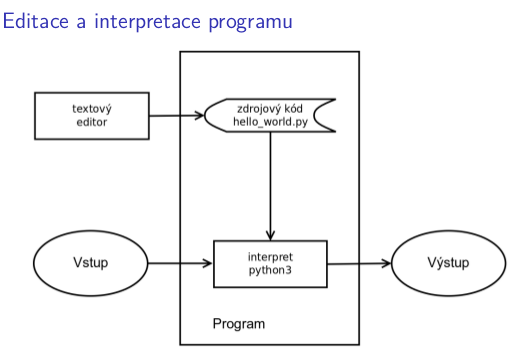
**I01: Základní programové struktury a techniky. Časová a paměťová složitost algoritmů. Datové struktury a jejich vlastnosti (pole, zásobník, fronta, záznam, prioritní fronta, hromada, rozptylovací tabulka). (Algoritmy a programování)**

**Všechny ukázky kódu jsou ukázky z jazyku Python!!!!**

**Základní programovací techniky a struktury**

*Algoritmus* = přesný, detailní a úplný postup (obvykle řešení problému)

*Program* = zápis algoritmu v konkrétním programovacím jazyce

*Výrazy* (expressions) - obsahují celá čísla, reálná čísla, operátory, oddělovače (závorky..)

Co se děje v zákulisí?

REPL:  *read* - přečtení uživatelského vstupu

*eval* - vyhodnocení výrazu

*print* - tisk výsledku

*loop* - opakované vykonání (smyčka)

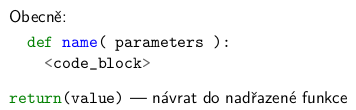
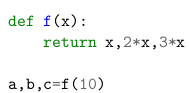
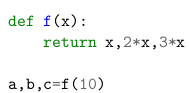
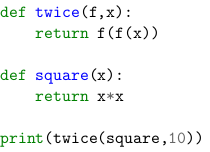
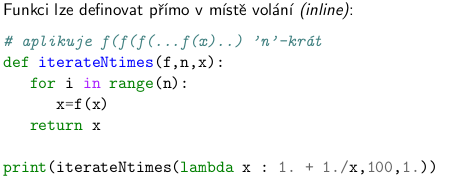
Základní části textu programů

* *Komentáře:* #comment
* *Klíčová slova:* import (import knihoven, modulů aj.)
* *Identifikátory:* jména proměnných a fcí - písmena, čísla, podtržítka, nezačíná číslem, není klíčové slovo
* *Operátory:* +, -, \*, /, =
* *Literály:* celá čísla, reálná čísla, řetězce (“Hello world!”)

**Řídící struktury**

* Porovnávání čísel: 8>3 => True
* Podmíněný příkaz: *if* <podmínka>: <blok>
* Větvení: *if-else*
* Vnořené větvení: několik *ifů* v sobě
* Zřetězené podmínky: *if-elif-else*
* Cykly (smyčky):
  + for cyklus: for <proměnná> in range(n): <blok>
    - v pythonu nabývá proměnná hodnoty 0,....,n-1
    - cykly se můžou vnořovat
  + while: while <podmínka>: <blok>
    - cyklus probíhá, dokud je podmínka True (je třeba proměnné iterovat uvnitř cyklu)
  + break, continue: přerušení cyklu
    - v těle cyklu for nebo while
    - break ukončí celý cyklus
    - continue přeruší aktuální iteraci a začne následující

Prostředky pro strukturování kódu:

* Bloky kódu (oddělené odsazenı́m)
* Programy = soubory - jméno.py
* **Funkce**
  + vracı́ hodnotu vypočı́tanou ze vstupnı́ch argumentů
  + čistá funkce (pure function) = výstup závisı́ pouze na vstupnı́ch parametrech, a to jednoznačně (tj. nezávisí na ničem jiném)
  + existující funkce - max(2,3)
  + funkce definované uživatelem -->
  + Složitějšı́ funkce volá jednoduššı́ - skládánı́ funkcı́
  + Počet parametrů je dán hlavičkou funkce
    - existují i fce s proměnným počtem parametrů (max…)
  + void funkce (procedura) - funkce bez návratové hodnoty (nebo vrací *None*)
  + více návratových hodnot - vrací n-tici (tuple)
  + vedlejší efekty fcí - můžou ovlivnit stav prostředí
    - výpis na obrazovku (print), zápis do souboru
    - změnu stavu proměnných volajícího
    - funkce pak není čistá (má vedlejší efekty)
  + funkce může být argumentem jiné funkce ------------------------->
  + Anonymní fce - lambda notace
* Funkcionálnı́ programovánı́ = čisté funkce, funkce vyššı́ho řádu, jednorázové přiřazenı́

Rozsah platnosti proměnných

* globální = definované v hlavním programu (mimo fce), viditelné všude
* lokální = definované uvnitř fce, viditelné pouze uvnitř této fce
* lokálnı́ proměnná se stejným jménem zastı́nı́ proměnnou globálnı́

Moduly

* Modul obsahuje tématicky souvisejı́cı́ funkce
* Modul = soubor .py
* Proč použı́vat moduly:
  + Program ve vı́ce souborech
  + Sdı́lenı́ kódu mezi programy (znovupoužitelnost)
  + Definice rozhranı́ (API), skrytı́ implementačnı́ch detailů
  + Odstraněnı́ konfliktů jmen funkcı́.
* Jak používat moduly:

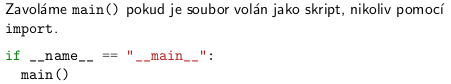
1. Zpřı́stupněnı́ modulu

* soubor v aktuálnı́m adresáři
* soubor v systémových adresářı́ch známých Pythonu

1. Import modulu:

* import math, import sys

1. Použitı́ kvalifikovaného jména (tečková notace):

* math.cos, sys.argv
* Jak moduly psát:
  + Každý .py soubor lze importovat -> Tı́m se vykonajı́ všechny přı́kazy v souboru
  + Odstranı́me globálnı́ kód
    - Testovacı́ kód do zvláštnı́ch funkcı́
    - Demonstračnı́ kód do funkce main()
    - Zavoláme main() pokud je soubor volán jako skript, nikoliv pomocı́ import

Strukturovaný kód

* Dobrý kód ≠ dlouhá sekvence přı́kazů.
* Kód má být hierarchicky strukturován
  + krátké sekvence přı́kazů (kódu) řešı́cı́ dobře definovanou část úlohy jsou sdružovány do většı́ch celků
  + ty sloužı́ jako stavebnı́ kameny pro složitějšı́ a většı́ podúlohy na vyššı́ úrovni atd.
  + podobně sdružujeme i datové struktury
* DRY = Do not Repeat Yourself
* Pokud lze část kódu logicky oddělit, měla by být oddělena
* Výhody strukturovaného kódu: přehlednost, zkrácenı́ kódu, úspora práce, snadnost jednotlivých kroků, znovupoužitelnost, modifikovatelnost, testovatelnost, rychlost vývoje

Datové typy:

* Jednoduché datové typy (mají hodnotovou sémantiku)
  + int (celé číslo), float (reálné číslo), boolean (logická hodnota)
* Složené typy/datové struktury
  + řetězec, n-tice (tuple), pole (má referenční sémantiku)
  + Hierarchicky sdružují data
  + Slouží k organizaci dat, související data jsou uložena a manipulována spolu
  + Operace na datových strukturách: Vytvoření, Čtení jednotlivých složek (elementů), Modifikace jednotlivých složek, Vyhledávání, Přidávání, odebírání, . . .

**Časová a paměťová složitost algoritmů**

* Trvání výpočtu v závislosti na vstupních datech
  + v nejhorším případě, v průměru. . .
* Který algoritmus je lepší?
* Jak velká data jsme schopni zpracovat?

**Empirická časová složitost**

* Změříme dobu běhu pro různá vstupní data
* Vyhodnocujeme algoritmus + implementace + hardware
* Kvantitativní výsledky
* Neposkytuje záruky, obtížná predikce

**Teoretická časová složitost**

* Studujme funkci *T(n)*
* Velikost problému *n*
  + vstupní parametr (parametrů může být více)
  + délka vstupních dat
* Čas běhu programu *T*
  + Ve fyzických jednotkách
  + V počtu ‘typických operací’ — přiřazení, porovnání, sčítání,. . . (výpočetní model)

**Asymptotická časová složitost**

* Přesný vzorec pro *T(n)* není nutný. . .
* Zajímají nás pouze kvalitativní rozdíly
* Multiplikativní konstanty zanedbáme
  + vliv počítače, programátora, programovacího jazyka. . .
  + “vždycky si můžeme koupit rychlejší počítač”
* Zajímá nás chování pro velká *n*
  + Rychlost růstu *T(n)* pro *n → ∞*
  + Pomaleji rostoucí části *T(n)* zanedbáme

**Asymptotická notace**

Řád růstu funkce / **big-O notation**

* *f(n): ℕ → ℝ0+* je *O(g(n))* pokud existují konstanty *c > 0* a *n0 > 0* takové, že *f(n) ≤ c·g(n)* pro všechna *n ≥ n0*
* Pro polynomiální *f(n)* — člen nejvyššího řádu, bez konstanty
  + př: *f(n)* = 456211n + 235166 ~ *O(n)*

*f(n)* = n(n + 2)/2 ~ *O(n2)*

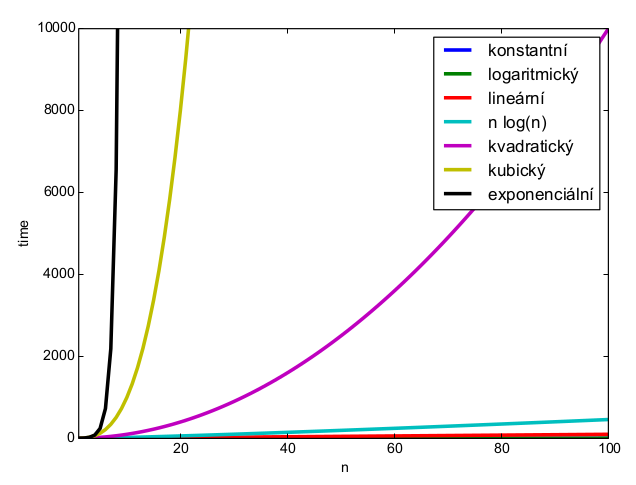
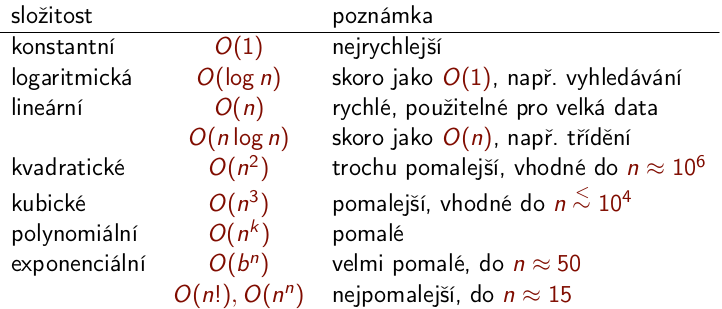
*f(n)* = 442(n + 12) log n ~ *O(n log n)*

*f(n)* = 4n3 + 100n2 + 1000n + 5000 ~ *O(n3)*

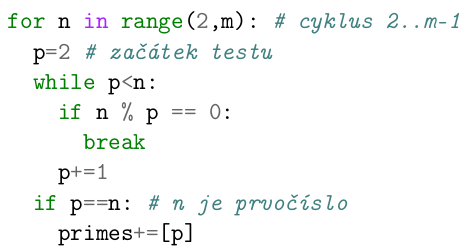
* *O(·)* notace je horní odhad, ale uvádíme ten nejlepší známý
  + Tedy f(n) = 4n3 + 100n2 je nejenom O(n3), ale zároveň i O(n4) a O(n10)
* **Big-Θ notation**
  + *Θ* notace je přesnější, ale v praxi se často setkáte s *O(·)* ve významu *Θ(·)*. Odhad *O(·)* je snazší nalézt
  + 4n3 + 100n2 je pouze Θ(n3), ale nikoliv i Θ(n4) … Θ je zároveň horní i dolní odhad
* **Limita pro velká n**
  + Zkráceně *f(n) ∼ g(n)* (je asymptoticky rovno)
  + Pro polynomy, pouze člen nejvyššího řádu, včetně konstanty.
    - *f(n)* = 4n3 + 100n2 + 1000n + 5000 *∼ 4n3*
    - *f(n)* = n(n + 2)/2 *∼ n2/2*

**Druhy odhadů**

* Časová složitost typicky závisí na datech (nejen na velikosti n)
* Průměrná složitost (Average complexity)
  + složitá teoretická analýza
  + lze odhadnout z experimentů na typických datech
* Nejhorší složitost (Worst-case complexity)
  + jen experimentálně nelze
  + teoretická analýza → různě přesné horní odhady
* Složitost může záležet na více parametrech vstupních dat (např. počet vstupních čísel a jejich maximální hodnota)

**Srovnání složitostí**

Příklad určení složitosti: hledání prvočísel

Analýza:

- Vnější for cyklus proběhne m−1-krát

- Každý vnitřní cyklus while proběhne max. n−1-krát, kde n < m

- Vnitřní cyklus tedy proběhne max. m2 -krát

- Složitost tohoto algoritmu je tedy **O(m2)**

Pozn.: Operace mimo vnitřní cyklus zanedbáme

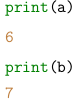
- Toto je horní odhad. Ve skutečnosti je složitost nižší, díky příkazu break

**Pole**

* Obsahuje N elementů (objektů,prvků)
* Přímý přístup (random access)
  + Pro čtení i zápis
  + V konstantním čase
* může a nemusí mít elementy stejného datového typu
  + Homogenní pole = všechny prvky jsou stejného typu
* prvky pole mohou být i složené datové typy (string, pole, tuple…)
  + vícerozměrná pole
* vytvoření pole:
  + výčtem: a=[0.3, 0.6, 0.1]
  + opakováním prvků: a=10\*[0]
  + z posloupnosti: list(range(1,11))
  + přidáváním nakonec: a=[ ] … a+=[0.2]
* indexace:
  + přístup k datům
  + x[ i ] … i-tý prvek, pro sekvenční typy: pole, řetězec, n-tice (tuple)
  + array slicing x[ i : j ] … vrátí i až j-1 prvek (indexace od 0)
    - vynechané argumenty: x[ i : ] … vrátí i-tý až poslední prvek

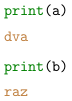
x[ : i ] … vrátí první až i - 1 prvek

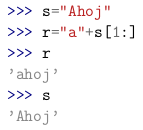
* x[-1] … vrátí poslední prvek

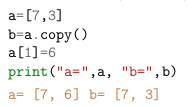
Hodnotová sémantika

* U objektu je důležitá hodnota, nikoliv identita. Proměnná reprezentuje hodnotu
* Primitivní typy v Pythonu se chovají jako hodnoty (values)
* Přiřazení vytvoří nový objekt.

Referenční sémantika

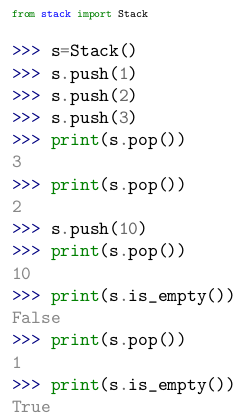
* Proměnná typu pole je referencí/odkazem (reference,link)
* Přiřazení vytvoří nový odkaz na existující objekt
* Pole lze měnit (mutable)
* Sdílení odkazů (sharing,aliasing)

Neměnnost (immutability)

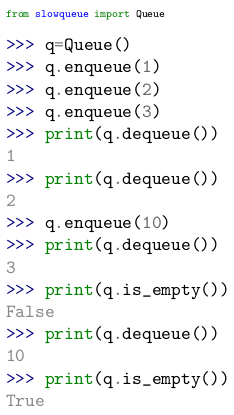
* Vlastnost datového typu.
* Neměnné objekty po vytvoření změnit nelze — řetězce,n-tice
* Neměnné typy (řetězce,n-tice) se také chovají jako hodnoty
* Práce s neměnnými objekty
  + Vytvoříme objekt nový, nezávislý na starém

Kopírování polí

* b=a.copy() (= hluboká kopie, změnou *b* nezměním *a*)
* vytvoří se nový objekt se stejným obsahem, při změně jedné proměnné se nezmění druhá (při *b=a* ~mělká kopie, se při změně *b* změní i *a*)



**Zásobník** (Stack)

* strukturovaný/složený datový typ
* obsahuje předem neznámé množství položek, typicky stejného typu (jako pole)
* podporuje následující operace (se složitostí O(1))
  + přidání položky na konec (**push**)
  + odebrání položky z konce (**pop**)
  + test, jestli je zásobník prázdný (**is\_empty**)
* položky jsou odebírány v opačném pořadí, než byly přidány, (**LIFO** — last in first out)
* zásobník může podporovat i další operace
  + nedestruktivní čtení z konce (**peek**)
  + zjištění počtu položek na zásobníku (**size**)

**Fronta** (Queue)

* strukturovaný/složený datový typ
* obsahuje předem neznámé množství položek, typicky stejného typu (jako pole)
* podporuje následující operace (se složitostí O(1))
  + přidání položky na konec (**enqueue**, **add**)
  + odebrání položky ze začátku (**dequeue**, **top**)
  + test, jestli je fronta prázdná (**is\_empty**)
* položky jsou odebírány ve stejném pořadí, jako byly přidány, (**FIFO** — first in first out)
* fronta může podporovat i další operace
  + nedestruktivní čtení ze začátku (**peek**)
  + zjištění počtu položek ve frontě (**size**)

Fronta - použití

* Komunikace mezi procesy (consumer,producer)
* Čekání na asynchronní periferie — klávesnice, disk, síť . . .
* ‘Férový přístup’ pro sdílení zdrojů (policy)
* Simulace čekání ve frontě
* Některé grafové a třídící algoritmy (prohledávání do šířky, merge sort)

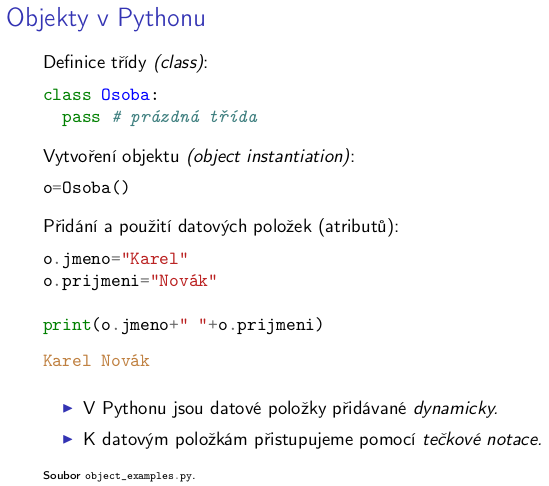
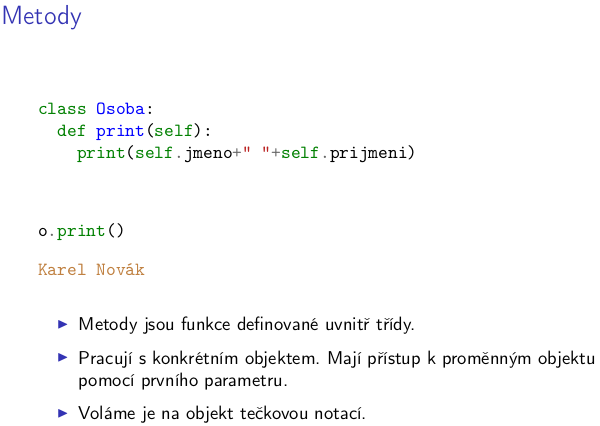
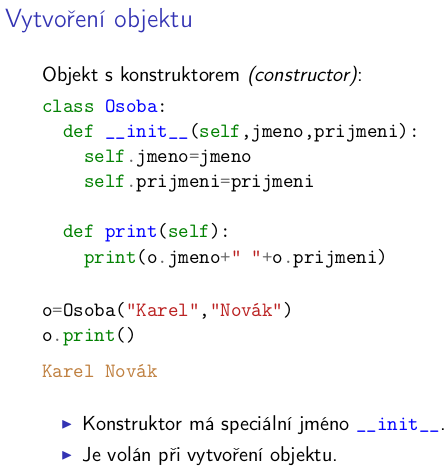
Fronta - pole s realokací

* Prvky ukládáme do pole
* Vkládání na konec pole (O(1))
* Prvky nemažeme, ale posouváme index počátku
* Pokud je vynechaných prvků hodně, překopírujeme frontu do nového pole
* Kopírujeme po poklesu využití paměti na 50 %
* Odebrání n prvků vyžaduje ∼ log2n kopírování, dohromady n/2 + n/4 + · · · ∼ n prvků → amortizovaná složitost odebrání prvku je O(1)
* Nevýhoda — neefektivní využití paměti

Fronta - dva zásobníky

* Prvky ukládáme do zásobníku inp (O(1))
* Prvky odebíráme ze zásobníku out (O(1))
* Když je out prázdný, přesuneme do něj inp
* Každý prvek je kopírován jen jednou, složitost zůstává O(1)

**Záznam**

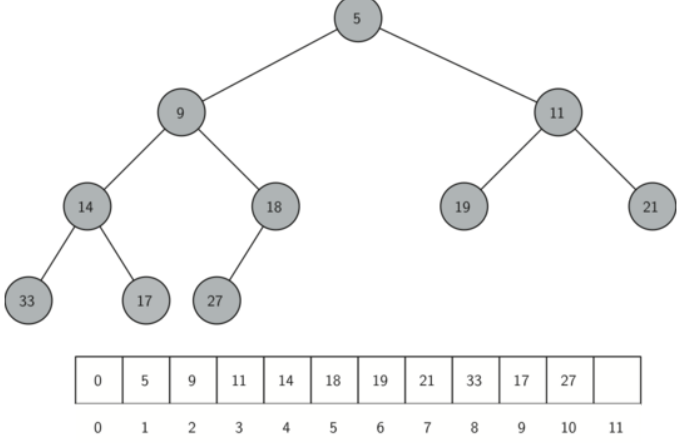
* strukturovaný/složený datový typ
* obsahuje položky (fields) / prvky (elements)/ členy
* položek je (obvykle) pevný počet, mají každá svůj typ
* položky jsou identifikované jménem
* typ záznamu má své jméno
* Příklady:
  + Datum = rok, měsíc, den
  + Osoba = jméno, příjmení, datum narození
  + Adresa = jméno ulice, číslo popisné, město, PSČ
  + Bod = x, y
  + Kruh = střed, poloměr
* V Pythonu záznamy nejsou, místo toho (dynamické) objekty
* **objekt** = záznam + metody
  + Obsahuje datové položky / atributy / proměnné (fields, attributes, instance variables)
  + Může obsahovat metody = funkce pracující s datovými položkami
  + Strukturu definuje třída (class), objekty jsou instance třídy
  + Základ objektově orientovaného programování (object oriented programming — OOP).

**Prioritní fronta** (Priority queue)

* Podporuje následující operace pro porovnatelné položky
  + přidání položky (**insert**)
  + odebrání **nejmenší** položky (**pop**)
* Může podporovat i další operace
  + nedestruktivní čtení ze začátku (**peek**)
  + zjištění počtu položek ve frontě (**size**)
  + změna položky
* Varianty (ekvivalentní)
  + odebíráme největší místo nejmenší
  + ukládáme klíč (prioritu) a hodnotu, třídíme podle klíče, vrací hodnotu
* Zásobník i fronta jsou speciálním případem prioritní fronty

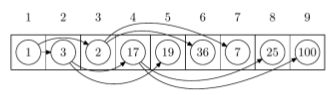
Aplikace prioritní fronty

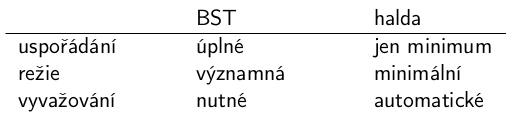
* Hledání *k* nejmenších prvků
* Prioritní rozvrhování a plánování (fronta s předbíháním)
* Simulace diskrétních událostí (priorita=čas)
* Základní prvek mnoha algoritmů
  + Heapsort (třídící algoritmus)
  + Huffmanovo kódování
  + Informované prohledávání grafu
  + Dijkstrův algoritmus pro hledání nejkratší cesty v grafu
  + Primův algoritmus pro hledání minimální kostry v grafu



**Hromada** (binární hromada/halda - heap)

* Prvky jsou uložené v poli a1, a2, ..., an
* Potomky ai jsou a2i , a2i +1 (pokud < n)
* Vlastnost haldy (heap property): uzel není větší, než jeho potomci, ai ≤ a2i , ai ≤ a2i +1
* Halda je binární strom, vrstvy až na poslední jsou plně obsazené
* Datová struktura pro rychlé hledání minimálního prvku umožňující přidávání
* Používáme pro implementace prioritní fronty
* Velmi efektivní implementace pomocí pole. Nepotřebuje odkazy
* Haldu lze sestrojit v čase O(n), odebrání nejmenšího prvku trvá O(log n)

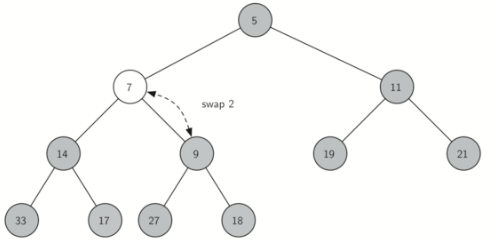
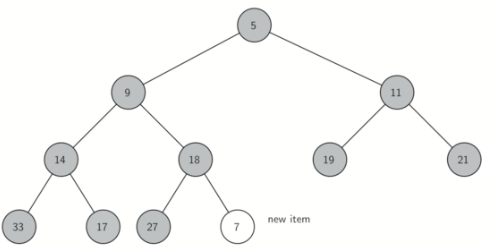
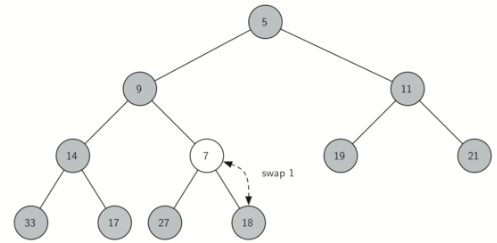




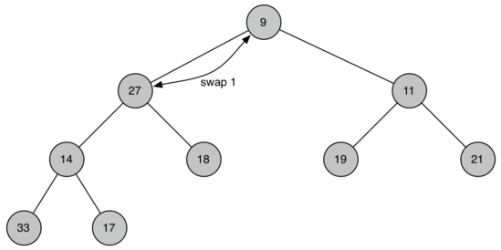
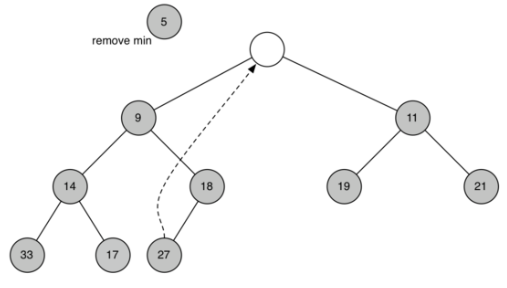
Srovnání haldy a stromu (binární vyhledávací strom, BST)

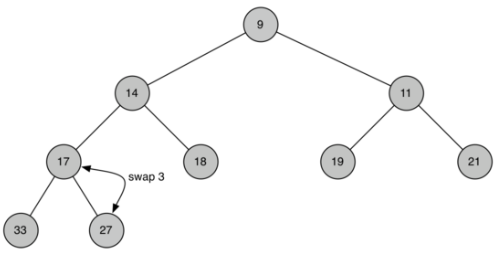
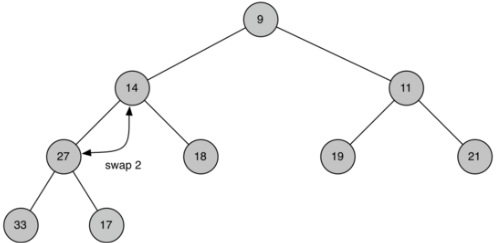
* halda je významně rychlejší, potřebuje méně paměti
* strom je univerzálnější, podporuje více operací

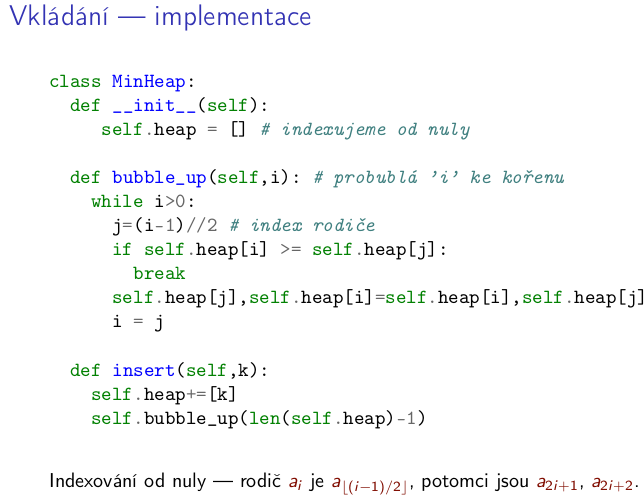
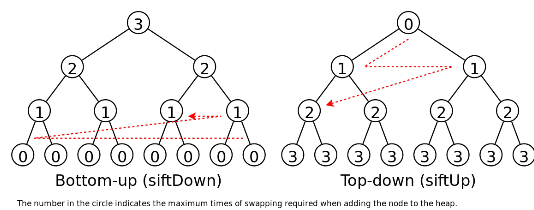
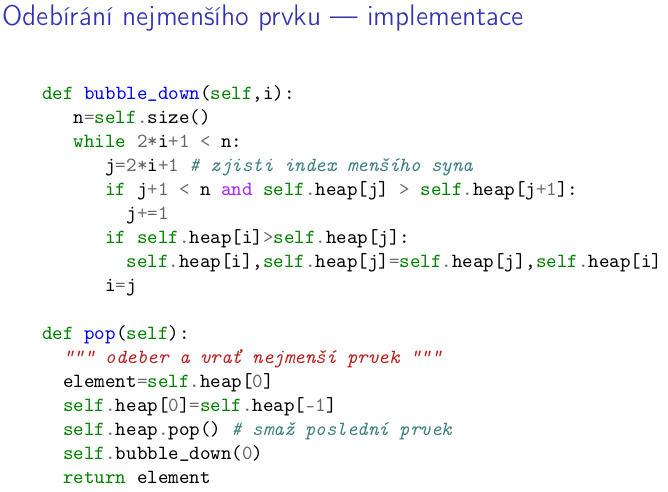
Vkládání do haldy

* Nový prvek přidáme na konec a probubláme kam patří

Odebrání nejmenšího prvku

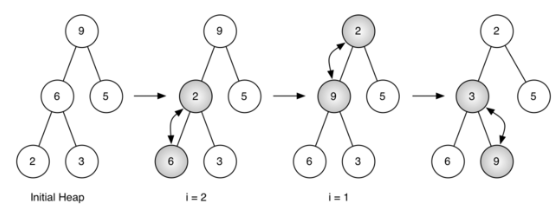
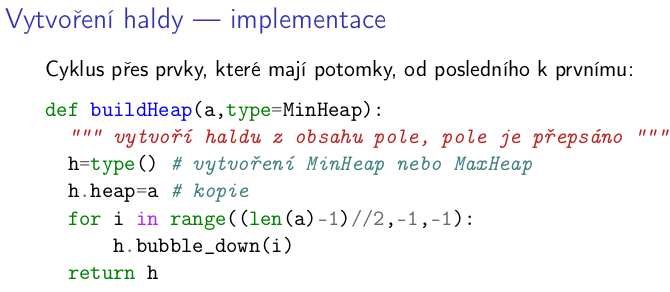
* Minimum je v kořeni stromu
* Halda (pole) se zkrátí o jeden prvek
* Tento prvek přesuneme do kořene a ‘probubláme’ dolů:
  + Rekurzivně měníme uzel s menším synem, dokud je porušená vlastnost haldy





Vytvoření haldy

Jak vytvořit haldu z pole

* Opakované volání insert — složitost O(n log n)
* Pole prohlásíme za haldu a opakovaně voláme bubble\_down
  + složitost O(n)

**Rozptylovací tabulka** (hash table)

= implementace množiny / asociativního pole

+ velmi rychlé vkládání i hledání, O(1) (nejhorší případ O(n))

– neudržuje uspořádání (hledání maxima/minima)

– méně efektivní využití paměti

Co je to hash?

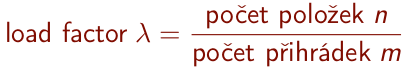
* hash — rozemlít, rozsekat, sekané maso, haše, . . . hašiš
* hash function — rozptylovací/transformační/hašovací/hešovací/funkce: objekt → celé číslo
* hash / fingerprint — haš/heš, otisk

Základní myšlenky a vlastnosti

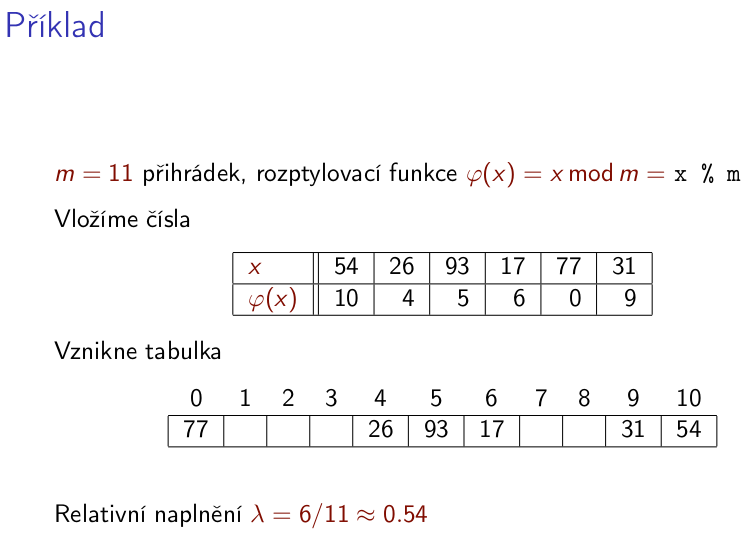
* pole m přihrádek (slots) pro ukládání položek
* položka (item) = klíč (key) + hodnota (value)
* klíč je unikátní
* je citlivá na volbu rozptylovací funkce a velikost tabulky
* potřebuje rozptylovací funkci a test na rovnost
* **rozptylovací funkce** (hash function) :

*φ: klíč → číslo přihrádky 0 . . . m − 1*

* více položek v jedné přihrádce = kolize (collision/clash)
* operace jsou rychlé, protože
  + víme, v které přihrádce hledat
  + v každé přihrádce je jen omezený počet položek

Relativní naplnění tabulky (load factor)

= Průměrný počet položek na přihrádku

* velké *λ* → hodně kolizí → zpomalení operací
* malé *λ* → hodně prázdných položek → nevyužitá paměť

Rozptylovací funkce (hash function)

* Nutné vlastnosti

- ‘Stejné’ klíče musí mít stejný otisk - *x = y ⇒ φ(x) = φ(y)*

- Neměnnost/nenáhodnost/konstantnost/opakovatelnost

* Požadované vlastnosti

- Rychlost výpočtu

- ‘Různé’ klíče mají mít pokud možno různý otisk —

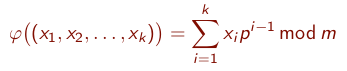
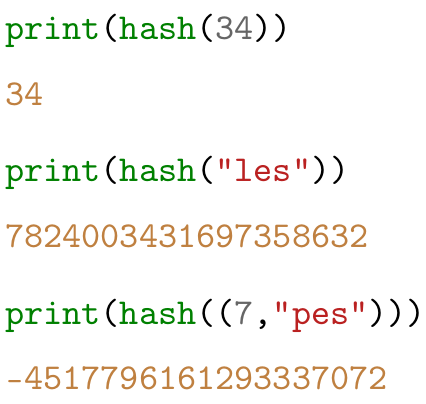
*x ≠ y ⇒ velká P[φ(x) ≠ φ(y)]*

- každý klíč jiný otisk = perfect hashing

- rovnoměrné využití všech přihrádek

- pravděpodobnost zvolení konkrétní přihrádky *1/m* (i pro strukturované vstupy)

- malé množství kolizí

* Kvalitu lze ověřit experimentálně.
* Souvislost s kryptografií a náhodnými čísly.
* Pro celá čísla *φ(x) = x mod m = x % m*
* Pro znaky *ord(c) % m*
* Pro k-tice ---------------------------------->

kde *p* je vhodné prvočíslo — dostatečně velké a nesoudělné s *m*

* Funkce **hash** (python) — pro neměnné hodnoty (immutable): čísla, řetězce, n-tice, logické hodnoty, funkce, neměnné množiny (frozenset), objekty. . .
  + nikoliv pro pole, množiny (set)
  + Vrací (velké) celé číslo
* **Použití hash fce:**
  + Rychlé ověřené rovnosti velkých objektů (DNA řetězce, otisky prstů, obrázky, . . . ):
    - Předpočítej otisk každého objektu v databázi
    - Pokud hash(x)=hash(y), pokračuj úplným porovnáním x a y

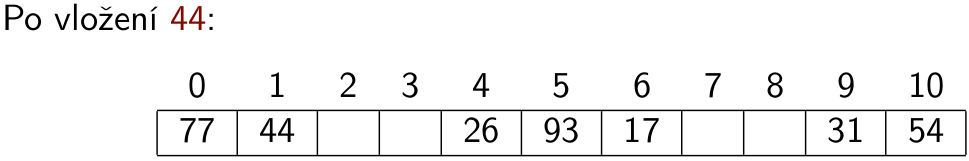
Velikost rozptylovací tabulky

* Vhodná velikost je prvočíselná — např. 11, 103, 1009 . . .
  + Jinak riziko kolizí pokud *φ(x) ∈ {k, 2k, 3k, . . . }*
* Dynamická realokace:
  + pokud se tabulka naplní (*λ > λmax* ) — vytvoříme větší tabulku *(m’ ≈ 2m)*
  + pokud se tabulka vyprázdní (*λ < λmin* ) — vytvoříme menší tabulku *(m’ ≈ m/2)*
* Možné hodnoty *m0 = 11, λmax = 0.75, λmin = 0.25*

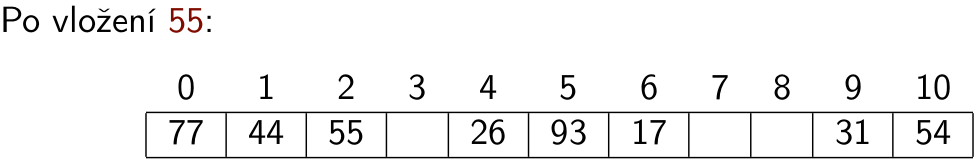
Řešení kolizí

= Co když dvě položky mají stejný otisk?

* **Zřetězení** (chaining)
  + Každá přihrádka je seznam (nebo pole)
  + Zaplnění *λ* může být > 1
* **Otevřené adresování** (open addressing)
  + Kapacita přihrádky je 1.Pokud je přihrádka *m0=φ(x)* obsazená, zkusíme jinou (*m1, m2*, ...)
  + Lineární zkoušení (linear probing) — zkusíme *mi = m0 + i*
  + Kvadratické zkoušení (quadratic probing) — zkusíme *mi = m0 + ai2 + bi*, např. *a = 1, b = 0*
  + Dvojité rozptylování (double hashing) — zkusíme *mi = m0 + iψ(x)*
  + Menší režie než zřetězení
  + Zaplnění *λ* nesmí být velké *(≈ 0.7)*
  + Rozptylovací funkce nesmí vytvářet shluky

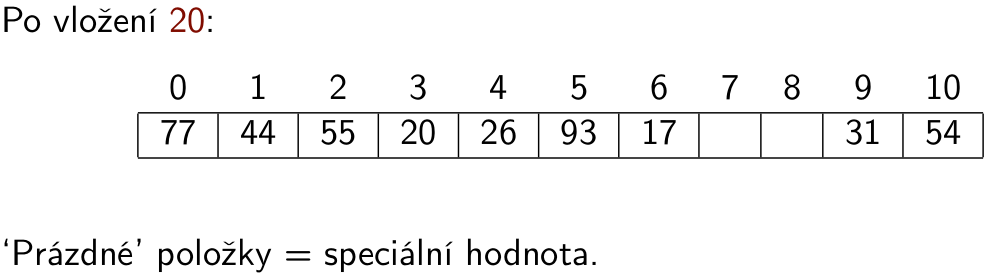
\_ ->

|

|

|

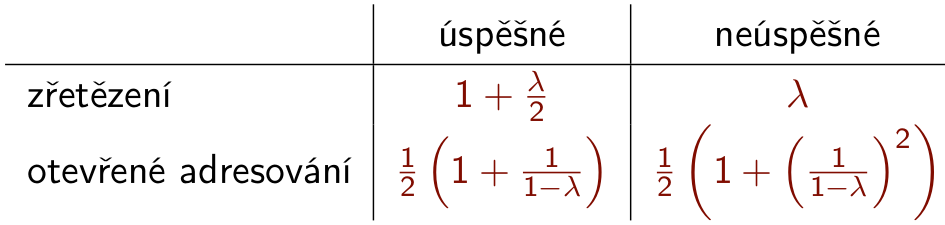
|

|

|

-

Počet porovnávání při hledání

Počet přístupů do paměti je větší o 1 + režie přihrádek (např. 2 přístupy na porovnání u spojového seznamu)

Mazání položek

* Zřetězení — smažeme ze seznamu přihrádky.
* Otevřené adresování — smazané položky označíme speciální hodnotou ‘přeskoč’.
* Mazání často není potřeba

**Odkazy na přednášky:**

[Úvod](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/01_uvod_beamer.pdf)

[Funkce, řetězce, moduly](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/02_funkce.pdf)

[Pole](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/03_pole.pdf)

[Složitost](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/04a_slozitost.pdf)

[Záznam, zásobník, fronta](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/06_fronta_a_zasobnik.pdf)

[Prioritní fronta, halda](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/08b_halda.pdf)

[Rozptylovací tabulka](https://cw.fel.cvut.cz/b181/_media/courses/b3b33alp/prednasky/09_rozptylovaci_tabulky.pdf)

[Stránka předmětu i s kódy atd.](https://cw.fel.cvut.cz/b181/courses/b3b33alp/prednasky/start)