Programovanie - skriptá

Juraj Ďuďák

Vygenerované z webu wk.sensorical.com. Vydavatel: Sensorical Working Group
PDF generated at: Thu, 01 Sep 2016 22:22:05 CEST

Obsah

Algoritmy a programovanie **1**

Algoritmy a programovanie 1
Dátový typ 3
Štruktúry (jazyk C) 5
Rekurzia 11
Algoritmy vyhľadávania 13
Algoritmy triedenia 21
Lineárny zoznam 25
Binárny strom 35
Numerické algoritmy 37
Grafové algoritmy 38

Zbierka úloh **43**

Štruktúry (riešené príklady) 43
Rekurzia (riešené príklady) 49
Dynamická alokácia pamäti (riešené príklady) 58
Vyhľadávanie (riešené príklady) 63
Triedenie poľa komplexných čísel (riešené príklady) 72
Triedenie poľa smerníkov na komplexné čísla (riešené príklady) 78
Triedenie poľa štruktúr (riešené príklady) 86
Triedenie poľa smerníkov (riešené príklady) 92
Neusporiadaný lineárny zoznam (riešené príklady) 102
Usporiadaný lineárny zoznam (riešené príklady) 106
Binárny strom - Morseova abeceda (riešené príklady) 112
Binárny strom - Huffmanovo kódovanie (riešené príklady) 114
Numerické algoritmy (riešené príklady) 125

Referencie

Zdroje článkov a prispievatelia 126
Zdroje obrázkov, licencie a prispievatelia 127

Licencie článkov

Licencia 128

Algoritmy a programovanie

Algoritmy a programovanie

<properties> Názov=Algoritmy a programovanie Forma=Podklady k prednáškam a cvičeniam Abstrakt=Úvod do algoritmizácie. Informačné technológie. Programové prostriedky a ich využitie v praxi. Návrh komplexnejších algoritmov. Triedenie, vyhľadávanie. Riešenie numerických problémov algoritmami diskrétnej matematiky. Zreťazené zoznamy a binárne stromy. Použitie jazyka C na riešenie algoritmov. Rozvrh=2/0/2 Hodnotenie=Skúška Poznámky= </properties>

Algoritmy a programovanie

Dátový typ

Štruktúry

Rekurzia

Algoritmy vyhľadávania

Algoritmy triedenia

Lineárny zoznam

Binárny strom

Numerické algoritmy

Grafové algoritmy

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy

Preslov

Táto sekcia **Algoritmy a programovanie** má byť teoretickým podkladom a základnou literatúrou pre predmet Programovanie prenášaný na Fakulte mechatroniky (FM) Trenčianskej univerzity Alexandra Dubčeka v Trenčíne (TnUAD). Predmet je zabezpečovaný katedrou informatiky [1] FM TnUAD. Materiály prezentované v tejto sekcii sú čerpané z uvedených zdrojov a z vlastných programátorských skúseností autorov. Všetky tu publikované texty a zdrojové kódy môžu byť ďalej slobodne šírené s uvedením zdroja: Fakulta mechatroniky TnUAD.

Autor sekcie **Algoritmy a programovanie** -- Juraj Ďuďák

Vybrané kapitoly z informačných technológií

- Algoritmy a problémy
  - Stochastický algoritmus
  - Problém batoha
  - Problém 8-mich dám
  - Problém obchodného cestujúceho
  - Problém 7-mich mostov mesta Kráľovca
  - Backtracking
- Internetové protokoly
  - Referenčný model ISO/OSI
  - protokol HTTP
  - protokol FTP
  - protokol POP3
  - protokol IMAP4
  - protokol DHCP
  - protokol DNS
- Šifrovacie algoritmy
  - Algoritmus RSA
  - Pseudonáhodné čísla
  - Hašovacia funckia
  - Cyklický redundantný súčet

Referencie

- Piotr Wróblewski: Algoritmy Datové struktury a techniky programování, COMPUTER PRESS, ISBN 8025103439
- Pavel Herout: Učebnice jazyka C, Kopp, EAN 9788072323838
- Mike Banahan, Declan Brady and Mark Doran: The C Book, Addison Wesley in 1991, http://publications.gbdirect.co.uk/c_book/

Dátový typ

Algoritmy a programovanie

Dátový typ
::Zásobník
::Zoznam
::Fronta
::Množina
::Strom
::Hromada
Štruktúry
Rekurzia
Algoritmy vyhľadávania
Algoritmy triedenia
Lineárny zoznam
Binárny strom
Numerické algoritmy
Grafové algoritmy

Dátový typ je spojenie oblastí hodnôt a operácií v jeden celok.

Rozdelenie dátových typov

- abstraktné dátové typy - sú typy, ktoré kladú dôraz na vlastnosti operácií a oblasti hodnôt
- konkrétne dátové typy - sú typy použité v konkrétnom programovacom jazyku
- dátové štruktúry - sú typy, pri ktorých operácie týkajúce sa len na konštruovanie oblastí hodnôt

Abstraktné dátové typy

Abstraktný dátový typ (ADT) je v informatike výraz pre typy dát, ktoré sú nezávislé na vlastnej implementácii. Hlavným cieľom je zjednodušiť a sprehľadniť program, ktorý robí operácie s daným dátovým typom. Je výhodné popisovať algoritmy a dátové štruktúry len podľa operácií, ktoré nám umožňujú vykonávať, ako podľa detailov ich implementácie.

Keď dátovú štruktúru popisujeme podľa operácií, ktoré má byť schopná vykonávať, voláme je Abstraktný dátový typ.

Abstraktná dátová štruktúra

Abstraktná dátová štruktúra je spôsob, ako efektívne uložiť dáta tak, aby práca s nimi bola relatívne jednoduchá. Je to abstraktný sklad pre dáta definované v rámci množiny operácií a pre výpočtové zložitosti pri vykonávaní týchto operácií, bez ohľadu na implementáciu v konkrétnej datovej štruktúre.

Výber abstraktnej dátovej štruktúry je rozhodujúci pre návrh algoritmov a pre odhad ich zložitosti, zatiaľ čo výber konkrétnych dátových štruktúr je dôležitý pre účinnú implementáciu týchto algoritmov.

Vlastnosti abstraktného dátového typu

Nedôležitejšie vlastnosti abstraktného typu dát sú:

- **Všeobecnosť implementácie**: raz navrhnutý ADT môže byť zabudovaný a bez problémov používaný v akomkoľvek programe.
- **Presný popis**: prepojenie medzi implementáciou a rozhraním musí byť jednoznačné a úplné.
- **Jednoduchosť**: pri používaní sa používateľ nemusí starať o vnútornú realizáciu a správu ADT v pamäti.
- **Zapúzdrenie**: rozhranie by malo byť pojaté ako uzavretá časť. Používateľ by mal vedieť presne čo ADT robí, ale nie ako to robí.
- **Integrita**: používateľ nemôže zasahovať do vnútornej štruktúry dát. Tým sa výrazne zníži riziko nechceného zmazania alebo zmena už uložených dát.
- **Modularita**: „stavebnicový“ princíp programovania je prehľadný a umožňuje jednoduchú výmenu časti kódu. Pri hľadaní chýb môžu byť jednotlivé moduly považované za kompaktné celky. Pri zlepšovaní ADT nie je nutné zasahovať do celého programu.

Ak sú ADT implementované objektovo, je väčšina týchto predpokladov splnená.

Konkrétne dátové typy

Konkrétne dátové typy sa v programovacých jazykoch používajú na deklaráciu premenných. Vo väčšine programovacích jazykoch sa dajú použiť tieto dátové typy:

Boolean (alebo bool)

je implementáciou pravdivostného typu. Môže obsahovať dve možné hodnoty True (Pravda) a False (Nepravda), ktorým tiež prislúcha určitý rozsah číselných hodnôt. V jazyku C je hodnota false reprezentovaná hodnotou 0 a true všetkými hodnotami rôznymi od 0.

Byte (alebo char či unsigned char)

je prirodzené číslo v rozsahu 0 až 255. Tento typ sa často používa na reprezentáciu znakov tabuľky ASCII. Operácie dovolené s týmto typom sú sčítanie, odčítanie, násobenie, delenie a ďalšie ktoré sa môžu líšiť od programovacieho jazyka.

Integer (alebo int)

je typ pre reprezentáciu celých čísel má však ohraničený rozsah možných hodnôt. Tento rozsah môže byť iný v každom programovacom jazyku. V jazyku C má rozsah (16 bit) od -2147483648 do 2147483647

Reálny typ (float či double)

je typ určený pre uchovávanie desatinných čísel.

Reťazec (String alebo pole znakov)

je typ, ktorý je určený na uchovanie „reťazcov“ čiže textových hodnôt.

Dátové štruktúry

Dátové štruktúry pozostávajú zo základných dátových typov, ktoré sú usporiadané podľa určitých pravidiel. Medzi základné dátové štruktúry patria

- Pole
  - Slúži na reprezentáciu polí rôznych typov: celočíselné, znakové reťazcové pole. Ale aj pole štruktúr. Pri viacrozmerných poliach môžeme reprezentavať napr. matice (2-rozmerné polia).
- Dátový typ štruktúra
  - Slúžia na reprezentáciu štruktúrovaných údajov. Ako príklad uvedieme:
    - bod v rovine - je reprezentovaný dvoma súradnicami (reálne čísla)

  - komplexné číslo - je reprezentovaný dvoma hodnotami (reálna a komplexná časť)
  - Zásobník (pamäť LIFO) - je reprezentovaný dátovou štruktúrou pre ukladanie položiek do pamäti, veľkosťou tejto pamäti a ukazateľom aktuálnej pozície zaplnenia.
  - Fronta (pamäť FIFO) - je reprezentovaný dátovou štruktúrou pre ukladanie položiek do pamäti, veľkosťou tejto pamäti a ukazateľom začiatočnej a konečnej pozície údajov v pamäti.
  - Zoznam - Dynamická dátová štruktúra. Zoznam prvkov podobný poľu s tým rozdielom že k prvkom zoznamu sa dostaneme len cez jeho susedov.
  - Množina
  - Strom - Dynamická dátová štruktúra.
  - Hromada - Dynamická dátová štruktúra.
- Vymenovaný typ (enum)

Odkazy

- http://en.wikipedia.org/wiki/Abstract_data_type
- http://sk.wikipedia.org/wiki/D%C3%A1tov%C3%BD_typ

Štruktúry (jazyk C)

Algoritmy a programovanie

Dátový typ
Štruktúry
Rekurzia
Algoritmy vyhľadávania
Algoritmy triedenia
Lineárny zoznam
Binárny strom
Numerické algoritmy
Grafové algoritmy

Definícia štruktúr

Štruktúry majú v jazyku C význam dátových kontainerov, ktoré obsahujú pomenované položky ľubovoľného dátového typu. Štruktúry sú podobné dátovému typu záznam (record) v jazyku Pascal. Položky štruktúry sú uložené postupne v pamäti v poradí, ako boli zadefinované. Veľkosť štruktúry je totožná súčtu veľkostí všetkých jej premenných.

Príklad definície štrutúry:

```
struct datum
{
    int   den;
    char  mesiac[8];
    int rok;
};
```

Vysvetlenie:

Riadok č. 1 - definícia štruktúry s názvom "**datum**". *den* a *rok* sú celočíselné typy (riadok č. 3 a 5), *mesiac* je reťazec (riadok č. 4). Definícia štruktúry musí byť uzatvorená v zložených zátvorkách '{' a '}' a za ukončujúcou zátvorkou je bodkočiarka ';'.

Použitie štruktúry a prístup k jej položkám

```
datum d; // d je premenná typu datum
d.den=1;
strcpy(d.mesiac,"januar");
d.rok=2010;
```

Pravidlá pre prácu s položkami štruktúry

1. K položkám štruktúry pristupujeme pomocou operátora bodka ´.´
2. Pre prácu s položkami štruktúry platia rovnaké pravidlá ako s premennými danej položky
   1. V našom príklade je d.den a d.rok dátový typ int
   2. d.mesiac je dátový typ jednorozmerné pole znakov.

Inicializácia

Položky štruktúry môžeme inicializovať pri vytváraní inštancie štruktúry. Ak štruktúru neinicializujeme, jej hodnoty zostanú tiež neinicializované (budú obsahovať náhodné hodnoty). Poradie inicializácie premenných štruktúry je rovnaké ako je poradie pri definícii. Nasledujúci kód vytvorí premennú d2 typu datum a inicializuje ho na "1. januar 2010". Tento kód robí urobí to isté ako v predchádzajúcom príklade.

```
datum d2={1,"januar",2010};
```

Hodnoty pri iniciaizácii musia zodpovedať dátovým typom štruktúry. V našom prípade musí byť ako prvá hodnota celé číslo, nasleduje pole znakov a akao posledný údaj je opäť celé číslo.

Práca s položkami štruktúry

Povolené operácie so štruktúrami

Prístup k položkám

d.den

Priradenie hodnôt položkám štruktúry

d.den=3
Pozor, pravidlá pre priradenie hodnôt pre dátový typ pole!

Priradenie rovnakých štruktúr

d2=d

Nepovolené operácie so štruktúrami

Porovnenie štruktúr

Štruktúry ako celok sa nedajú použiť v relačných výrazoch. Pri porovnávaní musíme pracovať s položkami štruktúry.

Vzorový príklad

Základná práca so štruktúrami

Zadanie: Vytvorte štruktúru, ktorá bude reprezentovať maticu. Matica je charakterizovaná počtom riadkov, počtom stĺpcov a samotným dátovou štruktúrou dvojrozmerné pole. Uvažujme max. rozmery 20x20.

Riešenie 1: Pre reprezenttáciu matice si vytvoríme 2rozmerné statické pole. Ako prvé si zadefinujeme štruktúru sMatica, ktorá nám bude reprezentovať našu maticu:

```
#define MAX 20
struct sMatica
{
   int r,s;
   int M[MAX][MAX];
};
```

V progmame použijeme túto štruktúru nasledovne:

```
int main()
{
  sMatica A,B;
  A.r=3;
  A.s=2;
  for(int i=0 ; i<A.r ; i++)
    for(int j=0 ; j<A.s ; j++)
       A.M[i][j]=i+j;

  B.r=4;
  B.s=3;
  for(int i=0 ; i<B.r ; i++)
    for(int j=0 ; j<B.s ; j++)
       B.M[i][j]=i*j;
}
```

Matici A sme nastavili rozmery 3x2, matici B rozmery 4x3. Pre prístup k položkám matice A použijeme zápis A.M. Tento výraz predstavuje dvojrozmerné pole celých čísel, preto musíme ešte špecifikovať index riadku a stĺpca. Matica A bude mať hodnoty:

Matica A

0	1
1	2
2	3

Matica B bude mať hodnoty:

Matica B

0	0	0
0	1	2
0	2	4
0	3	6

Štruktúry a funkcie

Úloha: vytvorte funkciu nacitajMaticu, ktorá načíta z klávesnice maticu. Pracujte s navrhnutou štruktúrou sMatica. Podobne vytvorte funciu vypisMaticu, ktorá bude danú maticu vypisovať. *Riešenie:* Vytvoríme finkciu nacitajMaticu s jedným parametrom typu sMatica. Funkcia bude bez návratovej hodnoty, ale parameter bude predávaný pomocou odkazu. Načítanie rozmerov matice bude mimo funkciu nacitajMaticu.

```
void nacitajMaticu(sMatica &X)
{
  for(int i=0 ; i<X.r ; i++)
    for(int j=0 ; j<X.s ; j++)
       scanf("%d",&X.M[i][j]);
}

void vypisMaticu(sMatica &X) // & nie je nutný, ale použijeme ho kvôli
efektivite
{
  for(int i=0 ; i<X.r ; i++)
  {   for(int j=0 ; j<X.s ; j++)
        printf("%8d",X.M[i][j]);
      printf("\n");
  }
}
```

Použitie vo funkcii main:

```
int main()
{
  sMatica A,B;
  A.r=3;      A.s=2;
  B.r=4;      B.s=3;

  nacitajMaticu(A);
  nacitajMaticu(B);
```

```
  vypisMaticu(A);
  vypisMaticu(B);
}
```

Štruktúry a návratová hodnota funckie

Úloha: Vytvorte funkciu scitajMatice, ktorá sčíta 2 matice a vráti výsledok tohto súčtu. Funkcia nebude nič vypisovať, na predanie výsledku použite nácratovú hodnotu. *Riešenie:* Vytvoríme funckiu scitajMatice, ktorá bude mať 2 parametre: matice, ktoré chceme sčítať. Ako návratový typ funkcie zvolíme dátový typ sMatica, čo reprezentuje maticu.

```
sMatica scitajMatice(sMatica &X,sMatica &Y) // & nie je nutný, ale
použijeme ho kvôli efektivite
{
  sMatica Z;  // keďže funkcia vracia dátový typ sMatica, musíme si
vytvoriť premennú tohto typu. Z bude zároveň výsledok súčtu matíc.
            // Predpokladáme, že matice X a Y majú rovnaké rozmery.
  Z.r=X.r;    // rozmery výslednej matice budú rovnakú ako rozmery
vstupných matíc
  Z.s=X.s;

  for(int i=0 ; i<Z.r ; i++)
    for(int j=0 ; j<Z.s ; j++)
      Z.M[i][j]=X.M[i][j]+Y.M[i][j];  // súčet matíc
  return Z; // výsledná matica ako návratová hodnota.
}
```

Použitie vo funkcii main:

```
int main()
{
  sMatica A,B,C;
  A.r=2;
  A.s=3;

  B.r=4;
  B.s=3;
  nacitajMaticu(A);
  nacitajMaticu(B);

  C=scitajMatice(A,B);
  vypisMaticu(C);
}
```

Štruktúry a smerníky

V jazyku C je možné vytvárať smerník na ľubovoľný dátový typ. Môžeme teda vytvoriť smerník na náš dátový typ štruktúra. Majme definovaný dátový typ *datum*:

```
struct datum{
   int den,mesiac,rok;
};
```

Ďalej, vytvorme si premennú typu smerník na štruktúru datum (premenna d1) a alokujme pre neho v pamäti miesto:

```
  datum *d1
  d=new datum;
```

K položkám štruktúr budeme potom pristupovať nasledovne:

```
  d1->den=2;
  d1->mesiac=3;
  d1->rok=2010;
```

Vidíme, že operátor bodka (.) sa zmenil na operátor šípka (->).

Dynamicky vytvorené pole štruktúr

Úlohou je dynamicky vytvoriť pole štruktúr o veľkosti n. Opäť použijeme operátor *new*:

```
  int n=10;
  datum *d1=datum[n];
  for(int i=0;i<n;i++)
  {   d1[i].den=i+1;
      d1[i].mesiac=1; // januar
      d1[i].rok=2010;
  }
```

V tomto prípade sme vytvorili pole štruktúr, preto sme pre prístup k položkám štruktúry použili prístupový operátor bodka (.).

Teraz vytvorme pole smerníkov na štruktúru datum.

```
  int n=10;
  datum **d1=new datum*[n]; // d1 je smerník na smerník na štruktúru
datum, alebo pole smerníkov na štruktúru datum.
  for(int i=0;i<n;i++)
  {   d1[i]=new datum; // alokácia miesta pre dátovú štruktúru datum

      d1[i]->den=i+1;
      d1[i]->mesiac=1; // januar
      d1[i]->rok=2010;
  }
```

Na rozdiel od predchádzajpceho príkladu, kde sme vytvorili pole štruktúr datum, teraz sme vytvorili pole smerníkov na štruktúru datum. Pre tieto smetníky sme alokovali potrebné pamäťové miesto (preto namiesto operátora bodka (.) používame operátor šípka (->)).

Smerníky a funkcie

Funkcie vracajúce smerník na premennú

Rekurzia

UPOZORNENIE: Článok nebolo možné vykresliť - na výstup sa zapíše čistý text.
Možné príčiny problému sú: (a) chyba v softvéri pdf-writer (b) problematická MediaWiki syntax článku (c) príliš široká tabuľka

Algoritmy a programovanieDátový typŠtruktúry_jazyk_CŠtruktúryRekurziaAlgoritmy vyhľadávaniaAlgoritmy triedeniaLineárny zoznamBinárny stromNumerické algoritmyGrafové algoritmyRekurzia (po latinsky: recurrere = bežať naspäť) je matematike a informatike využitie časti vlastnej vnútornej štruktúry. V definícii funkcie sa nachádza volanie samotnej funkcie. Inak povedané, funkcia volá samú seba.Rekuzia - definície, princípyRekurzia v grafike Najjednoduchsim prikladom rekuzie, ktorý sa dá zobraziť je dať 2 zrkadlá oproti sebe (Obr. 1). Ďalším príkladom môže byť spustenie programu vzdialená pracovná plocha, kde adresa vzdialeného počítača bude ten počítač, na ktorom bola aplikácia spustená (Obr. 2). Rekurzia sa občas objaví i na obaloch potravín (Obr. 3). Obr. 1 Rekurzívny obraz v zrkadláchObr. 2 Rekurzívny obraz ´vzdialenej´ pracovnej plochyObr. 3 Rekurzívny obal kakaaRekurzia v názvoch vecí Pracujeme z názvami a skratkami, ktoré sú už také zaužívané, že ani nerozmýšľame čo znamenajú. Čo znamenajú skratky GNU, wine (program v linuxe, nie víno), PNG, LAME (mp3 enkodér), PHP, YAML, VISA .... PNG - Oficiálne "Portable Network Graphics", Neoficiálne "PNG is Not Gif"LAME - LAME Ain't an MP3 EncoderWine -Wine Is Not an EmulatorPHP - PHP: Hypertext PreprocessorYAML - YAML Ain't Markup Language VISA - Visa International Service AssociationGNU - GNU's Not UnixSlovná definícia pojmu Rekurzia pozri Rekurzia Zoznam viacerých skratiek je v anglickej verzii wikipédie http://en.wikipedia.org/wiki/Recursive_acronym. Rekurzia v matematikePrirodzené číslaPrirodzené čísla definujme nasledovne: 0 patrí do množiny \mathbb{N}ak patrí n do množiny \mathbb{N}, potom n + 1 patrí tiež do množiny \mathbb{N}Množina prirodzených čísel je taká najmenšia množina reálnych čísel, ktorá spĺňa 2 predchádzajúce kritériá. PrvočíslaPrvočísla definujme ako: Číslo 2 je najmenšie prvočíslo. Prvočíslo je každé celé kladné číslo, ktoré nie je deliteľné žiadnym iným menším prvočíslom ako je toto číslo samotné. Katalánske číslaC_0=1C_{n+1}=\frac{(4n+2)C_n}{n+2}Ackermanova funkciaAckermanova funkciaAckermanova funkcia - http://en.wikipedia.org/wiki/Ackermann_function je príklad neprimitívnej rekurzívne definovanej funkcie, ktorá veľmi rýchlo rastie. Pre kladné m, n môžeme Ackermanovu funkciu vyjadriť nasledovne: A(m,n) = \begin{cases} n+1, & ak\ m=0 \\ A(m-1,1), & ak\ m>0,\ n=0 \\ A(m-1,A(m,n-1)), & ak\ m>0,\ n>0 \end{cases} Už pre malé hodnoty m a n dosahuje veľkých hodnôt. Napríklad A(4,2) je celé číslo, ktoré má 19 729 cifier. Rekurzia v informatike Princíp fungovania rekurzívnych funkcií: Zložitý problém môžeme riešiť rekurzívnym spôsobom tak, že: Daný problém rozložíme na elementárne podproblémy, ktoré dokážeme jednoducho vyriešiť. Tieto podproblémy musia byť rovnakého typu. Riešenie spočíva v opakovanom (rep. rekurzívnom) vykonávaní funkcie riešiacej daný problém. Dôležitou bodom je určiť podmienku, kedy sa ukončí riešenie úlohy. Konečný výsledok je zlúčenie všetkých čiastkových výsledkov. Príklad:Vypíšte za sebou čísla od n do 0. Nech n=5 Zložitý´ problém vypísať dané čísla: (Dajme tomu, že nepoznám cykly) Rozloženie problému na elementárne podproblémy vypíšem len jedno číslo

(to viem) Podmienka ukončenia výpisu Prestanem keď vypíšem posledné číslo - 0 Elementárny problémFunkcia, ktorý vypíše len jedno číslo (n) void vypis(int n) { cout<<n; } Doplnenie rekurzieVo funkcii vypis pridam volanie funkcie vypis, ktorá má parameter n o 1 menší void vypis(int n) { cout<<n; vypis(--n); } Analýza riešeniaVyrobili sme deadlock, čiže uviaznutie programu. Funckia vypis sa nikdy neukončí, bude sa volať do nekonečna (resp. dotiaľ, dokiaľ bude mať program dostatok pamäti. Potom spadne). Doplnenie ukončujúcej podmienky Vypisovanie ukončím pri výpise 1. void vypis(int n) { cout<<n; if(n>0) vypis(--n); } Typy rekurzieRekurzia (blog) http://dominik.blog.matfyz.sk/p13495-rekurziaPravá rekurzianastane, ak sa za rekurzívnym volaním nachádzajú ešte nejaké príkazy alebo ak rekurzívne voláme na viacerých miestach programu. Takýto prechod na nerekurzívny algoritmus je zložitejší a často až nereálny. Chvostová rekurziainak nazývaná nepravá alebo jednoduchá. Nastane vtedy, keď rekurzívna procedúra volá samu seba ako svoj posledný príkaz. Takáto rekurzia sa veľmi ľahko dá prepísať na cyklus. Ukážky rekurzívne definovaných funkciíFaktoriálDefinícia:0!=1 n!=n*(n-1)! Pri tvorbe funkcie pre výpočet faktoriálu budeme postupovať presne podľa rekurzívnej definície: long faktorial(int n) { if(n==0) return 1; // 0!=1 return n*faktorial(n-1); // n!=n*(n-1)! } Fibonacciho postupnosťDefinícia:Fib(0)=0 Fib(1)=1 Fib(i)=Fib(i-1)+Fib(i-2), pre i>1 Rovnako aj tu budeme postupovať presne podľa rekurzívnej definície: long fibonacci(int n) { if(n<2) return n; // fib(0)=0, fib(1)=1 return fibonacci(n-1)+fibonacci(n-2); // fib(n)=fib(n-1)+fib(n-2) } Najväčší spoločný deliteľ Najväčší spoločný deliteľ Výpočet NSD - http://sputsoft.com/2009/10/computing-the-greatest-common-divisor/, NSD - definícia http://en.wikipedia.org/wiki/Greatest_common_divisor dvoch celých čísel m,n rieši Euklidov algoritmus. Jeho najjednoduchšia forma sa dá popísať nasledovne: vstupné hodnoty: m, n Ak je m=n, tak koniec (krok 6) ak je m>n, tak m=m-n ak je n>m, tak n=n-m skok na krok 2 NSD(m,n)=m (keďze m a n sú rovnaké je jedno, ktorú premennú budeme považovať za výsledok) NSD vieme definovať aj rekurzívne: NSD(m,0)=\ mNSD(m,n)=NSD(n,m\ \mbox{mod }n)V jazyku C to môžeme zapísať nasledovne: long nsd(int m, int n) { if(n==0) return m; return nsd(n,m%n); } Hanoiské vežeHlavolam / Hanojské veže Hanojské vežehttp://en.wikipedia.org/wiki/Hanoi_towerhttp://cs.wikipedia.org/wiki/Hanojsk%C3%A9_v%C4%9B%C5%BEe je matematický hlavolam. Skladá sa z troch kolíkov (veží). Na začiatku je na jednom z nich položených niekoľko kotúčov rôznych polomerov, zoradených od najväčšieho (naspodku) po najmenší (hore). Úlohou riešiteľa je premiestniť všetky kotúče na druhú vežu (tretiu pritom využije ako pomocnú pre dočasné odkladanie) podľa nasledujúcich pravidiel: V jednom ťahu sa dá premiestniť len jeden kotúč Jeden ťah pozostáva zo vzatia vrchného kotúča z niektorej veže a jeho položenie na vrchol inej veže. Je zakázané položiť väčší kotúč na menší. Jednoduché riešenie Striedavo sa presúva najmenší kotúč a iný kotúč ako najmenšie. Ak sa presúva najmenší kotúč, potom sa vždy presunie o jednu vežu ďalej v stále rovnakom smere, a to doprava pri celkovom párnom počte kotúčov a doľava pri nepárnom (predpokladáme, že veže stoja v rade vedľa seba, počiatočné je najviac vľavo a cieľové najviac vpravo). Ak je už najmenší kotúč na poslednej veži v tomto smere, presunie sa na vežu na opačnom konci. Ak má byť presunutý iný kotúč ako najmenší, je to možné vykonať vždy len jediným spôsobom. Týmto spôsobom možno hlavolam vyriešiť na najmenší možný počet ťahov. Rekurzívne riešenie Riešenie pomocou rekurzie vychádza z úvahy, že riešenie musí obsahovať najmenej jeden ťah, v ktorom je presunutý najväčší kotúč. Ten však možno presunúť len vtedy, keď sú všetky ostatné kotúče nasadené na tretej veži. Označme počet kotúčov n. Najprv teda presunieme n-1 kotúčov (všetky okrem najväčšieho) na odkladaciu vežu, potom presunieme najväčší kotúč z počiatočnej veže na cieľovú a nakoniec presunieme n-1 kotúčov z odkladacej veže na cieľovú. Presun n-1 kotúčov však môžeme vykonať pomocou rekurzívneho volania tohto algoritmu pre n-1 kotúčov, pretože najväčší kotúč nám pritom nebráni (na neho môžeme položiť akýkoľvek iný, takže môžeme postupovať, ako by nebol). Presný postup je teda nasledovný: Ak je n > 1, potom rekurzívnym volaním tejto procedúry presunieme n-1 kotúčov (t.j. všetky okrem najväčšieho) z počiatočnej veže na odkladaciu. Presunieme najväčší kotúč z počiatočnej veže na cieľovú. Ak je n > 1, potom rekurzívnym volaním tejto procedúry presunieme n-1 kotúčov z odkladacej veže na cieľovú. Implementácia rekurzívneho algoritmu v jazyku C: void Hanoj(int n,char zaciatocna, char cielova, char pomocna) { if (n>1) Hanoj(n-1, zaciatocna, pomocna, cielova); printf("%c => %c \n", zaciatocna,cielova); if (n>1) Hanoj(n-1, pomocna, cielova, zaciatocna); } Po zavolaní funkcie Hanoj(4,'A','B','C'); dostaneme výpis: A => C A => B C => B

A => C B => A B => C A => C A => B C => B C => A B => A C => B A => C A => B C => B Odkazy

Algoritmy vyhľadávania

Algoritmy a programovanie

Dátový typ

Štruktúry

Rekurzia

Algoritmy vyhľadávania

Algoritmy triedenia

Lineárny zoznam

Binárny strom

Numerické algoritmy

Grafové algoritmy

Vyhľadávanie

Vyhľadávanie je základnou úlohou pri práci s väčším objemom dát. Podľa povahy dát (zložitosti, usporiadania, veľkosti) volíme vhodný algoritmus vyhľadávania. Algortimy vyhľadávanie môžeme rozdeliť do niekoľko skupín:

1. Sekvenčné vyhľadávanie
2. Vyhľadávanie v usporiadaných zoznamoch (resp. poliach)
3. Vyhľadávanie v rekurzívnych štruktúrach

Sekvenčné vyhľadávanie

Jednoduché sekvenčné vyhľadávanie

Princíp tohto vyhľadávania je jednoduchý. Majme pole údajov o dĺžke *n*, v ktorých budeme hľadať údaj *x*. Pod pojmom údaj si môžeme ptredstaviť ľubovoľný dátový typ (celé číslo, reálne číslo, štruktúru, smerník na nejaký dátový typ). Princíp: postupne prechádzame pole od indexu *0* až po index *n-1*. V prípade, ak zistíme zhodu hľadaného prvku s prvkom v poli, tak funkciu ukončíme s úspechom, v opačnom prípade skončíme s výsledkom "prvok sa nenašiel"

Dohodnuté návratové hodnoty funkcie:

Vyhľadávacia funkcia bude vracať index prvku, ktorý je zhodný s hľadaným prvkom. V prípade neúspechu (hľadaný prvok sa v poli nevyskytuje) vráti funkcia hodnotu -1.

<properties> Názov funkcie=hladajSekvencne Návratová hodnota=index nájdeného prvku (-1 pri neúspechu) Parametre=pole prvkov - pole
rozmer poľa - n
hľadaný prvok - x Zložitosť algoritmu=O(n) </properties>

```
int hladajSekvencne(int *pole,int dlzka,int x)
{   int i=0;
```

```
      while( (i<dlzka) && (pole[i]!=x) )
             i++;
    if(i<dlzka)  return i;
      else return -1;
}
```

Analýza riešenia: Algoritmus je vhodný pre vyhľadávanie údajov v dátovej štruktúre jednorozmerné pole. Údaje v tomto poli môžu byť neusporiadané. Podmienka i<dlzka zaručí, že sa bude prehľadávať len v rozsahu poľa *pole*. Vždy sa začína na indexe i=0; tento index sa postupne zvyšuje (i++), pokiaľ nenarazíme na hľadaný prvok (pole[i]!=x). Podmienka na riadku 5 je tam pre kontrolu prípadu, keď sme prešli celé pole a hľadaný prvok sme nenašli.

Nevýhoda: každý cyklus sa vykonávajú 2 porovnania: i<dlzka a pole[i]!=x

Jednoduché sekvenčné vyhľadávanie s nárazníkom

V predchádzajúcom prípade sa v každom cykle robili 2 porovnania. Určitými úpravami dokážeme tieto porovnania znížiť len na 1 porovnanie. Bude to však chcieť drobnú úpravu poľa, v ktorom budeme vyhľadávať. Pole si upravíme tak, že hľadaný prvok sa tam bude vždy vyskytovať. Skutočná veľkosť poľa *pole* musí byť teda o jednu položku väčšia ako je počet prvkov v poli. Iba po splnení tejto podmienky môžeme urobiť priradenie, ako je ukázané v riadku 3. Toto priradenie nazveme **nárazník**.

<properties> Názov funkcie=hladajSekvencneN Návratová hodnota=index nájdeného prvku (-1 pri neúspechu) Parametre=pole prvkov - pole
rozmer poľa - n
hľadaný prvok - x Zložitosť algoritmu=O(n) </properties>

```
int hladajSekvencneN(int *pole, int n, int x)
{
  pole[n] = x; // musi byt na to vyhradene miesto!
  int i = 0;
  while (pole[i] != x)
    i++;
  if (i < n)
    return i;
  else
    return -1;
}
```

Analýza riešenia: Algoritmus je vhodný pre vyhľadávanie údajov v dátovej štruktúre jednorozmerné pole. Údaje v tomto poli môžu byť neusporiadané. Priradenie pole[n] = x dovoľuje vynechať kontrolu i<dlzka (funkcia hladajSekvencne). Vieme, že prvok v poli vždy nájdeme, preto stačí na konci urobiť porovnanie (riadok 7), či sme prvok x našli na pozícii menšej ako n (v rozsahu poľa). Ak áno, vrátime hodnotu premennej i, čo je index nájdeného prvku, ak sme hľadaný prvok našli na pozícii n, vrátime hodnotu -1, pretože na pozícii n je náš nárazník.

Poznámka: Touto úpravou sme vylepšili čas, vyhľadávania, ale nie zložitosť algoritmu.

Nasledujúci obrázok ukazuje porovanie časov vyhľadávania doteraz spomínaných algorimnov pri opakovanom vyhľadávaní vo veľkých (cca 1 000 000 položiek) poliach. <pLines ymin=0 axiscolor=888888 cubic angle=90 plots legend xtitle=pocet_cisel ytitle=cas> ,hladajSekvencne,hladajSekvencneN 10 000, 0.016, 0.015 100 000, 0.235, 0.125 200 000, 0.859, 0.813 300 000, 1.281, 1.234 500 000, 2.156, 2.063 700 000, 3.031, 2.891 1 000 000, 4.328, 4.141 2 000 000, 8.656, 8.375 5 000 000, 21.531, 20.657 10 000 000, 42.422, 41.235 </pLines>

Vyhľadávanie v usporiadaných zoznamoch

Zložitosť O(n) pre algorimy lineárneho, resp. sekvenčného vyhľadávania dokážeme znížiť, ak sú údaje v ktorých budeme vyhľadávať usporiadané. V tomto prípade nemusíme pole prvkov v ktorom hľadáme prechádzať celé (od indexu 0 až po n-1), ale stačí si skontrolovať položky len na určitých miestach.

Binárne vyhľadávanie

Predpoklad správneho vyhľadávania je, že prvku sú v poli usporiadané. Postup vyhľadávania:

Použité premenné

- *pole* - jednorozmerné pole, v ktorom budeme vyhľadávať
- *x* - prvok, ktorý hľadáme
- *n* - veľkosť pola *pole*.
- *lavy*, *pravy* - rozsah poľa, (začiatok , koniec) kde budeme hľadať.

1. Rozdeľ pole na 2 polovice. Index stredného prvku vypočítaš ako:
   - stred=(lavy+pravy)/2
2. Ak platí lavy>pravy, tak koniec, prvok sa nenašiel.
3. Zober prvok v strede poľa (na indexe stred) a porovnaj ho s hľadaným prvkom *x*
4. Ak sa *x* zhoduje so stredným prvkom - koniec. Výsledok je index tohto prvku
5. V opačnom prípade
   - Ak je *x* väčšie ako stredný prvok, tak hľadaj v pravej časti poľa
     - lavy=stred+1, pravy zostava nezmenený (choď na krok 1)
   - Ak je *x* menšie ako stredný prvok, tak hľadaj v ľavej časti poľa
     - lavy=zostava nezmenený, pravy = stred-1 (choď na krok 1)

Ukážme si konkrétny príklad (šedé políčka predstavujú indexy poľa, biele sú prvky poľa):

```
x=18
n=10
```

pole=

0	1	2	3	4	5	6	7	8	9	10
1	2	6	18	20	23	29	32	34	39	43

Pre implementáciu algoritmu binárneho vyhľadávania si musíme vypočítať hodnoty ľavého a pravého indexu poľa, v ktorom hľadáme. Na začiatok je to jednoduché:

```
lavy=0
pravy=n-1
```

- Porovnajme hľadaný prvok x=18 so stredným prvkom:

```
stred=(lavy+pravy)/2
stred=5
```

pole=

0	1	2	3	4	5	6	7	8	9	10
1	2	6	18	20	23	29	32	34	39	43

- pole[5] != x (23 != 18)
  - platí že x<23 (resp. x<pole[stred])
  - budeme vyhľadávať vľavo (žltá časť)
    - lavy zostane nezmeneny (0)
    - pravy=stred - 1 (4)
- Porovnajme hľadaný prvok x=18 so stredným prvkom:

```
stred=(lavy+pravy)/2 = (0+4)/2=2
stred=2
```

pole=

0	1	2	3	4	5	6	7	8	9	10
1	2	6	18	20	23	29	32	34	39	43

- pole[2] != x (6 != 18)
  - platí že x>6 (resp. x>pole[stred])
  - budeme vyhľadávať vpravo (žltá časť)
    - lavy = stred + 1 (3)
    - pravy zostane nezmeneny (4)
- Porovnajme hľadaný prvok x=18 so stredným prvkom:

```
stred=(lavy+pravy)/2 = (3+4)/2=3
stred=3
```

- pole[3] == x (18 = 18)
- Stredný prvok je zhodný s hľadaným
- Algoritmus končí
  - Výsledok je index nájdeného prvku, teda *stred*. V našom príklade je to 3.

Binárne vyhľadávanie - zdrojový kód v jazyku C

<properties> Názov funkcie=hladajBinarne
hladajBinarneR Návratová hodnota=index nájdeného prvku (-1 pri neúspechu) Parametre=pole prvkov - pole
rozmer poľa - n
hľadaný prvok - x Zložitosť algoritmu=O(Log(n)) </properties>

Keďže samotné definícia binárneho vyhľadávania je rekurzívna, uvádzame aj rekurzívnu verziu funkcie.

Iteračná verzia

```
int hladajBinarne(int *pole,int dlzka, int x)
{       int najdene=0;
int lavy = 0, pravy = dlzka - 1, stred;
while ( (lavy <= pravy) && (najdene==0) )
{
```

```
    stred = (lavy + pravy) / 2;
    if (pole[stred] == x)
        najdene=1;
    else
           if (pole[stred] < x)
            lavy = stred + 1;
          else
             pravy=stred - 1;
}
if(najdene==1)
      return stred;
else
      return -1;
}
```

Rekurzívna verzia

```
int hladajBinarneR(int *pole,int lavy, int pravy, int x)
{   if(lavy>pravy)
            return -1;
      else
      {
            int stred=(lavy+pravy)/2;
            if(pole[stred]==x)
              return stred;
            else
            {   if( x<pole[stred] )
                  return hladajB(pole,lavy,stred-1,x);
               else
                  return hladajB(pole,stred+1,pravy,x);
            }
      }
}
```

Ternárne vyhľadávanie

Predpoklad správneho vyhľadávania je, že prvku sú v poli usporiadané. Postup vyhľadávania je podobný ako pri binárnom vyhľadávaním abšak pole nerozdelíme na polovicu ale na tretiny.

Použité premenné

- *pole* - jednorozmerné pole, v ktorom budeme vyhľadávať
- *x* - prvok, ktorý hľadáme
- *n* - veľkosť pola *pole*.
- *i*, *j* - rozsah poľa, (začiatok , koniec) kde budeme hľadať.
- *m1* - index prvku v prvej tretine
- *m2* - index prvku v druhej tretine

1. Rozdeľ pole na tretiny
   1. m1=(i*2 + j) / 3;
   2. m2=(i + j*2) / 3;

2. Porovnaj, či sa hľadaný prvok zhoduje z prvkom na indexe m1, ak áno, tak koniec. Výsledok je index m1
3. Porovnaj, či sa hľadaný prvok zhoduje z prvkom na indexe m2, ak áno, tak koniec. Výsledok je index m2
4. Ak platí i>j, skonči, prvok x sa v poli nenachádza.
5. Ak je hľadaný prvok menší ako prvok pole[m1], tak hľadaj v prvej tretine poľa
   - i zostáva nezmenené , j=m1-1. Choď na krok 1.
6. Ak je hľadaný prvok väčší ako prvok pole[m2], tak hľadaj v tretej tretine poľa
   - i = m2+1 , j zostáva nezmenené. Choď na krok 1.
7. inak hľadaj v druhej tretine poľa
   - i = m1+1 , j=m2-+. Choď na krok 1.

Uveďme príklad: V prechádzajúcom príklade hľadajme číslo x=39

Na začiatku máme:

```
i=0
j=10
```

Vypočítame m1, m2

```
m1=( i*2 + j ) / 3 = (0+10)/3 = 3
m2=( i + j*2 ) / 3 = (0+20)/3 = 6
```

pole=

0	1	2	m1=3	4	5	m2=6	7	8	9	10
1	2	6	18	20	23	29	32	34	39	43

Teraz hľadáme v tretej tretine poľa:

```
i=m2+1=7
j=10
```

Vypočítame m1, m2

```
m1=( i*2 + j ) / 3 = (14+10)/3 =8
m2=( i + j*2 ) / 3 = (7+20)/3 = 9
```

pole=

0	1	2	3	4	5	6	7	m1=8	m2=9	10
1	2	6	18	20	23	29	32	34	39	43

Platí, že x=pole[m2]. Výsledkom bude teda hodnota m2 (index nájdeného prvku v poli)

Ternárne vyhľadávanie - zdrojový kód v C

<properties> Názov funkcie=hladajTernarne Návratová hodnota=index nájdeného prvku (-1 pri neúspechu) Parametre=pole prvkov - pole
rozmer poľa - n
hľadaný prvok - x Zložitosť algoritmu=O(Log n) </properties>

```
int hladajTernarne(int pole[],int i,int j,int x) {
  int m1,m2;
```

```
  m1=( i*2 + j ) / 3;
  m2=( i + j*2 ) / 3;
  if(x==pole[m1])
    return m1;
  if(x==pole[m2])
    return m2;
  if(i>j) return -1; // prvok sa nenašiel
  if(x<pole[m1]) // hľadaj v prvej tretine
    return(hladajTernarne(pole, i,m1-1,x));
  if(x>pole[m2])  // hľadaj v druhej tretine
    return(hladajTernarne(pole, m2+1,j,x));
  else   // hľadaj v tretej tretine
    return(hladajTernarne(pole, m1+1,m2-1,x));
}
```

Porovnanie binárneho a ternárneho vyhľadávania

Provnávané varianty

rekurzívne verzie funkcií

Spôsov porovnávania

Veľkosť poľa v ktorom hľadáme - n (os x grafu)

Počet hľadaných čísel - 1000

Počet opakovaní každého hľadania - 1000

<pLines ymin=0 axiscolor=888888 cubic angle=90 plots legend xtitle=pocet_cisel ytitle=cas_[s]> ,binarne,ternarne 1 000, 2.115, 2.094 5 000, 2.537, 2.513 10 000, 2.674, 2.622 50 000, 3.028, 3.005 100 000, 3.252, 3.032 500 000, 3.642, 3.358 1 000 000, 3.719, 3.471 5 000 000, 4.129, 3.771 10 000 000, 4.410, 3.963 50 000 000, 5.679, 4.304 100 000 000, 5.806, 5.248 </pLines>

Vyhľadávanie v rekurzívnych štruktúrach

Ďalším špeciálnym typom vyhľadávania je vyhľadávanie v rekurzívne definovaných štruktúrach. Medzi tieto štruktúry patria stromy, konkrétne Binárny strom alebo B-strom. V týchto štruktúrach sa nedá implementovať lineárne vyhľadávanie ani binárne (resp. ternárane) vyhľadávanie, pretože samotná štruktúra takéhoto dátového typu to nedovoľuje.

Vyhľadávanie v binárnych stromoch

Najjednoduchšia forma stromu je binárny strom. Binárny strom je stromová dátová štruktúra, v ktorej každý uzol má najviac dvoch potomkov. Binárny strom sa skladá z

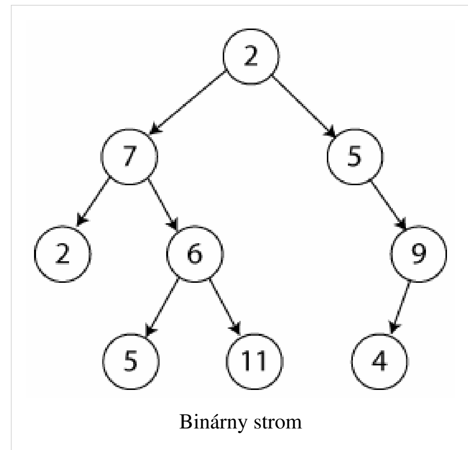
1. koreňového uzla
2. ľavý a pravý podstrom.
   - Oba podstromy sú taktiež binárne stromy.

Uzly na najnižšej úrovni stromu (2, 5, 11, 4) sa nazývajú *listy*. Vlastnosťou binárneho stromu je to, že údaje v ňom sú vždy usporiadané nasledovným spôsobom:

- naľavo od uzla sú tie prvky, ktoré majú hodnotu menšiu ako tohoto uzla
- napravo od uzla sú tie prvky, ktoré majú hodnotu väčšiu ako tohoto uzla

Pri tomto predpoklade môžeme sformulovať vyhľadávací algoritmus pre binárne stromy:

Budeme hľadať v binárnom strome, ktorý si označme *bstrom*. Tento každý uzol stromu môže mať žiadneho, jedného alebo dvoch potomkov. Potomkov budeme značiť *ľavý* a *pravý*. V strome *bstrom* budeme hľadať prvok *x*.



Binárny strom

1. Označ si koreň stromu v ktorom budeme hľadať ako *uzol*.
2. Porovnaj hodnotu uzla *uzol* s hľadaným prvkom *x*.
3. Ak sa tieto hodnoty zhodujú - koniec. Našli sme prvok *x*.
4. Ak aktuálny *uzol* nemá potomkov, tak koniec - Nenašli sme prvok *x*.
5. Ak je hodnota *x* menšia ako hodnota uzla *uzol*, tak označ si ako *uzol* potomka *ľavý*. Choď na krok 1.
6. Ak je hodnota *x* menšia ako hodnota uzla *uzol*, tak označ si ako *uzol* potomka *pravý*. Choď na krok 1.

Pseudokód algoritmu vyhľadávania:

```
hľadaj(strom, x)
uzol <- vrhol stromu strom
ak je uzol list
  koniec - x sme nenašli
ak je hodnota uzla == x tak
  koniec - vráť hodnotu x
ak je hodnota uzla < x tak
  hľadaj(uzol->lavy,x)
ak je hodnota uzla > x tak
  hľadaj(uzol->pravy,x)
```

značenie uzol->lavy, resp. uzol->pravy má význam, že ďalej budeme pracovať len s ľavou, resp. pravou časťou binárneho stromu.

Implementácia v jazyku C

Definícia binárneho stromu je v kapitole Binárny strom. Pre správne pochopenie nasledujúcej časti je potrebné vedieť pracovať s binárnym stromom.

Definujme dátovú štruktúru binárny strom:

```
struct TUzol
{   int data;
       TUzol *lavy,*pravy;
};
```

<properties> Názov funkcie=hladajRekurzivneStrom Návratová hodnota=smerník na nájdený prvok (v prípade neúspechu NULL) Parametre=binárny strom - strom

hľadaný prvok - x Zložitosť algoritmu=O(n) </properties>

```
TUzol* hladajRekurzivneStrom(TUzol *s, int x)
{
  if(s==NULL) return NULL;
  if(s->data == x ) return s;
  if(x < s->data) return hladajRekurzivneStrom(s->lavy);
  if(x > s->data) return hladajRekurzivneStrom(s->pravy);

}
```

Odkazy

- http://en.wikipedia.org/wiki/Linear_search
- http://en.wikipedia.org/wiki/Binary_search
- http://en.wikipedia.org/wiki/Ternary_search
- http://www.dcc.uchile.cl/~rbaeza/handbook/search_a.html
- http://www.cs.auckland.ac.nz/software/AlgAnim/searching.html
- http://www.cs.auckland.ac.nz/software/AlgAnim/trees.html

Algoritmy triedenia

Algoritmy a programovanie

Dátový typ
Štruktúry
Rekurzia
Algoritmy vyhľadávania
Algoritmy triedenia
::Triedenie výmenou
::Triedenie vkladaním
::Triedenie zlučovaním
::Triedenie delením
::Triedenie výberom
::Triedenie - ostatné princípy
Lineárny zoznam
Binárny strom
Numerické algoritmy
Grafové algoritmy

Triedenie

Triedaci algoritmus je v informatike algoritmus, ktorý zoraďuje prvky zoznamu v určenom poradí. Najpoužívanejšie sú numerické a lexikografické poradie. Efektívne triedenie je dôležité pre optimalizáciu použitia iných algoritmov (ako je napr. vyhľadávanie), ktoré vyžadujú pre svoju správnu funkčnosť utriedené zoznamy. Formálne povedané, výstupné utriedené údaje musia spĺňať dve podmienky:

1. Prvky vo výstupnom zozname majú neklesajúci trend (každý prvok je väčší alebo rovný ako predchádzajúci prvok)
2. Výstupom je permutácia vstupných resp. preusporiadanie týchto údajov.

Klasifikácia

Triediace algoritmy môžeme klasifikovať podľa:

Výpočtová zložitosť

(najhoršia, priemerná a najlepšia) pre zoznam s veľkosťou n položiek. Pre typické triediace algoritmy je prijateľná výpočtová zložitosť aspoň O(n.log n) a zlá O(n2). Ideálna zložitosť je O(n). Triediace algoritmy, ktoré používajú iba abstraktnú operáciu porovnania vždy majú priemernú zložitosť aspoň O(n.log n) porovnaní.

Využitie pamäte

(a využívanie ďalších počítačových zdrojov). Niektoré triediace algoritmy sú typu "in-place". To znamená, že im stačí O(1) alebo O(log n) pamäte na triedené položky.

Rekurzia

Niektoré algoritmy sú buď rekurzívne alebo nerekurzívne, niektoré môžu byť implementovateľné oboma spôsobmi (napr. Merge sort).

Stabilita

Stabilné triediace algoritmy - súvis s triedením dvojíc kľúč-hodnota, pričom existujú 2 také záznamy, že kľúče sú rovnaké. Stabilný algoritmus nezmení poradie týchto dvojíc vo výsledku.

Metóda triedenia

vkladanie, výmena, výber, zlúčenie, a pod.

Typy triediacich algoritmov

Vnútorné triedenie

Vnútorný triedenie je akýkoľvek triediaci proces, ktorý prebieha v hlavnej pamäti počítača. To je možné ak údaje, ktoré majú byť triedené majú takú veľkosť, že sa dokážu nahrať do hlavnej pamäti. Akékoľvek čítanie alebo zápis dát do/z externej pamäte môže značne spomaliť proces triedenia.

Uvažujem algoritmus Bubblesort, kde sa vymieňajú priľahlé záznamy s cieľom dostať ich do správneho poradia. V prípade, že časť triedeného súboru by bola b hlavnej pamäti a časť na externej pamäti, neustále by sa musel nahrávať do pamäti ten blok spboru, v ktorom sa vykonáva "prebulávanie".

Vonkajšie triedenie

Vonkajšie triedenie[1] je termín pre triedu triediacich algoritmov, ktoré môžu spracovať obrovské množstvo dát. Vonkajšie triedenie sa vyžaduje, ak sa triedené údaje nezmestia do hlavnej pamäte počítača (zvyčajne RAM) a namiesto toho musia byť umiestnené v pomalšej vonkajšej pamäti (zvyčajne pevný disk). Vonkajšie triedenie zvyčajne používa princíp zlučovania. Vo fáze triedenia sú dáta rozdelené do malých objemov ktoré sa zmestia do hlavnej pamäte. Pri fáze zlúčenia sú tieto bloky spojené do jedného väčšieho súboru.

Externý Mergesort

Príklad externého triedenia je algoristmus externého mergesortu.[2][3]. Ako príklad uvádzame triedenie 900MB súboru ktorý rozdelíme na 100MB bloky, ktoré sú nahraté do pamäte.

1. Načítaj 100MB dát do hlavnej pamäti a utrieď ich ľubovoľným triediacim algoritmom.
2. Zapíš utriedený súbor na disk.
3. Opakuj kroky 1. a 2. pokiaľ nie sú utriedené všetky 100MB bloky dát. Potom budeme tieto bloky spájať.
4. Načítaj do pamäti prvých 10MB z každého utriedeného súboru. Alokuj zvyšných 10MB pre výstupný buffer.
5. Vykonaj 9-násobný algoritmus merge (spájanie) zapíš výsledok do výstupného bufferu. Ak je výstupný buffer zaplnený, zapíš tieto údaje do výsledného utriedeného súboru. Ak je ľubovoľný z 9-tich vstupných bufferov prázdny, naplň ho ďalšími 10MB z jeho 100MB utriedeného súboru.

Porovnanie algoritmov

V nasledujúcej tabuľke je n počet záznamov, ktoré majú byť utriedené. Stĺpce "Priemerne" a "najhoršie" hovoria o časovej zložitosti, za predpokladu, že dĺžka všetkých kľúčov je konštantná, a preto všetky porovnania, výmeny a ďalšie potrebné operácie môže vykonať v konštantnom čase. "Pamäť" označuje množstvo pomocnej potrebnej okrem tej, ktorá je potrebná na samotné uloženie poľa ktoré sa bude triediť.

Názov	Najlepšie	Priemerne	Najhoršie	Pamäť	Metóda triedenia
Bubble sort					Výmena
Gnome sort		---			Výmena
Shake sort	---	---			Výmena
Selection sort					Výber
Shell sort	[4]	---			Vkladanie
Insert sort					Vkladanie
Merge sort					Zlučovanie
Heap sort					Výber
Quick sort					Delenie

Popis metód tiedenia

Existuje niekoľko základných druhov algoritmov univerzálneho vnútorného triedenia. Niektoré z pokročilejšách algoritmov využívajú viacero postupov.

Triedenie výberom

V súbore sa vždy nájde najmenší zo zostávajúcich položiek a uloží na koniec postupne vytváraného utriedeného súboru.

Triedenie vkladaním

Zo súboru neusporiadaných dát sa postupne berie položka po položke a vkladá sa na správne miesto v usporiadanom súbore (na začiatku je prázdny).

Triedenie výmenou

V súbore sa vždy nájde (nejakou metódou závislou na konkrétnom algoritme) nejaká dvojica prvkov, ktorá je v zlom poradí, a tieto prvky sa navzájom zamenia.

Triedenie zlučovaním

Vstupný súbor sa rozdelí na časti, ktoré sa (typicky rekurzívne) zoradia. Výsledné časti sa potom zlúčia takým spôsobom, aby aj výsledok bol zoradený.

Neexistuje žiaden "dokonalý" triediaci algoritmus, ktorý by bol ideálny pre všetky použitia. Rôzne algoritmy majú rôzne vlastnosti čo sa týka ich očakávané časovej a pamäťovej zložitosti, náročnosti implementácie a ďalších vlastností. Pre konkrétne podmienky sa tak často navrhujú špecifické varianty.

Princíp známych triediacich algoritmov

Bubble sort

Bubble sort predstavuje jednoduchý spôsob triedenia dát. Algoritmus začína na začiatku súboru údajov. Porovnáva prvé dva prvky, a ak je prvý väčší ako druhý, tak ich zamení. Toto robí pre každú dvojicu susedných prvkov až po koniec dátového súboru. Potom začína odznova s prvými dvoma prvkami, až pokiaľ nie je čo zameniť. Tento algoritmus je veľmi neefektívny, a málo použiteľný. Napríklad, ak budeme mať utriediť 100 prvkov, potom celkový počet porovnaní bude 10 000. Mierne lepšia varianta je Shake sort (Coctail sort), ktorý pracuje na princípe Buublesort ale prebublávanie sa deje striedavo smerom zľava doprava (a opačne). Modifikovaný Bubblesort zmenšuje počet porovnaní na 4950 (pri 100 prvkoch)

Selection sort

Princíp fungovania je, že sa vyberie z nezotriedenej časti postupnosti prvok s minimálnou hodnotou a vloží sa na koniec zotriedenej časti postupnosti. Na začiatku má zotriedená postupnosť 0 prvkov a nezotriedená má toľko prvokv, koľko je veľkosť poľa.

Shell sort

Pracuje podobne ako Selectionsort na princípe výberu. Shellsort vylepšuje vkladanie prvkov zadefinovaním kroku pri porovnávaní prvkov. Viacnásobným prechodom triedeným poľom a zmenšovaním kroku dosahuje lepšie výsledku ako InsertSort. Tento algoritmus je vhodné na zoznamy, ktoré sú skoro utriedené.

Merge sort

Merge sort využíva spájanie už zotriedených zoznamov do nového zoznamu, ktorý bude tiež zotriedený. Začína porovnávaním každých dvoch prvkov (tj 1 s 2,potom s 3, potom s 4, ...) a ich vzájomnou zámenou, aby boli zotriedené. Potom sa spojí každý z výsledných zoznamov (zatiaľ dvojprvkových) na štvrprvkový zoznam, a tak ďalej, až pokiaľ nie sú posledné dva zoznamy zlúčené do finálneho usporiadaného zoznamu.

Heap sort

Heapsort je efektívnejšia verzia Selectionsort-u. Funguje na výbere najmenšieho/najväčšieho prvku a umiestnením na začiatok/koniec zoznamu. Podobným spôsobom sa pokračuje zo zvyškom zoznamu. Pracuje sa s dátovou štruktúrou halda, čo je špeciálny prípad binárneho stromu.

Quick sort

Quicksort je algoritmus typu "rozdeľuj a panuj". Funguje na princípe rozdelenia triedeného poľa vzhľadom na vzťažný prvok (pivot) tak že prvky menšie ako pivot sú naľavo od pivota a prvky väčšie ako pivot sú napravo od pivota. Potom sa algoritmus rekurzívne aplikuje na tieto rozdelenia.

Odkazy

- Donald E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching.* Second Edition. Reading, Massachusetts: Addison-Wesley, 1998. ISBN 0-201-89685-0
- Algoritmy řazení ve slovníku algoritmů a datových struktur NIST [5]
- *Třídění* ve výukových materiálech k předmětu Základy algoritmizace [6]
- David R. Martin: Sorting Algorithm Animations - http://www.sorting-algorithms.com
- Algorithms in Java [7], Parts 1-4, 3rd edition by Robert Sedgewick. Addison Wesley, 2003.

[1] External_sorting - http://en.wikipedia.org/wiki/External_sorting
[2] Donald Knuth, *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998, ISBN 0-201-89685-0, Section 5.4: External Sorting, pp.248–379.
[3] Ellis Horowitzand Sartaj Sahni, *Fundamentals of Data Structures*, H. Freeman & Co., ISBN 0-716-78042-9.
[4] Robert Sedgewick: *Analysis of Shellsort and Related Algorithms* (http://www.cs.princeton.edu/~rs/shell/), Fourth Annual European Symposium on Algorithms, Barcelona, 1996

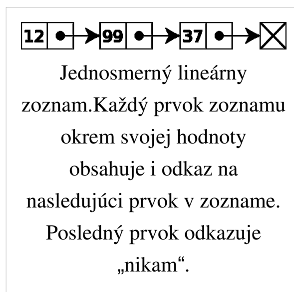
Lineárny zoznam

Algoritmy a programovanie

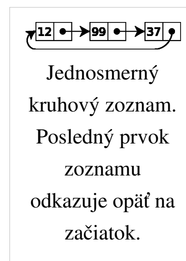
Dátový typ
Štruktúry
Rekurzia
Algoritmy vyhľadávania
Algoritmy triedenia
Lineárny zoznam
Binárny strom
Numerické algoritmy
Grafové algoritmy

Lineárny zoznam [1](lineárny spojový zoznam) je dynamická dátová štruktúra, navonok podobná poľu (umožňuje uložiť viacero hodnôt, ale iným spôsobom), obsahujúca jednu alebo viacero dátových položiek (štruktúr) rovnakého typu, ktoré sú navzájom lineárne previazané vzájomnými odkazmi pomocou smerníkov alebo referencií. Aby bol zoznam lineárny, nesmú v ňom existovať cykly so vzájomnými odkazmi.

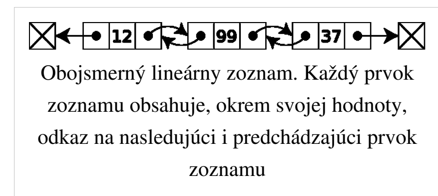
Lineárny zoznam poznáme jednosmerný a obojsmerný. V jednosmernom zozname odkazuje každá položka na nasledujúcu položku a v obojsmernom zozname odkazuje aktuálna položka na nasledujúcu ale aj predchádzajúcu položku. Definuje sa tu smerník alebo odkaz na určitý prvok zoznamu. Na konci (a začiatku) musí byť definovaná zarážka, ktorá označuje koniec zoznamu. Ak vytvoríme taký cyklus, že posledný prvok zoznamu odkazuje na prvý prvok zoznamu, dostaneme kruhový zoznam.



Jednosmerný lineárny zoznam.Každý prvok zoznamu okrem svojej hodnoty obsahuje i odkaz na nasledujúci prvok v zozname. Posledný prvok odkazuje „nikam“.



Jednosmerný kruhový zoznam. Posledný prvok zoznamu odkazuje opäť na začiatok.



Obojsmerný lineárny zoznam. Každý prvok zoznamu obsahuje, okrem svojej hodnoty, odkaz na nasledujúci i predchádzajúci prvok zoznamu

V nasledujúcom texte bude opisovaný **jednosmerný lineárny zoznam**.

Vlastnosti lineárneho zoznamu

- Jednosmerný lineárny zoznam je pamäťovo nenáročná dátová štruktúra
- Dovoľuje zoskupiť ľubovoľný počet prvkov. Počet prvkov je obmedzený len dostupnou pamäťovou kapacitou počítača.
- Dokáže jednoducho meniť svoju veľkosť počas behu programu.
- Prvky zoznamu, na rozdiel od polí neležia vedľa seba, ale v pamäti sú alokované podľa toho, kde je dostatok miesta.

Nasledujúci obrázok znázorňuje spôsob uloženia prvkov v pamäti pri použití poľa a lineárneho zoznamu.

Štruktúra lineárneho zozamu

Lineárny zoznam sa skladá z určitých prvkov (alebo uzlov), ktoré sú navzájom previazané vzájomnými odkazmi. Inak povedané stavebný prvok lineárneho zoznamuje samotnú hodnotu prvku (číslo, reťazec, štruktúra, ...) a odkaz na ďalší prvok. Ešte pripomeňme, že všetky prvky lineárneho zoznamu sú rovnakého typu.

Definujme štruktúru **TPrvok** (záznam pracovnej povahy), ktorý bude obsahovať:

- dátovú časť (ľubovolná položka, v našom prípade celé číslo)
- ukazovateľ na na ďalší záznam typu TPrvok

Definícia v jazyku C:

```
struct TPrvok
   {
      int data;
      TPrvok *dalsi;
   };
```

Vysvetlenie: Štruktúra TPrvok obsahuje dátovú časť označenú ako **data**. Odkaz na ďalší prvok je označený ako **dalsi**, a je to smetník na štruktúru TPrvok.

V nasledujúcom príklade budú ukážky definície prvkov lineárneho zoznamu, ktorý namiesto hodnoty celého čísla obsahuje inú dátovú časť.

```
struct TPrvok
   {
      char retazec[64];
      TPrvok *dalsi;
   };
//------------------------------------
struct TPrvok
```

```
   {
      TZlomok data;
      TPrvok *dalsi;
   };
//------------------------------------
struct TPrvok
   {
      double data[10];
      TPrvok *dalsi;
   };
```

- riadok 1: prvok lineárneho zoznamu je reťazec o dĺžke 63 znakov
- riadok 7: prvok lineárneho zoznamu je zlomok
- riadok 13: prvok lineárneho zoznamu je pole reálnych čísel o veľkosti 10.

Pre uloženie informácie o tom, kde má zoznam začiatok a kde končí si definujme ďalšiu štruktúru, ktorú nazveme **TZoznam**, ktorá bude obsahovať 2 smerníky na štruktúru TPrvok. Jeden smerník ukazuje na začiatok zoznamu a druhý smerník ukazuje na koniec zoznamu.

```
struct TZoznam {
   TPrvok *prvy;
   TPrvok *posledny;
} ;
```

Práca s lineárnym zoznamom

Pre lineárny zoznam platí nasledovné:

- Prvky *prvy*, *posledny*, *dalsi* sú ukazovatele na **TPrvok**.
- Časť štruktúry TPrvok označené ako *data* môže byť ľubovoľný dátový typ: reťazec, štruktúra, číslo...
- Ak je zoznam prázdny informačná štruktúra TZoznam má dva ukazovatele a oba majú hodnotu NULL. Teda ukazujú "nikam".
- Ak je v zozname jeden prvok tak ukazovatele *prvy* a *posledny* (TZoznam) ukazujú ne tento prvok a *dalsi* (TPrvok) má hodnotu NULL.
- Ak je v zozname dva a viac prvkov tak
  - smerník *prvy* (TZoznam) ukazuje na prvý prvok v lineárnom zozname,
  - smerník *posledny* (TZoznam) ukazuje na posledný prvok v lineárnom zozname,
  - Smerník *dalsi* (TPrvok) prvého prvku ukazuje na druhy prvok zoznamu a *dalsi* druhého prvku ukazuje na NULL.

Prechod cez lineárny zoznam

Uvedieme psseudokód, ktorý ukazuje princíp prechodu cez všetky prvky lineárneho zoznamu.

```
TZoznam LinZ;
// naplnenie zoznamu datami
begin
TPrvok *pom;
pom=LinZ.prvy;
rob
   begin
      vypis pom->data
```

```
      pom=pom->dalsi;
      pokial( pom != LinZ.posledny )
   end
end
```

Vysvetlenie uvedeného pseudokódu:

- Na riadku 1 si vytvoríme premennú typu TZoznam (Lineárny zoznam, ktorý udržuje informáciu o prvom a posledom prvku zoznamu).
- Naplnenie zoznamu je symbolicky naznačené na riadku 2.
- Zoznam pozostáva zo štruktúr typu smerník na TPrvok. Preto si na riadku 4 vytvárame premennú *pom* typu smerník na TPrvok.
  - Pre túto premennú netreba alokovať miesto v pamäti, pretože pomocou tejto premennej budeme prechádzať po existujúcich prvkoch zoznamu.
  - Na začiatok nastavíme premennú *pom* na prvý prvok zoznamu.
- V cykle (riadok 6) vypíšeme obsah prvku (riadok 8).
- Na riadku 9 je posunutie sa o jeden prvok ďalej.
- Cyklus ukončíme vtedy ak bude prvok pom zhodný z posledným prvkom zoznamu (smerní na posledný prvok zoznamu obsahuje premenná LinZ.

Vkladanie záznamov zo lineárneho zoznamu

Prvok môžeme do zoznamu vkladať dvoma spôsobmi:

1. Vloženie prvku na koniec (na začiatok). Vytváraný zoznam bude neusporiadaný.
2. Vloženie prvku na správne miesto. V tomto prípade sa snažíme aby bol zoznam stále usporiadaný od najmenšieho (najväčšieho) prvku po najväčší (najmenší).

Vkladanie nového prvku na koniec zoznamu

Pri vkladaní prvku do lineárneho zoznamu musíme rozlíšiť 2 rôzne situácie

1. Zoznam je prázdny
   - Vložený prvok bude jediným prvkom zoznamu. Bude prvým a zároveň posledným prvkom zoznamu
2. Zoznam nie je prázdny.
   - Vložený prvok umiestnime na koniec zoznamu. Tento nový prvok bude zároveň posledným prvkom zoznamu.

Na nasledujúcich obrázkoch bude znázornená situácia vloženia nového prvku do prázdneho a neprázdneho zoznamu na koniec.

Vkladanie nového prvku do usporiadaného zoznamu

Pri vkladaní prvku do usporiadaného lineárneho zoznamu musíme rozlíšiť nasledujúce situácie

1. Zoznam je prázdny
   - Vložený prvok bude jediným prvkom zoznamu. Bude prvým a zároveň posledným prvkom zoznamu. Toto riešenie je rovnaké ako v predchádzajúcom príklade.
2. Zoznam nie je prázdny.
   1. Nový prvok je menší ako prvý prvok.
   2. Nový prvok je väčší ako posledný prvok.
   3. Nový prvok treba umiestniť na správne miesto v zozname.

Na nasledujúcich obrázkoch bude znázornené jednotlivé prípady. Budeme vychádzať z neprázdneho zoznamu z nasledujúceho obrázka.

Do zoznamu vkladáme nový prvok s hodnotou "-5". Zistili sme že tento prvok je menší ako prvý prvok, preto ho umiestnime na začiatok zoznamu. Po vložení to bude nový prvý prvok zoznamu.

Do zoznamu vkladáme nový prvok s hodnotou "20". Zistili sme že tento prvok je väčší ako posledný prvok, preto ho umiestnime na koniec zoznamu. Po vložení to bude nový posledný prvok zoznamu.

Do zoznamu vkladáme nová prvok s hodnotou "5". Zistili sme že tento prvok treba umiestniť na správne miesto v zozname. Po nájdení správneho miesta (medzi prvky 0 a 10) ho medzi tieto prvky vložíme. Prvky medzi ktoré budeme nový prvok vkladať si označme ako *lavy* a *pravy*. Postupnosť operácií, ktoré musíme urobiť je na nasledujúcom obrázku:

1. Smerník *dalsi* nového prvku nastavíme na prvok *pravy*
2. Zmažeme smerník *dalsi*, ktorý ukazuje z prvku *lavy* na *pravy*.
3. Smerník *dalsi* prvku *lavy* nastavíme tak, aby ukazoval na nový prvok.

Implementácia lineárneho zoznamu v jazyku C

Dátová štruktúra opisujúca lineárny zoznam

```
struct TPrvok
   {
      int data;
      TPrvok *dalsi;
   };

struct TZoznam {
   TPrvok *prvy;
   TPrvok *posledny;
} ;
```

Vloženie prvku na koniec zoznamu

Vo funkcii Vloz_k budeme do zoznamu vkladať nový prvok na koniec zoznamu.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, do ktorého vkladáme nový údaj,
- int x - hodnota, ktorú budeme do zoznamu *z* vkladať.

Návratová hodnota: žiadna.

Do zoznamu budeme vkladať nový prvok, ktorého hodnota je definované vo vstupnom parametri x. Keďže sa zoznam skladá iba zo smerníkov na štruktúru TPrvok, je potrebné si vytvoriť (a alokovať miesto) smerník (riadok 3) na štruktúru TPrvok. Do tejto štruktúry je potom treba nastaviť hodnotu x (riadok 4). Premenná *novy* reprezentuje prvok, ktorý budeme do zoznamu vkladať. Ak je zoznam *z* prázdny, tak novo vložený prvok bude zároveň prvým aj posledným prvkom zoznamu (riadok 8 a 11). Ak v zozname *z* existuje nejaký prvok, tak teba nový prvok vložiť na koniec zoznamu. Poradie operácií, ktoré treba urobiť je:

1. Hodnota smerníku *dalsi* posledného prvku zoznamu je NULL. Po pridaní nového prvku bude mať tento smerník z.posledny->dalsi hodnotu adresy nového prvku (riadok 10)
2. Po pridaní nového prvku na koniec treba upraviť smerník *posledny* (*z.posledny*) zoznamu z: posledný prvok je v skutočnosti novo vložený prvok (riadok 11).

```
void Vloz_k(TZoznam &z, int x)   // vlozi prvok na koniec
{ // vlozime novy prvok, ktory ma hodnotu x
  TPrvok *novy = new TPrvok;
  novy->data = x;
  novy->dalsi = NULL;

  if (z.prvy == NULL)  // prazdny zoznam
       z.prvy = novy;
  else
       z.posledny->dalsi = novy;
  z.posledny = novy;
}
```

Vloženie prvku na začiatok zoznamu

Vo funkcii Vloz_z budeme do zoznamu vkladať nový prvok na začiatok zoznamu.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, do ktorého vkladáme nový údaj,
- int x - hodnota, ktorú budeme do zoznamu *z* vkladať.

Návratová hodnota: žiadna.

Význam riadkov 3 až 5 je rovnaký ako v predchádzajúcej funkcii. Ak je zoznam *z* prázdny, tak novo vložený prvok bude zároveň prvým aj posledným prvkom zoznamu (riadok 8 a 11). Ak v zozname *z* existuje nejaký prvok, tak teba nový prvok vložiť na začiatok zoznamu. Poradie operácií, ktoré treba urobiť je:

1. Hodnota smerníku *dalsi* vkladaného prvku zoznamu je NULL (riadok 5). Po pridaní nového prvku bude hodnota jeho smerníka *dalsi* obsahovať adresu prvku, ktorý bol pred vložením prvý (riadok 10)
2. Po pridaní nového prvku na začiatok treba upraviť smerník *prvy* (*z.prvy*) zoznamu z: prvý prvok je v skutočnosti novo vložený prvok (riadok 11).

```
void Vloz_z(TZoznam &z, int x)
{ // vlozime novy prvok, ktory ma hodnotu x
  TPrvok *novy = new TPrvok;
  novy->data = x;
  novy->dalsi = NULL;

  if (z.prvy == NULL)  // prazdny zoznam
     z.posledny = novy;
  else
     novy->dalsi=z.prvy
  z.prvy = novy;
}
```

Vloženie prvku do usporiadaného zoznamu

Vo funkcii Vloz budeme do zoznamu vkladať nový prvok do zoznamu na také miesto, aby bol zoznam stále usporiadaný.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, do ktorého vkladáme nový údaj,
- int x - hodnota, ktorú budeme do zoznamu *z* vkladať.

Návratová hodnota: žiadna.

Princíp vkladania vkladania nového prvku je opísaný v kapitole #Vkladanie nového prvku do usporiadaného zoznamu. V nasledujúcom texte bude vysvetlená len implementácia v jazyku C.

- riadok 7-8 : vkladanie do prázdneho zoznamu
- riadok 9-30 : vkladanie do neprázdneho zoznamu
  - riadok 10: hodnota x < hodnota prvého prvku zoznamu. Nový prvok bude vložený na začiatok zoznamu.
  - riadok 16-20: hodnota x > hodnota posledného prvku zoznamu. Nový prvok bude vložený na koniec zoznamu.
  - riadok 21-30: hodnota x menšia hodnota prvého prvku a zároveň menšia ako hodnota posledného prvku
    - riadok 23: 2 pomocné smerníky p1 a p2. p1 ukazuje na prvý prvok (*p1=z.prvy*), p2 ukazuje na druhý prvok zoznamu (*p2=z.prvy->dalsi*)
    - riadok 24: v cykle while hľadáme také miesto kde neplatí, že hodnota x je väčšia ako hodnota prvku p2. V prípade, ak platí že x>p2->data, tak smerníky p1 a p2 posunieme o jedno miesto z zozname ďalej (riadok 25 a 26).
    - Po skončení cyklu while platí že hodnota prvku x je menšia ako hodnota dátovej časti prvku p2. Našli sme teda miesto, kde budeme prvok *novy* vkladať. Nový prvok vložíme medzi prvky p1 a p2.
    - riadok 28-29: vloženie nového prvku medzi prvky p1 a p2.

```
void Vloz(TZoznam &z, int x)  // vlozi prvok zotriedene
{
   TPrvok *novy = new TPrvok;
   novy->data = x;
   novy->dalsi = NULL;

   if (z.prvy == NULL)  // prazdny zoznam
      z.prvy = z.posledny = novy;
   else        // v zozname je nejaky prvok
      if (x < z.prvy->data)  // ak je x < ako hodnota prveho
      {              // prvku vlozim novy prvok na zaciatok
         novy->dalsi = z.prvy;
         z.prvy = novy;
      }
      else
         if (x > z.posledny->data)  // x > ako posledny prvok zoznamu
         {
            z.posledny->dalsi = novy;
            z.posledny = novy;
         }
         else  // vlozenie niekde do stredu zoznamu
         {
            TPrvok *p1 = z.prvy, *p2 = z.prvy->dalsi;
```

```
            while (p2->data < x) // hladame miesto, kde vlozime prvok
            {   p1 = p2;
                p2 = p2->dalsi; // posun o jeden prvok
            }
            p1->dalsi = novy; // vlozenie noveho zaznamu

            novy->dalsi = p2; // na spravne miesto
         }
}
```

Vyhľadanie prvku v lineárnom zozname

Vo funkcii *hladaj* budeme v zozname hľadať prvok s danou hodnotou.

Parametre funkcie:

- TZoznam z - lineárny zoznam, v ktorom budeme vyhľadávať
- int x - hodnota, ktorú budeme v zozname *z* hľadať.

Návratová hodnota: V prípade úspechu smerník na nájdený prvok. V prípade neúspechu NULL.

Princíp funkcie o opísaný v kapitole Prechod cez lineárny zoznam.

```
TPrvok* hladaj(TZoznam z, int x)
{
  TPrvok *p = z.prvy;  // pomocny prvok typu TPrvok
  while (p != NULL) // opakuj, pokial neprides na koniec zoznamu
  {
    if (p->data == x) return p;        // ak sa prvok nasiel
    p = p->dalsi;      // chod na dalsi prvok zoznamu
  }
  return NULL; // ak si nic nenasiel, vrat 0
}
```

Výpis prvkov lineárneho zoznamu

Vo funkcii *vypis* budeme vypisovať postupne všetky prvku lineárneho zoznamu.

Parametre funkcie:

- TZoznam z - lineárny zoznam, ktorého prvky budeme vypisovať

Návratová hodnota: žiadna

Princíp funkcie o opísaný v kapitole Prechod cez lineárny zoznam.

```
int JePrazdny(TZoznam z)
{ // ak je zoznam prazdny, funkcia vrati 1, inak 0
  return z.prvy == NULL;
}
void vypis(TZoznam z)
{
  if (JePrazdny(z)) return;  // ak je zoznam prazdny, skonci
  TPrvok *p = z.prvy;  // pomocny prvok typu TPrvok
  while (p->dalsi != NULL) // prechádzaj cez vsetky prvky
  {
```

```
    cout << p->data << ", ";  // vypis datovu cast prvku
    p = p->dalsi;       // chod na dalsi prvok
  }
  cout << p->data << endl; // vypis posledny prvok
}
```

Zmazanie prvého prvku zoznamu

Vo funkcii *ZmazPrvy* budeme mazať prvý prvok zoznamu. Po zmazaní musia mať smerníky *prvy* a *posledny* zoznamu *z* aktuálne hodnoty.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, v ktorom budeme mazať prvý prvok.

Návratová hodnota: žiadna

Na riadku 4 je vytvorený pomocný smerník, ktorý bude mať hodnotu prvého prvku zoznamu. Ako prvé je potrebné nastaviť smerník *prvy* zoznamu *z* (*z.prvy*) na prvok, ktorý bude po zmazaní prvého prvku zoznamu nový prvý prvok. Toto je vlastne druhý prvok zoznamu (*p->dalsi*). Na koniec (riadok 6) treba zmazať prvý prvok.

```
void ZmazPrvy(TZoznam &z)
{   if (z.prvy == NULL)   // prazdny zoznam
     return;
   TPrvok *p=z.prvy;
   z.prvy=p->dalsi;
   delete p;
}
```

Zmazanie posledného prvku zoznamu

Vo funkcii *ZmazPosledny* budeme mazať posledný prvok zoznamu. Po zmazaní musia mať smerníky *prvy* a *posledny* zoznamu *z* aktuálne hodnoty.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, v ktorom budeme mazať prvý prvok.

Návratová hodnota: žiadna

- riadok 2: zoznam je prázdny. Funkcia skončí
- riadok 5: v zozname je len jeden prvok. Po jeho zmazaní bude zoznam prázdny.
- riadok 11: v zozname je viac ako 1 prvok. Po zmazaní posledného prvku bude nový posledný prvok ten, ktorý bol v pôvodnom zozname predposledný.
  - Hľadanie predposledného prvku: riadok 11 a 12. Predposledný prvok je taký prvok p, pre ktorý platí:
    - *p->dalsi* je posledný prvok zoznamu. Vieme, že posledný prvok zoznamu má hodnotu smerníka dalsi NULL. Preto ak platí výraz p->dalsi->dalsi == NULL, tak potom je prvok p predposledný.
    - Ak sme našli predposledný prvok, tak môžeme zmazať posledný prvok (riadok 13)
    - Prvok *p* sa stane posledným prvkom. Posledný prvok má hodnotu smerníka *dalsi* NULL (riadok 14)
    - Prvok *p* nastavíme ako posledný prvok zoznamu *z* (riadok 15).

```
void ZmazPosledny(TZoznam &z)
{   if (z.prvy == NULL)   // prazdny zoznam
     return;
  TPrvok *p = z.prvy;
  if (p->dalsi == NULL)   // je len jeden prvok
```

```
  { delete p;
    z.prvy = z.posledny= NULL;  // upravime strukturu z
    return;
  }
  // najdeme predposledny prvok
  while (p->dalsi->dalsi != NULL)
    p = p->dalsi;
  delete p->dalsi;      // zmazeme posledny
  p->dalsi = NULL;  // a nastavime NULL na aktualny posledny prvok
  z.posledny = p;  //upravime zoznam tak, aby ukazoval spravne na
}                     // posledny prvok
```

Zmazanie lineárneho zoznamu

Vo funkcii *Zmaz* budeme mazať všetky prvky zoznamu.

Parametre funkcie:

- TZoznam &z - odkaz na lineárny zoznam, ktorého prvky budeme mazať.

Návratová hodnota: žiadna

Je jedno, či budeme mazať zoznam od začiatku, alebo od konca. Implementácia mazania Zmaz2 je ale efektívnejšia pretože zložitosť funkcie *ZmazPrvy* je O(1) a zložitosť funkcie *ZmazPosledny* je O(n).

```
void Zmaz(TZoznam &z)
{
  while (!JePrazdny(z)) // pokial je v zozname nejaky prvok
    ZmazPosledny(z); // zmaze posledny prvok
}

void Zmaz2(TZoznam &z)
{
  while (!JePrazdny(z)) // pokial je v zozname nejaky prvok
    ZmazPrvy(z); // zmaze prvy prvok
}
```

Použitie lineárneho zoznamu v programe

```
void main()
{ TZoznam zoznam;
  zoznam.prvy =NULL;
  zoznam.posledny = NULL;
  int udaj,n=4; //pocet zaznamov
  for(int i=0;i<n;i++)
  {   cin>>udaj;
       Vloz_z(zoznam, udaj);
  }
  for(int i=0;i<n;i++)
  {   cin>>udaj;
       Vloz_k(zoznam, udaj);
  }
  vypis(zoznam);
```

```
Zmaz(zoznam);

for(int i=0;i<n;i++)
{   cin>>udaj;
     Vloz(zoznam, udaj);
}
vypis(zoznam);
Zmaz(zoznam);
```

Odkazy

[1] Lineární seznam - http://cs.wikipedia.org/wiki/Line%C3%A1rn%C3%AD_seznam

Binárny strom

UPOZORNENIE: Článok nebolo možné vykresliť - na výstup sa zapíše čistý text.
Možné príčiny problému sú: (a) chyba v softvéri pdf-writer (b) problematická MediaWiki syntax článku (c) príliš široká tabuľka

Algoritmy a programovanieDátový typŠtruktúry_jazyk_CŠtruktúryRekurziaAlgoritmy vyhľadávaniaAlgoritmy triedeniaLineárny zoznamBinárny stromNumerické algoritmyGrafové algoritmy Strom je dynamická dátová štruktúra. Stromy sa používajú v prípade hierarchických vzťahov a rekurzívnych štruktúr objektov.Definícia Binárny strom je v informatike stromová dátová štruktúra, ktorej každý vrchol má najviac dvoch potomkov. Zvyčajne sa označujú ako ľavý a pravý. Jedno z bežných použití binárneho stromu je binárny vyhľadávací strom; iné je binárna halda. Strom má hierarchickú štruktúru: Rodič - potomok Binárny strom je usporiadaný strom v ktorom má každý vrchol najviac dvoch synov. Z programátorského hľadiska je výhodné údajovú štruktúru strom definovať pomocou rekurzie: Strom typu T je buď http://cec.truni.sk/stoffov/dynamicke-udajove-struktury/Cast2/ prázdna množina alebo vrchol typu T spolu s konečným počtom pripojených disjunktných stromov typu T, ktoré nazývame podstromy Každý strom je tvorený ďalšími stromami, pre ktoré platí rovnaká definícia. Zviditeľníme ju na príklade binárneho stromu: Príklad binárneho stromuNa vedľajšom obrázku je vidieť, že aj podstrom sa skladá z koreňa a dvoch ďalších podstromov (niektorý z nich môže byť aj prázdny!). Každý list (t.j. vrchol bez synov) je tiež stromom, jeho ľavý aj pravý podstrom sú však prázdne. Binárny vyhľadávací strom Hodnoty v uzloch sú usporiadané tak, že pre každý uzol stromu u platíhttp://sk.wikipedia.org/wiki/Bin%C3%A1rny_vyh%C4%BEad%C3%A1vac%C3%AD_strom: hodnota uložená v u je väčšia alebo rovná ako hodnota uložená v ľavom podstrome u hodnota uložená v u je menšia alebo rovná ako hodnota uložená v pravom podstrome uPotom je možné v tomto strome jednoduchým spôsobom vyhľadať danú hodnotu h: nastav koreň ako aktuálny uzol (u) pokiaľ je v u uložená hodnota h, algoritmus končí (a vráti u) ak je u list, algoritmus končí (hodnota h nebola v strome nájdená) hodnota h sa porovná s hodnotou v u (nech je táto hodnota h') ak je h \le h', do u sa uloží ľavý potomok u inak sa do u uloží pravý potomok u pokračuj bodom 2 Reprezentácia binárneho stromu Štruktúra Tuzol binárneho stromu sa skladá z:hodnoty (hodnôt) uzla 2 ukazovateľov na ďalšie uzly (vetvy) struct TUzol { int data; TUzol *lavy,*pravy; }; V našom prípade je hodnota uzla premenná data, typu celé číslo. Premenná data môže byť ľubovoľného typu, napr. char, double, ale aj vlastne definovaných typov - štruktúra. Usporiadanie prvkov TUzol v binárnom stromeNa sprístupnenie stromu potrebujeme ukazovateľ na koreň, v programe teda stačí deklarovať: TUzol *strom; Ak niektorý vrchol nemá syna, hodnoty príslušných ukazovateľov sú NULL Operácie s binárnym stromom V nasledujúcich prípadoch budeme pracovať s binárnym

vyhľadávacím stromom. Vloženie hodnoty do stromuÚloha: vložiť do stromu nový uzol. Nový uzol treba vložiť na správne miesto tak, aby bol strom usporiadaný.Analýza:Strom je prázdny. Novo vložený uzol bude koreňom stromu. Strom nie je prázdny. V strome je už nejaký uzol. V tomto prípade treba zistiť, na ktorú stromu stromu sa má uzol vložiť. Rozdhodnutie robíme podľa hodnoty dátovej časti nového uzla: Ak je hodnota nového uzla menšia ako hodnota existujúceho uzla, tak ho dáme vľavo, inak ho vložíme vpravo. Ak je vpravo, alebo vľavo už nejaký uzol, postupujeme rekurzívne Algoritmus vloženia:Definujme procedúru Vloz, ktorá bude mať 2 parametre: Začiatočný uzol stromu, do ktorého budeme vkladať hodnotu. Keďže strom je rekurzívna štruktúra, algoritmus vloženia bude rekurzívny.Procedúra Vloz(Uzol U, hodnota H)Ak aktuálny uzol U neobsahuje žiadnu hodnotu do dátovej časti uzla U nastav hodnotu H Nastav hodnoty smerníkov lavy a pravy uzla U na hodnotu NULLKoniec Ak je hodnota H < ako hodnota v uzle U Ak nemá uzol U ľavého potomka Vytvor nový uzol U_n a nastav mu v dátovej časti prázdnu hodnotu Zavolaj procedúru Vloz s parametrami: Uzol U - ľavý (novo vytvorený) potomok uzla U (U_n), hodnota - HAk je hodnota H > ako hodnota v uzle U Ak nemá uzol U pravého potomka Vytvor nový uzol U_n a nastav mu v dátovej časti prázdnu hodnotu Zavolaj procedúru Vloz s parametrami: Uzol U - pravý (novo vytvorený) potomok uzla U (U_n), hodnota - HZnázornenie algoritmu vloženia Počiatočný stav. Strom reprezentuje jeden vrchol, ktorý nemá hodnotu (v našom prípade ju znázorňuje hodnota 0) Do stromu vkladáme hodnotu 5. V opísanom algoritme je táto situácia zachytená v kroku 1. Následne sa vykonajú kroky 1.1, 1.2 a 1.3 Do stromu vkladáme hodnotu 3. V algoritme platí krok 2. Krok 2.1 : uzol U nemá ľavého potomka tak takéhoto potomka vytvoríme (krok 2.1.1). Zavoláme procedúru Vloz s parametrami: uzol U - lavý potomok uzla U (teda novo vytvorený uzol). Po rekurzívnom zavolaní procedúry Vloz (krok 2.2) bude platiť podmienka v kroku 1. Do stromu vkladáme hodnotu 4. V algoritme platí krok 3. Krok 3.1 : uzol U nemá pravého potomka tak takéhoto potomka vytvoríme (krok 3.1.1). Zavoláme procedúru Vloz s parametrami: uzol U - pravý potomok uzla U (teda novo vytvorený uzol). Po rekurzívnom zavolaní procedúry Vloz (krok 3.2) bude platiť podmienka v kroku 2. Ďalší postup je rovnaký ako pri vkladaní hodnoty 3.Prehľadávanie stromu Za základnú operáciu sa však považuje prehľadávanie stromu (pohyb po strome). Vzhľadom na to, že strom je výhodne definovaný ako rekurzívna údajová štruktúra, aj na prehľadávanie sa využívajú rekurzívne metódy. Keďže strom je rekurzívna štruktúra prehľadávanie stromu bude tiež rekurzívne. Podľa toho ako budeme postupovať pri prehľadávaní stromu, rozlišujeme metódy prehľadávania na: preorder inorder postorder Majme nasledujúci binárny strom: Usporiadaný binárny stromPrehľadávanie preorder: Postup prehľadávanie (výpisu hodnôt) binárneho stromu: Vypíš hodnotu uzla prehľadaj ľavý podstrom prehľadaj pravý podstrom Pri tomto postupe bude výpis binárneho stromu na predchádzajúcom obrázku nasledovný: 10 5 1 0 4 7 6 9 12 Prehľadávanie inorder: Postup prehľadávanie (výpisu hodnôt) binárneho stromu: prehľadaj ľavý podstrom Vypíš hodnotu uzla prehľadaj pravý podstrom Pomocou tohoto spôsobu prehľadávania stromu dostaneme pri výpise postupnosť usporiadaných hodnôt uzlov stromu od najmenšej po najväčšiu hodnotu. Pri tomto postupe bude výpis binárneho stromu na predchádzajúcom obrázku nasledovný: 0 1 4 5 6 7 9 10 12 Prehľadávanie postorder: Postup prehľadávanie (výpisu hodnôt) binárneho stromu: prehľadaj ľavý podstrom prehľadaj pravý podstrom Vypíš hodnotu uzla Pri tomto postupe bude výpis binárneho stromu na predchádzajúcom obrázku nasledovný: 0 4 1 6 9 7 5 12 10 Zmazanie stromu Princíp zmazania je podobný ako prechádzanie stromom: Ak nemá uzol žiadneho potomka, zmažeme uzol Ak má potomka zmažeme ľavú časť stromu Zmažeme pravú časť stromu Implementácia v jazyku CVloženie hodnoty do stromu void PridajUzol(TUzol *uzol, int udaj) { if (!uzol-> data) // ak v danom uzle nie su ziadne data- pridame novy udaj { uzol->data = udaj; // do uzla skopirujeme data uzol->lavy = uzol->pravy = NULL; // nastavime pointre na lavy a pravy uzol return; } if(uzol->data >= udaj) // pridavame vlavo { if (!uzol->lavy) // ak uzol vlavo neexisuje { uzol->lavy = new TUzol; // tak ho vytvorime uzol->lavy->data=0; } PridajUzol(uzol->lavy, udaj); // opakujeme rekurzivne funkciu s lavym uzlom } else // pridavame vpravo { if (!uzol->pravy) { uzol-> pravy = new TUzol; uzol->pravy->data=0; } PridajUzol(uzol-> pravy, udaj); } } Výpis stromu inorder void vypis(TUzol *s) { if(!s) return; else { vypis(s->lavy); cout<<s->data<<endl; vypis(s->pravy); } } Vyhľadávanie v strome (inorder) Funkcia hladaj bude v strome vyhľadávať uzol s hodnotou x. V prípade úspechu vráti smerník na tento prvok. V opačnom prípade vráti hodnotu NULL. TUzol* hladaj(TUzol *s, int x) { if(s==NULL) return NULL; else { if(x < s->data) return hladaj(s->lavy,x); if(s->data==x) return s; if(x > s->data) return hladaj(s->pravy,x); } } Zmazanie stromu

Princíp zmazania je podobný ako vypísania. Ak nemá uzol žiadneho potomka, zmažeme aktuálny uzol Ak má potomka zmažeme ľavú časť stromu zmažeme pravú časť stromu void ZmazStrom(TUzol *uzol) { if (!uzol) return; if (uzol->lavy) // ak existuje lavy uzol { ZmazStrom(uzol->lavy); delete uzol->lavy; } if (uzol->pravy) // ak existuje pravy uzol { ZmazStrom(uzol->pravy); delete uzol->pravy; } } Zdroje

---

Numerické algoritmy

Algoritmy a programovanie

Dátový typ

Štruktúry

Rekurzia

Algoritmy vyhľadávania

Algoritmy triedenia

Lineárny zoznam

Binárny strom

Numerické algoritmy
::Algoritmy hľadania nulových miest
::Algoritmy numerického derivovania
::Algoritmy numerického integrovania
::Algoritmy interpolácie
::Algoritmy aproximácie
::Algoritmy numerického riešenia diferenciálnych rovníc

Grafové algoritmy

Numerická analýza je štúdia o algoritmoch, ktoré používajú číselné hodnoty (na rozdiel od všeobecnej symbolické manipulácie) pre problémy spokitej matematiky (na rozdiel od diskrétne matematiky).

Numerické algoritmy[1] nájdu uplatnenie vo všetkých oblastiach strojárenstva, fyzikálnych vied, ale v 21 storočí, dokonca aj v umení sú prvky prvky numerickej analýzy. Obyčajné diferenciálne rovnice sú použité pre opis pohybu nebeských telies (planéty, hviezdy a galaxie) , optimalizácia sa používa v menežérstve. Numerická lineárna algebra je dôležitá pre analýzu dát, stochastické diferenciálne rovnice a Markove reťaze sú nevyhnutné v simulácii živých buniek pre medicínu a biológiu.

Príchodom počítačov sa stáva výpočet numerických úloh jednoduchším. Známe numerické algoritmy riešiace daný problém dokážeme implementovať v ľubovoľnom programovacom jazyku.

Rozdelenie algoritmov

Numerické algoritmy môžeme rozdeliť do kategórií podľa toho, aké problémy riešia:

- Algoritmy hľadania nulových miest
- Algoritmy numerického derivovania
- Algoritmy numerického integrovania
- Algoritmy interpolácie
- Algoritmy aproximácie
- Algoritmy riešenia sústavy lineárnych rovníc

Softvérová podpora riešenia

Pre riešenie numerických algoritmov existuje viacero softvérových nástrojov, ktoré sa špecializujú na efektívne riešenie týchto problémov

- SciDaVis[2]

Referencie

[1] http://en.wikipedia.org/wiki/Numerical_algorithm
[2] http://scidavis.sourceforge.net/

Grafové algoritmy

Algoritmy a programovanie

Dátový typ
Štruktúry
Rekurzia
Algoritmy vyhľadávania
Algoritmy triedenia
Lineárny zoznam
Binárny strom
Numerické algoritmy
Grafové algoritmy
::Prehľadávanie do šírky
::Prehľadávanie do hĺbky
::Hľadanie najkratšej cesty

Teória grafov je časť diskrétnej matematiky, ktorá skúma vlastnosti grafov.

Graf

Graf *G* je usporiadaná dvojica *(V,E)*, kde

- *V* – množina vrcholov
- *E* – množina hrán

Každá hrana e () je pár (v,w), kde

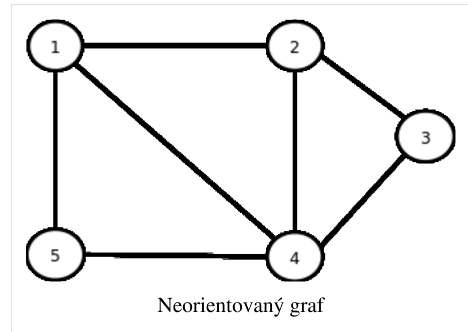
Neorientovaný graf

Graf alebo **neorientovaný graf** *G* je usporiadaná dvojica *G* = (*V*, *E*), kde:

- *V* je neprázdna konečná množina **vrcholov** grafu,
- *E* je množina neusporiadaných dvojíc typu {*u*, *v*}, kde *u* ≠ *v*, nazývaných **hrany** grafu.

Príklad neorientovaného grafu:

V={1,2,3,4,5}
E={(1,2),(1,4), (1,5), (2,4), (2,3), (3,4), (4,5)}



Neorientovaný graf

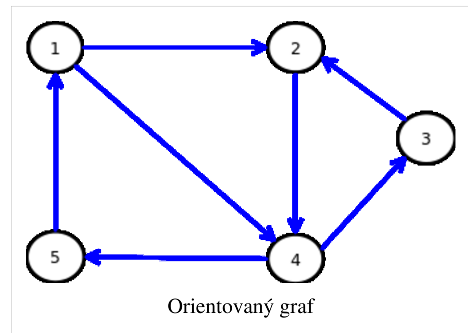
Orientovaný graf

Orientovaný graf alebo **digraf** *G* je usporiadaná dvojica *G* = (*V*, *E*), kde:

- *V* je neprázdna konečná množina **vrcholov** grafu,
- *E* je množina usporiadaných dvojíc typu (*u*, *v*), kde *u* ≠ *v*, nazývaných **orientované hrany** grafu.

Príklad neorientovaného grafu:

V={1,2,3,4,5}
E={(1,2),(1,4), (2,4), (3,2), (4,3), (4,5),(5,1)}



Orientovaný graf

Hranovo ohodnotený graf

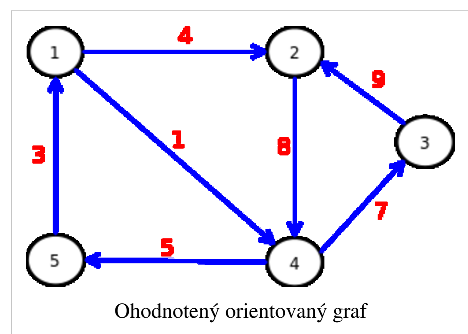
Hranovo ohodnotený graf je graf s ohodnotenými hranami.

Definícia: Graf **G(v,e)** sa nazýva hranovo ohodnotený, ak každej hrane je priradené nejaké číslo .

Definícia: **Kladne hranovo ohodnotený graf** je taký graf **G, w**, že

Príklad ohodnoteného orientovaného grafu:

V={1,2,3,4,5}
E={(1,2,4),(1,4 ,1), (2,4,8), (3,2,9), (4,3,7), (4,5,5),(5,1,3)}

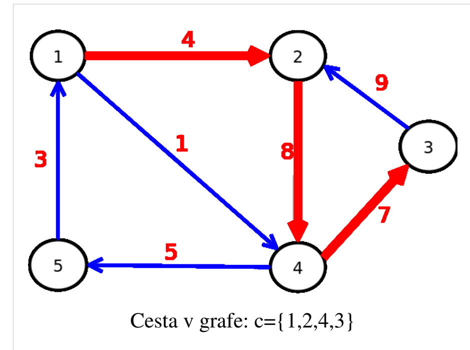


Ohodnotený orientovaný graf

Základné pojmy v teórii grafov

Cesta

je postupnosť vrcholov spojených hranami grafu (Napríklad: c={1,2,4,3})



Cesta v grafe: c={1,2,4,3}

Dĺžka cesty

- neohodnotený graf: je počet hrán v ceste (d=3),
- ohodnotený graf: súčet hodnôt na hraných grafu, ktoré sú v ceste (d=19)

Stupeň vrcholu *V*

počet hrán, vo vrchole *V*

Cyklus

V orientovanom grafe je cesta začínajúca a končiaca v tom istom vrchole a obsahujúca najmenej jednu hranu

Strom

je graf bez cyklov.

Reprezentácia grafov

Matica susednosti

Vzájomné súvislosti hrán a vrcholov sú popísané pomocou matice susednosti M_G.

- Nech G=<V,E>;,
- Matica susednosti M_G grafu *G* je štvorcová matica rádu *m*, kde

Príklad:

Matica susednosti pre neorienotvaný graf (na prvom obrázku)

Matica susednosti pre orienotvaný graf (na druhom obrázku)

Matica susednosti M_G grafu G s ohodnotenými hranami:

Každému vrcholu je priradený zoznam jeho susedov (využitie lineárneho zoznamu).

- Matica susednosti M_G grafu *G* je štvorcová matica rádu *m*, kde

Matica susednosti pre orienotvaný ohodnotený graf (na treťom obrázku)

Implementácia matice susednosti v jazyku C

Úloha:Vytvorte maticu susednosti pre graf *G*. Graf *G* bude zadávaný nasledovne:

prvý bude počet vrcholov (*pocetV*) a potom počet hrán (*pocetH*) grafu *G*. Potom bude nasledovať *pocetH* dvojíc vrcholov + cena hrany *c*.

Vzorový vstup:

```
5 7
1 2 4
1 4 1
2 4 8
3 2 9
4 3 7
4 5 5
5 1 3
```

Analýza úlohy

Vlastnosť matice susednosti neorientovaného grafu je, že daná matica je súmerná vzhľadom na hlavnú diagonálu. Túto vlastnosť môžeme formulovať nasledovne: ak vrchol *i* susedí s vrcholom *j*, tak potom aj vrchol *j* susedí s vrcholom *i*.

Pri orientovaných grafoch takéto tvrdenie nemôžeme povedať, pretože ak existuje iba orientovaná hrana s vrcholu *i* do vrcholu *j*, tak potom neexistuje spôsob ako sa z vrcholu *j* dostať do vrcholu *i*.

Príprava dátovej štruktúry dvojrozmerné pole celých čísel (matica susednosti)

```
 int i,j,pocetV,pocetH,v1,v2,c;
 cin>>pocetV>>pocetH;
 int **Mg=new int*[pocetV];
    for(i=0;i<pocetV;i++)
          Mg[i]=new int[pocetV];

// vynulovanie matice
//Mg predstavuje pr8zdny graf.
for(i=0;i<pocetV;i++)
  for(j=0;j<pocetV;j++)
    Mg[i][j]=0;
```

Vytvorenie orientovaného grafu

```
    for(i=0;i<pocetH;i++)
    {       cin>>v1>>v2>>c;
          Mg[v1-1][v2-1]=c;
    }
```

Vytvorenie neorientovaného grafu

```
    for(i=0;i<pocetH;i++)
    {       cin>>v1>>v2>>c;
          Mg[v1-1][v2-1]=c;
        Mg[v2-1][v1-1]=c;
    }
```

Grafové algoritmy

V súvislosti s grafmi existuje veľa úloh, ktoré sa dajú efektívne riešiť práve pomocou grafov. Napríklad:

- Prechádzanie stromom
- Hľadanie najkratšej cesty
- Prehľadávanie bludiska
- Hľadanie optimálnej cesty
- Ofarbenie grafu - problém ofarbenia politickej mapy sveta
- Toky v sieťach (elektrické schémy)
- Rozdhodovacie stromy
- a mnohé iné[1]

Medzi najzaujímavejšie úlohy patria úlohy o prehľadávaní grafu a hľadaní najkratšej (optimálnej) cesty. Hovoríme o nasledujúcich algoritmoch:

- Prehľadávanie do šírky

- Prehľadávanie do hĺbky
- Hľadanie najkratšej cesty

Referenicie

[1] http://en.wikipedia.org/wiki/Category:Graph_algorithms

Zbierka úloh

Štruktúry (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy

Štruktúra Zlomok

Úloha:

Zostavte program, ktorý pomôže pri práci so zlomkami – bude vedieť zlomok skrátiť a vypísať ho v čo najvhodnejšom tvare. Program načíta dva zlomky, každý hneď po načítaní upravene vypíše, ďalej vypíše ich súčin, súčet a podiel:

1. Zostavte funkciu na načítanie zlomku z klávesnice a funkciu, ktorá vypíše zlomok na obrazovku v tvare „a/b“, napr. „3/4“. Postupne ju zdokonaľujte:
   1. V prípade, že zlomok má celú časť, vypíše ho v tvare „a b/c“, napr. namiesto „9/4“ bude písať „2 1/4“.
   2. V prípade, že jeho zvyšná časť je nulová, nebude ju vypisovať, napr. namiesto „6/2“ nebude písať „3 0/2“, ale iba „3“ a podobne, namiesto „0/2“ iba „0“.
2. Vytvorte si pomocnú funkciu, ktorá zjednoduší (skráti) zlomok, napr. „36/48“ prevedie na „3/4“ a obohaťte ňou funkciu pre vypísanie zlomku (aby sa zlomok vypisoval už v zjednodušenom tvare).
3. Zostavte funkcie, ktoré vypočítajú súčin, súčet a podiel dvoch zlomkov a použite ich v hlavnom programe.

Vzorový vstup:

```
5 4
1 1/4
6 8
3/4
```

Vzorový výstup:

```
sucin: 15/16
sucet: 2
podiel: 1 2/3
```

Cieľom je precvičiť si prácu so štruktúrou – bolo by vhodné ju v programe využiť. Funkcia na načítanie zlomku môže využiť referenciu alebo štruktúru ako návratovú hodnotu. Pre krátenie zlomku môžeme využiť referenciu a pre matematické operácie so zlomkami zase návratovú hodnotu.

Pre krátenie zlomku treba funkciu na najväčší spoločný deliteľ – použijeme Euklidov algoritmus postupného odpočítavania menšieho čísla od väčšieho a jeho zdokonalenie cez operáciu modulo. Pri matematických funkciách môžeme využiť akýsi „akoby konštruktor“ na vytvorenie zlomku, no nie je to nutné.

Možné riešenie:

```
#include <iostream.h>
#include <conio.h>
struct TZlomok { int citatel, menovatel; };
void CitajZlomok(TZlomok &z)
{
   cin >> z.citatel;
   cin >> z.menovatel;
}

int NSD(int a, int b)
{
   while (b)
   {
      int c = a % b;
      a = b;
      b = c;
   }
   return a;
}

void SkratZlomok(TZlomok &z)
{
   int nsd = NSD(z.citatel, z.menovatel);
   z.citatel /= nsd;
   z.menovatel /= nsd;
}

void VypisZlomok(TZlomok z)
{
   SkratZlomok(z);
   int cela_cast = z.citatel / z.menovatel;
   if (cela_cast)
   {
      cout << cela_cast;
      int zvysok = z.citatel % z.menovatel;
      if (zvysok) cout << " " << zvysok << "/" << z.menovatel;
   }
   else
   {
      cout << z.citatel;
      if (z.citatel) // ak je nula, napise len ju a nie napr. 0/1
      cout << "/" << z.menovatel;
   }
```

```
    cout << endl;
}

TZlomok Zlomok(int cit, int men)
{
    TZlomok z;
    z.citatel = cit;
    z.menovatel = men;
    SkratZlomok(z);
    return z;
}

TZlomok SucinZlomkov(TZlomok z1, TZlomok z2)
{
    return Zlomok(z1.citatel * z2.citatel, z1.menovatel * z2.menovatel);
}

TZlomok SucetZlomkov(TZlomok z1, TZlomok z2)
{
    return Zlomok(z1.citatel * z2.menovatel + z2.citatel * z1.menovatel,
    z1.menovatel * z2.menovatel);
}

TZlomok PodielZlomkov(TZlomok z1, TZlomok z2)
{
    return Zlomok(z1.citatel * z2.menovatel, z1.menovatel * z2.citatel);
}

void main()
{
    TZlomok z1, z2;
    CitajZlomok(z1); VypisZlomok(z1);
    CitajZlomok(z2); VypisZlomok(z2);
    cout << "sucin: "; VypisZlomok(SucinZlomkov(z1, z2));
    cout << "sucet: "; VypisZlomok(SucetZlomkov(z1, z2));
    cout << "podiel: "; VypisZlomok(PodielZlomkov(z1, z2));
    getch();
}
```

Zásobník - pamäť typu LIFO

Intepreter postfixových aritmetických výrazov

Zadanie:

Vytvorte program v jazyku C, ktorý po spustení načíta od používateľa postfixový aritmetický výraz, vyhodnotí ho, vypíše jeho hodnotu do konzoly (na monitor) a skončí.

Analýza:

V postfixovom aritmetickom výraze sa operátor zapisuje vždy po svojich dvoch operandoch, ako to môžeme vidieť v nasledujúcom takomto výraze

```
5 9 8 + 4 6 * * 7 + *
```

ktorý má hodnotu 2075. Každý infixový aritmetický výraz, môže byť prepísaný na postfixový. Infixová verzia uvedeného postfixového výrazu je nasledovná:

```
5 * ( ( (9 + 8) * (4 * 6) ) + 7)
```

a tá má samozrejme po vyhodnotení rovnakú hodnotu 2075.

Pomocou abstraktného dátového typu (ADT) zásobníka sa dajú veľmi dobre vyhodnocovať práve postfixové aritmetické výrazy. Každý operátor si z vrcholu zásobníka načíta svoje dva operandy a po vykonaní operácie na nich vráti jej výsledok namiesto týchto dvoch operandov na vrchol zásobníka (využitie princípu dátovej štruktúry typu LIFO). Nasledujúci obrázok demonštruje tento jednoduchý princíp na vyhodnocovaní vyššie uvedeného postfixového aritmetického výrazu.

Spracovanie postfixového výrazu 5 9 8 + 4 6 * * 7 + *

index v pamati zásobníka / číslo kroku	0	1	2	3	4	operácia
1	5					
2	5	9				
3	5	9	8			
4	5					8+9
5	5	17				
6	5	17	4			
7	5	17	4	6		
8	5	17				6 * 4
9	5	17	24			
10	5					24*17
11	5	408				
12	5	408	7			
13	5					408 + 7
14	5	415				
15						415 * 5
16	2075					

Tabuľka ukazuje použitie zásobníka reprezentovaného celočíselným poľom (zasobnik.pamat[ ]) pre vyhodnotenie postfixového výrazu 5 9 8 + 4 6 * * 7 + * uloženého v znakovom poli (vyraz[ ]). Tento výraz postupne prechádzame zľava doprava. Ak narazíme na znak reprezentujúci číslo, vložíme ho na vrchol zásobníka (zasobnik.pamat[zasobnik.vrchol]), ak narazíme na znak reprezentujúci operátor, potom vyberieme z vrcholu zásobníka 2 operandy, aplikujeme na ne operátor a namiesto týchto dvoch operandov vložíme na vrchol zásobníka výsledok samotnej operácie. Na reprezentáciu zásobníka použijeme štruktúru LIFO. Pre základné operácie so zásobníkom, vloženie položky na vrchol zásobníka a vybratie položky z jeho vrcholu, si vytvoríme dve funkcie:

```
  stav zasobnika push(LIFO &zasobnik, int x);
  int pop(LIFO &zasobnik);
```

Príklad vstupu do programu:

5 9 8 + 4 6 * * 7 + *

Príklad výstupu z programu:

2075

Možné riešenie:

```
#include<iostream>
#include <string.h>
using namespace std;

#define MAX_PAMAT 40

enum stav_zasobnika{OK=-1, FULL=-2, EMPTY=-3};

struct LIFO
{
      int pamat[MAX_PAMAT]; //pole reprezentujuce ADT zasobnik
      int vrchol; //index vrcholu zasobnika, indexova premenna

};

//VLOZI novu polozku 'x' do pola 'zasobnik.pamat' (do zasobnika) a
zvacsi indexovu premennu //'zasobnik.vrchol' o 1
stav_zasobnika push(LIFO &zasobnik, int x)
{
      if(zasobnik.vrchol<MAX_PAMAT)
            zasobnik.pamat[zasobnik.vrchol++]=x;
      else
            return FULL;
      return OK;
}

//zmensi indexovu premennu 'zasobnik.vrchol' o 1 a VYBERIE poslednu
vlozenu polozku z pola //'zasobnik.pamat' (zo zasobnika)
int pop(LIFO &zasobnik)
{
      if(zasobnik.vrchol>0)
```

```
            return zasobnik.pamat[--zasobnik.vrchol];
      else
            return EMPTY;
}

int main()
{
      LIFO f; //deklaracia strukturovej premennej typu 'LIFO'
reprezentujucej zasobnik
      f.vrchol=0;
      int X, i;
      char vyraz[40];

      //cout<<"Vlozte postfixovy vyraz na vyhodnotenie:\n";
      cin.getline(vyraz,39)

      X=(int)strlen(vyraz); //v 'X' je dlzka nacitaneho retazca zo
standardneho vstupu
      //v cykle 'for' prechadzame vstupny vyraz ulozeny v poli 'vyraz'
zlava doprava. Ak narazime na znak
      //reprezentujuci cislo, vlozime ho na vrchol zasobnika
('f.pamat[f.vrchol]'), ak narazime na znak
      //reprezentujuci operator, potom vyberieme z vrcholu zasobnika 2
operandy, aplikujeme na ne
      //operator a namiesto tychto dvoch operandov vlozime na vrchol
zasobnika vysledok samotnej
      //operacie

      for(i=0; i<X; i++)
      {
            if (vyraz[i]=='+')
                  push(f, (pop(f) + pop(f)));
            if (vyraz[i]=='*')
                  push(f, (pop(f) * pop(f)));
            if ((vyraz[i]>='0') && (vyraz[i]<='9'))
                  push(f, 0);

            //vlozenie cisla (v datovom type 'int') reprezentovaneho
jednym alebo viacerymi znakmi v poli
            //'a' na vrchol zasobnika
            while ((vyraz[i]>='0') && (vyraz[i]<='9'))
                  push(f, (10 * pop(f) + (vyraz[i++] - '0')));
            //obsluzenie //ziskanie cisla typu 'int' reprezentovaneho
            //viaccifernych cis. //znakom ulozenym v 'a[i]' a
zvascsenie 'i' o 1
            //pre posun na dalsiu iteraciu cyklu 'while'
     }
```

```
      //na vrchole zasobnika je uz vysledok vyhodnoteneho postfixoveho
aritmetickeho vyrazu, takze ho
      //vypiseme
      cout<<pop(f);
      return 0;
}
```

Rekurzia (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy

Najväčší spoločný deliteľ

Zadanie Nájdite rekurzívny vzťah v Euklidovom algoritme pre výpočet najväčšieho spoločného deliteľa a využite ho vo funkcii, ktorá bude počítať najväčší spoločný deliteľ dvoch čísel, uvedených ako jej parametre. Funkciu použite v programe, ktorý načíta dve čísla a napíše hodnotu ich najväčšieho spoločného deliteľa.

Poznámka: V Euklidovom algoritme môžete namiesto klasického odpočítavania využiť zvyšok po delení, čo zníži počet krokov výpočtu a nebude potrebné sa zaoberať problémom „nesprávneho poradia“ čísel pri odpočítavaní.

Vzorové príklady

vstup		výstup
36 48	->	12
576 284	->	4
2553 7215	->	111

Analýza riešenia

Hľadaný rekurzívny vzťah je vidieť v postupe prevodu čísel postupným „modulovaním“ (operáciou modulo) – delenec a je nahradený deliteľom b, deliteľ b zase výsledkom a%b, ukončenie je pri nulovom b:

- NSD(a, b) = NSD(b, a%b) pre b>0,
- NSD(a, 0) = a

Možné riešenie:

```
#include <iostream.h>
#include <conio.h>
int NSD(int a, int b)
{
```

```
   if (b == 0) return a;
   return NSD(b, a%b);
}

void main()
{
   int a, b;
   cin >> a >> b;
   cout << NSD(a, b);
   getch();
}
```

Fibonacciho postupnosť

Zadanie Pre Fibonacciho postupnosť platí, že hodnota ďalšieho jeho prvku je súčtom dvoch predchádzajúcich. Zostavte program, ktorý bude z klávesnice čítať čísla n, kým nenačíta nulu a pre každé číslo n vypíše n-té Fibonacciho číslo v poradí, za predpokladu, že Fib(0) = 0 a Fib(1) = 1.

Vzorové príklady

vstup		výstup
4	->	3
9	->	34
20	->	6765
40	->	102334155
0	->	

Analýza riešenia

Fibonacciho postupnosť je rekurzívne definovaná ako:

- fib(0)=0
- fib(1)=1
- fib(i)=fib(i-1)+fib(i-2), pre i>1

Na tomto príklade je vhodné demonštrovať, čo sa stane, ak zabudneme v rekurzívnej funkcii uviesť podmienku pre ukončenie rekurzie – program havaruje kvôli pretečeniu zásobníka.

Možné riešenie:

```
#include <iostream.h>
long fib(int n)
{
  if (n < 2) return n;
  else return fib(n-1) + fib(n-2);
}
void main()
{
   int n;
   cin >> n;
   while (n)
   {
```

```
      cout << fib(n) << endl;
      cin >> n;
   }
}
```

Prevod čísel z 10-vej sústavy

Zadanie

Zostavte program, ktorý bude prevádzať prirodzené čísla do ľubovoľných číselných sústav so základom Z<10, využitím rekurzívnej funkcie. Túto funkciu postupne zdokonaľujte:

1. Funkciu vylepšite, aby vedela prevádzať aj do sústav so základom Z<=16.
2. Upravte funkciu tak, aby vedela prevádzať všetky celé čísla (čiže aj záporné a nulu).
3. Pokúste sa funkciu obohatiť o prevod reálnych čísel (čiže aj desatinných).

V programe načítajte 2 vstupné údaje: číslo N v 10-vej sústave a základ novej sústavy z.

Vzorové príklady

vstup		výstup
80 2	->	1010000
93 16	->	5D
0 8	->	0
-74 4	->	-1022
3.141592654 16	->	3.243F6A8A48AA
-0.1 2	->	-0.0001100110011001100

Je vhodné začať jednoduchou funkciou na prevod prirodzeného čísla. Ak máme základ cieľovej sústavy z<10, môžeme problém vyjadriť aj rekurentným vzťahom v matematickom tvare: P(n, z) = P(n/z, z) * 10 + n%z (pre n > 0), a teda by bolo možné vytvoriť funkciu, ktorej návratovou hodnotou by bolo celé číslo.

V prípade, že uvažujeme o vyššom základe, vo výsledku sa objavia aj symboly A, B, C, ... funkcia by už musela výsledok vracať vo forme reťazca. Pre zjednodušenie nám stačí funkcia, ktorá bude výsledok priamo vypisovať. Je treba si uvedomiť, ako sa počíta prevod čísla – číslo vydelíme základom sústavy, tento podiel prevedieme a za ním bude nasledovať zvyšok po pôvodnom delení.

Na vypísanie zvyšku pre vyššie sústavy by sme mohli pre zvyšky väčšie ako 9 použiť aj inkrementáciu znakového typu:

```
cout << char ('A' + n%z – 10);
```

Keďže základ sústavy sa počas celého prevodu samozrejme nemení, nemá význam v každom volaní funkcie vytvárať jeho kópiu v podobe vstupnej premennej, ale stačí nám referencia (ušetrí sa pár bajtíkov pri každom vnorení).

Ak chceme, aby funkcia dokázala prevádzať aj nulu a záporné čísla, musíme si pridať akúsi „úvodnú" funkciu, ktorá ošetrí problematické situácie a až potom zavolá hlavnú rekurzívnu funkciu prevodu.

Možné riešenie:

```
#include <iostream.h>
#include <conio.h>
const char znaky[] = "0123456789ABCDEF";
void PrevodCele(int n, int &zaklad)
{
```

```
    if (n == 0) return;
    PrevodCele(n/zaklad, zaklad); // prevedie celu cast podielu
    cout << znaky[n%zaklad]; // za tym napise zvysok
}

void PrevodReal(double r, int &zaklad, int presnost)
{
    if (r == 0) return;
    r *= zaklad; // posunie o jeden rad vlavo
    cout << znaky[int(r)]; // celu cast vypise
    PrevodReal(r - int(r), zaklad, presnost - 1); // zvysok prevedie
}

// uvodna funkcia na specialne pripady
void Prevod(double r, int zaklad, int presnost)
{
    // ak je zaporne
    if (r < 0)
    {
        r = -r; cout << '-';
    }
    // cela cast
    int n = int(r);
    if (n)
        PrevodCele(n, zaklad);
    else
        cout << 0; // ak je nula, vypise ju

    // desatinna cast (ak je)
    if (r -= n)
    {
        cout << '.';
        PrevodReal(r, zaklad, presnost);
    }
}

void main()
{
    double r;
    int sustava;
    cout << "cislo v 10-kovej sustave: "; cin >> r;
    cout << "cielova sustava: "; cin >> sustava;
    cout << "prevedene: ";
    Prevod(r, sustava, 6);
    getch();
}
```

Prvočísla

Zadanie:

Pomocou rekurzívneho riešenia vytvorte funkciu, ktorá otestuje, či je dané číslo prvočíslo.

Prvé riešenie - neoptimalizované

Analýza riešenia:

Testujeme číslo n na prvočíselnosť. Číslo n budeme postupne deliť číslami od 2 až po (n-1) aby sme zistili zvyšok po delení. Pri prvom zvyšku, ktorý je rovný 0 (teda číslo z delí číslo n bezo zvyšku) funkciu ukončíme a vrátime hodnotu 0 (n nie je prvočíslo). Ak je z<(n-1) tak funkciu is_prime1 voláme rekurzívne a v tomto volaní zvýšime parameter z o 1. (riadok č. 8). V prípade, že neplatí rovnosť z<(n-1) a ani jedno číslo z intervalu <2,n-1> nedelí číslo n bezo zvyšku, vrátime hodnotu 1 (n je prvočíslo).

Možné riešenie:

```
int is_prime1(int n, int z=2)
{
   if((n%z)==0)
      return 0;
   else
   {
      if(z<(n-1))
         return is_prime1(n,z+1);
      else
         return 1;
   }
}
```

Druhé riešenie - čiastočne optimalizované

Optimalizácia v tomto prípade znamená eliminovať počet delení modulo. Myšlienka nájdenia hornej hranice hodnoty premennej *z* z predchádzajúceho príkladu:

- Číslo n delíme modulo hodnotou 2. (výsledok je rôzny od 0)
  - nemá zmysel deliť číslom *z* väčším ako je n/2, pretože už nám nemôže vyjsť celočíselný podiel. Výsledok takéhoto delenia bude v intervale (0,1)
- Číslo n delíme modulo hodnotou 3. (výsledok je rôzny od 0)
  - nemá zmysel deliť číslom *z* väčším ako je n/3, pretože už nám nemôže vyjsť celočíselný podiel. Výsledok takéhoto delenia bude v intervale . Podiel nebude mať hodnotu 2, pretože v tom prípade by bolo číslo n delitelné číslom 3 bezo zvyšku.
- Delitel n budeme zväčšovať až pokiaľ platí

Horná hranica hodnoty d sa dá určiť nasledovne:

Možné riešenie:

```
int is_prime2(int n, int z=2)
{
   if((n%z)==0)
      return 0;
   else
   {
```

```
        if(z<sqrt(n))
            return is_prime2(n,z+1);
        else
            return 1;
    }
}
```

Tretie riešenie - viac optimalizované

V predchádzajúcom príklade sme delili číslo n hodnotami premennej d. Premenná d mala prvú hodnotu 2 a potom sa k nej vždy pripočítala hodnota 1. Teda, číslo n sme postupne delili hodnotami

```
2, 3, 4, 5, 6, 7, 8, ... .
```

Všimnime si, že niektoré delenia sú opäť zbytočné:

- ak delíme číslom 2, potom nemá zmysel deliť žiadnym jeho násobkom
- vo všeobecnosti platí: ak delíme číslom *j*, potom nemá zmysel deliť žiadnym násobkom čísla *j*.

Inak povedané, netreba deliť žiadnom zloženým číslom. Stačí deliť prvočíslami. A tu sa dostávame k rekurzívnej definícii prvočísel:

Rekurzívna definícia prvočísla:

- Číslo 2 je najmenšie prvočíslo.
- Prvočíslo je každé celé kladné číslo, ktoré nie je deliteľné žiadnym iným menším prvočíslom ako je toto číslo samotné.

Pri takto stanovenej podmienke je komplikovanejšie vytvoriť podmienku, ktoré by toto spĺňala. Preto skúsme napísať program, ktorý by zbytočne nedelil násobkami čísel 2 a 3. Ak nechceme deliť násobkami čísla 2, tak potom budeme postupne deliť premennú n hodnotami 3,5,7,9,11,...

Ak nechceme deliť násobkami čísla 3, tak potom budeme postupne deliť premennú n hodnotami 4,5,7,8,10,11,13,14,...

Prienikom týchto dvoch podmienok je, že nám zostava deliť číslami 5,7,10,13,17,19, ...

V tejto postupnosti je jednoduché pravidlo: striedavé pripočítavanie hodnoty 2 a 4. Vyjadrené aritmetickou postupnosťou:

Možné riešenie:

```
int is_prime3(int n, int z,int krok)
{
    int d=z+3+krok;
    if((n%d)==0)
        return 0;
    else
    {
        if(d<sqrt(n))
            return is_prime3(n,d,-krok);
        else
            return 1;
    }
}

int is_prime(int n)
```

```
{
   if((n%2)==0) return 0;
   if((n%3)==0) return 0;
   if((n%5)==0) return 0;
   return is_prime3(n,5,-1);
}
```

Rozbor zdrojového kódu:

Pre hľadanie prvočísel použijeme funkciu is_prime, ktorá otestuje číslo n na deliteľnosť číslami 2, 3 a 5. Potom zavoláme pomocnú rekurzívnu funkciu is_prime3(int n, int z,int krok) ktorá bude testovať deliteľnosť čísla n prvkami postupnosti . Aby sme sa vyhli zbytočným podmienkam, ktoré môžu celý algoritmus spomaliť, definujeme tretí parameter funkcie is_prime3 *krok*. Parameter krok je v našej postupnosti výraz , avšak tu nepočítame hodnotu mocniny ale využívame skutočnosť že tento výraz nadobúda hodnoty -1,1,-1,1, ...

V premennej d je vypočítaná hodnota aktuálneho deliteľa podľa vzťahu: .

Riešenie s globálnym poľom

Pre zjednodušenie tvorby programu si vytvorme globálne pole do ktorého uložíme prvých n prvočísel. Potom najväčšie číslo, ktoré môžeme testovať na prvočíselnosť je .

Analýza riešenia:

Testujeme číslo n na prvočíselnosť. Ako prvé vypočítame zvyšok po delení prvým prvočíslom v poli *pcisla* (delíme číslom pcisla[0]=2). V prípade, ak je výsledok 0, tak výpočet ukončíme a vrátime hodnotu (0 - nie je prvočíslo). V opačnom prípade rekurzívne zavoláme funkciu jePrvocislo, kde druhý parameter (má význam indexu v poli *pcisla*) zvýšime o 1. Teda v ďalšom volaní funkcie jePrvocislo už budeme deliť číslom pcisla[1]=3. Musíme však zabezpečiť, aby hodnota pcisla[i] v riadku č. 5 existovala. To zabezpečíme tak, že rekurzívne volanie dovolíme len ak je hodnota indexu i menšia ako počet hodnôt v pole pcisla.

Možné riešenie:

```
int prvocisla[]={2,3,5,7,11,13,17,19,23,29,31,37};

int is_prime4(int n, int i=0)
{
  if(n%prvocisla[i]==0)
     return 0;
  if(i<223 && n>pow(prvocisla[i],2))
     return is_prime4(n,i+1);
  else
     return 1;
}
```

Vzorové riešenie

Uvádzame postupnosť volaní funkcie **is_prime4(31)**

Vnorenie	Volaná funkcia	prvocisla[i]	n%prvocisla[i]	rekurzívne volanie
0	is_prime(31,0)	2	31%2=1	is_prime(31,1)
1	is_prime(31,1)	3	31%3=1	is_prime(31,2)
2	is_prime(31,2)	5	31%5=1	is_prime(31,3)
3	is_prime(31,3)	7	31%7=3	is_prime(31,4)
4	is_prime(31,4)	11	n.a.	neplatí podmienka n>prvocisla[i]^2 n=31, i=4, prvocisla[i]=11 funkcia vracia hodnotu **1**
3	návrat na is_prime(31,3)	7	n.a.	spätný rekurzívny chod
2	návrat na is_prime(31,2)	5	n.a.	spätný rekurzívny chod
1	návrat na is_prime(31,1)	3	n.a.	spätný rekurzívny chod
0	návrat na is_prime(31,0)	2	n.a.	spätný rekurzívny chod

Porovnanie efektivity navrhovaných algoritmov

Skôr ako budeme porovnávať ukázané algoritmy, poznamenajme, že tento spôsob zisťovania prvočíselnosti je pre veľké čísla veľmi neefektívny. Pri 8-cifernom čísle trvá výpočet funkcie is_prime približne 230 s. Pre zisťovanie prvočíselnosti existujú špeciálne algoritmy, napr. ASK test [1], Test Solovay-Strassena [2], Test Millera-Rabina [3] a iné. V nasledujúcej tabuľke a na obrázku sú výsledky porovnania efektívnosti týchto algoritmov pri hľadaní prvočísel na intervale (100,n) kde n sme menili od 50 000 do 2 000 000. V tabuľke ani v grafe nie je zahrnutá funkcia is_prime1, pretože pri n=100 000 pri rekurzívnom volaní funkcia neočakávane skončila. Aby sme mohli porovnávať aj funkciu is_prime4, musíme do poľa *prvocisla* pridať všetky prvočísla, ktoré sú menšie ako , kde n_max je v našom prípade 2 000 000. Pole *prvocisla* obsahuje prvých 233 prvočísel:

```
int prvocisla[]={

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,

191,193,197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,313,317,331,337,347,349,353,359,367,373,379,383,389,

397,401,409,419,421,431,433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,563,569,571,577,587,593,599,601,607,613,

617,619,631,641,643,647,653,659,661,673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,821,823,827,829,839,853,

857,859,863,877,881,883,887,907,911,919,929,937,941,947,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,1051,1061,1063,

1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,1163,1171,1181,1187,1193,1201,1213,1217,1223,1229,1231,1237,1249,1259,1277,1279,

1283,1289,1291,1297,1301,1303,1307,1319,1321,1327,1361,1367,1373,1381,1399,1409,1423};
```

Naprogramované funkcie sme testovali nasledovne:

```
#include <iostream.h>
#include <time.h>

int main()
{
   clock_t start, end;
   double elapsed;
   int n;
   n=50000; //100000, 200000, ... 2000000
   start = clock();
   for(int i=100;i<n;i++)
      is_prime(i);
```

```
    end = clock();
    elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;
    cout<<elapsed<<endl;
}
```

Tabuľka nameraných časov behu funkcií

n [tisíc]	is_prime_2 t[s]	is_prime t[s]	is_prime4 t[s]
50	0,16	0,086	0,061
100	0,398	0,134	0,126
200	1,034	0,345	0,304
500	1,245	1,237	0,958
1000	3,275	3,264	2,349
1500	17,203	5,831	3,9
2000	26	8,6	5,82

<pLines ymin=0 ymax=27 colors=FF0000,00FF00,0000FF size=600x250 plots legend> ,is_prime2, is_prime, is_prime4 50 tis, 0.16, 0.08, 0.061 100 tis, 0.398, 0.134, 0.126 200 tis, 1.034, 0.345, 0.304 500 tis, 1.245, 1.237, 0.958 1 mil, 3.275, 3.264, 2.349 1.5 mil, 17.203, 5.831, 3.9 2 mil, 26, 8.6, 5.82 </pLines>

Podobné časy pri nižších hodnotách n sú dané tým, že vzdialenosť susedných prvočísel je väčšia pri vyšších prvočíslach. Z grafu taktiež vidieť účinok optimalizácie algoritmu.

Odkazy

- http://sk.wikipedia.org/wiki/Zoznam_prvo%C4%8D%C3%ADsiel
- http://www.prime-numbers.org/

Dynamická alokácia pamäti (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy

Filtrovanie textu

Zadanie úlohy

Zostavte program na filtrovanie textu - vyhľadávanie riadkov s hľadaným slovom. Program z klávesnice načíta hľadané slovo (nebude mať viac ako 100 znakov). Ďalej bude čítať z klávesnice riadky textu (reťazce ukončené ENTERom), až kým nenačíta prázdny riadok. Môžeme predpokladať, že žiaden riadok nebude dlhší ako 1000 znakov a že riadkov nebude viacej ako 1000.

Po načítaní prázdneho riadku program začne na obrazovku vypisovať výsledky (do tohoto okamihu NESMIE nič písať na obrazovku) v tvare:

V prvom riadku bude text "nachadza N:", kde N je počet riadkov, v ktorých sa nachádza hľadané slovo.

- Pod týmto riadkom budú vypísané všetky riadky, v ktorých sa nachádza hľadané slovo.
- Pod nimi bude prázdny riadok a ďalej riadok s textom "nenachadza N:", kde N je počet riadkov, v ktorých sa hľadané slovo nenachádza.
- Pod týmto riadkom budú vypísané všetky riadky, v ktorých sa nenachádza hľadané slovo.

Vzorový príklad

Príklad vstupu

```
slovo
Program bude hladat riadky s urcitym slovom.
V tomto riadku sa nenachadza.
A ani v tomto dalsom

```

Príklad výstupu:

```
nachadza 1:
Program bude hladat riadky s urcitym slovom.
nenachadza 2:
V tomto riadku sa nenachadza.
A ani v tomto dalsom.

```

Netreba zabúdať na to, že každý textový reťazec je ukončený znakom '\0', preto musíme rezervovať miesto aj preň.

Pre načítavanie hľadaného slova je vhodné použiť príkaz cez cin.getline, pretože cin>> akosi necháva ENTER za slovom nepovšimnutý a prvý cin.getline je potom prázdny.

Analýza riešenia

Keďže v tejto úlohe vopred nevieme, koľko riadkov bude obsahovať, je vhodné pole riadkov vytvoriť staticky – je ich, podľa zadania, nanajvýš 1000. Riadky samotné by však statickébyť nemali – je zbytočné mať maticu 1000×1001 znakov (čo predstavuje 1 MB dát), keď väčšina riadkov bude oveľa kratších. Riadok preto najskôr načítame do pomocnej premennej (textový reťazec dostatočnej dĺžky), „zmeriame“ jeho dĺžku a dynamicky vytvoríme riadok, do ktorého sa vojde (pozor, musíme dať dĺžku o jedno väčšiu).

Úlohu je možné riešiť použitím jedného poľa riadkov, no potom je problém s určením, v ktorom riadku sa hľadané slovo nachádza a v ktorom nie – dá sa to riešiť dvojako:

1. hneď pri čítaní vstupu budeme počítať riadky s hľadaným slovom a riadky bez neho, aby sme toto mohli vopred vypísať, potom druhýkrát musíme prechádzať poľom a vypisovať riadky, v ktorých sa hľadané slovo nachádza a nakoniec tretíkrát ním prechádzať a opäť hľadať riadky, v ktorom sa slovo nenachádza – toto riešenie nie je veľmi efektívne, keďže 3-krát hľadáme to isté (a hľadanie slova v reťazci je relatívne náročná operácia),
2. do pomocného poľa si budeme hneď pri čítaní vstupu zapisovať, či v danom riadku je alebo nie je hľadané slovo (hodnotou 1 alebo 0) a zároveň ich počítať zvlášť do počítadiel – pri vypisovaní stačí pozerať do tohto poľa a netreba znovu hľadať slovo v reťazcoch.

Praktickejšie však bude ukladať si riadky do dvoch polí – zvlášť tie, v ktorých sa hľadané slovo nachádza a zvlášť tie, kde nie. Keďže ale nevieme, koľko ktorých je, musíme počítať s najhoršou možnosťou pre obe polia – že riadkov bude plný počet (1000). Či sa hľadané slovo v riadku nachádza, nám pomôže zistiť funkcia strstr [1], na kopírovanie textu použijeme funkciu strcpy [2]. **V žiadnom prípade nemôžeme použiť len jednoduché priradenie, lebo by sa skopíroval len ukazovateľ na reťazec a nie reťazec samotný!**. Keď už dynamicky vytvorené reťazce nepotrebujeme, je vhodné ich zmazať – po skončení programu sa síce vymažú automaticky, no je dobré zvyknúť si "upratať po sebe".

Riešenie v jazyku C

```
#include <iostream.h>
#include <string.h>
#include <conio.h>
int main()
{
   char text[1001], hladane[101], *najdene[1000], *nenajdene[1000];
   int pocet_najdenych = 0, pocet_nenajdenych = 0;
   cin.getline(hladane, 100);
   cin.getline(text, 1000);
   int dlzka = strlen(text);
   while (dlzka)
   {
      if (strstr(text, hladane)) // nachadza sa
      {
         najdene[pocet_najdenych] = new char[dlzka + 1];
         strcpy(najdene[pocet_najdenych], text);
         pocet_najdenych++;
      }
```

```
      else // nenachadza sa
      {
         nenajdene[pocet_nenajdenych] = new char[dlzka + 1];
         strcpy(nenajdene[pocet_nenajdenych], text);
         pocet_nenajdenych++;
      }
      cin.getline(text, 1000);
      dlzka = strlen(text);
   }
   int i;
   cout << "nachadza " << pocet_najdenych << ":\n";
   for (i = 0; i < pocet_najdenych; i++)
   {
      cout << najdene[i] << endl;
      delete[] najdene[i];
   }
   cout << "\nnenachadza " << pocet_nenajdenych << ":\n";
   for (i = 0; i < pocet_nenajdenych; i++)
   {
      cout << nenajdene[i] << endl;
      delete[] nenajdene[i];
   }
   getch();
}
```

Práca so špeciálnymi maticami

LU rozklad

V lineárnej algebre existuje metóda riešenia maticových rovníc nazvaná LU rozklad[1][2]. Podstatou tejto metódy je rozložiť štvorcovú maticu na hornú a dolnú trojuholníkovú maticu:

(vzťah 1)

Kde matice L a U sú:

(vzťah 2 a 3)

Dôvod prečo sa tento rozklad robí je, že operácie s trojuholníkovými maticami sú jednoduchšie ako so štvorcovými.

Úloha

Vytvorte program, v ktorom načítate 2 trojuholníkové matice (maticu L a maticu U) a tieto matice vynásobte. Takto získame pôvodnú maticu A.

Vstupné údaje

V programe načítajte rozmer matíc –n. Následne načítajte maticu L. Počet prvkov pozostáva z prvkov. Prvý načítaný prvok je 1, ďalší 1, … ,1 (pozri vzťah 3). Ako druhú maticu načítajte maticu U. Matica U má taktiež prvkov. Prvý prvok je , následne až , , ...,

Príklad vstupu:

```
4
1 2 1 3 4 1 5 6 7 1
2 4 3 5 4 6 5 7 6 9
```

Matice majú nasledovný tvar:

resp.

Výstup pre vzorový príklad:

Analýza problému

Pri vytváraní vhodnej štruktúry reprezentujúcu trojuholníkovú maticu budeme vychádzať so skutočného tvaru matice. Vytvoríme maticu, ktorá má v prvom riadku 1 prvok, v druhom 2 prvky a v n-tom riadku bude mať n prvkov. Vytvoriť takúto maticu môžeme pomocou dynamickej alpkácie pamäti nasledujúcim princípom:

```
int **A=new int*[n];
for(int i=0;i<n;i++)
   A[i]=new int[i+1];
```

A sme si definovali ako dvojitý smerník na int. Alebo inak povedané pole smerníkov o veľkosti n. Pre každý smerník v tomto poli alokujeme pamäť o veľkosti (i+1) int.

Pre násobenie matíc môžeme vytvoriť efektívnejší algoritmus ako pri násobení štvorcových matíc. Pri násobení štvorcových matíc (pri násobení Z=XY) je prvok výslednej matice rovný:

Pri násobení matíc LU môžeme tento vzorec modifikovať na

Túto modifikáciu je možné urobiť, pretože pri násobeniach, ktoré sme odstránili bolo násobenie nulou. Okrem iného, v našej implementácii matíc L a U nemôžeme použiť pôvodný vzorec, pretože matice sú trojuholníkové – v matici L neexistujú prvky nad hlavnou diagonálou a v matici U neexistujú prvky pod hlavou diagonálou.

V našom zdrojovom kóde vytvoríme funkciu sucinLU, ktorá bude násobiť matice L a U.

Riešenie v jazyku C

```
#include<iostream>
#include<math.h>

using namespace std;
void sucinLU(int **XL, int **XU, int **XA, int n)
{
   int i,j,k;
   for(i=0;i<n;i++)
     for(j=0;j<n;j++)
     {  XA[i][j]=0;
        for(k=0;k<min(i+1,j+1);k++)
             XA[i][j]+=XL[i][k]*XU[k][j-k];
     }
}

void vypis(int **XA, int n)
{
   int i,j;
   for(i=0;i<n;i++)
```

```
    {
       for(j=0;j<n;j++)
         cout<<XA[i][j]<<"\t";
       cout<<endl;
    }
}
int main()
{
    int n,i,j;
    cin>>n;

    // L - dolna trojuholnikova matica
    int **L=new int*[n];
    for(i=0;i<n;i++)
       L[i]=new int[i+1];

    // U - horna trojuholnikova matica
    int **U=new int*[n];
    for(i=0;i<n;i++)
       U[i]=new int[n-i];

    // A = LU
    int **A=new int*[n];
    for(i=0;i<n;i++)
       A[i]=new int[n];

    //nacitanie matice L
    for(i=0;i<n;i++)
       for(j=0;j<i+1;j++)
          cin>>L[i][j];

    //nacitanie matice U
    for(i=0;i<n;i++)
       for(j=0;j<n-i;j++)
          cin>>U[i][j];

    sucinLU(L,U,A,n);

    vypis(A,n);
}
```

Zaujímavosťou v zdrojovom kóde je násobenie matíc L a U, konkrétne riadok 12, keď namiesto očakávaného výrazu:

```
XA[i][j]+=XL[i][k]*XU[k][j];
```

je riadok

```
XA[i][j]+=XL[i][k]*XU[k][j-k];
```

Dôvod: Matica U je horná obdĺžniková matica, ktorá má tvar:

My ju však reprezentujeme nasledovne:

V tejto matici tvoria stĺpce nasledovné čísla

1. 2
2. 4, 4
3. 3, 6, 7
4. 5, 5, 6, 9

Zdroje a odkazy

[1] LU metoda http://kfe.fjfi.cvut.cz/~limpouch/numet/linalg/node9.html
[2] Priame metódy - http://www.math.sk/skripta/node73.html

Vyhľadávanie (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy

Keno 10

Úloha

Riešte problém kontroly tipovaných čísel v hre Keno. KENO 10 je číselná lotéria kenového typu, pri ktorej tipujúci tipuje 1 až 10 čísel z osemdesiatich čísel od 1 do 80. Pri každom žrebovaní je vyžrebovaných 20 výherných čísel. Vytvorte program pre začínajúcu stávkovú kanceláriu, ktorý overí koľko čísle uhádol tipujúci. Vstup do programu bude 20 vyžrebovaných čísel (čísla sú usporiadané od najmenšieho po najväčšie) a následne sa načítajú tipy tipujúceho: teda 1 až 10 čísel. Posledné číslo je 0.

Vstup:

,

,

kde 1<=n<=10

Výstup

počet čísel, ktoré tipujúci uhádol.

Analýza úlohy

Pre potreby vyh2ad8vanie použijeme rekurzívnu verziu binárneho vyhľadávania.

Riešenie v jazyku C

```
#include<iostream.h>
int hladajB(int *pole,int lavy, int pravy, int x)
{
   if(lavy>pravy)
      return -1;
   else
   {
      int stred=(lavy+pravy)/2;
      if(pole[stred]==x)
         return stred;
      else
      {
         if( x<pole[stred] )
            return hladajB(pole,lavy,stred-1,x);
         else
            return hladajB(pole,stred+1,pravy,x);
      }
   }
}

int main()
{
   int i,pocet=0,cisla[20],tip;
   for(i=0;i<20;i++)
     cin>>cisla[i];

   for(i=0;i<10;i++)
   {  cin>>tip;
     if(!tip) break;
     else
        if(hladajB(cisla,0,19,tip)!=-1)
          pocet++;
   }
   cout<<pocet;
}
```

Vyhľadávanie v usporiadanom zozname

Úloha

V logovacom súbore webového servera sú údaje o prístupe na niektorú z poskytovaných web stránok. Záznam v logu vyzerá asi takto:

```
127.0.0.1 - - [13/Mar/2009:12:55:30 +0100] "GET / HTTP/1.1" 200 4075
```

Jednotlivé časti takéhoto záznamu majú nasledujúci význam:

1. IP adresa počítača z ktorého sa pristupovalo. Pozostáva zo 4 čísel oddelených bodkou.
2. Oddelovací znak: "- -"
3. Dátum a čas udalosti uzatvorený do hranatých zátvoriek
    1. Časti dátumu sú oddelené lomkou „/“,
        - deň je vyjadrený 2 číslicami (02, 12, ...)
        - Mesiac je vyjadrený 3 znakmi
        - Rok je vyjadresný 4 číslicami
    2. čas je oddelený dvojbodkou „:“
        - Hodina, minúta aj sekunda je vyjadrená 2 číslicami
4. žiadaná stránka

Úlohou bude zistiť, či v určitú dobu nastala nejaká udalosť. Ak udalosť nastala, zaujíma nás v ktorom riadku súboru je o tom záznam.

Jeden riadok súboru má maximálne 500 znakov.

Analýza úlohy

Webový server ukladá záznamy do súboru postupne, pričom každý nový pridaný záznam má hodnotu času väčšiu ako záznam pred ním. Môžeme teda povedať že tento súbor je usporiadaný vzhľadom na čas udalosti. Keďže sú vstupné dáta usporiadané môžeme použiť algoritmus binárneho vyhľadávania (je možné použiť aj algoritmus lineárneho vyhľadávania, ale bolo by to neefektívne).

Prvou úlohou je spracovať vstupné údaje. Vstupné údaje sú uložené v súbore (predpokladajme súbor access.log [1]). V súbore treba prečítať všetky riadky. Toto jednoducho implementujeme tak, že budeme zo súboru čítať až potiaľ, pokiaľ neprečítame všetky riadky.

Predpokladajme, že v súbore nie je viac ako 1000 riadkov.

Návrh vhodnej dátovej štruktúry

Súbor budeme čítať po riadkoch, ale pri ukladaní načítaných údajov sa budeme snažiť čo najviac minimalizovať pamäťové nároky. Podľa zadania úlohy si nemusíme pamätať stránku ktorá je v danom riadku, zaujíma nás len čas a dátum prístupu. Vytvorme si teda dátovú štruktúru reprezentujúce tieto údaje:

```
struct SCas{
   int den;
   char mesiac[4];
   int rok,hod,min,sek;
};
```

Všetky položky sú celé čísla, okrem mesiaca, pretože ten je v logovacom súbore vyjadrený ako 3-znaková skratka mesiaca. Keďže prvá časť log záznamu má premenlivú dĺžku (rôzna IP adresa), nemôžeme si jednoducho vypočítať

index prvého znaku v dátume. Vždy ale vieme, že dátum začína za znakom ‘[’. Stačí teda zistiť, kde sa nachádza znak ‘[’a potom už môžeme jednoducho zistiť dátum a čas udalosti. Pre zistenie pozície znaku ‘[’ použijeme funkciu strchr [2]. Funkcia vráti smerník na prvý výskyt znaku z reťazci.

Získanie dátumu a času z riadku súboru

Zápisom

```
char data[]="127.0.0.1 - - [13/Mar/2009:12:55:30 +0100] \"GET /
HTTP/1.1\" 200 4075";
data=strchr(data,'[');
```

získame reťazec ktorý začína znakom ‘[’. Potom už vieme rozlíšiť ďalšie znaky. Keďže dátum má pevnú štruktúru, platí nasledovné :

reťazec *data*

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21 ...
znak	[	0	5	/	M	a	r	/	2	0	0	9	:	1	2	:	5	5	:	3	0	...

Vieme teda že platí:

- deň získame z číslic na prvej a druhej pozícii poľa data: data[1]*10+data[2]
  - Keďže ale pole data je znakové pole, všetky znaky musíme previať na číselné hodnoty. Chcem teda previesť znak '5' na hodnotu 5. Tento prevod je jednoduché odpočítanie znaku '0'.
- Mesiac získame spojením 4-tého až 6-teho znaku.
- Rok, hodinu, minútu a sekundu získame podobne ako deň.

V nasledujúcom zdrojovom kóde je vytvorená funkcia, ktorej parametrom je načítaný riadok z log súboru a funkcia vráti dátum a čas (štruktúra SCas) z tohoto záznamu:

```
SCas dajCas(char *data)
{
  SCas cas;
  data=strchr(data,'[');
  //retazec data vyzera teraz nasledovne:
  //[13/Mar/2009:12:55:30 +0100] "GET / HTTP/1.1" 200 4075

  // udaje o dni su na indexe 1 a 2
  cas.den=(data[1]-'0')*10+(data[2]-'0');

  //mesiac sa skladá z 3 znakov (index 4 az 6) a z ukoncujuceho znaku
  cas.mesiac[0]=data[4];
  cas.mesiac[1]=data[5];
  cas.mesiac[2]=data[6];
  cas.mesiac[3]=0;

  //rok tvoria 4 cislice (index 8 az 11)

cas.rok=(data[8]-'0')*1000+(data[9]-'0')*100+(data[10]-'0')*10+(data[11]-'0');

  cas.hod=(data[13]-'0')*10+(data[14]-'0');
```

```
  cas.min=(data[16]-'0')*10+(data[17]-'0');
  cas.sek=(data[19]-'0')*10+(data[20]-'0');

  // vsetky udaje o datume a case su zapisovane do premennej cas, ktoru
 na konci vraciame
  return cas;
}
```

Vstupné a výstupné hodnoty

Vstupné údaje

Vstupom do programu bude súbor access.log, ktorého formát je popísaný v prvej časti zadania. Potom sa načíta z klávesnice hľadaný dátum a čas vo formáte:

- deň: číslo od 1 do 31
- mesiac: číslo od 1 do 12
- rok:číslo od 1900 do 2100
- hodina:číslo od 0 do 23
- minúta:číslo od 0 do 59
- sekunda:číslo od 0 do 59

Vzorový vstup

Súbor access.log musí byť pripravený.

```
13 3 2009 12 55 32
```

Výstupné údaje

Program v prípade úspechu na výstupe poskytne nasledujúce informácie:

- počet záznamov v súbore
- pozíciu hľadaného záznamu v súbore
  - číslo riadku
  - percentuálne vyjadrenie pozície v súbore

V prípade neúspechu bude výstupom informácia, že v daný čas neexistuje žiaden záznam.

Vzorový vstup

```
Pocet zaznamov: 1680
Záznam sa nachádza na pozícii 420 (25.000 %)
```

Poznámka k vstupným údajom:

Na vstupe bude mesiac reprezentovaný ako číselný údaj (poradie mesiaca v roku) ale v súbore access.log je reprezentovaný ako 3-znaková skratka. Treba teda previesť číslo mesiaca na skratku. Toto docielime jednoduchým zápisom:

```
 SCas c; // c je struktura typu SCas
 int mesiac=3; //nacitany mesiac zo suboru

 // pomocne pole retazcom, ktore vyuzijem na prevod cisla mesiaca na
skratku mesiaca
```

```
 char
mesiace[12][4]={"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};

 //do casti 'mesiac' struktury c (typu SCas) skopirujem retazec z pola
mesiace na indexe mesiac-1
 strcpy(c.mesiac,mesiace[mesiac-1]);
```

Riešenie v jazyku C

```
#include<fstream.h>
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
struct SCas{
       int den;
       char mesiac[4];
       int rok,hod,min,sek;
       };

SCas dajCas(char *data)
{
  SCas cas;
  data=strchr(data,'[');

  cas.den=(data[1]-'0')*10+(data[2]-'0');

  cas.mesiac[0]=data[4];
  cas.mesiac[1]=data[5];
  cas.mesiac[2]=data[6];
  cas.mesiac[3]=0;


cas.rok=(data[8]-'0')*1000+(data[9]-'0')*100+(data[10]-'0')*10+(data[11]-'0');

  cas.hod=(data[13]-'0')*10+(data[14]-'0');
  cas.min=(data[16]-'0')*10+(data[17]-'0');
  cas.sek=(data[19]-'0')*10+(data[20]-'0');

  return cas;
}

/**
* Funckia vrati cislo mesiaca.
* Vstupnym parametrom je stratka mesiaca
*/
int dajMesiac(char mesiac[4])
{
    if(strcmp(mesiac,"Jan")) return 1;
```

```
    if(strcmp(mesiac,"Feb")) return 2;
    if(strcmp(mesiac,"Mar")) return 3;
    if(strcmp(mesiac,"Apr")) return 4;
    if(strcmp(mesiac,"May")) return 5;
    if(strcmp(mesiac,"Jun")) return 6;
    if(strcmp(mesiac,"Jul")) return 7;
    if(strcmp(mesiac,"Aug")) return 8;
    if(strcmp(mesiac,"Sep")) return 9;
    if(strcmp(mesiac,"Oct")) return 10;
    if(strcmp(mesiac,"Nov")) return 11;
    if(strcmp(mesiac,"Dec")) return 12;
    return 0;
}
/**
* Funckia porovna 2 datumy - d1 a d2
* ak je d1>d2 - funkcia frati hodnotu 1
* ak je d1<d2 - funkcia frati hodnotu -1
* ak je d1==d2 - funkcia frati hodnotu 0
* Vstupnymi parametrami jsu datumy d1 a d2
*/
int porovnajCas(SCas c1,SCas c2)
{
    int d1,d2;
    char
mesiace[12][4]={"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Okt","Nov","Dec"};
    d1=c1.den+dajMesiac(c1.mesiac)*100+c1.rok*10000;
    d2=c2.den+dajMesiac(c2.mesiac)*100+c2.rok*10000;
    if(d1>d2) return 1;
    if(d1<d2) return -1;
    d1=c1.sek+c1.min*100+c1.hod*10000;
    d2=c2.sek+c2.min*100+c2.hod*10000;
    if(d1>d2) return 1;
    if(d1<d2) return -1;
    return 0;
}

int hladajBinarne(SCas *pole,int dlzka, SCas x)
{
  int najdene=0;
  int lavy = 0, pravy = dlzka - 1, stred;
  while ( (lavy <= pravy) && (najdene==0) )
  {
    stred = (lavy + pravy) / 2;
    if (porovnajCas(pole[stred] ,x)==0 ) //pole[stred]==x
         najdene=1;
    else
          if (porovnajCas(pole[stred] , x) < 0) //pole[stred] < x
```

```
            lavy = stred + 1;
          else
             pravy=stred - 1;
   }
   if(najdene==1)
     return stred;
   else
         return -1;
}

int main()
{
 fstream in;
 in.open("access.log");
 if(!in) { cout<<"subor sa neda otvorit"; return 0;}
 char log[601];
 SCas data[1000];
 SCas x;
 int i=0;
 do
 {
    in.getline(log,600);
    data[i]=dajCas(log);
    i++;
 }while(!in.eof());
 in.close();  // zatvorenie suboru
 int mesiac;
 cin>>x.den>>mesiac>>x.rok>>x.hod>>x.min>>x.sek;
 char
mesiace[12][4]={"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Okt","Nov","Dec"};
 strcpy(x.mesiac,mesiace[mesiac-1]);

 int pozicia=hladajBinarne(data,i,x);
 cout<<"Pocet zaznamov: "<<i<<endl;
 if(pozicia>0)
   cout<<"Zaznam sa nasiel na pozicii: "<<pozicia;
 else
   cout<<"Zaznam sa nenasiel";
 system("pause");
}
```

Opis použitých funkcií

SCas dajCas(char *data)

Funkcia je opísaná v časti " 1.2.2 Získanie dátumu a času z riadku súboru"

int dajMesiac(char mesiac[4])

Funkcia vypočíta z reťazca reprezentujúceho skratku mesiaca číslo mesiaca. Parametrom funkcie je reťazec z dĺžkou 4 znaky (3 pre skratku a 1 pre ukončujúci znak). V príapade, že *mesiac* patrí do množiny {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Okt","Nov","Dec"}, funkcia vráti príslušné číslo mesiaca. V opačnom prípade vráti hodnotu 0.

int porovnajCas(SCas c1,SCas c2)

Vo funkcii *hladajBinarne* potrebujeme porovnávať dátumy. Premenné typu štruktúra sa nedajú porovnávať, preto je teba vytvoriť porovnávaciu funkciu. Funkcia porovnajCas má 2 parametre: c1 a c2, ktoré budeme porovnávať. V prípade, ak je c1>c2, funkcia vráti hodnotu 1, ak c1<c2, funkcia vráti hodnotu -1. Ak sú rovnaké tak funkcia vráti hodnotu 0.

int hladajBinarne(SCas *pole,int dlzka, SCas x)

Algoritmus binárneho vyhľadávania je opísaný na stránke Algoritmy vyhľadávania. V tejto variante je namiesto poľa celých čísel pole štruktúr typu SCas. Teda budeme vyhľadávať prvok x (tiež musí byť rovnakého typu ako sú všetky prvky poľa *pole*, teda SCas). Pre účely porovnávania hodnôt bola vytvorená funkcia *porovnajCas*.

Porovnanie efektívnosti vyhľadávacích algoritmov

Úloha

Porovnajte efektívnosť iteračnej a rekurzívnej verzie binárneho vyhľadávania. Pole, v ktorom budete vyhľadávať použite z predchádzajúceho príkladu. Ako kritérium pri porovnávaní zvoľte čas vykonania algoritmu. Výsledky porovnania zdôvodnite.

Analýza úlohy

Algoritmy binárneho vyhľadávania sú v časti Algoritmy vyhľadávania. Podľa zadania budeme vyhľadávať v poli štruktúr typu SCas.

V programe potrebuje odmerať čas, za ktorý sa algoritmus skončí. Využijeme na to funkciu *clock()* [3] zadefinovanú v knižnici *time.h*, ktorá nám vráti počet taktov procesora od štartu programu. Tento [daj prevedieme na sekundy tak, že túto hodnotu vydelíme hodnotou makra CLOCKS_PER_SEC [4].

Pri meraní trvania jedného vyhľadávania však zistíme, že daný čas sa nám nepodarí odmerať. Preto musíme vyhľadávanie opakovať n-krát a výsledný čas ešte podeliť hodnotou n.

Riešenie v jazyku C

```
#include <iostream.h>
#include <stdio.h>
#include <time.h>
int main()
{
   clock_t start, end;
```

```
    double elapsed;
    start = clock();

    // beh algoritmu

    end = clock();
    elapsed = ((double) (end - start)) / CLOCKS_PER_SEC;

    cout<<elapsed<<endl;
}
```

Triedenie poľa komplexných čísel (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

>Triedenie poľa komplexných čísel
>Triedenie poľa smerníkov na CPLX
>Triedenie poľa štruktúr
>Triedenie poľa smerníkov

Lineárny zoznam

Binárny strom

Numerické algoritmy

Zadanie

Vytvorte program, ktorý usporiada pole komplexných čísel. Úlohu riešte pomocou algoritmu Quicksort (definovaného v quicksort) a pomocou vstavanej funkcie jazyka C qsort. Porovnajte dosiahnuté výsledky (rýchlosť výpočtu).

Vstupné a výstupné údaje

V programe načítajte číslo n a následne n komplexných čísel. Pre uloženie týchto čísel vytvorte také veľké pole aké je potrebné.

Analýza problému

Komplecné číslo pozostáva z 2 častí: reálna zložka a imaginárna zložka. Komplexné číslo C zapíšeme ako:

C=**a**+**b**i

Ľubovoľné komplexné číslo C=a+i*b dokážeme zobraziť v komplexnej rovine ako vektor so začiatkom v bode [0,0] a koncom v bode C=[a,b].

V jazyku C si pre komplexné číslo vytvoríme dátovú štruktúru, ktorá ho bude reprezentovať. V štruktúre definujeme 2 časti: reálnu (Re) a imaginárnu (Im). Obe časti sú typu double. Štruktúru si pomenujeme CPLX.

```
struct CPLX{
   double Re,Im;
};
```

Veľkosť komplexného čísla |C| je vzdialenosť od počiatku súradnicovej sústavy. Vypočítame ho pomocou Pytagorovej vety ako:

Majme pole komplexných čísel, ktoré chceme zotriediť podľa ich veľkosti. V programe máme začítať hodnotu n a následne n komplexných čísel. Pre uloženie týchto čísel vytvoríme jednorozmerné pole komplexných čísel. Veľkosť poľa je daná hodnotou n. Pre vytvorenie takéhoto poľa použijeme dynamickú alokáciu pamäti.

Ako triediaci algoritmus použijeme Quicksort, pretože je efektívny.

V samotnom algorime triedenia sa vyskytujú časti, kde sa porovnávajú prvky poľa vzájomne medzi sebou. Jazyk C nedovoľuje porovnávanie premenných typu štruktúra. Aby sme dokázali porovnávať premenné typu CPLX (komplexné číslo) vytvoríme si porovnávaciu funkciu. Funkcia porovnaj bude mať 2 parametre - komplexné čísla ktoré budeme porovnávať. Návratová hodnota funkcie bude typu int.

Funkcia porovnaj:

```
int porovnaj(CPLX a, CPLX b)
{
   if(abs(a)>abs(b))
      return 1;
   if(abs(a)<abs(b))
      return -1;
   return 0;
}
```

Pre porovnanie hodnoty 2 komplexných čísel vypočítame absolútnu hodnotu každého komplexného čísla a porovnáme tieto hodnoty.

Parametre funkcie

a, b - komplexné čísla (dátový typ CPLX), ktoré budeme porovnávať.

Návratová hodnota

- kladné číslo - ak platí, že a>b
- záporné číslo - ak platí, že a<b
- 0 - ak platí, že a=b

Použitie porovnávacej funkcie

```
CPLX x, y;
// nacitanie hodnot x, y
if(porovnaj(x,y)>0)
  cout<<"Cislo x je vascie ako y";
if(porovnaj(x,y)<0)
  cout<<"Cislo x je mensie ako y";
else
  cout<<"Cisla x a y su rovnake";
```

```
}
```

Na skontrolovanie správnosti usporiadania poľa komplexných čísel môžeme použiť jednoduchú funkciu, ktorá overí či prvok s indexom i+1 je väčší ako prvok s indexom i. Pri prvej nezhode funkcia vráti hodnotu -1. Ak sa pre všetky prvky platí predpoklad prvok[i+1]>prvok[i] , tak vráti funkcia hodnotu 0.

```
int skontroluj(CPLX *data,int n)
{
  for(int i=1;i<n;i++)
  {
    if(porovnaj(data[i-1],data[i])>1)
        return -1;
  }
  return 0;
}
```

Ak máme pripravenú porovnávaciu funkciu, tak aplikovanie iného algoritmu triedenia je jednoduché. Stačí v danom kóde zameniť časť kódu, kde sa robí porovnávanie prvkov.

```
void QuickSort(CPLX *data,int lavy, int pravy)
{       if(lavy<pravy)
      {  int i = lavy, j = pravy;
          CPLX p = data[ (lavy + pravy) / 2 ];
          do {
                while ( porovnaj(p,data[ i ])>0 ) i++;  // pouzitie
funkcie porovnaj
                while ( porovnaj(data[ j ],p)>0 ) j--;  // pouzitie
funkcie porovnaj
                if (i <= j)
                      {    zamen(data[ i ], data[ j ]);
                         i++; j--;
                     }
               } while (i <= j);
                QuickSort(data, lavy, j);
                QuickSort(data, i, pravy);
        }
}
```

Riešenie pomocou funkcie qsort

Jazyk C obsahuje vstavanú funkciu qsort, ktorá usporiadava pole prvkov pomocou algoritmu Quicksort. Hlavička funkcie vyzerá nasledovne:

```
void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY
*fcmp)(const void *, const void *));
```

Opis funkcie qsort je na stránke qsort

Použitie funkcie qsort pre náš príklad:

```
 qsort((void *)ccisla2,n,sizeof(ccisla2[0]),porovnajQ);
```

kde porovnajQ je porovnávacia funkcia:

```
int porovnajQ(const void *a, const void *b)
{   if(abs((*(CPLX *)a))>abs((*(CPLX *)b)))
        return 1;
    if(abs((*(CPLX *)a))<abs((*(CPLX *)b)))
        return -1;
    return 0;
}
```

Podľa definície funkcie qsort musí mať porovnávacia funkcia 2 argumenty typu ukazateľ na void. Teda parametre a a b sú typu ukazateľ na void. My však potrebujeme porovnať 2 komplexné čísla (bez ukazovateľov). Ako prvé musíme a a b pretypovať z ukazateľa na void na ukazateľ na CPLX. Toto urobíme následovne:

```
a // ukazateľ na void
(CPLX *)a // a je ukazateľ na void pretypovaný na ukazateľ na CPLX
*(CPLX *)a // hondota komplexneho cisla, ktore sme ziskali
           // z ukazateľa na void a pretypovali na ukazateľ na CPLX
```

Riešenie v jazyku C

Úlohou je porovnať rýchlosť oboch implementovaných algoritmov triedenia. Aby mali oba algoritmy rovnaké počiatočné podmienky, vytvoríme si dve zhodné polia komplexných čísel (ccisla1, ccisla2). Potom každýý algoritmus bude triediť rovnaké vstupné dáta.

```
#include <iostream>
#include <math.h>
using namespace std;

struct CPLX{double re,im;};

void nacitajCPLX(CPLX &c)
{     cin>>c.re>>c.im;   }

double abs(CPLX x)
{   return sqrt( (x.re*x.re)+ (x.im*x.im) ); }

int porovnaj(CPLX a, CPLX b)
{
   if(abs(a)>abs(b))
      return 1;
   if(abs(a)<abs(b))
      return -1;
   return 0;
}

void zamen (CPLX &a, CPLX &b)
{
    CPLX tmp=a;
    a=b;
    b=tmp;
}
```

```
void QuickSort(CPLX *data,int lavy, int pravy)
{       if(lavy<pravy)
      {  int i = lavy, j = pravy;
          CPLX p = data[ (lavy + pravy) / 2 ];
          do {
                while ( porovnaj(p,data[ i ])>0 ) i++;
                while ( porovnaj(data[ j ],p)>0 ) j--;
                if (i <= j)
                      {    zamen(data[ i ], data[ j ]);
                        i++; j--;
                    }
               } while (i <= j);
                QuickSort(data, lavy, j);
                QuickSort(data, i, pravy);
        }
}

int skontroluj(CPLX data[ ],int n)
{   for(int i=1;i<n;i++)
        if(abs(data[i-1])>abs(data[i]))
            return -1;
    return 0;
}

void vypis(CPLX *data,int n)
{  for(int i=0;i<n;i++)
    {   cout<<data[i].re;
        if(data[i].im>0)
            cout<<"+i"<<data[i].im;
        else
            cout<<"-i"<<-data[i].im;
        cout<<" ("<<abs(data[i])<<")";
        cout<<endl;
   }
}

int porovnajQ(const void *a, const void *b)
{   if(abs((*(CPLX *)a))>abs((*(CPLX *)b)))
        return 1;
    if(abs((*(CPLX *)a))<abs((*(CPLX *)b)))
      return -1;
     return 0;
}

int main()
{   int n;
```

```
    cin>>n;
    CPLX *ccisla1=new CPLX[n];
    CPLX *ccisla2=new CPLX[n];

    for(int i=0;i<n;i++)
    {  nacitajCPLX(ccisla1[i]);
        ccisla2[i]=ccisla1[i];
    }

    int k;
    QuickSort(ccisla1,0,n-1);
    k=skontroluj(ccisla1,n);
    cout<<"kontola QuickSort:"<<k<<endl;

    qsort((void *)ccisla2,n,sizeof(ccisla2[0]),porovnajQ);
    k=skontroluj(ccisla2,n);
    cout<<"kontola qsort:"<<k<<endl;
    return 0;
}
```

Pre meranie dĺžky trvania algoritmov môžeme použiť funkciu clock.

Námety na zlepšenie

Program od používateľa vyžaduje najprv zadanie čísla n a následne zadanie n komplexných čísiel. Tento úkon zadávania komplexných čísiel z klávesnice môže byť pre väčšie hodnoty n pomerne prácny. Preto bez ujmy na názornosti príkladu môžeme nahradiť zadávanie komplexných čísiel z klávesnice ich automatickým generovaním. Týmto spôsobom môžeme generovať tisíce komplexných čísiel, triediť ich rôznymi algoritmami, porovnávať výkonnosť jednotlivých algoritmov, voliť rôzne reprezentácie ukladania komplexných čísiel v pamäti a vyhodnocovať efektivitu ich ukladania s pohľadu triedenia.

Aby sme mohli využiť myšlienku automatického generovania komplexných čísiel namiesto ich manuálneho zadávania je postačujúce modifikovať vyššie uvedený program nasledovným spôsobom. Načítavanie komplexných čísiel sa v programe realizuje na riadku 80 volaním funkcie nacitajCPLX(ccisla1[i]); Namiesto tohto riadku bude výhodné uviesť generujjCPLX(ccisla1[i]); Funkciu generujCPLX() síce ešte nemáme vytvorenú, ale jej význam je zrejmý. Jej jediným účelom je nahradiť načítanie dvoch reálnych čísiel z klávesnice ich vygenerovaním. Aby sme zachovali jednotnosť oboch funkcií (nacitajCLPX a generujCLPX), keďže plnia podobnú úlohu, budú mať obe funkcie rovnaký funkčný prototyp líšiaci sa len menom funkcie. Definícia funkcie môže byť nasledovná:

```
void generujCPLX(CPLX &c)
{      int x;
       while((x=rand())==0)
       ;
       c.re = (rand()-RAND_MAX/2)/(double)x;
       c.im = (rand()-RAND_MAX/2)/(double)x;
}
```

Keďže v definícii funkcie používame funkciu rand(), ktorá je definovaná v knižnici stdlib.h musíme hneď za druhý riadok pridať: #include <stdlib.h>

Navyše, aby pri každom spustení programu sa generovali iné postupnosti čísiel, je potrebné pridať volanie funkcie, ktorá inicializuje generátor náhodných čísiel. Preto pred prvé volanie funkcie rand() pridáme riadok srand(). Môžeme tak urobiť napríklad vložením srand(time(NULL)) hneď za deklarácie premenných vo funkcii main(), t.j. medzi riadky 74-75.

Kvôli inicializovaniu generátora pseudonáhoných čísel treba pridať knižnicu time.h.

Týtmo spôsobom sme veľmi jednoducho doplnili program o automatické generovanie komplexných čísiel. Teraz už môžeme využiť funkciu clock() pre stanovenie doby triedenia.

Triedenie poľa smerníkov na komplexné čísla (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

>Triedenie poľa komplexných čísel
>Triedenie poľa smerníkov na CPLX
>Triedenie poľa štruktúr
>Triedenie poľa smerníkov

Lineárny zoznam

Binárny strom

Numerické algoritmy

Zadanie

Vytvorte program, ktorý usporiada pole komplexných čísel. Úlohu riešte pomocou vstavanej funkcie jazyka C qsort a algoritmu Quick sort. Úlohu riešte pomocou jednorozmerného poľa smerníkov na komplexné číslo. Porovnajte dosiahnuté rýchlosť výpočtu tohoto riešenia a predchádzajúceho príkladu riešeného pomocou poľa komplexných čísel.

Analýza riešenia

V analýze problému úlohy sa sústredíme len na to, čo je v riešení tejto úlohy nové oproti riešeniu predchádzajúcej úlohy Triedenie poľa komplexných čísel. V našej úlohe pracujeme s dynamickými poliami ukazovateľov na dátový typ CPLX, čiže v konečnom dôsledku s ukazovateľmi na štruktúrové premenné typu CPLX. Z toho dôvodu potrebujeme upraviť väčšinu funkcií z riešenia predchádzajúcej úlohy Triedenie poľa komplexných čísel (riešené príklady).

Vstupné a výstupné údaje

V programe načítajte číslo n a následne n komplexných čísel. Pre uloženie týchto čísel vytvorte také veľké pole smerníkov na komplexné číslo, aké je potrebné.

Triedenie pomocou funkcie QuickSort

V našej triediacej funkcii QuickSort z minulého príkladu musíme použiť upravenú funkciu porovnaj . Jej úprava spočíva v zmene typu jej argumentov na typ ukazovateľ na štruktúrovú premennú typu CPLX, pretože vo funkcii main chceme vytvoriť a používať pole ukazovateľov na dátový typ CPLX.

Definícia funkcie porovnaj

Definícia funkcie **porovnaj** je teraz nasledovná

```
int porovnaj(CPLX *a, CPLX *b)
{
   if(abs(a)>abs(b))
      return 1;
   else
      return 0;
}
```

Parametre funkcie

a, b – ukazovatele na dátový typ CPLX, čiže na komplexné čísla, ktoré budeme porovnávať.

Návratová hodnota

- 1 - ak platí, že

hodnota komplex. čísla, na ktoré ukazuje ukazovateľ a > hodnota komplex. čísla, na ktoré ukazuje ukazovateľ b

- 0 – v ostatných prípadoch

Funkcia abs

Vo funkcii porovnaj voláme funkciu *abs*, ktorá očakáva argument dátového typu CPLX *, preto musíme nasledovne upraviť aj definíciu tejto funkcie

```
double abs(CPLX *x)
{
   return sqrt( (x->re * x->re) + (x->im * x->im) );
}
```

Parametre funkcie

x – ukazovateľ na dátový typ CPLX, čiže na komplexné číslo, ktorého modul (veľkosť) funkcia vypočíta.

Návratová hodnota

modul, čiže veľkosť komplexného čísla, na ktoré ukazuje ukazovateľ x

Funkcia QuickSort

Použitie porovnávacej funkcie porovnaj v upravenej funkcii *QuickSort* vidíme v jej definícii

```
void QuickSort(CPLX **data, int lavy, int pravy)
{
   if(lavy<pravy)
   {
      int i = lavy, j = pravy;
      CPLX *p = data[ (lavy + pravy) / 2 ];
      do
      {
         while ( porovnaj(p, data[i]) > 0 ) i++;
         while ( porovnaj(data[j], p) > 0 ) j--;
         if (i <= j)
         {
            CPLX *tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
            i++;
            j--;
         }
      }while (i <= j);
      QuickSort(data, lavy, j);
      QuickSort(data, i, pravy);
   }
}
```

Parametre funkcie

- *data* – ukazovateľ na ukazovateľ na dátový typ CPLX, čiže v našom prípade ukazovateľ na pole ukazovateľov na dátový typ CPLX, pričom prvky poľa (komplexné čísla), na ktoré ukazujú tieto ukazovatele funkcia QuickSort usporiadava
- *lavy* – indexová premenná krajného ľavého prvku triedeného poľa
- *pravy* – indexová premenná krajného pravého prvku triedeného poľa

Návratová hodnota

je typu void – funkcia nevráti žiadnu návratovú hodnotu

Funkcia porovnaj

Taktiež sme museli upraviť funkciu skontroluj, ktorá kontroluje správne usporiadanie prvkov poľa, na ktoré ukazujú ukazovatele typu CPLX *, čiže usporiadanie štruktúrových premenných typu CPLX, vzostupne podľa ich veľkosti. Upravená definícia tejto funkcie je nasledovná

```
int skontroluj(CPLX *data[],int n)
{
   for(int i=1;i<n;i++)
      if(porovnaj(data[i-1],data[i])>0)
         return 0;
   return 1;
}
```

Funkcia overuje či prvok poľa komplexných čísiel, na ktorý ukazuje ukazovateľ data[i] je väčší, ako prvok tohto poľa, na ktorý ukazuje ukazovateľ data[i-1]. Pri prvej dvojici prvkov neusporiadanej podľa tohto pravidla funkcia vráti 0, ak sú všetky prvky tohto poľa usporiadané vzostupne, funkcia vráti 1.

Parametre funkcie

- data – ukazovateľ na pole ukazovateľov na dátový typ CPLX. Ukazovatele v tomto poli ukazujú na dátový typ CPLX, teda na štruktúrové premenné tohto typu, v ktorých sú uložené komplexné čísla
- n – počet prvkov poľa ukazovateľov na dátový typ CPLX

Návratová hodnota

- 1 - prvky poľa komplexných čísiel, na ktoré ukazujú ukazovatele v poli ukazovateľov data sú usporiadané vzostupne
- 0 - prvky poľa komplexných čísiel, na ktoré ukazujú ukazovatele v poli ukazovateľov data nie sú usporiadané vzostupne

Triedenie pomocou knižničnej funkcie qsort

Jazyk C obsahuje vstavanú funkciu qsort, ktorá usporiadava pole prvkov pomocou algoritmu Quick sort.

Volanie funkcie qsort vo funkcii main nášho príkladu vyzerá nasledovne:

```
   qsort((void *)ccisla2ptp, n, sizeof(ccisla2ptp[0]), porovnajQptp);
```

V tomto volaní je dôležitá porovnávacia funkcia *porovnajQptp*, ktorej hlavička a iné vlastnosti sú určené, avšak jej telo sme si museli vytvoriť sami

```
int porovnajQptp(const void *a, const void *b)
{
   if(abs(*(CPLX **)a) > abs(*(CPLX **)b))
      return 1;
   if(abs(*(CPLX **)a) < abs(*(CPLX **)b))
      return -1;
   return 0;
}
```

V hlavičke funkcie predpísané typy parametrov sme si museli v tele funkcie qsort vo volaniach funkcie abs pretypovať na typy korektné pre túto funkciu. Najskôr pretypujeme ukazovatele const void * na ukazovateľ CPLX ** a potom tieto ukazovatele dereferencujeme na ukazovatele CPLX *, pretože takýto typ argumentu požaduje funkcia abs. Okrem iného je to aj typ prvku poľa ukazovateľov na CPLX. Postupnosť týchto pretypovaní je nasledovná:

```
   const void *a        //'a' je ukazovateľ typu 'const void *'
   (CPLX **)a        //pretypujeme ho na ukazovateľ typu 'CPLX **'
   *(CPLX **)a        //a teraz ho dereferencujeme na ukazovateľ typu
'CPLX *', pretože ukazovateľ takého
                //typu potrebuje v argumente funkcia 'abs'
```

Komplexné čísla nášmu programu nebude zadávať používateľ, ale ich vygeneruje zavolaná funkcia generujCPLX, ktorej definícia je nasledovná

```
void generujCPLX(CPLX *c)
{
   c->re = 1+ ( rand()%MAX_CISLO_1 );
   c->im = 1+ ( rand()%MAX_CISLO_2 );
}
```

Parametre funkcie

c – ukazovateľ na dátový typ CPLX, čiže na štruktúrovú premennú typu CPLX (komplexné číslo), ktorej vnútorné premenné napĺňa funkcia vygenerovanými hodnotami.

Návratová hodnota

je typu void – funkcia nevráti žiadnu návratovú hodnotu

Príklad vstupu a výstupu programu

Vzorový vstup

```
velkost triedeneho pola: 100000
```

Vzorový výstup

```
QuickSort: trvanie: 0.01 s
qsort: trvanie: 8 s
```

Zdrojový kód riešenia

Úlohou programu je porovnať rýchlosť oboch implementovaných algoritmov triedenia. Aby mali oba algoritmy rovnaké počiatočné podmienky, vytvoríme si dve zhodné polia komplexných čísel (ccisla1ptp, ccisla2ptp). Potom bude každý algoritmus triediť rovnaké vstupné dáta.

```
#include <iostream>
#include <math.h>
#include <time.h>

using namespace std;
struct CPLX{ double re, im; };

//Funkcia ocakava v 1 argumente ukazovatel na na strukturovu premennu
typu 'CPLX'.
//Funkcia vygeneruje komplexne cislo pomocou generatora pseudonahodnych
 cisiel 'rand'.
//nahodne cisla su generovane v rozsahu 0..RAND_MAX co je 32767
void generujCPLX(CPLX *c)
{
   c->re = 1+ (rand());
   c->im = 1+ (rand());
}

//Funkcia ocakava v 1. argumente ukazovatel na strukturovu premennu
typu 'CPLX'.
//Funkcia vypocita modul komplexneho cisla (cize jeho velkost).
double abs(CPLX *x)
{
   return sqrt( (x->re * x->re) + (x->im * x->im) );
}

//Funkcia ocakava 2 ukazovatele na strukturove premenne typu 'CPLX' ako
```

```
 argumenty.
//Funkcia podla velkosti vzajomne porovna strukturove premenne typu
'CPLX', na ktore ukazuju
//ukazovatele 'a' a 'b'.
int porovnaj(CPLX *a, CPLX *b)
{
   if(abs(a)>abs(b))
      return 1;
   else
      return 0;
}

//Funkcia triedi pole strukturovych premennych algoritmom Quick sort.
//Je jedno ci je jej prvym argumentom 'CPLX **data' alebo 'CPLX
*data[]', pretoze oba syntakticky
//v nasom pripade predstavuju ukazovatel na pole ukazovatelov na
strukturove premenne typu 'CPLX'.
//Funkcia s oboma argumentami funguje spravne.
void QuickSort(CPLX **data, int lavy, int pravy)
{
   if(lavy<pravy)
   {
      int i = lavy, j = pravy;
      CPLX *p = data[ (lavy + pravy) / 2 ];
      do
      {
         while ( porovnaj(p, data[i])>0 ) i++;
         while ( porovnaj(data[j], p)>0 ) j--;
         if (i <= j)
         {
            CPLX *tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
            i++; j--;
         }
      }while (i <= j);
      QuickSort(data, lavy, j);
      QuickSort(data, i, pravy);
   }
}

//Funkcia ocakava v 1 argumente ukazovatel na pole ukazovatelov na
strukturove premenne
//typu 'CPLX'. V tele potom pracuje s prvkami 'data[i]'(cize s
ukazovatelmi) tohto pola ukazovatelov
//na strukturove premenne typu 'CPLX'.
//Funkcia skontroluje spravne usporiadanie prvkov pola,
```

```
//cize usporiadanie strukturovych premennych typu 'CPLX' vzostupne
podla ich velkosti.

//Je jedno ci je jej prvym argumentom 'CPLX **data' alebo 'CPLX
*data[]', pretoze oba syntakticky
//v nasom pripade predstavuju ukazovatel na pole ukazovatelov na
strukturove premenne typu 'CPLX'.
//Funkcia s oboma argumentami funguje spravne.
int skontroluj(CPLX *data[],int n)
{
      for(int i=1;i<n;i++)
            if(porovnaj(data[i-1],data[i])>0)
                  return 0;
      return 1;
}

//porovnavacia funkcia do kniznicnej funkcie 'qsort'
int porovnajQptp(const void *a, const void *b)
{
   if(abs(*(CPLX **)a) > abs(*(CPLX **)b))
      return 1;
   if(abs(*(CPLX **)a) < abs(*(CPLX **)b))
      return -1;
   return 0;
}

int main()
{
   int n, i;
   cout<<"velkost triedeneho pola: ";
   cin>>n;

   //'ccisla1ptp' je ukazovatel na ukazovatel na CPLX
   CPLX **ccisla1ptp=new CPLX*[n]; //alokovanie miesta pre n-prvkove
POLE UKAZOVATELOV na
                                   //CPLX, na ktore ukazuje ukazovatel
'ccisla1ptp'
   for(i=0; i<n; i++)
      ccisla1ptp[i]=new CPLX; //inicializacia ukazovatela
'ccisla1ptp[i]', vkladame do neho ukazovatel na
                              //pamatove miesto pre premennu typu CPLX

   CPLX **ccisla2ptp=new CPLX*[n];
   for(i=0; i<n; i++)
      ccisla2ptp[i]=new CPLX;

   //nastavenie startovacieho cisla generatora pseudonahodnych cisiel
```

```
'rand', aby toto cislo bolo
   //vzdy ine, po novom zavolani funkcie 'main'. Takto tento generator
vygeneruje vzdy ine pseudonahodne cisla
   srand( (unsigned)time( NULL ) );

   //plnenie obsahu prvkov pola 'ccisla1ptp' a 'ccisla2ptp'
vygenerovanymi komplexnymi cislami
   for(i=0; i<n; i++)
   {
      generujCPLX(ccisla1ptp[i]);
      ccisla2ptp[i]=ccisla1ptp[i];
   }

   int k;
   clock_t c1, c2;

   c1 = clock();
   QuickSort(ccisla1ptp, 0, n-1);
   c2 = clock();
   k=skontroluj(ccisla1ptp, n);
   if(k)
      cout<<"QuickSort: trvanie "<< (c2 - c1)/CLOCKS_PER_SEC << " s"<<endl;
   else
       cout<<"Pole nie je utriedene!";

   c1 = clock();
   qsort((void *)ccisla2ptp, n, sizeof(ccisla2ptp[0]), porovnajQptp);
   c2 = clock();
   k=skontroluj(ccisla2ptp, n);
   if(k)
      cout<<"qsort: trvanie "<< (c2 - c1)/CLOCKS_PER_SEC << " s"<<endl;
   else
       cout<<"Pole nie je utriedene!";

   for(i=0;i<n;i++)
   {
      delete ccisla1ptp[i]; //mazeme pole ukazovatelov
      delete ccisla2ptp[i];
   }
   delete[] ccisla1ptp;  //mazeme ukazovatel na toto pole
   delete[] ccisla2ptp;

   return 0;
}
```

Záver

Z výsledkov merania času doby behu programu sme zistili že knižničná funkcia qsort je niekoľkokrát pomalšia ako nami implementovaný algoritmus QuickSort. Táto skutočnosť je zapríčinená univerzálnou štruktúrou funkcie qsort, v ktorej môžeme triediť pole ľubovoľného dátového typu. Pri jej použití sa musia prvky triedeného typu niekoľkokrát pretypovať z ukazateľa na konkrétny dátový typ (u nás CPLX) na ukazovateľ na void. Toto zaberá väčšinu času. Naša implementácia algoritmu QuickSort je priamo naprogramované pre usporiadania dátovej štruktúry CPLX a netreba žiadne ďalšie pretypovania, čím sa ušetrí výpočtový čas

Triedenie poľa štruktúr (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

>Triedenie poľa komplexných čísel
>Triedenie poľa smerníkov na CPLX
>Triedenie poľa štruktúr
>Triedenie poľa smerníkov

Lineárny zoznam

Binárny strom

Numerické algoritmy

Triedenie poľa štruktúr

Zadanie

Zostavte program, ktorý bude triediť dáta uložené v súbore na základe užívateľom zadaného kritéria. Program po spustení načíta dáta zo súboru a uloží ich do poľa štruktúr. Následne na monitor zobrazí výzvu, v ktorej bude mať užívateľ možnosť vybrať kritérium triedenia. Pre jednoduchosť uvažujme príklad, v ktorom budeme triediť zamestnancov nejakej fiktívnej firmy podľa mena, priezviska alebo podľa roku jeho narodenia. Dáta v súbore nech sú uložené vo formáte kde jednotlivé položky sú oddelené medzerami čiarkami v tvare:

```
Meno Priezvisko , Rok_narodenia
```

Metodický komentár

Cieľom tejto úlohy je precvičiť prácu so štruktúrami v jazyku C, použitie funkcie rýchleho triedenia (quick sort) z knižnice *stdlib.h*, načítavanie dát zo súboru, a pomôcť osvojiť si základné znalosti pri práci s ukazovateľmi (predávanie ako parameter do funkcie, pretypovávanie).

Vzorové dáta

```
Janko Maly , 1978
Martin Starsi , 1939
Andrea Mlada , 1990
Jozko Cierny , 1985
Alenka Biela , 1985
Janko Maly , 1983
Alzbetka Mudra , 1978
```

Zjednodušujúce predpoklady

Dĺžka mena ani priezviska neprekračuje rozsah 20 znakov. V zdrojovom súbore sa nebude nachádzať viac ako 100 záznamov. Zdrojový súbor sa nachádza v tom istom adresáry ako samotný program (*.exe súbor) a jeho názov je: data.txt.

Vzorová výzva programu

```
Vyber kriterium triedenia.
  Meno                    1
  Priezvisko              2
  Rok narodenia           3
  Pre ukoncenie programu  0
Volba: _
```

Vzorový vstup

```
1
```

Vzorový výstup

```
--------------------------------------------------------
 ZAMESTNANEC                               ROK NARODENIA
--------------------------------------------------------
 Alenka               Biela                   1985
 Alzbetka             Mudra                   1978
 Andrea               Mlada                   1990
 Janko                Maly                    1983
 Janko                Maly                    1978
 Jozko                Cierny                  1985
 Martin               Starsi                  1939
--------------------------------------------------------
```

Návod ako začať

Zo zadania vyplýva, že hlavným účelom programu je triediť (vzostupne príp. zostupne zoradzovať) zamestnancov podľa triediaceho kritéria (meno, priezvisko, rok). Aby sme pri manipulácii (triedení) s jednotlivými zamestnancami pracovali ucelene so všetkými dátami vzťahujúcimi sa ku konkrétnemu zamestnancovi naraz, bude výhodné použiť štruktúru, ktorá nám tieto dáta takpovediac "zabalí". Takto budeme mať údaje o jednom zamestnancovi pohromade a budeme môcť s nimi pohodlne manipulovať. Týmto sme vyriešili problém ako reprezentovať jedného zamestnanca. Teraz sa vynára otázka ako efektívne pracovať s celým súborom zamestnancov. Pretože vieme, že v zdrojovom súbore sa nebude nachádzať viac ako 100 zamestnancov, bude vcelku výhodné použiť statické pole štruktúr na pamätanie všetkých zamestnancov. Dĺžka tohto poľa bude 100 položiek. Jedna položka poľa potom bude predstavovať jedného konkrétneho zamestnanca. Utriedenie zamestnancov potom budeme realizovať utriedením tohto poľa. Pre triedenie poľa ponúka knižnica *stdlib.h* funkciu rýchleho triedenia *qsort*. Parametre a použitie funkcie *qsort* z knižnice *stdlib.h* sú na stránke qsort.

Možné riešenie v jazyku C

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>

#define NAZOV_SUBORU "data.txt"

// Vytvorenie pomenovanej struktury s nazvom "ZAMESTNANEC"
struct ZAMESTNANEC {
   char meno[21];
   char priezvisko[21];
   int rok_narodenia;
   };

// Uplne funkcne prototypy pouzivanych funkcii:

// Nacitanie jedineho zamestnanca zo suboru
int nacitajZAMESTNANEC(FILE *fr, ZAMESTNANEC *V);

// Vypisanie jedineho zamestnanca na monitor
void vypisZAMESTNANEC(ZAMESTNANEC V);

// Porovnanie dvoch zamestnancov podla roku narodenia
int porovnajROK(const void *V1, const void *V2);

// Porovnanie dvoch zamestnancov podla mena
int porovnajMENO(const void *V1, const void *V2);

// Porovnanie dvoch zamestnancov podla priezviska
int porovnajPRIEZV(const void *V1, const void *V2);

// Hlavna funkcia programu:
int main(int argc, char* argv[])
```

```
{
   ZAMESTNANEC pole[100];         // pole struktur pre uchovavanie
jednotlivych zamestnancov
   int Pocet_zamestnancov = 0;    // pocet nacitanych zamestnancov
   int volba;                     // kriterium triedenia / ukoncenia
programu
   FILE *fr;                      // pointer na subor
   int i;                         // pomocna premenna

   // Otvorenie zdrojoveho suboru v rezime "read"
   if((fr=fopen(NAZOV_SUBORU,"r"))==NULL)
   {
      printf("\n Pozadovany subor sa nepodarilo otvorit!\n");
      system("pause");
      exit(1);
   }

   // Nacitanie vsetkych zamestnancov zo suboru
   for(i=0; nacitajZAMESTNANEC(fr,&pole[i]); i++)
   {
      ;
   }
   Pocet_zamestnancov = i;
   fclose(fr); // zatvorenie suboru, data uz su nacitane..
   fr = NULL;  // zaroven nastavime na NULL pre pripad aby sme
               // v pripade chybneho pouzitia tohto poitera boli vcas
varovani

   // Hlavny cyklus programu
   do{
      printf("\n Vyber kriterium triedenia. \n"
               "   Meno                      1\n"
             "   Priezvisko                2\n"
             "   Rok narodenia             3\n\n"
             "   Pre ukoncenie programu    0\n\n"
               " Volba: ");
      scanf("%d",&volba);

      switch(volba)
      {
         case 0: break;
         case 1:
qsort((void*)pole,Pocet_zamestnancov,sizeof(ZAMESTNANEC),porovnajMENO);
   break;
                                              // namiesto sizeof(ZAMESTNANEC)
mozeme pouzit sizeof(pole[0])
         case 2:
```

```
qsort((void*)pole,Pocet_zamestnancov,sizeof(ZAMESTNANEC),porovnajPRIEZV);
 break;
         case 3:
qsort((void*)pole,Pocet_zamestnancov,sizeof(ZAMESTNANEC),porovnajROK);
    break;
         default:
            printf("\n Neocakavany vstup. Program bude ukonceny.\n");
            volba=0;
      }
      if(volba) // Vypisanie utriedeneho pola na monitor
      {
         clrscr(); //vycistime si obrazovku

printf("--------------------------------------------------------\n");
         printf(" ZAMESTNANEC                               ROK
NARODENIA\n");

printf("--------------------------------------------------------\n");
         for(i=0; i<Pocet_zamestnancov; i++)
            vypisZAMESTNANEC(pole[i]);

printf("--------------------------------------------------------\n");
      }
   }while(volba);

   system("pause");
   return 0;
}

// Definicie pouzivanych funkcii:
//-----------------------------------------------------------------------------
int nacitajZAMESTNANEC(FILE *fr, ZAMESTNANEC *V)
{
   int a;
   a = fscanf(fr,"%s %s , %d", V->meno, V->priezvisko, &V->rok_narodenia);
   if(a == EOF) a = 0;
   return a;
}
//-----------------------------------------------------------------------------
void vypisZAMESTNANEC(ZAMESTNANEC V)
{
   printf(" %-20s %-20s     %d\n", V.meno, V.priezvisko,
V.rok_narodenia);
}
//-----------------------------------------------------------------------------
int porovnajROK(const void *V1, const void *V2)
{
```

```
   return ((ZAMESTNANEC*)V1)->rok_narodenia -
((ZAMESTNANEC*)V2)->rok_narodenia;
}
//------------------------------------------------------------------
int porovnajMENO(const void *V1, const void *V2)
{
   return strcmp(((ZAMESTNANEC*)V1)->meno, ((ZAMESTNANEC*)V2)->meno);
}
//------------------------------------------------------------------
int porovnajPRIEZV(const void *V1, const void *V2)
{
   return strcmp(((ZAMESTNANEC*)V1)->priezvisko,
((ZAMESTNANEC*)V2)->priezvisko);
}
```

Komentár k uvedenému riešeniu

V programe sa na prvý pohľad nachádza zopár zvláštností. Prvou je prázdne telo cyklu *for* pri načítavaní zamestnancov zo súboru - pozri riadok xx programu (nachádza sa tam len bodkočiarka za účelom zdôraznenia vedomého vypustenia tela funkcie). Dôvodom je, že samotné načítanie zamestnancov sa nachádza v porovnávacej (testovacej) časti cyklu *for*. Ak sa úspešne načíta nejaký zamestnanec zo súboru potom funkcia *nacitajZAMESTNANEC* vracia nenulové kladné číslo, v opačnom prípade nulu a cyklus *for* sa skončí. Druhou zvláštnosťou je spôsob načítavania položiek "meno", "priezvisko" a "rok_narodenia" v tele definicie funkcie *nacitajZAMESTNANEC* - pozri riadok xx. V tomto prípade s výhodou využívame možností funkcie *fscanf*, pričom využívame skutočnosť, že jednotlivé položky sú oddelené medzerou a čiarkou. Treťou zvláštnosťou je použitie typu *const void* * v parametroch a následné pretypovávanie v tele na pointer na pomenovanú štruktúru ZAMESTNANEC vo všetkých troch porovnávacích funkciách - *porovnajMENO*, *porovnajPRIEZV* a *porovnajROK*. Toto priamo súvisí s požiadavkou na parametre porovnávacej funkcie, ktorá musí vyhovovať funkčnému prototypu funkcie triedenia - *qsort*.

Nedostatky uvedeného riešenia a námety na zlepšenie

Program nie je vhodný na triedenie väčšieho množstva dát. Pri triedení sa manipuluje s celými položkami (štruktúra ZAMESTNANEC), ktoré majú v tomto prípade veľkosť sizeof(ZAMESTNANEC) == 48 Bajtov. Funkcia *qsort* tak musí pri triedení kopírovať celý obsah, čo môže byť časovo náročné. Ak by navyše jednotlivé položky uchovávali ešte väčie množstvo dát (ďalšie údaje o zamestnancovi, ako napríklad pracovné zaradenie, osobné hodnotenie zamestnanca, kontaktné informácie a pod.) bola by situácia ešte horšia. Podobný nedostatok má aj funkcia vypisZAMESTNANEC. V tomto prípade však tento problém nie je kritický. Prvou možnosťou ako zefektívniť triedenie je redukovanie veľkosti štruktúry ZAMESTNANEC. To sa dá docieliť použitím dynamicky alokovanej pamäte pre uchovávanie mena a priezviska zamestnanca. V tomto prípade by mohla štruktúra ZAMESTNANEC vyzerať nasledovne:

```
struct ZAMESTNANEC {
   char *meno;
   char *priezvisko;
   int rok_narodenia;
   };
```

Potom by sa pamäťové nároky na uchovávanie redukovali na 12 Bajtov. Je však nevyhnutné pamätať na to, že adresy uchovávané v položkách "meno" a "priezvisko" môžu byť ľubovolné (sú neinicializované) a preto môžu ukazovať na

ľubovolné miesto v pamäti. Preto treba použiť dynamickú alokáciu pamäte (funkcia malloc, resp. operátor new) a ukazovatele "meno" a "priezvisko" správne nastaviť (využitím návratovej hodnoty funkcie malloc, resp. operátora new).

Ďalšou možnosťou ako zefektívniť triedenie je použitie poľa ukazovateľov na prvky typu ZAMESTNANEC. Ukazovateľ na typ ZAMESTNANEC (či už pôvodný alebo tu uvedený) má veľkosť 4 Bajty. Z pohľadu triedenia funkciou qsort je preto tento spôsob ešte výhodnejší.

Použitím kombinácie oboch možností (dynamicky alokovaný priestor pre uchovávenie jednotlivých záznamov typu ZAMESTNANEC a použitie poľa ukazovateľov na prvky typu ZAMESTNANEC pre účely triedenia) je možné dosiahuť hospodárne využitie pamäte a zároveň efektívne triedenie rozsiahlejšieho súboru dát.

Druhou možnosťou sa zaoberá nasledujúca časť - Triedenie poľa ukazovateľov.

Triedenie poľa smerníkov (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

>Triedenie poľa komplexných čísel
>Triedenie poľa smerníkov na CPLX
>Triedenie poľa štruktúr
>Triedenie poľa smerníkov

Lineárny zoznam

Binárny strom

Numerické algoritmy

Triedenie poľa ukazovateľov alebo Triedime štruktúry efektívne

Zadanie

Uvažujme rovnaké zadanie ako v predošlom. Navyše požadujme viacúrovňové (vnorené) triedenie podľa triediaceho kritéria s nižšou prioritou ak sú položky podľa aktuálneho triediaceho kritéria ekvivalentné. V našom jednoduchom príklade to znamená, že ak napríklad triedime zamestnancov podľa krstného mena a mená niektorých zamestnancov sú rovnaké, potom budú títo utriedení podľa priezviska. Naopak, ak budeme triediť podľa priezviska a priezviská niektorých zamestnancov budú rovnaké, potom ich utriedime podľa krstného mena. Túto situáciu môžeme rozšíriť aj o triedenie podľa roku narodenia. V tomto prípade môžeme definovať prioritu jednotlivým triediacim kritériam a triediť následne podľa priorít.

Metodický komentár

Cieľom tejto úlohy v porovaní s predchádzajúcou je efektívne triediť dáta použitím ukazovateľov a poukázať na ďalšie možnosti pri ich triedení. Využijú sa pokročilejšie techniky práce s ukazovateľmi, dynamická alokácia a realokácia pamäte.

Vzorové dáta

Ako v predošlom.

Zjednodušujúce predpoklady

Rovnaké ako v predošlom s tým rozdielom, že počet záznamov v súbore nie je obmedzený. Predpokladáme však, že systém má dostatok voľnej pamäte.

V súbore sa teda môže nachádzať niekoľko desiatok, ale aj stoviek tisícov záznamov.

Vzorová výzva programu

```
Vyber postupnost kriterii triedenia. (Prve kriterium ma najvacsiu prioritu.)
  Meno, Priezvisko, Rok    1
  Priezvisko, Meno, Rok    2
  Rok, Meno, Priezvisko    3
  Rok, Priezvisko, Meno    4
  Pre ukoncenie programu   0
Volba: _
```

Vzorový vstup

```
1
```

Vzorový výstup

```
--------------------------------------------------------
 ZAMESTNANEC                               ROK NARODENIA
--------------------------------------------------------
 Alenka               Biela                    1985
 Alzbetka             Mudra                    1978
 Andrea               Mlada                    1990
 Janko                Maly                     1978
 Janko                Maly                     1983
 Jozko                Cierny                   1985
 Martin               Starsi                   1939
--------------------------------------------------------
```

Návod ako začať

Prvým významným rozdielom v porovnaní s predchádzajúcim zadaním je skutočnosť, že počet záznamov (zamestnancov) v súbore nie je obmedzený a je vopred neznámy. Z toho dôvodu je použitie statického poľa štruktúr neefektívne. Riešením je dynamická alokácia pamäte. Mohli by sme tak dynamicky alokovať pole štruktúr podľa aktuálnej potreby programu. Ak sa však zamyslíme nad požiadavkou efektívneho triedenia zamestnancov prichádzame k záveru, že ani toto riešenie by nebolo vhodné. Dôvodom je činnosť funkcie *qsort*, ktorá pri triedení manipuluje s celými položkami poľa, ktoré by v tomto prípade boli štruktúry (keďže by sa jednalo o pole štruktúr či už staticky alebo dynamicky alokované). Funkcia *qsort* by tak musela pracovať pri triedení s pomerne veľkým

objemom dát. Riešením tejto situácie je použitie dynamicky alokovaného poľa ukazovateľov. Jednotlivé položky poľa by potom už neboli samotné štruktúry, ale adresy týchto štruktúr. Ako vieme, veľkosť adresy je 4 bajty (resp. 8B v 64-bitovom OS). Takže funkcia *qsort* bude triediť pole adries tak aby po zotriedení boli splnené požiadavky triedenia dané prioritami triediacich kritérií. Ak zhrnieme doteraz povedané, tak pre pamätanie celého súboru zamestnancov je výhodné použiť dynamicky alokované pole ukazovateľov, v ktorom pre každý prvok poľa budeme dynamicky alokovať pamäťový priestor pre pamätanie štruktúry uchovávajúcej údaje o jednom konkrétnom zamestnancovi.

Druhým významným rozdielom je požiadavka na viacúrovňové triedenie. Pri písaní porovnávacích funkcií zohľadňujúcich tri úrovne priorít triediacih kritérií je výhodné začať od písania funkcií pre jednoúrovňové triedenie (pozri predchádzajúci príklad), tieto ďalej využiť pri písaní dvojúrovňových, a tieto spolu zasa pri písaní trojúrovňových. Takto môžeme veľmi jednoducho a prehľadne podľa potreby napísať rôzne mnohoúrovňové porovnávacie funkcie.

Alokácia pamäti

Pre alokáciu pamäti je možné využiť funkciu malloc (funkcia jazyka C), alebo new (operátor jazyka C++). Uvedieme definície týchto spôsobov:

malloc

Funkcia malloc [1] definovaná v knižnici stdlib.h:

```
void * malloc ( size_t size );
```

size

veľkosť pamäti, ktorá sa alokuje

Návratová hodnota

V prípade úspechu vráti smerník na alokovanú pamäť, v prípade neúspechu vráti NULL. Návratová hodnota je smetník na void, preto si ho treba pretypovať na želaný typ.

new

Operátor new [2] je definovaný v knižnici iostream.h nasledovne:

```
[::] new [placement] new-type-name [new-initializer]
[::] new [placement] ( type-name ) [new-initializer]
```

Operátor new alokuje mapäť pre premenné (resp. objekty) alebo pole premenných, ktoré sú typu *type-name* a vracia ukazovateľ na túto alokovanú pamäť. Ukazovateľ je na rozdiel od funkcie malloc už pretypovaný na správny typ. Pri neúspechu vyvolá výnimku. Spracovanie vyvolanej výnimky je v nasledujúcom kóde:

```
int * i_arr;
try {
   i_arr = new int[0x3fffffff];
}
catch(...) {
   cout<<"Vynimka: nedostatok pamati pre alokaciu";
}
```

Možné riešenie v jazyku C

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>

#define NAZOV_SUBORU "data.txt"
#define KROK_REALOKACIE 100

// Vytvorenie pomenovanej struktury s nazvom "ZAMESTNANEC"
struct ZAMESTNANEC {
   char meno[21];
   char priezvisko[21];
   int rok_narodenia;
   };

// Uplne funkcne prototypy pouzivanych funkcii:

// Nacitanie jedineho zamestnanca zo suboru
int nacitajZAMESTNANEC(FILE *fr, ZAMESTNANEC *V);

// Vypisanie jedineho zamestnanca na monitor
void vypisZAMESTNANEC(ZAMESTNANEC V);

// Porovnanie dvoch zamestnancov podla roku narodenia
int porovnajROK(const void *V1, const void *V2);

// Porovnanie dvoch zamestnancov podla mena
int porovnajMENO(const void *V1, const void *V2);

// Porovnanie dvoch zamestnancov podla priezviska
int porovnajPRIEZV(const void *V1, const void *V2);

// Porovnanie najprv podla priezviska potom podla mena
int porovnajPRIEZV_MENO(const void *V1, const void *V2);
int porovnajMENO_PRIEZV(const void *V1, const void *V2);      //
podobne dalej...
int porovnajMENO_PRIEZV_ROK(const void *V1, const void *V2);
int porovnajPRIEZV_MENO_ROK(const void *V1, const void *V2);
int porovnajROK_PRIEZV_MENO(const void *V1, const void *V2);
int porovnajROK_MENO_PRIEZV(const void *V1, const void *V2);

// Hlavna funkcia programu:
int main(int argc, char* argv[])
{
   // Pozor, zmena v porovnani s predchadzajucim riesenim!
   ZAMESTNANEC **pole;
```

```
   // Pocet nacitanych zamestnancov
   int Pocet_zamestnancov = 0;
   // Prednastaveny maximalny pocet zamestnancov
   int maxPocet_zamestnancov = 100;
   // kriterium triedenia / ukoncenia programu
   int volba;
   // pointer na zdrojovy subor, z ktoreho budeme nacitavat
zamestnancov
   FILE *fr;
   // pointer na funkciu, pomocou ktorej budeme triedit
   int (*sortFun)(const void *,const void *);
   int i;      // pomocna premenna

   // Otvorenie zdrojoveho suboru v rezime "read"
   if((fr=fopen(NAZOV_SUBORU,"r"))==NULL)
   {
      printf("\n Pozadovany subor sa nepodarilo otvorit!\n");
      system("pause");
      exit(1);
   }

   // Prvotna alokacia pamate pre pole pointrov na pomenovanu strukturu
 ZAMESTNANEC

if((pole=(ZAMESTNANEC**)malloc(maxPocet_zamestnancov*sizeof(ZAMESTNANEC*)))==NULL)
   {
      printf("Nedostatok pamete!\n");
      system("pause");
      exit(1);
   }

//*************************************************************************
   //*  Pomocou operatora new sa kod na riadkoch 68 az 73 zapise
nasledovne
   //* try{
   //*   pole=new (ZAMESTNANEC*)[maxPocet_zamestnancov];
   //* }
   //* catch(...)
   //* {  cout<<"Nedostatok pamete";
   //*    exit(1);
   //* }

//************************************************************************

   // Najprv si musime alokovat priestor pre prveho zamestnanca
   // (aby data, ktore nacitame sme mali kam ulozit)
```

```
   if((pole[0]=(ZAMESTNANEC*)malloc(sizeof(ZAMESTNANEC)))==NULL)
   {
      printf("\n Nedostatok pamete...\n");
      exit(1);
   }


//*************************************************************************
   //*  Pomocou operatora new sa kod na riadkoch 87 az 91 zapise
nasledovne
   //* try{
   //*   pole[0]=new ZAMESTNANEC;
   //* }
   //* catch(...)
   //* {  cout<<"Nedostatok pamete";
   //*    exit(1);
   //* }

//************************************************************************
   // Nacitanie vsetkych zamestnancov zo suboru
   for(i=0; nacitajZAMESTNANEC(fr,pole[i]); i++)
   {
            //Alokujeme si miesto pre dalsieho zamestnanca:
      if((pole[i+1]=(ZAMESTNANEC*)malloc(sizeof(ZAMESTNANEC)))==NULL)
      {
         printf("\n Nedostatok pamete pre pridanie dalsieho
zamestnanca...\n");
         system("pause");
         exit(1);
      }

      // V pripade potreby realokujeme pole pinterov: (Tato situacia
nastane
      // ak pocet zamestnancov v subore je vacsi ako aktualne nastavna
hodnota
      // premennej maxPocet_zamestnancov)
      if(i == maxPocet_zamestnancov-2)
      {
         maxPocet_zamestnancov += KROK_REALOKACIE;  // Zvysime
maximalny pocet zamestnancov
         //printf("...realokujem...");

if((pole=(ZAMESTNANEC**)realloc(pole,maxPocet_zamestnancov*sizeof(ZAMESTNANEC*)))==NULL)
         {
            printf(" Nedostatok pamate pre realokaciu pola
pointerov!\n");
            system("pause");
```

```
            exit(1);
         }
      }
   }
   Pocet_zamestnancov = i;
   fclose(fr); // zatvorenie suboru, data uz su nacitane..
   fr = NULL;  // zaroven nastavime na NULL pre pripad aby
               // sme v pripade chybneho pouzitia tohto poitera boli
vcas varovani

   // Hlavny cyklus programu
   do{
      printf("\n Vyber postupnost kriterii triedenia. (Prve kriterium
ma najvacsiu prioritu.)\n"
             "   Meno, Priezvisko, Rok    1\n"
             "   Priezvisko, Meno, Rok    2\n"
             "   Rok, Meno, Priezvisko    3\n"
               "   Rok, Priezvisko, Meno    4\n\n"
             "   Pre ukoncenie programu   0\n\n"
             " Volba: ");
      scanf("%d",&volba);

      switch(volba)
      {
         case 0: break;
         case 1: sortFun = porovnajMENO_PRIEZV_ROK;   break;
         case 2: sortFun = porovnajPRIEZV_MENO_ROK;   break;
         case 3: sortFun = porovnajROK_MENO_PRIEZV;   break;
         case 4: sortFun = porovnajROK_PRIEZV_MENO;   break;
         default:
            printf("\n Neocakavany vstup. Program bude ukonceny.\n");
            volba=0;
      }
      if(volba) // Vypisanie utriedeneho pola na monitor
      {
         // Utriedenie zamestnancov:
         qsort((void*)pole,Pocet_zamestnancov,sizeof(pole[0]),sortFun);

         clrscr(); //vycistime si obrazovku

printf("--------------------------------------------------------\n");
         printf(" ZAMESTNANEC                              ROK
NARODENIA\n");

printf("--------------------------------------------------------\n");
         for(i=0;i<Pocet_zamestnancov;i++)
            vypisZAMESTNANEC(*pole[i]);
```

```
printf("-------------------------------------------------------\n");
      }
   }while(volba);

   // Na zaver nasleduje uvolnenie pamate, ktoru sme alokovali.
   // Najprv musime uvolnit pamet, ktoru sme alokovali pre jednotlivych
 zamestnancov.

   //Pocet alokovanych zamestnancov je o jednehoneho viac...
   for(i=0; i<skutocnyPocet_zamestnancov + 1;i++)
      free(pole[i]);

   //Potom uvolnime pamat alokovanu pre pointre
   free(pole);
   poleStud = NULL;  // a nastavime na NULL (nie je nevyhnutne
potrebne)

   system("pause");
   return 0;
}

// Definicie pouzivanych funkcii:
//---------------------------------------------------------------------------
int nacitajZAMESTNANEC(FILE *fr, ZAMESTNANEC *V)
{
   int a;
   a = fscanf(fr,"%s %s , %d", V->meno, V->priezvisko, &V->rok_narodenia);
   if(a == EOF) a = 0;
   return a;
}
//---------------------------------------------------------------------------
void vypisZAMESTNANEC(ZAMESTNANEC V)
{
   printf(" %-20s %-20s     %d\n", V.meno, V.priezvisko,
V.rok_narodenia);
}
//---------------------------------------------------------------------------
int porovnajROK(const void *V1, const void *V2)
{
   return (*(ZAMESTNANEC**)V1)->rok_narodenia -
(*(ZAMESTNANEC**)V2)->rok_narodenia;
}
//---------------------------------------------------------------------------
int porovnajMENO(const void *V1, const void *V2)
{
   return strcmp((*(ZAMESTNANEC**)V1)->meno, (*(ZAMESTNANEC**)V2)->meno);
```

```
}
//---------------------------------------------------------------------------
int porovnajPRIEZV(const void *V1, const void *V2)
{
   return strcmp((*(ZAMESTNANEC**)V1)->priezvisko,
(*(ZAMESTNANEC**)V2)->priezvisko);
}
//---------------------------------------------------------------------------
int porovnajPRIEZV_MENO(const void *V1, const void *V2)
{
   int a = porovnajPRIEZV(V1, V2);
   if(a == 0) a = porovnajMENO(V1, V2);
   return a;
}
//---------------------------------------------------------------------------
int porovnajMENO_PRIEZV(const void *V1, const void *V2)
{
   int a = porovnajMENO(V1, V2);
   if(a == 0) a = porovnajPRIEZV(V1, V2);
   return a;
}
//---------------------------------------------------------------------------
int porovnajMENO_PRIEZV_ROK(const void *V1, const void *V2)
{
   int a = porovnajMENO_PRIEZV(V1, V2);
   if(a == 0) a = porovnajROK(V1, V2);
   return a;
}
//---------------------------------------------------------------------------
int porovnajPRIEZV_MENO_ROK(const void *V1, const void *V2)
{
   int a = porovnajPRIEZV_MENO(V1, V2);
   if(a == 0) a = porovnajROK(V1, V2);
   return a;
}
//---------------------------------------------------------------------------
int porovnajROK_PRIEZV_MENO(const void *V1, const void *V2)
{
   int a = porovnajROK(V1, V2);
   if(a == 0) a = porovnajPRIEZV_MENO(V1, V2);
   return a;
}
//---------------------------------------------------------------------------
int porovnajROK_MENO_PRIEZV(const void *V1, const void *V2)
{
   int a = porovnajROK(V1, V2);
   if(a == 0) a = porovnajMENO_PRIEZV(V1, V2);
```

```
   return a;
}
```

Komentár k uvedenému riešeniu

Bude zverejnený neskôr.

Nedostatky uvedeného riešenia a námety na zlepšenie

Program nie je vhodný na triedenie extrémne rozsiahlych dát. Riešením tohto nedostatku je použitie iných triediacich funkcií, ktoré pracujú na princípe triedenia triedeného súboru po častiach. Ďalším nedostatkom je výpis utriedených dát na monitor. Táto operácia môže byť pri väčšom počte záznamov časovo veľmi zdĺhavá. Oveľa výhodnejšie a zároveň rýchlejšie je zapisovať výsledky do súboru. V tomto prípade by postačovalo modifikovať funkciu *vypisZAMESTNANEC* tak, aby vyhovovala tomuto účelu.

```
vypisZAMESTNANEC(FILE *fw, ZAMESTNANEC *V)
{
   // Zaroven efektivnejsie pristupujeme k polozkam zamestnanca
(parameter V je pointer...)
   fprintf(fw," %-20s %-20s     %d\n", V->meno, V->priezvisko,
V->rok_narodenia);
}
```

Ak by sme si ďalej zadefinovali funkciu na výpis všetkých zamestnancov celý program by sme zjednodušili a výpis zamestnancov by bol univerzálnejší v tom zmysle, že výpis zamestnancov či už do súboru alebo na monitor by mal jednotnú podobu. Funkcia na výpis všetkých zamestnancov by mohla vyzerať nasledovne:

```
vypisZAMESTNANCOV(FILE *fw, ZAMESTNANEC **pole, int Pocet_zamestnancov)
{
   printf("--------------------------------------------------------\n");
   printf(" ZAMESTNANEC                              ROK NARODENIA\n");
   printf("--------------------------------------------------------\n");
   for(i=0;i<Pocet_zamestnancov;i++)
      vypisZAMESTNANEC(fw, pole[i]);
   printf("--------------------------------------------------------\n");
}
```

Potom by jednoducho stačilo otvoriť nejaký súbor v režime "write" a zavolať túto funkciu na zápis všetkých zamestnancov do súboru. Ak by sme však chceli vypísať zamestnancov na monitor, potom namiesto ukazovateľa na otvorený súbor zadáme *stdout*. Čiže do programu vložíme riadok:

```
   vypisZAMESTNANCOV(stdout,pole,Pocet_zamestnancov);
```

Neusporiadaný lineárny zoznam (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

>Neusporiadaný lineárny zoznam
>Usporiadaný lineárny zoznam

Binárny strom

Numerické algoritmy

Zadanie

Vo vstupnom súbore máme zoznam študentov - v každom riadku sa nachádza meno, priezvisko, známka zo ZI, den, mesiac a rok narodenia. V poslednom riadku je 5 núl. Zostavte program, ktorý zo súboru načíta všetkých študentov a vytvorí lineárny zreťazený zoznam, ktorého jednotlivé prvky budú obsahovať údaje jedného riadku. Prvky do zoznamu na koniec zoznamu. Na konci zoznam vypíšte a zmažte.

Vzorový príklad

Vstup

```
Jan Mrkvicka A 4 6 1985
Ferdinand Tell C 4 12 1986
Viliam Tell F 1 1 1987
johanka Z_arku C 5 4 1238
0 0 0 0 0 0
```

Výstup

výpis zoznamu: je rovnaký ako vstupné dáta.

Analýza úlohy

Na reprezentáciu dát si vytvoríme štruktúry *TDatum* a *TStudent*, ktorá nám opisujú jedného študenta.

```
struct TDatum
{
  int d,m,y;
};
struct TStudent
{
  char meno[32],priezvisko[32];
  TDatum datum_narodenia;
```

```
  char znamka_ZI;
};
```

Príklad sa má riešiť pomocou lineárneho zoznamu. Vytvoríme si štruktúru TPrvok, ktorá bude jedným prvkom zoznamu.

```
struct TPrvok
{
   TStudent student;
   TPrvok *dalsi;
};
```

Nakoniec ešte potrebujeme samotnú štruktúru zoznam.

```
struct TZoznam
{
   TPrvok *prvy, *posledny;
};
```

Vo vstupnom súbore nám každý riadok reprezentuje jedného študenta. Takže budeme naraz načítavať všetky údaje v riadku a uložíme ich do patričných položiek štruktúry TStudent. Na načítavanie použijeme funkciu *Nacitaj(TStudent &s,istream &vstup)*. Zaujímavý je druhý parameter, je to odkaz na vstupný prúd (istream) - môže to byť *cin*, alebo vlastný dátový prúd (súbor).

Pre výpis prvku môžeme použiť funkciu *VypisPrvok(TPrvok *p)*, ktorá bude vypisovať obsah prvku *p* (p obsahuje štrukúru *TStudent*, ktorá obsahuje položky *meno*, *priezvisko*, *datum* a *znamka_ZI*)

V ukážke sú uvedené aj ďalšie funkcie, *VlozNaKoniec* (funkcia vloží prvok do zoznamu na koniec), *PorovnajS2* (funkcia bude porovnávať podľa dátumu, priezviska, mena), *Vypis* (funkcia vypíše celý zoznam), *Zmaz* (funkcia zmaže zoznam) a *ZmazPosledny* (funkcia zmaže zo zoznamu posledný prvok)

Riešenie v jazyku C

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
//-----------------datove struktury-------------------//
struct datum
    {    int d,m,y;
    };
struct TStudent
    {    char meno[32],priezvisko[32];
        datum datum_narodenia;
        char znamka_ZI;
    };
struct TPrvok
    {    TStudent student;
        TPrvok *dalsi;
    };
struct TZoznam
{
  TPrvok *prvy, *posledny;
```

```
};

//------------------funkcie----------------------------//

int JePrazdny(TZoznam z)
{
  return z.prvy == NULL;
}

void VypisPrvok(TPrvok *p)
{
    cout << p->student.priezvisko <<" "<<p->student.meno<<" ("<<p->student.znamka_ZI<<") ";
    cout<<p->student.datum_narodenia.d<<". "<<p->student.datum_narodenia.m<<". ";
    cout<<p->student.datum_narodenia.y<<endl;
}

void Vypis(TZoznam z)
{
  if (JePrazdny(z)) return;
  TPrvok *p = z.prvy;
  while (p->dalsi != NULL)
  {
    VypisPrvok(p);
    p = p->dalsi;
  }
  VypisPrvok(p);
}

void ZmazPosledny(TZoznam &z)
{
  if (z.prvy == NULL)  // prazdny zoznam
    return;

  TPrvok *p = z.prvy;
  if (p->dalsi == NULL)  // je len jeden prvok
  {
    delete p;
    z.prvy = z.posledny = NULL;
    return;
  }

  // najdeme predposledny prvok
  while (p->dalsi->dalsi != NULL)
    p = p->dalsi;
  delete p->dalsi;
  p->dalsi = NULL;
  z.posledny = p;
```

```
}

void Zmaz(TZoznam &z)
{
  while (!JePrazdny(z))
    ZmazPosledny(z);
}

void Nacitaj(TStudent &s,istream &vstup)
{
 vstup>>s.meno>>s.priezvisko>>s.znamka_ZI;
 vstup>>s.datum_narodenia.d>>s.datum_narodenia.m>>s.datum_narodenia.y;
}

void VlozNaKoniec(TZoznam &z,TStudent s)
{
 TPrvok *novy = new TPrvok;
  novy->student = s;
  novy->dalsi = NULL;

  if (z.prvy == NULL)  // prazdny zoznam
    z.prvy = novy;
  else
    z.posledny->dalsi = novy;
  z.posledny = novy;
}

int main()
{
  TZoznam zoznam;
  zoznam.prvy = zoznam.posledny = NULL;
  TStudent s;
  int n=0;

  ifstream vstup;
  vstup.open("prvaci.txt");

  Nacitaj(s,vstup);
  VlozNaKoniec(zoznam,s);
  while(s.datum_narodenia.d)
  {
          Nacitaj(s,vstup);
          if(s.datum_narodenia.d)
             VlozNaKoniec(zoznam,s);
          n++;
  }
  vstup.close();
```

```
  cout<<"--------vypis zoznamu--------------"<<endl;
  cout<<"Pocet studentov: "<<n<<endl;;
  Vypis(zoznam);
  Zmaz(zoznam);

  system("pause");
}
```

Usporiadaný lineárny zoznam (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

>Neusporiadaný lineárny zoznam
>Usporiadaný lineárny zoznam

Binárny strom

Numerické algoritmy

Zadanie

Vo vstupnom súbore máme zoznam študentov - v každom riadku sa nachádza meno, priezvisko, známka zo ZI, den, mesiac a rok narodenia. V poslednom riadku je 5 núl.

Zostavte program, ktorý zo súboru načíta všetkých študentov a vytvorí lineárny zreťazený zoznam, ktorého jednotlivé prvky budú obsahovať údaje jedného riadku. Prvky do zoznamu vkladajte tak, aby sa hneď pri vkladaní zotriedili. Triedenie bude podľa nasledujúcich požiadaviek.

1. Zoznam bude utriedený podľa abecedy (od A po Z)
2. Zoznam bude utriedený podľa priezviska
3. ak je priezvisko rovnaké, tak budeme triediť podľa mena
4. ak je meno rovnaké, tak budeme triediť podľa dátumu narodenia

Ďalej v programe budeme chcieť vyhľadať, či sa nejaký študent vyskytuje v zozname. Načítajte hľadaný reťazec a vyhľadajte daného študenta. Na toto vytvorte funkciu *hladaj*, ktorá bude zoznam prehľadávať a nájde všetky výskyty podľa zadaného kritéria.

Vzorový príklad

Vstup

Nasledujúce údaje sú načítané zo súboru

```
Jan Mrkvicka A 4 6 1985
Ferdinand Tell C 4 12 1986
Viliam Tell F 1 1 1987
johanka Z_arku C 5 4 1238
0 0 0 0 0 0
```

Nasledujúci údaj je načítaný z klávesnice

```
Tell
```

Výstup

```
Tell Ferdinand,C , 4.12.1986
Tell Viliam, F , 1.1.1987
```

Analýza riešenia

Budeme pokračovať v predchádzajúcom príklade (Neusporiadaný lineárny zoznnam (riešené príklady)).

Pri vkladaní údajov si musíme naprogramovať porovnávaciu funkciu, pomocou ktorej budeme porovnávať záznamy (študentov) aby sme ich mohli vkladať na správne miesto. Vo funkcii *PorovnajS(TStudent a, TStudent b)* porovnávame dvoch študentov, podľa zadaných kritérií. Funkcia vráti 1 ak je študent a v abecednom poradí na vyššej pozícii ako b, v opačnom prípade vráti hodnotu -1. Ak sú údaje rovnaké vráti 0 .

Pri vkladaní údajov rozoznávame 3 prípady: údaje vkladáme na začiatok zoznamu, na koniec zoznamu, alebo niekde (na správne miesto) do stredu zoznamu. Pri všetkých prípadoch musíme upraviť štrukrúru TZoznam, aby ukazovatele *prvy* a *posledny* ukazovali na začiatok, resp. na koniec zoznamu. Taktiež treba správne nastaviť ukazovatele *dalsi* (štruktúry TPrvok) aby sa zoznam niekde neprerušil.

Pri vyhľadávaní študenta budeme zoznamom postupne prechádzať od prvého prvku až pokiaľ prvok nenájdeme. Pri úspechu vráti funkcia ukazovateľ na prvok, ktorý obsahuje hľadaného študenta: *TPrvok* Najdi(TZoznam z, char *hladane,TPrvok *start=NULL,int podla_p=1)*. Parametrami funkcie sú: zoznam, v ktorom budeme hľadať, reťazec *hladane* - čo hľadáme. Podľa zadania máme vypísať všetky výskyty hľadaného reťazca. V prípade ak sú v zozname dvaja študenti s rovnakým priezviskom, funkcia nájde iba toho prvého. Aby funkcia pracovala univerzálnejšie, použijeme 3-ti parameter (Ukazovateľ na TPrvok), ktorý má význam: budeme hľadať od prvku *start*. Posledný argument má význam: ak bude *podla_p=1* budeme hľadať priezvisko, ak bude *podla_p=0*, budeme vyhľadávať podľa mena. Pri ďalšom hľadaní získame prvok *start* jednoducho: použijeme posledne nájdený prvok.

Pre výpis prvku môžeme použiť funkciu *VypisPrvok(TPrvok *p)*, ktorá bude vypisovať obsah prvku p (p obsahuje štrukúru TStudent, ktorá obsahuje položky meno, priezvisko, datum a znamka_ZI)

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
//----------------datove struktury------------------//
struct datum
    {    int d,m,y;
    };
struct TStudent
```

```
    {    char meno[32],priezvisko[32];
        datum datum_narodenia;
        char znamka_ZI;
    };
struct TPrvok
    {    TStudent student;
        TPrvok *dalsi;
    };
struct TZoznam
{
  TPrvok *prvy, *posledny;
};

//-------------------funkcie----------------------------//

int JePrazdny(TZoznam z)
{
  return z.prvy == NULL;
}

void VlozNaKoniec(TZoznam &z,TStudent s)
{
 TPrvok *novy = new TPrvok;
  novy->student = s;
  novy->dalsi = NULL;

  if (z.prvy == NULL)  // prazdny zoznam
    z.prvy = novy;
  else
    z.posledny->dalsi = novy;
  z.posledny = novy;
}
// nova funkcia !!!
int PorovnajS(TStudent a, TStudent b)
{  // podla priezviska, mena, datumu
 int porovanie;
 porovanie=strcmp(strlwr(a.priezvisko),strlwr(b.priezvisko));
 if(porovanie!=0) return porovanie;
 porovanie=strcmp(strlwr(a.meno),strlwr(b.meno));
 if(porovanie!=0) return porovanie;
 long da,db;

da=a.datum_narodenia.d+a.datum_narodenia.m*100+a.datum_narodenia.y*10000;

db=b.datum_narodenia.d+b.datum_narodenia.m*100+b.datum_narodenia.y*10000;
 if(da>db) return 1;
 if(da<db) return -1;
```

```
 return 0;
}

// nova funkcia !!!
void Vloz(TZoznam &z, TStudent x)  // vlozi prvok zotriedene
{
  TPrvok *novy = new TPrvok;
  novy->student=x;
  novy->dalsi = NULL;
  if (z.prvy == NULL)  // prazdny zoznam
    z.prvy = z.posledny = novy;
  else
      if ( PorovnajS(x,z.prvy->student)==-1 )  // x < z.prvy->x / treba na
zaciatok zoznamu
      {
        novy->dalsi = z.prvy;
        z.prvy = novy;
      }
      else
          if (PorovnajS(x,z.posledny->student)==1 )  // x > z.posledny->x
/ treba na koniec zoznamu
          {
            z.posledny->dalsi = novy;
            z.posledny = novy;
          }
          else
          {
                TPrvok *p1 = z.prvy, *p2 = z.prvy->dalsi;
                while (PorovnajS(p2->student,x)==-1)  //p2->x < x
                {
                  p1 = p2; p2 = p2->dalsi;
                }
                p1->dalsi = novy;
                novy->dalsi = p2;
          }
}
// nova funkcia !!!
TPrvok* Najdi(TZoznam z, char *hladane,TPrvok *start=NULL,int
podla_p=1)
{ // ak je posledny parameter uvedeny, bude sa hladat od toho prvku
dalej
  // inak sa hlada od zaciatku

  TPrvok *p;
  if(start==NULL)
    p = z.prvy;
  else
```

```
    p=start->dalsi;
  char hladane_pr[32];

  while (p != NULL)
  { if(podla_p)
        strcpy(hladane_pr,p->student.priezvisko);  // aby som si nezmenil
 povodne priezvisko
    else
        strcpy(hladane_pr,p->student.meno);  // aby som si nezmenil
povodne priezvisko
    if (strcmp(strlwr(hladane_pr),strlwr(hladane))==0) return p;
    p = p->dalsi;
  }
  return NULL;
}

void VypisPrvok(TPrvok *p)
{
    cout << p->student.priezvisko <<" "<<p->student.meno<<" ("<<p->student.znamka_ZI<<") ";
    cout<<p->student.datum_narodenia.d<<". "<<p->student.datum_narodenia.m<<". ";
    cout<<p->student.datum_narodenia.y<<endl;
}

void Vypis(TZoznam z)
{
  if (JePrazdny(z)) return;
  TPrvok *p = z.prvy;
  while (p->dalsi != NULL)
  {
    VypisPrvok(p);
    p = p->dalsi;
  }
  VypisPrvok(p);
}

void ZmazPosledny(TZoznam &z)
{
  if (z.prvy == NULL)  // prazdny zoznam
    return;

  TPrvok *p = z.prvy;
  if (p->dalsi == NULL)  // je len jeden prvok
  {
    delete p;
    z.prvy = z.posledny = NULL;
    return;
  }
```

```
  // najdeme predposledny prvok
  while (p->dalsi->dalsi != NULL)
    p = p->dalsi;
  delete p->dalsi;
  p->dalsi = NULL;
  z.posledny = p;
}

void Zmaz(TZoznam &z)
{
  while (!JePrazdny(z))
    ZmazPosledny(z);
}

void Nacitaj(TStudent &s,istream &vstup)
{
 vstup>>s.meno>>s.priezvisko>>s.znamka_ZI;
 vstup>>s.datum_narodenia.d>>s.datum_narodenia.m>>s.datum_narodenia.y;
}
int main()
{
  TZoznam zoznam;
  zoznam.prvy = zoznam.posledny = NULL;
  TStudent s;
  int n=0;

  ifstream vstup;
  vstup.open("prvaci.txt");

  Nacitaj(s,vstup);
  Vloz(zoznam,s);
  while(s.datum_narodenia.d)
  {
          Nacitaj(s,vstup);
          if(s.datum_narodenia.d)
             Vloz(zoznam,s);
          n++;
  }
  vstup.close();
  cout<<"--------vypis zoznamu--------------"<<endl;
  cout<<"Pocet studentov: "<<n<<endl;;
  //  Vypis(zoznam);
  int kriterium;
  cout<<"Podla coho chces vyhladavat? meno(0)/priezvisko(1)";
  cin>>kriterium;
  cout<<endl<<"Zadaj retazec, ktory chces vyhladat: ";
```

```
  char przv[32];
  cin>>przv;

  cout<<"-----------------------------"<<endl;
  TPrvok *ja=Najdi(zoznam,przv,NULL,kriterium);  // hladame prvy vyskyt
  int pocet=0;
  while(ja)
  {   pocet++;
      if(ja)
            VypisPrvok(ja);
      ja=Najdi(zoznam,przv,ja,kriterium);  // hladame dalsie vyskyty od
 podledneho najdeneho studenta
  }
  cout<<"-----------------------------"<<endl;
  cout<<"Pocet najdenych studentov: "<<pocet<<endl;
  Zmaz(zoznam);
  system("pause");
}
```

Binárny strom - Morseova abeceda (riešené príklady)

UPOZORNENIE: Článok nebolo možné vykresliť - na výstup sa zapíše čistý text.
Možné príčiny problému sú: (a) chyba v softvéri pdf-writer (b) problematická MediaWiki syntax článku (c) príliš široká tabuľka

Algoritmy a programovanieAlgoritmy a programovanie - zbierka úlohŠtruktúry (riešené príklady)ŠtruktúryRekurzia (riešené príklady)RekurziaDynamická alokácia pamäti (riešené príklady)Dynamická alokácia pamätiVyhľadávanie (riešené príklady)VyhľadávanieTriedenie(riešené príklady)TriedenieLineárny zoznam (riešené príklady)Lineárny zoznamBinárny strom (riešené príklady)Binárny strom>Binárny strom - Morseova abeceda (riešené príklady)Binárny strom - Morseova abeceda >Binárny strom - Huffmanovo kódovanie (riešené príklady)Binárny strom - Huffmanovo kódovanieNumerické algoritmy (riešené príklady)Numerické algoritmyZadaniehttp://cec.truni.sk/stoffov/dynamicke-udajove-struktury/Cast2/ Vytvorte program na dekódovanie správy napísanej v Morseovej abecede. Správu prečítajte z textového súboru. Analýza úlohy Pri dekódovaní využijeme binárny vyhľadávací strom. V jeho vrcholoch budú uložené jednotlivé písmená abecedy, v koreni je ako špeciálny znak medzera. Pre potreby tohto príkladu si upravíme dátovú štruktúru binárny strom nasledovne dátová časť bude jeden znak (znak kódovanej abecedy) smerníky lavy a pravy nahradíme smerníkmi bodka a ciarka. Smerníky majú i naďalej význam ľavého a pravého potomka, ale pre ľahšiu orientáciu v kódovacom strome si ich premenujeme. struct TUzol{ char znak; TUzol *bodka, *ciarka; }; Zo súboru čítame postupnosti bodiek a čiarok. Znak bodka znamená presun ukazovateľa na ľavý podstrom, znak čiarka na pravý podstrom. Bnárny strom reprezentujúci kódovanie morseovej abecedyDekódovanie Morseovej abecedy Ak prečítame zo súboru kód --.. pôjdeme z koreňa vpravo, vpravo, vľavo, vľavo. Skončíme vo vrchole s písmenom z. Takýmto spôsobom pre každý

prečítaný kód nájdeme rýchlo zodpovedajúce písmeno. Princíp dekódovanie je v nasledujúcom pseudokóde: procedure dekoduj(TUzol strom) begin p - pomocný prvok TUzol c - premenná typu znak p = strom // p je pomocná premenná s ktorou prechádzame strom pokiaľ sa zo súboru načítal znak (do premennej c) begin ak je c znak '.' tak p=p->bodka // presunieme sa na ľavý podstrom inak ak je c znak '-' tak p=p->ciarka // presunieme sa na pravý podstrom inak // načátali sme oddeľovací znak begin vypíš hodnotu uzla p p=strom //v ďalšom cykle budeme dekódovať ďalší znaka musíme začať od koreňa stromu end end end Pred čítaním kódu ďalšieho písmena nastavíme ukazovateľ opäť na koreň stromu! Pre jednoduchosť budeme predpokladať, že na vstupe je správa v dohodnutom formáte - za kódom písmena ako oddeľovač nasleduje vždy znak /, za každým slovom aspoň dva znaky //. Vytvorený program má tri časti - vytvorenie dekódovacieho stromu, otvorenie súboru pre čítanie a postupné dekódovanie správy. Vzorový príkladVzorový vstup-../-.--/-./.-/--/../-.-./-.-/.//.../-/.-./..-/-.-/-/..-/.-./-.--/ Vzorový výstupdynamicke struktury Riešenie v jazyku C #include <stdlib.h> #include <iostream.h> #include <fstream.h> struct TUzol{ char znak; TUzol *bodka, *ciarka; }; //--- TUzol *vytvor(char z, TUzol *b, TUzol *c) { TUzol *novy = new TUzol; novy->znak = z; novy->bodka = b; novy->ciarka = c; return(novy); } //--- void dekoduj(TUzol *strom) { ifstream fr; fr.open("sprava.txt"); TUzol *p; char c; p = strom; while (fr>>c) { if (c == '.') p = p->bodka; else if (c == '-') p = p->ciarka; else { cout<<p->znak; p = strom; } } fr.close(); } //--- void vypis(TUzol *v) { if (v != NULL) { vypis(v->bodka); cout<<v->znak; vypis(v->ciarka); } } //--- int main() { TUzol *strom = NULL; /* vytvorenie dekódovacieho stromu */ strom = vytvor(' ', vytvor('e', vytvor('i', vytvor('s', vytvor('h', NULL, NULL), vytvor('v', NULL, NULL)), vytvor('u', vytvor('f', NULL, NULL), NULL)), vytvor('a', vytvor('r', vytvor('l', NULL, NULL), NULL), vytvor('w', vytvor('p', NULL, NULL), vytvor('j', NULL, NULL)))), vytvor('t', vytvor('n', vytvor('d', vytvor('b', NULL, NULL), vytvor('x', NULL, NULL)), vytvor('k', vytvor('c', NULL, NULL), vytvor('y', NULL, NULL))), vytvor('m', vytvor('g', vytvor('z', NULL, NULL), vytvor('q', NULL, NULL)), vytvor('o', NULL, NULL)))); cout<<"Sprava: "; dekoduj(strom); return 0; } Vysvetlenie zdrojového kóduHlavný program main Ako prvé je potrebné vytvoriť binárny strom (riadok 56). Ďalší krok je vytvoriť kódovací strom Morseovej abecedy ako je na obrázku. Na to použijeme funkciu vytvor. Zaujímavý je spôsob jej použitia. Funkcia vytvor má 3 parametre: znak, ktorý predstavuje dekódovanú postupnosť bodiek a čiarok. Ďalej sú to dva smerníky na potomkov aktuálneho uzla. Ľavý potomok (bodka) a pravý potomok (ciarka). Na riadku 59 je prvý krát použitá funkcia vytvor. Pri tomto volaní vytvárame koreň stromu. Znak obsiahnutý v koreni je medzera. (' '). Druhým parametrom funkcie má byť uzol ktorý je ľavý potomok. Namiesto vytvorenia nového uzla a jeho následného vloženia do stromu strom, opäť použijeme funkciu vytvor, pomocou ktorej vytvárame znak 'e'. Druhý parameter funkcie (pravý potomok uzla - ciarka) je znak 't' - riadok 67. V nasledujúcom kóde je uvedený ekvivalentný zápis pri vytváraní stromu. V tomto príklade nebude použité vnáranie funkcie vytvor. TUzol *strom = NULL; TUzol *h = vytvor('h',NULL,NULL); TUzol *v = vytvor('v',NULL,NULL); TUzol *f = vytvor('f',NULL,NULL); TUzol *s = s=vytvor('s',h,v); TUzol *u = u=vytvor('u',f,NULL); TUzol *i = vytvor('i',s,u); TUzol *l = vytvor('j',NULL,NULL); TUzol *p = vytvor('p',NULL,NULL); TUzol *j = vytvor('j',NULL,NULL); TUzol *r = vytvor('r',l,NULL); TUzol *w = vytvor('w',p,j); TUzol *a = a=vytvor('a',r,w); TUzol *e = a=vytvor('e',i,a); TUzol *b = vytvor('b',NULL,NULL); TUzol *x = vytvor('x',NULL,NULL); TUzol *c = vytvor('c',NULL,NULL); TUzol *y = s=vytvor('y',NULL,NULL); TUzol *z = s=vytvor('z',NULL,NULL); TUzol *q = s=vytvor('q',NULL,NULL); TUzol *d = u=vytvor('d',b,x); TUzol *k = vytvor('k',c,y); TUzol *g = vytvor('g',z,q); TUzol *o = s=vytvor('o',NULL,NULL); TUzol *m = u=vytvor('m',g,o); TUzol *n = vytvor('n',d,k); TUzol *t = vytvor('t',n,m); strom= vytvor(' ',e,t); Funkcia vytvorOpis funkcie Funkcia vytvorí a vráti nový uzol binárneho stromu. Hodnota časti znak nového uzla bude použitá z 1. parametra funkcie. Parametre funkciez - znak, ktorý bude obsahovať nový uzol. Tento znak je zakódované písmeno v kódovacom strome Morseovej abecedy. b - smerník na ľavého potomka nového uzla. Určuje ďalší kódový znak v kódovacom strome. Ďalší znak pokračuje kódom '.' (bodka) c - smerník na pravého potomka nového uzla. Určuje ďalší kódový znak v kódovacom strome. Ďalší znak pokračuje kódom '-' (čiarka) Poznámka: ak má nový uzol

potomkov (podľa kódovacieho stromu napríklad znak 'e' má potomkov 'i' a 'a'). Tieto uzly už musia byť vytvorené. Smerníky na tieto existujúce uzly sú v parametroch funkcie vytvor.Návratová hodnotaFunkcia vráti smerník na novo vytvorený uzol kódovacieho stromu Morseovej abecedy. Funkcia dekoduj Na začiatku funkcie je otvorený súbor so vstupnými údajmi. Potom nasleduje samotné dekódovanie. Princíp dekódovania bol vysvetlený v časti #Dekódovanie Morseovej abecedyDekódovanie Morseovej abecedyOdkazyhttp://en.wikipedia.org/wiki/Morse_code

---

Binárny strom - Huffmanovo kódovanie (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

>Binárny strom - Morseova abeceda

>Binárny strom - Huffmanovo kódovanie

Numerické algoritmy

Zadanie

Vytvorte program na zakódovanie správy pomocou Huffmanovho kódovania. Text načítajte z textového súboru.

Analýza

Huffmanovo kódovanie je metóda bezstratového kódovania dát založená na entropii kódovaného textu. Pri tvorbe samotného Huffmanovho kódu využijeme dátovú štruktúru binárny strom. Pri vytváraní Huffmanovho kódu potrebujeme poznať celú abecedu, ktorá bola použitá v správe a ďalej je potrebné vedieť pravdepodobnosti výskytov znakov abecedy v texte. Pre získanie týchto pravdepodobností môžeme postupovať dvoma spôsobmi:

1. Pravdepodobnosti výskytu znakov v texte si vypočítať z:
   1. textu, ktorý budeme kódovať,
   2. veľkej vzorky textu v danom (prirodzenom) jazyku. Prirodzený jazyk sa myslí slovenčina, čeština, angličtina.
2. Pravdepodobnosti výskytu znakov v texte použiť známe pravdepodobnosti pre daný prirodzený jazyk.

Výsledkom tejto analýzy bude súbor "znaky.txt" v ktorom bude zoznam všetkých znakov abecedy spolu s ich pravdepodobnosťou výskytu v texte.

Výpočet pravdepodobnosti výskytu znakov v texte

Uvažujme vstupný textový súbor ("text.txt") v ktorom je text v prirodzenom jazyku. Úlohou je zistiť pravdepodobnosť výskytu všetkých znakov v danom texte. Inak povedané, je treba zistiť pravdepodobnosť výskytu všetkých znakov abecedy. Kvôli zjednodušeniu bude súbor text.txt obsahovať len veľké písmená abecedy a nebude obsahovať slovenské písmená(v súbore nie sú písmená ľ,š,č,ť,ž,ý,á,í,é,ä,ú,ň,Ĺ,ď).

Získanie pravdepodobnosti výskytu je jednoduché. Stačí spočítať početnosť všetkých znakov. Využijeme fakt, že vstupnú abecedu tvoria len veľké písmená abecedy bez znakov s mäkčeňmi a dĺžňami, ďalej číslice a interpunkčné znamienka (,.-/()+-")

```
void pravdepodobnost_znakov()
{
     ifstream in;
     in.open("text.txt");
     if(!in) return;

     int znaky[64];
     for(int i=0;i<64;i++)
          znaky[i]=0;
     char c;
     do
     {    c=in.get();
          znaky[c-' ']++;
     }while(c>=0);
     in.close();
}
```

V tabuľke ASCII má znak medzery hodnotu 32. Tento znak je zároveň prvým nie kontrolným znakom. Posledný znak, ktorého pravdepodobnosť budeme počítať je znak apostrov, ktorý má hodnotu 96. Preto nám stačí vytvoriť pole 64 znakov (riadok č.7). Na riadkoch 11 až 14 je načítavanie jedného znaku zo súboru, až pokým nenarazíme na koniec súboru. Pri načítaní znaku zvýšime počet jeho výskytov o 1 (riadok 13).

Úlohou je ale vygenerovať súbor, v ktorom budú znaky abecedy usporiadané podľa ich pravdepodobnosti výskytu v texte od najmenšieho po najväčšie. Upravíme teda prechádzajúci kus kódu nasledovne:

```
struct Znak{
       char znak;
       double pp;
       };

int porovnajZ(const void *a, const void *b)
{

   if( (*(Znak*)a).pp > (*(Znak*)b).pp ) return 1;
   if( (*(Znak*)a).pp < (*(Znak*)b).pp ) return -1;
    return 0;
}
void pravdepodobnost_znakov()
{
     ifstream in;
```

```
      in.open("text.txt");
      int n=64;
      if(!in) return;
      int znaky[64];
      for(int i=0;i<64;i++)
           znaky[i]=0;
      char c;
      do
      {    c=in.get();
          if(c==' ')
            c='_';
          znaky[c-' ']++;
      }while(c>=0);
      double suma=0;
      for(int i=0;i<n;i++)
      {  // len kontrolny vypis v pripade ladenia programu
      //    cout<<(char)(i+32)<<" "<<znaky[i]<<endl;
         suma+=znaky[i];
      }
      in.close();

      Znak abc[n];
      for(int i=0;i<n;i++)
      {
         abc[i].znak=(char)(i+32);
         abc[i].pp=znaky[i]/suma;
      }
      qsort(abc,n,sizeof(abc[0]),porovnajZ);
      ofstream out;
      out.open("znaky.txt");
      int index=0;
      for(index=0;abc[index].pp==0;index++);
      out<<64-index<<endl;
      for(int i=index;i<n;i++)
      {
         out<<abc[i].znak<<"\t"<<abc[i].pp<<endl;
      }
      out.close();
}
```

Zmeny v kóde sú až na riadku 35. Pre potreby zotriedenia znakov podľa ich pravdepodobností musíme tieto znaky uložiť do vhodnej štruktúry, kde by bola informácia o znaku a jeho pravdepodobnosti výskytu. Takáto štruktúra je na riadku 1. Štruktúra *Znak* obsahuje premennú *znak* typu char a premennú *pp* typu *double*, čo je vlastne pravdepodobnosť znaku.

Na riadku 35 si vytvoríme pole štruktúr Znak o veľkosti 64. Toto pole naplníme hodnotami (riadky 35 až 40) údajmi získanými analýzou textu. Prvý prvok poľa *abc* reprezentuje znak medzera ' '. Na zotriedenie tohoto poľa využijeme funkciu qsort. Na riadkoch 45 až 48 je zápis údajov do súboru vo formáte aký bol dohodnutý na začiatku. Znaky,

ktoré mali v skúmanom texte nolový výskyt do súboru znaky.txt nezapisujeme. Na toto nám poslúži cyklus na riadku 45, ktorý premennú index nastaví na index prvého znaku v poli abc, ktorý sa vo vstupnom texte vyskytuje.

Známe pravdepodobnosti výskytu znakov v texte v prirodzenom jazyku

Pravdepodobnosti pre anglickú abecedu

```
a  .082
b  .015
c  .028
d  .043
e  .127
f  .022
g  .02
h  .061
i  .07
j  .002
k  .008
l  .04
m  .024
n  .067
o  .075
p  .019
q  .001
r  .06
s  .063
t  .091
u  .028
v  .01
w  .023
x  .001
y  .02
z  .001
```

Návrh vhodnej dátovej štruktúry

Pri tvorbe Huffmanovho kódu budeme vytvárať binárny strom, čo bude vlastne kódovací strom Huffmanovho kódu. Listy (koncové uzly) tohoto binárneho stromu budú reprezentovať abecedu kódovania. V uzle budú teda informácie o konkrétnom znaku a jeho pravdepodobnosti výskytu. Po vytvorení takéhoto kódovacieho stromu musíme znakom vstupnej abecedy prideliť kódové slová. Kódové slovo pozostáva iba z jednotiek a núl. Do štruktúry opisujúcej znak abecedy ešte pridáme časť "kódové slovo", v ktorom bude uložený samotný Huffmanov kód pre daný znak abecedy.

```
struct THuff
{   char znak;
    float pp;
    char *kodove_slovo;
};

struct TUzol
{     THuff *data;
```

```
        TUzol *lavy,*pravy;
};
```

- THuff - dátová štruktúra opisujúca znak vstupnej abecedy:
  - znak - samotný znak abecedy
  - pp - pravdepodobnosť výskytu znaku
  - kodove_slovo - Huffmanov kód pre daný znak. Doplní sa až po vytvorení kódovacieho stromu.
- TUzol - uzol binárneho, resp. kódovacieho stromu.
  - data - smetník na dátovú časť uzla stromu - THuff
  - lavy, pravy - smerníky na ľavý a pravý podstrom.

Vzorový príklad

Vstupný údaj pre tento príklad je tabuľka pravdepodobností výskytu znakov abecedy v texte. Túto tabuľku si môžeme vypočítať (zo súboru *text.txt*) pomocou funkcie *pravdepodobnost_znakov*, alebo použiť už pripravenú tabuľku.

Vzorový vstup

Formát súboru *znaky.txt* je nasledovný: prvý údaj je počet znakov abecedy (n). V ďalších n riadkoch je znak abecedy a jeho pravdepodobnosť výskytu.

```
8
E 0.05
G 0.05
A 0.1
C 0.1
F 0.1
H 0.1
B 0.2
D 0.3
```

Vzorový výstup

Huffmanov kód pre abecedu definovanú v súbore znaky.txt.

Znak	**Kód**
E	0000
G	0001
H	001
A	010
F	011
C	100
B	101
D	11

Postup pri riešení

Riešenie si rozdelíme na 2 hlavné časti a niekoľko podúloh:

1. Fáza: Fáza redukcie (smer hore)

1. Znaky kódovanej usporiadame podľa pravdepodobnosti výskytu
2. Postupne redukujeme abecedu spojením dvoch znakov s najmenšou pravdepodobnosťou do jedného zástupného znaku s pravdepodobnosťou rovnou súčtu týchto dvoch pp.
3. Ak obsahuje redukovaná abeceda 2 zástupné znaky – krok 4, inak krok 2

2. Fáza: Fáza zostavenia kódu (smer dole)

4. Dvom zástupným znakom redukovanej abecedy priradíme kódové slová 0 a 1. Kódové slová priraďujeme od vrcholu kódovacieho stromu.
5. Potupne, keď prechádzame smerom dole, pridávame kódovým znakom, ktoré sú potomkami zástupného znakom slová 0 a 1
6. Pokiaľ sa nedostaneme k pôvodnej abecede, tak krok 5.

Riešenie v jazyku C

```
int main(int argc, char *argv[])
{
 pravdepodobnost_znakov();
 ifstream in;
 char subor[]="znaky.txt";
 in.open(subor);
 int n;                          // pocet znakov abecedy
 TUzol **abc;                    // pole ukazatelov na TUzol
 if(in)
 {   in>>n;
     abc=new TUzol*[n];
     for(int i=0;i<n;i++)
     {   abc[i]=new TUzol;
         abc[i]->data=new THuff;
         abc[i]->lavy=abc[i]->pravy=NULL;
         in>>abc[i]->data->znak;
         in>>abc[i]-> data-> pp;
     }
 }
 else
   return 1;                             // nepodarilo sa otvorit
vstupny subor

 int min1,min2;
 min1=min2=0;
 TUzol *strom=new TUzol;  // binarny strom, ktory budeme vytvarat
 strom->data = new THuff;
 while(n>1)                          // vytvaranie noveho binarneho stromu
 z pola TUzlov
 {    Hladaj2Min(abc,n,min1,min2);
```

```
     strom=VlozDoStromu(abc[min1],abc[min2]);
     UsporiadajKodoveSlova(abc,strom,min1,min2,n);
 }

 char *slovo=new char[32];
 slovo[0]=0;
 ZapisKodoveSlova(strom,slovo);      // priradenie kodov jednotlivym
znakom
 vypis(strom);            // vypis zakodovanych znakov

 zmazStrom(strom);

 //zmazanie pola smernikov na uzly
 for(int i=0;i<n;i++)
 {
   delete abc[i]->data;
   delete abc[i];
 }
 delete []abc;
}
```

Vysvetlenie kódu

riadok 3

výpočet pravdepodobnosti výskytu znakov vo vstupnej abecede. V prípade, ak máme pripravený súbor znaky.txt s pravdepodobnosťami, tak tento riadok treba zakomentovať alebo odstrániť.

riadok 8-11

Na uloženie vstupných údajov si vytvoríme pole smerníkov na štruktúru TUzol. Pole smerníkov na TUzol je výhodnejšie použiť vzhľadom na neskoršie operácie kopírovania uzlov. *abc* je dvojitý smerník na dátovú štruktúru TUzol. Na riadku 11 pre abc alokujeme n smerníkov na štruktúru TUzol

riadok 12-19

Načítanie vstupných údajov do dátovej štruktúry abc - pole smerníkov na TUzol. Pre dátovú časť štruktúry TUzol si treba alokovať miesto (riadok 14), pretože v definícii je premenná data definovanná ako smerník na štruktúry THuff. Na riadku 16 je načítanie znaku a na riadku 17 je načítanie jeho pravdepodobnosti.

riadok 25

strom - koreň binárneho stromu, ktorý bude tvoriť kódovací strom.

riadok 27-30

1. Fáza riešenia. Premenná n má význam počet znakov v postupne redukovanej abecede. Abecedu redukujeme, pokiaľ je n väčšie ako 1.

riadok 35

2. Fáza riešenia. Funkcia *ZapisKodoveSlova* pridelí znakom abecedy (listy stromu) do časti *kodove_slovo* kódové slová.

riadok 38-46

zmazanie binárneho stromu a zmazanie poľa smerníkov na šštruktúru TUzol, ktorá bola využitá pre uloženie vstupných znakov abecedy.

Ďalšie funkcie

void Hladaj2Min(TUzol **abc,int n,int &min1,int &min2)

```
void Hladaj2Min(TUzol **abc,int n,int &min1,int &min2)
{
   int i,index_i=0;
   float mmin1,mmin2;         // hodnoty pp dvoch min prvkov
   min1=min2=0;               // indexy, ktore hladame
   mmin1=mmin2=1;
   for(i=0;i<n;i++)
      if(abc[i]->data->pp < mmin1 )  // hladanie prveho minima
      { min1=i;
        mmin1=abc[i]->data-> pp;
      }
   for(i=0;i<n;i++)  // hladanie 2 minima, neberieme v uvahu prvok s indexom min1
      if((abc[i]->data-> pp <= mmin2) && (i!=min1) )
      { min2=i;
        mmin2=abc[i]->data-> pp;
      }
}
```

Opis funkcie:

Funkcia hľadá 2 prvky v poli vstupných znakov (pole smerníkov na TUzol) 2 prvky s najmenšou pravdepodobnosťou výskytu.

Parametre:

*TUzol **abc* - pole smerníkov na TUzol. každý prvok poľa obsahuje v dátovej časti znak vstupnej abecedy a pravdepodobnosť výskytu

int n - veľkosť poľa abc

int &min1 - výstupný parameter. Index prvého prvku poľa abc s najmenšou pravdepodobnosťou výskytu znaku v texte

int &min2 - výstupný parameter. Index druhého prvku poľa abc s najmenšou pravdepodobnosťou výskytu znaku v texte

Návratová hodnota:

žiadna

TUzol* VlozDoStromu(TUzol *a,TUzol *b)

```
TUzol* VlozDoStromu(TUzol *a,TUzol *b)
{  //vytvaram kodovaci strom zospodu hore
  TUzol *otec=new TUzol;
  otec->data = new THuff;
  otec->lavy=a;
  otec->pravy=b;
  otec->data->pp= a->data->pp + b->data->pp;
  otec->data->znak=0;
  return otec;
}
```

Opis funkcie:

Funkcia vytvorí rodičovský uzol uzlov *a* a *b*. Dátová časť rodičovského uzla neobsahuje žiaden znak (riadok 8) a pravdepodobnosť výskytu tohoto 'zástupného' znaku je vypočítaná ako súčet pravdepodobností výskytov jeho dvoch potomkov (riadok 7).

Parametre:

*TUzol *a,TUzol *b* - vstupné parametre funkcie. Uzly *a* a *b* sú existujúce uzly (TUzol)

Návratová hodnota:

Smerník na novo vytvorený uzol kódovacieho stromu.

void UsporiadajKodoveSlova(TUzol **abc,TUzol *novy,int min1,int min2,int &n)

Pri tvorbe stromu vznikajú v poli prvkov prázdne miesta, ktoré treba zrušiť. Namiesto dvoch uzlov, z ktorých sme vytvorili nový uzol, ktorého pravdepodobnosť výskytu znaku v texte je rovná súčtu pravdepodobností výskytov týchto znakov, vložíme na správne miesto nový uzol.

Nový uzol

```
void UsporiadajKodoveSlova(TUzol **abc,TUzol *novy,int min1,int
min2,int &n)
{
   int i;
   abc[min1]=novy;           // na index min1 vlozim novy uzol
   for(i=min2;i<n-1;i++)     // ostatne uzly posuniem o 1 dolava
     abc[i]=abc[i+1];
   n--;               // tym padom sa mi zmensi pocet uzlov v poli
  abc[n]=NULL;
}
```

Opis funkcie:

Po vytvorení nového uzla v kódovacom strome treba tento nový uzol vložiť na správne miesto do poľa uzlov. Tento nový uzol sa vloží do poľa abc na miesto s indexom min1. Prvok na indexe min2 už nepatrí do tohoto poľa, lebo je potomkom novo vytvoreného uzla. Preto treba preusporiadať pole abc ako ukazujú nasledujúce obrázky:

Parametre:

*TUzol **abc* - pole smerníkov na TUzol v ktorom sú uložené znaky vstupnej abecedy a v priebehu programu sa tu ukladajú zástupné znaky.

*TUzol *novy* - smerník na nový uzol, ktorého potomkovia sú uzly poľa *abc* na indexoch *min1* a *min2*

int min1,int min2 - indexy svoch prvkov s najmenšou pravdepodobnosťou výskytu znaku v texte

int &n - výstupný parameter : veľkosť poľa *abc*.

Návratová hodnota:

žiadna

void ZapisKodoveSlova(TUzol *ks,char *hodnota)

```
// funkcia zmaze posledny znak v retazci 'hodnota'
void substring(char *hodnota)
{
    hodnota[strlen(hodnota)-1]=0;
}
```

```
void ZapisKodoveSlova(TUzol *ks,char *hodnota)
{
    if(!ks) return;
    else
    { // kodove slovo zapisujeme len pre listy stromu
      if(ks->lavy==NULL && ks->pravy==NULL)
      {       ks->data->kodove_slovo=new char[strlen(hodnota)];
              strcpy(ks->data->kodove_slovo,hodnota);
      }
       ZapisKodoveSlova(ks->lavy,strcat(hodnota,"0"));
        substring(hodnota);
       ZapisKodoveSlova(ks->pravy,strcat(hodnota,"1"));
        substring(hodnota);
   }
}
```

Opis funkcie:

Listom kódovacieho stromu priradí kódové slová Huffmanovho kódu.

ZapisKodoveSlova je rekurzívna funkcia, ktorá ľavému potomkovi priradí znak 0 a pravému potomkovi znak 1. Ak sa pri rekurzívnom volaní narazí na list stromu (riadok 12), tak sa do dátovej časti daného uzla zapíše kódové slovo, ktoré je v parametri funkcie *hodnota*. Na riadku 13 je alokázia potrebného miesta pre toto kódové slovo a na riadku 14 je skopírovanie tohto reťazca do časti *kodove_slovo*. Rekurzívne volanie sa ukončí ak narazíme na neexistujúci uzol (riadok 9). Pomocná funkcia *substring* zmaže z reťazca *hodnota* posledný znak. Je to potrebné kvôli spätnému rekurzívnemu volaniu, keďže pri rekurzívnom volaní k tomuto parametru pridávame ďalšie znaky.

Parametre:

*TUzol *ks* - koreň kódovacieho stromu

*char *hodnota* - kódové slovo pre daný uzol. Na začiatku je je premenná *hodnota* prázdny reťazec.

Návratová hodnota:

Žiadna. Po skončení funkcie všetky listy stromu *ks* obsahujú reťazec s hodnotou kódového slova pre daný znak.

void zmazStrom(TUzol *uzol)

```
void zmazStrom(TUzol *uzol)
{
  if (!uzol) return;
  if (uzol->lavy)   // ak existuje lavy uzol
      zmazStrom(uzol->lavy);
  if (uzol->pravy) // ak existuje pravy uzol
       zmazStrom(uzol->pravy);
  delete uzol;
}
```

Opis funkcie:

Zmaže binárny strom. Je použitý algoritmus postorder

Parametre:

*TUzol *uzol* . binárny strom, ktorý sa má zmazať.

Návratová hodnota:

Žiadna

void vypis(TUzol *s)

```
void vypis(TUzol *s)
{      if(!s)
          return;
       else
       {
          vypis(s->lavy);
        if(s->data->znak)
        {
           cout<<s->data->znak<<" | ";
             cout<<s->data->kodove_slovo;
             cout<<endl;
          }
          vypis(s->pravy);
       }
}
```

Opis funkcie:

Vypíše binárny, resp. kódovací strom. Je použitý algoritmus inorder

Parametre:

*TUzol *s* . binárny strom, ktorý sa má vypísať

Návratová hodnota:

Žiadna

Prílohy

Vstupný súbor je upravený text z http://sk.wikipedia.org/wiki/Dejiny_Ko%C5%A1%C3%ADc.

vstupný súbor text.txt [1]

Numerické algoritmy (riešené príklady)

Algoritmy a programovanie - zbierka úloh

Štruktúry

Rekurzia

Dynamická alokácia pamäti

Vyhľadávanie

Triedenie

Lineárny zoznam

Binárny strom

Numerické algoritmy
::Algoritmy numerickej interpolácie
::Algoritmy numerickej aproximácie
::Numerické integrovanie
::Numerické derivovanie

Algoritmy numerickej interpolácie

Zadanie: Riešte problém interpolácie dát. K dispozícii máme n bodov v priestore (x,y). Našou úlohou bude vypočítať vhodnú interpolačnú krivku.

Algoritmy numerickej aproximácie

Zadanie 1: Riešte problém problém aproximácie dát. K dispozícii máme n bodov v rovine (ich súradnice x a y). Úlohou bude vypočítať rovnicu aproximujúcej krivky metódou najmenších štvorcov pre:

1. Lineárnu aproximáciu v tvare
2. Logaritmickú aproximáciu v tvare
3. Exponenciálnu aproximáciu v tvare
4. Mocninovú aproximáciu v tvare

Zadanie 2:

Vytvorte funkciu, ktorá pre zadaná vstupy vypočíta všetky typy aproximácií (lineárnu, logaritmickú, exponenciálnu a mocninovú) a určí ktorá aproximácia je optimálna pre zadané vstupné body.

Numerické integrovanie

Zadanie 3:

Numerické derivovanie

Zadanie 4: Vypočítajte hodnoty prvej derivácie interpolačného polynómu v Newtonovom a Lagrangeovom tvare. Porovnajte tieto hodnoty. Pre výpočet derivácie použite metódu rozdielových diferencií.

Zdroje článkov a prispievatelia

Algoritmy a programovanie *Zdroj*: http://www.kiwiki.info/index.php?oldid=11372 *Prispievatelia*: Juraj

Dátový typ *Zdroj*: http://www.kiwiki.info/index.php?oldid=6641 *Prispievatelia*: Juraj

Štruktúry (jazyk C) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6645 *Prispievatelia*: Juraj

Rekurzia *Zdroj*: http://www.kiwiki.info/index.php?oldid=6646 *Prispievatelia*: Juraj

Algoritmy vyhľadávania *Zdroj*: http://www.kiwiki.info/index.php?oldid=6647 *Prispievatelia*: Jumanji, Juraj

Algoritmy triedenia *Zdroj*: http://www.kiwiki.info/index.php?oldid=6653 *Prispievatelia*: Jumanji, Juraj

Lineárny zoznam *Zdroj*: http://www.kiwiki.info/index.php?oldid=6655 *Prispievatelia*: Jehro77, Juraj

Binárny strom *Zdroj*: http://www.kiwiki.info/index.php?oldid=6656 *Prispievatelia*: Juraj

Numerické algoritmy *Zdroj*: http://www.kiwiki.info/index.php?oldid=6657 *Prispievatelia*: Juraj

Grafové algoritmy *Zdroj*: http://www.kiwiki.info/index.php?oldid=6661 *Prispievatelia*: Juraj

Štruktúry (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6668 *Prispievatelia*: Jehro77, Juraj

Rekurzia (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6669 *Prispievatelia*: Jehro77, Juraj, Mstepanovsky

Dynamická alokácia pamäti (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6670 *Prispievatelia*: Juraj

Vyhľadávanie (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6671 *Prispievatelia*: Juraj

Triedenie poľa komplexných čísel (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6672 *Prispievatelia*: Juraj

Triedenie poľa smerníkov na komplexné čísla (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6673 *Prispievatelia*: Juraj

Triedenie poľa štruktúr (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6674 *Prispievatelia*: Juraj

Triedenie poľa smerníkov (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6675 *Prispievatelia*: Juraj

Neusporiadaný lineárny zoznam (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6678 *Prispievatelia*: Juraj

Usporiadaný lineárny zoznam (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6679 *Prispievatelia*: Jehro77, Juraj

Binárny strom - Morseova abeceda (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6681 *Prispievatelia*: Juraj

Binárny strom - Huffmanovo kódovanie (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6682 *Prispievatelia*: Juraj

Numerické algoritmy (riešené príklady) *Zdroj*: http://www.kiwiki.info/index.php?oldid=6683 *Prispievatelia*: Juraj

Zdroje obrázkov, licencie a prispievatelia

Image:recursionMirror.jpg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:RecursionMirror.jpg *Licencia*: neznámi *Prispievatelia*: Juraj
Image:recursionLogMeIn.jpg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:RecursionLogMeIn.jpg *Licencia*: neznámi *Prispievatelia*: Juraj
Image:recursionCacao.jpg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:RecursionCacao.jpg *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:hanoi.png *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Hanoi.png *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:binStrom.png *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:BinStrom.png *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:ll1.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Ll1.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:ll2.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Ll2.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:ll3.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Ll3.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:binStrom.gif *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:BinStrom.gif *Licencia*: neznámi *Prispievatelia*: Juraj
Obrázok:Undirected graph.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Undirected_graph.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Obrázok:Directed graph.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Directed_graph.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Obrázok:Directed graph w.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Directed_graph_w.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Obrázok:Directed graph path.svg *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Directed_graph_path.svg *Licencia*: neznámi *Prispievatelia*: Juraj
Súbor:morse_bin_strom.gif *Zdroj*: http://www.kiwiki.info/index.php?title=Súbor:Morse_bin_strom.gif *Licencia*: neznámi *Prispievatelia*: Juraj

Licencia

Creative Commons Attribution Share Alike
GNU/GPL/KI
http://creativecommons.org/licenses/by-sa/3.0/