$  v podstromu začínajícího uzlem  $nsl_1$  platí  $c_1 < d < c_2$ .
  - .....
  - Pro každý prvek  $d$  v podstromu začínajícího uzlem  $nsl_{k-1}$  platí  $c_{k-1} < d < c_k$ .

- Pro každý prvek  $d$  v podstromu začínajícího uzlem  $nsl_k$  platí  $d > c_k$ .
- Listy jsou v B-stromu jen v jeho poslední (nejspodnější) úrovni.

Podle maximálního počtu následníků označujeme i řád B-stromu. B-strom s kapacitou uzlu  $r$  má řád  $r + 1$ , neboť nelistový uzel může mít až  $r + 1$  následníků.

### 2.8.1 Vyhledání prvku

#### 1. Počáteční krok

Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat  $u$ . Na začátku jím bude kořen stromu.

Hledaná hodnota nechť je  $x$ .

#### 2. Průběžný krok

Provedeme vyhledání hodnoty mezi prvky uloženými v aktuálním uzlu  $u$ . Protože prvky jsou v uzlu setříděné, lze k tomu použít binární vyhledávání. To použijeme v případě, kdy kapacita uzlů je zvolena dostatečně velká, aby se to vyplatilo. Vyhledání může skončit třemi způsoby:

- Prvek byl v aktuálním uzlu  $u$  nalezen, čímž vyhledávání úspěšně končí.
- Prvek nebyl v aktuálním uzlu  $u$  nalezen a tento uzel je list. Tím vyhledávání končí - hledaný prvek není ve stromu obsažen.
- Prvek nebyl v aktuálním uzlu  $u$  nalezen a tento uzel je nelistový. V tom případě vyhledávání skončilo v místě, kde je odkaz na následníka, ve kterém vyhledávání má pokračovat (tj. na následníka, kterým začíná podstrom, jež by hledaný prvek měl obsahovat). Tohoto následníka učiníme novým aktuálním uzlem a opět se provede krok 2.

### 2.8.2 Přidání prvku

Chceme-li do B-stromu přidat prvek, znamená to najít příslušný uzel, do kterého nový prvek patří, a následně ověřit, zda přitom nedošlo k přeplnění, a pokud ano, provést rozdelení uzlu.

#### 1. Přidání prvku

Označme přidávaný prvek  $x$ . Provedeme je vyhledání ve stromu. Použijeme k tomu běžný algoritmus pro vyhledání. Ten může skončit dvěma způsoby:

- Prvek  $x$  byl ve stromu nalezen. Tím přidávání končí, neboť prvek  $x$  už je ve stromu obsažen (u vyhledávacích stromů se nepředpokládá vícenásobný výskyt stejného prvku).
- Vyhledávání skončilo v listovém uzlu  $u$  v místě, kam nový prvek podle velikosti vzhledem k ostatním prvkům patří. Prvek na toto místo vložíme. Pokud uzel  $u$  předtím nebyl zcela zaplněn, operace přidání tím končí. Jinak provedeme rozdelení uzlu.

#### 2. Rozdelení uzlu

Jestliže uzel  $u$  má po přidání  $r+1$  prvků, tedy došlo k jeho přeplnění uzlu, rozdělíme ho na tři části:  $p$  prvků na začátku uzlu  $u$  + prvek uprostřed uzlu  $u$  +  $p$  prvků na konci uzlu.

Části s  $p$  prvků budou tvořit nové listy. Prvek, jež je v uzlu  $u$  uprostřed, se přesune do předchůdce na místo, kde byl původní odkaz na list. Po přesunu nalevo a napravo od něho vytvoříme nové odkazy na nově vzniklé listy.

Zřejmě po přidání uzlu do předchůdce v něm může dojít rovněž k jeho přeplnění, pokud předtím byl zcela zaplněn. To se řeší stejným způsobem - rozdelením tohoto uzlu na tři části s počtem prvků  $p+1+p$ . Dvě jeho části s  $p$  prvků budou tvořit nové uzly a zbývající prostřední prvek vložíme do jeho předchůdce. Takto postupujeme směrem nahoru, až buďto narazíme na uzel, u kterého po vložení dalšího prvku nedojde k přeplnění, anebo se nakonec dostaneme až ke kořenu. Pokud i u něho dojde k přeplnění, rozdělí se a střední prvek v tomto rozdelení bude nyní nový kořen. V této situaci dojde ke zvětšení výšky stromu.

### 2.8.3 Odebrání prvku

Chceme-li z B-stromu odebrat prvek, znamená to vyhledat uzel, ve kterém se prvek nachází, prvek z něho odstranit a následně ověřit, zda odebráním prvku nepoklesl počet prvků v daném uzlu pod přípustnou mez, a pokud ano, je nutné to vyřešit.

#### 1. Odebrání prvku:

Označme odstraňovaný prvek  $x$ . Provedeme jeho vyhledání ve stromu. Použijeme k tomu běžný algoritmus pro vyhledání. Ten může skončit třemi způsoby:

- Prvek  $x$  nebyl ve stromu nalezen - není co odebrat.
- Prvek  $x$  byl nalezen v listovém uzlu. Prvek z uzlu odstraníme. Pokud list je po zrušení prvku aspoň z poloviny zaplněn, operace odebrání končí. Jinak přejdeme ke kroku 2.
- Prvek  $x$  byl nalezen v uzlu  $u$ , který není listem. Prvek  $x$  z uzlu odstraníme a na volné místo v uzlu  $u$  přesuneme buďto největší prvek z jeho levého podstromu, což je poslední prvek v nejpravějším listu podstromu, anebo nejmenší prvek z jeho pravého podstromu, což je první prvek v nejlevějším listu podstromu. Pokud list, odkud jsme prvek přesunuli, je stále aspoň z poloviny zaplněn, operace odebrání prvku končí. Jinak přejdeme ke kroku 2.

#### 2. Zmenšení počtu prvků v uzlu

Sem se dostaváme v situaci, kdy po odstranění prvku ze stromu je nyní ve stromu list  $v$ , který má jen  $p-1$  prvků. Jak se tento stav řeší, závisí na zaplnění přímých sousedů listu. Jsou dvě možnosti:

- List  $v$  má aspoň jednoho přímého souseda, který má více než  $p$  prvků. Přímým sousedem zde máme na mysli uzel, který je nejen vedle uzlu  $v$ , ale má i stejnýho předchůdce. Pak do listu  $v$  přesuneme prvek z předchůdce a na prázdne místo v předchůdci přesuneme příslušný prvek ze souseda. Následující obrázek ukazuje tento přesun pro oba možné přímé sousedy, nejdříve pro levého souseda, pak pro pravého souseda.
- List  $v$  má jen přímé sousedy, které mají právě  $p$  prvků. Pak vytvoříme nový list s r prvky tak, že sloučíme prvky z listu  $v$  + prvek z předchůdce + prvky ze souseda.  
Je zřejmé, že tímto ubyl jeden prvek v předchůdci. Pokud tento má nyní jen  $p-1$  prvků, řeší se to analogicky v závislosti na tom, kolik prvků mají jeho přímí sousedé. Takto se můžeme případně dostat až k uzlu, který je následníkem kořene. Pokud je vytvořen nový uzel sloučením s jeho přímým sousedem a pokud kořen v této chvíli má jen jeden prvek, dojde přitom k vytvoření nového kořene a zároveň ke snížení výšky stromu. Na následujícím obrázku je tato situace pro případ, kdy je pro sloučení vzat levý přímý soused.

**Časová složitost operací**  $\theta(\log(n))$ .

## 2.9 Hašování

Datová struktura použitá v hašování pro uložení prvků je tabulka. Tabulka se skládá z řádků. U hašování pro řádky tabulky používáme označení příhrádky. V každé příhrádce je místo pro uložení jednoho datového prvku. Počet příhrádek v tabulce, tedy kapacitu tabulky označme  $m$ . Na jednotlivé příhrádky v tabulce se odkazujeme (adresujeme je) čísla 0 až  $m-1$ . Při implementaci hašování se tabulka snadno realizuje pomocí pole. Datový typ prvků pole se zvolí takový, aby se do něho daly uložit údaje, které ukládáme do příhrádek tabulky. Tím každý prvek pole reprezentuje jednu příhrádku tabulky a indexování prvků pole odpovídá adresování jednotlivých příhrádek v tabulce.

Základem hašování je hašovací funkce. Je to zobrazení, které hodnotě prvku (nebo vyhledávacímu klíči prvku, pokud prvek je strukturovaný typ) přiřadí číslo některé z příhrádek v tabulce, tedy číslo v rozmezí 0 až  $m-1$ . Hašovací funkce se typicky sestaví ze dvou funkcí. Ta první hodnotu prvku zobrazí na celé (nezáporné) číslo. Druhá celé číslo zobrazí na číslo příhrádky v tabulce, tedy na celé číslo z intervalu  $<0,m-1>$ .

Cílem hašovací funkce je rovnoměrné rozmístění prvků v tabulce. Z toho plynne, že první funkce, která převádí hodnotu prvku na celé číslo, by měla mít vlastnosti:

- Zobrazovat hodnoty prvků na co největší počet různých celých čísel.
- Zobrazení na celá čísla by mělo být rovnoměrné (na jednotlivá čísla by se měl zobrazovat přibližně stejný počet prvků, které chceme do hašovací tabulky uložit).

Dalším přirozeným požadavkem na hašovací funkci je, aby její výpočet nebyl příliš časově náročný.

### 2.9.1 Hašovací funkce pro řetězce

Řetězec si označme  $z_1 z_2 \dots z_k$ , kde  $z_i$  je znak z řetězce a  $k$  je délka řetězce.

**První část hašovací funkce** Jedna z jednodušších funkcí zobrazující řetězec na celé číslo je  $c_1(z_1 z_2 \dots z_k) = p * \text{asc}(z_1) + q * \text{asc}(z_2) + \text{asc}(z_k) + k$ , kde  $p$  a  $q$  jsou zvolené konstanty, nejlépe prvočísla (např.  $p=127$ ,  $q=31$ ), protože ty mají nejlepší předpoklady pro rovnoměrné zobrazení do množiny celých čísel. Funkce  $\text{asc}$  převádí znak na jeho ASCII hodnotu (nebo Unicode hodnotu).

Dokonalejší, ale na druhé straně náročnější na výpočet, je funkce  $c_2(z_1 z_2 \dots z_k) = p^{k-1} * \text{asc}(z_1) + p^{k-2} * \text{asc}(z_2) + \dots + p * \text{asc}(z_{k-1}) + \text{asc}(z_k)$ , kde  $p$  je konstanta, opět nejlépe prvočíslo (např.  $p=31$ ).

**Drugá část hashovací funkce** která převádí celé číslo na číslo příhrádky v hašovací tabulce, je velmi jednoduchá. Používá se pro ni operace modulo. Obecný zápis hašovací funkce je  $h(x) = c(x) \bmod m$ , kde  $c(x)$  je první část hašovací funkce a  $m$  je rozsah (počet příhrádek) hašovací tabulky. Opět je nejlepší zvolit  $m$  prvočíslo, protože to nemá žádného netriviálního vlastního dělitele, čímž poskytuje nejlepší předpoklady pro rovnoměrné rozmístění prvků v tabulce.

Vedle prvočíselného počtu příhrádek v tabulce se v praxi používá i počet příhrádek, který je mocnou čísla 2. Tento počet nemá tak dobré předpoklady pro rovnoměrné rozmístění prvků v tabulce, ale výpočet hašovací funkce je snadnější, protože místo operace modulo lze použít jednodušší operaci bitového součinu.

### 2.9.2 Metoda otevřeného adresování

V případě, kdy pozice v tabulce vypočítaná hašovací funkcí je obsazena, počítá další pozice tak dlouho, dokud se nenajde volná pozice anebo se nejistí, že tabulka už je zaplněna.

**Lineární hledání** nové pozice počítáme funkcí  $H(x, i) = (h(x) + i) \bmod m$ , kde  $h(x)$  je výchozí hašovací funkce,  $i$  je celočíselný parametr a  $m$  je rozsah tabulky.

Lineární umisťování za sebou vede k vytváření nežádoucích shluků. **Shlukem** nazýváme větší počty za sebou následujících obsazených příhrádek tabulky. Pokud je vypočítaná primární pozice obsazena a je přitom uvnitř takového shluku, znamená to při lineárním hledání, že musíme projít všechny příhrádky v tomto shluku za primární pozicí, než se dostaneme k nějaké volné sekundární pozici, abychom prvek do ní mohli uložit. Přitom shluky prodlužují nejen operaci přidání prvku do tabulky, ale také vyhledávání prvku v tabulce. Při vyhledávání začínáme na primární pozici a pokud na ní prvek není a tato pozice je přitom obsazena (je na ní jiný prvek), procházíme sekundární pozice tak dlouho, dokud prvek nenajdeme nebo se nedostaneme k volné pozici, což je příznakem toho, že hledaný prvek v tabulce není.

**Kvadratické hledání** Proto místo lineárního hledání se často používá kvadratické hledání. U něho sice také vznikají shluky, ale už v menší míře. Hašovací funkce používaná pro kvadratické hledání má

většinou jednoduchý tvar  $H(x, i) = (h(x) + i^2) \bmod m$ . U ní je už určitý problém, že během hledání se můžeme touto funkcí dostat znova na stejnou pozici, kterou jsme již prošli, aniž jsme přitom vyčerpali celou tabulkou.

**Dvojí hašování** je propracovanější metoda. Hašovací funkce má tvar  $H(x, i) = (h(x) + i * h_2(x)) \bmod m$ , kde  $h_2$  je sekundární hašovací funkce v rozmezí hodnot 1 až  $m-1$ . V praxi nejčastější vychází z primární funkce. Máme funkci  $c(x)$ , primární hašovací funkci  $h(x) = c(x) \bmod m$ . Sekundární hašovací funkce vznikne jako  $h_2(x) = 1 + (c(x) \bmod (m - 1))$ . Pozici nyní můžeme počítat rekursivně

$$H(x, 0) = h(x)$$

$$H(x, i) = (H(x, i - 1) + h_2(x)) \bmod m \text{ pro } i = 1, 2, 3, \dots$$

**Vyhledávání v tabulce** Při vyhledávání v hašovací tabulce nejprve vypočítáme hodnotu hašovací funkce pro hledaný prvek  $x$ . Podíváme se do tabulky na příhrádku, na kterou ukazuje hodnota hašovací funkce. Mohou nastat případy:

- Příhrádka je prázdná – hledaný prvek není v tabulce.
- V příhrádce je hledaný prvek  $x$  – vyhledávání tím úspěšně končí.
- V příhrádce je jiný prvek než  $x$ . Začneme postupně počítat další možné pozice a srovnávat prvky na nich s hledaným prvkem  $x$ , dokud buďto hledaný prvek nenalezneme anebo se nedostaneme na prázdnou příhrádku anebo nevyčerpáme všechny možné pozice.

### 2.9.3 Zřetězení

Předchozí metoda otevřeného adresování má dvě nevýhody:

- Počet prvků, jež lze do tabulky uložit, je omezen její velikostí. Pokud dopředu neznáme, kolik prvků bude do tabulky ukládáno, může se stát, že ji stanovíme malou a dojde k jejímu přeplnění. Následné zvětšení velikosti tabulky je většinou časově náročné.
- Při vyhledávání, zejména v dost zaplněné tabulce, procházíme v důsledku otevřeného adresování i prvky, které mají jinou hodnotu hašovací funkce, čímž se doba vyhledávání zvětšuje.

Tyto nevýhody odstraňuje metoda zřetězení, která k ukládání dalších prvků se stejnou hodnotou hašovací funkce využívá seznamy. Hašovací tabulka v tomto případě obsahuje ukazatele na začátek (první uzel) jednotlivých seznamů.

Pokud potřebujeme do hašovací tabulky prvky nejen přidávat, ale i odebírat je, pak je nevhodnější použít metodu zřetězení pro řešení kolicí. Odebrání se pak realizuje vyhledáním prvku v příslušném seznamu a zrušením uzlu, který odebraný prvek obsahuje.

V případě, že odebrání bude v hašovací tabulce s otevřeným adresováním, je realizace odebrání o něco komplikovanější. Budeme rozeznávat tři různé stavby příhrádky tabulky:

- Příhrádka je prázdná – v této příhrádce nikdy nebyl uložen žádný prvek.
- Příhrádka je volná – v této příhrádce byl uložen prvek, ale ten byl pak z tabulky odebrán.
- Příhrádka je obsazena – v této příhrádce je uložen prvek.

Při vyhledávání budeme volné příhrádky přeskakovat. Vyhledávání skončí, až když hledaný prvek najdeme nebo se dostaneme na prázdnou příhrádku (nebo jsme prošli celou tabulkou, či její podstatnou část). Při přidávání budeme hledat první prázdnou nebo volnou příhrádku.

**Časová složitost** vyhledávání je dána složitostí hašovací funkce (můžeme považovat za konstantní - řetězce mají omezenou délku). Při vkládání bez kolizí  $\theta(1)$ . Složitost roste s počtem kolizí v tabulce. Volba hašovací tabulky je zejména podstatná u metody otevřeného adresování. Volíme ji aspoň o 10% větší, než je očekávaný počet prvků.

**Perfektní hašování** Je hašování bez kolizí. Používá se v případech, kdy jsou známy prvky, které budou uloženy do hašovací tabulky. Pro tyto prvky se navrhne samostatná hašovací funkce, při jejímž použití nenastanou žádné kolize.

**Minimální perfektní hašování** Je perfektní hašování, kdy navíc po uložení prvků nezůstanou v tabulce žádné volné příhrádky. Počet příhrádek v tabulce je tedy roven počtu uložených prvků. Nalezení takové hašovací funkce je ale obtížné.

#### 2.9.4 Extendible hashing

Metoda hašování nazývaná extendible hashing nevyžaduje počáteční stanovení velikosti hašovací tabulky a navíc poskytuje konstantní časovou složitost vyhledání prvku bez ohledu na to, kolik prvků je do tabulky vloženo. Hašovací datová struktura se skládá ze dvou částí - adresáře a příhrádky. Adresář je pole odkazů na příhrádky. Velikost pole adresáře je  $2^d$  odkazů. Hodnota  $d$  se dynamicky mění v závislosti na počtu prvků uložených v hašovací strukture. Na začátku je zpravidla  $d$  malé. Při postupném zvyšování počtu ukládaných prvků do hašovací struktury se v určitých okamžicích hodnota  $d$  zvyšuje. Příhrádka je místo pro uložení pevně stanoveného počtu datových prvků. Velikost všech příhrádek je stejná. Základem metody je opět hašovací funkce, která datový prvek zobrazí na celé číslo. Posledních  $d$  bitů tohoto čísla je indexem v poli adresáře, kde najdeme odkaz na příhrádku, ve které je uložen daný prvek.

**Vyhledání prvku v hašovací struktuře** Jednotlivé byty hodnoty hašovací funkce si označíme:  
 $\dots b_{d+1}b_db_{d-1}\dots b_3b_2b_1$  ( $b_1$  je nejméně významný bit)

- Vypočítáme hodnotu hašovací funkce hledaného prvku.
- Vezmeme posledních  $d$  bitů  $b_d \dots b_1$  vypočtené hodnoty hašovací funkce. Ty jsou indexem v poli adresáře. V něm v prvku, který odpovídá tomuto indexu, zjistíme odkaz na příslušnou příhrádku.
- V příhrádce procházíme v ní uložené prvky a srovnáváme je s hledaným prvkem, dokud nenalezneme prvek shodný s hledaným prvkem nebo je všechny neprojdeme (pak hledaný prvek není nalezen).

#### Přidání prvku do hašovací struktury

- Vypočítáme hodnotu hašovací funkce přidávaného prvku.
- Vezmeme posledních  $d$  bitů  $b_d \dots b_1$  vypočtené hodnoty hašovací funkce. Ty jsou indexem v poli adresáře. V něm v prvku pole, který odpovídá tomuto indexu, zjistíme odkaz na příslušnou příhrádku.
- Je-li v příhrádce místo pro uložení přidávaného prvku, prvek do příhrádky vložíme. Jinak je nutné příhrádku rozdělit.

#### Rozdelení příhrádky

- Pokud mají všechny prvky uložené v příhrádce stejnou hodnotu posledních  $d$  bitů  $b_d \dots b_1$  hodnot svých hašovacích funkcí, příhrádku rozdělíme na dvě a do první dámme prvky, jejichž hašovací funkce

má hodnotu dalšího bitu  $b_{d+1}$  rovnu 0. Do druhé dáme prvky, jejichž hodnota bitu  $b_{d+1}$  je 1. Po rozdelení přihrádek následně i zdvojnásobíme velikost adresář (hodnotu d zvýšíme o 1). Při zdvojnásobení velikosti adresáře zachováme odkazy na přihrádky, které se nerozdělily. Pokud by bit  $b_{d+1}$  byl u všech prvků stejný, vzali bychom další bit  $b_{d+2}$  a adresář bychom zvětšili čtyřikrát (hodnotu d bychom zvýšili celkově o 2). Atd.

- Jestliže prvky uložené v přihrádce nemají shodnou hodnotu posledních d bitů hodnot svých hašovacích funkcí, rozdělíme je do dvou přihrádek podle prvního (bráno zleva) z d bitů, který není u všech prvků identický. Nechť je to bit  $b_i$ , kde  $i \leq d$  a  $i$  je největší index takový, že hodnota tohoto bitu hašovacích funkcí všech prvků není stejná. Následně v adresáři příslušně upravíme odkazy na tyto přihrádky.

### Odebrání prvku z hašovací struktury

- Vyhledáme odebíraný prvek.
- Pokud byl nalezen, odstraníme ho z dané přihrádky.

Při větším odebírání, kdy vznikne více prázdných přihrádek, lze některé přihrádky sloučit. Případně po větším sloučení je někdy možné zmenšit velikost adresáře na polovinu. Jednoduchým způsobem lze sloučit prázdnou přihrádku, na kterou je jen jeden odkaz z adresáře, s přihrádkou, na kterou je opět jen jeden odkaz z adresáře a zároveň hodnoty indexů těchto odkazů v adresáři se liší jen v jednom bitu. Bylo by možné sloučit i v případě, že na obě sloučované přihrádky je stejný počet  $2^m$  odkazů a přitom hodnoty indexů jednotlivých odkazů v adresáři se liší právě v m bitech. Běžně se ale sloučování přihrádek při odebírání nedělá.

**Uložení na vnější paměti** Použití této metody je vhodné, pokud by standardní hašovací tabulka byla tak rozsáhlá, že by se nevešla do paměti. U rozšířitelného hašování stačí, aby v paměti byl uložen jen adresář. Přihrádky mohou být uloženy v souboru na vnější paměti.

# 3

**Programovací jazyky, jejich syntaxe a sémantika.** Přehled paradigmat: funkcionální, procedurální, logické, objektové. Symbolické výrazy a vyhodnocovací proces jazyka Scheme. Vytváření abstrakcí pomocí procedur. Procedury vyšších řádů: aplikace a mapování. Seznamy a hierarchická data. Indukce a rekurze: princip a příklady. Typy rekursivních výpočetních procesů. Lexikální a dynamický rozsah platnosti. Vlastnosti typových systémů.

## 3.1 Programovací jazyky, jejich syntaxe a sémantika

**Programovací jazyk** Programovací jazyk jsou smluvená pravidla, v souladu se kterými je vytvořen program. Program je předpisem s přesně daným tvarem a významem, podle kterého vzniká výpočetní proces. Výpočetní proces je proces, při kterém jsou zpracovávána vstupní data. Každý programovací jazyk musí mít popsánu svoji syntax a sémantiku. Dělení:

- dle míry abstrakce
  - nižší – kód stroje, jazyk symbolických adres, autokód, bajtkód
    - \* těsně vázané na hardware počítače
    - \* programové konstrukce jsou spjaty s instrukční sadou procesoru
    - \* prakticky žádné prostředky abstrakce → zdlouhavé programování → zanášení chyb do programu (špatně odhalitelné)
    - \* vazba na hardware umožňuje počítač „plně využít“
  - vyšší – většina jazyků
    - \* nejsou vázané na hardware
    - \* výrazně vyšší stupeň abstrakce → snadnější a rychlejší tvorba programů → menší riziko vzniku chyb
  - každý vyšší jazyk by měl programátorovi dát k dispozici:
    - \* *primitivní výrazy* (např. čísla, symboly, ...)
    - \* *prostředky pro kombinaci* primitivních výrazů do složitějších
    - \* *prostředky pro abstrakci* (možnost pojmenování složených výrazů a možnost s nimi dále manipulovat)
- dle způsobu překladu a spuštění
  - překlad přes nižší programovací jazyk (např. Pascal, C, Java, Cobol)
    - \* *překladač (komplátor, compiler)* programovacího jazyka je program, který načte celý program a poté provede jeho překlad do některého nižšího programovacího jazyka, typicky do *assembleru* nebo do nějakého bajtkódu, při překladu mohou nastat tři situace
      - překlad byl proveden do *kódu stroje*, pak překlad končí, protože jeho výsledkem je již program zpracovatelný procesorem, může být přímo spuštěn
      - překlad byl proveden do *assembleru*, pak je potřeba dodatečný překlad do kódu stroje, který obvykle provádí rovněž překladač
      - překlad byl proveden do *bajtkódu*, pak překlad končí, bajtkód ale není přímo zpracovatelný procesorem, musí tedy být ještě *interpretován* nebo *přeložen* (méně časté) dalším programem.
    - \* konstrukčně složitější než interpret
    - \* výsledkem je větší rychlosť, ale také náročnost na správně zapsaný kód
  - překlad přes vyšší programovací jazyk
    - \* typickou volbou překlad do některého z cílových jazyků, jejichž překladače jsou široce rozšířené, např. C

- interpretace (např. BASIC, Perl, Python, shell, Ruby)
  - \* *interpret programovacího jazyka* je program, který čte výrazy programu a postupně je přímo vykonává, interpret tedy (obvykle) z daného programu neprodukuje kód v jazyku stroje
  - \* kvůli efektivitě výsledného výpočetního procesu se může dodatečně kompilovat během interpretace, takovou komplikaci označujeme anglickým termínem *just in time compilation*.

**Syntaxe** Soubor přesně daných pravidel definujících, jak zapisujeme programy v daném programovacím jazyku. Syntax většiny programovacích jazyků je popsána v jejich standardech pomocí *formálních gramatik*, případně jejich vhodným rozšířením, například *Backus-Naurový formy*.

- *syntaktická chyba* je chybou v zápisu programu – kód, který není z pohledu daného jazyka syntakticky správný, nelze v podstatě chápát ani jako program v daném jazyku – odhalí překladač během překladu, interpret odhalí až v momentě, kdy se pokusí načíst chybný vstupní výraz

**Sémantika** *Význam programu.* V případě ohodnocování syntakticky neplatných řetězců není výpočet proveden. Sémantika popisuje procesy, které řídí počítac při vykonávání programu v daném programovacím jazyce. Například tím, že popisuje vztah mezi vstupem a výstupem programu, nebo popisem jak program poběží na určité platformě, tedy vytvořením modelu výpočtu.

- *sémantická chyba* je chyba ve významu programu, mezi typické sémantické chyby patří např. provádění operací nad daty neslučitelných typů, např. sčítání čísel a řetězců znaků, sémantické chyby lze odhalit během překladu či interpretace pouze v omezené míře, např. překladače C jsou během překladu schopny zjistit všechny konflikty v typech dat se kterými program pracuje, technicky ale již není třeba možné při překladu vyloučit chyby způsobené „dělením nulou“, protože hodnota dělitele může být proměnlivá a může během činnosti programu nabývat různých hodnot

## 3.2 Přehled paradigmat: funkcionální, procedurální, logické, objektové

**Funkcionální paradigma** Programy ve funkcionálních jazycích sestávají ze symbolických výrazů. Výpočet je potom tvořen *postupnou aplikací funkcí*, kdy funkce jsou aplikovány s hodnotami, které vznikly v předchozích krocích aplikací, výsledné hodnoty aplikace jsou použity při aplikaci dalších funkcí a tak dále. Díky absenci vedlejších efektů je možné chápát programy ve funkcionálních jazycích jako *ekvacionální teorie*. Rovněž se nepoužívají příkazy skoku ani strukturované cykly. Cykly se nahrazují *rekurzivními funkcemi* a *funkcemi vyšších řádů*. V některých funkcionálních jazycích také hraje důležitou roli *líné vyhodnocování* (jazyk Haskell, 1987) a *makra* (zejména dialekty LISPu). Nejznámější teoretický model funkcionálního programování je  $\lambda$ -kalkul, i když existují i modely jiné, např. modely založené na přepisovacích systémech a uspořádaných algebrách. Typickými zástupci funkcionálních programovacích jazyců jsou Common LISP, Scheme, ML a Anfix (dříve Aleph). Mezi čistě funkcionální jazyky patří Miranda, Haskell, Clean a další.

**Procedurální paradigma** Procedurální paradigma je spolu s funkcionálním nejstarší z paradigmat programování, někdy též bývá nazýváno paradigmatem *imperativním*. Programy napsané v procedurálním stylu jsou tvořeny *sekvencemi příkazů*, které jsou postupně vykonávány. Klíčovou roli při programování v procedurálním stylu hraje *příkaz přiřazení* (zápis hodnoty na dané místo v paměti) a související *vedlejší efekt* (některým typem přiřazení může být modifikována datová struktura a podobně). Sekvenční style vykonávání příkazů lze částečně ovlivnit pomocí *operací skoků a podmíněných skoků*. Formálním modelem procedurálních programovacích jazyců je *von Neumannův RAM stroj*. Z hlediska stylu programování můžeme procedurální paradigma rozdělit na:

- *Naivní*  
Naivní paradigma bývá někdy chápáno jako „samostatné paradigma“, častěji se mezi paradigma

programování ani neuvádí. Naivní procedurální jazyky se vyznačují jakousi všudypřítomnou chaotičností, mají obvykle nesystematický navrženou syntaxi a sémantiku, v některých rysech jsou podobné nestrukturovaným procedurálním jazykům. Mezi typické zástupce patří programovací jazyky rodiny BASIC (první jazyk BASIC byl navržen v roce 1968 za účelem zpřístupnit počítače a programování mimo komunitu počítačových vědců).

- *Nestrukturované*

Nestrukturované procedurální paradigma je velmi blízké assemblerům, programy jsou skutečně lineární sekvence příkazů a skoky jsou v programech realizovány příkazem typu „GO TO“, to jest „jdi na řádek“. V raných programovacích jazycích držících se tohoto stylu byly navíc všechny řádky programu číslované a skoky šly realizovat pouze na základě uvedení konkrétního čísla řádku, což s sebou přinášelo velké komplikace. Později se objevily příkazy skoku využívající návěstí. Typickými zástupci v této kategorii byly rané verze jazyků FORTRAN a COBOL.

- *Strukturované*

E. Dijkstra upozornil na nebezpečí spjaté s používáním příkazu „GO TO“. Poukázal zejména na to, že při používání příkazu „GO TO“ je prakticky nemožné ladit program, protože struktura programu nedává příliš informací o jeho vykonávání (v nějakém bodě výpočtu). Strukturované procedurální paradigma nahrazuje příkazy skoku pomocí podmíněných cyklů typu „opakuj ~ dokud platí podmínka“ a jiných strukturovaných konstrukcí, které se do sebe „vnořují“. Typickými zástupci programovacích jazyků z této rodiny jsou jazyky C (1978), PASCAL (1970), Modula-2 (1978) a ADA (1977-1983).

**Logické paradigma** Pro logické paradigma je charakteristický jeho *deklarativní charakter*, při psaní programů programátoři spíš popisují problém, než aby vytvářeli předpis, jak problém řešit (to samozřejmě platí jen do omezené míry). Programy v logických programovacích jazycích lze chápat jako *logické teorie* a výpočet lze chápat jako *dedukci důsledků*, které z této teorie plynou. Formálním modelem logického programování jsou speciální inferenční mechanismy. Programy se skládají ze speciálních formulí – tzv. *klaузулí*. Výpočet se v logických jazycích zahajuje zadáním cíle, což je existenční dotaz, pro který buď je nebo není během výpočtu nalezena odpověď. Typickým představitelem logického programovacího jazyka je PROLOG (1972). Logické programovací jazyky rovněž nalezly uplatnění v deduktivních databázích, například jazyk DATALOG (1978).

**Objektové paradigma** Objektové paradigma je založené na myšlence vzájemné *interakce objektů*. Objekty jsou nějaké entity, které mají svůj vnitřní stav, který se během provádění výpočetního procesu může měnit. Objekty spolu komunikují pomocí *zasílání zpráv*. Objektově orientované paradigma není „jen jedno“, ale lze jej rozdělit na řadu podtypů. Ze všech programovacích paradigm má objektová orientace nejvíce modifikací (které jsou leckdy myšlenkově proti sobě). Například při vývoji některých jazyků byla snaha odstranit veškeré rysy připomínající procedurální paradigma: cykly byly nahrazeny „iterátory“, podmíněné příkazy byly nahrazeny *výjimkami*, apod. Jiné jazyky se této striktní představy nedrží. Prvním programovacím jazykem s objektovými rysy byla Simula (1967). Mezi čistě objektové jazyky řadíme např. Smalltalk (1971) a Eiffel (1985). Dalšími zástupci jazyků s objektovými rysy jsou C++, Java, Ruby, Sather, Python, Delphi a C#.

### 3.3 Symbolické výrazy a vyhodnocovací proces jazyka Scheme

**Definice 13** (symbolický výraz).

1. každé číslo je symbolický výraz  
(zapisujeme 12, -10, 2/3, 12.45, 4.32e-20, -5i, 2+3i, apod.)
2. každý symbol je symbolický výraz  
(zapisujeme sqrt, +, quotient, even?, muj-symbol, ++?4tt, apod.)

3. jsou-li  $e_1, e_2, \dots, e_n$  symbolické výrazy (pro  $n \geq 1$ ),  
pak výraz ve tvaru  $(e_1 \ e_2 \ \dots \ e_n)$  je symbolický výraz zvaný seznam

**Definice 14** (elementy jazyka).

1. interní reprezentace každého symbolického výrazu je element jazyka
2. každá primitivní procedura je element jazyka

**Symbolický výraz** Symbolický výraz je čistě syntaktický pojem. Symbolické výrazy jsou zavedeny jako sekvence znaků, splňující podmínky definice ???. Naproti tomu elementy jazyka jsou sémantické pojmy. Elementy jazyka jsou prvky množiny (všech elementů jazyka). Elementy jazyka používáme jako hodnoty, hrají klíčovou roli ve vyhodnocovacím procesu. Role elementů jazyka je úzce svázáná s jejich významem (sémantikou). *Reader* převádí symbolické výrazy na elementy jazyka, které jsou interními reprezentacemi symbolických výrazů. Naopak *printer* slouží k převodu elementů do jejich externí reprezentace.

**Věta 3.** Nechť  $E$  je primitivní procedura a  $E_1, \dots, E_n$  jsou libovolné elementy jazyka. Výsledek aplikace primitivní procedury  $E$  na argumenty  $E_1, \dots, E_n$  v tomto pořadí značíme  $\text{Apply}[E, E_1, \dots, E_n]$ . Pokud je výsledkem této aplikace element  $F$ , píšeme  $\text{Apply}[E, E_1, \dots, E_n] = F$ .

Z matematického hlediska je *Apply* částečné zobrazení z množiny všech konečných posloupností elementů do množiny všech elementů. Pro některé  $n$ -tice elementů však toto zobrazení není definované, proto „částečné“.

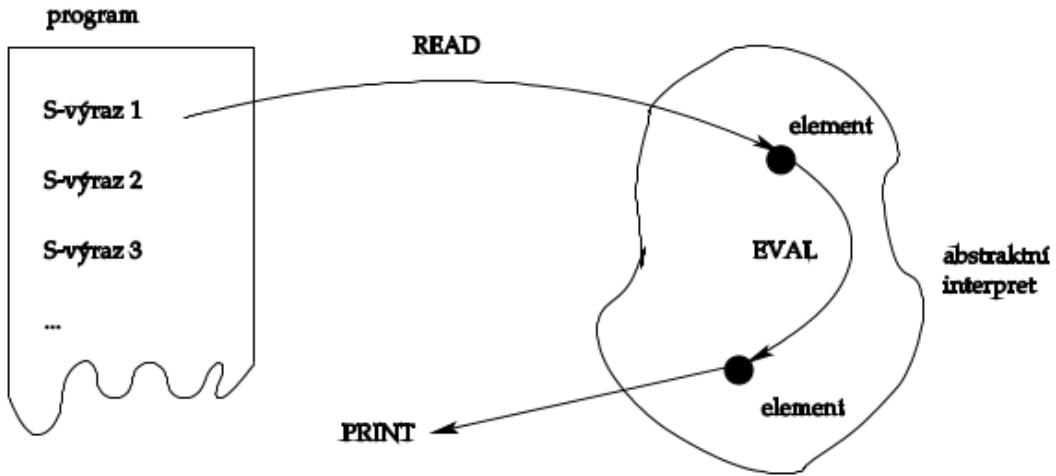
**Definice 15** (abstraktní interpret). Množina všech elementů jazyka spolu s pravidly jejich vyhodnocování nazýváme abstraktní interpret jazyka *Scheme*.

Vyhodnocování programů, které se skládají z posloupnosti symbolických výrazů probíhá v cyklu, kterému říkáme REPL. Jméno cyklu je zkratka pro „Read“, „Eval“, „Print“ a „Loop“, což jsou jednotlivé části cyklu. Budeme-li předpokládat, že máme vstrupní program, jednotlivé fáze cyklu REPL můžeme popsat následovně:

- Read: Pokud je prázdný vstup, činnost interpretu končí. V opačném případě *reader* načte první vstupní symbolický výraz. Pokud by se to nepodařilo v důsledku syntaktické chyby, pak je činnost interpretu ukončena chybovým hlášením „CHYBA: Syntaktická chyba“. Pokud je symbolický výraz úspěšně načten, je převeden do své interní reprezentace. Výsledný element, označíme jej  $E$ , bude zpracován v dalším kroku.
- Eval: Element  $E$  z předchozího kroku je vyhodnocen. Vyhodnocení provádí část interpretu, které říkáme *evaluator*. Pokud během vyhodnocení došlo k chybě, je napsáno chybové hlášení a ukončena činnost interpretu. Pokud vyhodnocení končí úspěchem, pak je výsledkem vyhodnocení element. Označme tento element  $F$ . Výsledný element  $F$  bude zpracován v dalším kroku.
- Print: Provede převod elementu  $F$  z předchozího kroku do jeho externí reprezentace. Tuto část činnosti obstarává *printer*. Výsledná externí reprezentace je zapsána na výstup (vytištěna na obrazovku, zapsána do souboru, apod.).
- Loop: Vstupní symbolický výraz, který byl právě zpracován v krocích „Read“, „Eval“ a „Print“ je odebrán ze vstupu. Dále pokračujeme krokem „Read“.

**Definice 16** (vyhodnocení elementu  $E$ ). Výsledek vyhodnocení elementu  $E$ , značeno  $\text{Eval}[E]$ , je definován:

1. Pokud je  $E$  číslo, pak  $\text{Eval}[E] := E$ .
2. Pokud je  $E$  symbol, mohou nastat dvě situace:
  - (a) Pokud  $E$  má aktuální vazbu  $F$ , pak  $\text{Eval}[E] := F$ .



Obrázek 2: Schéma cyklu REPL

- (b) Pokud  $E$  nemá vazbu, pak ukončíme vyhodnocování hlášením „**CHYBA: Symbol  $E$  nemá vazbu**“.
- 3. Pokud je  $E$  seznam tvaru  $(E_1 \ E_2 \ \dots \ E_n)$ , pak nejprve vyhodnotíme jeho první prvek a výsledek vyhodnocení označíme  $F_1$ , formálně:  $F_1 := \text{Eval}[E_1]$ . Vzhledem k hodnotě  $F_1$  rozlišíme tři situace:
  - (a) Pokud je  $F_1$  procedura, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky  $E_2, \dots, E_n$  seznamu  $E$ , výsledky jejich vyhodnocení označíme  $F_2, \dots, F_n$ . Formálně:
 
$$F_1 := \text{Eval}[E_1],$$

$$F_2 := \text{Eval}[E_2],$$

$$\vdots \quad \vdots$$

$$F_n := \text{Eval}[E_n].$$
 Dále položíme  $\text{Eval}[E] := \text{Apply}[F_1, F_2, \dots, F_n]$ , tedy výsledkem vyhodnocení  $E$  je element vzniklý aplikací procedury  $F_1$  na argumenty  $F_2, \dots, F_n$ .
  - (b) Pokud je  $F_1$  speciální forma, pak  $\text{Eval}[E] := \text{Apply}[F_1, E_2, \dots, E_n]$ .
  - (c) Pokud  $F_1$  není procedura ani speciální forma, skončíme vyhodnocování hlášením „**CHYBA: Nelze provést aplikaci: první prvek seznamu  $E$  se nevyhodnotil na proceduru ani na speciální formu**“.
- 4. Ve všech ostatních případech klademe  $\text{Eval}[E] := E$ .

### 3.4 Vytváření abstrakcí pomocí procedur

**Abstrakce při programování** Tímto obvykle myslíme možnost vytváření obecnějších programových konstrukcí, které je možné používat ke zvýšení efektivity při programování. Efektivitou ale nemáme nutně na mysli třeba jen rychlosť výpočtu, ale třeba provedení programu, které je snadno pochopitelné, rozšířitelné a lze jej podle potřeby upravovat.

**Speciální forma** Speciální formy jsou elementy jazyka, při jejichž aplikaci se nevyhodnocuje zbytek seznamu tak, jako u procedur. Speciální formy jsou aplikovány s argumenty jimiž jsou *elementy v jejich nevyhodnocené podobě* a každá speciální forma si sama určuje jaké argumenty a v jakém pořadí (zdali vůbec) bude vyhodnocovat. Proto je nutné každou speciální formu vždy detailně popsát. Při popisu se na rozdíl od procedur musíme zaměřit i na to, jak speciální forma manipuluje se svými argumenty.

```
(define na2 (lambda (x) (* x x)))
```

Při vyhodnocování předchozího výrazu je aktivována speciální forma **define**, která na symbol **na2** naváže hodnotu vzniklou vyhodnocením druhého argumentu, tím je  $\lambda$ -výraz, který se vyhodnotí na proceduru, takže po vyhodnocení předchozího výrazu bude na **na2** navázána nově vytvořená procedura „umocni hodnotu na druhou“.

Vytváření abstrakcí pomocí procedur tedy spočívá v tom, že *konkrétní části kódu nahrazujeme aplikací abstraktnějších procedur*.

**Definice 17** ( $\lambda$ -výraz). *Každý seznam ve tvaru*

*(lambda (param<sub>1</sub>) (param<sub>2</sub>) ... (param<sub>n</sub>)) (telo))*,

*kde  $n \geq 0$ , (param<sub>1</sub>) (param<sub>2</sub>) ... (param<sub>n</sub>) jsou vzájemně různé symboly a (telo) je libovolný symbolický výraz, se nazývá  $\lambda$ -výraz. Symboly (param<sub>1</sub>) (param<sub>2</sub>) ... (param<sub>n</sub>) se nazývají formální argumenty (někdy též parametry). Číslo n nazýváme počet formálních argumentů (parametrů).*

**lambda** musí být skutečně speciální forma, kdyby byla procedura, skončil by následující výraz chybou, neboť symbol **x** by neměl vazbu.

```
(lambda (x) (* x x))
```

$\lambda$ -výraz je chápán jako *předpis pro vznik procedury*, nelze jej ale ztotožňovat s procedurou samotnou.

```
(lambda (x y nepouzity) (* (+ 1 x) y))
```

- **x, y** – vázané symboly
- **+, \*** – volné symboly

Procedury vznikají *vyhodnocováním  $\lambda$ -výrazů*.

**Definice 18** (speciální forma **define**). *Speciální forma **define** se používá se dvěma argumenty ve tvaru:*

```
(define <jméno> <výraz>)
```

*Při aplikaci speciální forma nejprve ověří zdali je argument <jméno> symbol. Pokud tomu tak není, aplikace speciální formy je ukončena hlášením „**CHYBA: První výraz musí být symbol**“. V opačném případě je vyhodnocen argument <výraz>. Označme F výsledek vyhodnocení tohoto elementu, to jest  $F := \text{Eval}[\text{výraz}]$ . Dále je v prostředí vytvořena nová vazba symbolu <jméno> na element F. Pokud již symbol <jméno> měl vazbu, tato původní vazba je nahrazena elementem F. Ve standardu jazyka Scheme [R6RS] se uvádí, že výsledná hodnota aplikace speciální formy **define** není definovaná.*

### 3.5 Procedury vyšších řádů: aplikace a mapování

**Procedury vyšších řádů** Procedury vyšších řádů jsou takové procedury, jimž jsou při aplikaci předávány další procedury jako argumenty nebo které vracejí procedury jako výsledek své aplikace.

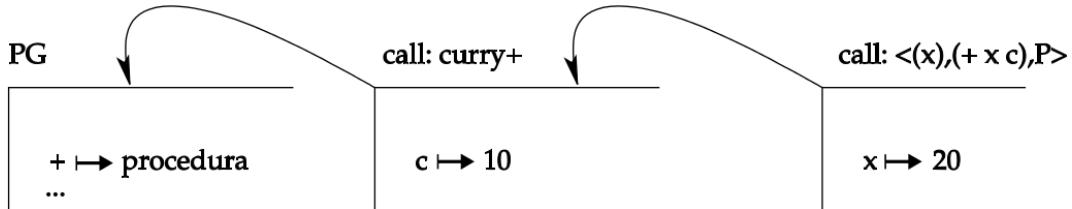
```
(define infix
  (lambda (x operace y)
    (operace x y)))
```

(infix 10 + 20)	⇒	30
(infix 10 - (infix 2 + 5))	⇒	3
(infix 10 (lambda (x y) x) 20)	⇒	10
(infix 10 (lambda (x y) 66) 20)	⇒	66
(infix 10 (lambda (x) 66) 20)	⇒	CHYBA: Chybný počet argumentů při aplikaci
(infix 10 20 30)	⇒	CHYBA: Nelze aplikovat: 20 není procedura

## Program 1.

```
(define curry+
  (lambda (c)
    (lambda (x)
      (+ x c)))))

(define f (curry+ 10))
f
(f 20)           ⇒ „procedura, která hodnotu svého argumentu přičte k hodnotě 10“
                  ⇒ 30
```



Obrázek 3: Vznik prostředí během aplikace procedur

**Mapování** Mapování je činnost, při které se vezme daný seznam, vytvoří se nový seznam stejné délky, jehož prvky jsou nějakým způsobem „změněné“ prvky původního seznamu, výstupem je tedy seznam stejné délky s pozměněnými prvky. Takové činnosti se říká *mapování procedury přes seznam*. K provedení této operace a ke konstrukci nového seznamu ve Scheme slouží procedura `map`. Procedura `map` bere dva argumenty. Prvním z nich je procedura `<proc>` jednoho argumentu. Druhým argumentem je seznam. Proceduru `<proc>` nazýváme *mapovaná procedura*. Aplikace `map` tedy probíhá ve tvaru

`(map <proc> <seznam>)`

Výsledkem této aplikace je seznam výsledků aplikace procedury `<proc>` na každý prvek seznamu `<seznam>`. Označme prvky seznamu `<seznam>` postupně  $\langle prvek_1 \rangle \langle prvek_2 \rangle \dots \langle prvek_n \rangle$ . Aplikací procedury `map` tedy dostaváme seznam

$\text{Apply}(\langle proc \rangle \langle prvek_1 \rangle) \text{ Apply}(\langle proc \rangle \langle prvek_2 \rangle) \dots \text{ Apply}(\langle proc \rangle \langle prvek_n \rangle)$

Proceduru `map` lze použít rovněž s více než se dvěma argumenty. Seznam nemusí být jen jeden, ale může jich být jakýkoli kladný počet  $k$ :

`(map <proc> <seznam1> <seznam2> ... <seznamk>)`

Všechny seznamy  $\langle seznam_i \rangle$  musí mít stejnou délku. Mapovací procedura `<proc>` musí být procedurou  $k$  argumentů. Jinými slovy: musí být procedurou tolik argumentů, kolik je seznamů. Výsledkem je pak seznam výsledků aplikací procedury `<proc>` na ty prvky seznamů, které se v předaných seznamech nacházejí na stejných pozicích.

## 3.6 Seznamy a hierarchická data

**Tečkový pár** Tečkové páry jsou elementy jazyku. Tečkový pár je vytvořen spojením dvou libovolných elementů do uspořádané dvojice. Prvky v páru pak nazýváme *první prvek páru* a *druhý prvek páru*. Pár je zkonstruován primitivní procedurou `cons`, selektory prvků jsou `car` a `cdr`.

Můžeme konstruovat páry jejichž (některé) prvky jsou opět páry, tímto způsobem lze vytvářet libovolně složité hierarchické struktury postupným „vnořováním páru“ do sebe.

Než přikročíme k definici seznamu, potřebujeme zavést nový speciální element jazyka. Tímto elementem je *prázdný seznam* (nazývaný též *nil*). Tento element reprezentuje „seznam neobsahující žádný prvek“. Externí reprezentace prázdného seznamu je  $()$ . Element „prázdný seznam“ se *vyhodnocuje sám na sebe*. Tedy:

$$() \Rightarrow ()$$

**Poznámka 1.** V některých interpretech se prázdný seznam nevyhodnocuje sám na sebe a vyhodnocení  $()$  skončí chybou „*CHYBA: Pokus o vyhodnocení prázdného seznamu*“. Element prázdný seznam můžeme v takových interpretech získat pomocí kvotování tím, že potlačíme vyhodnocení výrazu  $()$ , viz příklady:

$$\begin{aligned} (\text{quote } ()) &\Rightarrow () \\ '() &\Rightarrow () \end{aligned}$$

**Definice 19.** Seznam je každý element  $L$  jazyka Scheme splňující právě jednu z následujících podmínek:

1.  $L$  je prázdný seznam (to jest  $L$  je vzniklý vyhodnocením  $'()$ ), nebo
2.  $L$  je pár ve tvaru  $(E . L')$ , kde  $E$  je libovolný element a  $L'$  je seznam. V tomto případě se element  $E$  nazývá hlava seznamu  $L$  a seznam  $L'$  se nazývá tělo seznamu  $L$  (řidčeji též ocas seznamu  $L$ ).

**Hiearchická data** Hierarchická data jsou taková data, ve kterých potřebujeme uchovávat více hodnot. Mějme například zlomek, můžeme mít zvlášť proceduru pro výpočet čitatele a zvlášť proceduru pro výčet jmenovatele, to je ale potencionálním zdrojem chyb, navíc nám bobtná kód a stává se nepřehledným. Zlomek tedy můžeme uchovávat jako tečkový pár. Dalším příkladem může být bod (uspořádaná dvojice hodnot), úsečka (dvojice bodů), kružnice (dvojice bodu a čísla), apod.

### 3.7 Indukce a rekurze: princip a příklady

**Rekurze** Rekurze ve smyslu *metody definice funkcí a procedur*. Rekurzivní procedura je procedura, která ve svém těle provádí aplikaci sebe sama. Rekurzí lze vyřešit mnoho problémů elegantně a efektivně.

Rekurzi můžeme využít například pro:

- výpočet faktoriálu:

$$n! = \begin{cases} 1 & \text{pokud } n \leq 1, \\ n \cdot (n-1)! & \text{jinak.} \end{cases}$$

- výpočet Fibonacciho čísel:

$$F_n = \begin{cases} n & \text{pokud } n \leq 1, \\ F_{n-1} + F_{n-2} & \text{jinak.} \end{cases}$$

**Indukce** Indukci používáme jako obecný dokazovací princip, jímž jsme schopni dokazovat vlastnosti rekurzivně definovaných funkcí a procedur.

U indukce používáme následující značení:

- $P$  - vlastnost (prvočíslo, sudé, ...)
- $P(n)$  - vlastnost čísla  $n$  („ $n$  je prvočíslo“, „ $n$  je sudé“, ...)

**Věta 4** (princip indukce přes přirozená čísla). *K tomu, abychom ověřili, že vlastnost  $P$  platí pro každé  $n \in \mathbb{N}_0$ , stačí prokázat platnost následujících dvou bodů:*

1. platí  $P(0)$
2. pokud platí  $P(i)$ , pak platí  $P(i+1)$ .

## 3.8 Typy rekurzivních výpočetních procesů

**Lineárně rekurzivní výpočetní proces** Rekurzivní proces, který má netriviální fáze navíjení a odvíjení. Během fáze navíjení je budována „série odložených výpočtů“, přitom počet prostředí roste lineárně vzhledem k velikosti vstupních argumentů. Po dosažení limitní podmínky nastává fáze odvíjení, ve které je zpětně dokončeno vyhodnocování výrazů, které bylo započato a „odloženo“ ve fázi navíjení. Činnost lineárně rekurzivního výpočetního procesu nelze jednoduše přerušit, ani není možné „skočit“ na nějaké místo rekurze. Níže uvedená procedura generuje lineárně rekurzivní výpočetní proces.

```
(define fac
  (lambda (n)
    (if (<= n 1)
        1
        (* n (fac (- n 1))))))
```

**Stromově rekurzivní výpočetní proces** Výpočetní proces, který svým průběhem připomíná lineární výpočetní proces, jen s tím rozdílem, že počet aplikací rekurze neroste lineárně s velikostí vstupu. Stromově rekurzivní proces je typicky generován procedurami, které ve svých rekurzivních předpisech vytvárají dvě nebo více aplikací sebe sama. Pro stromově rekurzivní výpočetní procesy je charakteristické postupné střídání fází navíjení a odvíjení. Níže uvedená procedura generuje stromově rekurzivní výpočetní proces.

```
(define fib
  (lambda (n)
    (if (<= n 1)
        n
        (+ (fib (- n 1))
            (fib (- n 2)))))))
```

**Lineárně iterativní výpočetní proces** Výpočetní proces, který je generovaný koncově rekurzivními procedurami. Nedochází při něm k vytváření „odložených výpočtů“. Každý lineárně iterativní výpočetní proces je během své činnosti jednoznačně určen:

1. vazbami symbolů v prostředí  $\mathcal{P}$  svého běhu,
2. předisem, jak změnit stav vazeb v  $\mathcal{P}$  na základě aktuálních vazeb,
3. limitní podmínkou ukončující iterativní proces.

Lineárně iterativní výpočetní proces může být během svého výpočtu přerušen a posléze oět obnoven v místě přerušení. Fáze průběhu lineárně iterativního procesu je dána vazbami symbolů v prostředí jeho běhu, viz 1. bod předchozího výpisu. Pokud tyto vazby uchováme a lineárně iterativní proces opustíme, je možné jej opět aktivovat s počátečními hodnotami nastavenými na uschované hodnoty. Tím pádem se iterativní proces „rozběhne“ od místa zastavení.

## 3.9 Lexikální a dynamický rozsah platnosti

Rozsah platnosti spočívá v hledání vazeb symbolů, v mnoha jazycích se místo pojmu „symbol“ používá pojem „proměnná“. Mějme proceduru `curry+ (??)`.

```
(define c 100)
(define f (curry+ 10))
(f 20)
```

**Lexikální rozsah platnosti** U lexikálního rozsahu platnosti se vazba symbolů (které nejsou nalezeny v lokálním prostředí) hledá v lexikálně nadřazených prostředích. To znamená v prostředí vzniku procedury. Používá většina programovacích jazyků. Výsledkem (`f 20`) bude 30.

**Dynamický rozsah platnosti** Vazba symbolů se hledá v prostředí aplikace procedury. V současnosti je téměř nepoužívaný. Výsledkem (`f 20`) je 120.

## 3.10 Vlastnosti typových systémů

Jazyky dělíme podle několika vlastností.

### Kdy je možná kontrola typů

**Staticky typované jazyky** Možná kontrola typů již před interpretací nebo během překladu programu, pouze na základě znalosti jeho syntaktické struktury.

**Dynamicky typované jazyky** Pouhá struktura kódu (vždy) nestačí ke kontrole typů a typy elementů musí být kontrolovány až za běhu programu. Je rovněž možné, že jedno jméno (symbol nebo proměnná) může během života programu nést hodnoty různých typů.

### Síla

**Silně typované jazyky** Pro každou operaci má přesně vymezený datový typ argumentů. Pokud je operace použita s jiným typem argumentů, dochází k chybě.

**Slabě typované jazyky** Mají definovanou sadu konverzních pravidel, která umožňují, pokud je to nutné, převádět mezi sebou data různých typů. Pokud je provedena operace s argumenty, které jí typově neodpovídají, je provedena konverze typů tak, aby byla operace proveditelná.

### Bezpečnost

**Bezpečně typované jazyky** Tyto jazyky se chovají korektně z hlediska provádění operací mezi různými typy elementů. To znamená, že pokud je operace pro daný typ elementů proveditelná, její provedení nemůžezpůsobit havárii programu.

**Nebezpečně typované jazyky** Výsledek operace mezi různými typy může vést k chybě při běhu programu. Takovým jazykem je například C, chyba tohoto typu může být způsobena např. přetečením paměti nebo dereferencí ukazatele ukazujícího na místo paměti nepatřící programu.

## 4 Makra

**Motivace** Chceme upravit *if* tak aby při absenci alternativního výrazu vracel  $\#f$ .

Nyní máme (*i f* (= 1 2) 'blah)  $\Rightarrow$  nedefinovaná hodnota.

Chceme: (*new - i f* (= 1 2) 'blah)  $\Rightarrow \#f$ .

Potřebujeme zavést předpis, který bude provádět "transformaci kódu". Spoustu možností jak vyřešit (nová transformační procedura nebo třeba pomocí kvazikvótování). Nejlepší řešení  $\rightarrow$  zavedení [maker](#).

### 4.1 Dva základní pohledy na makra

#### 4.1.1 První pohled

Makra jsou "rozšířením syntaxe jazyka"

**Definice 20.** *makro = dáno definicí svého transformačního předpisu*

- po načtení výrazu (READ) je v něm provedena makroexpanze. Tuto fázi provádí tzv. *preprocesor*
- až po dokončení expanze všech maker nastává vyhodnocování výrazu
- nemá smysl uvažovat pojmem "aplikace makra"
- takto na makra pohlíží většina PJ: C, DrScheme, Common LISP, ...

Výhody přístupu:

- preprocesor a vlastní *eval* jsou zcela nezávislé
- preprocesor může být aktivován okamžitě po načtení výrazu
- umožňuje snadnou komplikaci kódu (v komplikovaném kódu již pochopitelně "žádná makra nejsou")

Nevýhody přístupu:

- makra jsou "mimo jazyk" (často se zapisují odlišně, třeba v C)
- makra nejsou elementy prvního řádu

#### 4.1.2 Druhý pohled

Makra jsou "speciální elementy jazyka"

**Definice 21.** *makro = element jazyka obsahující ukazatel na transformační proceduru*

- transformační procedura ... klasická procedura
- je potřeba rozšířit eval: případ, kdy se první prvek seznamu vyhodnotí na makro
- makra jsou "uživatelsky definované speciální formy"
- takto na makra budeme pohlížet my (dále třeba PJ: M4, T<sub>EX</sub>)

Výhody přístupu:

- makra jsou elementy prvního řádu
- s makry lze pracovat "jako s daty", mohou dynamicky vznikat/zanikat za běhu programu
- můžeme uvažovat koncept "anonýmního makra"

Nevýhody přístupu:

- k makroexpanzi dochází až při činnosti eval
- prakticky znemožňuje účinnou komplikaci kódu
- při neuváženém používání maker komplikuje ladění programu

## 4.2 Hygienická makra

"hygienická" ... umožňuje vytvářet bezpečná makra

Základní rysy:

- definována v R5RS (kromě Scheme, pokud vím, nikdo nemá)
- kompletně jiný přístup k makrům než *define-macro*
- makra jsou definována pomocí (několika) přepisovacích pravidel

Výhody:

- prakticky odpadají složitě kvazikvotované výrazy
- nemůže nastat *symbol capture*
- makra jsou v souladu s lexikálním rozsahem platnosti
- makra lze definovat lokálně

Nevýhody:

- některá makra se tímto způsobem nedělají pohodlně

## 4.3 Použití

### 4.3.1 Normální makra

#### Motivační příklady definice maker

```
(define-macro new-if
  (lambda (test expr . alt)
    (list 'if test expr
          (if (null? alt)
              #'f
              (car alt)))))

(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    (list 'if <test> <expr>
          (if (null? <alt>)
              #'f
              (car <alt>)))))
```

Vilém Vychodil (KI, UP Olomouc) PP 2A, Lekce 3 Makra I 11 / 51

#### Motivační příklady definice maker

```
; new-if: pomocí kvazikvotování
(define-macro new-if
  (lambda (test expr . alt)
    `(#f ,test
      ,expr
      (begin #f ,@alt)))))

; new-if: pomocí kvazikvotování
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    `(#f ,<test>
      ,<expr>
      (begin #f ,@<alt>))))
```

Vilém Vychodil (KI, UP Olomouc) PP 2A, Lekce 3 Makra I 12 / 51

#### Příklad použití makra

```
; new-if: pomocí kvazikvotování
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    `(#f ,<test>
      ,<expr>
      (begin #f ,@<alt>)))

(let ((x 10))
  (new-if (even? x) (+ x 1)))
  ↓ aktivace transformační procedury makra
(if (even? x) (+ x 1) (begin #f))
  ↓ vyhodnocení výrazu v prostředí, kde x má vazbu 10
```

11

Vilém Vychodil (KI, UP Olomouc) PP 2A, Lekce 3 Makra I 13 / 51

### 4.3.2 Hygienická makra

#### Vytvoření hygienického makra

- (define-syntax <název> <transformační-procedura>)
- <transformační-procedura> vzniká pomocí spec. formy syntax-rules

#### Vytvoření transformační procedury hygienického makra

- (syntax-rules <klicová-slova> <pravidlo<sub>1</sub>> <pravidlo<sub>2</sub>> ...)
- <klicová-slova> ... seznam symbolů, které jsou dále chápány jako klíčová slova (seznam může být prázdný)
- <pravidlo<sub>n</sub>> ... přepisovací pravidla, viz dále

Přepisovací pravidla jsou pravidla tvaru (<vzor> <nahrazní>), kde

- <vzor> je výraz specifikující konkrétní případ použití makra, viz dále
- <nahrazní> je libovolný výraz, kterým bude „volání makra“ nahrazeno v případě shody s daným vzorem



Vilém Vychodil (KI, UP Olomouc) PP 2A, Lekce 4 Makra II 31 / 41

Makro `and` realizované jako hygienické makro

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...) ; dva a více argumentů
     (if test1 (and test2 ...) #f))))
```

Makro `setf!` (v tomto případě slouží `car`, `cdr` a `ref` jako klíčová slova

```
(define-syntax setf!
  (syntax-rules (car cdr ref)
    ((setf! (car pair) value) (set-car! pair value))
    ((setf! (cdr pair) value) (set-cdr! pair value))
    ((setf! (ref vector index) value)
     (vector-set! vector index value))
    ((setf! symbol value) (set! symbol value))))
```



Vilém Vychodil (KI, UP Olomouc) PP 2A, Lekce 4 Makra II 33 / 41

## 5 Líné vyhodnocování, přísliby a proudy

### 5.1 Přísliby a líné vyhodnocování

Základní myšlenka:

- místo vyhodnocení daného výrazu pracujeme s *příslibem* jeho budoucího vyhodnocení
- *příslib* = nový typ elementu (element prvního řádu)
- speciální forma `delay`, která pro daný výraz vrací příslib jeho vyhodnocení
- procedura `force`, která pro daný příslib aktivuje výpočet a vrátí hodnotu vzniklou vyhodnocením příslíbeného výrazu

Líné vyhodnocování = vyhodnocování založené na příslibech (někdy taky "call by need")

**Poznámka.**

- Při líném vyhodnocování dochází k propagaci chyb (chyba se projeví "na jiném místě" než "kde vznikla").
- pomocí příslibů je možné "odložit časově složitý výpočet na později" a aktivovat jej, až je skutečně potřeba jej provést.

### 5.2 Proud (Streams)

- proudy jsou nejčastěji používanou aplikací líného vyhodnocování
- neformálně: proudy jsou líně vyhodnocované seznamy
- konstruktor *cons-stream* a selektory *stream-car* a *stream-cdr*
- prázdný seznam je proud
- každý tečkový pár (*e . p*), kde *e* je libovolný element a *p* je příslib proutu, je proud.
- výpočet je řízen daty, dochází k propagaci chyb

Výsledek je vrácen okamžitě

```
(define fs (stream-map fib (stream 1 ... 30 31 ... 50)))
fs => (1 . #<promise>)
přístup ke dalším prvkům se bude postupně zpomalovat
```

ukázka propagace chyb v proudech:

```
(define s (stream 1 2 3 4 5 'blah 6 7))
(define r (stream-map - s))
r => (-1 . #<promise>)
(display-stream r 4) => (-1 -2 -3 -4)
(display-stream r 6) => (-1 -2 -3 -4 -5 CHYBA)
```

### 5.2.1 Nekonečné proudy

- neformálně: "potenciálně nekonečná lineární datová struktura"
- v každém okamžiku průchodu nekonečným proudem máme vždy k dispozici aktuální prvek a příslib
- v praxi se konstruuje rekurzivní procedurou bez limitní podmínky

#### Nekonečné proudy

##### Formálně lze zavést jako limity prefixových generátorů

Seznam  $r$  je **prefix** seznamu  $t$ , pokud lze  $t$  vyjádřit jako spojení  $r$  s nějakým seznamem  $l$  (v tomto pořadí).

Množinu seznamů  $\mathcal{S}$  nazveme **prefixový generátor**, pokud

- ① pro každé  $n \in \mathbb{N}$ , systém  $\mathcal{S}$  obsahuje seznam délky  $n$ ;
- ② pro každé dva  $s, t \in \mathcal{S}$  platí: buď  $s$  je prefix  $t$ , nebo  $t$  je prefix  $s$ .

**Nekonečný proud** (příslušný prefixovému generátoru  $\mathcal{S}$ ) je element reprezentující posloupnost  $(e_i)_{i=0}^{\infty}$ , kde  $e_i$  je element nacházející se na  $i$ -té pozici libovolného seznamu  $s \in \mathcal{S}$  majícího alespoň  $i + 1$  prvků.

```
; ; rekurzivní procedura bez limitní podmínky
(define ones-proc
  (lambda ()
    (cons-stream 1 (ones-proc))))
; ; proud hodnot  $(2^i)_{i=0}^{\infty}$ 
(define pow2
  (let next ((last 1))
    (cons-stream last
      (next (* 2 last)))))
```

## 6 Aktuální pokračování a únikové funkce

### Motivace

Během posledních dvou semestrů jsme ukázali, že s procedurami a makry je možné pracovat jako s daty. Dále jsme ukázali, že výpočet lze řídit daty (streams). Nyní ukážeme, že s *výpočetním časem, historií a budoucností výpočtu* je možné pracovat jako s daty. Motto:

### Budeme potřebovat tři fundamentální pojmy

- ① *Kontext* – procedura jednoho argumentu reprezentující výpočet, který nastane okamžitě po vyhodnocení jistého výrazu
- ② *Úniková procedura* – procedura, po jejíž aplikaci se ukončí zbylý výpočet a jako výsledek je okamžitě vrácena hodnota její aplikace
- ③ *Aktuální pokračování* neboli *kontinuace* – je úniková procedura vytvořená z kontextu aktuálního výrazu

### Motivace

O co jde?

- aktuální pokračování – analogie „příkazu skoku“
- oproti skoku je však **MNOHEM** mocnější
- umožní nám vytvářet množství typů nelokálních skoků
- umožní (do jisté míry) zhmatnit budoucnost výpočtu a manipulovat s ní jako s daty (například ji vyvolat v okamžiku, kdy už „budoucí výpočet“ proběhl – v jistém smyslu se tedy vrátíme do minulosti).
- časem vytvoříme
  - *korutiny* – podprogramy, které se budou vzájemně přepínat
  - *paralelní systém* – současný běh několika výpočtů
  - *nedeterministické operátory* – povede na programy, které budou schopny samy hledat „řešení problému“ pouze na základě jeho popisu
  - a mnohem víc…
- aktuální pokračování – doména jazyka Scheme  
(ostatní PJ mají jen „trapné náhražky“, třeba standard POSIX definuje `longjmp...` slabý odvar)

## POJEM 1: Kontext

Neformálně: „Kontext je zhmotnění zbytku výpočtu, který by byl zahájen okamžitě po vyhodnocení nějakého podvýrazu“

### Příklady:

kontext podvýrazu  $(/ 25 \times)$  ve výrazu  $(+ 2 (* 3 (/ 25 \times)))$  je výpočet, který proběhne po vyhodnocení  $(/ 25 \times)$  v  $(+ 2 (* 3 (/ 25 \times)))$ , to jest: hodnota vzniklá vyhodnocením  $(/ 25 \times)$  bude násobena číslem 3 a tento výsledek bude přičten k 2.

kontext podvýrazu  $*$  ve výrazu  $(+ 2 (* 3 (/ 25 \times)))$  je výpočet, který nejprve otestuje, zda-li je výsledná hodnota (vzniklá vyhodnocením  $*$ ) procedura, pokud ne, končí se chybou; pokud ano, je tato procedura aplikována na 3 a výsledek vyhodnocení  $(/ 25 \times)$ ; k výsledku aplikace je poté ještě přičteno 2.

### Ještě něco o kontextu

**Pojem:** program s dírou (hole „□“)

díra představuje (nespecifikovaný) výraz, o jehož kontextu se zajímáme

```
(+ 2 (* 3 (/ 25 5)))
(+ 2 (* 3 □))
(+ 2 (* □ (/ 25 5)))
(+ 2 (□ 3 (/ 25 5)))
```

**Formálně:** Kontext je **procedura jednoho argumentu** reprezentující výpočet, který by nastal po dosazení skutečné hodnoty místo díry

Pro předchozí příklady si lze představit jako procedury vzniklé vyhodnocením následujících  $\lambda$ -výrazů

```
(lambda (□) (+ 2 (* 3 □)))
(lambda (□) (+ 2 (* □ (/ 25 5))))
(lambda (□) (+ 2 (□ 3 (/ 25 5))))
```

## POJEM 2: Úniková procedura

Procedura se nazývá *úniková procedura*, pokud je-li aktivována, tak způsobí okamžité přerušení vykonávání zbytku výpočtu a výsledkem vyhodnocení (celého vstupního výrazu, ve kterém byla použita) je právě výsledek její aplikace

Budeme předpokládat, že máme k dispozici proceduru *escaper*, která pro danou proceduru vrátí tutéž proceduru, která ale bude úniková (posléze ukážeme, že *escaper* lze ve Scheme naprogramovat)

\*  $\Rightarrow$  procedura násobení  
(*escaper* \*)  $\Rightarrow$  úniková procedura násobení

Použití na top-level je stejné

(\* 10 20)  $\Rightarrow$  200  
((*escaper* \*) 10 20)  $\Rightarrow$  200, avšak:

(+ 2 (\* 10 20))  $\Rightarrow$  202  
(+ 2 ((*escaper* \*) 10 20))  $\Rightarrow$  200

Další příklad únikové funkce

```
(define p (escaper
  (lambda (x)
    (if (even? x)
        x
        (- x)))))
```

příklad použití:

```
(p 10)  $\Rightarrow$  10
(p 11)  $\Rightarrow$  -11

(define p (escaper
  (lambda (x)
    (if (even? x)
        x
        (- x)))))
```

```
(define test
  (lambda (x)
    (if (= 1 (p x))
        tohle-nikdy-neproběhne
        tohle-taky-ne)))
```

Demonstrace toho, že *if* nikdy neproběhne

```
(test 10)  $\Rightarrow$  10
(test 11)  $\Rightarrow$  -11
(eval `(test 10))  $\Rightarrow$  10
(eval `(test (+ 5 6)))  $\Rightarrow$  -11
```

### POJEM 3: Aktuální pokračování (kontinuace)

Jazyk Scheme dává k dispozici proceduru `call/cc`, použití:

`(call/cc <receiver>),`

kde `<receiver>` (příjemce) musí být *procedura jednoho argumentu*.  
(`call/cc` je zkratka pro `call-with-current-continuation`)

**Při volání (`call/cc <receiver>`) se provede:**

- ① Vytvoří se kontext `(call/cc <receiver>)` v aktuálně vyhodnocovaném výrazu
- ② `<receiver>` je zavolán s argumentem (který nazýváme **aktuální pokračování** neboli **kontinuace**) jímž je procedura vzniklá vyhodnocením `(escaper <kontext>)`

Takže při použití

```
... (call/cc  
     (lambda (f)  
     ...  
     (f ...)  
     ...)) ...
```

bude uvnitř na `f` navázána *úniková procedura*.

Tuto únikovou proceduru můžeme aktivovat s jedním argumentem.

Pokud se tak stane, je okamžitě přerušeno vyhodnocování dalšího kódu v receiveru a bude se pokračovat vyhodnocováním kontextu `(call/cc ...)` s hodnotou se kterou bylo aplikováno `f`.

```
(* 2 (+ 3 (/ 25 5))) => 16
```

Takže například (`f` není použito)

```
(* 2 (call/cc  
      (lambda (f)  
      (+ 3 (/ 25 5))))) => 16
```

předchozí si lze představit:

```
(* 2 ((lambda (f)  
        (+ 3 (/ 25 5)))  
        (escaper (lambda (□) (* 2 □))))))
```

```

(+ 1 (call/cc (lambda (f) 2)))  => 3
(+ 1 (call/cc (lambda (f) (f 2))))  => 3
(+ 1 (call/cc (lambda (f)
                     (if (even? (f 2)) 100 200))))  => 3
(+ 1 (call/cc
          (lambda (f)
            (* 2 (call/cc
                      (lambda (g)
                        (f (g 20)))))))  => 41
(+ 1 (call/cc
          (lambda (f)
            (* 2 (call/cc
                      (lambda (g)
                        (g (f 20)))))))  => 21

```

### Aplikace

- okamžité opuštění rekurze (výskok z rekurze)

Modelový příklad

```

;; procedura realizující součin čísel v seznamu
(define list-product
  (lambda (l)
    (if (null? l)
        1
        (* (car l) (list-product (cdr l)))))

(list-product '(1 2 3 4 5))  => 120

```

- chceme zefektivnit výpočet
- násobení čehokoliv nulou je nula

přidání limitní podmínky pro nulu

má nevýhodu, fáze *odvýjení* bude pořád probíhat

```

(define list-product
  (lambda (l)
    (cond ((null? l) 1)
          ((= (car l) 0) 0)
          (else (* (car l) (list-product (cdr l))))))


```

*;; rekurzivní verze procedury používající call/cc*

```

(define list-product
  (lambda (l)
    (call/cc
      (lambda (exit)
        (let proc ((l l))
          (cond ((null? l) 1)
                ((= (car l) 0) (exit 0))
                (else (* (car l) (proc (cdr l)))))))))

```

## 7 Zásobníkový model vyhodnocování programů (totálně nemám tušení co to je za hrůzu :D )

### Implementace `call/cc`

#### Ukázali jsme

- `call/cc` a jeho aplikace pro řízení výpočtu  
(řízení cyklů, korutiny, nedeterminismus, paralelismus)

#### Všimli jsme si:

- `call/cc` je silný, ale nebezpečný, nástroj  
(je třeba používat obezřetně, pokud vůbec)

#### Přirozená otázka:

- Jak těžké je implementovat `call/cc`?  
(odpověď: uvidíme na této přednášce)

#### Základní myšlenka implementace:

- Vyhodnocovací proces musíme umět uchopit tak, abychom se na něj mohli dívat jako na posloupnost elementárních kroků vyhodnocení (povede na vytvoření **iterativní verze procedury „Eval“**)

#### Rekurzivní vs. iterativní „Eval“

- současná verze „Eval“ je (stromově) rekurzivní
- nejprve vytvoříme interpret jehož „Eval“ bude iterativní
- místo rekurzivního volání „Eval“ budeme používat *iterativní volání* spojené s manipulací s *dodatečnými zásobníky*

#### Motivační příklad: počet atomů v seznamu

```
; ; počet atomů v seznamu (rekurzivní verze)
(define atoms
  (lambda (l)
    (cond ((null? l) 0)
          ((pair? (car l))
           (+ (atoms (car l))
              (atoms (cdr l))))
          (else (+ 1 (atoms (cdr l)))))))
```

```

; ; počet atomů v seznamu (iterativní verze)
(define atoms
  (lambda (l)
    (let iter ((l (list l))
              (accum 0))
      (cond ((null? l) accum)
            ((null? (car l)) (iter (cdr l) accum))
            ((pair? (car l))
             (iter (apply list
                           (caar l)
                           (cdar l)
                           (cdr l)))
                           accum))
            (else (iter (cdr l) (+ accum 1)))))))
(atoms '((a (b)) (c) d))
[((a (b)) (c) d)] | 0
[(a (b)) ((c) d)] | 0
[a ((b)) ((c) d)] | 0
[((b)) ((c) d)] | 1
[(b) () ((c) d)] | 1
[b () () ((c) d)] | 1
[() () ((c) d)] | 2
[() ((c) d)] | 2
[((c) d)] | 2
[(c) (d)] | 2
[c () (d)] | 2
[() (d)] | 3
[(d)] | 3
[d ()] | 3
[()] | 4
[] | 4
⇒ 4

```

### Motivační příklad vyhodnocování

Předpokládáme, že máme **dva zásobníky**:

- ① **EXEC** (Execute Stack): zásobník elementů čekajících na vyhodnocení
- ② **RSLT** (Result Stack): zásobník výsledků vyhodnocení

1. EXEC: (\* 20 70)]  
RSLT: ]
2. EXEC: \*, #<inspect>, (20 70)]  
RSLT: ]
3. EXEC: #<inspect>, (20 70)]  
RSLT: #<primitive-procedure>]
4. EXEC: 20, 70, #<run>, 2]  
RSLT: #<primitive-procedure>]
5. EXEC: 70, #<run>, 2]  
RSLT: 20, #<primitive-procedure>]
6. EXEC: #<run>, 2]  
RSLT: 70, 20, #<primitive-procedure>]
7. EXEC: ]  
RSLT: 1400]

### Implementace vyhodnocování pomocí zásobníků

- je implementovaná (téměř tak), jak jsme naznačili v příkladu
- navíc: prostředí, uživatelsky def. procedury, makra, etc.

### Zavedeme nové speciální elementy

název	symbol s vazbou elementu a popis
#<inspect>	the-inspect rozlišení akcí podle 1. prvku
#<run>	the-run aplikace
#<envrun>	the-run aplikace s daným prostředím
#<return>	the-return vrácení z RSLT zpět na EXEC
#<return-on-true>	the-rottrue podmíněné vrácení kvůli <i>if</i>
#<discard>	the-discard zrušení jednoho prvku z RSLT
#<define-assign>	the-defasgn kvůli <i>define</i>
#<set-assign>	the-setasgn kvůli <i>set!</i>

### Pravidlo 0 (počátek vyhodnocování elementu v prostředí)

```
EXEC: element, #<env>]
RSLT: ]
```

### Pravidlo 1 (symbol se vyhodnotí na svou vazbu v daném prostředí)

```
EXEC: lambda, #<env>]
RSLT: ]
↓
EXEC: ]
RSLT: #<special-form>]
```

### Implementace pravidla 1 v našem interpretu

```
(define rule-eval-symbol?
  (lambda (*EXEC* *RSLT*)
    (and (scm-symbol? (car *EXEC*))
         (scm-env? (cadr *EXEC*)))))

(define rule-eval-symbol
  (lambda ()
    (set! *RSLT*
      (cons
        (let ((binding (lookup-env
                        (cadr *EXEC*)
                        (car *EXEC*) #t #f)))
          (if binding
              (pair-cdr binding)
              (error "EVAL: Symbol not bound")))
        *RSLT*))
    (set! *EXEC* (caddr *EXEC*))))
```

**Pravidlo 2** (seznam se rozdělí na první prvek, inspektor a zbytek)

```
EXEC: (f arg1 arg2 ... argn), #<env>]  
RSLT: ]  
↓  
EXEC: f, #<env>, #<inspect>, (arg1 arg2 ... argn), #<env>]  
RSLT: ]
```

**Pravidlo 3** (na vrcholu EXEC je #<inspect>)

Je-li na vrcholu RSLT procedura (primitivní/uživatelská):

```
EXEC: #<inspect>, (arg1 ... argn), #<env>]  
RSLT: #<procedure>]  
↓  
EXEC: arg1, #<env> ... argn, #<env>, #<run>, n, #<env>]  
RSLT: #<procedure>]
```

**Pravidlo 3** (na vrcholu EXEC je #<inspect>)

Je-li na vrcholu RSLT forma (speciální forma/makro):

```
EXEC: #<inspect>, (arg1 ... argn), #<env>]  
RSLT: #<form>]  
↓  
EXEC: #<run>, n, #<env>]  
RSLT: arg1, ..., argn, #<form>]
```

**Pravidlo 4** (na vrcholu EXEC je #<run>)

nej složitější pravidlo, připravuje aplikaci  
rozlišuje se primitivní procedura / uživatelská procedura / speciální forma  
/ makro / **manipulující procedura** / **manipulující forma**  
(poslední dvě jsou novinky)

**Pravidlo 4** (na vrcholu EXEC je #<run>)

Případ: aplikace primitivní procedury (spec. formy – analogické)

```
EXEC: #<run>, n, #<env>]  
RSLT: res1 ... resn, #<primitive-procedure>]  
↓  
EXEC: ]  
RSLT: výsledek aplikace]
```

Případ: aplikace uživatelské procedury

```
EXEC: #<run>, n, #<env>]  
RSLT: arg1 ... argn, #<user-defined-procedure>]  
↓  
EXEC: tělo procedury, #<env: nové prostředí s vazbami>]  
RSLT: ]
```

Případ: aplikace makra

```
EXEC: #<run>, n, #<env>]  
RSLT: arg1 ... argn, #<macro>]  
↓  
EXEC: #<run>, n, #<env>, #<return>, #<env>]  
RSLT: argn ... arg1, #<user-defined-procedure>]
```

Vazba na pravidlo #<return>:

```
EXEC: #<return>]  
RSLT: element]  
↓  
EXEC: element]  
RSLT: ]
```

### Manipulující procedure/formy

- jsou procedure/formy, které mají dva argumenty jimiž jsou samotné zásobníky EXEC a RSLT
- výsledek aktivace manipulující procedure/formy je dvouprvkový seznam (novy-EXEC novy-RSLT)
- to jest manipulující procedure/formy mají možnost přímo manipulovat s obsahem obou zásobníků a tím pádem řídit průběh výpočtu (vymyslel V. Vychodil, Palacký University :-))

**Všimněte si:** únikové procedure lze chápat jako speciální manipulující procedure, které „vyklidí obsah zásobníků a na RSLT zapíšou výsledek“

K tomu abychom mohli dokončit interpret potřebujeme naprogramovat základní formy: `if`, `begin`, `define`, `set!` a některé procedure jako `apply`, `append` (viz zdroják).

### Ilustrativní příklad implementace `if`

```
EXEC: (if test expr alt), #<env>]  
RSLT: ]  
:  
EXEC: #<run>, 3, #<env>]  
RSLT: test, expr, alt, #<manipulator-if>]  
↓  
EXEC: test, #<env>, #<return-on-true>, #<env>]  
RSLT: expr, alt]  
:  
EXEC: #<return-on-true>, #<env>]  
RSLT: cokoliv nebo #f, expr, alt]  
↓  
EXEC: expr nebo alt, #<env>]  
RSLT: ]
```

## 8 Objektové programování

**Objektové programování:** třídy a objekty, zprávy a metody. Zapouzdření, polymorfismus, dědičnost. Metody objektů, jejich typy, způsoby ochrany. Metody tříd, abstraktní třídy. Vícenásobná dědičnost, rozhraní. Příklady objektově orientovaných jazyků a jejich rysy.

### 8.1 Třídy a objekty, zprávy a metody

#### 8.1.1 Třída

Třída představuje předpis, na jehož základě budou vytvářeny objekty. Deklaruje atributy a definuje chování objektů dané třídy.

#### 8.1.2 Objekt

Konkrétní instance dané třídy, nacházející se v daném stavu (nabývající konkrétní hodnoty) a poskytující rozhraní definované třídou.

#### 8.1.3 Zprávy a metody

Zprávy a metody spolu úzce souvisejí. Metody vykonávají v rámci třídy (objektu) určitou funkcionality. Zprávy slouží pro komunikaci mezi objekty. V různých programovacích jazycích je tato funkcionality implementována různě. Například v Javě se posílají zprávy voláním veřejných metod, C# či Common Lisp využívají svůj systém posílání zpráv.

### 8.2 Zapouzdření, polymorfismus, dědičnost

#### 8.2.1 Zapouzdření

Zapouzdření v objektech znamená, že k obsahu objektu se nedostane nikdo jiný, než sám vlastník. K okolí je objekt přístupný pouze pomocí definovaného rozhraní (metody daným způsobem zprostředkovávající, či modifikující, data objektu).

#### 8.2.2 Dědičnost

Třídy jsou organizovány ve stromové hierarchii. Potomek třídy od svého rodiče přejímá (dědí) jeho metody a vlastnosti (public a protected). Zděděné metody lze překrýt vlastními implementacemi. Při tvorbě dědičnosti bychom se měli řídit pravidlem *is-a*.

**is-a:** Mějme entity A, B. Třída reprezentující entitu B může dědit z třídy A pouze pokud je tvrzení „B je A“ pravdivé (i.e. „Pes je Zvíře“).

**Vícenásobná dědičnost:** viz ?? – ??

V některých programovačích jazycích může být třída prohlášena za neděditelnou pomocí klíčových slov **final** (Java), nebo **sealed** (C#).

### 8.2.3 Polymorfismus

Situace, kdy několik objektů různých tříd poskytje stejné rozhraní a navenek se s nimi pracuje stejně, avšak jejich chování se liší dle implementace. Můžeme také hovořit o polymorfismu v souvislosti s dědičností – jedná se o situaci, kdy je na nějakém místě očekáván objekt dané třídy, či libovolné třídy z ní odvozené. Rozhraní dané třídy je vždy podmnožinou rozhraní libovolné její potřídy.

## 8.3 Metody objektů, jejich typy, způsoby ochrany

### 8.3.1 Metody

Metoda je ve třídě předepsaná pomocí:

- Názvu metody
- Návratového typu – u statický typovaných jazyků
- Argumentů
- Modifikátoru přístupu – **public** (lze volat odkudkoli), **protected** (lze volat pouze z metod stejné, či odvozené třídy), **private** (lze volat pouze z metod stejné třídy)
- Dodatečných atributů – metody mohou být statické, abstraktní (virtuální), konečné (finální)

**Poznámka 2.** Speciálním případem metod může být například konstruktor, destruktor, či gettery a settery.

Setkáváme se s pojmem **přetěžování metod**. Jedná se o situaci, kdy třída obsahuje více metod se stejným názvem. Takové metody se musí lišit počtem, nebo typem argumentů. Při volání metody je potom na základě argumentů zvolena požadovaná varianta.

### 8.3.2 Typy metod

Metody lze definovat jako:

- statické – kapitola ??
- abstraktní (virtuální) – kapitola ??
- konečné (finální) – Třída potomka je již nemůže předefinovat

### 8.3.3 Způsoby ochrany

Metody, stejně jako atributy objektu chráníme pomocí modifikátorů přístupu. Ty určují, odkud lze k metodám přistupovat.

- **public** – lze volat odkudkoli
- **protected** – lze volat pouze z metod stejné, či odvozené třídy
- **private** – lze volat pouze z metod stejné třídy

**Poznámka 3.** Je-li metoda privátní, neznamená to, že ji může volat pouze objekt dané třídy pouze v rámci sebe samotného. Může tuto metodu volat i na jiném objektu stejné třídy.

## 8.4 Metody tříd, abstraktní třídy

### 8.4.1 Metody tříd

Neplést s metodami objektů! Jedná se o metody k jejichž volání není potřeba vytvářet objekt dané třídy. Říkáme jim statické. Dle logiky věci nemohou tyto metody pracovat s daty konkrétního objektu. Nelze tedy přistupovat k hodnotám atributů, či volat ne-statické metody v rámci též třídy. Mohou však modifikovat proměnné třídy (neplést s vlastnostmi objektu), označované jako statické proměnné. Změna takovéto proměnné se projeví v rámci celého programu.

Implementačně mohou být řešeny tak, že se při prvním volání metody dané třídy vytvoří speciální objekt této třídy, obsahující pouze statické metody a statické proměnné.

### 8.4.2 Abstraktní třídy

Jedná se o třídy, z nichž nelze vytvořit objekt. Slouží ke generalizování vlastností a základního chování určité skupiny tříd (jejich potomků). Abstraktní třída může své metody libovolně buď implementovat, nebo označit za abstraktní. V tom případě pak ne-abstraktní potomek třídy musí dané metody implementovat.

Hlavním rozdílem, mezi abstraktní třídou a rozhraním (??) je to, že abstraktní třída může obsahovat implementace metod.

**Příklad:** Mějme dva typy databází – SQL databáze (Oracle, PostgreSQL, MySQL, atp.) a objektové databáze (např. MongoDB). Program chceme postavit tak, abychom mohli implementace (v tomto případě operací `select` a `insert`) jednoduše zaměňovat (pomocí polymorfismu).

Z logiky věci vyplývá, že na vrcholu naší hierarchie bude rozhraní definující operace `select` a `insert`. Z něj pak budou dělit dvě větve tříd. První větev bude určena pro SQL databáze a jejím vrcholem bude abstraktní třída implementující metody `select` a `insert` obecným způsobem platným pro značnou část relačních databází. Z této třídy budou dělit jednotlivé implementace pro specifické databáze (a případně překryjí generické chování vlastní implementací).

Druhou větví budou třídy určené pro jednotlivé objektové databáze. Ty bohužel žádnou abstraktní třídu mít nemohou, jelikož nemají žádný standardizovaný dotazovací jazyk.

## 8.5 Vícenásobná dědičnost, rozhraní

### 8.5.1 Vícenásobná dědičnost

V některých programovacích jazycích může třída dědit z více rodičovských tříd současně. Jedná se například o C++, Common Lisp, Python, ... Jazyky podporující pouze „jednoduchou“ dědičnost jsou například C#, Java, nebo Ruby.

Při použití vícenásobné dědičnosti narázíme zejména na problém zvaný *The diamond problem*. Mějme třídy  $A, B, C, D$ , takové, že  $B$  dědí  $A$ ,  $C$  dědí  $A$  a  $D$  dědí z obou  $B$  i  $C$ . Nechť třídy  $B$  a  $C$  definují metodu  $m$ , zatímco  $D$  ji nedefinuje. Zavoláme-li metodu  $m$  na objektu třídy  $D$ , která její implementace bude zavolána? Různé jazyky tento problém řeší různě. Optimálním způsobem je využití C3 linearizačního algoritmu zvaného MRO (Method Resolution Order), který podle daných pravidel vytvoří lineárně uspořádaný graf a následně vybírá pouze dalšího následovníka v řadě.

### 8.5.2 Rozhraní

Rozhraní je předpis tvořený seznamem metod splňujícím určitá kritéria.

- Předepsané metody nemají implementaci (tělo), pouze hlavičku
- Předepsané metody nedefinují modifikátor přístupu. Metody musejí být následně implementovány jako veřejné

Případně také může obsahovat konstanty.

Pokud třída ve své definici říká, že implementuje dané rozhraní, pak to znamená, že implementuje **všechny** jeho metody. Jaké to má dopady na programování?

- Můžeme se spolehnout, že objekty třídy implementující dané rozhraní, budou disponovat danou množinou metod
- Polymorfismus – pokud máme objekty více tříd implementujících dané rozhraní, pak se budou metody vykonávající danou věc jmenovat stejně
- Návrhový vzor *Program to an interface not an implementation* – kód by neměl být závislý na konkrétní implementaci, ale měl by záviset na rozhraní. Potom by měl fungovat s libovolnou třídou implementující dané rozhraní

**Poznámka 4.** *Rozhraním nelze předepsat atributy objektu, pouze metody a konstanty.*

## 8.6 Příklady objektově orientovaných jazyků a jejich rysy

### 8.6.1 Java

- Není čistě objektová – obsahuje i primitivní datové typy
- Statické typování
- Pouze „jednoduchá“ dědičnost; Možnost implementovat více rozhraní
- Podporuje přetěžování metod
- Nepodporuje přetěžování operátorů
- Metody nejsou elementy prvního řádu; od Java 8 ale podporuje lambda výrazy

### 8.6.2 C#

- Není čistě objektový – obsahuje i primitivní datové typy
- Statické typování
- Podporuje přetěžování metod i operátorů
- Metody nejsou elementy prvního řádu, ale podporuje lambda výrazy

### 8.6.3 Smalltalk

- Čistě objektový
- Pouze „jednoduchá“ dědičnost
- Dynamické typování ⇒ Nepodporuje přetěžování metod
- Řízení přístupu – veřejné metody, soukromé data
- Všechny operace jsou řešeny zasíláním zpráv

#### **8.6.4 Common Lisp**

- Dynamický jazyk
- Objektový systém CLOS (Common Lisp Object System)
- Multiparadigmatický jazyk – OOP je jedno z mnoha paradigmat, které lze kombinovat
- Dynamické typování  $\Rightarrow$  Nepodporuje přetěžování metod
- Vícenásobná dědičnost
- Metody se definují mimo definici třídy

#### **8.6.5 Python**

- Multiparadigmatický
- Vícenásobná dědičnost – linearizační algoritmus MRO
- Nepodporuje rozhraní
- Dynamické typování  $\Rightarrow$  Nepodporuje přetěžování metod
- Nepodporuje řízení přístupu – privátní vlastnosti a metody se označují pouze podtržítkem na začátku názvu
- Podporuje přetěžování operátorů
- Metody jsou elementy prvního řádu, podporuje lambda výrazy