Vizualizácia základných algoritmov

BAKALÁRSKA PRÁCA

Pavol Szórád

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Študijný odbor: 9.2.1 INFORMATIKA

Školiteľ bakalárskej práce:
doc. RNDr. Juraj Procházka, CSc.

BRATISLAVA 2007

Abstrakt

SZÓRÁD, Pavol: *Vizualizácia základných algoritmov.* [Bakalárska práca] / Pavol Szórád. - Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra informatiky. - Školiteľ: doc. RNDr. Juraj Procházka, CSc. -Bratislava: FMFI, 2007. -77 s.

Práca prezentuje spôsob, ako zlepšiť výučbu základných algoritmov a dátových štruktúr – vypracovaním internetovej stránky *Zbierka základných algoritmov a dátových štruktúr* so zameraním na vizualizáciu algoritmov. Práca je rozdelená na dve časti. V prvej je spracovaná problematika výberu optimálneho prostriedku na reprezentáciu zbierky a zvolenia princípov na jej tvorbu. Pripravená je šablóna na prehľadný popis algoritmov. V druhej časti sú aplikované tieto zásady a je spracovaných päť kapitol zo základov algoritmov a dátových štruktúr. Súčasťou práce je kompletná internetová stránka vypracovaná podľa uvedených princípov obohatená o vizualizácie a externé odkazy.

Kľúčové slová:
Algoritmická konštrukcia. Algoritmus. Asymptotická notácia. Backtracking. Binárny strom. Bubblesort. Dátová štruktúra. Front. Heapsort. Quicksort. Rekurzia. Vizualizácia. Vyhľadávanie. Strom. Triedenie. Teória grafov. Zásobník.

Predkladaná bakalárska práca vznikla pod odborným dohľadom doc. RNDr. Juraja Procházku, CSc., za čo mu úprimne ďakujem.

Autor

Čestne vyhlasujem, že túto bakalársku prácu som vypracoval samostatne s použitím citovaných zdrojov.

…….……

Obsah

Úvod

Algoritmy a dátové štruktúry sú v istom zmysle základným kameňom informatiky. Zjednodušene povedané, dátová štruktúra dovoľuje uchovávanie informácie a algoritmus zabezpečuje jej spracovávanie. Vzhľadom na to, že informatizácia prenikla takmer do všetkých sfér ľudskej spoločnosti, venuje sa množstvo úsilia na nájdenie takých algoritmov a dátových štruktúr, ktoré by umožňovali čo najefektívnejšie narábať s veľkým množstvom dát. Najlepšie riešenia konkrétnych problémov však bývajú zvyčajne mimoriadne zložité, a teda obťažné na pochopenie.

Z hľadiska pedagogického procesu sú zaujímavé jednoduché riešenia, ktoré názorne popisujú niektoré charakteristické postupy v procese algoritmizácie. O týchto algoritmoch a dátových štruktúrach môžeme povedať, že patria medzi *základné*. To znamená, že ich vlastnosti sú dobre známe a sú vo významnej miere používané v súčasných aplikáciách. Pochopenie princípu ich fungovania je často nevyhnutné, pretože všetky oblasti informatiky v značnej miere predpokladajú zvládnutie týchto poznatkov.

Problematika základných algoritmov a dátových štruktúr je preto jedným z ťažiskových tematických okruhov štúdia informatiky. Jej výučba by mala byť na vysokej úrovni, pričom meradlom kvality nie je čo najväčší rozsah učiva, ale vhodne zvolená metóda výučby. Mala by ponúkať ucelený pohľad na problematiku a zároveň dovoľovať hĺbkové štúdium konkrétneho algoritmu alebo dátovej štruktúry. To možno docieliť vypracovaním pomocného textu k prednáške, ktorý dopĺňa výklad a v značnej miere sa opiera o metódu názornosti. Táto metóda patrí medzi najefektívnejšie metódy výučby. Vizuálny vnem na základe názornej ukážky algoritmu je pri prvej konfrontácii s algoritmom často dôležitejší ako presný zápis kódu algoritmu.

Na mnohých univerzitách vo svete vznikajú k jednotlivým prednáškam z oblasti algoritmov a dátových štruktúr interaktívne internetové stránky s cieľom poskytnúť študentom kvalitný učebný materiál obohatený práve o vizualizácie algoritmov. Internetové stránky nie sú ohraničené podobnými limitáciami ako tlačený text. Ich dynamickosť umožňuje neustále pridávanie nového obsahu a v konečnom dôsledku vytvorenie komunity čitateľov a prispievateľov. Podľa našich prieskumov neexistuje na žiadnej univerzite na Slovensku podobná iniciatíva. Našou ambíciou však nie je

spracovať vizualizáciu ku všetkým existujúcim algoritmom, ako skôr vytvoriť vhodné podmienky a dať impulz k tomu, aby sa tento proces naštartoval.

Cieľom tejto práce je teda zostaviť prehľadne a názorne koncipovanú zbierku základných algoritmov a dátových štruktúr so zameraním na ich vizualizáciu. V rámci tohto cieľa sme kládli dôraz na stanovenie presných kritérií na proces spracovávania údajov pre tento účel. Následnou analýzou týchto teoretických poznatkov sme zvolili formu uverejnenia zbierky – internetovú stránku. Tá umožňuje použiť animácie ako hlavný prostriedok vizualizácie algoritmov a má rad ďalších pozitívnych vlastností ako napríklad dobrá dostupnosť a možnosť pružne meniť obsah. V zbierke spracovávame týmto spôsobom celkovo päť kapitol zo základných algoritmov a dátových štruktúr, ktoré predstavujú jadro tejto problematiky.

Cieľovou skupinou, pre ktorú je práca určená, je široká základňa študentov informatiky a študentov príbuzných vedných odborov. Výsledky práce využijú pri samoštúdiu a tiež ako doplnok k prednáškam z algoritmov a dátových štruktúr. Stránka je samozrejme verejná a prístupná pre všetkých, ktorí sa chcú v danej problematike zdokonaliť.

1 Zbierka základných algoritmov a dátových štruktúr

1.1 Norma zbierky

Prehľadnosť zbierky zabezpečíme zavedením spoločnej *normy*. Norma definuje požiadavky na vytvorenie obsahu, jeho členenie a spracovávanie tém. Zachováva tiež konzistentnosť pri formátovaní textu, použití obrázkov, tabuliek, grafov atď. Môže sa líšiť stupňom detailnosti podľa zvolenej *formy* učebnice.
Predkladaný návrh normy vznikol analýzou priebehu výučby algoritmov. Predstavuje jedno z mnohých možných riešení organizácie textu a vizualizácií k prednáške zo základov algoritmov a dátových štruktúr.

Vytvorenie zbierky základnych algoritmov a dátových štruktúr podľa tejto normy pozostáva z nasledovných krokov:

- Presné definovanie rozsahu učiva a jeho rozčlenenie na časti (kapitoly a témy).
- Usporiadanie tém podľa zvoleného kľúča (zvyčajne od jednoduchých tém po zložitejšie).
- Identifikovanie jednotlivých algoritmov a dátových štruktúr.
- Spracovanie textu, obrázkov, príkladov zdrojového kódu a ostatných súčastí podľa šablóny.
- Kontrola konzistentnosti (ak nastali zmeny v šablóne; ak témy vypracovávali viacerí ľudia).
- Údržba (priebežné pridávanie obsahu, opravy chýb).

1.2 Šablóna algoritmu

Šablóna je vopred dohodnuté a schválené členenie textu na logické celky v rámci témy. V jednej norme môže byť definovaných viac šablón, pre každý typ obsahu iná. Pre typ obsahu „algoritmus“ sa ukazuje byť najvhodnejšie nasledujúce členenie:

Zaradenie a popis

V časti „Zaradenie algoritmu“ sa uvedie, do akých rôznych množín problémov algoritmus patrí. Príklad: „Mergesort je *triediaci algoritmus*, ktorý realizuje metódu *triedenia zlučovaním*.“ Z popisu algoritmu musí byť čitateľovi jasné, pre aké konkrétne použitie je daný algoritmus vhodný. Taktiež sa uvedie za akých podmienok. Popis môže vystihovať základnú myšlienku algoritmu, tak ako aj niektoré jeho netriviálne charakteristiky. Má byť krátky a výstižný. Ak však existuje nejaká podstatná vlastnosť algoritmu, ktorú chceme spomenúť priamo v popise, môžeme ju uviesť. Viditeľne však oddelíme tento text od pôvodného krátkeho popisu. Takto sa síce daná vlastnosť nachádza v popise, nenarúša však pôvodný zámer popisu – výstižnosť.

Motivácia a špecifikácia úlohy

Motivácia je cieľ použitia algoritmu zapísaný prirodzeným jazykom. Napríklad „nájdenie čo najkratšej cesty z bodu A do bodu B“. Tento cieľ zapísaný formálne sa volá špecifikácia úlohy. Obsahuje informácie o vstupných a výstupných podmienkach. Ak je pre skupinu úloh táto časť rovnaká, môže byť uvedená v spoločnej časti pre túto skupinu (tj. na začiatku kapitoly), a nie je potrebné ju uvádzať pri každom algoritme. Napríklad cieľ všetkých triediacich algoritmov sa dá zjednodušene špecifikovať ako utriedenie jednorozmerného poľa prvkov.

Myšlienka algoritmu

Čo najzrozumiteľnejšie formulovaná myšlienka algoritmu v prirodzenom jazyku. Vo väčšine prípadov to znamená vymenovať poradie hlavných krokov algoritmu. Čitateľ by mal mať predstavu o použitých algoritmických konštrukciách a dátových štruktúrach. Podľa tohto textu by už malo byť zrejmé, ako napísať kód programu.

Vizualizácia

Najdôležitejšou netextovou časťou popisu algoritmov je *vizualizácia*. Algoritmus by mal byť po prečítaní časti *myšlienka algoritmu* a zhliadnutí vizualizácie intuitívne pochopiteľný. Možnosti vizualizácie sa líšia podľa použitej formy učebnice. Tlačený text nedovoľuje zobrazovať napríklad animácie. Prezentácia a internetová stránka však poskytujú možnosť prehrávať multimediálny obsah a tým rapídne rozširujú možnosti

vizualizácie algoritmov. Napriek tomu je to často vynechávaná časť výučby algoritmov, najmä pre nedostatok kvalitných vizualizácií.

Zápis algoritmu

Zápis algoritmu v niektorom programovacom jazyku alebo v pseoudokóde. Výber jazyka alebo pseudokódu je koncipovaný so zreteľom na maximalizáciu prehľadnosti kódu.

Vlastnosti algoritmu

Do časti *vlastnosti algoritmu* sa píšu všetky dôležité charakteristiky, ktoré nemajú miesto v ostatných častiach šablóny. Hlavnou časťou je uvedenie časovej a pamäťovej efektívnosti (zložitosti) algoritmu.

Experimentovanie

Niekedy je na hlbšie pochopenie algoritmu potrebné zabezpečiť istý druh konfrontácie s jeho konkrétnou implementáciou. Do časti *experimentovanie* sa môžu dať odkazy na takéto implementácie. Najčastejšie sú to internetové stránky s appletmi, kde sa dajú v kontrolovanom prostredí vyskúšať popisované algoritmy.

1.3 Forma zbierky

Pod formou rozumieme druh média určeného na prezentáciu učebného textu. Vo väčšine prípadov je to tlačený text. Čo sa však týka multimediálneho obsahu, je celkom nepostačujúca. Omnoho lepším riešením je internetová stránka. Je dostupná pre širší okruh čitateľov ako ktorákoľvek kniha, umožňuje rýchlo a ľahko meniť obsah a predovšetkým ponúka možnosť zobrazovania animovaných vizualizácií.

1.4 Internetová stránka

Predkladaná práca vznikla na základe priamej aplikácie vyššie uvedených zásad. Má dve časti: vytlačenú textovú verziu a internetovú stránku. Internetová stránka je dostupná na adrese http://www.sprite.edi.fmph.uniba.sk/~szorad/. Dizajn stránky je *open-source* a je stiahnutý zo stránky www.oswd.org. Používa technológie XHTML a

CSS. Stránku je možno spustiť *offline* z disku počítača bez zmeny funkčnosti. Dizajn má dva varianty, jeden je optimalizovaný pre prehliadač Internet Explorer a druhý pre ostatné prehliadače. Je navrhnutý tak, aby nepôsobil rušivým dojmom a aby ponúkal maximálny komfort pri prezeraní stránky. Štandardne je rozdelený na hornú lištu a bočný panel. Horná lišta zobrazuje aktuálnu tému a ponúka prechod do zoznamov tém. Bočný panel ponúka možnosť prechodu medzi témami a obsahuje osnovu stránky.

Z úvodnej stránky je možné dostať sa priamo k nasledovným podstránkam:

- *Výber kapitoly* – tu sú zoradené všetky spracované témy podľa kapitol a obtiažnosti.
- *Algoritmy* – abecedný zoznam spracovaných algoritmov.
- *Dátové štruktúry* – abecedný zoznam spracovaných dátových štruktúr.
- *Externé odkazy* – rôzne odkazy na externé stránky.
- *Vizualizácie* – zoznam všetkých vizualizácií na stránke, rozdelený do častí Statické obrázky a Animácie
- *Experimentovanie* – zoznam odkazov na java applety s experimentovaním na algoritmoch

Spracovanie tém je takmer totožné s vytlačenou textovou verziou, obsahuje však navyše časti *vizualizácia* a *experimentovanie* pri každom algoritme.V časti *experimentovanie* sa nachádzajú odkazy na zaujímavé stránky k danému algoritmu. Sú to najčastejšie java applety, ktoré umožňujú trasovať algoritmus na užívateľom vybraných vstupných dátach. V časti *vizualizácia* sa nachádzajú samotné obrázky a animácie, na ktoré je vlastne zbierka zameraná. Všetky obrázky (až na výnimky, pri ktorých je to explicitne uvedené) sú šírené pod licenciou public domain. Animácie sú formátu *flash* a sú vytvorené v programe Macromedia Flash MX 2004. Šírené sú pod licenciou public domain.

2 Základy algoritmizácie

2.1 Prerekvizity

Táto zbierka algoritmov a dátových štruktúr predpokladá, že čitateľ sa už mal možnosť zoznámiť s informatikou a hlavne s programovaním v niektorom programovacom jazyku. Podkladom pre stanovenie prerekvizít nutných na pochopenie ostatných častí tejto zbierky algoritmov bol dokument „Vzdelávací štandard z informatiky pre gymnázium“, ktorý bol vypracovaný Štátnym pedagogickým ústavom. Na základe časti „Algoritmy a algoritmizácia“ tohto dokumentu sú zostavené nasledovné požiadavky na vedomosti a zručnosti.

Sú to schopnosti:

- zostaviť jednoduchý algoritmus z bežného života (využívajúci sekvenciu príkazov, podmienky a cykly),
- zostaviť jednoduchý algoritmus z množiny príkazov,
- zapísať algoritmus.
- chápať pojem postupnosť príkazov, vetvenie, cyklus, procedúra,
- vysvetliť pojem premennej,
- poznať vstupno-výstupné operácie jedného programovacieho jazyka,
- čítať a modifikovať jednoduchý program.

2.2 Dátová štruktúra

Počítač dokáže uchovávať a spracovávať veľké množstvo údajov. Formálne dátové štruktúry umožňujú štrukturovať tento priestor na logické celky. Dátová štruktúra je spôsob ako uchovávať a organizovať dáta tak, aby boli ľahko prístupné a modifikovateľné.

Použitie

Žiadna dátová štruktúra sa nehodí na všetky možné aplikácie, a preto je dôležité poznať možnosti a obmedzenia viacerých z nich. Niekedy používame dátové štruktúry, aby nám

umožnili robiť viac, napríklad uskutočňovať rýchle vyhľadávanie alebo triedenie. Niekedy zase potrebujeme, aby dátová štruktúra bola špecializovaná a aby nám v konečnom dôsledku dovoľovala menej. Napríklad zásobník je limitovanou formou všeobecnejšej dátovej štruktúry. Tieto obmedzenia dovoľujú nezaťažovať sa podrobnosťami pri písaní algoritmov. Voľba správnej dátovej štruktúry umožňuje vykonávanie kritických operácií za použitia minima zdrojov – podľa možnosti výpočtového času a operačnej pamäte.

Pri návrhu mnohých typov programov je voľba dátových štruktúr primárnou úlohou. Skúsenosti z budovania veľkých systémov ukázali, že zložitosť implementácie, kvalita a výkonnosť konečného výsledku závisí do veľkej miery na voľbe najvhodnejších dátových štruktúr. Potom, ako sú zvolené dátové štruktúry, je voľba algoritmov pomerne zjavná. Niekedy to funguje aj opačne – dátová štruktúra sa zvolí, pretože na riešenie úlohy existuje algoritmus, ktorý najlepšie pracuje s istou dátovou štruktúrou. V každom prípade je voľba dátovej štruktúry kritická. Navoľ si, aby si nemal jednoslabičné predložky na konci riadku

Pojem dátový typ

Každá konštanta, premenná, výraz alebo funkcia sú určitého typu. Tento typ určuje množinu hodnôt, do ktorej daná konštanta prináleží, alebo ktoré môže daná premenná, či výraz nadobudnúť, alebo ktoré možno danou funkciou vypočítať. Zvyčajne je typ konštanty, premennej alebo funkcie explicitne stanovený v rámci deklarácie príslušného objektu. Objekt musí byť vždy deklarovaný skôr ako je použitý, lebo kompilátor musí zvoliť reprezentáciu príslušného objektu v pamäti počítača.

Podstatné vlastnosti pojmu dátový typ teda sú:

- Typ údajov určuje množinu hodnôt, do ktorej daná konštanta prináleží, alebo ktoré môže daná premenná či výraz nadobudnúť, alebo ktoré môže daným operátorom alebo funkciou vypočítať.
- Typ hodnoty konštanty, premennej alebo výrazu sa dá určiť z ich deklarácie alebo zápisu bez nevyhnutnosti uskutočniť vlastný výpočtový proces programu.

- Každý operátor alebo funkcia predpokladá argumenty presne definovaných typov a poskytuje výsledok tiež presne definovaného typu. V prípade, že operátor pripúšťa argumenty rôznych typov (napríklad ak sa operátor „+“ používa na spočítavanie celých i reálnych čísel), dá sa typ výsledku určiť podľa špecifických pravidiel príslušného programovacieho jazyka.

Abstraktný dátový typ

Pretože dátové štruktúry sú vyššou úrovňou abstrakcie, umožňujú operácie na množine dát, ako sú napríklad pridávanie položky do zoznamu, alebo nájdenie prvku s najväčšou prioritou vo fronte. Takáto dátová štruktúra, ktorá umožňuje operácie, sa volá abstraktný dátový typ (abstract data type). Abstraktný dátový typ dokáže minimalizovať závislosti v kóde, čo je dôležité pri častých zmenách. Jednotlivé implementačné detaily sú skryté a to dovoľuje nahrádzať už raz zvolené špeciálne dátové štruktúry za také, ktoré zdieľajú spoločné vlastnosti na vyššej úrovni.

Programovacie jazyky prinášajú množinu vstavaných typov, napríklad celočíselný typ (integer), alebo čísla s desatinnou čiarkou (floating-point numbers). Tieto dátové typy sú abstrakciami, lebo skrývajú detaily o tom, ako procesor jednotlivé operácie na nich vykonáva.

2.2.1 Základné dátové štruktúry

Dátový (údajový) typ je presne definovaný svojím menom, neprázdnou množinou hodnôt a operácií. Každá premenná, výraz alebo funkcia môžu nadobúdať iba hodnoty jedného typu. Programovacie jazyky ponúkajú veľkú škálu rôznych vstavaných dátových typov. Dátové typy rozdeľujeme na jednoduché (štandardné, definované) a štrukturované. Taktiež môžu byť ordinálneho typu. Ordinálny typ je taký, pre ktorý má zmysel a je možné definovať funkcie:

- *Pred* (predchodca): $Pred(t_i) = t_i + 1$
- *Succ* (nasledovník): $Succ(t_i) = t_i - 1$
- *Ord* (poradové číslo): $Ord(t_i) = i-1$

Jednoduché (primitívne) dátové typy

Štandardné typy

Sú to programátormi najčastejšie, intuitívne používané typy. Taktiež sú s menšími rozdielmi vstavané do väčšiny programovacích jazykov. Patria sem:

- *integer* – množina celých čísel,
- *char* – množina znakov,
- *boolean* – množina hodnôt true alebo false,
- *real* – podmnožina reálnych čísel.

Definované typy

Sú to typy, pri ktorých sa v podstate vytvorí nový dátový typ pomocou definície množiny, ktorej tento prináleží. Definujú sa v úseku definícií a deklarácií stanovením mena, typu a množiny hodnôt.

Množina hodnôt môže byť:

- *interval* – neprázdna súvislá množina hodnôt z definovaného ordinálneho typu,
- *vymenovaný typ* – množinu hodnôt tvoria vymenované hodnoty $t_1<t_2<...<t_n$

Štrukturované dátové typy

Štrukturovaný dátový typ používa vo svojej definícii jeden a viac primitívnych typov. Základnými metódami štrukturovania sú pole (array), záznam (record), množina (set) a postupnosť (súbor).

Príklady štruktúrovaných dátových typov:

- bunka s využitím smerníka,
- pole (array, dimension),
- dátum a čas (date, time, datetime),
- množina (set),
- slovník (dictionary),
- zoznam (list),
- zásobník (stack),
- front (queue),
- strom (tree),

- graf,
- halda (heap).

Pole

Pole sa skladá z pevného počtu zložiek rovnakého typu. Ku každej tejto položke pristupujeme pomocou indexu. Je to vhodná štruktúra, ak máme spracovávať konečnú skupinu rovnakých údajov.

Záznam

Záznam je štrukturovaný dátový typ, ktorý sa skladá z pevného počtu položiek rovnakého typu. Využitie má pri hromadnom spracovávaní údajov, ktoré treba spojiť do jedného logického celku – napríklad pri zaznamenávaní údajov o osobe (meno, priezvisko, vek, adresa, atď.).

Súbor

Dátový typ súbor sa používa, ak potrebujeme, aby sa údaje uchovali aj po vypnutí počítača – je potrebné ich uložiť na vonkajšie pamäťové médium. Súbor je štrukturovaný údajový typ, ktorý sa skladá teoreticky z neobmedzeného počtu zložiek rovnakého typu. Prakticky je pamäť obmedzená veľkosťou vonkajšej pamäte.

String

String je štrukturovaný dátový typ, ktorým definujeme reťazce (tzn. nejakú postupnosť znakov). Funguje približne ako pole typu char, ale ponúka aj veľa ďalších funkcií.

2.2.2 Spájaný zoznam (Linked list)

Najjednoduchší spôsob vzájomného združenia alebo pospájania prvkov nejakej množiny je ich zoradenie do zoznamu. V týchto prípadoch je potrebný iba jeden spojovací článok pre každý prvok množiny, ktorý stačí na určenie nasledujúceho prvku v množine.

Spájaný zoznam je dátovou štruktúrou, ktorá aplikuje tento princíp. Skladá sa z postupnosti uzlov rovnakého typu, kde každý uzol pozostáva z dvoch zložiek. Prvou

zložkou sú *referencie* (smerníky) ukazujúce na nasledovníka a predchodcu uzla a druhá zložka je samotná *hodnota kľúča* spolu s ďalšími dátami. Na rozdiel od poľa, v ktorom je poradie stanovené indexmi, je teda poradie v spájanom zozname určované ukazovateľmi vo všetkých objektoch.

Ak je *x* prvok zoznamu, *next[x]* ukazuje na jeho nasledovníka v zozname a *prev[x]* na jeho predchodcu. Ak *prev[x]=nil*, prvok nemá predchodcu a preto je prvým prvkom (*head*) zoznamu. Ak next*[x]=nil*, prvok nemá nasledovníka a je posledným prvkom (*tail*) zoznamu. Atribút *head[L]* ukazuje na prvý prvok zoznamu L. Ak *head[L]=nil*, zoznam je prázdny.

Hlavnou výhodou spájaného zoznamu je fakt, že usporiadanie dát v spájanom zozname sa môže líšiť od usporiadania dát v pamäti. Nové usporiadanie jednotlivých uzlov môže teda spĺňať rôzne špeciálne potreby. Spájané zoznamy tiež poskytujú jednoduchú a flexibilnú reprezentáciu pre dynamické množiny podporujúce operácie ako sú napríklad Search, Insert a Delete.

Typy spájaného zoznamu

Zoznam s jednoduchou väzbou

Tento typ zoznamu má smerník iba na svojho nasledovníka. Ak je uzol posledný, smerník má hodnotu *nil*.

Zoznam s dvojitou väzbou

Je to typ spájaného zoznamu, ktorý má smerník na svojho nasledovníka a aj smerník na svojho predchodcu. Ak je uzol prvý, hodnota smerníka na predchodcu má hodnotu *nil* a obdobne, ak je uzol posledný, tak hodnota smerníka na nasledovníka má tiež hodnotu *nil.*

Kruhový zoznam

V kruhovom zozname smerník *prev* hlavy (head) ukazuje na posledný prvok a smerník *next* posledného prvku ukazuje na hlavu zoznamu.

Zarážky

Spájané zoznamy majú niekedy špeciálny uzol, ktorý sa volá zarážka (*sentinel node*). Zarážka označuje začiatok (koniec) zoznamu a nemá žiadnu hodnotu kľúča. Používa sa hlavne na zjednodušenie a zrýchlenie niektorých operácií. Použitím zarážky sa docieli, že každý uzol má predchodcu (nasledovníka) a že každý zoznam má prvý (posledný) uzol. V princípe to znamená, že miesto toho, aby smerníky mali hodnotu *nil*, budú ukazovať na zarážku.

Operácie

Vyhľadávanie v spájanom zozname

Procedúra *List-Search(L,k)* nájde v zozname *L* prvý prvok s kľúčom *k* jednoduchým lineárnym prehľadávaním, pričom vráti ukazovateľ na tento prvok. Ak sa v danom zozname taký prvok nenachádza, procedúra vráti hodnotu *nil*. Na vyhľadanie prvku v zozname pozostávajúcom z *n* prvkov, potrebuje procedúra čas $\Theta(n)$.

```
LIST-SEARCH
  x:=head[L]
  while (x<>nil) and (key[x]<>k) do
    x:=next[x]
  return x
```

Vkladanie do spájaného zoznamu

Procedúra *List-Insert* umiestni daný prvok *x* na začiatok zoznamu. Čas behu je $O(1)$.

```
LIST-INSERT
  next[x]:=head[L]
  if head[L]<>nil then
    prev[head[L]]:=x
  head[L]:=x
  prev[x]:=nil
```

Vymazávanie zo spájaného zoznamu

Procedúra *List-Delete* odstráni prvok *x* zo spájaného zoznamu *L*. Aby mohol byť prvok odstránený, musí naň procedúra dostať ukazovateľ. Prvok bude potom zo zoznamu vyňatý jednoduchou zmenou ukazovateľov. Ak teda chceme vymazať prvok s daným kľúčom, musíme najprv vyvolať procedúru *List-Search*

```
LIST-DELETE
  next[prev[x]]:=next[x]
  prev[next[x]]:=prev[x]
```

Vlastnosti

Spájaný zoznam je dátový typ odkazujúci sám na seba, lebo jednotlivé uzly sú toho istého typu. Umožňuje vkladanie a mazanie uzlov v konštantnom čase, ale neumožňuje náhodný prístup. Aby bol zoznam lineárny, nesmú existovať cykly vo vzájomných odkazoch.

2.2.3 Zásobník (Stack)

Zásobník (stack) je najdôležitejšia dátová štruktúra s obmedzeným prístupom. Pre zásobník je charakteristický spôsob narábania s údajmi – posledné uložené údaje budú čítané ako prvé. Preto sa používa taktiež výraz LIFO z anglického „Last In, First Out“. Vyžadujeme, aby implementácia zásobníka podporovala dve základné operácie: *push* (vlož prvok do zásobníka) a *pop* (vyber prvok zo zásobníka). Pre manipuláciu s uloženými dátovými položkami sa udržuje *ukazovateľ zásobníka*, ktorý udáva relatívnu adresu poslednej pridanej položky – *vrchol zásobníka*.
Zásobníky hrajú dôležitú úlohu pri syntaktickej analýze a konštrukcii prekladačov. Prvoradý význam majú pri realizovaní rekurzií, kde slúžia na zapamätávanie návratovej adresy. V praxi sa zásobníky realizujú ako jednorozmerné polia alebo lineárne spájané zoznamy.

2.2.4 Front (Queue)

Je to ďalšia významná abstraktná dátová štruktúra s obmedzeným prístupom k postupnosti prvkov. Na rozdiel od zásobníka však aplikácia používajúca front spracováva dáta v poradí, v akom prišli (odtiaľ ten názov). Front má označenie FIFO – „First In, First Out“, teda prvky, ktoré prvé prišli, aj prvé odídu. Vyžadujeme tiež len dve operácie: insert (vlož prvok do frontu – enqueue) a remove (vyber prvok z frontu – dequeue). Front sa používa keď sa rôzne objekty ukladajú po poradí, aby sa dali spracovávať neskôr.

2.3 Algoritmus

Pod pojmom algoritmus rozumieme presne definovanú konečnú postupnosť pravidiel, ktorých realizácia nám pre vstupné dáta umožní po konečnom počte krokov získať zodpovedajúce výstupné dáta. Možno ho špecifikovať ako počítačový program, formulovať ho v nejakom prirodzenom jazyku, alebo realizovať ako hardvérový dizajn. Mal by byť ale formulovaný natoľko precízne, že ho môže realizovať mechanické alebo elektronické zariadenie. Algoritmus udáva, ako sa vstupné dáta postupne transformujú na výstupné dáta. Opisuje teda zobrazenie f: $V -> W$ z množiny prípustných vstupných dát V do množiny výstupných dát W. Reprezentovať ho môžeme aj tak, že udáme matematicky presne definovaný stroj, ktorý ho realizuje krok za krokom. Jednoduchým modelom je napríklad Turingov stroj.

Vlastnosti algoritmu

Správnosť

Algoritmus je správny, ak pre každý zmysluplný vstup sa jeho vykonávanie zastaví v konečnom čase, pričom vráti správny výsledok. O čiastočnej správnosti hovoríme, ak sa algoritmus nezastaví na všetky zmysluplné vstupy. Nekorektný algoritmus sa buď nezastaví na všetky zmysluplné vstupy, alebo sa zastaví a ponúkne inú ako očakávanú hodnotu.

Elementárnosť

Jednotlivé činnosti postupu musia byť pre realizátora elementárne, teda také, ktoré vie zrealizovať. Napríklad pre žiaka prvého ročníka základnej školy nie je elementárna činnosť vypočítať tretiu mocninu čísla.

Determinizmus

Každý krok algoritmu musí byť jednoznačne a presne definovaný a je presne určené poradie krokov. V každom bode výpočtu musí byť jasné, ako má vykonávanie algoritmu pokračovať. Pre takýto presný zápis algoritmov boli navrhnuté

programovacie jazyky, v ktorých má každý príkaz jasne definovaný význam. Vyjadrenie algoritmu v programovacom jazyku sa nazýva program.

Rezultatívnosť

Pre rovnaké vstupné dáta a pri rovnakých podmienkach dáva realizácia postupu vždy rovnaké výstupné dáta.

Všeobecnosť

Algoritmus nerieši len jeden konkrétny problém (napr. vypočítať súčin 2x2), ale rieši všeobecnú triedu problémov rovnakého typu (napr. súčin dvoch čísel).

Konečnosť

Každý algoritmus musí skončiť po vykonaní konečného počtu krokov. Tento počet krokov môže byť ľubovoľne veľký (podľa rozsahu a hodnôt vstupných údajov), ale pre každý jednotlivý vstup musí byť konečný. Postupy, ktoré túto podmienku nespĺňajú, sa môžu nazývať výpočtové metódy. Špeciálnym príkladom nekonečnej výpočtovej metódy je reaktívny proces, ktorý priebežne reaguje s okolitým prostredím.

Efektivita

Aby algoritmus bol efektívny, musí realizovať úlohu v čo najkratšom čase a s použitím čo najmenšieho počtu prostriedkov.

Vstup

Algoritmus zvyčajne pracuje s nejakými vstupmi, veličinami, ktoré sú mu odovzdané pred začatím jeho vykonávania, alebo v priebehu jeho činnosti. Vstupy majú definované množiny hodnôt, ktoré môžu nadobúdať.

Výstup

Algoritmus má aspoň jeden výstup, veličinu, ktorá je v požadovanom vzťahu k zadaným vstupom, a tým tvorí odpoveď na problém, ktorý algoritmus rieši.

2.3.1 Základné algoritmické konštrukcie

Príkaz

Príkazy sú tie časti programu, ktoré menia stav programu (hodnoty premenných, obsahy vstupných a výstupných dát). Teda sú to časti, v ktorých pri vykonávaní programu dochádza k spracovaniu dát (na rozdiel od deklarácií). V imperatívnych programovacích jazykoch sa predpisy spracovania dát vyjadrujú pomocou príkazov.
Príkazy, ktoré nemožno ďalej rozložiť, sa nazývajú *elementárne príkazy.*

Sekvencia

Postupnosť za sebou vykonávaných príkazov môžeme považovať za jeden príkaz a označiť ako postupnosť (sekvenciu). Ide o sled za sebou idúcich štandardných algoritmických konštrukcií. Konvencia určuje, že sekvenčne riadené bloky majú len jeden vstup a jeden výstup.

Vetvenie

Vetvenie je miesto v programe, z ktorého sú dosiahnuteľné aspoň dve ďalšie miesta. Vo vyšších programovacích jazykoch ide o podmienený príkaz – teda o príkaz, ktorého vykonanie závisí od podmienky.
Podmienka je boolovský výraz, ktorý takto priamo ovplyvňuje tok riadenia v programe.
Poznáme úplné, neúplné a viacnásobné (case) vetvenie.

Cyklus (iterácia)

Postupnosť príkazov, ktoré sa môžu vykonať viackrát. Podľa výstavby sa cyklus delí na záhlavie cyklu a telo cyklu. Záhlavie cyklu obsahuje riadiace informácie pre počet prechodov cyklom. Telo cyklu je viackrát vykonateľná postupnosť príkazov.
Rozlišuje sa medzi viacerými druhmi cyklu:

Cyklus s daným počtom opakovaní

Počet prechodov sa určí krokovacou premennou, ktorá sa začínajúc začiatočnou hodnotou v každom kroku zväčší (inkrement), alebo zmenší (dekrement) o určenú hodnotu až po konečnú hodnotu. Pri jej dosiahnutí sa cyklus vykoná posledný raz.

Cyklus s podmienkou na začiatku

Ukončenie vykonávania cyklu je riadené *podmienkou prerušenia* v záhlaví cyklu. Telo cyklu sa vykonáva dovtedy, kým je splnená podmienka. Ak je podmienka hneď zo začiatku nesplnená, telo cyklu sa nevykoná ani raz.

Cyklus s podmienkou na konci

Ukončenie vykonávania cyklu je tiež riadené podmienkou prerušenia tak ako pri cykle s podmienkou na začiatku. Podmienka je však na konci, a teda telo cyklu sa vždy aspoň raz vykoná. Príkazy sa vykonávajú, až kým nenastane situácia, že je splnená podmienka prerušenia (na rozdiel od predchádzajúceho prípadu).

Podprogram

Je to pomenovanie pre ucelenú časť programu (určitá postupnosť príkazov) s možnosťou používať jej meno ako príkaz na vykonávanie tejto postupnosti na rôznych miestach programu. Podprogramy zvyčajne vykonávajú nejakú špecifickú úlohu a sú relatívne nezávislé od zvyšku programu. Podprogram je dôležitý prvok v algoritmoch, ktorý umožňuje štrukturalizáciu programu. Syntax mnohých programovacích jazykov priamo podporuje vytváranie, volanie a návrat súvisiaci s podprogramami.
Podprogramy majú mnoho výhod: vyhýbanie sa duplikovaniu kódu v rámci programu; možnosť opätovného použitia vo viacerých programoch; rozdelenie zložitejšieho problému na menšie logické podproblémy; zlepšenie čitateľnosti programu.
Časti podprogramu sú:

- telo podprogramu, ktoré sa vykoná, keď sa zavolá podprogram,
- parametre, ktoré sa odovzdajú podprogramu pri volaní,
- hodnota, ktorú podprogram (*funkcia*) vráti keď skončí.

Podprogramy ďalej delíme na:

Procedúry

Procedúra je podprogram, ktorý nevracia žiadnu návratovú hodnotu. Môže byť s parametrami, alebo bez parametrov.

Funkcie

Funkcia je podprogram, ktorý vracia návratovú hodnotu vopred určeného typu. Preto musí byť v tele funkcie príkaz, ktorý toto zabezpečí (napríklad príkaz „return“ v programovacom jazyku C).

2.3.2 Rekurzia

Rekurzia je často používaná technika zjednodušovania zložitejších problémov rozdelením pôvodného problému na podproblémy toho istého typu. Hovoríme, že objekt je rekurzívny, ak sa čiastočne skladá, alebo je definovaný pomocou seba samého. Sila rekurzie spočíva v možnosti definovať nekonečnú množinu objektov konečným príkazom. Podobným spôsobom možno nekonečný počet výpočtov opísať pomocou konečného rekurzívneho programu. Hoci program nemusí obsahovať explicitné opakovania, rekurzívne algoritmy sú najvhodnejšie pri riešení výpočtov, kde je problém priamo definovaný rekurzívnym spôsobom.

Vlastnosti

Dôležitým prostriedkom na vyjadrenie rekurzie v programoch je procedúra (podprogram), ktorej identifikátor slúži na rekurzívnu aktiváciu jej tela. Ak procedúra volá sama seba, hovoríme o *priamej rekurzii*. Ak podprogram A volá podprogram B a ten zas volá podprogram A, hovoríme o *nepriamej rekurzii*. Použitie rekurzie nemusí teda byť okamžite zrejmé z textu programu. Je zvykom združovať množinu lokálnych objektov (tj. lokálne premenné, konštanty, typy) s procedúrou. Každým rekurzívnym volaním takejto procedúry vznikne nová množina jej lokálnych premenných. Tieto premenné majú síce tie isté identifikátory, aké mali pri predošlom volaní procedúry, ale

ich hodnoty sú odlišné. Ku konfliktom neprichádza vďaka pravidlu viditeľnosti identifikátorov (vzťahujú sa vždy na ostatne vytvorenú množinu premenných). Dôležitým prvkom v rekurzii je ukončovacia podmienka, ktorá zaručí, že počet vnorení nebude nekonečný.

Použitie

Rekurzívne riešenia problémov slúžia v prvom rade na pochopenie problému. V princípe je možné transformovať ľubovoľný rekurzívny algoritmus na iteratívnu verziu, ale stráca sa jeho prehľadnosť. Rekurzia však nemusí byť automaticky neefektívna. Príkladom je quicksort alebo iné algoritmy operujúce na dynamických dátových štruktúrach.

Príklady rekurzie

Prirodzené čísla

- 1 je prirodzené číslo,
- nasledovníkom prirodzeného čísla je prirodzené číslo.

Stromové štruktúry

- *k* je strom (nazývaný prázdny strom),
- ak *l,m* sú stromy, tak *k* s ľavým synom *l* a pravým synom *m* je tiež strom.

Funkcia n-faktoriál n!

- 0!=1,
- ak n>0, tak n!=n.(n-1)!.

Fibonacciho postupnosť

- $f(0) = 0$,
- $f(1) = 1$,
- $f(n) = f(n-1) + f(n-2)$.

Fraktály

- Fraktál je nekonečné vetvenie definované určitým pravidlom. Je to jeden z prostriedkov súčasného modelovania, pričom platí, že po akomkoľvek zväčšení vyzerá obrázok úplne rovnako.

2.3.3 Prehľadávanie s návratom (backtracking)

Definícia

Jeden zo spôsobov hľadania riešenia pre všeobecné problémy je metóda pokusov a omylov. Celý proces sa rozkladá na niekoľko čiastkových úloh, ktoré sa často vyjadrujú pomocou rekurzie a spočívajú v preskúmaní konečného počtu podúloh. Opakovaním pokusov (vyhľadávaní) sa postupne vybuduje a skúma (zmenšuje) strom podúloh. Takéto algoritmické problémy sa dajú vyriešiť pomocou *prehľadávania s návratom*. Algoritmus funguje tak, že sa zoberú do úvahy všetky možnosti, ktoré môžu viesť k vyriešeniu problému. Potom sa postupne rekurzívne riešia jednotlivé podproblémy. Množina rekurzívnych volaní vytvára stromovú štruktúru, kde je postupne kontrolovaná každá podmnožina možností. Z toho vyplýva, že ak riešenie existuje, algoritmus ho určite časom nájde.

Vlastnosti

Backtracking je vo všeobecnosti neefektívny prístup, používajúci brute-force prístup. Pomocou rôznych optimalizačných techník sa dá zredukovať hĺbka stromu aj počet podstromov. Technika sa volá prehľadávanie s návratom, pretože po každom navštívenom liste stromu sa algoritmus vráti do zásobníka. Tam má uložené miesta, kde sa vnoril. Vyberie ďalšiu vetvu a znova sa vnorí.

Aby sa problém dal riešiť pomocou prehľadávania s návratom, jeho podproblémy sa musia určitým spôsobom podobať na pôvodný problém. Väčšina problémov sa dá na takýto tvar upraviť.

Použitie

Problém ôsmych dám

Je potrebné rozmiestniť osem dám na šachovnici tak, aby žiadna z nich neohrozovala niektorú z ostatných figúrok

```
procedure vyskusaj(tah:...);
begin
  zapis_tah_do_sachovnice;
  if koniec then
    vypis
  else
    // vyskúšaj nasledujúci ťah – všetky možnosti
    vyskusaj(nasledujúci_ťah);
  vymaz_tah_zo_sachovnice;
end;
```

Problém šiestich tiav

Šesť tiav pôjde v karaváne 6 dní. Vedecky sa ukázalo, že ak ťava pozerá na dopravnú značku celý deň, tak sa ju naučí. Preto zavesili na chrbty tiav 6 rôznych značiek. Úloha je navrhnúť rozloženie tiav do 6 karaván (pre každý deň iné) tak, aby sa každá naučila 5 rôznych značiek. Každý deň je jedna ťava vedúca a tá sa nič neučí.

Problém stabilného manželstva

Existuje množina A mužov a množina B žien. Každý muž a každá žena si stanovili rôzne požiadavky na svojich partnerov. Keď sa vyberie n takých dvojíc, v ktorých muž a žena nie sú spolu zosobášení, ale obaja by dali prednosť svojmu vybratému partnerovi pred svojím skutočným manželským partnerom, tak takéto priradenie partnerov sa volá nestabilné. Ak takéto dvojice neexistujú, hovoríme o stabilnom zväzku.

2.3.4 Pokročilé techniky

Rozdeľuj a panuj (Divide and Conquer)

Množstvo algoritmických problémov je, čo sa týka štruktúry, rekurzívnych a je teda možné rozdeliť ich na podproblémy, ktoré sa dajú riešiť obdobným spôsobom. Riešenie takýchto problémov pomocou metódy divide & conquer pozostáva z nasledujúcich častí:

- Rozdelenie problému na časti (divide).
- Rekurzívne vyriešenie každého z podproblémov (conquer). Ak je problém dostatočne malý, vyriešime ho nerekurzívne.
- Spojenie riešení podproblémov do riešenia pôvodného problému (combine).

Príkladom techniky divide & conquer sú algoritmy triedenia Quicksort a Mergesort.
Jednoduchší variant divide & conquer sa nazýva *decrease and conquer*. Tento vyrieši identický podproblém a výsledok použije na riešenie väčšieho problému. Príklad tejto techniky je binárne vyhľadávanie.

Dynamic Programming:

Keď optimálne riešenie problému je možné zostrojiť pomocou optimálnych riešení jeho podproblémov a je nutné podproblémy riešiť opakovane (keď je veľa rovnakých vetiev v algoritme prehľadávania s návratom), je možné ušetriť čas použitím techniky dynamického programovania. Najprv sa vyrieši každý podproblém systémom „zdola“ – od najmenšieho po najväčší. Riešenie jednotlivých podproblémov sa ukladá do tabuľky. Takto sa každý podproblém navštívi a rieši len raz.

Hlavný rozdiel medzi dynamickým programovaním a divide & conquer je, že podproblémy v divide & conquer sú relatívne nezávislé a v dynamickom programovaní sa prekrývajú. Rozdiel medzi dynamickým programovaním a priamou rekurziou je v zapamätávaní si rekurzívnych volaní. Keď sú podproblémy nezávislé a nie je tam žiadne opakovanie, pamätanie údajov nepomáha k nájdeniu riešenia. Ak však je problém vhodne riešiteľný dynamickým programovaním, je možné zredukovať exponenciálnu časovú zložitosť na polynomiálnu.

Greedy technika

Greedy algoritmus je podobný dynamickému programovaniu s tým rozdielom, že nie je potrebné poznať riešenie všetkých podproblémov v každom bode riešenia. Namiesto toho môže byť zvolená možnosť, ktorá je v danom okamihu najvýhodnejšia. Táto technika neskúma všetky možnosti a pre veľa problémov nedáva správne výsledky. Ak však funguje, je to najrýchlejšia metóda. Naprogramovať algoritmus, ktorý používa greedy techniku je jednoduché. Dokázať jeho správnosť však býva zvyčajne omnoho zložitejšie.

Najznámejší algoritmus používajúci greedy techniku je Kruskalov algoritmus na nájdenie najlacnejšej kostry grafu.

2.3 Zložitosť a asymptotická notácia

Rast funkcií

Stupeň rastu času behu algoritmu charakterizuje účinnosť algoritmu a umožňuje navzájom porovnávať alternatívne algoritmy. Niekedy je možné pomerne presne určiť čas behu algoritmu, ale zvyčajne to nemá praktický význam. Pre dostatočne veľké vstupy je možné zanedbať multiplikatívne konštanty ako aj výrazy nižšieho rádu. Preto sa zaoberáme *asymptotickou* účinnosťou algoritmov. Algoritmus, ktorý je asymptoticky efektívnejší, bude najrýchlejší pre všetky dostatočne veľké vstupy.

3 Vyhľadávanie

3.1 Problém vyhľadávania

Zisťovanie všetkých objektov v dátovom súbore, spĺňajúcich určitú podmienku – kritérium vyhľadávania. Vyhľadávanie v zozname prvkov je jeden zo základných vyhľadávacích algoritmov. Cieľ je nájsť umiestnenie prvku množiny podľa určitého kľúča.

Algoritmy

Najjednoduchší vyhľadávací algoritmus je *sekvenčné vyhľadávanie*, ktoré postupne skontroluje každý prvok. Má časovú zložitosť ***O(n)***, kde je n počet prvkov.
Rýchlejšie je *binárne vyhľadávanie*, s časovou zložitosťou ***O(****log n****)***, predpokladá sa ale utriedený vstupný dátový súbor.
Hašovacie tabuľky sa tiež používajú na vyhľadávanie. V priemernom čase potrebujú na vyhľadanie konštantný čas, ale potrebujú veľa pamäte a v najhoršom prípade až ***O(n)*** času na vyhľadanie.
Ďalšími dátovými štruktúrami zameranými na rýchle vyhľadávanie sú samovyvažovacie *vyhľadávacie stromy*. Čas potrebný na nájdenie prvku je ***O(****log n****)*** a s určitými úpravami sa dajú zlepšiť aj časy vkladania a mazania prvkov.
Problém vyhľadávania sa dá modifikovať aj na nájdenie všetkých prvkov väčších (menších) ako hľadaný prvok. Implemetácia v sekvenčnom vyhľadávaní, binárnom vyhľadávaní a v stromoch je triviálna. Toto však nie je možné pri hašovacích tabuľkách.

Metódy

Rozlišuje sa medzi vnútornými a vonkajšími metódami vyhľadávania. Pri vnútorných metódach sa predpokladá, že celý dátový súbor je v operačnej pamäti. Ak sa prevažná časť dátového súboru počas vyhľadávania nachádza v externej pamäti, hovoríme o vonkajšom vyhľadávaní. Typickými vnútornými metódami vyhľadávania sú sekvenčné vyhľadávanie, binárne vyhľadávanie a vyhľadávanie v binárnych stromoch. Vyhľadávanie v B-stromoch a hašovanie sú prevažne vonkajšie metódy.

3.2 Sekvenčné vyhľadávanie

Zaradenie a popis

Sekvenčné vyhľadávanie je vyhľadávacia technika na nájdenie danej hodnoty v neutriedenom poli prvkov. Je ľahko programovateľná a používaná najmä vtedy, ak počet prvkov medzi ktorými sa vyhľadáva, nie je príliš veľký.

Pôvodný algoritmus prináša z hľadiska časovej zložitosti nekonzistentné výsledky – môže sa stať, že hľadaný prvok bude vždy umiestený na konci poľa. Pozmenený algoritmus rieši tento problém náhodným usporiadaním vstupného poľa.

Motivácia a špecifikácia

Máme neutriedené pole A a máme nájsť, kde sa v ňom nachádza prvok s hodnotou *v.*

Vstup: postupnosť *n* čísel A=<a1,a2,...,an> a hodnota *v.*

Výstup: Index *i* taký, že v=A[i] alebo v=NIL, ak sa v nenachádza v A.

Myšlienka algoritmu

Vstupné pole sa náhodne usporiada, aby sa zabránilo opakovanému výskytu hľadaného prvku na konci poľa.

Následne sa prechádza sekvenčne poľom, kým nenájde hľadaný prvok. Ak taký prvok v poli neexistuje, algorimus po *n* krokoch skončí a do *v* zapíše špeciálnu hodnotu NIL.

Zápis algoritmu

```
var A: array [1..N]
zamiesaj(A);
i:=0;
repeat
  i:=i+1
until ((A[i]=v) and (i=N))
if a[i]<>v then v=NIL;
```

Vlastnosti algoritmu

Algoritmus má priemernú časovú zložitosť ***O(n)*** za predpokladu, že distribúcia prvkov v poli je rovnomerná. Ak sa v poli nachádza viac prvkov s hľadaným kľúčom, časová zložitosť sa znižuje.

3.3 Binárne vyhľadávanie

Zaradenie a popis

Binárne vyhľadávanie, alebo bisekcia, je vyhľadávacia technika na nájdenie danej hodnoty v utriedenom poli prvkov. V každom kroku sa interval rozdelí na dva rovnaké podintervaly a prvok sa hľadá už len v jednom z nich.

Motivácia a špecifikácia

Máme utriedené pole A a máme nájsť, kde sa v ňom nachádza prvok s hodnotou *v.*

Vstup: postupnosť *n* čísel A=<a1,a2,...,an> a hodnota *v.*

Výstup: Index *i* taký, že v=A[i] alebo v=NIL, ak sa *v* nenachádza v A.

Myšlienka algoritmu

Binárne vyhľadávanie nájde medián, urobí porovnanie a na základe tohto sa rozhodne o pokračovaní v hornej alebo dolnej časti zoznamu a rekurzívne sa pokračuje od začiatku. Toto sa vykonáva, až kým sa nenájde hľadaný prvok.

Zápis algoritmu

```
int search( key, r )
typekey key;  dataarray r;

{ int high, i, low;
for ( low =(-1), high=n;  high-low > 1;  )
     {
     i = (high+low) / 2;
     if ( key <= r[i].k )  high = i;
          else             low  = i;
     }
if ( key ==r[high].k )  return( high );
     else              return( -1 );
}
```

Vlastnosti algoritmu

Binárne vyhľadávanie je algoritmus s logaritmickou časovou zložitosťou ***O(log n)***. Presnejšie, 1 + $\log_2 n$ iterácií je potrebných na získanie výsledku. Je teda značne rýchlejšie ako lineárne vyhľadávanie, ktoré má zložitosť ***O(n)***. Dá sa vyjadriť rekurzívne aj iteratívne, avšak v mnohých programovacích jazykoch je rekurzívny zápis oveľa čitateľnejší. Binárne vyhľadávanie je príkladom algoritmu typu *rozdeľuj a panuj*.

3.4 Ďalšie algoritmy vyhľadávania

Okrem bližšie popisovaných spôsobov vyhľadávania existujú aj ich rôzne ich variácie, Sú zaujímavé, lebo sú ľahko programovateľné a zvyčajne rýchlejšie ako algoritmy, z ktorých pôvodne vychádzajú.

Jump search

Tento algoritmus je zrýchlenie sekvenčného vyhľadávania. Prechádza poľom tak, že kontroluje každý *j*-ty prvok. Takto nájde oblasť, kde sa hľadaný prvok nachádza a túto oblasť skontroluje obyčajným sekvenčným vyhľadávaním. Pre n prvkov je optimálne, keď j= √ n.

Interpolation search

Je to variácia binárneho vyhľadávania, teda vstupné pole musí byť utriedené a postupne sa určité časti poľa vynechávajú z budúceho hľadania. Pole sa prehľadáva tak, že ďalšia pozícia sa vypočíta na základe interpolácie hľadaného kľúča a hodnôt na koncoch intervalu. Jednoduchý príklad je vyhľadávanie v slovníku. Ak hľadáme slovo „algoritmus“, vieme že máme knihu otvoriť niekde na začiatku. Podľa tam nájdeného kľúča potom vieme približne určiť, koľko strán a ktorým smerom máme listovať.

```
function search( key : typekey; var r : dataarray ) : integer;
var high, j, low : integer;
begin
low := 1; high := n;
while (r[high].k >= key) and (key > r[low].k) do
     begin
     j := trunc( (key-r[low].k) / (r[high].k-r[low].k) *
                    (high-low) ) + low;
     if      key > r[j].k then low  := j+1
     else if key %lt; r[j].k then high := j-1
                           else low  := j
     end;
if r[low].k = key then search := low  {*** found(r[low]) ***}
                  else search := -1;  {*** notfound(key) ***}
end;
```

4 Binárne prehľadávacie stromy

4.1 Dátová štruktúra strom

Strom je označenie pre dôležitú dynamickú štruktúru. Stromy sa používajú v prípade hierarchických vzťahov a rekurzívnych štruktúr objektov.

Je definovaný nasledujúcim spôsobom: *Štruktúra strom* so základným typom T je buď prázdna štruktúra, alebo vrchol typu T spolu s konečným počtom pripojených disjunktných stromových štruktúr so základným typom T, ktoré nazývame *podstromy.*
Strom je vo svojej podstate súvislý, acyklický a neorientovaný graf. Keď je neorientovaný graf síce acyklický, ale môže byť rozložený do viacerých komponentov, voláme ho *les*.

Nech G = (V, E) je neorientovaný graf. Potom nasledujúce tvrdenia sú ekvivalentné (uvádzané bez dôkazu).

- G je strom.
- Ľubovoľné dva vrcholy v G sú spojené jedinou jednoduchou cestou.
- G je súvislý, ale keď sa odoberie ľubovoľná hrana z E, výsledný graf nie je spojitý.
- G je súvislý a |E| = |V|-1.
- G je acyklický a |E| = |V|-1
- G je acyklický, ale keď sa pridá do E ľubovolná hrana, výsledný graf obsahuje cyklus.

Vlastnosti

Usporiadaný strom je taký, v ktorom sú vetvy každého vrcholu usporiadané. Z toho vyplýva, že napríklad dva stromy na obrázku pod textom sú rozdielne objekty. Vrchol *y*, ktorý je bezprostredne pod vrcholom *x*, sa nazýva (priamy) nasledovník vrcholu *x*. Ak existuje hrana z vrcholu *u* do vrcholu *w*, tak potom *u* je otcom *w* a *w* je synom *u*. Ak je vrchol *x* na *i*-tej úrovni, tak *y* bude na (i+1)-tej úrovni. O vrchole *x* potom hovoríme, že

je priamy predchodca vrcholu *y*. *Koreň stromu* je na nultej úrovni. Maximálna úroveň ľubovolného prvku v strome sa volá hĺbka alebo výška.

Ak nejaký prvok nemá nasledovníkov, nazýva sa koncový prvok alebo *list* stromu. Prvok, ktorý nie je listom, nazývame *vnútorným vrcholom*. Počet (priamych) nasledovníkov vnútorného vrcholu nazývame jeho stupňom. Maximálny stupeň spomedzi všetkých vrcholov určuje stupeň stromu. Počet vetiev (hrán) alebo vrcholov, ktoré treba prejsť, aby sme sa dostali od koreňa k vrcholu *x*, sa nazýva *dĺžka cesty* vrcholu *x*. Vo všeobecnosti platí, že dĺžka cesty vrcholu na *i*-tej úrovni sa rovná hodnote *i*. Dĺžka cesty celého stromu je definovaná ako súčet dĺžok ciest jednotlivých vrcholov stromu.

Reprezentácia stromov

Keďže stromy sú rozvetvené rekurzívne štruktúry je výhodné na ich reprezentáciu použiť smerníky. Nemá však zmysel definovať premenné s pevnou stromovou štruktúrou. Vrcholy stromu sú preto definované ako premenné s pevnou štruktúrou, teda pevne stanoveného typu a kde stupeň stromu určuje počet smerníkov na podstromy daného vrcholu. Odkaz na prázdny strom sa vyjadruje konštantou *nil*.

Stromy sa dajú reprezentovať aj ako pole záznamov (*array of record*), kde každý záznam obsahuje relevantné informácie o vrchole.

4.2 Binárne stromy

Binárny strom je usporiadaný strom druhého stupňa, v ktorom každý vrchol má najviac dvoch synov. Definovaný je rekurzívne.
Binárny strom T je štruktúra definovaná na konečnej množine prvkov (vrcholov), kde každá je buď prázdna, alebo sa skladá z koreňa (vrchola) a z dvoch disjunktných stromov nazývaných ľavý a pravý podstrom koreňa.
Príkladom binárneho stromu je rodokmeň, kde matka a otec tvoria nasledovníkov (!) príslušnej osoby.

Vlastnosti

Strom je *dokonale vyvážený*, ak pre každý vrchol platí, že počet vrcholov v jeho ľavom a pravom podstrome sa líši najviac o jeden vrchol.

Pravidlo rovnomernej distribúcie známeho počtu *n* vrcholov sa dá najlepšie formulovať rekurzívne:

- zvolí sa jeden vrchol za koreň stromu,
- vytvorí sa ľavý podstrom s počtom koreňov nl = n div 2,
- vytvorí sa pravý podstrom s počtom koreňov nr = n – nl – 1.

Transformácia n-árneho stromu na binárny

Ak chceme reprezentovať obyčajný n-árny strom ako binárny, použijeme algoritmus priamej transformácie z n-árneho stromu do binárneho.

Každý vrchol *N* v pôvodnom strome zodpovedá vrcholu *N′* v binárnom strome. Ľavý syn *N′* je vrchol zodpovedajúci prvému synovi *N* a pravý syn *N′* je vrchol zodpovedajúci nasledujúcemu bratovi *N* '. To znamená vrcholu, ktorý je ďalší v poradí medzi synmi otca *N*.

Na problém sa môžeme pozerať ako na spájaný zoznam. Synovia každého uzla *N* sú v spájanom zozname spojení pomocou svojich pravých odkazov a uzol *N* má smerník na tento spájaný zoznam cez svoj ľavý odkaz.

Binárne prehľadávacie stromy

Binárny vyhľadávací strom je dátová štruktúra založená na binárnom strome. Kľúče vo vrcholoch sú usporiadané tak, že hodnota uložená v *u* je väčšia alebo rovná ako hodnota uložená v ľavom podstrome *u.* A obdobne, hodnota uložená v *u* je menšia alebo rovná ako hodnota uložená v pravom podstrome *u.* Takto je možné ľahko vyhľadávať jednotlivé kľúče binárnym vyhľadávaním.

Sú to taktiež dátové štruktúry podporujúce operácie na dynamických množinách vrátane Search, Minimum, Maximum, Predecessor, Successor, Insert a Delete. Možno ich použiť ako slovník, alebo prioritný front.

Základné operácie trvajú čas úmerný výške stromu. Na úplnom binárnom strome s *n* uzlami tieto operácie trvajú v najhoršom prípade čas $\Theta(\lg n)$. Ak je strom lineárny (každý uzol má najviac jedného potomka), tak tieto operácie trvajú v najviac čas $\Theta(n)$.

4.3 Prechádzanie stromom

Prechod stromom (*tree traversal*) je proces, pri ktorom sa systematickým spôsobom navštívi každý vrchol stromu iba raz. Je to bežný proces, ak sa má na každom prvku stromu vykonať nejaká operácia. Na túto sa dá pozerať ako na jednoduchý sekvenčný proces – jednotlivé vrcholy navštevujú v určitom poradí. Podľa toho sú rozdelené metódy prechádzania do skupín.

Rozlišujú sa tri základné usporiadania, ktoré sú prirodzeným dôsledkom stromovej štruktúry. Tak ako samotná štruktúra, sa dajú vhodne rekurzívne vyjadriť.
Popísané algoritmy sú riešené pre binárne stromy, ale dajú sa zovšeobecniť aj pre ostatné stromové štruktúry. Označme v binárnom strome koreň *K*, jeho ľavého syna *L* a pravého syna *R*.

Preorder

Preorder tree walk vypíše hodnotu koreňa pre hodnotami v oboch podstromoch, teda v poradí *K, L, R.* Prechod stromu pomocou preorderu a zároveň vkladanie hodnôt do nového stromu je častá technika na vytvorenie úplnej kópie binárneho stromu. Preorder sa tiež používa na vyhodnotenie výrazu uloženého v strome. Prechádza sa výrazom sprava doľava a ukladajú sa elementy do zásobníka.

```
preorder(node)
  print node.value
  if node.left  != null then preorder(node.left)
  if node.right != null then preorder(node.right)
```

Inorder

Inorder tree walk vypíše hodnotu koreňa medzi hodnotami v ľavom a pravom podstrome, teda v poradí *L, K, R.* Táto metóda prechodu stromom prechádza hodnoty v stúpajúcom poradí. To je preto, lebo z definície je každý prvok v ľavom podstrome menší ako v koreni a v pravom podstrome väčší ako v koreni. Podobne, prechod stromom metódou reverzného inorderu dáva hodnoty utriedené zostupne.

```
inorder(node)
  if node.left  != null then inorder(node.left)
  print node.value
  if node.right != null then inorder(node.right)
```

Postorder

Postorder tree walk vypíše najprv hodnoty v podstromoch a až potom hodnotu koreňa, teda v poradí *L, R, K*. Táto metóda sa tiež volá prehľadávanie do hĺbky. Tak ako je preorder vhodne spojený so zásobníkom, tak je postorder spojený s frontom. Hodnoty sa vypíšu v poradí závisiacom od svojej vzdialenosti od koreňa.

```
postorder(node)
  if node.left  != null then  postorder(node.left)
  if node.right != null then postorder(node.right)
  print node.value
```

Level-order

Level-order tree walk prechádza najprv všetky vrcholy na tej istej úrovni a až potom ide o úroveň nižšie. Takýto prechod sa tiež volá prehľadávanie do šírky.

Príklad uvedený v kóde používa zásobník a potrebuje nanajvýš pamäť, ktorej veľkosť sa rovná počtu vrcholov na danej úrovni (maximálne $n/2$, kde n je celkový počet vrcholov).

```
levelorder(root)
  q = empty queue
  q.enqueue(root)
  while not q.empty do
    node := q.dequeue()
    visit(node)
    if node.left != null
      q.enqueue(node.left)
    if node.right != null
      q.enqueue(node.right)
```

4.4 Dotazy na binárne prehľadávacie stromy

Najčastejšou operáciou vykonávanou na binárnom prehľadávacom strome je vyhľadávanie kľúča uloženého v strome. Okrem operácie vyhľadávania Search podporujú binárne vyhľadávacie stromy aj operácie Minimum, Maximum, Successor a Predecessor. Tieto operácie sa všetky dajú vhodne implementovať tak, že čas ich behu je ***O(h)***, kde h je výška stromu.

Vyhľadávanie

Na vyhľadanie vrcholu s daným kľúčom sa používa procedúra Tree-Search. Ak máme kľúč *k* a ukazovateľ na koreň stromu, tak procedúra vráti ukazovateľ na vrchol s kľúčom *k*. Ak taký vrchol neexistuje, vráti *nil*.

Procedúra začína hľadanie v koreni a prehľadáva strom smerom dolu. Pri každom vrchole, na ktorý narazí, porovná hodnotu v tomto vrchole s hľadaným kľúčom *k.* Ak sa rovnajú, vrchol sa našiel a prehľadávanie skončí. Ak je *k* menšie ako hodnota vo vrchole, vyhľadávanie pokračuje v ľavom podstrome *x.* Symetricky, ak je *k* väčšie ako hodnota vo vrchole, pokračuje vyhľadávanie v pravom podstrome *x.* Uzly navštívené počas rekurzie tvoria cestu od koreňa k hľadanému vrcholu. Ten, ak existuje, je v maximálnej hĺbke *h* a teda čas behu Tree-Search je ***O(h)***.

```
Tree-Search(x,k)
if (x = nil) or (k = key[x]) then
  return x
if (k < key[x])
  then return Tree-Search(left[x],k)
  else return Tree-Search(right[x],k)
```

Náhradou rekurzie za cyklus dostávame iteratívnu verziu procedúry, ktorá je na väčšine počítačov rýchlejšia.

```
Iterative-Tree-Search(x,k)
while (x=nil) and (k!=key[x]) do
  if (k < key[x])
  then x:=left[x]
  else x:=right[x]
return x
```

Minimum a maximum

Prvok s najmenšou hodnotou kľúča sa z definície nachádza v liste „najviac vľavo“. Teda sa dá nájsť sledovaním ukazovateľov na ľavých synov, až kým nenarazíme na *nil.* Vlastnosť binárnych prehľadávacích stromov priamo zaručuje korektnosť algoritmu. Ak uzol *x* nemá ľavý podstrom, potom minimum podstromu s koreňom *x* je *key[x]*. Ak však má podstrom, najmenší prvok je možné nájsť v podstrome s koreňom v uzle *left[x]*. Nájdenie maxima v strome je analogické k nájdeniu minima. Obe procedúry majú časovú zložitosť ***O(h)***, kde *h* je výška daného stromu.

```
Tree-Minimum(x)
while (left[x] != nil) do
  x:=left[x]
return x
```

Nasledovník a predchodca

Niekedy je potrebné nájsť pre daný uzol binárneho prehľadávacieho stromu jeho nasledovníka alebo predchodcu. Štruktúra binárnych prehľadávacích stromov toto umožňuje bez nutnosti porovnávať kľúče. Procedúra Tree-Successor vráti nasledovníka uzla *x* v strome, ak taký neexistuje, vráti hodnotu *nil.* Kód procedúry je rozdelený na dva prípady. Ak je pravý podstrom uzla *x* neprázdny, potom je nasledovník uzla *x* vrchol s najmenšou hodnotou kľúča v tomto (pravom) podstrome. Tento zistíme použitím procedúry Tree-Minimum(right[x]). Ak je však pravý podstrom *x* prázdny (a *x* má nasledovníka *y*), potom je *y* najnižší predok *x*, ktorého ľavý syn je tiež predok *x.* Zjednodušene povedané, od *x* ideme hore, kým nenarazíme na vrchol, ktorý je ľavým synom svojho otca.

```
Tree-Successor(x)
if (right[x] != nil) then
  return Tree-Minimum(right[x])
y:=p[x]
while (y != nil) and (x = right[y]) do
  x:=y
  y:=p[y]
return y
```

4.5 Vkladanie a vymazávanie

Vkladanie a vymazávanie vrcholov v binárnom strome sú operácie, ktoré menia túto dynamickú množinu. Preto treba zabezpečiť, aby sa aj po ich vykonaní zachovali vlastnosti binárneho prehľadávacieho stromu. Časová zložitosť oboch operácií je ***O(h),*** kde *h* je hĺbka stromu.

Vkladanie

Procedúra Tree-Insert vloží do binárneho stromu *T* novú hodnotu *v.* Procedúre je odovzdaný uzol *z,* pre ktorý platí *key[z]=v. left[z]=nil, right[z]=nil*. Táto procedúra modifikuje *T* a niektoré časti *T* tak, že je možné vložiť *z* na príslušnú pozíciu v strome.

Procedúra Tree-Insert začína v koreni a sleduje cestu nadol. Ukazovateľ *x* postupne prechádza túto cestu a zároveň je *y* stále nastavený na jeho otca. V cykle while sa ide nadol a od porovnania kľúčov *key[y]* a *key[x]* závisí výber ľavého alebo pravého syna. Keď sa algoritmus dostane na spodok stromu, bude mať *x* hodnotu *nil.* Vtedy sa jednoduchým nastavením smerníkov vloží prvok *z* do stromu.

```
Tree-Insert(T,x)
y := nil
x := root[T]
while (x!= nil) do
  y := x
  if (key[z] < key[x])
  then x := left[x]
  else x := right[x]
p[z] := y
if (y = nil)
  then root[T] := z
  else if (key[z] < key[y])
         then left[y] := z
         else right[y] := z
```

Vymazávanie

Procedúra pre vymazávanie uzla *z* z binárneho prehľadávacieho stromu má ako parameter ukazovateľ na *z*. Táto procedúra predpokladá tri prípady:

Ak *z* nemá potomkov, tak jeho rodičovi zmeníme smerík na *nil* a vymažeme uzol *v.*

Ak má *z* iba jedného potomka, tak daný uzol odstránime vytvorením spojenia medzi jeho rodičom a jeho synom.

Ak má *z* dvoch potomkov, odstránime nasledovníka (successor) *y* uzla *z*, ktorý určite nemá ľavého syna (bol by ním *z*) a nahradíme obsah z obsahom uzla *y.*

```
Tree-Delete(T,z)
if (left[z] = nil) or (right[z] = nil)
  then y := z
  else y := Tree-Successor(z)
if (left[y] != nil)
  then z := left[y]
  else x := right[y]
if (x != nil)
  then p[x] := p[y]
if (p[y] = nil)
  then root[T] := x
  else if (y = left[p[y]])
         then left[p[y]] := x
         else right[p[y]] := x
if (y != z) then
  key[z] := key[y]
  //skopíruj aj ostatné satelitné dáta
return y
```

5 Triedenie

5. 1 Definícia triedenia

Pod triedním rozumieme proces usporiadania danej množiny objektov v špecifickom poradí. Účelom triedenia je uľahčiť neskoršie vyhľadávanie prvkov triedenej množiny.

Teda usporiadanie postupnosti objektov
a1, a2, ..., an, kde n>=1
do postupnosti
ai1, ai2, ..., ain
tak, aby platila relácia usporiadania <=
aij<=aij+1 pre *j=1..n-1*.

V postupnosti sa môžu vyskytnúť aj také objekty, ktoré sú si rovné vzhľadom na reláciu usporiadania, teda pre ktoré platí: *ak = am*.
Metóda triedenia sa nazýva *stabilná*, keď nezmení relatívnu postupnosť zhodných objektov, teda ak *j<k* a *aj=ai*, a ak sa *aj* nachádza po triedení na mieste *is* a *ak* na mieste *it*, tak *is<it*. Stabilita je dôležitá, ak sú objekty predtriedené vzhľadom na inú reláciu usporiadania (napríklad podľa určitých sekundárnych kľúčov – vlastností, ktoré samotný primárny kľúč neodráža).

Vlastnosti

Rozlišujeme taktiež medzi vnútornými a vonkajšími metódami triedenia. Ak sa pri triedení nepoužíva externá pamäť, hovoríme o vnútornej metóde triedenia, alebo o triedení in-situ (na mieste). Ak prevažná časť dátového súboru zostane počas triedenia v externej pamäti, hovoríme o vonkajšom triedení. Typickými vnútornými metódami sú bublinkové triedenie, bucketsort, quicksort, triedenie vkladaním a triedenie haldou. Triedenie zlučovaním je vonkajšia triediaca metóda.

Časová zložitosť

Všeobecná metóda triedenia potrebuje na triedenie *n* objektov ***O(n $log_2 n$)*** porovnaní. Mieru efektívnosti určuje tiež počet potrebných *porovnaní kľúčov C* a počet *presunov prvkov M.* Tieto počty sú funkciami poču *n* prvkov, ktoré sa majú triediť.

Triedenie poľa prvkov

Pole je postupnosť prvkov rovnakého druhu. Pri triedení usporadúvame pre jednotlivé objekty ich kľúče, ktoré sú tiež všetky rovnakého druhu. Preto je pole kľúčov vhodnou abstrakciou, nad ktorou môžme uvažovať ľubovoľný triediaci problém.

5.2 Priame metódy

Pretože dobré triediace algoritmy vyžadujú rádovo *n.logn* porovnaní, je vhodné rozobrať najprv niekoľko jednoduchých a zrejmých metód nazývaných *priame metódy*, z ktorých všetky vyžadujú rádovo n^2 porovnaní kľúčov. Dôvodov je viacero:

- priame metódy sú osobitne vhodné na objasnenie charakteristických čŕt hlavných zásad triedenia,
- ich algoritmy sú ľahko pochopiteľné a krátke,
- hoci dômyselné metódy vyžadujú menej operácií, sú tieto operácie zvyčajne zložitejšie v detailoch; teda priame metódy sú rýchlejšie pre dostatočne malé *n*, nepoužívajú sa však pre veľké *n.*

Metódy triedenia, ktoré triedia prvky na mieste, možno rozdeliť na tri základné kategórie podľa toho, z ktorej metódy vychádzajú (v zátvorke je príklad konkrétnej implementácie):

- triedenie výmenou (Bubble Sort),
- triedenie výberom (Selection Sort),
- triedenie vkladním (Insertion Sort, Shell Sort).

5.2.1 Bubble Sort

Zaradenie a popis

Bubble Sort je triediaci algoritmus, ktorý triedi prvky priamou výmenou. Je implementačne jednoduchý, ale neefektívny, preto sa používa takmer výhradne iba na výučbu. Pohyb prvkov pri triedení pripomína stúpanie bubliniek, preto sa toto triedenie nazýva bublinkové triedenie.

Myšlienka algoritmu

Postupne sa v cykle prechádza poľom, porovnávajú sa dva susedné prvky a ak nie sú v správnom poradí, vymenia sa. Táto procedúra sa opakuje, až kým nie sú potrebné žiadne výmeny.

Zápis algoritmu

```
procedure BubbleSort(var A: array of Integer);
var
  I, J, T: Integer;
begin
  for I := High(A) downto Low(A) do
   for J := Low(A) to High(A) - 1 do
     if A[J] > A[J + 1] then
     begin
       T := A[J];
       A[J] := A[J + 1];
       A[J + 1] := T;
     end;
end;
```

Vlastnosti algoritmu

Časová zložitosť algoritmu je v priemernom prípade je $O(n^2)$. V najhoršom prípade je časová zložitosť tiež $O(n^2)$. Bubble Sort triedi porovnávaním, pracuje *in situ* (na mieste) a je stabilný – prvky s rovnakou hodnotou sú po utriedení v tom istom poradí ako na začiatku.

5.2.2 Selection Sort

Zaradenie a popis

Selection Sort je triediaci algoritmus, ktorý realizuje triedenie priamym výberom.
Vychádza z predpokladu, že najmenší prvok môžeme zaradiť priamo na začiatok triedeného poľa, najmenší prvok zo zvyšku poľa zase na jeho začiatok atď. Podľa toho, či triedi prvky vzostupne/zostupne, sa môže označovať ako MinSort/MaxSort.

Myšlienka algoritmu

V prvom kroku sa vyberie najmenší prvok postupnosti a preradí sa na prvé miesto. V ďalšom kroku sa vyberie najmenší prvok zostávajúcej postupnosti *n-1* prvkov a preradí sa na druhé miesto. Algoritmus takto pokračuje, až po *n* krokoch je postupnosť utriedená.

Zápis algoritmu

```
procedure SelectionSort(var A: array of Integer);
var
  I, J, T: Integer;
begin
  for I := Low(A) to High(A) - 1 do
   for J := High(A) downto I + 1 do
     if A[I] > A[J] then
     begin
       T := A[I];
       A[I] := A[J];
       A[J] := T;
     end;
end;
```

Vlastnosti algoritmu

Časová zložitosť algoritmu je v priemernom prípade je $O(n^2)$. V najhoršom prípade je časová zložitosť tiež $O(n^2)$. Algoritmus pracuje na mieste a je stabilný. Selection Sort je výnimočný tým, že počet vykonaných operácií nie je nijako ovplyvnený počiatočným usporiadaním prvkov. Taktiež vykonáva optimálnych *n* výmien a teda iba $O(n)$ zápisov na disk. Ak je teda zápis na disk časovo nákladná operácia, môže byť Selection Sort dobrou voľbou.

5.2.3 Insertion Sort

Zaradenie a popis

Insertion Sort je triediaci algoritmus, ktorý realizuje triedenie priamym vkladaním. Vychádza z predpokladu, že do už utriedenej postupnosti sa na správne miesto vloží ďalší prvok. Ak sa miesto, na ktoré sa nový prvok vkladá, zisťuje *binárnym vyhľadávaním*, hovoríme o triedení binárnym vkladaním.

Myšlienka algoritmu

Prvky sa rozdelia na dve postupnosti: zdrojovú a cieľovú. Za predpokladu, že cieľová postupnosť dĺžky *i* je už utriedená, vsunie sa ďalší prvok zo zdrojovej postupnosti na správne miesto a vznikne utriedená cieľová postupnosť dĺžky *i+1*. Algoritmus začína s hodnotou indexu 2 (tj. prvý prvok je utriedený, keďže je jediný) a v rámci každého kroku sa jeho hodnota zväčší o jednotku. Pri hľadaní vhodného miesta v cieľovej postupnosti je účelné striedať porovnania prvkov s ich presunmi. Prvok *x* sa dostane na svoje miesto tak, že ho porovnáme s jeho ľavým susedom *y*. Ak je väčší, tak *y* posunieme o jedno miesto vpravo. Proces končí, ak je ďalší prvok väčší ako *x*, alebo sme dosiahli koniec cieľovej postupnosti.

Zápis algoritmu

```
procedure InsertionSort(var A: array of Integer);
var
  i, j, x: Integer;
begin
  for i := 2 to High(A) do begin
    x:=A[i]; a[0]:=x;
    j:=i-1;
    while x<A[j] do begin
      a[j+1]:=a[j]; j:=j-1;
    end;
    a[j+1]:=x;
end;
```

Vlastnosti algoritmu

Časová zložitosť algoritmu je v priemernom prípade je $n^2/4$. V najhoršom prípade je časová zložitosť tiež $\boldsymbol{O(n^2)}$. Algoritmus pracuje *in situ*, je stabilný a pracuje „online“ – triedi prvky tak,ako prichádzajú na vstup. Je vhodný na triedenie už čiastočne utriedených postupností, v tomto prípade je časová zložitosť $\boldsymbol{O(n + d)}$, kde *d* je počet *inverzií*.

5.2.4 Shell Sort

Zaradenie a popis

Shell Sort je triediaci algoritmus, ktorý realizuje triedenie priamym vkladaním so zmenšovaním kroku. Je to vlastne zlepšenie triedenia vkladaním a bublinkového triedenia. Táto metóda je jedna z najrýchlejších pre triedenie menších postupností (1000 a menej prvkov).

Myšlienka algoritmu

Postupne sa porovnávajú prvky vzdialené od seba *i* miest, v prípade potreby sa vymenia. Zníži sa *i* (spôsobom popísaným nižšie) a opakuje sa postup. Nakoniec je *i=1* a prebehne obyčajné triedenie vkladaním. Triediaci algoritmus nekladie žiadne špecifické požiadavky na postupnosť krokov. Ak je h1, h2, … ht postupnosť krokov, tak platí ht=1, hi+1<hi. Ukazuje sa však, že niektoré postupnosti vedú k lepšej časovej zložitosti algoritmu.

Zápis algoritmu

```
procedure ShellSort(var f:array of integer);
var
  i, j, h, v, N: integer;
begin
  N := length(f);
  h := 1;
  repeat
    h := ( 3 * h ) + 1; //výber krokov
  until h > N;
  repeat
    h := ( h div 3 );
    for i := ( h + 1 ) to N do begin
      v := f[i];
      j := i;
      while ( ( j > h ) and ( f[j-h] > v ) ) do begin
        f[j] := f[j - h];
        dec( j, h );
      end;
      f[j] := v;
    end;
  until
    h = 1;
end;
```

Vlastnosti algoritmu

Pôvodné triedenie vkladaním je efektívne, ak je pole prvkov takmer utriedené. V Shell Sort-e každý ďalší prechod profituje z predchádzajúcich triedení – vzhľadom na to, že každé triedenie s krokom *i* kombinuje výsledky predchádzajúcich triedení. Musí intuitívne platiť podmienka, že ak triedime s krokom *i* postupnosť, ktorá bola utriedená s krokom *k*, tak táto ostáva utriedená s krokom *k*. Je tiež zrejmé, že akákoľvek postupnosť krokov bude prijateľná, pretože v najhoršom prípade urobí všetku triediacu prácu posledný prechod.

Časová zložitosť algoritmu závisí od použitých krokov pri triedení.
Pri použití Shellovej postupnosti (začína na polovičnej veľkosti poľa a zakaždým sa delí 2) je to $\boldsymbol{O(n^2)}$. Hibbardova postupnosť: $(2^k - 1)$ má časovú zložitosť $\boldsymbol{O(n^{3/2})}$. Sedgewickove postupnosti: $(9(4^i) - 9(2^i) + 1)$ alebo $(4^{i+1} + 3(2^i) + 1)$ majú časovú zložitosť $\boldsymbol{O(n^{4/3})}$. Existujú aj postupnosti dosahujúce zložitosť $\boldsymbol{O(n \log^2 n)}$, ale existencia postupnosti, ktorá prináša v najhoršom prípade časovú zložitosť $\boldsymbol{O(n \log n)}$, zostáva otázna.

5.3 Rýchle metódy

Predchádzajúce algoritmy triedenia boli síce jednoduché na programovanie, triedili však vstupné postupnosti v nevyhovujúcom čase. Existujú aj rýchlejšie metódy na triedenie, ktoré tiež triedia porovnávaním prvkov. Pracujú v priemernom čase $\boldsymbol{O(n \log n)}$. Sú takisto použiteľné na triedenie postupnosti prvkov, o ktorých dopredu nič nevieme.
Popisované algoritmy sú:

- HeapSort,
- QuickSort,
- MergeSort.

Pred popisom algoritmu HeapSort popíšeme dynamickú dátovú štruktúru halda, ktorú tento algoritmus používa. Sem je umiestnený aj popis dátovej štruktúry prioritný front, pretože sa dá implementovať použitím haldy.

5.3.1 Halda (heap)

Halda je stromová dátová štruktúra, ktorá spĺňa dve podmienky:

- Lokálnu podmienku na usporiadanie (tzv. *vlastnosť haldy*), ktorá vyžaduje, aby pre každý vrchol stromu platilo, že hodnota kľúča v tomto vrchole je menšia ako hodnoty kľúčov jeho potomkov. Zápis lokálnej podmienky je: ak *B* je potomok *A*, tak potom key(*A*) ≥ key(*B*). Takéto haldy sa volajú *max heaps*. Ak otočíme podmienku a v koreňovom vrchole bude najmenší prvok, dostávame *min heap.*
- Štrukturálnu podmienku na to, ako strom vyzerá – líši sa podľa jednotlivých typov. Budeme sa zaoberať *binárnou haldou*. Tá je reprezentovaná *úplným* (až na poslednú úroveň, kde je *úplný zľava*) binárnym stromom.

Haldy sa používajú na implementáciu prioritných frontov. Ich efektívnosť je taktiež veľmi dôležitá pre mnoho algoritmov na grafoch. Najznámejšie využitie haldy je v triediacom algoritme Heapsort.

Reprezentácia haldy

Haldu budeme reprezentovať v poli *A*. Každý uzol stromu korešponduje s prvkom poľa, ktorý uchováva hodnotu tohto uzla. Pole *A* je objekt s dvoma atribútami: *length[A]* je celkový počet prvkov v poli a *heap-size[A]* je počet prvkov haldy uloženej v poli. Posledný prvok haldy v poli je teda *A[heap-size[A]]*. Koreňom stromu je prvok *A[1]*. Otcom *i*-teho uzla je prvok *Parent(i)*, ľavým synom je prvok s indexom *Left(i)* a pravým synom je prvok s indexom *Right(i).* Tieto funkcie sa počítajú nasledovne:

- *Parent(i)* = spodná celá časť z (i/2),
- *Left(i)* = 2i,
- *Right(i)* = 2i + 1.

Operácie na halde

Časté operácie na halde sú:

- *delete-max* (*delete-min*): odstráni sa najväčší prvok (a upraví podľa toho haldu),
- *increase-key* (*decrease-key*): zmení hodnotu kľúča a aj polohu vrcholu v halde,

- *insert*: vloží do haldy nový prvok,
- *merge*: spojí dve haldy do jednej, ktorá obsahuje vrcholy z oboch predchádzajúcich,
- *heapify:* udržuje vlastnosť haldy,
- *build-heap:* vytvorí z poľa prvkov haldu.

Udržiavanie haldy (heapify)

Vstupom procedúry je pole *A* a index *i* tohto poľa. Ak je procedúra vyvolaná, predpokladá sa, že binárne stromy s koreňmi *Left(i)* a *Right(i)* sú haldy. Ale prvok *A[i]* môže mať menšiu hodnotu ako jeho potomkovia, čo porušuje vlastnosť haldy. Preto Heapify premiestňuje hodnotu v *A[i]* na také miesto, aby podstrom s koreňom *i* zostal haldou. Procedúra funguje tak, že v každom kroku je určený najväčší prvok z *A[i]*, *A[Left(i)]* a *A[Right(i)]*, a jeho index je uložený do premennej *largest*. Ak je hodnota *A[i]* najväčšia, potom podstrom s koreňom v uzle *i* je halda a procedúra končí. Ak nie, najväčšiu hodnotu má jeden zo synov, a preto je *A[i]* vymenená za *A[largest],* čo spôsobí, že uzol *i* (spolu s jeho synmi) bude spĺňať vlastnosť haldy. Uzol *largest* má teraz pôvodnú hodnotu *A[i]* a podstrom s týmto koreňom môže porušovať vlastnosť haldy. Preto sa pre tento podstrom vyvolá rekurzívne procedúra Heapify. Čas behu procedúry heapify je ***O(h)***, kde *h* je výška haldy.

```
Heapify(A,i)
l := Left(i)
r := Right(i)
if (l <= heap-siZe[A]) and (A[l] > A[i])
  then largest := l
  else largest := i
if (r <= heap-size[A]) and (A[r]>A[largest])
  then largest := r
if largest != i then
  exchange (A[i],A[largest])
  Heapify(A,largest)
```

Vytváranie haldy (build-heap)

Pre vytvorenie haldy z poľa použijeme procedúru Heapify. Všetky prvky v podpoli *A[(n/2+1)..n]* sú listy a každý z nich tvorí jednoprvkovú haldu. Procedúra Heapify postupne prechádza zvyšné uzly a na každom vykonáva operáciu Heapify. Poradie

prechodu uzlami zabezpečuje, že sa Heapify vždy vykonáva na uzle, ktorého synovia sú už haldy. Z neusporiadaného poľa sme schopní vytvoriť haldu v lineárnom čase.

```
Build-Heap(A)
heap-size[A] := length[A]
for i := length[A] div 2 downto 1 do
  Heapify(A,i)
```

5.3.2 Prioritný front

Prioritný front je dátová štruktúra pre uchovávanie množiny S prvkov, z ktorých každý má priradenú špeciálnu hodnotu – *kľúč*.

Prioritný front podporuje nasledovné operácie:

- Insert(S,x) – vloží prvok x do množiny S,
- Maximum(S) – vráti prvok S s najväčšou hodnotou kľúča,
- Extract-Max(S) – vráti prvok S s najväčšou hodnotou kľúča a odstráni ho z množiny S.

Prioritný front sa používa napríklad na plánovanie činností na zdieľanom počítači. Prioritný front udržiava prehľad o činnostiach, ktoré sa majú vykonať a o ich prioritách. Ak sa nejaká činnosť skončí, alebo preruší, vyberie sa ďalšia činnosť s najvyššou prioritou pomocou Extract-Max. Nová činnosť sa vkladá použitím Insert.

Implementácia

Na implementáciu prioritného frontu je možné použiť haldu. Tá umožnuje, že na množine *n* prvkov bude časová zložitosť operácií najviac ***O(lg n)***. Operácia Heap-Maximum vráti najväčší prvok v konštantnom čase – vráti hodnotu A[1] z haldy.

Procedúra Heap-Extract-Max je podobná telu cyklu for procedúry Heapsort. Čas behu Heap-Extract-Max je ***O(lg n)***, keďže vykonáva len o konštantné množstvo operácií viac ako Heapify. Procedúra Heap-Insert vkladá do haldy *A* nový uzol. Rozšíri haldu pridaním nového listu do stromu a prechádza od tohto miesta smerom ku koreňu, kým nenájde správne miesto na vloženie. Čas behu tejto procedúry je ***O(lg n)***.

```
Heap-Extract-Max
if (heap-size[A] < 1) then
  error "heap underflow"
max := A[1]
A[1] := A[heap-size[A]]-1
Heapify(A,1)
return max

Heap-Insert(A,key)
heap-size[A] := heap-size[A]+1
i := heap-size[A]
while (i > 1) and (A[Parent(i) < key]) do
  A[i] := A[Parent(i)]
  i := Parent(i)
A[i] := key
```

5.3.3 Heapsort

Zaradenie a popis

Heapsort (*triedenie haldou*) je triediaci algoritmus, ktorý triedi prvky postupnosti použitím špeciálnej dynamickej dátovej štruktúry halda. Je to jeden z najlepších všeobecných algoritmov triedenia založených na porovnávaní prvkov a je efektívnou verziou Selection Sortu. Hoci je v priemernom prípade o niečo pomalší ako quicksort, jeho najhoršia časová zložitosť je ***O(n log n)***.

Myšlienka algoritmu

Triedenie prebieha v dvoch fázach: vytvorenie haldy a výpis haldy v utriedenom poradí. Algoritmus teda najprv vytvorí zo vstupného poľa haldu použitím procedúry *Build-heap*. Najväčší prvok v halde je uložený na prvej pozícii poľa. Presunieme ho teda na jeho správne miesto (na koniec poľa – výmenou za prvok, ktorý sa tam nachádza). Teraz máme haldu menšiu o jeden prvok a v koreni je prvok, ktorý porušuje vlastnosť haldy. Zavolá sa procedúra Heapify, ktorá obnoví vlastnosť haldy a zas je na prvej pozícii poľa najväčší prvok. Toto sa opakuje až po haldu veľkosti 2.

Zápis algoritmu

```
Heapsort(A)
Build-Heap(A)
for i := length[A] downto 2 do
  exchange (A[1],A[i])
  heap-size[A] := heap-size[A]-1
  Heapify(A,1)
```

Vlastnosti algoritmu

Čas behu algoritmu je ***O(n log n),*** keďže vytvorenie haldy trvá ***O(n)*** a každé z *n-1* volaní udržiavania haldy (heapify) trvá ***O(n log n)***. Paradoxne dostávame najlepší čas behu algoritmu práve vtedy, keď je vstupná postupnosť už čiastočne utriedená v opačnom poradí. Vtedy totiž fáza vytvárania haldy nevyžaduje žiadne presuny,
Heapsort nie je stabilné triedenie. Má len konštantné nároky na pamäť, teda triedi prvky *in-situ*.

5.3.4 Quicksort

Zaradenie a popis

Quicksort alebo triedenie rozdeľovaním je jeden z najrýchlejších známych triediacich algoritmov, založených na porovnávaní prvkov. Jeho priemerná doba výpočtu je ***O(n log n)*** a je najlepšia zo všetkých podobných algoritmov a jeho programovanie je jednoduché. Nevýhodou je, že pri nevhodnom usporiadaní vstupných dát môže byť časová a pamäťová náročnosť tohto algoritmu až ***O(n^2)***.
Napriek tomu, že Quicksort nemá zaručenú časovú zložitosť ***O(n log n)***, reálne aplikácie a testy ukazujú, že na pseudonáhodných vstupných údajoch je najrýchlejší zo všetkých všeobecných triediacich algoritmov. To znamená rýchlejší ako Heapsort a Mergesort, ktoré sú formálne rýchlejšie. Maximálna časová náročnosť ***O(n^2)*** však nedovoľuje jeho použitie v kritických aplikáciách.

Myšlienka algoritmu

Quicksort je algoritmus používajúci techniku divide and conquer. Rozdeľuje pole nasledovným spôsobom: vyberie sa jeden prvok nazývaný *pivot* a presunú sa všetky prvky menšie ako pivot naľavo od neho a všetky prvky väčšie ako pivot napravo. Potom tento proces rekurzívne spustíme na pravej a ľavej časti poľa. Algoritmus končí vnáranie vtedy, keď dostane utriediť podpole veľkosti 2 (prípadne 1) a začína sa vynárať. Keď sa vynoria všetky vnorenia rekurzie, je pole utriedené.

Proces výmeny prvkov je možné robiť efektívne v lineárnom čase a na mieste. Keď je vybraný medián *x*, prehľadávajú sa prvky zľava, pokiaľ sa nenájde prvok $a>x$, potom sa

prehľadávajú prvky sprava, pokiaľ sa nenájde prvok $b<x$. Tieto prvky sa vymenia a pokračuje proces prehľadávania a výmeny, pokiaľ sa pravé a ľavé prehľadávania nestretnú.

Zápis algoritmu

```
procedure quicksort (l,r : integer);
var i,j,x,w : integer;
begin
  i:=l; j:=r;
  x:=akt[(l+r) div 2];
  repeat
    while akt[i] &lt; x do i:=i+1;
    while x &lt; akt[j] do j:=j-1;
    if i &lt;= j then begin
      w:=akt[i];
      akt[i]:= akt[j];
      akt[j]:=w;
      i:=i+1; j:=j-1
    end
  until i &gt; j;
  if l &lt; j then quicksort(l,j);
  if i &lt; r then quicksort(i,r)
end;
```

Vlastnosti algoritmu

Čas behu algoritmu je v priemernom prípade ***O(n log n)***. Dobrá časová zložitosť algoritmu spočíva v správnom výbere pivota. V ideálnom prípade je to medián (alebo hodnota blízka mediánu). Ak je však pivot zakaždým minimum alebo maximum daného podpoľa, časová zložitosť narastá na $O(n^2)$.

Preto je vhodné použiť na výber pivota jednu z nasledujúcich techník:

- *Náhodný prvok* – často používaná metóda. Ak ide naozaj o náhodný výber, čas triedenia je ***O(n log n)***.
- *Metóda mediánu z troch čísel* (prípadne z ľubovoľného konštantného počtu) – vyberú sa z množiny tri prvky, z ktorých sa nájde medián a ten sa zvolí za pivota.

Qucksort nie je stabilné triedenie a triedi prvky *in-situ.*

Mergesort

Zaradenie a popis

Mergesort alebo triedenie zlučovaním je triediaci algoritmus, ktorý triedi v čase ***O(n log n)***. Je stabilný, ale netriedi na mieste. Je to algoritmus využívajúci techniku *divide and conquer*. Charakteristickou vlastnosťou je zlučovanie už utriedených podpostupností (*merging*). Používa sa napríklad pri triedení *online,* kde sa postupne dodávajú položky, ktoré majú byť utriedené (namiesto prístupu k celému poľu).

Myšlienka algoritmu

Mergesort funguje principiálne v 3 krokoch:

1. Rozdelenie neutriedenej postupnosti na dve približne rovnaké časti.
2. Zoradenie každej časti zvlášť (rekurzívne rozdeľovanie každej časti, až po postupnosť dĺžky 1, ktorá je utriedená. Vtedy sa začína vynárať z rekurzie.
3. Zlúčenie oboch častí.

Mergesort zlepšuje túto rutinu dvoma myšlienkami:

- Na utriedenie malej postupnosti je potrebných menej krokov ako na utriedenie veľkej.
- Je potrebných menej krokov na vytvorenie utriedenej postupnosti z dvoch utriedených podpostupností, ako z dvoch neutriedených podpostupností. Napríklad je nutné prejsť každou podpostupnosťou len raz, ak sú už utriedené.

Zápis algoritmu

```
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
    end if
```

Vlastnosti algoritmu

Asymptotická zložitosť pre priemerný aj najhorší prípad je $\boldsymbol{O}(n \log_2 n)$.

Veľkou nevýhodou oproti algoritmom rovnakej asymptotickej rýchlosti (napríklad Heapsort) je, že Mergesort potrebuje pre svoju prácu navyše pole veľkosti N. Existuje síce aj modifikácia Mergesortu, ktorá toto pole nepotrebuje, ale jej implementácia je veľmi zložitá a pomalá. Okrem toho je Mergesort v rôznych porovnaniach pomalší ako Quicksort alebo Heapsort.

Mergesort je však stabilný triediaci algoritmus, lepšie sa paralelizuje a má vyšší výkon na sekvenčných médiách s nižšou prístupovou dobou, lebo vyžaduje len malé množstvo pamäte s priamym prístupom. V mnohých programovacích jazykoch je Mergesort implicitným triediacim algoritmom (napríklad v Jave alebo v GNU C Library).

5.4 Triedenie v lineárnom čase

Triedenie porovnávaním

Algoritmy uvedené v predchádzajúcom texte určujú poradie prvkov iba na základe vzájomného porovnávania vstupných prvkov. Preto sa tiež volajú triedenia porovnávaním. Dá sa dokázať, že každé triedenie porovnávaním musí v najhoršom prípade na utriedenie postupnosti n čísel vykonať $\Omega(n \lg n)$ porovnaní. Teda mergesort a heapsort sú asymptoticky optimálne a neexistuje žiadne triedenie porovnávaním, ktoré by bolo od nich rýchlejšie viac ako o konštantný faktor.

Triedenie v lineárnom čase

Existujú však algoritmy, ktoré majú časovú zložitosť lepšiu ako $\boldsymbol{O(n \lg n)}$, predpokladajú však, že vstupné dáta majú isté špeciálne vlastnosti. Používajú teda iné nástroje ako porovnávanie na utriedenie prvkov. Taktiež nepracujú na mieste, ale potrebujú extra pamäť. Príklady takýchto algoritmov sú Counting sort, Bucket sort a Radix sort. Counting sort and Radix sort predpokladajú, že vstup sa skladá z celých čísel nejakého malého rozsahu. Bucket sort predpokladá, že vstup je generovaný náhodným procesom a distibuuje jednotlivé prvky náhodne po celom intervale.

Napriek lineárnej časovej zložitosti nie sú tieto algoritmy použiteľné v praxi z nasledujúcich dôvodov:

- Efektívnosť týchto triediacich algoritmov závisí na náhodnom usporiadaní prvkov. Ak táto podmienka nie je splnená, výsledkom je ich znížený výkon.
- Pre beh algoritmov je potrebná pamäť navyše – až do veľkosti usporadúvaného poľa. Ak je triedené pole veľmi veľké, predstavuje to problém.
- Vnútorné cykly týchto triedení obsahujú veľa inštrukcií, takže hoci sú lineárne, často nie sú rýchlejšie ako Quicksort.

5.4.1 Counting Sort

Zaradenie a popis

Counting sort je triediaci algoritmus, ktorý pracuje v čase ***O(n)***. Nie je to teda triedenie porovnávaním. Využíva to, že vopred sa vie rozsah čísel v poli *A*, ktoré sa má triediť. Používa pomocné pole *C,* ktoré je veľké podľa tohto rozsahu (to znamená maximálna hodnota mínus minimálna hodnota plus jedna). Týmto sa algoritmus stáva nepraktickým pre veľké rozsahy čísel kvôli narastajúcej časovej a pamäťovej zložitosti. Counting sort je vhodný napríklad na triedenie čísel z rozsahu od 0 do 100, ale nie je vhodný na abecedné utriedenie menného zoznamu. Môže však byť vhodne použitý v algoritme triedenia Radix sort, keď je tento rozsah príliš veľký na triedenie priamo Counting sortom.

Myšlienka algoritmu

Vytvorí sa pole *C*, ktorého veľkosť je od minima po maximum triedeného poľa *A*. V prípade, že je možná len pevná indexácia (napríklad so začiatkom len v 0), musia byť hodnoty z *A* mapované na *C*. Ak ide len o funkciu, kde sa od každého prvku odčíta minimum poľa *A,* aby sa získal zodpovedajúci index v poli *C,* triedenie sa volá Counting sort (ak sa prvky mapujú zložitejšou funkciou, algoritmus triedenia je Bucket sort). Ak priamo nepoznáme maximum a minimum, je potrebné tieto hodnoty zistiť jedným prechodom poľa *A*.

Každý index *i* v poli *C* je nasledovne použitý na spočítanie, koľko prvkov poľa *A* má hodnotu menšiu ako *i*. Počty uložené v *C* sú potom použité na vloženie prvkov do *A* na svoje správne pozície, čím dostaneme utriedené pole.

Zápis algoritmu

```
countingsort(A[], B[], k)
  for i = 1 to k do
     C[i] = 0
  for j = 1 to length(A) do
     C[A[j]] = C[A[j]] + 1
  for j = 1 to k do
     C[i] = C[i] + C[i-1]
  for j = 1 to length(A) do
     B[C[A[j]]] = A[j]
     C[A[j]] = C[A[j]] - 1
```

Vlastnosti algoritmu

Counting sort má časovú zložitosť $\Theta(n+k)$, kde *n* a *k* sú dĺžky polí *A* (vstupné pole) a *C* (pomocné spočítavacie pole). Aby bol algoritmus efektívny, nesmie byť *k* v porovnaní s *n* príliš veľké. Ak *k* je $O(n)$, tak je čas behu algoritmu $\Theta(n)$. Counting sort je stabilný triediaci algoritmus, ale netriedi prvky na mieste.

5.4.2 Radix Sort

Zaradenie a popis

Radix sort je triediaci algoritmus, ktorý pracuje v čase $O(n)$. Používa inú metódu triedenia ako triedenie porovnávaním. Utriedi vstupné hodnoty podľa jednotlivých číslic. Ak začína od poslednej (najmenej dôležitej) tak hovoríme o LSD (least significant digit) variácii Radix sortu. Podobne, ak sa najprv triedi podľa prvej číslice, je to MSD (most significant digit) Radix Sort. Po skončení je vstupné pole utriedené. Radix sort využíva na utriedenie časti dát iný algoritmus triedenia (najčastejšie Counting Sort) a od vlastností tohto algoritmu závisí, či je Radix sort stabilné triedenie.

Radix sort má uplatnenie často pri triedení záznamov, ktoré ako kľúč používajú viacero položiek. Ak chceme napríklad utriediť dátumy podľa roku, mesiaca a dňa, štandardným porovnávaním by sme najprv porovnali roky, ak by nastala zhoda,

porovnali by sme mesiace a ak by tiež nastala zhoda, porovnali by sme dni. Iný spôsob je použiť na triedenie myšlienku Radix sortu. Utriedime najprv položky podľa dňa, potom podľa mesiaca a nakoniec podľa roku.

Myšlienka algoritmu

Na prvý pohľad je triedenie Radix sortom nelogické. Utriedia sa vstupné prvky podľa poslednej číslice. Vstupná postupnosť sa teda rozdelí na desať podpostupností, kde všetky čísla končia rovnakou číslicou. Každú podpostupnosť následne utriedime podľa predposlednej číslice. Proces sa opakuje, až kým nie je vstup utriedený podľa všetkých číslic. Ak sú čísla rôzne dlhé, doplnia sa zľava nulami.

Zápis algoritmu

```
for i:=1 to d do
  utrieď stabilným triedením podľa cifry i
```

Vlastnosti algoritmu

Časová zložitosť algoritmu závisí od použitého pomocného stabilného triedenia. Najčastejšie je používaný Counting sort – ak sú čísla v rozsahu 1..*k* a *k* nie je príliš veľké. Zložitosť algoritmu sa dá vyjadriť ako $\Theta(nd+kd)$, kde *n* je veľkosť vstupu (počet čísel na utriedenie), *k* je rozsah hodnôt vstupu a *d* je počet cifier podľa ktorých sa triedi. Ak je *d* konštanta a k = ***O(n)***, tak čas behu algoritmu je ***O(n)***.

5.4.3 Bucket Sort

Zaradenie a popis

Bucket sort je triediaci algoritmus, ktorý pracuje v čase ***O(n)***. Netriedi prvky porovnávaním, ale predpokladá, že všetky prípustné čísla sú generované náhodným procesom, ktorý prvky distribuuje uniformne na intervale. Algoritmus je rozdelí pole na konečný počet podintervalov (bucketov) a následne je každý bucket usporiadaný

osobitne (použitím iného triediaceho algoritmu, alebo rekurzívnym volaním Bucket sortu). Bucket sort je zovšeobecnením triediaceho algoritmu Pigeonhole sort.

Myšlienka algoritmu

Bucket sort pracuje nasledovným spôsobom:

1. Vytvorí sa pole veľkosti rozsahu hodnôt, reprezentujúce počiatočne prázdne buckety.
2. Prechádza sa vstupom a prvky sa vkladajú prvku do zodpovedajúcich bucketov.
3. Utriedi sa každý neprázdny bucket.
4. Vložia sa prvky z neprázdnych bucketov naspäť do pôvodného poľa.

Zápis algoritmu

```
n  := length [A]
For i := 1 to n do
    Insert A[i] into list B[nA[i]]
For i := 0 to n-1 do
    Sort list B with Insertion sort
Concatenate the lists B[0], B[1], . . B[n-1] together in order.
```

Vlastnosti algoritmu

Ak chceme, aby mal Bucket sort časovú zložitosť ***O(n)***, musí každý bucket obsahovať (v ideálnom prípade) len jeden prvok. Ak toto nie je splnené, triedenie prvkov v rámci bucketu trvá ďalší čas. Preto je Bucket sort vhodný len na triedenie vstupov, pri ktorých predpokladáme vhodné rozloženie prvkov po celom intervale.

Triedenie je stabilné, ak je požitý pomocný triediaci algoritmus tiež stabilný.

6 Grafy

6.1 Definícia a vlastnosti grafov

Graf je usporiadaná dvojica $G=(V,E)$, kde:

- V je konečná neprázdna množina vrcholov,
- E je (aj prázdna) množina hrán.

Vrchol (*node*) sa dá intuitívne predstaviť ako mesto na mape a hrana (*edge*) ako cesta medzi dvoma mestami. Potom je V množina všetkých miest a E množina všetkých ciest medzi nimi.

Grafická reprezentácia

Najčastejšia grafická reprezentácia grafov je zobrazovať vrcholy ako body alebo krúžky a hrany ako čiary medzi nimi. Vrcholy a hrany môžu byť *ohodnotené.* To znamená, že majú asociovanú číselnú hodnotu. Tieto grafy sa volajú hranovo alebo vrcholovo ohodnotené grafy (*edge- and vertex-weighted graphs*). Priradená hodnota sa nazýva váha (*weight*).

Základné pojmy

Hrana sa nazýva slučka (*self-loop*), keď je tvaru (u,u). Graf sa nazýva jednoduchý (*simple*), keď neobsahuje slučky a násobné hrany (viac hrán medzi rovnakými vrcholmi). Ak graf obsahuje slučky a násobné hrany, nazýva sa multigraf (*multigraph*). Hovoríme, že hrana (u,v) susedí (*inciduje*) s vrcholom u a vrcholom v. Stupeň vrcholu je počet hrán, ktoré s ním susedia. Vrchol v susedí s vrcholom u, ak existuje hrana incidentná k obom vrcholom (tj. hrana medzi nimi). Počet vrcholov grafu značíme N. Hovoríme, že graf je riedky (*sparse*), keď počet jeho hrán M je malý v porovnaní s počtom všetkým možných hrán $N(N-1)/2$. Inak hovoríme, že graf je hustý (*dense*).

Orientované grafy

Ak je hranám priradený smer, hovoríme o orientovaných grafoch (*directed graphs*) a hrany sa potom nazývajú orientované hrany. Podobne, ak nie je hranám priradený smer,

sú to neorientované hrany a graf je tiež neorientovaný (*undirected*). Hrany v orientovaných grafoch sa zobrazujú šípkami, ktoré určujú ich smer.
Odchádzajúci stupeň vrcholu *(out-degree)* je počet orientovaných hrán, ktoré majú v danom vrchole začiatok. Prichádzajúci stupeň vrcholu (*in-degree*) je počet hrán končiacich vo vrchole.

Cesty v grafe

Cesta *(path)* z vrcholu *x* do vrcholu *y* je postupnosť vrcholov (v_0, v_1,..., v_k) taká, že $v_0=x$ a $v_0=x$ a hrany (v_0,v_1),(v_1,v_2),... patria do množiny hrán *E*. Dĺžka takejto cesty je *k.* Cesta je jednoduchá, ak obsahuje každý vrchol iba raz. Cestu nazývame cyklom, ak má začiatok a koniec v tom istom vrchole. Cyklus je jednoduchý, ak obsahuje každý vrchol iba raz, s výnimkou počiatočného.

Dosiahnuteľnosť

Vrchol *v* nazývame dosiahnuteľným z vrcholu *u,* ak existuje cesta z *u* do *v.* Neorientovaný graf nazývame spojitým alebo súvislým *(connected)*, keď existuje cesta z každého vrcholu do každého iného. Komponent grafu je maximálna množina vrcholov s vlastnosťou, že každý jej vrchol je dosiahnuteľný z každého iného vrcholu v komponente.

Špeciálne typy grafov

Neorientovaný graf nazývame stromom *(tree)*, keď je acyklický a je súvislý. Hovoríme, že strom je zakorenený (*rooted*), keď sme jasne vyznačili jeho najvyšší vrchol – koreň *(root)*. Každý vrchol v strome má práve jedného otca *(parent)* a rôzny počet synov *(children)*. Neorientovaný graf, ktorý neobsahuje cykly je les *(forest)*. Orientovaný acyklický graf sa nazýva DAG *(directed acyclic graph)*. Graf je úplný, ak obsahuje hranu medzi každou dvojicou vrcholov. Graf je bipatitný, ak môže byť rozdelený na dve množiny *V* a *W* tak, že hrany vedú len z *V* do *W* alebo z *W* do *V.*

6.2 Základné algoritmy a problémy na grafoch

Úvod

Hoci je teória grafov pomerne mladou vednou disciplínou, sú grafy veľmi dôležitou časťou informatiky. Mnoho problémov sa dá formálne vyjadriť a riešiť práve pomocou nich. Grafy sa hodia na reprezentáciu rôznych typov sietí, napríklad cestnej siete, počítačovej siete, sústavy vodovodov a podobne. Jednou z prvých prác z oblasti teórie grafov bola práca Leonharda Eulera o siedmich mostoch v Kráľovci (dnešný Kaliningrad). Zaoberal sa otázkou, či existuje taká trasa, ktorá prechádza cez každý z vtedajších siedmich mostov mesta práve raz a vracia sa do začiatočného bodu. Euler sformuloval problém ako graf a dokázal, že takáto trasa existuje iba vtedy, ak každý vrchol grafu má párny počet hrán (čo nebol prípad daného mesta).

Spracované algoritmy

Zo všetkých problémov a algoritmov teórie grafov tvoria reprezentatívnu vzorku nasledujúce algoritmy:

- Prehľadávanie do hĺbky,
- Topologické triedenie,
- Najlacnejšia kostra,
- Najkratšia cesta.

Rôzne iné algoritmy a problémy

Enumeration

Enumeration (vymenovanie) sa v teórii grafov zaoberá nasledovnou otázkou: Koľko neizomorfných grafov má danú vlastnosť?

Podgrafy a indukované podgrafy

Často sa dá zistiť niektorá vlastnosť grafu len tak, že sa skontroluje, či danú vlastnosť majú všetku jeho podgrafy. Zisťovať však maximálne podgrafy pre určitý problém je zväčša NP-úplný problém. Ďalšie problémy sú:

- Nájsť kliku – najväčší kompletný podgraf.

- Zisťovanie planarity grafu.

Farbenie grafu

Rôzne problémy týkajúce sa farbenia grafov.

- Problém štyroch farieb.
- Úplné farbenie.

Problémy hľadania cesty

- Hamiltonovská cesta.
- Najlacnejšia kostra grafu.
- Problém čínskeho poštára.
- Sedem mostov mesta Kráľovec.
- Problém najkratšej cesty.
- Problém obchodného cestujúceho.

Toky v sieťach

- Úloha o maximálnom toku.

Problémy pokrytia

- Vrcholové pokrytie grafu.

6.2.1 Prehľadávanie do hĺbky

Zaradenie a popis

Prehľadávanie do hĺbky (*Depth-first search - DFS*) je algoritmus, pomocou ktorého je možné efektívne preskúmať každý vrchol a každú hranu grafu systematickým spôsobom. Algoritmus začne vo vrchole a preskúmava graf čo najďalej (najhlbšie) a až potom pokračuje v backtrackingu.

DFS sa používa napríklad pri riešení nasledovných problémov:

- Nájdenie komponentov v grafe.
- Topologické triedenie.
- Hľadanie silne súvislých komponentov.

Myšlienka algoritmu

Prehľadávanie do hĺbky je neinformované prehľadávanie, ktoré pracuje nasledovným spôsobom: algoritmus začne v počiatočnom vrchole a rekurzívne sa zavolá na prvom susednom vrchole. Pamätá si, ktorými vrcholmi už prechádzal a do tých vrcholov viac nevstupuje. Takto pokračuje, až kým nenarazí na cieľový vrchol alebo už nemá kam pokračovať (tj. vrchol nemá žiadnych susedov, ktorých algoritmus ešte nenavštívil).

Ak graf obsahuje viac komponentov, skontroluje sa, či boli navštívené všetky vrcholy a ak nie, vyberie sa jeden z nich a spustí sa z neho prehľadávanie (takto vieme zistiť počet a konfiguráciu komponentov v grafe). V nerekurzívnej verzii sa všetky nové prehľadávané vrcholy ukladajú do zásobníka.

Zápis algoritmu

```
int graph[N][N];
int color[N];

/*prehladaj z vrcholu v */
void visit(int v)
{
  int i;
  /* zaciatok */
  color[v]=GRAY;
  for(i=0;i<n;i++)
    if(graph[v][i]!=0)
     if (color[i]==WHITE)
       visit(i);
  /* koniec */
 color[v]=BLACK;
}

/* prehladavanie do hlbky */
void dfs(void)
{
  int i;
  /* kazdy vrchol zafarbime na bielo */
  for(i=0;i<n;i++)
    color[i]=WHITE;
  /* zacni z neofarbeneho vrcholu */
  for(i=0;i<n;i++)
    if (color[i]==WHITE)
      visit(i);
}
```

Vlastnosti algoritmu

Keďže prehľadávanie do hĺbky perspektívne skontroluje každý vrchol a každú hranu v grafe, jeho časová zložitosť je $O(|V| + |E|)$, kde V je množina vrcholov a E množina hrán. Pamäťová zložitosť je oveľa nižšia ako pri prehľadávaní do šírky.

6.2.2 Topologické triedenie

Topologické triedenie

Základnou operáciou na orientovaných acyklických grafoch (dag-och) je spracovanie vrcholov grafu v takom poradí, že žiaden vrchol nie je spracovaný skôr ako vrchol, ktorý na neho ukazuje. Topologické usporiadanie vrcholov grafu *G* je také lineárne usporiadanie všetkých jeho vrcholov, že ak *G* obsahuje hranu *(u,v),* potom *u* sa nachádza v tomto usporiadaní pred vrcholom *v.* Taktiež platí, že ak graf *G* obsahuje cyklus (nie je dag), tak topologické usporiadanie neexistuje. Rovnako to platí aj opačne. V praxi sa topologické triedenie používa na plánovanie jednotlivých častí projektu, na vhodné poradie vykonávania jednotlivých podprocesov hlavného procesu a podobne.

Reverzné topologické triedenie

Často potrebujeme interpretovať hrany v grafe naopak, teda hrana (x,y) znamená, že vrchol *x* závisí na vrchole *y.* Napriklad pri poradí definícií sa stáva, že jedna definícia sa odvoláva na predchádzajúcu. Takéto usporiadanie sa nazýva reverzné topologické usporiadanie a je ekvivalentné topologickému triedeniu grafu s otočenou orientáciou hrán. Reverzné topologické usporiadanie sa dá jednoducho získať prehľadávaním do hĺbky. Vždy, keď prehľadávanie v rekurzii odchádza z vrcholu, vypíše sa číslo vrcholu a nakoniec sú čísla vrcholov vypísané v reverznom topologickom usporiadaní.

Zápis algoritmu

```
int n,g[100][100]; /* graf */
int saw[100]; /* navštívili sme */
int nto,torder[100];
```

```
void dts_rts(int v)
{
  int i;
  saw[v]=1;
  /* prehladame dalsie vrcholy */
  for(i=0;i<n;i++)
    if (g[v][i])
      if(saw[i]==0)
        dfs_rts(i);
  /* pridaj do reverzneho usporiadania */
  torder[nto++]=v;
}

void tsort(void)
{
  int i;
  nto=0;
  for(i=0;i<n;i++)
    if (saw[i]==0)
      dfs_rts(i);
  /* hladame topologicke usp. */
  for(i=n-1;i>=0;i--)
    printf("%d\n",torder[i]);
}
```

6.2.3 Najlacnejšia kostra

Kostra grafu

V teórii grafov je *kostra grafu* množina všetkých vrcholov a určitá podmnožina hrán pôvodného grafu. Podmnožina hrán musí spĺňať podmienku, že ak v pôvodnom grafe existovala z vrcholu *v* do vrcholu *u* cesta, tak existuje aj v kostre. V neorientovanom grafe G je kostra podgraf G - strom T taký, že existuje cesta medzi ľubovoľnými dvoma vrcholmi grafu G len po hranách stromu T. *Najlacnejšia kostra* je taká, ktorá ma spomedzi všetkých kostier G minimálny súčet ohodnotení.

Základná vlastnosť

Kostry v neorientovaných grafoch majú vlastnosť, ktorá umožňuje vytvoriť algoritmus konštruujúci pre daný graf G najlacnejšiu kostru. Táto vlastnosť znie nasledovne (uvádzané bez dôkazu):

Pre ľubovoľné rozdelenie vrcholov grafu G do dvoch disjunktných množín *V* a *W*, obsahuje minimálna kostra grafu najkratšiu z hrán, ktoré vedú medzi *V* a *W*.

Algoritmy

Kruskalov algoritmus

Algoritmus je založený na uvedenej vlastnosti kostier. Pred hľadaním si usporiadame hrany podľa ich ceny. Potom ich postupne vyberáme a skúšame ich pridať ku kostre ***T*** (začína sa s prázdnou kostrou). Ak vznikne pridaním hrany cyklus, hranu vynecháme a pokračujeme ďalšou.

Primov algoritmus

Tento algoritmus postupne pridáva do kostry vrcholy. Začneme (s prázdnou kostrou) v ľubovoľnom vrchole grafu. V jednom kroku algoritmu priberieme vrchol *v* taký, že hrana *e*=(*u,v*) je najlacnejšia zo všetkých hrán, pre ktoré platí, že *u* je z ***T*** a *v* je z ***G–T.***

6.2.4 Najkratšia cesta

Najkratšia cesta

V teórii grafov je problém najkratšej cesty hľadanie takej cesty medzi dvoma vrcholmi, že súčet ohodnotení hrán na tejto ceste je najmenší zo všetkých možných ciest medzi týmito vrcholmi. Je to štandardný problém napríklad pri hľadaní čo najkratšej cesty medzi dvoma mestami na mape.

Typy

Problém najkratšej cesty má dve zovšeobecnenia.

Single-source shortest path problem

Toto zovšobecnenie hľadá všetky čo najkratšie cesty z jedného daného vrcholu do všetkých ostatných vrcholov v grafe. Príkladom takého algoritmu je Dijkstrov algoritmus.

All-pairs shortest path problem

V tomto zovšeobecnení hľadáme najkratšie cesty medzi všetkými dvojicami vrcholov v grafe. Algoritmus Floyd-Warshall je známym príkladom ako riešiť tento problém.

Použitie

Ak by sme reprezentovali nedeterministický abstraktný stroj ako graf, kde vrcholy popisujú stavy a hrany popisujú možné prechody medzi nimi, môžeme použiť algoritmus hľadania najkratšej cesty na nájdenie optimálnej postupnosti prechodov na dosiahnutie daného cieľa. Napríklad by vrcholy predstavovali všetky možné stavy Rubikovej kocky a každá orientovaná hrana by zodpovedala jednému pohybu na kocke. Potom nájdenie najkratšej cesty by znamenalo nájsť riešenie, ktoré používa najmenší možný počet pohybov na kocke.

7 Záver

V súlade s hlavným cieľom sme v predloženej záverečnej práci zostavili prehľadne koncipovanú internetovú stránku *Zbierka základných algoritmov a dátových štruktúr.* Našou snahou bolo, aby táto zbierka slúžila ako zbierka a zároveň učebnica algoritmov a dátových štruktúr s dôrazom kladeným na ich vizualizáciu. Sledujúc vedľajší cieľ sme zároveň vytvorili dokument popisujúci proces výberu a spracovania tém zaradených do tejto zbierky a spracovali učivo podľa týchto zásad.

Výsledkom práce je platforma, ktorá predstavuje prechod na novú úroveň výučby algoritmov. Predovšetkým pomocou vizualizácie (animáciou alebo statickým obrázkom) popisuje charakteristické črty základných algoritmov a dátových štruktúr. Na ilustráciu využitia zbierky ako vhodného doplnku k výučbe bolo vypracovaných päť bazálnych kapitol z oblasti algoritmov a dátových štruktúr.

Práca je svojím zameraním a spôsobom spracovania danej problematiky na Slovensku ojedinelá. Je preto opodstatnené snažiť sa o postupné rozširovanie zbierky o ďalšie algoritmy a dátové štruktúry. Perspektívnou možnosťou úpravy existujúcej internetovej stránky je pridávanie nových algoritmov, náhrada existujúcich vizualizácií kvalitnejšími a výrazné rozšírenie časti venovanej experimentovaniu.

Práca vznikla so zámerom výrazného uľahčenia a skvalitnenia šttúdia informatiky. To je podmienené nielen pravidelnou aktualizáciou stránky, ale najmä jej častým používaním študentmi a pedagógmi. Veríme, že sa práci podarí naplniť jej posledný cieľ – dostať sa do povedomia akademickej obce ako užitočná pomôcka pri výučbe algoritmov a dátových štruktúr.

8 Zoznam použitej literatúry

[1] CLAUS, V. – SCHWILL, A.: *Lexikón informatiky.* 1. vyd. Bratislava. 1991. 544 s. ISBN 80-08-00755-9

[2] KATUŠČÁK, D.: *Ako písať vysokoškolské a kvalifikačné práce.* 2. vyd. Bratislava. 1998. 119 s. ISBN 80-85697-82-3

[3] WIRTH, N.: *Algoritmy a štruktúry údajov.* 2. vyd. Bratislava. 1989. 488 s. ISBN 80-05-00153-3

[4] CORMEN, T. H. – LEISERSON, C. E. – RIVEST, R. L. – STEIN, C.: *Introduction to algorithms.* 2. vyd. Cambridge. 2003. 1180 s. ISBN 0-262-03293-7

[5] PÍSEK, S.: *Začíname programovat v Delphi.* 1. vyd. Praha. 2000. 304 s. ISBN 80-247-9008-4

[6] WRÓBLEWSKI, P.: *Algoritmy: Datové struktury a programovací techniky.* 1. vyd. Brno. 2004. 351 s. ISBN 80-251-0343-9

[7] VÝBOH, M.: *Vizualizácia algoritmov.* 1. vyd. Bratislava. 2003. 59 s.

[8] ŠTEFKOVIČ, A.: *Vizualizácia algoritmov.* 1. vyd. Bratislava. 1989. 31.

[9] WILLIAMS, P. 2007. *Data Structures.* http://www.informatics.susx.ac.uk/courses/dats/notes/html/index.html (2007-06-01)

[10] BIN MUHAMMAD, R.: *Design and Analysis of Algorithms.* http://www.personal.kent.edu/%7Ermuhamma/Algorithms/algorithm.html (2007-05-20)

[11] MORRIS, J.: *Data Structures and Algorithms*. http://www.cs.auckland.ac.nz/software/AlgAnim/ds_ToC.html (2007-05-20)

[12] GOTTLIEB, A.: *Basic Algorithms.* http://cs.nyu.edu/courses/fall02/V22.0310-002/class-notes.html (2007-05-20)

[13] SIEGEL, A: *Basic Algorithms.* http://www.cs.nyu.edu/courses/spring07/V22.0310-002/index.html (2007-06-01)

[14] MEHLHORN, K.: *Data Structure and Algorithms.* http://cg.scs.carleton.ca/~morin/misc/sortalg/ (2007-06-01)

[15] MORIN, P.: *Sorting Algorithms.* http://cg.scs.carleton.ca/~morin/misc/sortalg/ (2007-06-01)

[16] SKIENA, S.: *Data Structures.* http://www.cs.sunysb.edu/~skiena/214/lectures/index.html (2007-05-22)

[17] PROCHÁZKA, J.: *Algoritmy a dátové štruktúry.* http://www.dcs.fmph.uniba.sk/ads/data/ALGORITM.pdf (2007-06-06)

[18] MITZENMACHER, M.: *Data Structures and Algorithms.* http://www.fas.harvard.edu/~libcs124/cs124/124.html (2007-05-24)

[19] HUGHES, J.: *Data Structures and Algorithms.* http://www.cs.brown.edu/courses/csci0160/slides.html (2007-05-25)

[20] KAPUR, N.: *Computer Algorithms.* http://www.cs.caltech.edu/~cs138/index.html (2007-06-03)

[21] LUEBKE, D.: *Algorithms.* http://www.cs.virginia.edu/~luebke/cs332 (2007-06-01)

[22] PODLUBNÝ, I.: *Úvod do programovania v jazyku C.* http://people.tuke.sk/igor.podlubny/C/Kap11.htm (2007-06-01)

[23] BLAHO,A.: *Programovanie v Delphi.* http://www.delphi.input.sk/index.html (2007-06-01)

[24] TOUSSAINT, G.: *Data Structures and Algorithms.* http://cgm.cs.mcgill.ca/%7Egodfried/teaching/algorithms-web.html (2007-06-01)

[25] Inspired.sk http://www.inspired.sk/ (2007-05-22)

[26] Google Directory. http://directory.google.com/Top/Computers/Algorithms/ (2007-05-22)

[27] Wikipédia – slobodná encyklopédia. http://www.wikipedia.org/ (2007-05-22)

[28] NIST. http://www.nist.gov/dads/ (2007-05-22)

[29] PRUHS, K. *Algorithms.* http://www.cs.pitt.edu/%7Ekirk/algorithmcourses/ (2007-06-01)

[30] TVAROŽEK, J.: *Úvod do teórie grafov.* http://www.ksp.sk/~zuzu/kp_teoria_2.pdf (2007-05-22)

[31] TVAROŽEK, J.: *Základné programovacie techniky.* http://www.ksp.sk/~zuzu/kp_teoria_1.pdf (2007-05-22)

A Prílohy

K práci je priložená CD príloha obsahujúca offline verziu internetovej stránky *Zbierka základných algoritmov a dátových štruktúr.* Stránka je identická s verziou stránky http://www.sprite.edi.fmph.uniba.sk/~szorad/ zo dňa 13. 6. 2006.