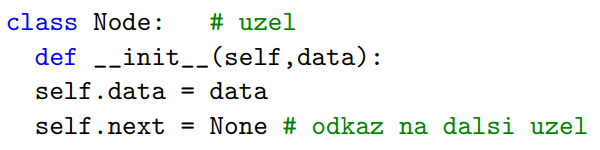
# I02

*Spojový seznam. Vyhledávání a řazení. Principy objektového programování. (Základy programování)*

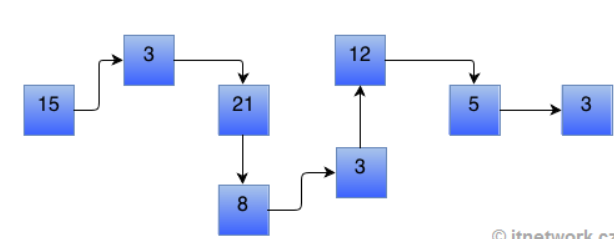
## Úvodní pojmy

* algoritmus – přesný, detailní a úplný postup
* program – zápis algoritmu v nějakém programovacím jazyce

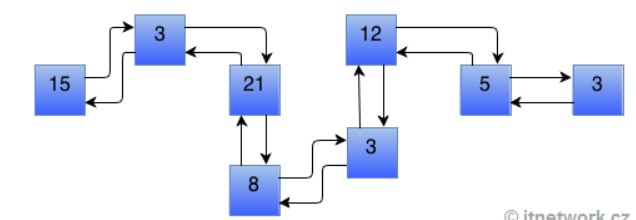
## Spojový seznam



* pole má fixní velikost, je třeba alokovat paměť
* v Pythonu je seznam (list), lze ale použít datový typ pole (array)
* funkce seznamu: insert(), remove(), index\_of(), size()
* spojový seznam realizuje seznam dynamické délky
* každý prvek má datovou část a ukazatel na další prvek
* první prvek je *head*

****

*Jednosměrný spojový seznam(singly linked list)*

****

Dvousměrný spojový seznam (doubly linked list)

## Třídící algoritmy

Máme určitou posloupnost prvků na vstupu a my se je snažíme setřídit podle určitého pravidla. Nejčastěji podle velikosti.

Máme dvě možnosti setřídění:

* vzestupně
* sestupně

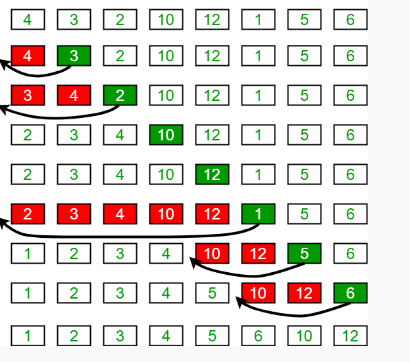
Třídění dělíme na:

1. **vnitřní třídění** -> třídění v paměti RAM
2. **vnější třídění** -> třídění mimo RAM; v případech, kdy dat je mnoho a nevejdou se do paměti, viz. větší databáze atd.

**Třídění přímým vkládáním (Insert Sort)**

Je to třídící algoritmus, kdy nesetříděné pole p1.

První prvek pole p1 budeme považovat za už setříděný. Vezmeme druhý prvek a koukáme doleva a hledáme pozici, kam ho vložit a pak posuneme prvky napravo.



- časová složitost: O(N^2) nejhorší případ, O(N) v nejlepším případě

- prostorová složitost: O(1)

Není moc dobrý, pomalý. Jednoduchý pro naprogramování.

**Bubble sort**

Algoritmus prochází pole a koukne na dva prvky vedle sebe. Pokud prvek i+1 je větší než prvek i. Dojde k prohození prvku i a i+1. Takto algoritmus prochází pole dokud není setříděné.

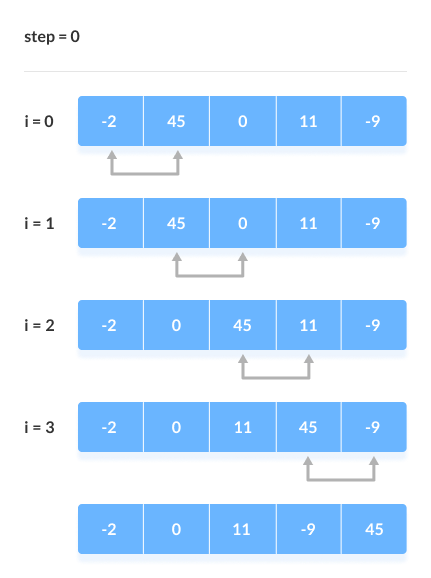
-časová složitost: O(N^2) < - nejhorší, O(N) <- setříděné

-prostorová náročnost: O(1) -< protože využívá jen jeden prvek pro změnu, tedy konstantní velikost

Nevýhody:

-často dochází k prázdným průchodům

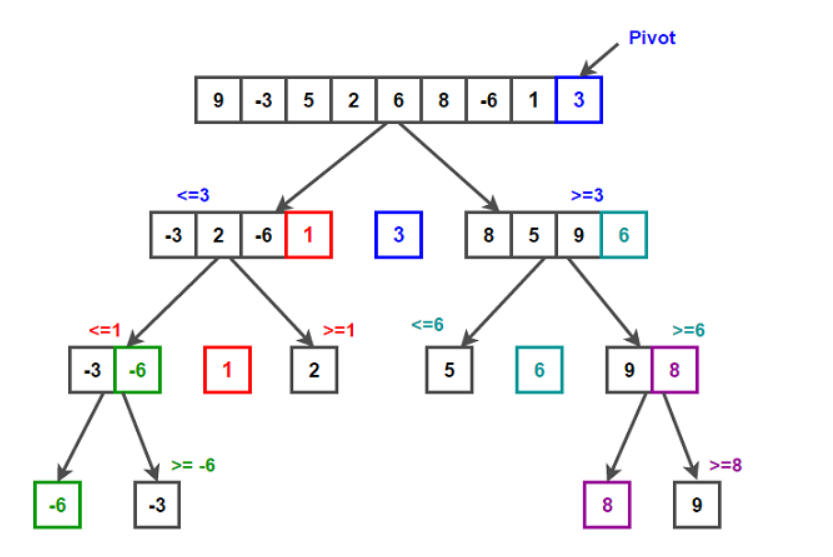
algoritmus je citlivý na konfiguraci dat



**QuickSort**

Označuje se jako rychlé třídění. Rekurzivní algoritmus.

Vyberu pivot a podle něj rozdělim prvky v poli na dvě části. Prvky menší dám doleva a prvky větší doprava. Pak vezmu první půlku a zvolim pivot a zase rozdělim tuto část na dvě půlky. A to samý s pravou půlkou. Takto pokračuju až se dostanu do fáze, kdy už v jednotlivých podčástech je jenom jeden prvek. A pak jedu od spoda a dávám ty jednotlivé podpole dohromady.



Ten pivot musím volit opatrně, nejčastěji se volí první nebo poslední prvek.

-časová složitost: O(N\*logN) <- nejlepší případ, O(N^2)

- prostorová složitost: O(N)

**Selection sort**

Je to jednoduchý třídící algoritmus.

Vstupní pole je imaginárně rozděleno na setříděnou (levý konec) a nesetříděnou část (pravý konec).

Algoritmus probíhá následovně:

Vezmu první prvek v poli 14 a hledám nejmenší prvek v poli. Je to 10. Prohodim je a vyberu druhý prvek 33 a hledám zase nejmenší prvek v poli. Je to 14, kterou nahradim 33.

Takto algoritmus běží do té doby, dokud není pole seřazené.







- časová složitost: O(N^2) <- nejhorší případ, O(N) <- nejlepší případ

-prostorová složitost: O(1)

**Merge sort**

Rekurzivní algoritmus

Tento algoritmus začne následovně:

-Rozdělí vstupní pole na dvě části a tyto dvě části rozdělí zase na dvě půlky a tak dál.

- Nakonec nám zbyde pole pouze s jedním prvkem uvnitř.

- jedu od konce a spojuju listy, kdy je rovnám podle velikosti.

Tedy mám dvě pole:

[27, 38] a [3,43]

Jedu takhle:

-kouknu na první prvek prvního pole 27 a srovnám ho s prvním prvkem druhého pole 3.

- 3 je menší než 27, vložim ho do výsledného pole [3]

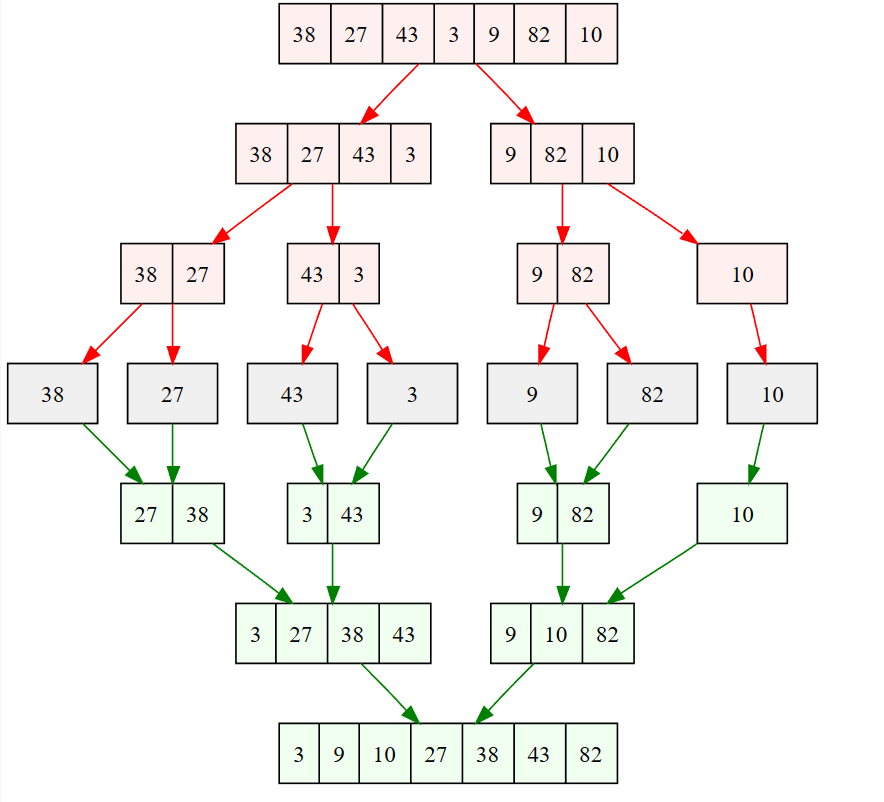
- vezmu prvek z druhého pole 43 a srovnám ho s 27

- 27 je menší než 43, vložim ho do výsledného pole [3,27]

- beru další prvek s prvního pole 38 a srovnám ho s 43, menší než 43, vložim do výsledného pole [3,27,38]

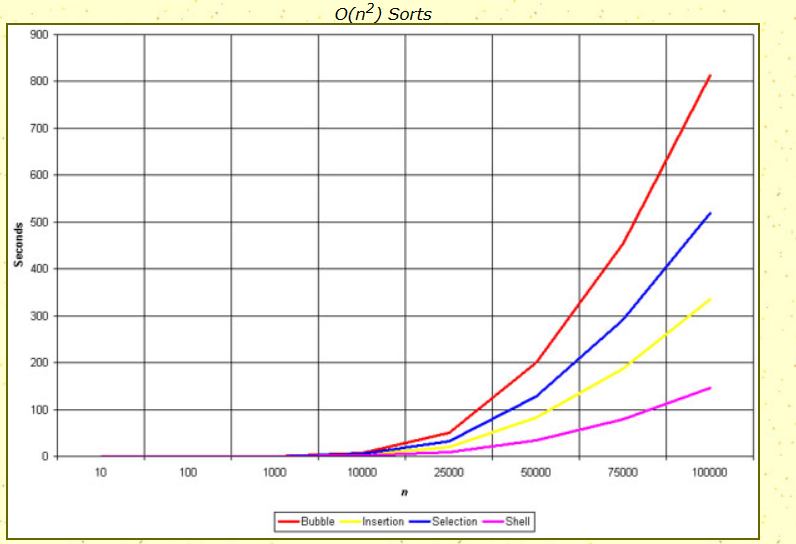
-další prvek v prvním poli nemám, tak naskládám zbytek do výsledného pole.

A takhle jedu dál, viz oraázek.



- časová složitost: O(N\*log(N))

- prostorová složitost: O(N)





## Vyhledávací algoritmy

Jsou to algoritmy využívané k vyhledání prvku ve vstupní sekvenci.

**Lineární vyhledávání**

Je to prohledávání, kdy postupně procházíme jednotilvé prvky a porovnáváme s danou hodnotou.

- časová složitost: O(N) <- nejhorší případ

**Binární hledání**

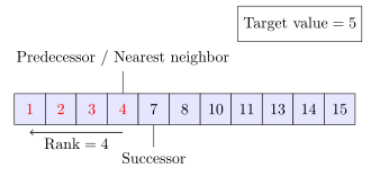
Neboli dělení na dvě půlky.

Vstupní data musí být už setříděná.

Mám setříděné vstupní pole a kouknu na prostřední prvek pole 7. Je menší než hledaný prvek 5, tak budu koukat doleva a na prvky napravo se nekoukám. Levý interval rozdělim zase na půlku a vyberu zase stranu podle velikosti mého hledaného prvku.

Tímto způsobem se postupně blížím k hledanému prvku.

-časová složitost: O(logN) <- nejhorší případ, O(1) <- nejlepší případ



Pro větší datasety je binární prohledávání lepší než lineární a naopak.

## Prohledávání grafů

* binární vyhledávací strom
  + struktura pro rychlé vyhledání porovnatelných dat
  + podporované operace: vložení prvku, odstranění prvku, dotaz na přítomnost prvku
  + rychlé operace (složitost *O(*log *n)* nebo lepší)
  + vlastnosti
    - klíč v uzlu není menší než všechny uzly v levém podstromu
    - klíč v uzlu není větší než všechny uzly v pravém podstromu
  + nejhorší případ – degenerovaný strom – hloubka *n-1* 🡪 skoro všechny prvky jsou v jednom podstromu, složitost podporovaných operací je pak *O(n)*
  + pro zabránění nejhoršímu případu lze stromy průběžně vyvažovat
* graf – uzly + hrany (orientované/neorientované)
  + reprezentace je možná pomocí matice sousednosti
  + hledání všech nejkratších cest v grafu – Floyd-Warshallův algoritmus
    - hledá všechny nejkratší cesty
    - dynamické programování – postupně aktualizujeme nejkratší cesty mezi vrcholy
    - nalezne matici délek nejkratších cest mezi všemi dvojicemi uzlů na základě matice sousednosti s nezápornými cykly
  + nejkratší cesty z daného uzlu – Dijkstrův algoritmus
    - na vstupu je orientovaný graf s nezápornými cenami hran a počáteční uzel
    - hledá nejkratší cestu do všech ostatních uzlů
    - prioritní prohledávání do šířky s aktualizací při nalezení kratší cesty

## Prohledávání stavového prostoru

* stavový prostor
  + množina stavů *S*
  + počáteční stav *s0*
  + množina cílových stavů *T*
  + seznam akcí *A*
  + přechodová funkce *f: S × A* 🡪 *S*
* hledáme sekvenci akcí a stavů, která převede počáteční stav do žádaného cílového stavu
* příklady: plánování cesty, procházení bludištěm, řešení sudoku, …
* prohledávání do hloubky (depth first search)
  + *„V každém stavu zkusím něco udělat. Když už to nejde, tak se vrátím a zkusím něco jiného.“*
  + pokud jsem v cíli, hotovo
  + existuje-li neprozkoumaný následník, rekurzivně ho prozkoumám
  + pokud neexistuje, vrátím se o jednu akci zpět
  + prozkoumané uzly označuji, abych se do nich nemusel vracet
  + pamatuji si cestu do aktuálního uzlu
  + nerekurzivní řešení: otevřené uzly ukládáme do zásobníku
  + vlastnosti
    - používá zásobník
    - malá paměťová náročnost *O(D)*, *D* je max. hloubka stavového prostoru
    - velká časová náročnost *O(bD)*, *b* je faktor větvení
    - vhodné pro fyzické prohledávání nebo pokud je změna stavu výpočetně náročná
    - je vhodné omezení maximální hloubky, v případě nenalezení řešení lze omezení zvýšit
    - nenajde vždy nejkratší řešení
    - při špatném rozhodnutí může náprava trvat dlouho
* prohledávání do šířky (breadth first search)
  + hledáme v pořadí délky sekvence akcí
  + nejprve prozkoumáme všechny sousedy, pak všechny jejich sousedy atd.
  + lze si představit jako šíření vlny z daného bodu
  + velká paměťová náročnost *O(bd)*, *d* je délka řešení
  + velká časová náročnost *O(bd)*
  + vždy najde řešení (bez omezení hloubky)
  + vždy najde nejkratší řešení
  + zásobník nahradíme frontou
* informované prohledávání (informed search)
  + víme, kam jdeme 🡪 začneme nejslibnějšími akcemi
  + urychlení řešení
  + může, ale nemusí zaručit nalezení řešení
  + prioritní (hladové) vyhledávání (greedy search)
    - akce mají ceny
    - pro každý stav odhadneme cenu dosažení minima
    - akce čekající na zpracování ukládáme do prioritní fronty a bereme je v pořadí podle odhadnuté ceny
    - nalezení řešení je zaručeno
    - není zaručeno nalezení minima
    - časová a prostorová složitost jako u prohledávání do šířky
  + algoritmus *A\**
    - prioritní hledání ignoruje cenu aktuální cesty, zavedeme tedy cenu cesty z kořene do aktuálního uzlu
    - pořadí vyhledávání je určeno celkovou cenou aktuální + zbývající (odhad) cesty
    - výhoda: neprodlužuje příliš dlouhé cesty dál
    - není zaručeno optimální řešení
    - časová a prostorová složitost jako u prohledávání do šířky
* problém splnitelnosti
  + speciální případ prohledávání stavového prostoru
  + stav se skládá z proměnných
  + cílové stavy musí splňovat podmínky
  + cesta dosažení cíle není důležitá
  + příklady: logické hádanky (8 dam na šachovnici, sudoku, aritmogram, …)
  + popis problému
    - množina proměnných, každá má svůj obor
    - množina podmínek
    - hledáme takové hodnoty proměnných, aby byly splněny všechny podmínky
  + algoritmy
    - hrubou silou
    - postupné přiřazování
      * stav je často velký – snažíme se nekopírovat
      * včasná eliminace špatných větví („pruning“)
    - propagace omezujících podmínek
    - lokální hledání
      * postupná optimalizace (postupně zlepšujeme předchozí odhad)
      * využití předchozích řešení
      * minimalizujeme počet konfliktů
    - strukturou problému
      * graf závislostí
      * rozklad na podúlohy (stromová struktura)

## Principy objektového programování

Objekt je základní abstraktní jednotkou používanou v objektovém programování. Pomocí objektu se snažíme napodobit objekt reálného světa (objekt je např. člověk nebo databáze). Objekt má svoje atributy a metody. Atributy a metody jsou úzce svázány.

**Atributy** jsou vlastnosti neboli pata, které objekt uchovává. Je to označení pro proměnné objektu.

**Metody** jsou operace, které můžeme s objektem provádět. Jsou to jinak řečeno funkce objektu. Metody mohou mít parametry i návratovou hodnotu.

Označení třída v objektivním programování vzor, podle které se vytváří objekt (instance).

Tedy například máme třídu člověk a od ní se vytvoří instance Josef a Martin. Obě instance mají jistě ty samé metody a atributy, jako třída (např. jmeno a vek) a metody (JdiDoPrace() a Pozdrav()), ale hodnoty v nich se liší (první instance má v atributu jmeno hodnotu "Josef" a ve vek 22, druhá "Martin" a 45).

OOP vychází ze třech základních principů:

* **zapouzdření** (encapsulation)
* **dědičnost** (inheritance)
* **polymorfismus**

**Zapouzdření**

Třída může obsahovat libovolný počet atributů a metod.

Zapouzdření je že metody a atributy jsou skryty uvnitř třídy. To znamená, že jsou dostupné pouze uvnitř třídy a né z vnějšku. Zapouzdření umožňuje udržovat a spravovat každý objekt nezávisle na jiném objektu. Díky zapouzdření nemohou jiné objekty měnit stav objektu přímo a způsobit tak chybu.

**Dědičnost**

Jedná se prostředek, který umožňuje vytvářet potomky, které budou dědit vlastnosti svých rodičovských tříd. To znamená, že můžeme vytvoříme třídu, která bude dědit atributy a metody své rodičovské třídy a k tomuto potomkovy přidávat další metody a atributy.

Prostě můžeme vytvářet znovu využitelné třídy.

Jinak řečeno, můžeme vytvářet nové třídy na základě starých.

**Polymorfismus**

Je to prostředek oop, kdy můžeme přepsat zděděné metody v potomků u dědičnosti.

Podstatou polymorfismu je tedy metoda nebo metody, které mají všichni potomci definované se stejnou hlavičkou, ale jiným tělem.