BPC-TIN

Otázky a odpovědi BPC-TIN
(Teoretická informatika)

Text: jedla
Korektura: –
Formátování: –

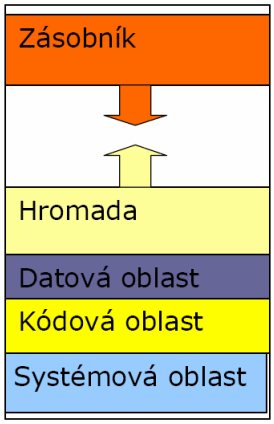
28. února 2021

Obsah

1 Správa paměti, statické přidělování paměti, dynamické přidělování paměti, garbage collector, reprezentace informace v paměti.

Každá paměť, která je přiřazena procesu se dělí na 4 základní bloky:

- Segment instrukcí(Code)/Kódová oblast
- Datový segment(Data)/Datová oblast
- Halda/hromada(Heap)
- Zásobník(Stack)



Code – je to množina strojových instrukcí, které naprosto jednoznačně provádí program. Dle těchto instrukcí PC postupuje ve výpočtu.
Data – v této části můžou být uložena data, která jsou známa při překladu programu (Hodnoty polí, konstant, proměnných, nějaké textové řetězce).
Bloky Code a Data jsou známy v době překladu a jejich velikost se v průběhu nemění.
Heap – slouží k alokaci dynamické paměti. Je založena na stromové datové struktuře. Obsahuje objekty a instance proměnných(atributy třídy).
Stack – je důležitý pro volaní správné funkcí. Funguje na principu LIFO. Funguje tak že se ukládají funkce do stacku, se kterými se posunuje taktéž ukazatel na pozici v registeru. Po dokončení té funkce je odstraněna ze zásobníku a ukazatel změněn na předchozí funkci. Obsahuje metody/funkce, lokální proměnné a reference na proměnné. Video pro lepší vysvětlení https://www.youtube.com/watch?v=uwV0hotRrLw.
Bloky Heap a Stack jsou dynamické a jejich velikost roste/zmenšuje u každého z jiným směrem v průběhu programu.

Statické přidělování paměti

Staticky se ukládají datové struktury, které jsou definovány při překladu programu. K jednotlivým paměťovým úsekům lze přistupovat pomocí názvu proměnné. V průběhu se adresa nemůže měnit.

Dynamické přidělování paměti

Dynamicky se paměť přiděluje na základě požadavku při průběhu programu. K dynamicky přidělenému paměťovému úseku se dá přistoupit pouze nepřímo pomocí ukazatele. Ukazatel je součástí statické nebo dynamické struktury. Dynamicky přidělovaná paměť se čerpá z vyhrazeného prostoru paměti počítače.

Dynamické přidělování paměti bez regenerace

Regenerace paměti je její pročištění od nepoužívaných částí paměti.
Dynamické přidělovaní paměti bez regenerace přiděluje požadované úseky postupně tak jak jsou za sebou umístěny až do vyčerpaní vyhrazené paměti. Využívá pracovního ukaza-

tele, který ukazuje na první adresu volné paměti. Nejčastěji pomocí operace "new"zapíše do paměti a změní ukazatel na novou hodnotu, která ukazuje na novou adresu volné části a v indikátoru paměti hodnotu obsazení označí true. Operace "free/delete"okamžitě neuvolní paměť, ale přepíše indikátor paměti na false. Kdy pak regenerace probíhá po větších částech.

Dynamické přidělování paměti s regenerací

Na rozdíl od dynamického přidělovaní bez regenerací se zde regeneruje pro každé operaci "free/delete". S tím přichází problém s fragmentací paměti. Po uvolnění paměti by tyto části mohli vytvářet sekvence malých, oddělených a přitom sousedních prvků. Často je defragmentace těchto volných bloků spojena s operací "free/delete". Snaží se slučovat volné úseky se sousedními volnými úseky.

Garbage collector

Je nejpokročilejším způsobem dynamického přidělovaní paměti. Oproti předchozím způsobům je méně efektivní. Skládá se ze tří fází:

- **Allocation** – přiděluje po sobě jdoucí úseky stejně jako metoda bez regenerace až do vyčerpání celého vyhrazeného prostoru.
- **Marking** – nastává pouze pokud je vyčerpán celý prostor. Prochází prostorem a vyhledává a označuje úseky, které nejsou aktivní a jejich návrat do společné paměti způsobí regeneraci.
- **Garbage collecting** – provádí se defragmentace přesunem všech uvolněných úseků do jednoho souvislého úseku. Tím se vytvoří nový souvislý úsek pro alokaci.

Tyto tři fáze se opakují dokola, dokud nedojde k situaci, že nový úsek není dostačující pro fázi alokace. Z tohoto důvodu dojde k ukončení programu.

Reprezentace informace v paměti – pokud je datový typ primitivní je uložen na přímo v paměti a u objektů je reprezentován pouze ukazatelem na místo v paměti.

2 Jazyk UML a objektově orientovaný návrh - dědičnost, generalizace, asociace 1:n, n:1, n:n, agregace a kompozice.

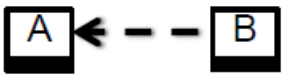
Jazyk UML je grafický jazyk pro popis programových systémů. Slouží pro vizualizaci, specifikaci, návrh a dokumentaci systémů. K zobrazení se využívají diagramy, kde nejčastěji používané jsou:

1. Strukturální
   (a) Diagram tříd
   (b) Diagram případů užití
   (c) Diagram komponent
   (d) Diagram nasazení
2. Behaviorální
   (a) Diagram aktivit
   (b) Diagram sekvencí
   (c) Diagram stavů.

Nejpoužívanější jsou diagramy tříd a případů užití. Diagram tříd popisuje strukturu systému, znázorňuje datové struktury a operace u objektů a souvislosti mezi nimi. Skládá se z tříd, rozhraní, abstraktních tříd. Tyto tři prvky se dále skládají z názvu třídy/rozhraní, atributů (rozhraní neobsahuje atributy), a operace (metody/funkce) Diagram případů užití se nejčastěji používá při komunikaci se zákazníkem a méně technicky znalou stranou. Skládá se z herců (actor) a případů užití a systém. Tyto strany jsou propojeny jak mezi sebou tak i sami se sebou pomocí těchto propojení – asociace, generalizace, rozšíření vztahu, vztah zahrnuje
Objektově orientovaný návrh je jeden ze způsobů jak reprezentovat informaci. Vychází z principů reálného světa, neboli je jednoduše srozumitelný pro člověka. Není spojen s žádným programovacím jazykem ale jazyk, který bude použit pro implementaci musí splňovat objektově orientované principy. Výhodou OO návrhu může být, že při návrhu lze určit co jaká část programu komunikuje z jakou a co každá dělá, takže se sníží počet chyb v kódu a tím i náklady. OO návrh se nezabývá konkrétní implementací ale pouze vazbami mezi objekty.
Kdy není vhodný OO přístup? Není vhodné ho použít jestliže na cílové platformě neexistuje překladač OO jazyka. Nebo potřebuji to v jazyce, který nepodporuje OOP. Přepis stávající kódu by byl neekonomický, hlavně u projektů s krátkou životností.

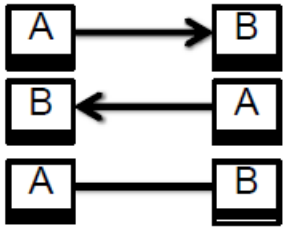
Vztahy mezi třídami

Závislost – je dynamický a zároveň nejslabší vztah. Ukazuje jak co na sobě závisí.

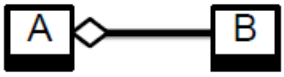


Asociace – je pevný vztah. Určuje vztah mezi dvěma prvky, které mohou existovat nezávisle na sobě. Asociace může mít směr od jednoho prvku k druhému nebo obousměrně.

Objekt ve směru šipky může nalézt odkaz na následujíc objekty.



Násobnost asociací – určuje kolik vazeb může mít danný objekt. Například 1:n může být 1 objekt a ten mít reference na n objektů ke kterým má asociaci (faktura:n * položka).
Agregace – je typ asociace. reprezentuje vztah typu celek–část. Zde je u celku umístěn kosočtverec. Celek je entita, která drží kolekci prvků. Část může existovat bez celku nebo být součástí jiných kolekcí.



Kompozice – je typ asociace. Je to nejsilnější vztah a je podobná agregaci. S rozdílem v tom že část nemá bez celku smysl. pokud zanikne celek zaniknou i části. U celku je násobnost vždy 1.

Dědičnost/generalizace – se využívá jestliže mají některé třídy společné vlastnosti. Tím například nemusíme vytvářet duplicitní kód. Směr šipky udává od koho třída dědí (obrázek b dědí z a). Díky tomuto lze jednodušeji rozšířit třídu o atributy a operace. Neplést si z asociací nijak nesouvisí.

3 Třídy složitosti paměťové a časové. Notace Theta. Notace Omega. Notace velké-O. Asymptotický popis složitosti algoritmu. Posouzení složitosti známých algoritmů řazení. Posouzení složitosti algoritmu vyhledávání. Srovnání lineárních a nelineárních struktur. Vztah časové a paměťové složitosti.

Složitost je vztah algoritmu k prostředkům (čas a velikost paměti). Paměťová složitost je závislost paměťových nároků na vstupních datech. Zatímco časová je dána hrubým odhadem počtu kroků, který daný algoritmus musí provést na základě délky vstupních dat.

Asymptotická složitost

Jelikož ne vždy lze určit přesnou složitost algoritmu tak byla vyvinuta asymptotická složitost. Tato složitost aproximuje chovaní funkce ze tří pohledu:

- Nejlepší případ Ω (Omega) – značí spodní hranici trvání algoritmu.
- Průměrný případ Θ (Theta) – odhaduje nejpravděpodobnější dobu trvání algoritmu.
- Nejhorší případ notace O (Omicron, big-O) – značí horní hranici trvání algoritmu. Je nejčastěji používaná.

Konstantní	$O(1)$	Nezávisí na velikosti vstupních dat
Logaritmická	$O(\log n)$	Počet operací odpovídá logaritmu např. vstup $1\,000\,000\,000 == 30$ operací
Lineární	$O(n)$	počet operací je závislý na velikosti vstupních dat
Kvazilineární	$O(n \log n)$	Zástupce quicksort
Kvadratická	$O(n^2)$	Např 500 vstupů $== 250\,000$ operací
Kubická	$O(n^3)$	Např. 200 vstupů $== 8\,000\,000$ operací
Exponenciální	$O(2^n)$	Exponenciální růst počtu operací
Faktoriální	$O(n!)$	Faktoriální růst počtu operací

Polynomiální algoritmy – Jsou takové algoritmy jejichž notace v big-O je ohraničena polynomiální funkcí shora. Spadají zde například $\log n$, $k{*}n$, $3n^3{+}4n$, $2{*}n \log n$ a podobné. Nepatří sem exponenciální, faktoriální a jim podobné. Abychom algoritmus označili jako efektivní jeho vykonání by mělo být možné v polynomiálním čase jinak ho lze označit jako neefektivní. Při vysokých hodnotách v kryptografii by ne-polynomiální algoritmy byli nepoužitelné.

Algoritmy řazení.

Třídění pomocí algoritmů Bubble Sort, Insert Sort a Select Sort má složitost $O(n^2)$. Algoritmus Quick Sort má složitost $\theta(n \log n)$, kdy v nejhorším případě může nabývat až $O(n^2)$. Algoritmus Merge Sort má složitost $O(n \log n)$.

Lineární a nelineární struktury.

Lineární – zde spadají pole, seznamy, zásobník, fronta.

Struktura	Přidat	Vyhledat	Smazat	Výběr dle indexu
Pole	O(n)	O(n)	O(n)	O(1)
Seznam	O(1)	O(n)	O(n)	O(n)
Pole proměnlivé délky	O(1)	O(n)	O(n)	O(1)
Zásobník	O(1)	–	O(1)	–
Fronta	O(1)	–	O(1)	–

Nelineární – zde můžou spadat stromy, které pokud jsou nějak vyvažovány, tak mají náročnost O($\log n$). Pokud se nevyvažují tak náročnost je O(n).

Vztah časové a paměťové složitosti.

Většina algoritmů je kompromisem mezi těmito dvěma druhy složitosti. Pokud bychom měli algoritmus s vysokými nároky na časovou složitost (aby byl co nejrychlejší) tak jeho paměťová složitost bude vzrůstat. Většinu algoritmů je nějakým způsobem optimalizovaná nebo ji lze optimalizovat.

Třídy složitosti

Vyjadřuje jak náročný je výpočet je nezbytný k vyřešení problému.
Rozdělení:

- Třída P – schůdné algoritmy. Jsou proveditelné v polynomiálním čase na deterministickém turingově stroji.
- Třída NP – neschůdné algoritmy. Je možné je provést v polynomiálním čase na **nedeterministickém** TS (nebyl doposud sestaven). Spadají pod ně i všechny algoritmy z třídy P
- Třída NP-těžké – přinejmenším tak těžké, jako nejtěžší z NP. Nemusí být vykonatelné pomocí TS.

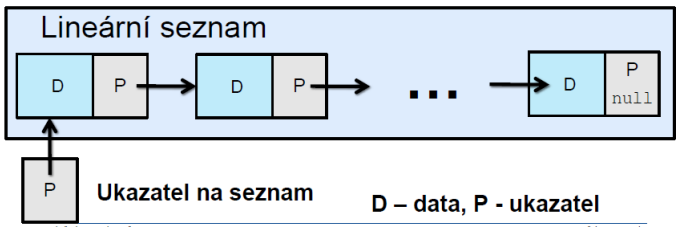
4 Abstraktní datový typ (ADT). ADT lineární seznam. ADT cyklický seznam. Operace vkládání, mazání a vyhledávání prvku v ADT lineární seznam. ADT zásobník, ADT fronta.

Abstraktní datový typ je množina druhů dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to nezávisle na konkrétní implementaci. Zjednodušeně ADT je reprezentováno rozhraním, kde uživatele tohoto rozhraní zajímá pouze, jak se používá a ne jak je implementováno. Poté co je konkrétní ADT implementován v programovacím jazyce stává se z něj datová struktura.
ADT dělíme podle počtu datových položek na statický a dynamický datový typ. Statický datový typ má neměnnou velikost a dynamický mění velikost dle provedené operace. Dále se dělí jestli mají jednoznačného bezprostředního následníka na lineární a nelineární. Lineární mají následníka a u nelineárních neexistuje přímý jednoznačný následník.
Dělení ADT:

- Lineární
  - Pole – statický
  - Seznam – dynamický
  - Zásobník – dynamický
  - Fronta – dynamický
- Nelineární
  - Strom – dynamický
  - Množina (Set) – dynamický

Lineární seznam

ADT lineární seznam je seznam, kde každý uzel má unikátního následníka. Výhodou lineárního seznamu je že má efektivní vkládání a mazání ale neefektivní přístup k prvkům. Na rozdíl od pole, kde je rychlá indexace prvků. Každý prvek obsahuje data a ukazatel na další prvek v seznamu.

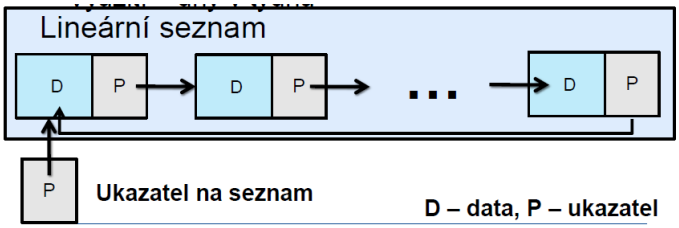


Možné operace s ADT seznamem:

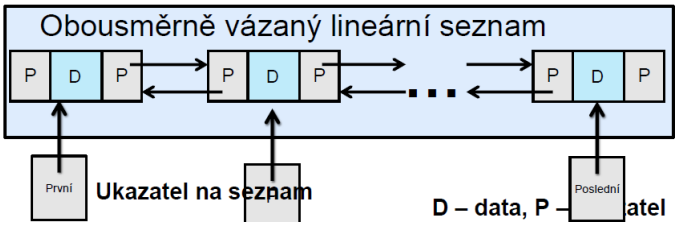
- Nalezení délky N seznamu
- Výpis všech prvků seznamu
- Vytvoření prázdného seznamu

- Získaní k-tého prvku ze seznamu
- Vložení novéhé prvku za k-tý prvek seznamu
- Smazaní prvku ze seznamu
- Nalezení následujícího prvku za aktuálním v seznamu
- Nalezení předchozího prvku před aktuálním v seznamu

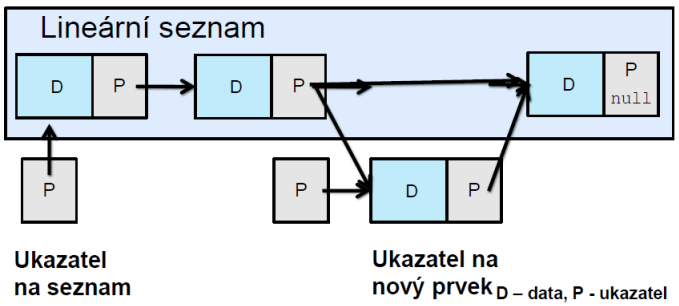
Cyklický lineární seznam – stejný jako lineární ale poslední prvek nemá ukazatel null ale odkaz na první prvek seznamu.



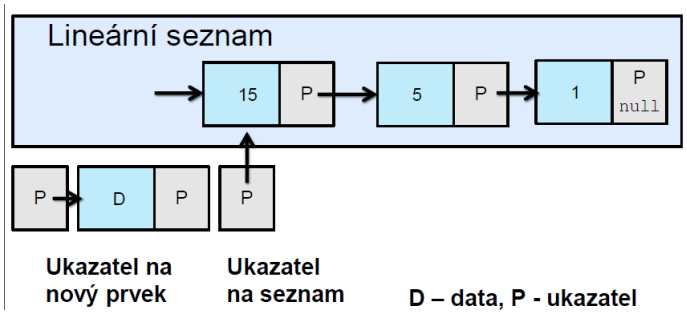
Obousměrně vázaný lineární seznam – nemá pouze ukazatel na další prvek ale i na předchozí. Umožňuje procházení v obou směrech.

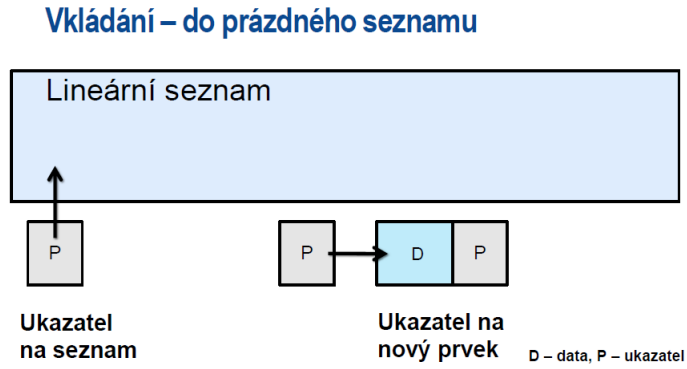


Vkládaní do lineárního seznamu – vložit prvek na určitou pozici funguje tak, že se najde pozice k kde se zde vloží prvek který bude odkazovat na prvek, který byl předtím na pozici k a v prvku k-1 se přepíše ukazatel na nový prvek. Operace fungují stejně jen jsou vázány na obě strany.

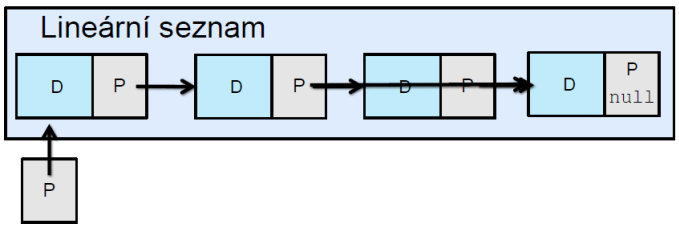


Pokud vkládáme na začátek tak se přepisuje ukazatel pole na první prvek a ve vkládaném prvku se přidá ukazatel na předchozí první prvek.





Mazaní v lineárním seznamu – mazaní prvku na pozici k funguje tak že prvek k-1 přepíše ukazatel na prvek k+1. U prvního prvku se přepíše pouze ukazatel na další prvek.



Vyhledávaní v lineárním seznamu – dá se vyhledávat podle prvku nebo podle dat. Pomalu iteruje seznamem dokud nenarazí na ten prvek nebo konec seznamu. U obousměrného se dá hledat od začátku a od konce.

Zásobník

Je dynamická datová struktura umožnující vkládaní a odebírání hodnot tak, že naposledy vložená hodnota se odebere jako první (LIFO). Základní operace jsou Vložený na vrchol, odebraní z vrcholu a test na prázdnost zásobníku.

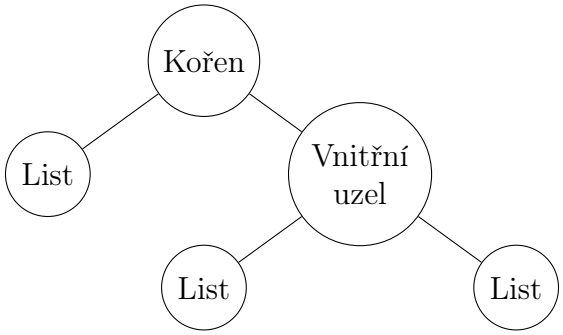
Fronta

Je dynamická datová struktura, kde se odebírají prvky v tom pořadí v jakém byli vloženy (FIFO). Základní operace jsou stejné jako u zásobníku. Existuje tzv. prioritní fronta, která funguje na principu fronty ale bere z ní podle priority.

5 Abstraktní datový typ strom. Abstraktní datový typ binární strom. Úplný binární strom. Abstraktní datový typ binární vyhledávací strom (operace vložení, odstranění, smazání uzlu stromu).

ADS typu strom – je nelineární dynamická abstraktní datová struktura. Každý uzel stromu může mít n potomků a zároveň má pouze jednoho předchůdce. Nejčastější zástupci jsou obecný strom nebo n-ární strom. Rozdíl je v tom, že n-ární má maximální počet potomků rovný n. Skládá se z uzlů, které se pojmenovávají:

- Kořenu – existuje pouze jeden. Je to uzel bez předchůdce.
- List – jsou to uzly, které nemají následníka
- Vnitřní uzel – jsou to uzly které nejsou kořenem ani listem stromu.

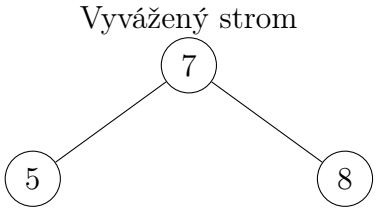


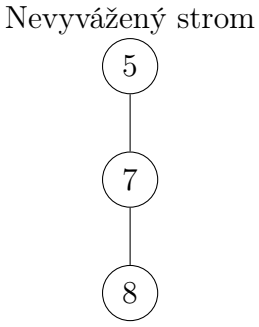
Binarní strom

Binární strom je strom, který má nanejvýše 2 potomky na jeden uzel.
Za **Úplný binární strom** se považuje binární strom, který se plní po úrovni hloubky z levé strany. Dokud není plně zaplněn levý potomek tak se neplné pravý. Hezká ukázka zde https://home.cs.colorado.edu/~main/supplements/pdf/notes10a.pdf

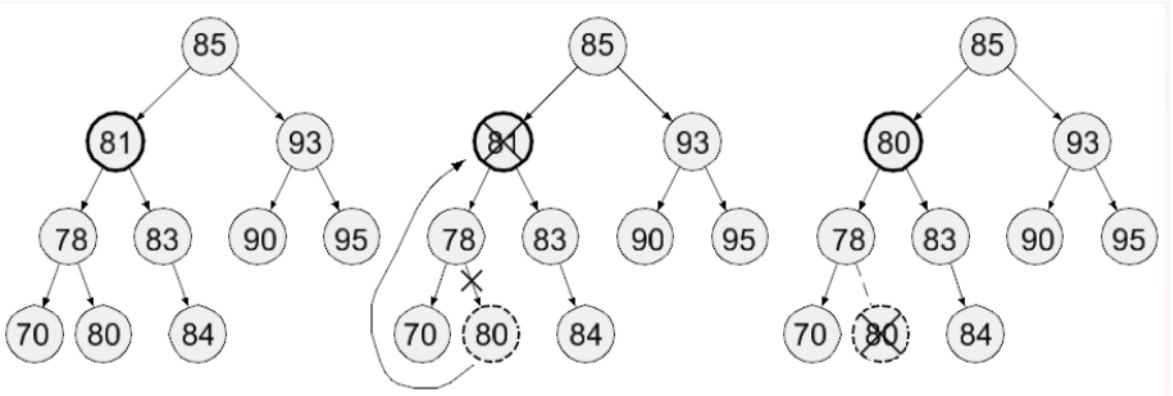
Binarní vyhledávací stromy

U těchto stromů musí platit, že levá část potomků musí být vždy menší než kořen a pravá část vždy vetší. Tzn. prvky jsou seřazeny. Tyto stromy mohou být nevyvážené ale častěji se také vyvažují, aby bylo dosaženo jejich optimální složitosti $O(\log n)$. Mohl by totiž nastat případ, kdy by ze stromu vznikl pouze strom.

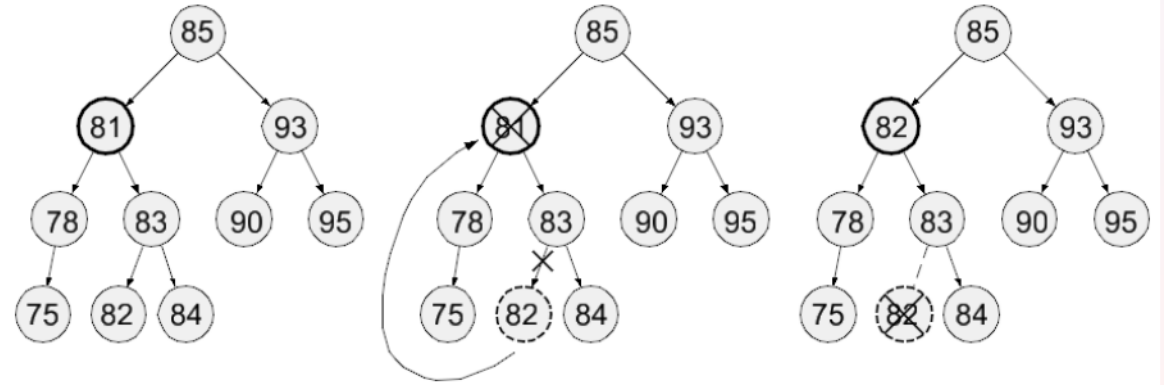




U nevyvážené stromu lze vidět, že by se jednalo spíše o seznam. U vyvážených bin stromů by rozdíl hloubky levé a pravé části měl být 0 nebo 1.
Vyhledávaní v binárních vyvážených stromech – postupuje se od kořene směrem dolů, kde se porovnává jestli prvek bude v levém nebo pravém uzlu pokud to už není námi hledaný prvek. Tak se postupuje vždy na každém navštíveném uzly dokud nenajdeme hledanou hodnotu nebo konec uzlu. Při hledání min/max se vždy vydáme do leva/prava dokud nenarazíme na uzel, který má svůj levý/pravý uzel roven null.
Vložení do binárních stromů – postupujeme stejně jako při vyhledávaní jen z rozdílem toho, že vkládáme až najdeme hodnotu null nebo úplně stejnou položku v tom případě záleží, jak se s tím v implementaci poradíme (přepíšeme/nepřepíšeme).
Odstranění prvků – nemají žádného potomka, má jednoho potomka, má dva potomky.
Nemají potomka – uzel se pouze odstraní a z jeho rodiče se smaže reference na něj.
Mají jednoho potomka – máme-li uzel tak jeho rodič zamění referenci toho uzlu za referenci potomka.
Mají dva potomka – dvě varianty. Buď pravého prvku (PL) nebo levého prvku (LP).
Pro pravý prvek to bude nalezneme uzel, který je nejvíc napravo v levém stromu a ten zaměníme za uzel, který chceme smazat. Takže nejpravější uzel v levém stromu převezme referenci z mazaného uzlu a rodič mazaného uzlu změní referenci na nejpravější uzel levého stromu.
Pro levý prvek je to podobné jen se vybírá nejlevější uzel z pravého stromu. Viz obrázky.
PL

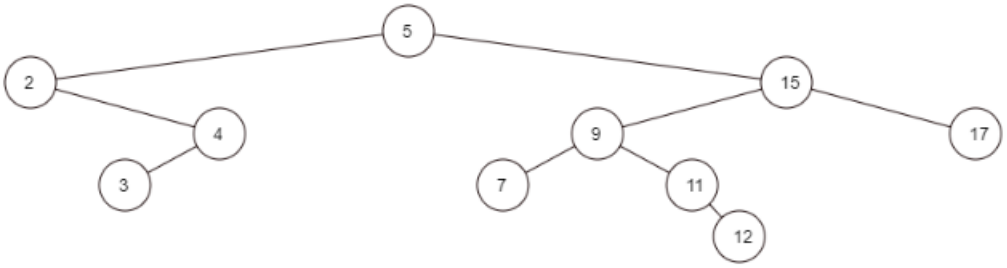


LP

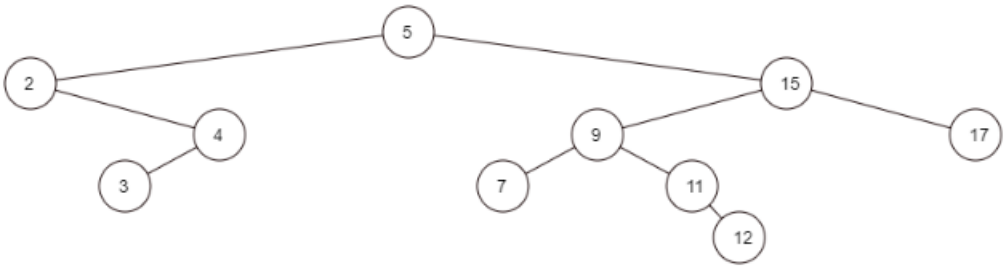


6 Průchody stromy in-order, pre-order, post-order.

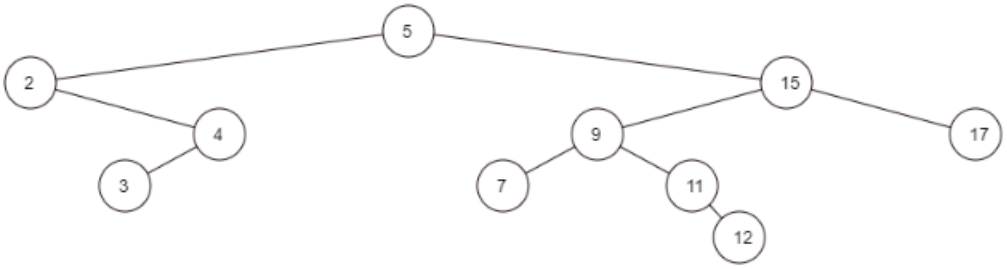
Pre-order – nejprve se zpracuje kořen, poté levý podstrom a nakonec pravý podstrom. Využívá se k vytvoření kopie stromu. Tak aby vypadal stejně.



Čtení pomocí pre-order bude posloupnost 5, 2, 4, 3, 15, 9, 7, 11, 12, 17.
In-order – nejprve se zpracuje levý podstrom poté kořen a nakonec pravý podstrom. Výsledkem je v příkladu čísel posloupnost od nejmenšího k největšímu. Využívá se k získaní popořadě jdoucích hodnot závislých tak jak máme vytvořen komparátor.



Čtení pomocí in-order bude posloupnost 2, 3, 4, 5, 7, 9, 11, 12, 15, 17.
Post-order – nejprve se zpracuje levý podstrom poté pravý a nakonec kořen. Využívá se k mazání stromu od listů ke kořenu.



Čtení pomocí post-order bude posloupnost 3, 4, 2, 7, 12, 11, 9, 17, 15, 5.

7 Problematika nevyvážených stromů. Vyvažování stromů AVL - rotace: jednoduchá levá, jednoduchá pravá, dvojitá levá, dvojitá pravá. Red-Black stromy. Posouzení z pohledu časové a paměťové složitosti. ADT hashovací tabulky. Řešení kolizí hashovacích tabulek. Srovnání výkonnosti binárních vyhledávacích stromů a hashovacích tabulek.

Nevyvážený strom je problematický v tom, že jeho složitost může klesnout až na $O(n)$ z $O(\log n)$. Z tohoto důvodu má smysl stromy vyvažovat.
Vyvážený strom má hloubku levého a pravého podstromu rovnou vždy 0 nebo 1. Pokud má hloubku větší tak se označuje za nevyvážený a z pohledu problematiky by se měl vyvážit například metodou AVL nebo Red-Black stromy.

Vyvažování

AVL stromy – mají výhodu v tom, že jsou dobře vyvážené ale hrozí zde problém mnohonásobné rotace. Takže vkládání může být méně efektivní. Mají velice efektivní vyhledávání.
Vyvažuje se dvěma typy rotace, u kterých pak dále rozlišujeme stranu. Vizualizace https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
Red-Black stromy – nejsou tak dobře vyvážené jako AVL, ale řeší problém mnohonásobné rotace. Efektivní vkládání, ale méně efektivní vyhledávání.
Při vyvažování platí pravidla:

- Každý uzel je červený nebo černý.
- Kořen je vždy černý.
- Všechny listy jsou černé.
- Potomci červeného jsou vždy černí.
- Každá cesta z libovolného uzlu do jeho podřízených listů obsahuje stejný počet černých uzlů.

Vizualizace https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Hashovací tabulka

Hashovací tabulka spojuje klíč s odpovídající hodnotou. Klíč je vypočítán z obsahu položky tak aby byl klíč co nejjednoznačnější a nedocházelo ke kolizím, vše vychází z pravděpodobnosti o hashovacích funkcích. Využívají se nejčastěji v databázích nebo k rychlému vyhledávaní v polích.
Vkládání – Z vkládaného prvku se se udělá hash. Tento hash se přiřadí do pole, na místo, které mu odpovídá. Jestliže je místo obsazeno přiřadíme mu další volné místo dle algoritmu.
Vkládání – Při vyhledávání využijeme klíče, který nám vrátí index položky. Jestliže na odpovídajícím indexu se nenachází daná položka tak pomocí algoritmu vypočteme, kde je další místo, kde se položka může nacházet.

Hashovací tabulka – řešení kolizí

Řetězení tabulek – pokud dojde ke kolizi vložíme prvky do lineárního seznamu za sebe.
Otevřené adresování linear probing – pokud dojde ke kolizi, tak prvek vkládáme na další možné volné místo. Zde si musíme hlídat, aby se tabulka celá nenaplnila.
Otevřené adresování double hashing – podobné linear probing jen se neposunujeme na další volné místo ale hashujeme dále.

Hashovací tabulka vs binární strom

Hashovací tabulka je rychlejší na vyhledávání pokud je dobře napsána hashovací funkce ve špatném případě lze dojít k O(n). Nevýhoda je, že nelze vyhledávat pokud máme jen částečný klíč nebo potřebujeme v nějakém intervalu. Další nevýhodou je složitost vytvoření hashovacích tabulek oproti stromu.

8 Jednoduché a pokročilejší řadící techniky a jejich srovnání. Stabilita řadícího algoritmu. Bubble sort. Insertion sort. Selection sort. Shell sort. Merge sort. Heap sort. Quick sort.

Typy řazení

Řazení výběrem – najde se vždy nejmenší se zbývajících prvků a uloží se na konec už seřazených.
Řazení vkládáním – z neseřazené množiny se postupně bere prvek po prvku a vkládá se na správné místo přičemž začáteční množina už seřazených je prázdná.
Řazení záměnou – v množině najdeme vždy dvojici, která je ve špatném pořadí a přehodíme ji.
Řazení slučováním – vstupní množinu rozdělíme na části, které se seřadí. Tyto seřazené části se poté sloučí takovým způsobem aby byli seřazené.

Porovnání algoritmů

Třídění pomocí algoritmů Bubble Sort, Insert Sort a Select Sort má složitost $O(n^2)$. Algoritmus Quick Sort má složitost $\theta(n \log n)$, kdy v nejhorším případě může nabývat až $O(n^2)$. Algoritmus Merge Sort má složitost $O(n \log n)$.

Stabilita řízení

Dělíme na stabilní a nestabilní. Vstupní data můžou obsahovat několik prvků se stejnou hodnotou. Podle vzájemné polohy těchto prvků před a po seřazení se rozlišují tyto druhy. Stabilní algoritmus zachovává vzájemné pořadí položek se stejnou hodnotou a u nestabilního jejich zachování není zaručeno. Stabilní je vhodné využívat pokud řadíme podle dvou parametrů například jméno příjmení. Př. jestli řadíme osoby podle křestního jména a poté podle příjmení tak Stabilní algoritmus by měl odpovídat očekávání. (První Karel Novák a následoval by Václav Novák). Pokud bychom na toto použili nestabilní algoritmus tak by druhé řazení mohlo zpřeházet výsledky prvního. (První by mohl být Václav a až za ním Karel.)

Bubble sort

Je jednoduchý stabilní řadící algoritmus se složitostí $O(n^2)$.
Porovnávají se dva sousední prvky, pokud je nižší číslo nalevo od vyššího tak se prohodí. A tak probublávají postupně dokud se neseřadí. Pokud jsou čísla v průběhu už správně seřazena tak je neprohodí ale postupuje dále.
Animace https://www.algoritmy.net/article/3/Bubble-sort

Insertion sort

Je jednoduchý stabilní řadící algoritmus se složitostí $O(n^2)$.
Máme jeden prvek a ten vložíme do už seřazené množiny, která je prázdná. Vezmeme následující prvek a ten umístíme na správné místo v již seřazení množině. A takto pokračujeme dokud nedorazíme nakonec.
Animace https://www.algoritmy.net/article/8/Insertion-sort

Selection sort
Je jednoduchý nestabilní řadící algoritmus se složitostí $O(n^2)$.
Funguje na principu výběru největšího prvku, který pak dá na začátek. Pak vezme druhý největší a ten dá nakonec seřazené části.
Animace https://www.algoritmy.net/article/4/Selection-sort

Shell sort
Je nestabilní kvadratický řadící algoritmus se složitostí $O(n^2)$.
Funguje na principu insertion sort. Rozdíl je v tom že z počátku neřadí prvky které jsou vedle sebe ale prvky které mají určitou mezeru mezi sebou. Tato mezera se poté každou iteraci snižuje. Jak dojde na mezeru 1 tak probíhá už pouze insertion sort.
Animace https://www.algoritmy.net/article/154/Shell-sort

Merge sort
Je stabilní složitý řadící algoritmus se složitostí $O(n \log n)$.
Dělíme pole neustále na poloviny dokud nemáme pole o jednom prvku. Jakmile mám rozděleno na jednotlivé prvky tak porovnám prvky vzájemně dva vedlejší prvky. Takže vzniknou pole o dvou prvcích, které jsou už seřazené. Takto postupuji až k začátku dělení. Jakmile má pole více elementů tak porovnám první prvek prvního pole s prvním prvkem druhého pole a seřadím je pokud je větší porovnám s druhým prvkem prvního pole. Pokud je zase vetší tak je seřazeno. Pokud je menší tak ho přiřadím před druhý prvek prvního pole a pokračuji s druhým prvkem druhého pole. Ten už porovnávám pouze s prvkem který byl vyšší než první prvek druhého pole. Tento proces opakuji dokud nedojdu k seřazenému poli.
Video https://www.youtube.com/watch?v=qdv3i6X0PiQ

Heap sort
Je nestabilní složitý řadící algoritmus se složitostí $O(n \log n)$. Je jeden z nejefektivnějších řadících algoritmů.
Funguje na principu prioritní fronty (stromová struktura). Prvně vytvoříme úplný binární strom (řazení vždy zleva doprava). Přetvoříme tento strom na binární haldu/heap. nový heap vytvoříme tak že probubláváme prvky stromu od nejnižších podstromů dle toho, jestli použijeme min/max heap při min haldě bude kořenem vždy nejmenší číslo a při max to bude naopak. Z vytvořeného binárního stromu vezmeme kořen a dáme ho do množiny seřazených prvků a v stromu přepíšeme kořen na poslední element stromu. Tento strom zase řadíme dokud klidně opakovaně dokud nemáme na každém kořenu min/max hodnotu. Tak pokračujeme dokud nezůstane žádný prvek
Video https://www.youtube.com/watch?v=LbB357_RwlY

Quick sort
Je nestabilní složitý řadící algoritmus se složitostí $O(n^2)$ ale očekávaná složitost je $\Theta(n \log n)$.
Prvně vybereme pivot (se špatně vybraným pivotem se může zhoršit složitost). Dále všechny menší prvky než pivot přesuneme na jednu stranu a všechny vetší na druhou. Dále vybereme nový pivot z prvků na levé straně a provedeme stejný postup co ze základním polem. To stejné uděláme s pravou stranou. Pivot vždy zůstává na místě a nic se s nedělá, jelikož je považován za setříděný. Tak postupuje dokud nám na každé straně nezůstane jen 1 prvek.
Video https://www.youtube.com/watch?v=ZHVk2blR45Q

9 Grafy, formální definice. Vyhledávání v grafech. Algoritmus BFS (prohledávání do šířky). Reprezentace BFS v paměti. Algoritmus DFS (prohledávání do hloubky). Omezené prohledávání do hloubky (DLS). Iterativní prohledávání do šířky (IDLS), Dijkstrův algoritmus (Uniform Cost Search), A*

Graf/ teorie grafů

Graf je definován jako uspořádaná dvojice množiny vrcholů V a množin hran E (*G(V,E)*). Kde vrcholy (vertices/nodes) jsou uzly grafu a hrany (edges) jsou spoje mezi vrcholy. Graf může být jakýkoliv rovinný nebo prostorový útvar, který bude znázorňovat důležité vazby mezi důležitými prvky (vrcholy).
Hrana – znázorňuje vztah mezi vrcholy. Rozlišujeme na hrany orientované a neorientované. U orientovaných lze stanovit počáteční a koncový vrchol a u neorientovaného to nelze. Hrany lze ohodnotit, kdy hodnota může například představovat délku, zátěž atd.
Vrchol – je nějaký prvek, který chceme spojit s druhým vrcholem pomocí hrany tak aby nám vznikl graf (ne vždy tyto prvky musí být spojené). Vrchol lze ohodnotit, kde hodnota může představovat například počet připojených hran. Pokud využije zrovna označení připojených hran tak tím získáme stupeň vrcholu. Pokud vybereme vrchol s největší hodnotou ze všech vrcholů tak tuto hodnotu lze nazvat stupeň grafu.
Dělení grafů dle typu hran:

- Neorientované grafy – obsahují pouze neorientované hrany. Hrana je obousměrná.
- Orientované grafy – obsahují pouze orientované hrany. Hrana je pouze jednosměrná.
- Smíšené grafy – obsahují oba typy hran.

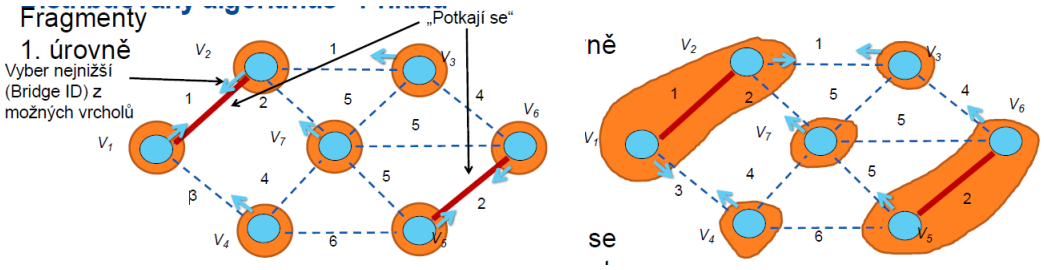
Vyhledávaní v grafu

Druhy prohledávaní:

- Slepé prohledávání – prohledávají náhodně bez přemýšlení.
- Informované metody – snaží se odhadnout, kudy pokračovat ve vyhledávání aby byli co nejdříve v cíli.

Nalezení kostry grafu spanning tree – kostra grafu je nějaký strom, který neobsahuje cykly. Kostra se vytváří tak, aby propojila všechny body a celková váha byla co minimální. Při hledání kostry s co nejmenší váhou tento postup nazýváme minimal spanning tree. Nejznámějším algoritmem je Kruskalův algoritmus a distribuovaný algoritmus.
Kruskalův algoritmus – funguje na principu shlukování dvou množin hran. Ze všech hran se vybere hrana s nejnižší hodnotou a množiny vrcholů jenž hrana propojuje se seskupí do jedné množiny. Tak se postupuje tak dlouho dokud všechny vrcholy nejsou v jedné množině. Pokud cesta propojuje vrcholy ze stejné množiny vrcholů tak se nepoužije a pokračuje se z následující hranou. Využívá se pokud známe celou topologii grafu.
Animace https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Distribuovaný algoritmus – zde se pracuje na principu, že se kostra tvoří na každé množině zároveň. Z každého vrcholu se vyrazí směrem po nejmenší hodnotě hrany. Pokud se vrcholy na cestě potkají tak vytvoří množinu pokud se nepotkají nic se neděje. Dále se pak vysílají znovu po nejmenší hodnotě hrany. Toto se děje dokud se celá kostra nenajde. Tento algoritmus se nejčastěji využívá v telekomunikačních sítích, kde neznáme celou topologii grafu.



Slepé prohledávání

BFS – využívá frontu do kterého prvně vloží vstupní vrchol. Z tohoto vrcholu se vezmou všichni sousedé a vloží se do fronty, z které se bere postupně a jejich sousedé se přidávají do fronty. Při tomto se každá už navštívený vrchol ukládá do nějakého pole/seznamu navštívených abychom ho nenavštěvovali znovu.
Video https://www.youtube.com/watch?v=oDqjPvD54Ss
DFS – pracuje na principu, že prozkoumává prvně vrchol na jednu stranu a v ní pokračuje dokud nedojde na konec nebo nedorazí do už navštíveného vrcholu. Využívá zásobník.
Video https://youtu.be/pcKY4hjDrxk?t=279
BFS vs DFS rozdíl je že DFS je méně paměťově náročné ale BFS je optimální.
DLS – vychází z DFS ale je omezen na maximální hloubku.
IDLS – iterativní prohledávaní do hloubky. Vychází z DFS ale v každé iteraci se prohledává jen o hloubku níž. Video https://www.youtube.com/watch?v=Y85ECk_H3h4
Dijkstrův algoritmus – využívá prioritní frontu a nějaký seznam navštívených vrcholů. Z prvního vrcholu se přidají všechny soudní vrcholy z jejich hodnotou hrany do prioritní fonty. Z prioritní fronty se vždy vezme nejmenší prvek. Z nejnižšího vrcholu se prozkoumají znovu sousedé ale tentokrát se nepoužije pouze vzdálenost mezi nimi ale přičte se už i vzdálenost k aktuálnímu vrcholu. Tak se bere dokud se nedorazí k cílovému vrcholu.
Video https://www.youtube.com/watch?v=GazC3A4OQTE

Informovaná metoda

A* – využívá prioritní frontu a nějaký seznam navštívených vrcholů. Každému z vrcholu přidělíme vzdálenost od cíle. Postupuje se stejně jako u Dijktrova algoritmu ale s tím rozdílem že sčítáme vzdálenost + hodnotu hrany. Dle této sečtené hodnoty vybíráme další navštívený prvek z prioritní fronty.
Video https://www.youtube.com/watch?v=ySN5Wnu88nE

10 Evoluční algoritmy. Genetické algoritmy, genetické programování. Pojmy populace, mutace, křížení, chromozom. Princip evolučních algoritmů.

Optimalizace a jejími úlohami se setkáváme při řešení praktických úloh, při kterých hledáme to nejlepší možné řešení. O nejlepším řešení se rozhodujeme dle parametrů.

Evoluční algoritmy

Je jeden ze způsobů optimalizace. Byli vytvořeny, aby sjednotili optimalizační metody co využívají „evoluční“ principy. Evoluční algoritmy zastřešují pod sebou řadu přístupů využívající modely biologické evoluce. Tyto modely jsou přirozený výběr (výběr silnějšího jedince, podle fitness funkce), náhodný genetický drift (mutace, decimuje jedince s vysokou fitness) a reprodukční proces (křížení jedinců). Spadají zde přístupy jako genetické algoritmy, genetické programovaní, evoluční strategie, evoluční programovaní.
Obecný evoluční algoritmus funguje zjednodušeně, že prvně máme nějakou základní populaci. S této populace uděláme výběr jedinců. Vybrané jedince zkřížíme. Tyto jedinci následně zmutují a vytvoří se nová populace. Tyto kroky se opakují v N iteracích. Kde ukončení stanovuje předem stanovený počet iterací, dosažení požadovaného jedince popřípadě minimální změnou fitness populace po několik iterací.

Genetické algoritmy

Genetické algoritmy jsou založeny na Darwinově myšlence že přežije nejsilnější. Využívá metody křížení, mutace a selekce.
TODO Kódování řetězci má také svou analogii v gentice, kdy v podstatě řetězce odpovídají chromozómům, jednotlivé pozice v řetězci jednotlivým genům a konkrétní hodnoty na těchto pozicích pak alelám.
Genetické algoritmy pracují tak že prvně vyberou nějakou populaci přípustných řešení. Abychom s nimi mohly pracovat musíme je převést do řetězce/pole. Poté přichází na řadu výběr jedinců, kteří se budou podílet na nové generaci. Vetšinou to probíhá tak, že ohodnotíme každého jedince pomocí fitness funkce a na základě tohoto parametru vybereme nejvhodnější kandidáty (metod výběru je více). Tito vybraní jedinci se mezi sebou kříží. Nakonec každý kandidát mutuje.
Výhody GA jsou že nevyžadují žádné speciální znalosti o cílové funkci, jsou odolné, aby nesklouzli do lokálního optima.
Nevýhody GA jsou že mají problém s nalezením přesného optima, vyžadují velké množství vyhodnocování cílové funkce.

Genetické programování

Hledá samotnou funkci a ne její parametry. Oproti GA se liší v reprezentaci jedinců. V GP jsou jedinci vytvořeni stromem s proměnlivou délkou chromozomu. Zjednodušeně je to převedení GA do programovacích jazyků.

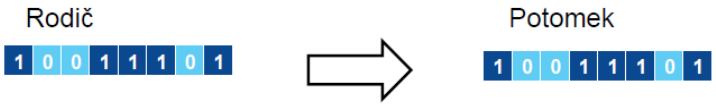
Základní pojmy

Populace – je množina jedinců o určité velikosti, z které následně vybíráme jedince na operace (křížení, mutace atd) při evoluci. Konkrétní populace se nazývá genotyp.
Jednice – člen populace, který je definován jedním chromozomem.
Chromozom – je to řetězec tvořený geny, který má za úkol se odlišit od zbytku populace

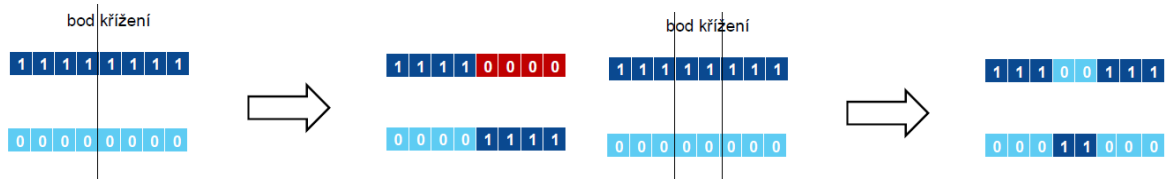
(DNA). Lze ho kódovat např. binárně, reálnými čísly, znaky, objekty(instance třídy v programovaní).
Gen – na i-té pozici reprezentuje stejnou charakteristiku v každém jedinci.
Alela – hodnota které může nabývat gen (např 0, 1).
Fitness funkce – kvantitativně vyjadřuje kvalitu každého řešení. Např. dosažení požadované přesnosti algoritmu, množství času potřebné pro výpočet algoritmu, množství chyb mezi skutečným a požadovaným výstupem.
Více druhů metod vytvoření fitness funkce (Hrubá, standardizovaná, přizpůsobená, normalizovaná)
Selekce ruletový výběr varianta 1 – pravděpodobnost výběru závisí na kvalitě jedince (kolik místa na ruletě zabírá). Pokud jedince bude převyšovat ostatní o hodně nové populace budou tvořeny jeho geny, k tomu se dají použít techniky potlačení/podpory.
Selekce ruletový výběr varianta 2 – jedinci jsou seřazeni vzestupně podle hodnoty fitness a velikost místa je určena rovnicí. Tímto se potlačují nadprůměrní jedinci, kteří by negativně ovlivňovali další generace.
Selekce turnajový výběr – náhodně se vybere n jedinců a postupným porovnáváním je vybrán nejlepší.

Metody křížené

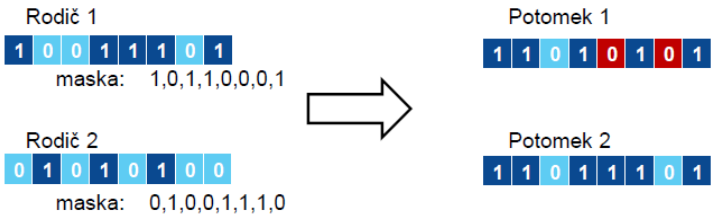
Elitářství – zaručuje monotonní hodnotu fitness nejlepšího jedince, předchází ztrátě nejlepšího řešení.



Křížení n-bodové – dělí se v n bodech a vymění se mezi sebou.



Křížení uniformní – rozvrací kód chromozomu, dobrý pro vnešení diverzity.



Mutace – Aplikuje se s malou pravděpodobností, důležitá pokud máme málo jedinců.