1. Problém

Termín problém označuje takové podmínky, nebo situaci nebo stav, který je nevyřešený, nebo nechtěný, nebo nežádoucí. Problém obvykle vyžaduje nějaké řešení. V takovém případě je nutné pochopit nejdůležitější aspekty daného stavu, neboť jen tak lze nalézt způsob řešení problému. Problém není to samé co úloha. Pro řešení úlohy jsou obvykle stanoveny i postupy a pravidla jak úlohu řešit.

Postup při řešení problému

1. identifikace problému - zjištění, že situaci nelze snadno zvládnout obvyklými způsoby
2. definice problému - vymezení problému o skupiny s podobným řešením
3. výběr strategie
    1. pokus a omyl
    2. použití starého řešení
    3. strategie vhledu (porozumění problému)
    4. vlastní nové řešení
4. využívání nových informací
5. kontrola efektivity řešení
6. zhodnocení dosaženého výsledku

2. Algoritmus

Algoritmus je přesný návod či postup, kterým lze vyřešit daný typ úlohy. Pojem algoritmu se nejčastěji objevuje při programování, kdy se jím myslí teoretický princip řešení problému (oproti přesnému zápisu v konkrétním programovacím jazyce). Obecně se ale algoritmus může objevit v jakémkoli jiném vědeckém odvětví. Jako jistý druh algoritmu se může chápat i např. kuchyňský recept. V užším smyslu se slovem algoritmus rozumí pouze takové postupy, které splňují některé silnější požadavky

Vlastnosti algoritmů

Konečnost (finitnost)

Každý algoritmus musí skončit v konečném počtu kroků. Tento počet kroků může být libovolně velký (podle rozsahu a hodnot vstupních údajů), ale pro každý jednotlivý vstup musí být konečný. Postupy, které tuto podmínku nesplňují, se mohou nazývat výpočetní metody. Speciálním příkladem nekonečné výpočetní metody je reaktivní proces, který průběžně reaguje s okolním prostředím. Někteří autoři však mezi algoritmy zahrnují i takovéto postupy.
Obecnost (hromadnost, masovost, univerzálnost)
Algoritmus neřeší jeden konkrétní problém (např. „jak spočítat 3×7“), ale obecnou třídu obdobných problémů (např. „jak spočítat součin dvou celých čísel“), má širokou množinu možných vstupů.

Determinovanost

Každý krok algoritmu musí být jednoznačně a přesně definován; v každé situaci musí být naprosto zřejmé, co a jak se má provést, jak má provádění algoritmu pokračovat, takže pro stejné vstupy dostaneme pokaždé stejné výsledky. Protože běžný jazyk obvykle neposkytuje naprostou přesnost a jednoznačnost vyjadřování, byly pro zápis algoritmů navrženy programovací jazyky, ve kterých má každý příkaz jasně definovaný význam. Vyjádření výpočetní metody v programovacím jazyce se

nazývá program. Některé algoritmy ale determinované nejsou, pravděpodobnostní algoritmy v sobě mají zahrnutu náhodu.

Výstup (resultativnost)

Algoritmus má alespoň jeden výstup, veličinu, která je v požadovaném vztahu k zadaným vstupům, a tím tvoří odpověď na problém, který algoritmus řeší (algoritmus vede od zpracování hodnot k výstupu)

Elementárnost

Algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků.
V praxi jsou proto předmětem zájmu hlavně takové algoritmy, které jsou v nějakém smyslu kvalitní. Takové algoritmy splňují různá kritéria, měřená např. počtem kroků potřebných pro běh algoritmu, jednoduchost, efektivitu či eleganci. Problematikou efektivity algoritmů, tzn. metodami, jak z několika známých algoritmů řešících konkrétní problém vybrat ten nejlepší, se zabývají odvětví informatiky nazývané algoritmická analýza a teorie složitosti.

Druhy algoritmů

Algoritmy můžeme klasifikovat různými způsoby. Mezi důležité druhy algoritmů patří:

- Rekurzivní algoritmy, které využívají (volají) samy sebe.
- Hladové algoritmy se k řešení propracovávají po jednotlivých rozhodnutích, která, jakmile jsou jednou učiněna, už nejsou dále revidována.
- Algoritmy typu rozděl a panuj dělí problém na menší podproblémy, na něž se rekurzivně aplikují (až po triviální podproblémy, které lze vyřešit přímo), po čemž se dílčí řešení vhodným způsobem sloučí.
- Algoritmy dynamického programování pracují tak, že postupně řeší části problému od nejjednodušších po složitější s tím, že využívají výsledky již vyřešených jednodušších podproblémů. Mnoho úloh se řeší převedením na grafovou úlohu a aplikací příslušného grafového algoritmu.
- Pravděpodobnostní algoritmy (někdy též probabilistické) provádějí některá rozhodnutí náhodně či pseudonáhodně.
- V případě, že máme k dispozici více počítačů, můžeme úlohu mezi ně rozdělit, což nám umožní ji vyřešit rychleji; tomuto cíli se věnují paralelní algoritmy.
- Genetické algoritmy pracují na základě napodobování biologických evolučních procesů, postupným „pěstováním“ nejlepších řešení pomocí mutací a křížení. V genetickém programování se tento postup aplikuje přímo na algoritmy (resp. programy), které jsou zde chápány jako možná řešení daného problému.
- Heuristický algoritmus si za cíl neklade nalézt přesné řešení, ale pouze nějaké vhodné přiblížení; používá se v situacích, kdy dostupné zdroje (např. čas) nepostačují na využití exaktních algoritmů (nebo pokud nejsou žádné vhodné exaktní algoritmy vůbec známy).

Přitom jeden algoritmus může patřit zároveň do více skupin; například může být zároveň rekurzivní a typu rozděl a panuj.

3.Počítačový program

Počítačový program je v informatice postup operací, který popisuje realizaci dané úlohy. Počítačový program může být vytvořen programátorem zápisem algoritmu v nějakém programovacím jazyce (dříve byl často realizován přímo v hardware – zapojením vodičů, děrným štítkem apod.). Zapsaný

program může být v počítači prováděn interpretem nebo může být pomocí překladače nejprve přeložen do strojového kódu a teprve pak přímo prováděn mikroprocesorem.

Charakteristika

Všechny počítačové programy označujeme souhrnně jako software (do software jsou někdy zahrnována i data). Programy můžeme je rozdělit na dvě základní skupiny:

* systémový software – zajišťuje chod počítače
* aplikační software – umožňuje využít počítač uživatelem k užitečné činnosti

Spuštěný program (tj. program umístěný operační paměti počítače a prováděný procesorem) se nazývá proces. Program může být spuštěn vícekrát a vytvořit tak více procesů. Proces se v operační paměti skládá z vlastního programu a dynamicky se měnících dat. Proces může být složen z více vláken.

V současných multitaskingových systémech je spuštěno zároveň více procesů, které jsou zdánlivě prováděny současně.

Programovací jazyky

První počítače byly programovány buď přímým zapojením obvodů nebo ve strojovém kódu procesoru. V současné době se programy obvykle zapisují v některém z programovacích jazyků ve formě zdrojového kódu, aby byl zápis srozumitelný pro člověka.

4. Objekt

Instance třídy (v některých programovacích jazycích také objekt) je konkrétní datový objekt v paměti odvozený z nějakého vzoru (třídy) používaný v objektově orientovaných programovacích jazycích (Java, C++, Simula 67, atd.). Objekt představuje základní stavební prvek objektově orientovaného programování.

Každý takový objekt má své vlastní atributy a metody podle vzoru (třídy).

Instance bývá obvykle vytvořena pomocí konstruktoru a klíčového slova new.

Příklad vytvoření instance třídy v jazyce Java:

JmenoTridy jmenoNoveInstance = new JmenoTridy();

Příklad vytvoření instance třídy v jazyce Delphi Object Pascal:

JmenoNoveInstance := JmenoTridy.Create(...);

5. Třída

Třída je základní konstrukční prvek objektově orientovaného programování sloužící jako továrna na objekty. Definuje jejich vlastnosti a metody. Vlastnosti mohou odlišovat jednotlivé objekty, např. u objektu člověk to mohou být jméno, věk, výška, pohlaví aj. Metody určují chování objektu, to čeho je schopný, např. udělej úkoly, zavři okno aj.

Abstraktní třída

Pomocí abstraktní třídy na rozdíl od klasické (neabstraktní) třídy nemůžeme vytvářet objekty. Abstraktní třída má implementované jen některé svoje metody, které se na všech objektech vykonají stejně. Neimplementované (abstraktní) metody se mohou lišit v různých podtřídách abstraktní třídy. Lze tedy říct, že se jedná o šablonu pro vytváření specifické skupiny tříd.

6. Spojové datové struktury

Lineární seznam (také lineární spojový seznam) je dynamická datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázany vzájemnými odkazy pomocí ukazatelů nebo referencí. Aby byl seznam lineární, nesmí existovat cykly ve vzájemných odkazech.

Lineární seznamy mohou existovat jednosměrné a obousměrné. V jednosměrném seznamu odkazuje každá položka na položku následující a v obousměrném seznamu odkazuje položka na následující i předcházející položky. Zavádí se také ukazatel nebo reference na aktuální (vybraný) prvek seznamu. Na konci (a začátku) seznamu musí být definována zarážka označující konec seznamu. Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o kruhový seznam.

7. Správnost programu

??? testování ???

8. Analýza algoritmů

Vývoj počítačů vytvořil potřebu pro rozvoj teorie, která věnuje pozornost problematice:
- analýzy algoritmů (programů)
- složitosti algoritmů
- hledání optimálních algoritmů

Analýza algoritmu = určení požadovaných prostředků na jeho vykonání
Prostředky mohou být paměť, šířka komunikačního pásma, …
Nejčastěji nás zajímá čas výpočtu

Čas výpočtu algoritmů

Budeme uvažovat výpočet algoritmů na „standardních počítačích“ s jedním procesorem a obvyklou množinou instrukcí a zobrazením čísel.
Čas výpočtu obecně roste s velikostí vstupu.

Velikost vstupu je obvykle počet zpracovávaných položek, např. počet prvků, které máme seřadit.
Čas výpočtu algoritmu pro danou velikost vstupu potom bude dán počtem vykonaných elementárních kroků
Označme c operace počet elementárních
kroků potřebných pro vykonání určité operace
Je-li vykonána n-násobně, přispěje k času vykonání nc operace

Splnitelnost výrokových formulí

if (booleovský výraz)
příkaz;

Nechť booleovský výraz pozůstává z

1. n proměnných logického typu v1, v2, …, vn
2. m logických operátorů not, and, …
3. závorek

Jestliže přiřazení hodnoty proměnné trvá čas c h a
čas každé logické operace čas c o , potom
vyhodnocení podmínky (výrokové formuly) trvá
T(n,m) = n c h + m c o
Výroková formule (podmínka) je splnitelná existuje-li přirazení hodnot true a false proměnným v1, v2, …, vn takové, že hodnota formule (logického výrazu) je true.
Problém – zjistit zda-li je zadaná formule splnitelná
Algoritmus – pro každé možné přiřazení hodnot proměnným v1, v2, …, vn formuli vyhodnoť
Počet různých přiřazení je 2n a čas výpočtu algoritmu je $T(n,m) = 2n(n\,c\,h + m\,c\,o)$
Časovou složitostí rozumíme růst času výpočtu
v nejhorším nebo průměrném případě

Asymptotická složitost

Asymptotická složitost vyjadřuje limitní růst
času výpočtu, když velikost problému
neomezeně roste.

Funkce g(n) je $\Theta(f(n))$ existují-li kladné konstanty c1, c2 a n0 takové, že $0 \le c1f(n) \le g(n) \le c2f(n)$ pro všechna $n \ge n0$
Růst času výpočtu uvedeného algoritmu násobení matic jsme intuitivně navrhli charakterizovat členem n3
Čas výpočtu uvedeného algoritmu je $\Theta(n3)$

$c1\ n3 \le an3 + bn2 \le c2\ n3 \qquad a,b > 0$

$c1 \le a + b/n \le c2$

c1 = a, c2 = a + b/ n0 , n0 libovolné kladné
Polynom adnd + … + a0, ad >0 je $\Theta(nd)$
Θ notace omezuje funkci shora i zdola

Asymptotické omezení shora

Funkce g(n) je O(f(n)), existují-li kladné konstanty c a n0 takové, že $0 \le g(n) \le cf(n)$ pro všechna $n \ge n0$
an3 + bn2 je tedy O(n3), ale i O(n4), …
Pro algoritmus řazení vkládáním:
-nejhorší případ $\Theta(n2)$ i O(n2)

-nejlepší případ Θ(n)
-pro každý vstup O(n2)

Asymptotické omezení zdola

Funkce g(n) je Ω(f(n)), existují-li kladné konstanty c a n0 takové, že
$0 \leq cf(n) \leq g(n)$ pro všechna $n \geq n0$
Pro algoritmus řazení vkládáním:
-nejlepší případ Ω(n)
-nejhorší případ Ω(n2)
-pro každý vstup Ω(n)

Polynomiální algoritmy

1. Čas, který potřebují ke zpracování vstupních dat algoritmy považované za rychlé, bývá zpravidla shora omezen funkcemi typu n, n.log (n), n2, …
2. Jako horní hranici lze vždy použít polynom
3. Proto se těmto algoritmům říká polynomiální nebo také prakticky použitelné

Exponenciální algoritmy

1. Pomalé (prakticky nepoužitelné) algoritmy užívají metodu „hrubé síly“ (brute force), kterou probírají všechny možnosti
2. Je-li např. dána n prvková množina a probírají-li se všechny její:
   1. podmnožiny: je potřeba alespoň 2n kroků
   2. permutace: je potřeba alespoň n! kroků
   3. zobrazení do sebe: je potřeba alespoň nn kroků
3. Odhad počtu kroků je alespoň exponenciální

9. Rekurze

K získání výsledku operace potrebujeme znát výsledek téže operace (vetšinou s jiným vstupem), pro jistý vstup. Obvykle metoda volající sama sebe. Všechna rekurzivní volání metod jsou aktivní naráz a každá má své kopiepromenných) pametove nárocné. Pomocí rekurze lze definovat množiny (spojové seznamy, stromy), fraktální útvary (Hilbertova krivka, Kochova vlocka nebo algoritmy (faktoriál, Hanojské veže, pruchod stromem). Lze eliminovat použitím zásobníku, nekdy také prevedením na iteracní algoritmus.

10. Abstraktní datový typ

Abstraktní datový typ (ADT) je v informatice výraz pro typy dat, které jsou nezávislé na vlastní implementaci. Hlavním cílem je zjednodušit a zpřehlednit program, který provádí operace s daným datovým typem. ADT umožnuje vytvářet i složitější datové typy, např. operace s ADT typu zásobník, fronta a pole. Všechny ADT lze realizovat pomocí základních algoritmických operací (přiřazení, sčítání, násobení, podmíněný skok,…).

Základní ADT

asociativní pole, zásobník, seznam, fronta, množina, zobrazení, textový řetězec, strom, halda

Oddělení rozhraní a implementace

Při programování je ADT reprezentováno rozhraním, které skrývá vlastní implementaci. Klientského programátora, který ADT používá, zajímá jak objekt používat (jeho rozhraní), ale ne už vlastní implementace.

Robustnost ADT spočívá v tom, že implementace je skrytá a programátorovi jsou nabídnuty pouze ovládací prvky. Vlastní implementaci ale lze změnit.

Je rozdíl mezi abstraktním datovým typem a datovou strukturou použitou pro jeho implementaci. Například seznam jako ADT může být implementován jako pole, nebo jako spojový seznam. Seznam je ADT s definovanými operacemi (jako vložit_prvek, smazat_prvek atd.), ale spojový seznam je datová struktura, která používá ukazatele a může být použita k vytvoření seznamu jako ADT.

Vlastnosti abstraktního datového typu

Nejdůležitější vlastnosti abstraktního typu dat

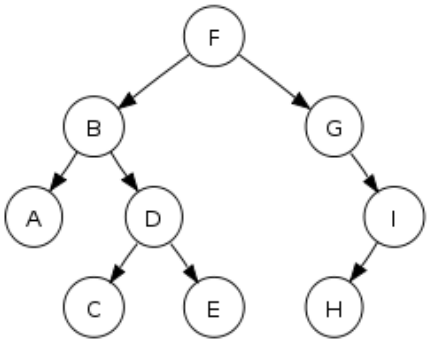
Všeobecnost implementace: jednou navržený ADT může být zabudován a bez problémů používán v jakémkolikv programu.
Přesný popis: propojení mezi implementací a rozhraním musí být jednoznačné a úplné.
Jednoduchost: při používání se uživatel nemusí starat o vnitřní realizaci a správu ADT v paměti.
Zapouzdření: rozhraní by mělo být pojato jako uzavřená část. Uživatel by měl vědět přesně co ADT dělá, ale ne jak to dělá.
Integrita: uživatel nemůže zasahovat do vnitřní struktury dat. Tím se výrazně sníží riziko nechtěného smazání nebo změna již uložených dat.
Modularita: „stavebnicový“ princip programování je přehledný a umožňuje snadnou výměnu části kódu. Při hledání chyb mohou být jednotlivé moduly považovány za kompaktní celky. Při zlepšování ADT není nutné zasahovat do celého programu.

Pokud je ADT programován objektově, jsou většinou tyto vlastnosti splněny.

11. Zásobník, fronta, seznam

Zásobník
Operace výberu vyjme prvek vložený naposledy, LIFO (last in, first out). Vložení = push, výber = pop. Implementace statickým polem a spojovým seznamem O(1), dynamickým polem složitejší.
Fronta
Výber vyjme prvek vložený nejdríve, FIFO (first in, first out). Implementace -”-.
Seznam
Vkládat a vybírat lze na libovolné pozici. Pozice je bud parametrem metody, nebo uložena okamžitá pozice v seznamu. Implementace pomocí pole O(n), spojovým seznamem s uloženou pozicí O(1) a s predávanou pozicí O(n). Oddelení dat seznamu a operací nad nimi – iterátor. Možnost více iterátoru nad jedním seznamem (každý uchovává vlastní pozici v seznamu). Kruhový a obousmerný seznam. Implementace seznamu polem (uložen index další položky místo ukazatele na ni).

12. Strom, průchody stromem, binární vyhledávací stromy



Výsledky způsobů procházení v binárním vyhledávacím stromu,

N = navštívený uzel, **L** = levý, **R** = pravý

- ***Preorder*** (NLR): F, B, A, D, C, E, G, I, H
- ***Inorder*** (LNR): A, B, C, D, E, F, G, H, I
- ***Postorder*** (LRN): A, C, E, D, B, H, I, G, F
- Procházení do šířky (*po vrstvách*) ***Level-order***: F, B, G, A, D, I, C, E, H

Binární vyhledávací strom (BST) je datová struktura založená na binárním stromu, v němž jsou jednotlivé prvky (uzly) uspořádány tak, aby v tomto stromu bylo možné rychle vyhledávat danou hodnotu. To zajišťují tyto vlastnosti:

* Jedná se o binární strom, každý uzel tedy má nanejvýš dva syny − levého a pravého.
* Každému uzlu je přiřazen určitý klíč. Podle hodnot těchto klíčů jsou uzly uspořádány.
* Levý podstrom uzlu obsahuje pouze klíče menší než je klíč tohoto uzlu.
* Pravý podstrom uzlu obsahuje pouze klíče větší než je klíč tohoto uzlu.

Hlavní výhodou binárních vyhledávacích stromů je vysoká efektivita vyhledávání hodnot v nich. Lze je využít také k efektivnímu řazení hodnot − in-order průchod binárním vyhledávacím stromem vydá seznam uložených hodnot uspořádaný podle velikosti.

13. Grafy a jejich implementace

Reprezentace grafu v informatice

- seznamy sousednosti

- matice sousednosti

- pro orientované i neorientované grafy

Seznamy sousednosti

- pro každý vrchol je vytvořen seznam sousedů
- sousedi uloženy libovolném pořadí

Matice sousednosti

Označme vrcholy čísly 1, …, |V|.
Matice sousednosti je matice S = (sij), i,j = 1, …, |V|, přičemž
je-li $(i,j) \in H$, sij = 1

jinak sij = 0

14. Prohledávání grafu

Do šířky

Prohledávání do šířky (anglicky Breadth-first search, zkráceně BFS) je grafový algoritmus, který postupně prochází všechny vrcholy v dané komponentě souvislosti. Algoritmus nejprve projde všechny sousedy startovního vrcholu, poté sousedy sousedů atd. až projde celou komponentu souvislosti.

Jak funguje

Prohledávání do šířky je algoritmus či metoda, která postupuje systematickým prohledáváním grafu přes všechny uzly. Nepoužívá při svém prohledávání žádnou heuristickou analýzu. Pouze prochází všechny uzly a pro každý projde jejich všechny následovníky. Přitom si poznamenává předchůdce jednotlivých uzlů a tím je poté vytvořen strom nejkratších cest k jednotlivým uzlům z kořene (uzlu v kterém jsme začínali).

Z hlediska algoritmu, veškeré následovníky uzlu získané expandujícím uzlem jsou vkládané do FIFO fronty. FIFO fronta znamená, že první uzel, který do fronty vstoupil jí také první opustí. V typických implementacích, jsou uzly, které nebyly ještě objeveny označeny jako FRESH. Uzly které se dostávají do fronty, které jsou v této chvíli právě vyšetřovány na jejich následníky jsou označeny jako OPEN a naposled uzly, které z fronty byly už vybrány a už s nimi nebudeme pracovat, jsou označeny jako CLOSE. Close uzly již nikdy v tomto běhu algoritmu nebudou prozkoumávany, mají vyplněné všechny informace. To znamená vzdálenost od kořenového (počátečního) uzlu, stav uzlu (CLOSE) a předchůdce.

Do hloubky

Prohledávání do hloubky (v angličtině označované jako depth-first search nebo zkratkou DFS) je grafový algoritmus pro procházení grafů metodou backtrackingu. Pracuje tak, že vždy expanduje prvního následníka každého vrcholu, pokud jej ještě nenavštívil. Pokud narazí na vrchol, z nějž už nelze dále pokračovat (nemá žádné následníky nebo byli všichni navštíveni), vrací se zpět backtrackingem.

15. Topologické řazení

Pro množinu prvku máme definované dvojice, u kterých je urceno poradí, v jakém se mají vyskytovat. Na základe toho vytvoríme acyklický orientovaný graf, který prohledáváme do hloubky. Pri dokoncení (obarvení na cerno) každého vrcholu ho pridáme na zacátek seznamu. Výsledný seznam je obrácenou posloupností dokoncování vrcholu a udává poradí, v jakém se mají prvky vyskytovat. Použití napríklad u složitých výrobních procesu, kde se nekteré prvky musí montovat postupne a jiné nezávisle.

16. Tabulka s přímým adresováním

k → Ak je prosté zobrazení, každá položka tabulky má své místo jednoznačně určené hodnotou Ak přímo odvozenou z k

Optimální implementace tabulky je pomocí pole – indexy jsou přímo klíče v tabulce

S=T=A=1

Výhody:

rychlý přístup

jednoduchá implementace

Nevýhody:

Velikost tabulky je daná rozsahem klíče, pro praktické účely bývá většinou neúnosná

Řídké pole – nerovnoměrný počet klíčů vzhledem k rozsahu tabulky.

Příklady:

Telefonní síť – klíčem je telefonní číslo uživatele

Telefonní seznam – klíčem je jméno

17. Rozptylové tabulky s vnějším řetězením

Pri velkém množství možných hodnot klíce oproti množství ukládaných prvku (typicky retezec klícem) není vhodné ci možné použít tabulku s prímým adresováním. Použijeme tedy rozptylovou funkci, která množinu klícu transformuje na množinu celých císel z menšího rozsahu, aby bylo využití pole dostatecne efektivní. Tato císla jsou pak indexy v poli. Více klícu se tím ale zobrazí na stejný index. Pokud pak potrebujeme uložit dva prvky se stejným indexem (tedy dojde ke kolizi), mužeme použít vnitrní nebo vnejší retezení. Pri vnitrním retezení se kolidující prvek ukládá na jiné volné místo v tomtéž poli a ke stávajícímu je uložen index tohoto místa. Docílíme tím vetší pametové efektivity, ale komplikuje se ukládání dalších prvku, jejichž místo jsme takto obsadili. Druhým zpusobem je vnejší retezení, kdy je vyhrazeno další místo pro kolidující prvky a opet jsou ukládány odkazy. Prípadne lze k položkám tabulky pripojit spojové seznamy a do nich ukládat všechny položky pro daný index. Ve všech prípadech se zvyšuje složitost, protože prohledávání seznamu má složitost O(n). Duležitá je pritom predevším rozptylová funkce, která by mela ideálne rozdelovat klíce na indexy rovnomerne (obvykle delení modulo nebo prepocet na menší interval a zaokrouhlení). Dále je duležitý pomer velikosti tabulky a poctu vkládaných prvku, nejlepší volbou je velikost rovná poctu prvku, kdy ani pri pridání dalších prvku se príliš nezhorší doba vyhledávání.

18. Prioritní fronta

Prioritní fronta je dynamická množina umožnující efektivní realizaci operací vkládání libovolných nových prvku a jejich vybírání v poradí jejich velikosti pomocí následujících operací:

Get Max(S),

Extract Max(S),

Insert(S, k).

Jedna z nejužitecnejších aplikací haldy. Napr. I rozvrhování úloh podle relativních priorit v

multitaskingovém systému:

- pri dokoncení bežící úlohy se vybere nová s nejvyšší prioritou,
- nové úlohy jsou prubežne zarazovány.

I událostmi rízená simulace:

- prvky haldy jsou události s casovou znackou, které se mají simulovat,
- simulace se musí provádet v poradí casových znacek,
- nové události se zarazují podle casových znacek,
- místo operace Max potrebujeme Min.

19. Halda

Halda je v informatice stromová datová struktura splňující tzv. vlastnost haldy: pokud je B potomek A, pak x(A) >= x(B). To znamená, že v kořenu stromu je vždy prvek s nejvyšším klíčem (klíč udává funkce x). Taková halda se pak někdy označuje jako max heap (heap je v angličtině halda), halda s reverzním pořadím prvků se analogicky nazývá min heap. Díky této vlastnosti se haldy často používají na implementaci prioritní fronty. Efektivita operací s haldou je klíčová pro mnoho algoritmů.

Operace s haldou

- INSERT - přidání nového prvku do haldy
- DELETE MAX nebo DELETE MIN - vyjmutí kořenu v max heap nebo v min heap
- DELETE(v) - smaže uzel „v“
- MIN, MAX - vrátí minimální resp. maximální klíč v haldě
- DECREASE KEY(v, okolik) - zmenšení klíče uzlu „v“ o hodnotu „okolik“
- INCRESE KEY(v, okolik) - zvětšení klíče uzlu „v“ o hodnotu „okolik“
- MERGE - spojení dvou hald do jedné nové validní haldy obsahující všechny prvky obou původních
- MAKE - dostane pole N prvku a vytvoří z nich haldu

Popis haldy

Haldu bychom mohli tedy definovat jako datovou struktura splňující dvě základní podmínky:

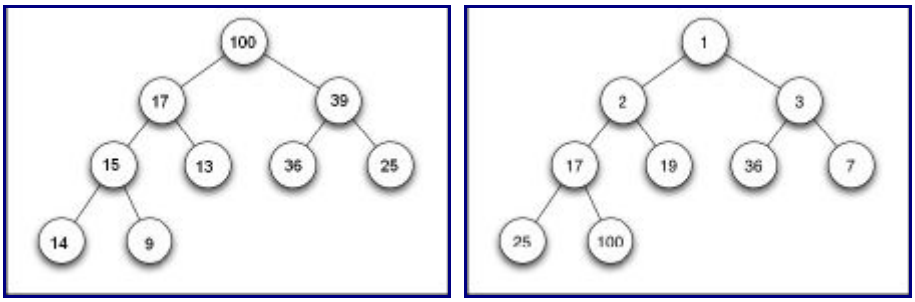
- lokální podmínku na uspořádání - prvek reprezentující otce je menší než prvek reprezentovaný synem apod.
- strukturální podmínku na stromy, ze kterých jsou vytvořené

Právě podle těchto podmínek se haldy rozdělují na d-regulární, Fibonacciho, Leftist a další (mohou se lišit jak lokální, tak strukturální podmínkou).

Jak již bylo naznačeno, rozlišujeme haldy Min-Heap a Max-Heap.

- U Min-Heap jsou klíče dětí jednoho uzlu vždy větší než klíče svého otce. To způsobuje, že na kořenech stromu lze nalézt pouze prvek s minimálním klíčem.
- Opačně je pro Max-Heap stanovena podmínka, že klíče dětí jednoho uzlu musí být vždy menší než klíče jejich otce. Zde se na kořeni stromu vždy nachází prvek s maximálním klíčem.

Příklady Max-Heap a Min-Heap:



Matematicky se u obou variant jedná pouze o rozdíl v jejich opačném pořadí prvků. Protože je definice vzestupně i sestupně libovolná, je pouze otázkou interpretace, zda se u konkrétní implementaci jedná o Min-Heap nebo o Max-Heap.

20. Algoritmy řazení O(n log2 n)

Řazení haldou (heapsort)

Na začátku je neseřazené pole. Postupně obnovujeme vlastnost haldy pro první dva, tři, čtyři atd. prvky, až máme na poli vytvořenu haldu. Následně zaměníme nejvyšší prvek (vyjmeme ho) s posledním a obnovíme vlastnost haldy na n−1 prvcích. To opakujeme až zbude jednoprvková halda a celé pole je seřazené. Nestabilní.

Shellovo řazení (shellsort)

Posloupnost rozdělíme na podposloupnosti s krokem h (každý h-tý počínaje prvním, počínaje druhým až počínaje h − 1-ním), které nezávisle seřadíme vkládáním (insertsortem). Následně snižujeme hodnotu h a provádíme totéž až do seřazení s h = 1. Prvky daleko od své výsledné pozice se k ní dostanou v menším počtu kroků, než u obyčejného insertsortu. Shell – počáteční h = polovina délky pole, snížení vydělením dvěma; Knuth – posloupnost 3i + 1, počáteční h = její prvek nejbližší třetině délky pole. Jednoduchý a efektivní algoritmus, složitá analýza, nestabilní.

Řazení dělením (splitsort, quicksort)

Určíme prvek, jehož hodnota bude rozdělovat pole (pivot). Procházíme z obou stran pole a menší hodnoty vpravo prohazujeme s většími vlevo, až se oba průchody potkají. Na tuto pozici přesuneme pivot – vlevo jsou hodnoty menší, vpravo větší. Totéž provedeme s částí pole vlevo od pivotu i vpravo od něj. Rekurze končí, je-li předán pouze jeden prvek k seřazení. Nejčastěji se pivot určuje jako krajní prvek pole (nezasahuje pak do procesu dělení). Lze použít zásobník místo rekurze. Snadno implementovatelné, nestabilní.

Řazení slučováním (mergesort)

Pole je rekurzivně rozdělováno na poloviny. Dojdeme až na úroveň dvojic prvků, které seřadíme (sloučíme).

21. Dolní omezení pro porovnávací řazení

Porovnávací řazení lze znázornit úplným rozhodovacím stromem, kde ve vnitřních vrcholech dochází vždy k porovnání dvou prvků a jejich případnému prohození (např. levý potomek =

původní posloupnost, pravý = posloupnost s prohozenou dvojicí). Listy tvoří různé permutace prvků, úplný rozhodovací strom obsahuje každou permutaci alespoň jednou. Strom má tedy nejméně n! listů, zároveň se počet listů dá omezit 2h , kde h je výška stromu. V nejhorším případě bude nutné projít celou výšku stromu, T (n) = h. Potom n! ≤ 2h , log2 n! ≤ h , n
n! ≈ n a n log2 n − n log2 e ≤ h = T (n). Získali jsme omezení pro nejhorší případ Ω(n log2 n). Řazení haldou a slučováním mají v nejhorším případě složitost O(n log2 n), jsou tedy asymptoticky optimální.

22. Generičnost

Máme-li skupinu reálných předmětů se společnými vlastnostmi popsánu třídou, lze přidáním dalších vlastností omezit tuto skupinu jen na ty předměty, které se podobají i v nově přidaných vlastnostech. Stejně lze ale vlastnosti i ubírat, čímž zahrneme pod takovou třídu více předmětů. Odebereme-li z definice třídy všechny vlastnosti, zahrneme tím pod ni všechny ostatní třídy, vytvoříme třídu obecnou, obvykle pojmenovanou Object. Toho lze využít při implementaci ADT, které jsou pak univerzální – generické.

23. Dědičnost

Vytvoření specializovanější třídy přidáním dodatečných vlastností nebo operací k jiné třídě se nazývá dědičnost. Nová třída dědí vlastnosti a metody této třídy a přidává k nim další, je podtřídou (potomkem) a třída, ze které byla odvozena je nadtřídou (rodičovskou třídou). Vzniká hierarchie tříd, kdy nadtřída třídy je i nadtřídou jejích potomků a obráceně. Instanci podtřídy pak můžeme reprezentovat i jako instanci její nadtřídy, ovšem pak lze přistupovat pouze k vlastnostem a metodám definovaným v této nadtřídě. Nejvyšší nadtřídou bývá třída Object. ADT s položkami typu nadtřídy lze používat pro položky libovolné její podtřídy, lze vytvořit generické ADT.

24. Rozhraní

Rozhraní se podobá třídě, ale obsahuje pouze hlavičky metod, nikoliv jejich implementaci. Třída může rozhraní použít (implementovat, zdědit) a potom v ní musí být obsažena implementace všech metod rozhraní. Jedna třída může implementovat několik rozhraní, čímž lze obejít nebezpečnou vícenásobnou dědičnost. Instanci třídy implementující rozhraní lze reprezentovat jako instanci tohoto rozhraní s přístupem omezeným pouze na metody rozhraní. Podobnou kategorií jsou abstraktní třídy, které mohou obsahovat jak pouze hlavičky metod (označovaných jako abstraktní), tak i metody implementované.

25. Algoritmická řešitelnost problémů

Problém definujeme jako binární relaci mezi množinou instancí problému I (tj. množinou všech možností vstupu) a množinou řešení S. Dvě instance mohou mít stejné řešení, stejně tak jedna může mít více řešení. Algoritmická řešitelnost zkoumá, zda pro všechny formulovatelné problémy lze nalézt algoritmus řešení. Ve 30.l. 20.st. objevil Alan Turing formální opis algoritmu – Turingův stroj, ve spojení s Alonzem Churchem vytvořili Churchovu-Turingovu tezi a sice, že každý algoritmus (ne problém!) lze vykonat Turingovým strojem. Tezi lze vyvrátit nalezením algoritmu nevykonatelného TS. Moderní programovací jazyky pak byly navrženy tak, aby libovolný program šlo převést na TS a naopak. Problém, který nelze vyřešit pomocí TS, tedy ani pomocí programu, je

pak algoritmicky neřešitelný. Vytvoříme-li takový rozhodovací problém (řešení je ano/ne), který pro každý rozhodovací algoritmus a alespoň jeden jeho vstup dává pro tento vstup opačnou odpověď, než daný algoritmus se stejným vstupem, takový problém není řešitelný žádným z těchto algoritmů a je tedy nerozhodnutelný. První takový problém – problém zastavení – našel Turing: program má o všech vytvořitelných programech rozhodnout, zda pro každý ze vstupů daný program zastaví (tehdy vrací jeho výstup v podobě přirozeného čísla + 1) nebo nezastaví (pak vrací 0). Takový program ale nedokáže rozhodnout sám o sobě (pokud by byl vytvořitelný, patřil by mezi zkoumané programy také a musel by ve svém výstupu vrátit svůj výstup + 1).

26. Klasifikace problémů

Problémy můžeme rozdělit na algoritmicky řešitelné a neřešitelné, v případě rozhodovacích rozhodnutelné a nerozhodnutelné. Řešitelné problémy můžeme dále dělit na základě toho, jak závisí doba výpočtu nejefektivnějšího algoritmu na velikosti vstupních dat. Potom rozlišujeme problémy, jejichž řešení je možné získat v polynomiálním čase (lehké, složitost O(nk)) a v nepolynomiálním čase (těžké). Třída složitosti P pak obsahuje všechny problémy řešitelné s O(nk). Další skupinou problémů jsou takové, které sice nelze řešit v polynomiálním čase, ale lze ověřit správnost poskytnutého řešení v polynomiálním čase. To je třída složitosti NP (nedeterministicky polynomiální). Platí $P \subseteq N P$, ale předpokládá se, že $P = N P$. Najdeme-li pro dva rozhodovací problémy A, B funkci, která vstupy A převede na vstupy B tak, aby B dávalo stejné výsledky jako A, je problém A převeditelný na problém B a není tedy těžší než B. Pokud lze převodní funkci najít v polynomiálním čase, je A polynomiálně převeditelný na B ($A \leq P B$). Pak $B \in P \land A \leq P B \Rightarrow A \in P$ a složitost algoritmu A je součtem složitosti B a složitosti výpočtu převodní fce. Nejtěžšími problémy jsou NP-úplné problémy. Na ně lze polynomiálně převést každý problém z třídy NP, tedy T je NP-úplný, jestliže patří do NP a je NP-těžký ($T \in N P \land S \leq P T, \forall S \in N P$). Existuje-li polynomiální algoritmus pro libovolný NP-úplný problém, pak $P = N P$, naopak neexistuje-li pro žádný, pak $P = N P$.