1. Úvod do technologie programování a programovacích stylů, objektově orientovaný návrh, základní UML diagramy, psaní programů v Javě

- úvod do technologie programování a programovacích stylů
  - viz otázky z PGS

- objektově orientovaný návrh
  - za počátek objektově orientovaného prg. lze považovat vznik jazyka Simula67, první OOJ je Smalltalk
  - čistě objektově orientovaný jazyk – vše je objekt vč.základních datových typů a konstrukcí
  - objektová orientovanost se postupně rozšířila do všech oblastí vývoje SW
  - výhodou je strukturovanost
    - **přehlednost** kódu
    - **sdružování** podprogramů do **modulů**
  - funkce jsou aktivní (mají chování), data pasivní; pokud máme zpracovávat podobná data, do všech fcí musíme přidat podmínky => vzrůstá složitost programu až do fáze, kdy je již nevýhodné upravovat a rozšiřovat stávající SW a je nutná restrukturalizace
  - problém: znovupoužitelnost kódu v dalších aplikacích – knihovny a fce jsou pouze na nízké úrovni => principy abstrakce, dědičnosti

  - pojmem objektově orientovaný návrh rozumíme SW organizovaný jako množina objektů vzájemně komunikující
  - na začátku návrhu analýza požadavků, posléze rozšiřování detailů až implementace
  - cíle OO návrhu:
    - **robustnost** – reakce SW na jakékoli vstupy
    - **přizpůsobivost** – schopnost běžet po malých úpravách na různých platformách
    - **znovupoužitelnost** – možnost využití knihoven, fcí...
  - principy OO návrhu:
    - **abstrakce** – cílem je rozdělit složitý problém na jednoduché
      - příkladem jsou ADT – matematický model datových struktur (v Javě třídy)
    - **zapouzdření** – cílem je ukrýt implementační detaily, které nejsou pro uživatele důležité a poskytnout metody pro přístup k datům (vzpomeň si na třídy a máš to :-)).
    - **modularita** – rozdělení SW systému do modulů
  - **dědičnost** – umožňuje budování hierarchie tříd, které se postupně rozšiřují
    - používá se v případech, kde se chceme vyhnout opakování kódu
    - vychází se ze superclass, další třídy jméno extends superclass
    - změnit vlastnosti metody v odděděné třídě lze pomocí:
      - přetížení (jiné parametry metody)
      - překrytí (stejné parametry metody)
    - konstruktor se volá přes super()
    - finální metody nelze překrýt, ale lze je přetížit
    - pokud vyžadujeme překrytí, používáme abstract
  - **polymorfismus** – možnost využití stejné syntaktické podoby metody s různou vnitřní interpretací
    - abstraktní třídy, interface (rozhraní)

- jazyk UML
  - unifikovaný modelovací jazyk
  - prostředek pro OO analýzu a návrh, skládá se z grafických prvků, které se vzájemně kombinují do podoby diagramů
  - účelem diagramů je vytvoření určitých pohledů na navrhovaný systém = model
  - model jazyka UML udává, co má systém dělat, ale neříká, jak to implementovat
  - 8 typů diagramů:
    - **diagram tříd a objektů** – popisují statickou strukturu systému
    - **modely jednání (diagram případů užití)** – dokumentují možné případy použití systému, události, na které musí systém reagovat
    - **scénáře činností (diagramy posloupností)** – popisují scénář průběhu určité činnosti
    - **diagramy spolupráce** – zachycují komunikaci spolupracujících objektů
    - **stavové diagramy** – popisují dynamické chování objektu nebo systému
    - **diagramy aktivit** – průběh aktivit procesu nebo činností
    - **diagramy komponent** – popisují rozdělení systému na funkční celky a definují náplň jednotlivých komponent
    - **diagramy nasazení** – popisují umístění funkčních celků

- psaní programů v Javě
  - základní kroky:
    - návrh – vlastní návrh tříd (identifikace, atributy, fce)
      - např. CRC karty – množina karet, na nichž jsou podrobnější popisy tříd (název třídy, odpovědnosti, spolupracovníci)
      - kandidáti na třídy – podstatná jména v návrhu; odpovědnosti – slovesa; spolupracovníci – jak jsou vlastně třídy provázané
    - implementace a kódování
    - testování, ladění a optimalizace

2. Abstraktní datové typy zásobník, fronta, seznamy, řady, vektory a jejich implementace

- zásobník
  - použitá strategie LIFO, použití: Undo, Backspace...
  - metody pro práci se zásobníkem:
    - push(o) vkládá o na vrchol
    - pop() vyjímá prvek z vrcholu – pokud je prázdný, chyba
    - size() velikost
    - isEmpty() true, když je prázdný
    - top() vrací objekt na vrcholu zásobníku bez jeho vymazání
  - implementace:
    - polem
      - přímý – ve směru rostoucích adres (dno je na indexu 0, vrchol roste)
      - zpětný – opak přímého
    - seznamem zřetězených prvků
      - prvky se vkládají na čelo seznamu (od head)
  - SP – stack pointer – ukazuje na první volnou položku

- fronta
  - datová struktura používající strategii FIFO, použití: hromadné objednávání (vstupenek...), systémová oblast (požadavky na procesor)
  - metody pro práci s frontou:
    - enqueue(o) vkládá objekt na konec fronty
    - dequeue() odstraňuje objekt z čela fronty
    - size() počet prvků ve frontě
    - isEmpty() true, když je fronta prázdná
    - front() vrací objekt z čela fronty bez jeho odstranění
  - implementace:
    - polem – o tom bych radši pomlčel ;-)
    - seznamem zřetězených prvků
      - máme ukazatel na head a na tail – odtud se vyjímá nebo vkládá

- obousměrná fronta
  - obdobná frontě, akorát lze přidávat a vybírat prvky na obou koncích fronty
  - metody:
    - insertFirt(o) vkládá o na začátek fronty
    - insertLast(o) vkládá o na konec fronty
    - removeFirst() odstraňuje prvek z čela fronty
    - removeLast() odstraňuje objekt z konce fronty
    - size() počet objektů ve frontě
    - isEmpty() true, když je prázdná
    - first() vrací objekt v čela fronty bez jeho vymazání
    - last() vrací objekt z konce fronty bez jeho vymazání
  - implementace
    - seznam obousměrně zřetězených prvků
      - držíme zase head a tail s prvky, které jsou NULL, a namísto jednoho provázání máme dvě

- vektor
  - vektor je lineární posloupnost prvků V, která obsahuje n prvků. Každý prvek vektoru V je přístupný prostřednictvím indexu r v rozsahu [0, N-1]. Vektor připomíná datový typ pole, ale není to pole
  - metody pro práci s vektorem:
    - elemAtRank(r) vrací objekt na indexu r
    - replaceAtRank(r,o) přepíše objekt na pozici r objektem o a vrátí původní
    - insertAtRank(r,o) vloží objekt o na pozici r
    - removeAtRank(r) odstraní a vrátí objekt na pozici r
  - implementace:
    - polem – při vložení se prvky posunou doprava

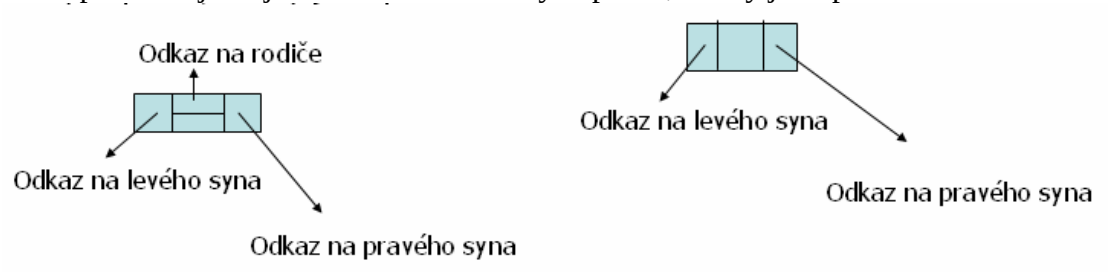
- seznam
  - seznam je lineární posloupnost prvků, která je propojena ukazateli (pointery). Prvek se vkládá na určitou pozici.
  - metody:
    - first() vrací odkaz na první prvek
    - last() vrací odkaz na poslední prvek
    - before(p) vrací odkaz na prvek před prvkem na pozici p
    - after(p) vrací odkaz na prvek za prvkem na pozici p

- isFirst(p) true, když p ukazuje na první prvek
- isLast(p)
- replaceElement(p, o) zamění prvek na pozici p prvek o a vrátí původní
- swapElements(p,w) zamění prvky na pozici p a w
- insertFirst(o) vloží prvek o na první pozici
- insertLast(o) vloží na konec seznamu
- insertBefore(p, o) vloží prvek o před prvek na pozici p
- insertAfter(p, o) vloží prvek o za prvek na pozici p
- remove(p) odstraní prvek na pozici p

- implementace:
  - obousměrně zřetězeným seznamem
    - head a tail (obsahují NULL) a odtud se řetězí prvky

3. Stromové struktury (AVL, BVS, B, Red-Black) a jejich implementace

- **binární strom**
  - každý vnitřní uzel může mít maximálně 2 následníky (syny)
  - implementuje se jako seznam zřetězených prvků, někdy jako pole



- **AVL strom**
  - AVL strom je výškově vyvážený binární vyhledávací strom, pro který platí, že pro libovolný vnitřní uzel stromu se výška levého a pravého syna liší nejvýše o 1
  - Vyváženost AVL stromu se kontroluje po každé operaci vložení a zrušení prvku, v případě že je vyváženost porušena, provádí se opětovné vyvážení pomocí jedné popř. několika rotací v jednotlivých částech stromu.
  - Implementace je obdobná jako u BVS, datová struktura pro uzel stromu je doplněna o celočíselnou proměnnou reprezentující stupeň vyváženosti uzlu, který může nabývat následujících hodnot:
    - 0 – oba podstromy jsou stejně vysoké
    - 1 – pravý podstrom je o 1 vyšší
    - 2 – pravý podstrom je o 2 vyšší
    - -1 – levý podstrom je o 1 vyšší
    - -2 – levý podstrom je o 2 vyšší
  - Rotace :
    - Jednoduchá pravá (levá) – používáme pokud vyvažujeme přímou větev, tj. jsou-li znaménka stupně vyváženosti stejná
    - Dvojitá pravá (levá) – používá se tehdy pokud nejde použít jednoduchá rotace – vyvažujeme-li „zalomenou" větev.
  - vkládání a vyjmutí – viz aplety

- **Red-Black stromy**
  - BVS, u kterých je časová složitost operací v nejhorším případě rovná O(log n)

- každý uzel stromu je obarven černě nebo červeně, kořen stromu je černý, listy jsou černé, na kterékoli cestě od kořene k listu leží stejný počet černých uzlů, červený uzel má pouze černé syny

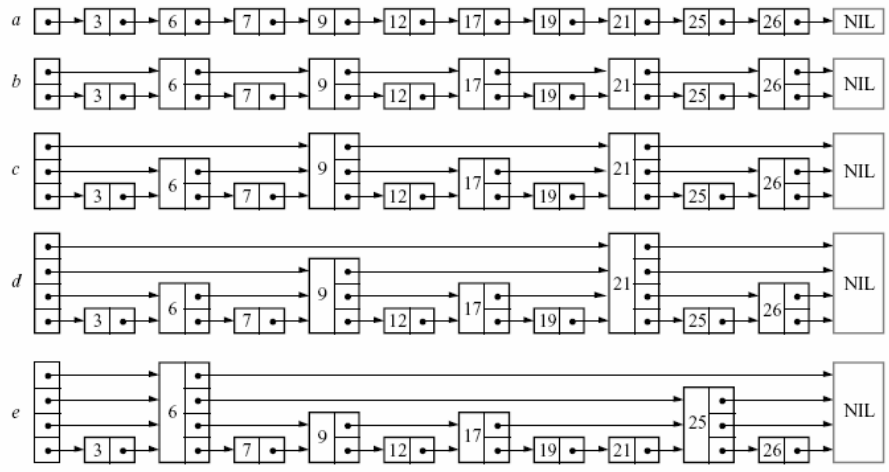
- **B-strom**
  - B-strom řádu *m* je strom, kde každý uzel má maximálně m následníků a ve kterém platí:
    - Počet klíčů v každém vnitřním uzlu, je o jednu menší než je počet následníků (synů)
    - Všechny listy jsou na stejné úrovni (mají stejnou hloubku)
    - Všechny uzly kromě kořene mají nejméně m/2 následníků (m/2 - 1 klíčů)
    - Kořen je buďto list, nebo má od 2 do m následníků
    - Žádný uzel neobsahuje více než m následníků (m-1 klíčů)

  - Vložení prvků:
    - Nový prvek se vždy vkládá do listové stránky, ve stránce se klíče řadí podle velikosti.
    - Pokud dojde k přeplnění listové stránky, stránka se rozdělí na dvě a prostřední klíč se přesune do nadřazené stránky (pokud nadřazená stránka neexistuje, tak se vytvoří)
    - Pokud dojde k přeplnění nadřazené stránky předchozí postup se opakuje dokud nedojde k zařazení nebo k vytvoření nového kořene.

4. Skip-list – použití a implementace

- je datová struktura, která může být použita jako náhrada za vyvážené stromy.
- představují pravděpodobnostní alternativu k vyváženým stromům (struktura jednotlivých uzlů se volí náhodně)
- Na rozdíl od stromů má **skip list** následující výhody:
  - jednoduchá implementace
  - jednoduché algoritmy vložení/zrušení
  - časová složitost vyhledávání je obdobná jako u stromů

- složitost vyhledávání v jednotlivých strukturách



spojový seznam n
ukazatele mezi 2
(n/2) + 1

ukazatele mezi 4
(n/4) + 1

ukazatele mezi 2^i
log n

náhodné ukazatele
(skip list)
???

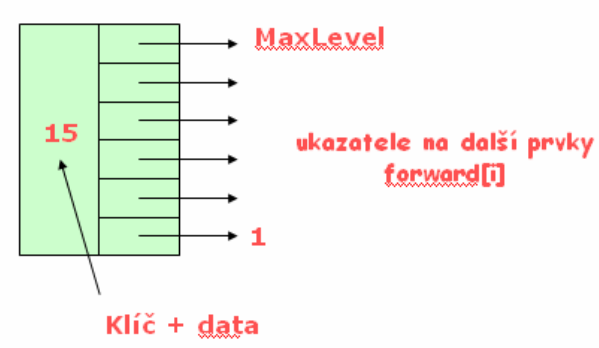
- prky v seznamu jsou uspořádány
- seznam obsahuje prvky, které mají k ukazatelů $1 \leq k \leq$ max_level
- uzel s *k*-ukazateli se nazývá uzel úrovně k
- seznam úrovně k – obsahuje prvky s maximálně k ukazateli
- ideální skip-list – každý 2^i-tý prvek má ukazatel, který ukazuje o 2^i prvků dopředu

- Pokud má každý 2^itý uzel 2^i ukazatelů na následující uzly, pak jsou uzly jednotlivých úrovní rozloženy následovně:
  50% uzlů úrovně 1
  25% uzlů úrovně 2
  12.5% uzlů úrovně 3
  atd.

Výhoda: složitost vyhledávání O(log n)
Nevýhoda: po provedení operací insert/delete je nutné provádět restrukturalizaci seznamu

Řešení: ponechat rozložení uzlů ale vyhnout se restrukturalizaci – tj. uzly úrovně k jsou vkládány náhodně s uvedeným pravděpodobnostním rozložením

Prvek skip listu



- každý prvek úrovně k má k ukazatelů

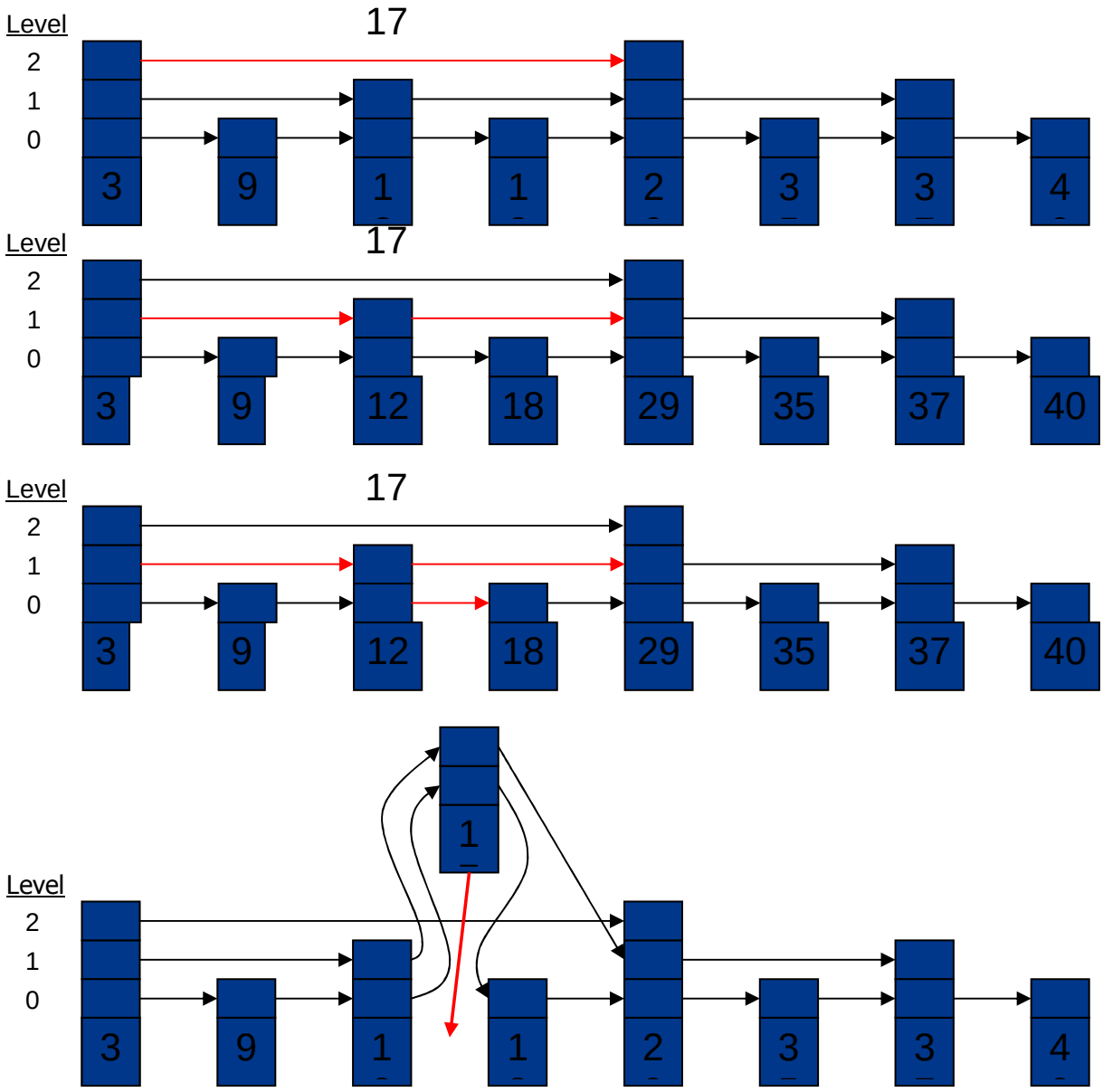
Inicializace seznamu

- je vytvořena hlavička seznamu (obsahuje MaxLevel ukazatelů), všechny ukazatele se inicializují na NIL
- MaxLevel = $\log_2(N)$, kde N je maximální počet prvků

Vyhledávání

- Začínáme v nejvyšší úrovni
- Dokud je hledaný prvek větší než prvek na který ukazuje ukazatel,
  - posouváme se vpřed v dané úrovni .
- Pokud je hledaný prvek menší než následující klíč,
  - přesuneme se o jednu úroveň níž.
- Opakujeme postup pokud není prvek nalezen, nebo pokud není jisté (v úrovni 1), že prvek neexistuje.

Vložení prvku



Zrušení prvku:

- Vyhledávacím algoritmem nalezněte pozici pro zrušení prvku
  - zapamatujte pozici předchůdce
- Zrušte uzel, je-li to nutné zmenšete MaxLevel.

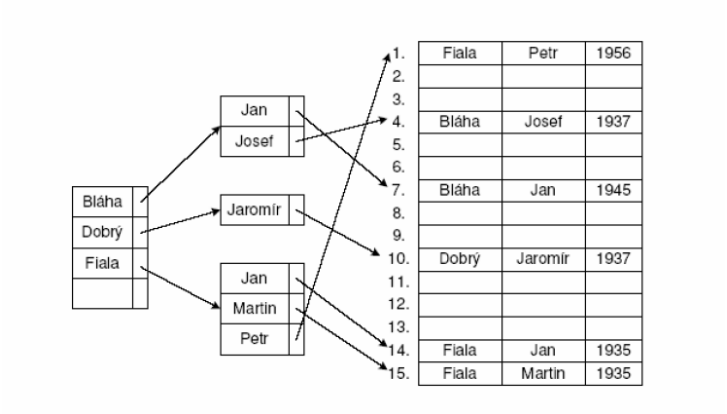
5. Tabulky s rozptýlenými položkami, vyhledávání v tabulkách

- tabulka = datová struktura, která umožňuje vkládat a později vybírat informace podle identifikačního klíče; mohou být:
  - pevně definované
  - s proměnným počtem položek
- A == průměrná délka prohledávání

- konvence:
  - k – klíč, kterým identifikujeme položku
  - A_k – adresní klíč, tj. adresa položky (ve většině případů je to index)
- druhy tabulek podle způsobu organizace:
  - tabulky s přímým přístupem
    - každá položka v tabulce má své místo jednoznačně určené
    - výhody: jednoduchá implementace, ryhlý přístup
    - nevýhody: řídké pole, velikost omezená rozsahem klíče
  - obyčejné vyhledávací tabulky
    - vyhledává se podle hodnoty klíče
    - pořadí prvků může být definované nebo náhodné
    - strategie vyhledávání:
      - sekvenční
        - položky v tabulce můžou být neuspořádané, vyhledává se postupným porovnáváním klíčů až do konce tabulky
        - výhoda: snadná implementace, jednoduchá modifikace tabulky
        - A = (n + 1)/2
      - binární
        - prvky musí být seřazeny podle hodnoty klíče
        - porovnám s prostředním prvkem, pokud není shoda, omezím se na pravý nebo na levý interval...
        - A = log(n)
      - Fibonacciho
        - princip stejný jako binární vyhledávání, pouze nevolím prostřední prvek, ale prvek na pozicích poměru Fib.posl.
        - složitost stejná jako binární, akortá prvky na počátku jsou nalezeny rychleji
  - tabulky se sekvenčním přístupem
  - tabulky s rozptýlenými položkami
    - viz dále

- hledání podle sekundárního klíče
  - invertovaný soubor
    - jedná se o tabulku seřazenou podle sekundárního klíče, data tvoří primární klíč
  - invertovaný seznam
    - inverzovaný soubor implementovaný jako zřetězený seznam

- vícerozměrné vyhledávání
  - vyhledávání podle více klíčů realizujeme vícenásobným přístupem



- tabulky s rozptýlenými položkami
  - používají se v případě, že rozsah klíče N >> rozsah tabulky
  - pro určení pozice v tabulce používáme hash funkci A_k = h(k), která klíči k jednoznačně učí A_k.
  - může se stát, že pro k_1 != k_2 je $h(k_1) = h(k_2)$, tzv. synonymické položky, dochází ke kolizím
  - požadavky na rozptylovací funkci:
    - pro každé k je jednoznačně definovaná
    - minimum kolizí
    - pravděpodobnostní rozdělení Ak = h(k) na intervalu <0; p-1> je rovnoměrné
  - realizace hash funkce i = h(k):
    - i je částí k
    - i je částí operace nad k
    - i je zbytkem po dělení rozsahem tabulky p
    - i je zbytkem po dělení N, N je nejbližší menší pročíslo, než je rozsah tabulky p

  - **tabulky s otevřeným rozptýlením**
    - každá pozice tabulky je přístupná položce s libovolným klíčem, vznikají shluky položek
    - při kolizi A_k se hodnota přepočítá
    - hash fce např. h = h mod p
    - podle způsobu řešení kolizí máme:
      - tabulky s otevřeným rozptýlením a nedefinovaným způsobem ukládání synonymických položek
      - tabulky s otevřeným rozptýlením a konstantním krokem s
        - na klíč aplikujeme $h_0(k)$, když kolize, tak $h_1(k)$... dokud nenajdeme volné místo
          - $h_0(k) = h(k)$
          - $h_1(k) = (h(k) + s) \bmod p$
          - …
          - $h_n(k) = (h(k) + ns) \bmod p$
        - p je rozsah tabulky, s je přirozené číslo nesoudělné s p

Příklad: Tabulka s

- 8 pozicemi
- položkami s klíči DAVID, HANA, DANA, HELENA, EMIL, EVA, BOŽENA
- hashovací funkcí

$$h(k) = \text{pořadí 1. písmene v abecedě}$$

S = 1

0	HELENA	2
1	BOŽENA	1
2		
3	DAVID	1
4	DANA	2
5	EMIL	2
6	EVA	3
7	HANA	1

st. složitost = 12/7

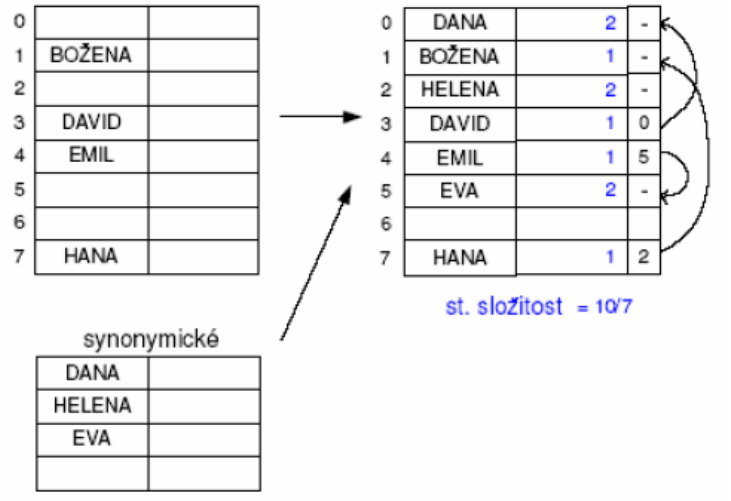
S = 3

0		
1	BOŽENA	1
2	HELENA	2
3	DAVID	1
4	EMIL	1
5	EVA	4
6	DANA	2
7	HANA	1

st. složitost = 12/7

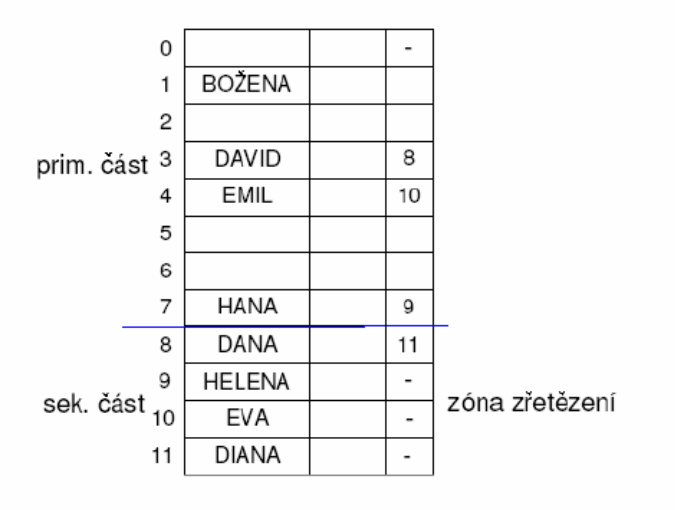
  - tabulky s otevřeným rozptýlením a ukládáním synonym s lineární vícenásobnou ukládací funkcí (to je asi na nic)
  - tabulky s s otevřeným rozptýlením a ukládáním synonym s kvadratickou vícenásobnou ukládací funkcí (to je asi na nic)

- **tabulky s otevřeným rozptýlením a vnitřním zřetězením**
  - ve fázi zařazovánní si necháme synonymické položky stranou a po zařazení položek synonymické položky vložíme na zbylé pozice v tabulce a příslušná synonyma zřetězíme do řetězu synonym



  - přesun synonym do tabulky:
    - od začátku
    - od konce
    - s vícenásobnou hash funkcí – to je výhodné, jelikož to nezničí rovnoměrné rozložení prvků

- **tabulky s uzavřeným rozptýlením a vnějším zřetězením**
    - tabulku rozdělíme na dvě části:
        - primární část – pouze položky, která nejsou synonyma
        - sekundární část – položky, které jsou synonymy k položkám v primární části



- **metoda rozptýlených indexů**
    - tabulku chápeme jako vektor seznamů synonymických položek
    - zařazování nových prvků
        - na konec
        - uspořádaně – složitější implementace, ale urychlení

6. Algoritmy zpracování textů – operace s řetězci, porovnání se vzorem (KMP, Boyer-Moore algoritmus)

- přátelé, tohle opisovat nebudu – je to docela přehledný v přednášce Vyhledavani_retezcu.pdf

7. Komprese dat, rozdělení kompresních metod, princip kompresních metod (Huffmann, aritmetické kódování, LZW, JPG, Fraktálová komprese)

- cíl komprese: redukovat objem dat za účelem:
    - přenosu dat
    - archivace dat
    - ochrany před viry atd.
- kvalita komprese:
    - rychost komprese
    - symetrie x asymetrie kompresního algoritmu
        - symetrické alg. – stejný čas potřebný pro kompresi i dekompresi
        - asymetrické alg. – časy potřebné pro kompresi a dekompresi se liší
    - kompresní poměr – poměr objemu komprimovaných dat ku objemu nekomprimovaných dat
- komprese:
    - bezztrátová – po kódování a dekódování je výsledek stejný
        - nižší kompresní poměr

        - používají se pro kompresi textů a tam, kde nelze připustit ztrátu info
    - ztrátová – po kódování a dekódování dochází ke ztrátě
        - vyšší kompresní poměr
        - komprese obrázků, zvuku...
- metody komprese:
    - jednoduché – založené na kódování opakujících se posloupností znaků (RLE)
    - statistické – založené na četnosti výskytu znaků v komprimovaném souboru (Huffmannovo a aritmetické kódování)
    - slovníkové – založené na kódování všech vyskytujících se posloupností (LZW)
    - transformační – založené na ortogonálních příp. jiných transformacích (JPEG, fraktálová komprese)

- **LZW (Lempel – Ziv – Welch metoda)**
    - princip:
        - vyhledání opakujících se posloupností znaků, ukládání těchto posloupností do slovníku pro další použití a přiřazení jednoznakového kódu těmto posloupnostem
        - jednoprůchodová metoda – kóduje se při čtení souboru

Algoritmus komprese a vytvoření slovníku

```
S := přečti znak ze vstupu;

while (jsou další znaky na vstupu) do
begin
 C := přečti znak ze vstupu;
 if S+C je v kódovací tabulce then
   S := S+C
 else begin
     zapiš na výstup kód pro S
        přidej do kódovací tabulky (S+C)
        S := C
      end;
end;
zapiš na výstup kód pro S;
```

Výsledný výstupní řetězec:

65 66 67 256 258 257 68 259

Příklad: Komprese řetězce ABCABCABCDABC

Postup kódování

S (prefix)	C (suffix)	výstup (kód)
A	B	A(65)
B	C	B(66)
C	A	C(67)
A	B	—
AB	C	AB(256)
C	A	—
CA	B	CA(258)
B	C	—
BC	D	BC(257)
D	A	D(68)
A	B	—
AB	C	—
ABC	—	ABC(259)

kód	posloupnost
0..255	jednotlivé znaky
256	AB
257	BC
258	CA
259	ABC
260	CAB
261	BCD
262	DA

Algoritmus dekomprese a vytvoření slovníku

```
přečti OLD_CODE;
zapiš OLD_CODE na výstup;
while (na vstupu jsou další kódy) do
begin
  přečti NEW_CODE;
  if NEW_CODE není v kódovací tabulce then
    begin
      S := posloupnost zakódovaná kódem OLD_CODE;
      S := S+C;
    end
  else S := posloupnost zakódovaná kódem NEW_CODE;

  zapiš S na výstup;
  C := první znak S;
  přidej do kódovací tabulky (OLD_CODE+C);
  OLD_CODE := NEW_CODE;
end;
```

Vstupní řetězec:

65 66 67 256 258 257 68 259

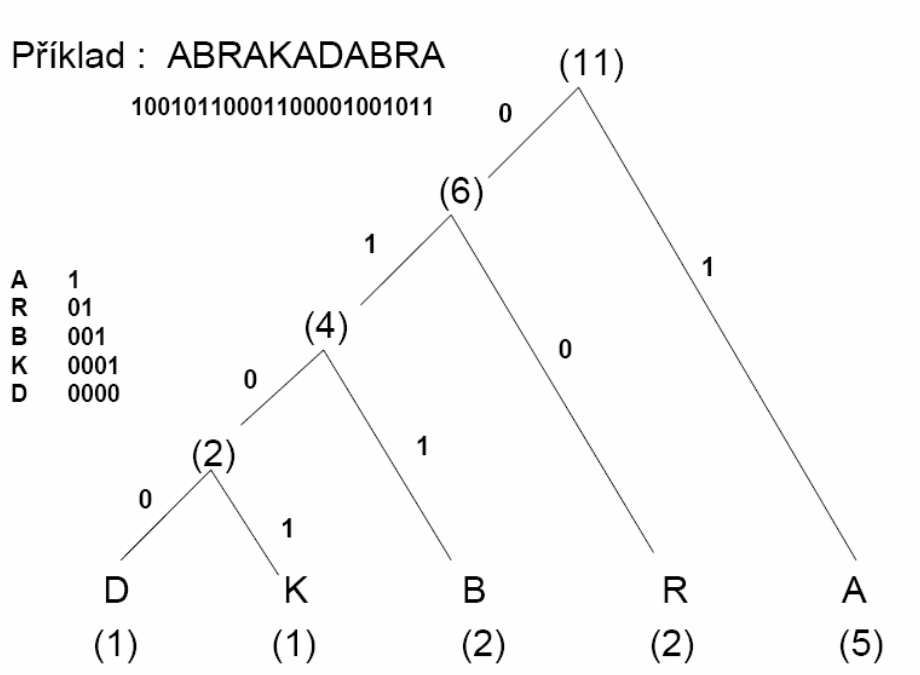
OLD_CODE	NEW_CODE	S	C	Výstup
A(65)				A
A(65)	B(66)	B	B	B
B(66)	C(67)	C	C	C
C(67)	AB(256)	AB	A	AB
AB(256)	CA(258)	CA	C	CA
CA(258)	BC(257)	BC	B	BC
BC(257)	D(68)	D	D	D
D(68)	ABC(259)	ABC	A	ABC

Výstupní řetězec:

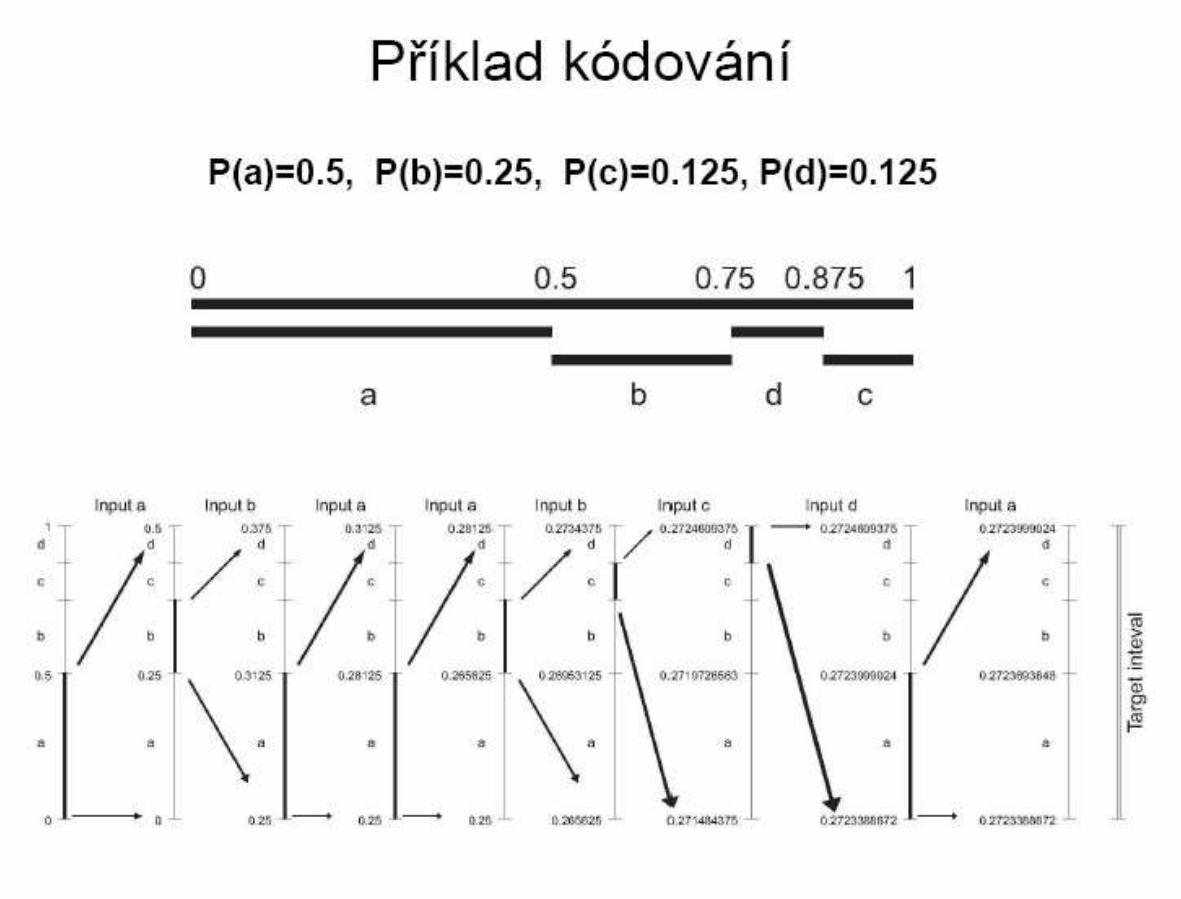
ABCABCABCDABC

- **Huffmannovo kódování**

- využívá optimálního (nejkratšího) prefixového kódu (kód žádného znaku není prefixem jiného znaku
- kódové symboly mají proměnnou délku
- algoritmus kódování
  - zjištění četnosti jednotlivých znaků v kódovaném souboru
  - vytvoření binárního stromu – seřadíme zleva doprava podle:
    - četnosti
    - podstrom vlevo před listem, větší podstrom před menším
    - pořadí v abecedě
  - uložení stromu
  - nahrazení symbolů jednotlivými kódy (posloupností bitů)



- **Aritmetické kódování**
- statistická metoda
- kóduje celou zprávu jako jedno kódové slovo
- princip:
    - Aritmetické kódování reprezentuje zprávu jako podinterval intervalu <0,1). Na začátku uvažujeme celý tento interval. Jak se zpráva prodlužuje, zpřesňuje se i výsledný interval a jeho horní a dolní mez se k sobě přibližují. Čím je kódovaný znak pravděpodobnější, tím se interval zúží méně a k zápisu delšího (to znamená hrubšího) intervalu stačí méně bitů. Na konec stačí zapsat libovolné číslo z výsledného intervalu.
- algoritmus komprese:
    - zjištění pravděpodobností P(i) výskytu jednotlivých znaků ve vstupním souboru
    - Stanovení příslušných kumulativních pravděpodobností K(0)=0, K(i)=K(i-1)+P(i-1) a rozdělení intervalu <0,1) na podintervaly I(i) odpovídající jednotlivým znakům (seřazeným podle abecedy)tak, aby délky těchto intervalů vyjadřovaly pravděpodobnosti příslušných znaků: I(i) = <K(i), K(i+1))
    - uložení použitých pravděpodobností
    - vlastní komprese



Příklad kódování

```
Vstup: a     L = 0
            H = 0 + 0.5·1 = 0.5
       b     L = 0 + 0.5(0.5 − 0) = 0.25
            H = 0 + 0.5(0.5 − 0) + 0.25(0.5 − 0) = 0.375
       a     L = 0.25
            H = 0.25+0.5 □ (0.375−0.25) = 0.3125
       a     L = 0.25
            H = 0.25+0.5 □ (0.3125−0.25) = 0.28125
       b     L = 0.25+0.5 □ (0.28125−0.25) = 0.265625
            H = 0.25+0.5 □ (0.28125−0.25)+0.25 □
                (0.28125−0.25) = 0.2734375
       c     L = 0.265625+0.5 □ (0.273437c−0.265625)+0.25
                □ (0.2734375−0.265625)
                 = 0.271484375
            H = 0.265625+0.5 □ (0.2734375−0.265625)+0.25 □
                (0.2734375−0.265625)+0.125 □ 0.25 (0.2734375
                 0.265625)=0.2724609375
```

- dekódování:
  - rekonstrukce použitých pravděpodobností
  - vlastní dekomprese

- **JPEG**

- v současné době nejpoužívanější kompresní metoda pro obrázky
- vhodná pro fotky, nevhodná pro technické výkresy – dochází k viditelnému rozmazání
- princip:
  - části obrazu se transformují do frekvenční oblasti (výsledek je matice frekvenčních koeficientů)
  - z matice koeficientů se odstraní koeficienty, které odpovídají vyšším frekvencím (rychlejší změny jasu – např. hrany v obraze)
  - zbývající koeficienty se vhodným způsobem zkomprimují
- diskrétní kosinová transformace
  - tranformuje kódovanou oblast do frekvenční oblasti
  - je bezztrátová, existuje k ní inverzní transformace
  - postup:
    - zdrojový obraz se rozdělí na 8x8 pixelů
    - hodnoty jasu se tranformují z intervalu $[0, 2^p-1]$ na interval $[2^{p-1}, 2^{p-1}-1]$
    - provede se diskrétní kosinová tranformace dle šílených vztahů, který sem radši ani nepíšu ;-)

- **fraktálová komprese**

Komprese obrazu pomocí IFS je výpočetně náročný úkol, naopak dekódování je velmi rychlé. Jde tedy o silně asymetrický proces

Snad tento důvod neumožnil větší rozšíření této kompresní metody (je však třeba říci, že byly představeny rychlé metody, například institut v anglickém Bathu představil tzv. BFT - Bath Fractal Transform, zajímavé práce o aplikaci FK v multimédiích vypracoval John

Kominek a dobře optimalizovaný algoritmus fungoval v programu Fractal Imager od Iterated Systems, pokoušející se prorazit před standardem JPEG).

Druhým důvodem by mohla být nejistá kvalita obrazu. Metodou fraktálové komprese je vhodnější kódovat spíše přírodní scenérie, než interiéry.

Ačkoliv je většina digitálních fotografií právě z exteriéru, dávají výrobci přednost standardu JPEG, který je v obou zmíněných ohledech flexibilnější.

Žádný z existujících grafických formátů, kromě nestandardizovaného formátu FIF (Fractal Image Format od Iterated Systems) a STN (STiNG od Altamiry, nyní vlastněné spol. LizardTech) neposkytuje nezávislost na rozlišení. JPEG obrázek můžeme dekódovat pouze ve stejném rozlišení, v jakém byl kódován. Pokud ho zvětšíme, budě trpět pixelací (tj. pixely jako čtvercové body se roztáhnou na skutečně viditelné čtverce) nebo po vyhlazení interpolací bude rozmazaný. Obrázek kódovaný pomocí jedné z metod fraktálové komprese je definován množinou transformací a je tedy podle definice fraktál. To ale znamená, že ho lze donekonečna zvětšovat a nikdy neztratíme detaily. Ve skutečnosti fraktálová komprese skutečně přináší nezávislost na rozlišení, ale detaily získané při zvětšení jsou umělé - dopočítané. Podobně jako se u JPEG ztráta projevuje vlněním u hran a blokovitostí, trpí obrázky kódované fraktálovou kompresí různým typem blokových artefaktů (podle použité metody) a vykreslováním neexistujících detailů.

S použitím postprocessingových vyhlazovacích algoritmů lze ovšem i tento nešvar napravit a přínosem jsou pak ostré, téměř spojité hrany skoro jako ve vektorovém obrázku. Tuto vlastnost fraktálové komprese využila spol. Altamira ve svém poměrně dost drahém softwaru Genuine Fractals pro zvětšování obrázků.