BPC-TIN

Teoretická informatika

Otázky ke státnicím

Bakalářský obor Informační bezpečnost, FEKT VUT

https://github.com/VUT-FEKT-IBE/BPC-IBE-SZZ

Text: kámen u cesty
Korektura: kámen u cesty, czechbol

28. května 2023

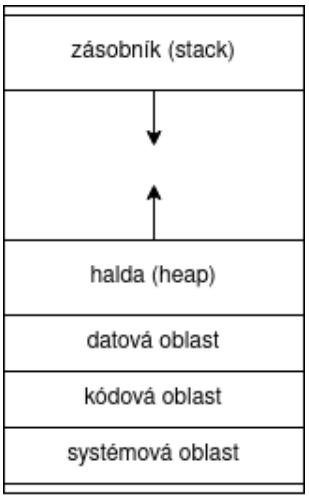
Obsah

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

V 32bitových systémech je každému procesu přidělen virtuální adresní prostor o velikosti 4 GiB (2^{32} b), v 64bitových systémech je to 2^{64} b. Do horních adres (`0xFFFFFFFF` a níže) se zapisuje *stack*, do spodních (`0x00000000` a výše) zbytek dat (včetně části prostoru která je vyhrazena jádru).

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

- Segment instrukcí (*Code*)
- Datový segment (*Data*)
- Halda/hromada (*Heap*)
- Zásobník (*Stack*)



Code (nebo také ***Text***) je množina strojových instrukcí, které program provádí.

V části ***Data*** jsou uložena data známá při překladu programu (hodnoty polí, konstant, proměnných, některé textové řetězce).

Bloky *Code* a *Data* jsou známy v době překladu a jejich velikost se v průběhu nemění.

Heap slouží k alokaci dynamické paměti; obsahuje objekty a instance proměnných (atributy třídy). Je založena na stromové datové struktuře.

Stack funguje na principu LIFO a je důležitý pro volaní správných funkcí. S uložením funkce na stack se posunuje ukazatel na pozici v registru (*stack pointer*). Po dokončení té funkce je odstraněna ze zásobníku a ukazatel změněn na předchozí funkci. Obsahuje metody/funkce, lokální proměnné a reference na proměnné.

Bloky *Heap* a *Stack* jsou dynamické a jejich velikost roste/zmenšuje se u každého jiným směrem v průběhu programu[1].

[1] Fungování haldy a zásobníku: https://courses.physics.illinois.edu/cs225/fa2020/resources/stack-heap/

1.1 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 ani velikost bloku nemůže měnit.

1.2 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. **Regenerace** paměti je její pročištění od nepoužívaných částí paměti.

1.2.1 Dynamické přidělování paměti bez regenerace

Dynamické přidělovaní paměti bez regenerace přiděluje požadované úseky postupně tak jak jsou za sebou umístěny až do vyčerpaní vyhrazené paměti. Využívá pracovního ukazatele, 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*; regenerace probíhá později po větších částech.

1.2.2 Dynamické přidělování paměti s regenerací

Na rozdíl od dynamického přidělovaní bez regenerace se zde regeneruje pro každé operaci *free/delete*. S tím přichází problém s fragmentací paměti. Po uvolnění paměti by tyto části mohly 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.

1.3 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, pokud nedojde k situaci, že nový úsek není dostačující pro fázi alokace: v tom případě 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].

[2] *Integer* zabírá 32 bitů, *objekt* (v C *struct*) obsahující dva integery bude reprezentován jako odkaz na část paměti kde začíná první integer. Nejspíše bude zabírat více než 64 b kvůli metadatům.

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 operací (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. Tyto strany jsou propojeny mezi sebou i samy se sebou pomocí těchto vztahů: 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 OON 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. OON se nezabývá konkrétní implementací, ale pouze vazbami mezi objekty.

OON přístup není vhodné využívat pokud na cílové platformě neexistuje překladač OOP jazyka, návrh je určen pro jazyk nepodporující OOP nebo by přepis stávajícího kódu byl neekonomický, hlavně u projektů s krátkou životností.

2.1 Vztahy mezi třídami

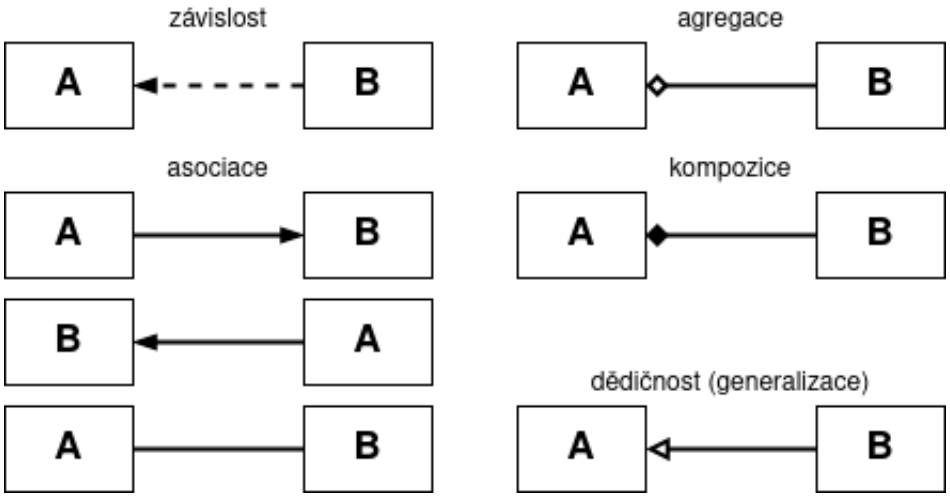
Závislost je dynamický a zároveň nejslabší vztah. Ukazuje jak co na sobě závisí.

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 následující objekty.

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 při popisu vztahu tříd. Šipka směřuje k rodiči („B dědí z A“). Jde o vztah na úrovni třídy a ne instance, s asociací nijak nesouvisí.



Obrázek 1: Vztahy instancí a tříd

Násobnost vztahů určuje počet vazeb objektu. „$1 : n$“ popisuje jeden objekt s n referencemi se kterými je ve vztahu (faktura : n položek).

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

Teorie vyčíslitelnosti zkoumá otázku algoritmické řešitelnosti problému z pohledu řešitelnosti, ne časové náročnosti. Vyčíslitelnost je zkoumána pomocí teoretických výpočetních modelů – deterministický a nedeterministický konečný automat, zásobníkový automat, Turingův stroj a další.

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. V potaz by se měly brát oba dva typy složitosti, jelikož může nastat situace, kdy lze problém řešit dvěma algoritmy: jedním s malou časovou složitostí a druhým s malou paměťovou. Výběr z těchto složitostí závisí na hardware a situaci použití.

3.1 Asymptotická složitost

Jelikož vždy nelze určit přesnou složitost algoritmu, byla definována asymptotická složitost, která chování funkce aproximuje ze tří pohledů:

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

Konstantní	$O(1)$	Nezávisí na velikosti vstupních dat
Logaritmická	$O(\log n)$	Náročnost s velikostí logaritmicky klesáě $10^9 \to 30$.
Lineární	$O(n)$	Počet operací je závislý na velikosti vstupních dat.
Kvazilineární	$O(n \log n)$	Poslední produkčně použitelná náročnost, např. quicksort.
Kvadratická	$O(n^2)$	500 prvků $\to$ 250 000 operací.
Kubická	$O(n^3)$	200 prvků $\to$ 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 $\log n$, kn, $3n^3 + 4n$, $2n \log n$, naopak sem nepatří 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 by nepolynomiální algoritmy byly v kryptografii nepoužitelné.

3.2 Posouzení složitosti algoritmů

Vybrané algoritmy a jejich složitost jsou popsány v dalších otázkách. Zde jsou jenom lehce shrnuty; složitost je udávána z pohledu časové složitosti.

3.2.1 Algoritmy řazení

Třídění pomocí algoritmů Bubble Sort, Insert Sort a Select Sort má složitost $\mathrm{O}(n^2)$. Algoritmus Quick Sort má složitost $\Theta(n \log n)/\mathrm{O}(n^2)$. Algoritmus Merge Sort má složitost $\mathrm{O}(n \log n)$.

3.3 Algoritmy hledání

Dva základní algoritmy pro hledání jsou *Binary search*[3] a *Linear search*[4]. Binární funguje lépe než lineární, ale pouze pro větší a seřazené seznamy.

Tabulka 1: Big-O složitosti vyhledávacích algoritmů

algoritmus	časová náročnost	paměťová náročnost
lineární vyhledávání	$\mathrm{O}(n)$	$\mathrm{O}(1)$
binární vyhledávání	$\mathrm{O}(\log n)$	$\mathrm{O}(1)$

3.4 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.

[3]Binární vyhledávání funguje na principu půlení seřazeného seznamu. Pokud je hledaná hodnota menší než střední hodnota, vyhledává se v první polovině, pokud je větší tak v druhé. Takto se iteruje dokud není číslo nalezeno. Více např. https://www.youtube.com/watch?v=KXJSjte_OAI.

[4]Lineární vyhledávání je naprosto jednoduchý lineární průchod seznamem, kde se postupně hledá požadovaná hodnota. Alternativa je *Linear Sentinel search*, která přidá hledanou hodnotu na konec seznamu. Více např. https://www.geeksforgeeks.org/linear-search/ nebo https://algorithmoftheweek.blog/2020/06/10/sentinel-linear-search/.

– 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.

3.5 Turingův stroj

– Turingův stroj – nelze měnit program.
– Univerzální Turingův stroj – program lze přepsat; dnešní PC, mobily, atd.
– Nedeterministický Turingův stroj – doposud nesestaven, sestrojení by znamenalo konec asymetrické kryptografie, neví se jestli ho lze vůbec sestrojit (P vs PN).
– Paralelní Turingův stroj - z pohledu vyčíslitelnosti je ekvivalentní s běžným, jen přináší více výkonu.
– Kvantový Turingův stroj – založen na superpozicích stavů, dokáže řešit „exponenciální explozi“ a převést ji na problém se složitostí v polynomiálním čase.

3.6 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.

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.

4.1 Lineární a nelineární struktury

Lineární jsou pole, seznamy, zásobník, fronta.

Tabulka 2: Složitost akcí v lineárních strukturách

struktura	přidání	vyhledávání	mazání	výběr dle indexu
pole	$O(n)$	$O(n)$	$O(n)$	$O(1)$
pole proměnlivé délky	$O(1)$	$O(n)$	$O(n)$	$O(1)$
seznam	$O(1)$	$O(n)$	$O(n)$	$O(n)$
zásobník	$O(1)$	–	$O(1)$	–
fronta	$O(1)$	–	$O(1)$	–

Nelineární jsou například stromy, které, pokud jsou nějak vyvažovány, mají náročnost $O(\log n)$. Pokud se nevyvažují, náročnost je $O(n)$. Tyto složitosti jsou pro všechny operace stejné.

4.2 Abstraktní datový typ

Abstraktní datový typ je množina druhů dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to nezávisle na konkrétní implementaci. 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.

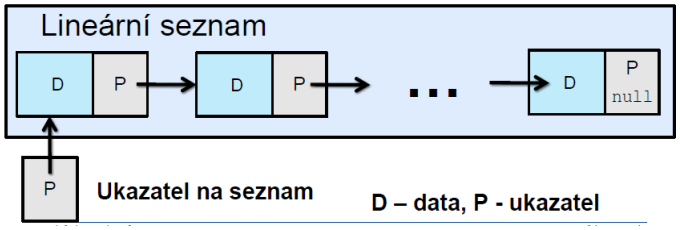
ADT dělíme podle počtu datových položek na statický a dynamický datový typ. Statický datový typ má neměnnou velikost a dynamický mění velikost dle provedené operace. Dále se dělí dle jednoznačného bezprostředního následníka na lineární a nelineární. Lineární mají následníka a u nelineárních neexistuje přímý jednoznačný následník.

4.3 Lineární seznam

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

Tabulka 3: Dělení abstraktních datových typů

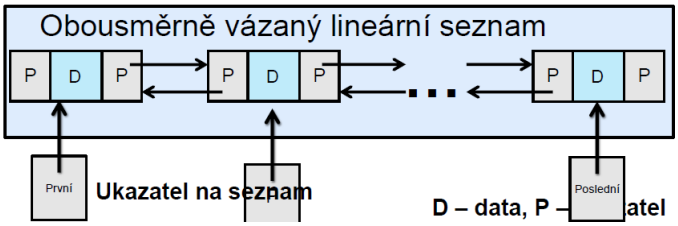
datový typ	bezprostřední následník	velikost
pole	lineární	statická
seznam	lineární	dynamická
zásobník	lineární	dynamická
fronta	lineární	dynamická
strom	nelineární	dynamická
množina	nelineární	dynamická



Obrázek 2: Lineární seznam

Cyklický lineární seznam Je 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.



Obrázek 3: Obousměrně vázaný lineární seznam

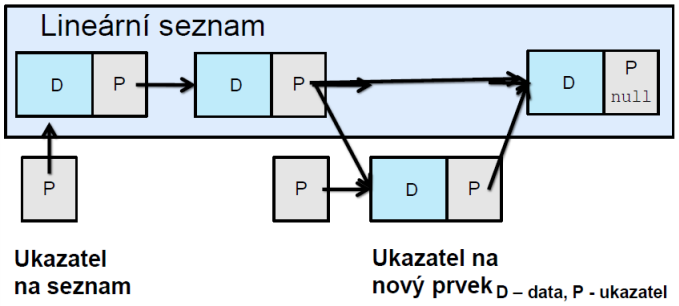
4.3.1 Operace s ADT seznamem

Nalezení délky, výpis všech prvků, získání n-tého prvku, vložení nového prvku za k-tý prvek, smazání prvku, nalezení předcházejícího a následujícího prvku.

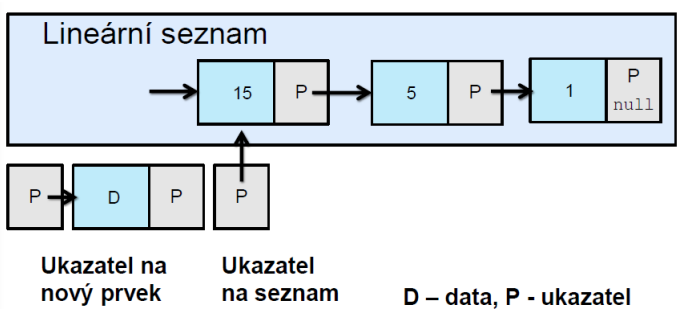
Pro **vložení** prvku je nejprve nalezena pozice mezi prvky existujícími: k. Ukazatel $k-1$ prvku je přepsán na k-tý a ukazatel nového (k-tého) prvku je přepsán na $k+1$, na který původně ukazoval $k-1$. Operace fungují stejně, jen jsou vázány na obě strany (viz obrázek 4).

Pro **smazání** prvku je ukazatel $k-1$ prvku přepsán na prvek $k+1$. Pokud je mazán první prvek, změní se ukazatel na seznam.

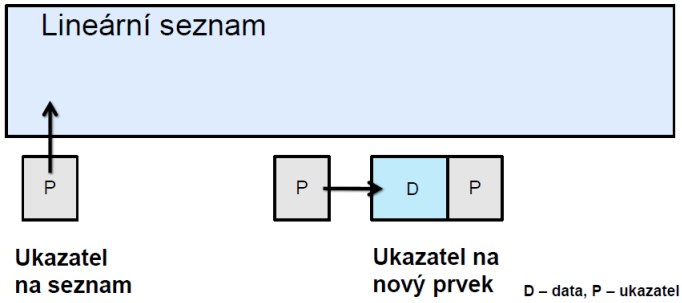
Vyhledávat lze dle indexu nebo podle dat. Seznam se postupně prochází dokud není nalezen prvek nebo konec seznamu.



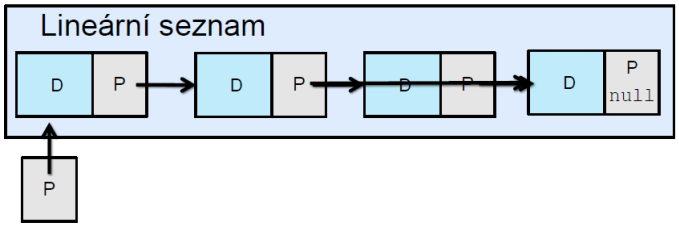
Obrázek 4: Vkládání do lineárního seznamu.



Obrázek 5: Vkládání do lineárního seznamu na první pozici. Ukazatel pole se přepisuje na první prvek a ve vkládaném prvku se přidá ukazatel na předchozí první prvek.



Obrázek 6: Vkládání do prázdného lineárního seznamu.



Obrázek 7: Mazání v lineárním seznamu

4.4 Zásobník

Dynamická datová struktura umožnující vkládaní a odebírání hodnot tak, že naposledy vložená hodnota se odebere jako první (LIFO). Základní operace jsou „Vložení na vrchol“, „Odebraní z vrcholu“ a „Test na prázdnost zásobníku“.

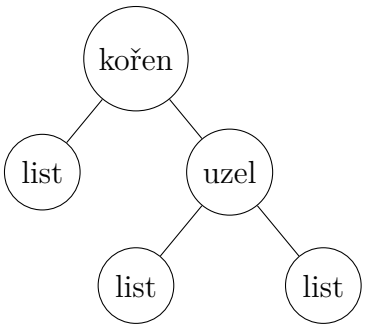
4.5 Fronta

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

ADT typu strom je nelineární dynamická abstraktní datová struktura. Každý uzel stromu může mít n potomků a zároveň má pouze jednoho předka. Nejčastější zástupci jsou obecný strom nebo n-ární strom. n-ární má maximální počet potomků rovný n, obecný strom není počtem potomků omezen. Skládá se z uzlů, které se pojmenovávají:

- kořen, který existuje pouze jeden (pouze ve vrcholu),
- listy (uzly bez následníků),
- vnitřní uzly (nejsou kořenem ani listem stromu).



Obrázek 8: Binární strom

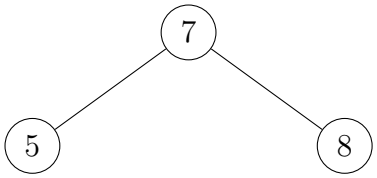
5.1 Binarní strom

Binární strom je strom, který má nanejvýše dva 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, nezaplní se pravý[5].

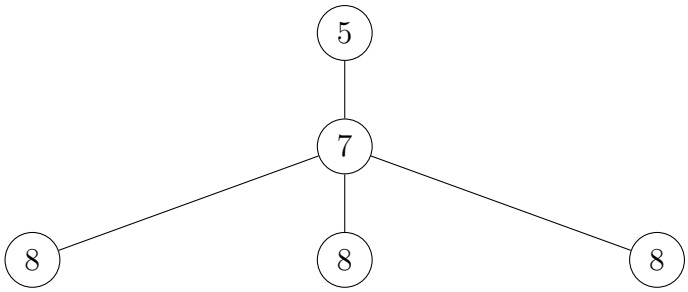
5.2 Binarní vyhledávací stromy

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 seznam (viz obrázek 10 na straně 14). U vyvážených binárních stromů by rozdíl hloubky levé a pravé části měl být 0 nebo 1.

[5]Ukázka úplného binárního stromu např. https://home.cs.colorado.edu/~main/supplements/pdf/notes10a.pdf.



Obrázek 9: Vyvážený strom



Obrázek 10: Nevyvážený strom

Vyhledávaní v binárních vyvážených stromech Postupuje se od kořene dolů. Při každém sestupu se porovná hledaná hodnota s hodnotou listu: pokud je shodná, byl prvek nalezen. Pokud není, algoritmus sestoupí do levého uzlu (pokud je hledaná hodnota menší než aktuání uzel) nebo do pravého uzlu (pokud je hledaná hodnota větší). Pokud je sestoupeno až k listu a hodnota neodpovídá, hledaný prvek ve stromu není.

Vložení do binárních stromů Postup je stejný jako při vyhledávání. Vkládá se pokud je nalezena hodnota *null* nebo položka se stejnou hodnotou (v tom případě ji implementace může a nemusí přepsat).

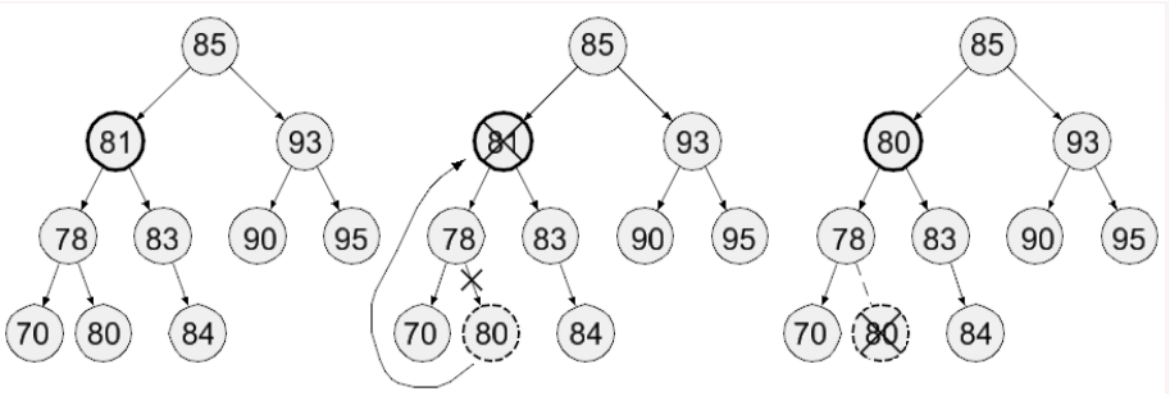
Odstranění prvků Pokud uzel nemá žádné potomky, pouze se odstraní a z jeho rodiče se smaže jeho reference. Pokud má jehoho potomka, reference v rodiči se změní na referenci tohoto potomka. Pokud má potomky dva, existují dvě možnosti: PL a LP.

Pro pravý prvek (PL) nalezneme uzel, který je nejvíc napravo v levém stromu a ten zaměníme za uzel, který chceme smazat: nejpravější uzel v levém stromu převezme referenci ze mazaného uzlu a rodič mazaného uzlu změní referenci na nejpravější uzel levého stromu. Pro levý prvek (LP) je to podobné, jen se vybírá nejlevější uzel z pravého stromu. Viz obrázky 11 a 12.

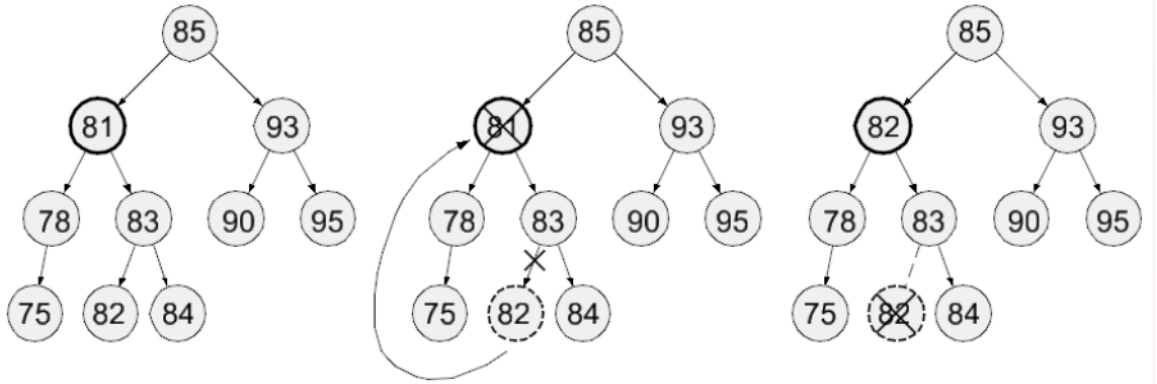
5.3 Procházení stromů

Strom a výsledky procházení jsou na obrázku 13.

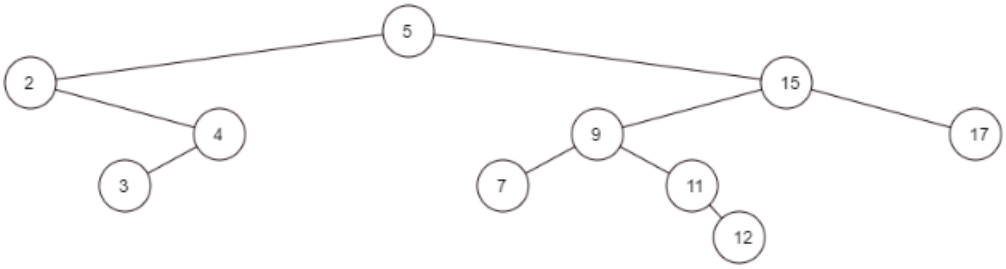
1. **Pre-order**. Nejprve se zpracuje kořen, poté levý podstrom a nakonec pravý podstrom. Využívá se k vytváření kopií stromů.
2. **In-order**. Nejprve se zpracuje levý podstrom, poté kořen a nakonec pravý podstrom. Výsledkem je seřazený seznam.
3. **Post-order**. Nejprve je zpracován levý podstrom, poté pravý a nakonec kořen. Využívá se k mazání stromu od listů ke kořenu.



Obrázek 11: Odstranění prvku s využitím PL.



Obrázek 12: Ostranění prvků s využitím LP.



Obrázek 13: Čteční stromu.
Pre-order: 5, 2, 4, 3, 15, 9, 7, 11, 12, 17.
In-order: 2, 3, 4, 5, 7, 9, 11, 12, 15, 17.
Post-order: 3, 4, 2, 7, 12, 11, 9, 17, 15, 5.

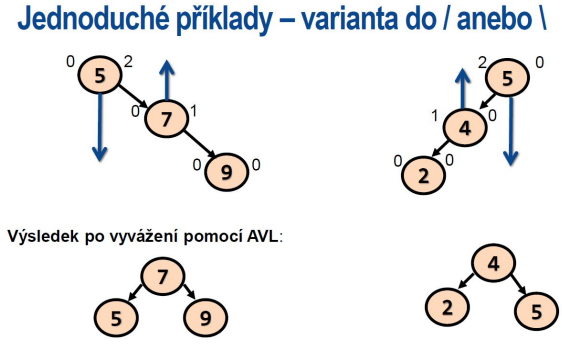
6 Problematika nevyvážených stromů. Vyvažování stromů AVL – rotace: jednoduchá levá, jednoduchá pravá, dvojitá levá, dvojitá pravá. Red-Black stromy. Posouzení z pohledu časové a paměťové složitosti. ADT hashovací tabulky. Řešení kolizí hashovacích tabulek. Srovnání výkonnosti binárních vyhledávacích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

Složitost u nevyváženého stromu může klesnout z $O(\log n)$ až na $O(n)$, a proto se stromy vyvažují. Vyvážený strom má rozdíl hloubky levého a pravého podstromu rovnou vždy 0 nebo 1. Pokud má hloubku větší, označuje se za nevyvážený a z pohledu problematiky by se měl vyvážit například metodou AVL nebo Red-Black stromů.

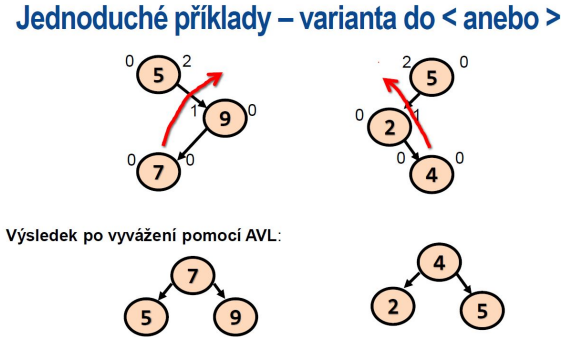
6.1 AVL stromy

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[6].



Obrázek 14: Rotace ↑ nebo ↓.



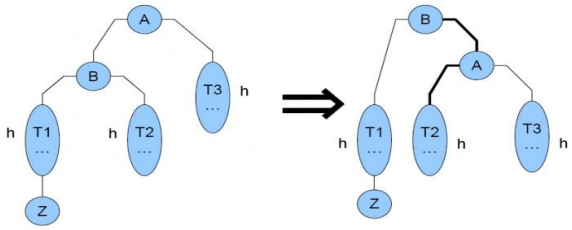
Obrázek 15: Rotace ↗ nebo ↖.

6.2 Red-Black stromy

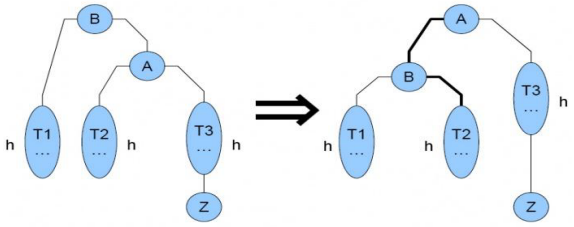
Nejsou tak dobře vyvážené jako AVL, ale řeší problém mnohonásobné rotace. Efektivní vkládání, ale méně efektivní vyhledávání[7]. Při vyvažování platí pravidla:

- Každý uzel je červený nebo černý.
- Kořen je vždy černý.

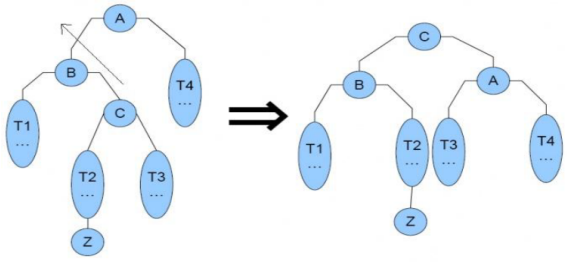
[6] Vizualizace https://www.cs.usfca.edu/~galles/visualization/AVLtree.html.
[7] Vizualizace https://www.cs.usfca.edu/~galles/visualization/RedBlack.html.



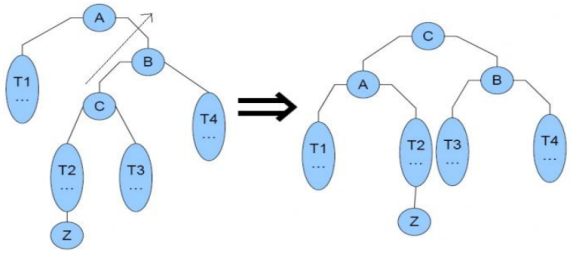
Obrázek 16: Jednoduchá levá rotace (podle prezentací)



Obrázek 17: Jednoduchá pravá (podle prezentací)



Obrázek 18: Dvojnásobná levá rotace (podle prezentací)



Obrázek 19: Dvojnásobná pravá rotace (podle prezentací)

– Potomci červeného jsou vždy černí.
– Každá cesta z libovolného uzlu do jeho podřízených listů obsahuje stejný počet černých uzlů.

6.3 Hashovací tabulka

Hashovací tabulka spojuje klíč s odpovídající hodnotou. Klíč je vypočítán z obsahu položky tak, aby byl co nejjednoznačnější a nedocházelo ke kolizím; vše vychází z pravděpodobnosti o hashovacích funkcích. Využívají se nejčastěji v databázích nebo k rychlému vyhledávaní v polích. Paměťová složitost je $O(n)$.[8]

Při **vkládání** se z vkládaného prvku udělá hash. Tento hash se přiřadí do pole, na místo, které mu odpovídá. Jestliže je místo obsazeno přiřadíme mu další volné místo dle algoritmu.

Vyhledávání Využijeme klíče, který nám vrátí index položky. Jestliže na odpovídajícím indexu se nenachází daná položka, pomocí algoritmu vypočteme, kde je další místo, kde se položka může nacházet.

6.3.1 Řešení kolizí

Řetězení tabulek: Prvek je vložen do lineárního seznamu.

[8] https://courses.physics.illinois.edu/cs225/fa2020/resources/hashtable/.

Tabulka 4: Časová složitost v dobře a špatně optimalizovaných tabulkách

operace	průměrná	nejhorší
vkládání	O(1)	O(n)
mazání	O(1)	O(n)
vyhledání	O(1)	O(n)

Otevřené adresování linear probing: Prvek je vložen na další vhodné místo.

Otevřené adresování double hashing: Prvek je zhashován znovu.

6.4 Hashovací tabulka vs binární strom

Tabulka 5: Porovnání hashovací tabulky a binárního stromu

ADT	Výhody	Nevýhody
binární strom	Rychlé vkládání, mazaní a hledání (pokud je vyvažován)	Algoritmus mazání je časově náročný.
AVL a Red-black stromy	Stejné jako binární stromy, vyvažovány jsou vždy.	Jsou složité.
hashovací tabulka	Rychlé vkládání, při existenci klíče rychlé vyhledávání.	Pomalý přístup pokud klíč není nalezen. Neefektivní využití paměti.

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

7 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.

7.1 Typy řazení

Řazení výběrem: najde se vždy nejmenší ze 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 způsobem, který vede k seřazení.

7.2 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 $O(n \log n)$.

7.3 Stabilita řazení

Dělíme na stabilní a nestabilní. Vstupní data můžou obsahovat několik prvků se stejnou hodnotou. Stabilní algoritmus při řazení zachovává vzájemné pořadí položek se stejnou hodnotou a u nestabilního jejich zachování není zaručeno. Stabilní je vhodné využívat při řazení dle dvou parametrů.

Když je řazen seznam osob dle křestního jména a poté příjmení, stabilní algoritmus vrátí výsledky v očekávaném pořadí (Adam Novák, Karel Novák, Václav Novák). Nestabilní může výsledky prvního řazení přeházet (Václav Novák, Adam Novák, Karel Novák).

7.4 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.[9]

7.5 Insertion sort

Je jednoduchý stabilní řadící algoritmus se složitostí $O(n^2)$. Množina prvků je rozdělena na seřazenou (na počátku prázdnou) a neseřazenou část. Prvky se jeden po druhém přesouvají z neseřazené části a jsou umísťovány do seřazené na správné místo.[10]

7.6 Selection sort

Je jednoduchý nestabilní řadící algoritmus se složitostí $O(n^2)$. Množina prvků je rozdělena na seřazenou (na počátku prázdnou) a seřazenou část. V neseřazené části je nalezen prvek s nejvyšší hodnotou a je umístěn do seřazené části. Poté je nalezen druhý největší prvek a je umístěn do seřazené.[11]

7.7 Shell sort

Je nestabilní kvadratický řadící algoritmus se složitostí $O(n^2)$. Funguje na principu insertion sort. Nejprve seřadí prvky, mezi kterými je určitá mezera (první a pátý, pátý a devátý, druhý a šestý) a v každém kroku je tato mezera zmenšena; když je snížena na 1, dojde k degradaci na bežný insertion sort.[12]

7.8 Merge sort

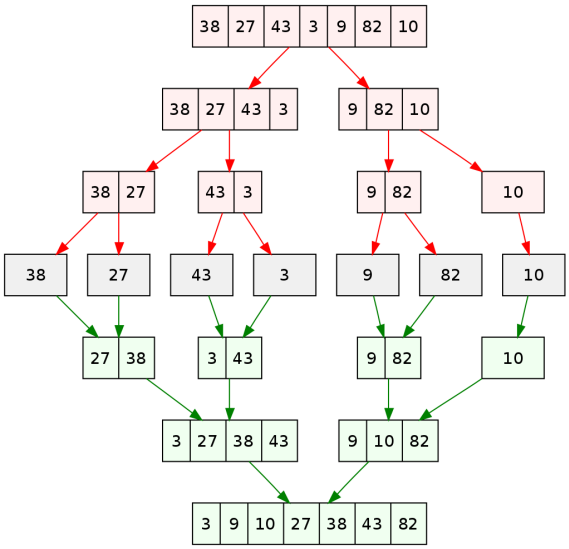
Je stabilní složitý řadící algoritmus se složitostí $O(n \log n)$.

Nesetříděné pole je děleno na poloviny, dokud nová pole nemají velikost 1. Poté se skládají dohromady: porovná se první prvek z pole A s prvním prvkem pole B a seřadí se, porovná se druhý prvek z pole A s druhým prvkem pole B, seřadí se a umístí za první prvky. Tento proces je opakován dokud nejsou spojeny všechny prvky původního pole.[13]

7.9 Heap sort

Je nestabilní složitý řadící algoritmus se složitostí $O(n \log n)$. Je jeden z nejefektivnějších řadících algoritmů.

[9] Animace Bubble sort: https://www.algoritmy.net/article/3/Bubble-sort.
[10] Animace Insertion sort: https://www.algoritmy.net/article/8/Insertion-sort.
[11] Animace Selection sort: https://www.algoritmy.net/article/4/Selection-sort.
[12] Animace Shell sort: https://www.algoritmy.net/article/154/Shell-sort.
[13] Video Merge sort: https://www.youtube.com/watch?v=qdv3i6X0PiQ.



Obrázek 20: Merge sort

Funguje na principu prioritní fronty (stromová struktura). Nejprve je prvky naplněn binární strom. Z tohoto stromu je vytvořena binární halda průchodem BFS. Z binárního stromu je kořen umístěn do množiny seřazených prvků; kořen stromu je umístěn do seřazené části, na jeho místo je umístěn nejnižší list a strom je znovu seřazen. Nový kořen stromu je znovu umístěn do seřazené části a takto se algoritmus opakuje dokud ve stromu zbývají uzly.[14]

7.10 Quick sort

Je nestabilní složitý řadící algoritmus se složitostí $O(n^2)$, resp. $\Theta(n \log n)$.

Je vybrán pivot (špatným výběrem lze složitost zhoršit; často bývá vybrán střed). Všechny prvky menší než pivot se přesunou nalevo a větší napravo. V obou polovinách je vybrán nový pivot v jejich středu a algoritmus se rekurzivně opakuje. Pivot vždy zůstává na místě a je považován za setříděný.[15]

[14] Video Heap sort: https://www.youtube.com/watch?v=LbB357_RwlY.
[15] Video Quick sort: https://www.youtube.com/watch?v=ZHVk2blR45Q.

8 Grafy, formální definice. Vyhledávání v grafech. Algoritmus BFS (prohledávání do šířky). Reprezentace BFS v paměti. Algoritmus DFS (prohledávání do hloubky).

8.1 Graf a 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 prvek, který chceme spojit s druhým vrcholem pomocí hrany tak, aby nám vznikl graf (ne vždy tyto prvky musí být spojené). Vrchol lze ohodnotit; stupeň grafu je označení pro počet připojených hran. Stupeň grafu je nejvyšší hodnota ze všech stupňů jeho vrcholů. Grafy lze **dělit dle hran** na:

- 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.

8.2 Vyhledávaní v grafu

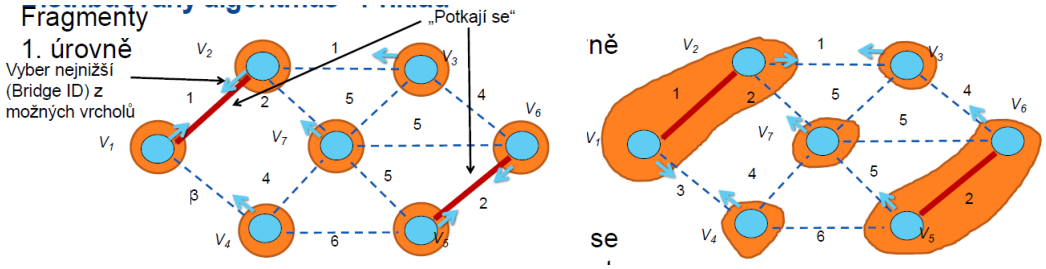
Druhy prohledávaní:

- Slepé prohledávání: prohledávají náhodně bez přemýšlení.
- Informované metody: snaží se odhadnout, kudy pokračovat ve vyhledávání aby byly 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.[16]

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.



Obrázek 21: Distribuovaný algoritmus

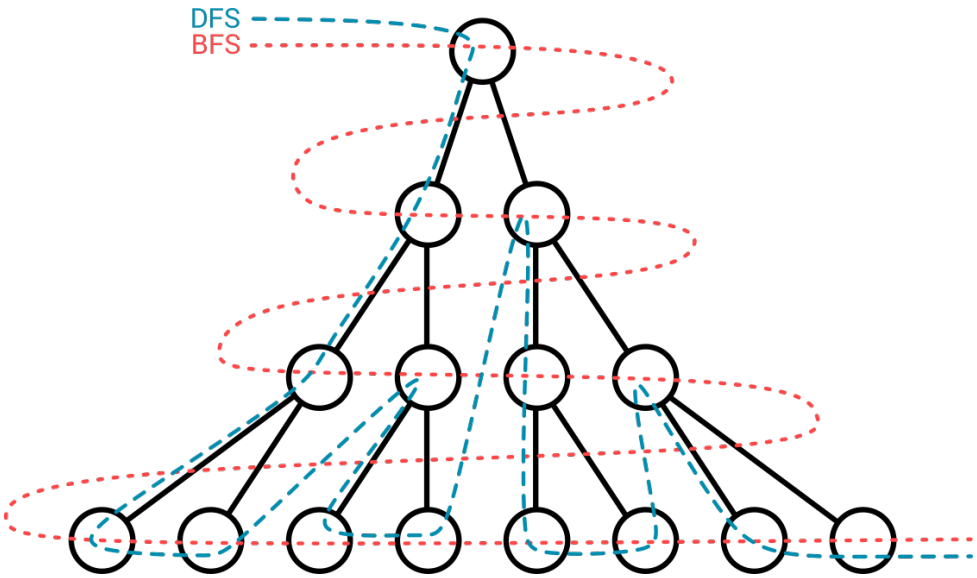
8.3 Slepé prohledávání

BFS využívá frontu, do které nejprve vloží vstupní vrchol. Z tohoto vrcholu se vezmou všichni sousedé a vloží se do fronty, ze které se bere postupně a jejich sousedé se přidávají do fronty. Při tomto se každý už navštívený vrchol ukládá do nějakého pole/seznamu navštívených, abychom ho nenavštěvovali znovu.[17]

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.[18]

DFS je méně paměťově náročné, BFS je optimální.

[16] Animace Kruskal: https://www.cs.usfca.edu/~galles/visualization/Kruskal.html.
[17] Video BFS: https://www.youtube.com/watch?v=oDqjPvD54Ss.
[18] Video DFS: https://www.youtube.com/watch?v=pcKY4hjDrxk?t=279.



Obrázek 22: Vyhledávání BFS a DFS

9 Omezené prohledávání do hloubky (DLS). Iterativní prohledávání do hloubky (IDLS), Dijkstrův algoritmus (Uniform Cost Search), A*

9.1 Pokračování slepého prohledávání

DLS vychází z DFS, ale je omezen na maximální hloubku.

IDLS je iterativní prohledávaní do hloubky. Vychází z DFS ale v každé iteraci se prohledává jen o hloubku níž.[19]

Dijkstrův algoritmus využívá prioritní frontu a seznam navštívených vrcholů. Z prvního vrcholu se přidají všechny sousední vrcholy s 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.[20]

9.2 A*

Informovaná metoda A* využívá prioritní frontu a seznam navštívených vrcholů. Každému z vrcholů přidělíme vzdálenost od cíle. Postupuje se stejně jako u Dijktrova algoritmu, akorát se sčítá vzdálenost s hodnotou heuristiky uzlu. Dle této sečtené hodnoty je vybrán další navštívený prvek z prioritní fronty.[21]

[19] Video IDLS: https://www.youtube.com/watch?v=Y85ECk_H3h4.
[20] Video Dijkstrův algoritmus: https://www.youtube.com/watch?v=GazC3A4OQTE.
[21] Video A*: 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.

10.1 Evoluční algoritmy

Je jeden ze způsobů optimalizace. Byly vytvořeny, aby sjednotily optimalizační metody, co využívají „evoluční“ principy. Evoluční algoritmy zastřešují řadu přístupů využívajících 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 začíná s základní populací. Z této populace se vytvoří výběr jedinců, které se zříží. Tito jedinci následně zmutují a vytvoří novou populaci. Tyto kroky se opakují v N iteracích. Ukončení je stanoveno předem: počtem iterací, dosažení požadovaného jedince, minimální změnou fitness populace po několik iterací.

10.2 Genetické algoritmy

Genetické algoritmy jsou založeny na Darwinově myšlence „přežije nejsilnější“. Využívá metody křížení, mutace a selekce. Kódování řetězci má také svou analogii v gentice, kde řetězce odpovídají chromozomům, jednotlivé pozice v řetězci jednotlivým genům a konkrétní hodnoty na těchto pozicích pak alelám.

Genetické algoritmy začínají s populací přípustných řešení, které jsou převedeny na řetězce/pole. Poté jsou vybráni jedinci, kteří se budou podílet na nové generaci: každý je ohodnocen fitness funkcí a na výsledku jsou vybráni nejvhodnější kandidáti (ale metod výběru je více). Tito jedinci se mezi sebou kříží a na konci cyklu mutují.

Výhody GA jsou, že nevyžadují žádné speciální znalosti o cílové funkci, jsou odolné proti klouzání do lokálního optima. Mají problém s nalezením přesného optima a vyžadují velké množství vyhodnocování cílové funkce.

Princip: Výchozí populace → výběr → křížení → mutování → nová populace → ukončit? Pokud ne, zpátky na výběr, atd.

10.3 Genetické programování

Hledá samotnou funkci a ne její parametry (hledá optimální program pro řešení úlohy, ne jen vlastnosti jedinců). 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.

10.4 Základní pojmy

Populace je množina jedinců o určité velikosti, ze které jsou vybírání pro operace (křížení, mutace atd.) při evoluci. Konkrétní populace se nazývá genotyp.

Jedinec člen populace, který je definován jedním chromozomem.

Chromozom je řetězec tvořený geny. Má za úkol odlišovat se od zbytku populace (DNA). Lze ho kódovat binárně, reálnými čísly, znaky, objekty, ...

Gen na i-té pozici reprezentuje stejnou charakteristiku v každém jedinci.

Alela je hodnota, kterou 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. Je více druhů metod vytvoření fitness funkce: hrubá, standardizovaná, přizpůsobená, normalizovaná.

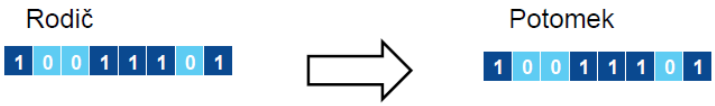
Selekce ruletový výběr, varianta 1: pravděpodobnost výběru závisí na kvalitě jedince (kolik místa na ruletě zabírá). Pokud jedinec ostatní převyšuje výraznou měrou, nová populace bude tvořena téměř výhradně jeho geny. K tomu se využívají techniky potlačení/podpory.

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.

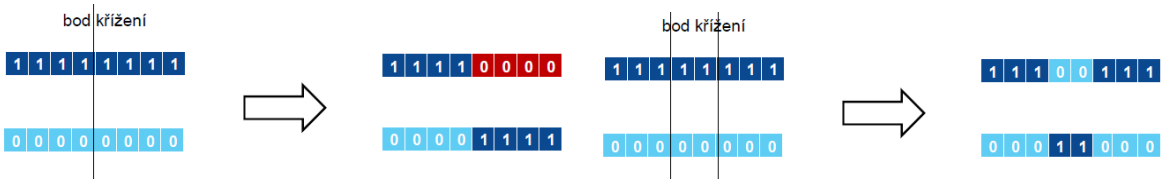
Selekce turnajový výběr: náhodně se vybere n jedinců a postupným porovnáváním je vybrán nejlepší.

10.5 Metody křížení

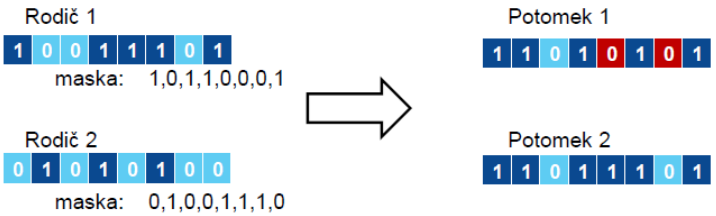
Napodobují genetickou evoluci a pomáhají ve vývoji. Čtyři základní metody jsou ukázány na straně 28 na obrázcích 23 až 26.



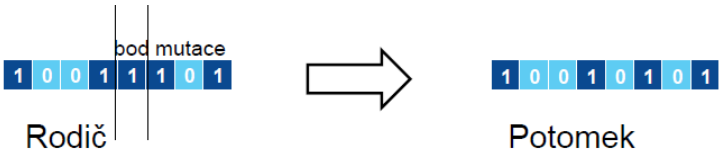
Obrázek 23: **Elitářství** zaručuje monotonní hodnotu fitness nejlepšího jedince a předchází ztrátě nejlepšího řešení.



Obrázek 24: **n-bodové křížení** dělí genotyp v n bodech.



Obrázek 25: **Uniformní křížení** rozvrací kód chromozomu a je dobré pro vnášení diverzity.

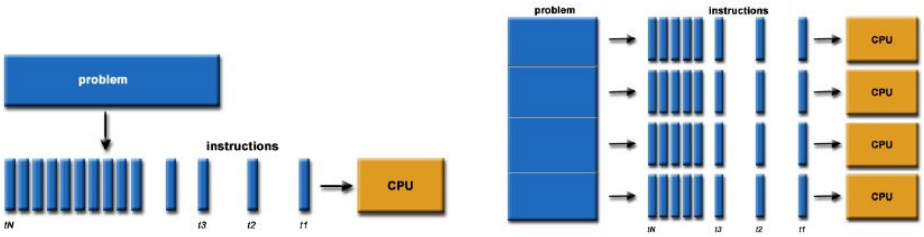


Obrázek 26: **Mutace** se aplikuje s malou pravděpodobností a je důležitá především pro malé počty jedinců.

11 Paralelní výpočty a architektury, procesy, vlákna a jejich synchronizace, Deadlock.

Sériové výpočty běží na jediném CPU s použitím jednoho vlánka, kde je problém rozdělen na posloupnost instrukcí. Každá instrukce se vykonává postupně, kdy v daném času může být spuštěna pouze jedna instrukce.

Paralelní výpočty na rozdíl od sériových běží na několika CPU. Zde jsou problémy rozděleny na části a až tyto části problému jsou rozděleny na posloupnost instrukcí. Instrukce každého problému jsou poté vykonávány souběžně na několika CPU.



Obrázek 27: Sériové a paralelní výpočty

Veškeré paralelní výpočty lze provádět sekvenčně, ale ne všechny sekvenční algoritmy lze paralelizovat. Příklady výpočetních zdrojů:

- Počítač s několika procesory
  - Multiprocesor se sdílenou pamětí (propojení více procesorů dohromady)
  - Multiprocesor obsahující několik procesorů (jader) na jednom čipu (obyčejné CPU).
- Libovolný počet počítačů propojený počítačovou sítí (Cluster computer)
- Spojení výše uvedeného do jednoho systému.

11.1 Sériové a paralelní modely (architektura)

Modely dělíme hlavně dle Flynnovy klasifikace z dvou hledisek: toku instrukcí a toku dat.

Tabulka 6: Modely dle Flynnovy klasifikace

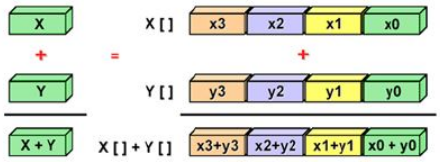
	Single Intruction	Multiple Instructions
Single Data	sériový	paralelní
Multiple Data	paralelní	paralelní

SISD Nejjednodušší model, který si lze představit jako (jednojádrový) procesor. Vykonává pouze jednu instrukci nad jedinými daty uloženými v paměti. Instrukce jsou posílány do řídící jednotky z paměti, následně jsou dekódovány a poslány procesní jednotky. Ta data zpracuje a pošle je zpátky.

Nejčastěji se vyskytuje v počítačích založených na von Neumann architektuře.

SIMD Systémy, u kterých existuje celá řada zpracovaných datových toků na základě jediného seznamu instrukcí. V každém okamžiku veškeré procesory vykonávají stejnou instrukci, kdy každá výpočetní jednotka může pracovat s libovolnými daty. Tento model je vhodný pro problémy, kde je charakteristická velká míra pravidelnosti (zpracovaní obrazu/videa, násobení matic). Existují dvě varianty v podobě procesorového pole a vektorové „potrubí/pipeliny“.

Příklad sčítání dvou polí vektorů. Pole jsou rozděleny na bloky a každý procesor pracuje s daným blokem. Položky jsou poté vybírány na základě ID procesu/vlákna.



MISD Jeden datový proud je napojen na několik procesorů. Každý procesor pracuje se stejnými daty, ale může s nimi vykonávat jiné operace. MISD jsou vhodné pro vícenásobné frekvenční filtry pracující na jednom signálovém proudu nebo kryptografické algoritmy pokoušející se prolomit jednu zakódovanou zprávu.

Lze je použít například ve vícenásobných vyhledávacích algoritmech, kde tyto algoritmy mohou pracovat se stejnými daty. Také mohou tyto algoritmy používat různou strategii a hledat různé vzory.

MIMD Tento model umožňuje zpracovávat různé posloupnosti instrukcí a pracovat s různými daty. Spouštění může být synchronizované, asynchronizované, deterministické nebo nedeterministické.

Jsou vhodné pro intenzivní paralelní výpočty. Tento model se využívá ve většině superpočítačů a výpočetních klusterech propojených přes síť.

11.2 Architektura paralelní počítačové paměti

Sdílená paměť Umožňuje všem procesorům přístup ke všem pamětím jako globálnímu adresovému prostoru. Procesory mohou pracovat nezávisle a sdílet stejné paměťové prostředky, změna paměti je viditelná všem procesorům.

Distribuovaná paměť Vyžadují komunikační síť pro propojení meziprocesorové paměti. Každý procesor má vlastní lokální paměť (tj. neexistuje sdílená paměť jako v předchozím případě). Procesory pracují nezávisle a je na programátorovi, aby zajistil synchronizaci dat.

Výhodou je škálovatelnost s počtem procesorů a rychlý přístup k paměti. Nevýhodou je zodpovědnost programátora za spousty detailů při synchronizaci procesů.

Hybridní distribuovaná sdílení paměť Využívá se obvykle SMP (Symetrické MultiProcessory: HW architektura kde více procesorů sdílí jeden paměťový prostor). Procesory na daném SMP mohou adresovat paměť tohoto procesoru jako globální. SMP znají pouze vlastní paměť a ne jiného SMP. Pro přesun dat se využívá sítová komunikace.

11.3 Paralelní programovací modely

Sdílená paměť Úkol sdílí společný adresní prostor, který čte a zapisuje asynchronně. Výhodou je neexistence vlastnictví, takže není výslovně potřebné specifikovat komunikaci dat mezi úkoly. Řídit přístup ke sdílené paměti lze pomocí zámků a semaforů.

Vláknový model Jeden proces může mít více souběžných cest provádění. Vlákno lze popsat jako podprogram v rámci hlavního programu, kde má každé vlákno lokální data a kde mezi sebou vlákna komunikují pomocí globální paměti. Mezi vlákny musí být zajištěna synchronizace aby nedošlo k přístupu ke stejné globální adrese více jak jedním vláknem.

11.4 Procesy a vlákna

Vlákna jsou méně náročná na systémové prostředky, sdílí paměť, snadnější programování, ale mají omezenou škálovatelnost. **Procesy** jsou náročnější na vytvoření, komunikace probíhá předáváním zpráv (nesdílí data), jsou náročnější na programování, ale dobře se škálují.

Synchronizace vláken Bez zajištění synchronizace může dojít k souběhu nebo uváznutí (deadlock). Cílem synchronizace je zabránit konfliktům mezi vlákny při přístupu ke sdíleným prostředkům.

Souběh K souběhu dochází v případě, kdy ke stejným datům přistupuje více vláken a jedno druhému přepíše data. Tím vzniká chyba za běhu programu.

Deadlock Nastává v případě, kdy na sebe dvě vlákna nebo dva procesy čekají na situaci, kdy se jim uvolní vzájemně uzamčené zdroje. Stává se při nesprávném použití synchronizačních mechanismů. Je nedeterministický a tedy obtížně testovatelný.