Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

ŠTÁTNY PEDAGOGICKÝ ÚSTAV
NATIONAL INSTITUTE FOR EDUCATION

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Algoritmy a údajové štruktúry 2

Predmet: Algoritmy a údajové štruktúry

Línia: Vlastný odborový kontext informatiky a informatickej výchovy

Operačný program VZDELÁVANIE

EURÓPSKA ÚNIA
Európsky sociálny fond

Európska únia
Európsky sociálny fond

Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov ES

Algoritmy a štruktúry údajov 2

Identifikácia modulu

Aktivita projektu:	1.2 Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ
Línia aktivity:	Vlastný odborový kontext informatiky a informatickej výchovy
Predmet:	Algoritmy a údajové štruktúry

Garant predmetu:

RNDr. Michal Winczer, PhD.
KZVI FMFI UK, Bratislava
winczer@fmph.uniba.sk

Autori:

RNDr. Michal Winczer, PhD.
KZVI FMFI UK, Bratislava
winczer@fmph.uniba.sk

RNDr. František Galčík, PhD.
ÚINF PF UPJŠ, Košice
frantisek.galcik@upjs.sk

Zaradenie modulu

Moduly Algoritmy a údajové štruktúry voľne nadväzujú na moduly Programovanie 1 až Programovanie 9. Sústreďujú sa na jednoduché údajové štruktúry, zavádzajú koncept zložitosti algoritmu a tvorbu algoritmov a jednoduchú analýzu ich zložitosti.

Modul je rozdelený na štyri témy.

Abstrakt modulu

V module zoznámime čitateľov s jednoduchými údajovými štruktúrami: záznam a strom. Ukážeme základné myšlienky tvorby jednoduchých algoritmov vyhľadávania a triedenia s prihliadnutím na vedomosti, ktoré účastníci vzdelávania získali na základe absolvovania modulov Algoritmy a údajové štruktúry 1 a Programovanie 1 až 9.

Obsah

Cieľ modulu

Po absolvovaní tohto modulu sa od účastníka vzdelávania očakáva, že

- bude vedieť použiť záznam a pole záznamov pri riešení jednoduchých úloh,
- bude vedieť algoritmus jednoduchého vyhľadávania prvkov v poli,
- bude vedieť algoritmus jednoduchého triedenia prvkov v poli,
- bude vedieť, základné informácie o stromových údajových štruktúrach.

Vstupné vedomosti

Požadované prerekvizity

Všetky moduly Programovanie 1 až 9 a Algoritmy a údajové štruktúry 1

Predpokladané vstupné vedomosti, skúsenosti a zručnosti

Predpokladáme, že účastník vzdelávania:

- vie napísať jednoduchý program a ovláda údajové štruktúry prebrané v moduloch Programovanie 1 až 9 a Algoritmy a údajové štruktúry 1,
- pozná textový súbor, vie definovať funkcie a procedúry.

Preverenie vstupných vedomostí

Účastník vzdelávania vie naprogramovať takúto aplikáciu:

Jednoduchá úloha s využitím jednorozmerného poľa. Napríklad určenie počtu výskytov najväčšieho prvku v poli obsahujúcom celé čísla.

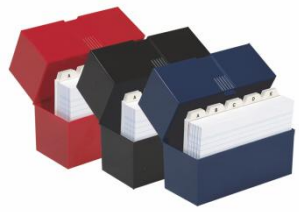
Poznámka

Všetky projekty spomínané v texte sú dostupné v systéme Moodle projektu DVUi medzi súbormi k tomuto modulu.

Záznamy

1. Definovanie vlastného typu záznam

V doterajšej programátorskej práci sme využívali na uloženie údajov premenné. Každá premenná musí byť nejakého typu. Typ premennej hovorí o tom, aké hodnoty do nej vieme uložiť. Stretli sme sa s premennými jednoduchého typu (**Integer, Boolean, Real, String,** …), ktoré vedeli uchovať jednu hodnotu daného typu. Pracovali sme aj s premennými štruktúrovaného typu jednorozmerné a dvojrozmerné pole. Tie nám umožnili „pod jedným menom“ uchovať veľa hodnôt rovnakého typu. Napríklad namiesto premenných **Teplota1, Teplota2, ..., Teplota7** reálneho typu sme vytvorili jednorozmerné pole **Teploty** so 7 políčkami indexovanými číslami od 1 po 7 (**array[1..7] of Real**). Niekedy však potrebujeme uchovať „pod jedným menom“ viacero hodnôt rôzneho typu. V takejto situácii nám už polia nepomôžu. Ďalšou vlastnosťou polí bolo to, že k hodnotám v nich uložených sme pristupovali prostredníctvom indexov. Avšak tieto indexy môžu len „málo hovoriť“ o tom, čo predstavuje hodnota uložená na príslušnom indexe. Spomeňme si na predchádzajúci modul, v ktorom sme dvojrozmerné polia využili na uloženie tabuľky. Stĺpcový index nám nič neprezradil o význame na ňom uloženej hodnoty. Riešením spomenutých problémov je ďalší štruktúrovaný typ: **záznam**. Ako uvidíme neskôr, s použitím záznamov dokážeme viacero logicky súvisiacich premenných spojiť a zastrešiť pod spoločné meno štruktúrovanej premennej. Hlavným prínosom využitia záznamov je väčší poriadok v našich programoch a ich lepšia čitateľnosť. V tejto a nasledujúcej podkapitole si na príklade jednoduchej „databázovej“ aplikácie – kartotéky (registra) žiakov základnej školy vysvetlíme, ako vytvoriť a používať záznamy v našich programoch.



Kartotéku (napr. kedysi kníh a výpožičiek v knižnici) si zvyčajne predstavujeme ako nejakú usporiadanú zbierku záznamov, v ktorej je každý záznam na samostatnej kartičke. Kartička však nie je len kúsok papiera. Kartička v kartotéke má zvyčajne predtlač vo forme formulára, ktorý sa skladá z viacerých položiek. Ich vyplnením uchovávame informácie. Každá z položiek má meno (napr. Meno, Priezvisko, Trvalý pobyt, Adresa, Ročník, ...) a priestor pre vyplnenie hodnoty. Môžeme povedať, že údaje v záznamoch sú uchované štruktúrovane, keďže ich štruktúra je určená predtlačou formulára. Záznamy v jazyku Pascal sú veľmi podobné záznamom v kartotéke. Prv než v jazyku Pascal vytvoríme záznam, musíme preň definovať „predtlač formulára“ záznamu – **typ záznamu**. Predpokladajme, že o každom žiakovi školy chceme uchovávať 3 informácie: meno, ročník a v rámci ročníka jeho triedu. Predtlač záznamu by teda mala obsahovať tieto 3 spomenuté položky. Predtlač pre záznamy v jazyku Pascal vytvárame pomocou kľúčového slova **type**, ktorým definujeme vlastný záznamový typ. Pripomeňme, že s vytváraním vlastných typov pomocou **type** sme sa už stretli pri práci s poľami ako parametrami a návratovými hodnotami funkcií a procedúr (modul Programovanie 9).

Vlastné (nami definované) typy musia byť pomenované. Aby sme odlíšili mená typov od mien premenných, procedúr a funkcií, zaužívalo sa písať písmeno **T** ako prvé písmeno názvu typu. Jazyk Pascal takéto pomenovanie nevyžaduje. Avšak aby naše programy boli prehľadnejšie, budeme túto konvenciu dodržiavať.

```
type
  TZiak = record
    Meno: String;
    Rocnik: Integer;
    Trieda: Char;
  end;
```

To, že ide o definíciu záznamového typu, prezrádza kľúčové slovo **record**, ktoré rovnako ako kľúčové slovo **begin** musí byť ukončené prislúchajúcim **end**-om. Priestor medzi **record**-om a **end**-om slúži na vymenovanie položiek, z ktorých sa záznamy tohto záznamového typu budú skladať. Pre každú položku uvádzame jej meno a typ podobne, ako sme to robili pri deklarovaní premenných.

Deklarovanie vlastného záznamového typu nevytvára žiaden záznam – definuje len predtlač (šablónu) pre záznamy. Na vytvorenie záznamu potrebujeme deklarovať (lokálnu alebo globálnu) premennú daného záznamového typu.

```
var ZiakSkoly: TZiak;
```

Na prístup k položkám záznamu využívame tzv. bodkovú notáciu. Túto notáciu sme už mnohokrát použili na prístup k vlastnostiam komponentov. Nasledovné príkazy priradenia ilustrujú naplnenie obsahu záznamu v premennej **ZiakSkoly** tak, aby uchovávala informácie o žiakovi 2.A triedy Jankovi Hraškovi.

```
ZiakSkoly.Meno := 'Janko Hraško';
ZiakSkoly.Rocnik := 2;
ZiakSkoly.Trieda := 'A';
```

Takto vyzerá obsah premennej **ZiakSkoly** záznamového typu **TZiak** po vykonaní priradení:

ZiakSkoly: TZiak	
Meno	Janko Hraško
Rocnik	2
Trieda	A

Podobne, ako sme mohli po uvedení indexu používať prvky poľa, môžeme používať položky záznamu. Môžeme do jednotlivých položiek hodnoty ukladať príkazom priradenia alebo ich čítať a používať vo výrazoch, či volaniach funkcií. Pamätajme však, že obsah premenných po vytvorení a pred prvým priradením hodnoty je nedefinovaný. To platí aj pre položky premennej záznamového typu. Obsah každej položky je nedefinovaný do okamihu, kedy jej priradíme hodnotu.

V programoch môžeme používať aj viacero premenných záznamového typu **TZiak**. Pozrime si nasledujúci fragment programu a skúsme zistiť, čo bude jeho výsledkom.

```
var
  Ziak1, Ziak2: TZiak;
begin
  Ziak1.Meno := 'Janko Hraško';
  Ziak1.Rocnik := 2;
  Ziak1.Trieda := 'A';
  Ziak2.Meno := Ziak1.Meno;
  Ziak2.Rocnik := Ziak1.Rocnik;
  Ziak2.Trieda := Ziak1.Trieda;
end;
```

Pridajte do procedúry príkazy, ktoré vypíšu skutočný obsah premenných **Ziak1** a **Ziak2** (obsah všetkých ich položiek) do textovej plochy (**TMemo**) po skončení vykonávania príkazov priradenia.

Najprv pomocou kľúčového slova **var** vytvoríme dve záznamové premenné **Ziak1** a **Ziak2** typu **TZiak**. Každá z týchto premenných sa skladá z 3 položiek (**Meno**, **Rocnik** a **Trieda**). Prvé tri príkazy naplnia obsah príslušných položiek záznamovej premennej **Ziak1.** Zvyšné tri príkazy postupne po položkách skopírujú obsah záznamovej premennej **Ziak1** do premennej **Ziak2.** Ak potrebujeme skopírovať obsah jednej záznamovej premennej do inej záznamovej premennej rovnakého typu, namiesto postupného kopírovania po položkách vieme použiť skrátenú formu:

```
Ziak2 := Ziak1;
```

Na záver úvodného zoznámenia sa so záznamovými premennými si skúsme naprogramovať funkciu **ZiakToStr**, ktorá vráti obsah záznamovej premennej typu **TZiak** vo forme reťazca podobne, ako funkcia **IntToStr** vráti reťazec s obsahom zodpovedajúcim hodnote číselného parametra.

```
function ZiakToStr(Ziak: TZiak): String;
begin
  Result := Ziak.Meno + ', ' + IntToStr(Ziak.Rocnik) +
            '.' + Ziak.Trieda;
end;
```

Záznamový typ môže byť aj typom návratovej hodnoty vo funkciách.

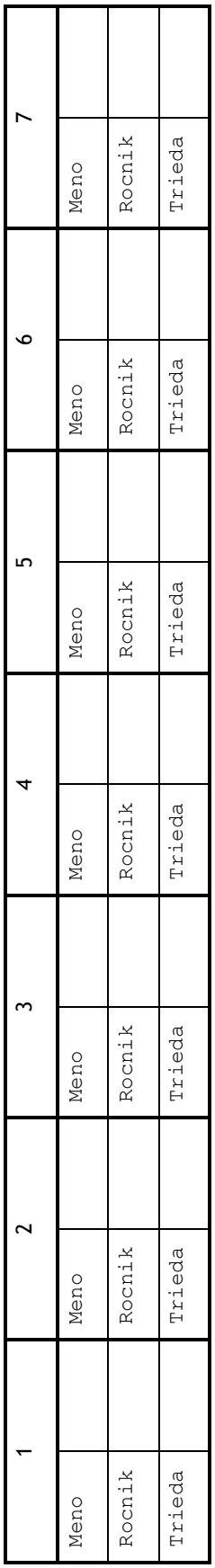
Finta: Záznamový typ ako návratový typ funkcie vieme použiť na to, aby funkcia dokázala vrátiť viac než len jednu hodnotu. Namiesto jednej hodnoty, necháme funkciu vrátiť záznam, do ktorého položiek uložíme hodnoty, ktoré chceme vrátiť.

S využitím tejto funkcie vieme jednoducho robiť výpisy záznamov. Všimnime si najmä to, že štruktúrovaný typ záznam môžeme použiť aj ako parameter procedúr a funkcií.

```
var
  Ziak1: TZiak;
begin
  // ...
  Memo1.Lines.Add(ZiakToStr(Ziak1));
end;
```

2. Pole záznamov

```
var Ziaci:
  array[1..7] of TZiak;
```



Už sme sa naučili vytvoriť záznam o jednom žiakovi. Na škole je však žiakov veľa. Na uloženie veľkého a dokonca dopredu neznámeho počtu záznamov o žiakoch použijeme pole záznamov a celočíselnú premennú, ktorá bude uchovávať skutočný počet žiakov v kartotéke (s týmto spôsobom uloženia variabilného počtu údajov v poli sme sa už stretli). Vlastný záznamový typ **TZiak**, ktorý sme definovali pomocou **type** je rovnocenný s inými typmi jazyka Pascal. Vieme ho používať úplne rovnako ako iné typy jazyka Pascal. Podobne, ako sme mohli vytvoriť pole čísel, vieme vytvoriť aj pole záznamov o žiakoch:

```
var
  Ziaci: array[1..1000] of TZiak;
  PocetZiakov: Integer = 0;
```

Ukážme si, ako by mohla vyzerať procedúra **PridajZiaka**, ktorá pridá nový záznam o žiakovi do poľa záznamov (kartotéky).

```
procedure PridajZiaka(Meno: String; Rocnik: Integer;
  Trieda: Char);
begin
  Inc(PocetZiakov);
  Ziaci[PocetZiakov].Meno := Meno;
  Ziaci[PocetZiakov].Rocnik := Rocnik;
  Ziaci[PocetZiakov].Trieda := Trieda;
end;
```

Skúsme sa ešte zamyslieť, ako by sme takú kartotéku naprogramovali, keby sme nemali záznamy a polia záznamov. Asi by sme použili podobný spôsob ako pri uložení súradníc naklikaných bodov v moduloch Programovanie.

```
var
  MenaZiakov: array[1..1000] of String;
  RocnikyZiakov: array[1..1000] of Integer;
  TriedyZiakov: array[1..1000] of Char;
  PocetZiakov: Integer = 0;
```

Namiesto jedného poľa záznamov by sme potrebovali toľko polí, koľko údajových položiek si potrebujeme o každom žiakovi uchovávať. Ak by sme si takto potrebovali o každom žiakovi pamätať trebárs 10 položiek, náš program by sa stal veľmi neprehľadným.

Aktivita 1.

Pozrite si pripravený projekt **Skola.** Nájdete v ňom už naprogramované procedúry na uloženie poľa záznamov do súboru, ale i načítanie poľa záznamov zo súboru. Prečítaním zdrojového kódu skúste prísť na to, ako fungujú.

Údaje uchovávame do databáz a kartoték nielen preto, aby boli uchované, ale aj preto, aby sme dokázali získať zaujímavé informácie z uložených údajov. Pozrime sa, ako by sme naprogramovali funkcie na zistenie počtu žiakov v zadanom ročníku, resp. v zadanej triede nejakého ročníka.

```
function PocetVRocniku(Rocnik: Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to PocetZiakov do
    if Ziaci[I].Rocnik = Rocnik then
      Inc(Result);
end;
```

```
function PocetVTriede(Rocnik: Integer; Trieda: Char): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to PocetZiakov do
    if (Ziaci[I].Rocnik = Rocnik) and
         (Ziaci[I].Trieda = Trieda) then
      Inc(Result);
end;
```

Na záver ešte naprogramujme procedúru **NovyRok**, ktorá aktualizuje kartotéku žiakov na začiatku nového školského roka. Počas tejto aktualizácie chceme z kartotéky odstrániť deviatakov a všetkým ostatným žiakom chceme zvýšiť ich ročník. Ak by nebolo odstraňovania záznamov deviatakov, tak túto úlohu hravo zvládneme jedným for-cyklom. Zamyslime sa preto, čo sa stane, ak sa niektorý zo záznamov v poli rozhodneme odstrániť. Odstránenie záznamu by malo spôsobiť, že všetky prvky poľa záznamov, ktoré sú napravo od neho, sa posunú smerom doľava o jeden index. Dokonca môžeme tvrdiť, že každý prvok sa má posunúť doľava o práve toľko indexov, koľko prvkov naľavo od neho bolo odstránených. Na prvý pohľad asi jednoduchšia možnosť je spraviť si pomocné lokálne pole záznamov, ktoré by sme postupne plnili tými záznamami, ktoré neboli odstránené. Po skončení by sme potom len obsah pomocného poľa prekopírovali do kartotéky - globálneho poľa záznamov.

```
procedure NovyRok;
var
  I: Integer;
  ZiaciPoAktualizacii: array[1..1000] of TZiak;
  NovyPocet: Integer;
begin
  NovyPocet := 0;
  for I := 1 to PocetZiakov do
    if Ziaci[I].Rocnik < 9 then
    begin
      Ziaci[I].Rocnik := Ziaci[I].Rocnik + 1;
      Inc(NovyPocet);
      ZiaciPoAktualizacii[NovyPocet] := Ziaci[I];
    end;

  for I := 1 to NovyPocet do
    Ziaci[I] := ZiaciPoAktualizacii[I];

  PocetZiakov := NovyPocet;
end;
```

V skutočnosti pomocné pole **ZiaciPoAktualizacii** nepotrebujeme. Keďže prvky z poľa záznamov len odstraňujeme, v poli **ZiaciPoAktualizacii** máme stále menší alebo rovnaký počet záznamov, než sme doposiaľ prezreli. Matematicky povedané stále platí, že **NovyPocet <= I**. Vďaka tomu môžeme každý záznam skopírovať na správnu pozíciu už priamo v spracovávanom poli záznamov bez toho, aby sme ovplyvnili záznamy, ktoré ešte len budú spracované.

Podmienku, ktorá je splnená pri každom prechode istého miesta v programe, nazývame **invariantom** programu. Invarianty programov sú využívané pri formálnom dokazovaní správnosti programov. Podobne ako matematici v matematike formálne dokazujú tvrdenia, v informatike sú vymyslené postupy, ako formálne dokázať, že program počíta to, čo má.

```
procedure NovyRok;
var
  I, NovyPocet: Integer;
begin
  NovyPocet := 0;

  for I := 1 to PocetZiakov do
    if Ziaci[I].Rocnik < 9 then
```

```
    begin
      Ziaci[I].Rocnik := Ziaci[I].Rocnik + 1;
      Inc(NovyPocet);
      Ziaci[NovyPocet] := Ziaci[I];
    end;
  PocetZiakov := NovyPocet;
end;
```

3. Využitie záznamov pri implementácii zoznamov

V prechádzajúcom module sme sa stretli s údajovou štruktúrou zoznam. Videli sme viacero spôsobov jej implementácie. Teraz si ukážeme ďalší, v ktorom bude zoznam uložený vo forme spájaného zoznamu v poli záznamov. Tento spôsob implementácie je modifikáciou implementácie zoznamu pomocou dvoch polí **Prvok** a **Nasledovnik** z predchádzajúceho modulu. Základnou myšlienkou tejto implementácie bolo to, že pre každý prvok sme si v poli **Prvok** pamätali hodnotu prvku a v poli **Nasledovnik** spoločný index do oboch týchto polí, na ktorom boli údaje nasledujúceho prvku v zozname. Namiesto dvoch polí využijeme jedno pole záznamov, ktorého prvky budú záznamového typu **TPrvok**. Pozrime sa, ako by mohol vyzerať záznamový typ **TPrvok** na uloženie údajov o prvkoch zoznamu s hodnotami typu **String**.

Výhoda použitia typu **THodnota** namiesto uvedenia konkrétneho typu hodnoty spočíva v tom, že ak sa v budúcnosti rozhodneme zmeniť typ hodnôt uložených v prvkoch zoznamu, postačí to spraviť na jednom mieste v programe - v definícii typu **THodnota**.

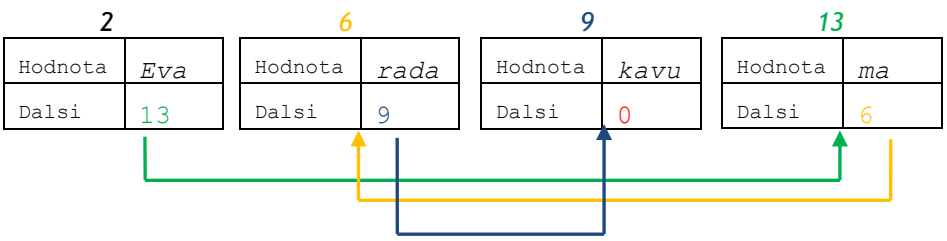
```
type
  THodnota = String;

  TPrvok = record
    Hodnota: THodnota;
    Dalsi: Integer;
  end;
```

Každý záznam typu **TPrvok** má dve položky: **Hodnota**, ktorá slúži na uloženie hodnoty prvku a **Dalsi**, ktorá uchováva index záznamu v poli záznamov s údajmi nasledujúceho prvku v zoznamu. Keďže posledný prvok zoznamu nemá nasledovníka, dohodnime sa, že v položke **Dalsi** bude mať uloženú hodnotu 0. Ak indexy v poli záznamov budú začínať číslom 1, tak 0 je hodnotou, ktorá nemôže byť platným indexom v poli záznamov. Vďaka tomu vieme na základe hodnoty v položke **Dalsi** jednoznačne určiť, či na danom prvku zoznam končí alebo ešte pokračuje. Na uloženie celej údajovej štruktúry zoznamu použijeme nasledovné premenné:

```
const
  MaxPocet = 1000;
var
  Prvky: array[1..MaxPocet] of TPrvok;
  Zaciatok: Integer = 0;
  Koniec: Integer = 0;
  ZaciatokVolnych: Integer;
```

Základné myšlienky a princípy tohto spôsobu implementácie možno nájsť v predchádzajúcom module.

Pole **Prvky** slúži na uloženie záznamov prvkov zoznamu. Aby mohol zoznam začínať a končiť na ľubovoľnej pozícii, využijeme premenné **Zaciatok** a **Koniec** na uloženie indexu záznamu prvého, resp. posledného prvku v zozname. Premenná **ZaciatokVolnych** bude uchovávať index záznamu prvého prvku v spájanom zozname voľných (nepoužívaných) záznamov. Pozrime sa, ako by mohol byť uložený 4-prvkový zoznam v poli záznamov **Prvky**. V tomto príklade je premenná **Zaciatok** rovná 2 a premenná **Koniec** má hodnotu 9. Nad záznamami sú uvedené ich indexy v poli záznamov **Prvky**.



Ukážme si, ako by mohli vyzerať podprogramy zabezpečujúce základný „manažment“ záznamov pre prvky zoznamu.

```
procedure InicializujPrvky;
var
  I: Integer;
begin
  ZaciatokVolnych := 1;
  for I := 1 to MaxPocet-1 do
    Prvky[I].Dalsi := I+1;
  Prvky[MaxPocet].Dalsi := 0;
end;

function VytvorPrvok(Hodnota: THodnota): Integer;
begin
  Result := ZaciatokVolnych;
  ZaciatokVolnych := Prvky[ZaciatokVolnych].Dalsi;
  Prvky[Result].Hodnota := Hodnota;
  Prvky[Result].Dalsi := 0;
end;

procedure UvolniPrvok(IndexZaznamu: Integer);
begin
 Prvky[IndexZaznamu].Dalsi := ZaciatokVolnych;
 ZaciatokVolnych := IndexZaznamu;
end;
```

Pozor, neriešime prípad, keď vznikne požiadavka na vytvorenie prvku zoznamu v situácii, kedy sú už všetky záznamy v poli záznamov obsadené nejakým prvkom zoznamu. Skúste navrhnúť takú úpravu funkcie **VytvorPrvok**, aby sa správala korektne aj pri takejto situácii.

Procedúra **InicializujPrvky** vytvorí počiatočný spájaný zoznam voľných záznamov. V tomto zozname je nasledovníkom záznamu na indexe **I** záznam na indexe **I+1**. Výnimkou je posledný záznam v poli, ktorý nemá nasledovníka, a preto ako index jeho nasledovníka uvádzame „dohodnutý“ (neplatný) index 0. Funkcia **VytvorPrvok** vyberie jeden prvok zo zoznamu voľných prvkov, inicializuje hodnoty jeho položiek a nakoniec vráti jeho index v poli záznamov. Procedúra **UvolniPrvok** zaradí záznam prvku zoznamu, ktorý už viac nechceme používať, medzi voľné záznamy. Na záver sa pozrime, ako by mohla vyzerať procedúra na pridanie nového prvku na začiatok zoznamu a procedúra, ktorá overí, či zadaná hodnota je uložená v zozname.

```
procedure PridajNaZaciatok(Hodnota: THodnota);
var
  NovyZaciatok: Integer;
begin
  if Zaciatok = 0 then
  begin
    Zaciatok := VytvorPrvok(Hodnota);
    Koniec := Zaciatok;
    Exit;
  end;

  NovyZaciatok := VytvorPrvok(Hodnota);
  Prvky[NovyZaciatok].Dalsi := Zaciatok;
  Zaciatok := NovyZaciatok;
end;

function JeVZozname(Hodnota: THodnota): Boolean;
var
  Aktualny: Integer;
begin
  Result := false;

  Aktualny := Zaciatok;
  while Aktualny <> 0 do
  begin
```

Špeciálne riešime prípad, kedy je zoznam prázdny, t.j. neobsahuje žiaden prvok. Túto situáciu rozpoznáme tak, že každá z premenných **Zaciatok** a **Koniec** má hodnotu 0. Pri vzniku tejto situácie len vytvoríme nový prvok (poznačíme si obsadenie záznamu v poli záznamov) a obe premenné **Zaciatok** aj **Koniec** nastavíme na index jeho záznamu.

```
    if Prvky[Aktualny].Hodnota = Hodnota then
    begin
      Result := true;
      break;
    end;

    Aktualny := Prvky[Aktualny].Dalsi;
  end;
end;
```

Čo sme sa naučili

Štruktúrovaný typ záznam umožňuje uložiť viacero hodnôt (aj rôzneho typu) do jednej premennej. K jednotlivým hodnotám (položkám) záznamu pristupujeme prostredníctvom bodkovej notácie a mena položky. Hlavnou výhodou záznamov je väčšia prehľadnosť a čitateľnosť programov. Ukázali sme si, ako s využitím poľa záznamov naprogramovať jednoduchú „databázovú“ aplikáciu, či implementovať údajovú štruktúru spájaný zoznam.

Úlohy na precvičenie

Zadanie 1.	Navrhnite záznamový typ **TBodka** na uloženie informácie o farebnej bodke. Každá bodka má svoju pozíciu, farbu a polomer.
Zadanie 2.	Navrhnite záznamový typ **TDatum** na uloženie dátumu. Naprogramujte funkciu `function JePlatny(Datum: TDatum): Boolean;` ktorá overí, či obsah položiek dátumu je platný, t.j., či v zázname uložený dátum existuje v kalendári (napr. 24.13.2010 nie je platný dátum).
Zadanie 3.	Navrhnite záznamový typ **TVizitka** na uloženie údajov z vizitky.
Zadanie 4.	Do projektu **Zoznam** doprogramujte funkciu, ktorá vráti hodnotu uloženú na zadanej pozícii v zozname: `function Ity(Poradie: Integer): THodnota;`
Zadanie 5.	Do projektu **Zoznam** doprogramujte funkciu, ktorá vráti počet hodnôt uložených v zozname (dĺžku zoznamu): `function Dlzka: Integer;`
Zadanie 6.	Do projektu **Zoznam** doprogramujte procedúru, ktorá pridá do zoznamu hodnotu na zadanú pozíciu: `procedure PridajNaPoziciu(Pozicia: Integer; Hodnota: THodnota);`
Zadanie 7.	Do projektu **Zoznam** doprogramujte procedúru, ktorá odstráni (odoberie) zo zoznamu hodnotu na zadanej pozícii: `procedure OdoberNaPozicii(Pozicia: Integer);`

Vyhľadávanie a triedenie

S vyhľadávaním má skúsenosti asi každý z nás. Hoci v súčasnosti je synonymom vyhľadávania skôr Google než predstava vyhľadávania v slovníku, alebo telefónnom zozname. Ale aj Google, alebo informačný systém pri vyhľadávaní informácií musí vykonávať podobné činnosti, ako keď sme informácie vyhľadávali ručne, iba je schopný prehľadať oveľa viac a omnoho rýchlejšie. V tejto časti sa zoznámime so základnými princípmi vyhľadávania a triedenia údajov.

Ale kto by dnes už hľadal v knihách? Alebo, že by predsa?

Vyhľadávanie

Predstavte si situáciu, že máte ísť na nákup a na papier si zapisujete postupne zoznam vecí, ktoré máte kúpiť. Ak chceme skontrolovať, či sme nezabudli napríklad na olej, musíme prejsť celý zoznam položku po položke. Dá sa to urobiť aj šikovnejšie?

Olej? Nie je lepšia bravčová masť?

Aktivita 1.	Použite pripravené lístky s číslami. Vymyslite si náhodné číslo od 1 po 1000 (požiadajte suseda aby vám ho vymyslel). Nájdite medzi Vašimi lístkami lístok s daným číslom (alebo zistite, že tam taký nie je).
Aktivita 2.	Diskutujte o tom, ako ste postupovali. Išlo vám to dobre, alebo to bolo veľmi prácne a prečo?

V module Algoritmy a údajové štruktúry 1 sme sa zamýšľali nad prácou, ktorú musí počítač vynaložiť pri riešení konkrétnej úlohy. V Aktivite 1 ste museli prácu vynaložiť vy, snažili sme sa týmto mechanickým spôsobom vytvoriť lepšiu predstavu pojmu zložitosť. V úvahách o práci potrebnej na riešenie budeme pokračovať aj v tomto module.

Predstava zložitosti konkrétneho riešenia je dôležitá pre pochopenie toho, či je uvedené riešenie prakticky použiteľné pre prípady problémov, ktoré musíme riešiť.

Predpokladajme, že ste lístky mali za sebou v náhodnom poradí (neboli nijako usporiadané). Ak ste mali šťastie, lístok s hľadaným číslom ste našli medzi prvými lístkami. Ak ste nemali toľko šťastia, bol až niekde medzi poslednými. Keby ste opakovali vyhľadávanie rôznych lístkov, v priemere by ste museli prezrieť polovicu lístkov, kým by ste ho v kope našli (samozrejme keby sa v kope lístok niekde nachádzal, inak by ste museli vždy prezrieť všetky lístky).

Text v žltom rámčeku je nepovinný. Sledujeme ním dva ciele:

Ukážka, ako sa dá exaktne presvedčiť o tom, čo možno intuitívne cítime.

Asi dôležitejšie je poukázanie na to, že medzi matematikou a uvažovaním o zložitosti algoritmov sú veľmi úzke vzťahy.

Zaujíma nás, koľko lístkov musíme v priemernom prípade prezrieť, kým nájdeme lístok, ktorý hľadáme.
Nech je všetkých lístkov n, predstavme si, že sú na nich čísla od 1 po n, každé práve raz. Lístky môžu byť v ľubovoľnom poradí. Hľadáme napr. lístok s číslom h.
Takže musíme spočítať v koľkých poradiach je lístok s číslom h na prvom, v koľkých na druhom, atď. až v koľkých je na n-tom mieste, tieto hodnoty spočítať a vydeliť počtom všetkých poradí.
Pripomeňme si, že všetkých poradí n prvkov je $n!$ V koľkých z nich je lístok s číslom h na prvom mieste? Keď je na prvom mieste lístok s číslom h, za ním môže byť $n-1$ zvyšných lístkov v ľubovoľnom poradí, už vieme, že tých je $(n-1)!$. Keď je náš lístok na druhom mieste, opäť zvyšných $n-1$ lístkov môže byť v ľubovoľnom z $(n-1)!$ poradí, atď. až po n-té miesto. Teda celkový počet prezretých lístkov vo všetkých možných poradiach je:

$$1\times(n-1)!+2\times(n-1)!+3\times(n-1)!+\cdots n\times(n-1)!=(n-1)![1+2+3+\cdots+n]=$$

$$\frac{(n-1)!\,n(n+1)}{2}.$$

Keď celkový počet vydelíme počtom všetkých poradí n!, dostaneme priemernú hodnotu:

$$\frac{(n-1)!\,n(n+1)}{2n!}=\frac{n+1}{2}.$$

Keď máme napríklad 10 lístkov, pri hľadaní náhodne vybraného lístka prezrieme v priemere 5,5 lístka, kým ho nájdeme.

Aktivita 3.

Diskutujete o tom ako by ste daný problém naprogramovali.

a) v poli
b) v spájanom zozname

Predpokladajme, že v poli **Listok** (globálna premenná) máme na pozíciách 1 až 100 uložených v nejakom poradí 100 rôznych čísiel.

Nestačilo by deklarovať pole **Listok** ako **array[1..MaxPocetListkov]**...? Stačilo, ale neskôr uvidíme, že tá **+1** sa nám zíde.

```
const
  MaxPocetListkov = 100;
var
  Listok: array[1..MaxPocetListkov + 1] of Integer;
```

Napíšme funkciu, ktorá pre danú hodnotu h vráti pozíciu prvku v poli **Listok**, na ktorej sa h nachádza. Pre jednoduchosť predpokladajme, že budeme hľadať vždy len prvky, ktoré sa v poli **Listok** budú nachádzať.

```
function VzdyNajdi(h : Integer) : Integer;
var
  I : Integer;
begin
  I := 1;
  while Listok[I] <> h do Inc(I);
  // tu platí: Listok[I] = H
  Result := I;
end;
```

To bolo jednoduché. Skúsme navrhnúť riešenie, ktoré nebude predpokladať, že hľadáme prvok, ktorý je v poli **Listok**. V prípade, že hľadaný prvok sa v poli nenachádza nech funkcia **Najdi** vráti hodnotu -1.

Riešenie je zložitejšie. Rozdiely sú označené červeno. V cykle máme komplikovanú podmienku: musíme kontrolovať, či sme „nevybehli“ z poľa a či sme už nenašli hľadaný prvok.

```
function Najdi(H : Integer) : Integer;
var
  I : Integer;
begin
  Result := -1;
  I := 1;
  while (I <= MaxPocetListkov) and (Listok[i] <> H) do Inc(I);
  // tu platí: (I > MaxPocetListkov) or (Listok[I] = H)
  if I <= MaxPocetListkov then // našli sme ho
    Result := I;
end;
```

Spomínaný užitočný „trik“ sa nazýva *zarážka*.

Okrem toho, že je takéto riešenie zložité, vyžaduje aj viac práce, pretože pre každý prvok poľa **Listok** počítač musí dvakrát zisťovať platnosť podmienky (a vo funkcii **VzdyNajdi** to bolo iba raz). Uvedieme užitočný „trik“, ktorý nám umožní zbaviť sa dvojitej podmienky v cykle. Použijeme 101-vú pozíciu v poli **Listok** a pred začatím hľadania prvku **H** ho sem zapíšeme!

Oproti **Najdi** sme ušetrili polovicu porovnávaní!

```
function Najdi2(H : Integer) : Integer;
var
  I : Integer;
begin
  Result := -1;
  Listok[MaxPocetListkov + 1] := H;
  I := 1;
  while Listok[I] <> H do Inc(I);
  // tu platí: Listok[i] = h
  if I <= MaxPocetListkov then // našli sme ho
    Result := I;
end;
```

Pravdepodobne ste rýchlo prišli na to, že keď je lístkov príliš veľa, lepšie sa v nich hľadá, keď sú usporiadané podľa hodnôt, ktoré sú na nich. Napríklad, prvý bude lístok s najmenšou hodnotou, druhý lístok s druhou najmenšou hodnotou, atď. až po lístok najväčšou hodnotou.

No to je fakt. Ešte som nevidel neusporiadaný slovník, encyklopédiu ani nič, v čom by sme mali byť schopní rýchlo niečo nájsť.

Aktivita 4.

Predstavte si telefónny zoznam, slovník, encyklopédiu v knižnom vydaní. Opíšte ako by ste vyhľadávali niečie telefónne číslo, hľadané slovo, alebo heslo.

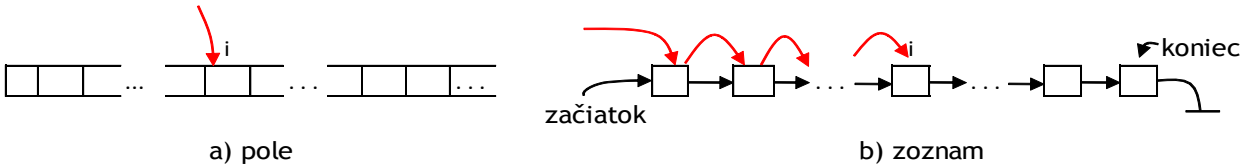
Prečo je to šikovnejšie? V čom je hlavný rozdiel oproti Aktivite 1?

Úloha 1.

Pokúste sa zistiť, čo všetko spôsobilo, že v Aktivite 4 sme vedeli rýchlo nájsť položku, prípadne zistiť, že sa v zozname nenachádza.

Návod: Snažte sa nájsť aj ďalšiu príčinu okrem usporiadania, ktorá nám umožňuje rýchle prechádzanie položkami.

Hlavným dôvodom, prečo nám pri vyhľadávaní pomohlo usporiadanie prvkov je, že vieme k jednotlivým prvkom v údajovej štruktúre pristúpiť veľmi rýchlo. Napríklad k ľubovoľnému prvku v poli vieme pristúpiť (zistiť jeho hodnotu alebo ju zmeniť) rovnako rýchlo (obr. 1.a), je jedno, či je prvok na prvej alebo poslednej pozícii, či niekde uprostred. Keby sme ale používali údajovú štruktúru zoznam, nebolo by to tak, pretože čas prístupu k prvku je úmerný jeho poradiu v zozname (obr. 1.b).



a) pole b) zoznam

Obr. 1. Červené šípky znázorňujú cez koľko prvkov musíme prejsť kým pristúpime k i-temu a) v poli a b) v zozname.

V ďalšom si ukážeme podrobnejšie, ako sa dá v utriedenom zozname, ktorý je uložený v poli, vyhľadávať oveľa rýchlejšie.

Binárne vyhľadávanie

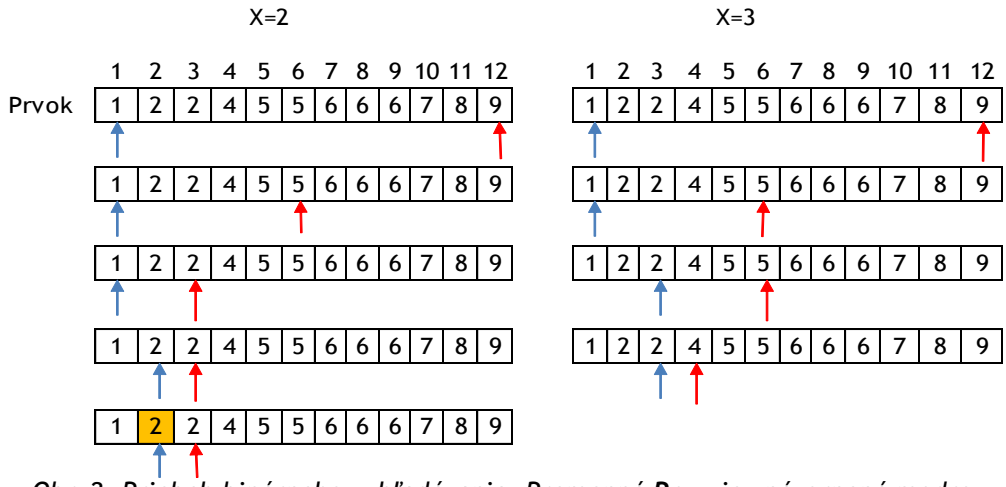
Toto je náročnejšie.

Hoci sa názov môže zdať nezrozumiteľný, v skutočnosti zachytáva podstatu rýchleho spôsobu vyhľadávania v zozname. Navyše je takmer isté, že ho poznáte a aj prakticky používate, len ste možno nevedeli, že ho informatici tak zvláštne nazývajú. Keď niečo hľadáte v utriedenom zozname, napr. v slovníku, zvyčajne odhadnete podľa toho, aké je prvé písmeno v hľadanom slove, kde treba slovník približne otvoriť, aby ste sa v ňom dostali čo najbližšie. Ak máte šťastie, trafíte na prvý pokus stranu, kde je hľadané slovo. Reálnejšie ale je, že sa vám to na prvý pokus nepodarí a pokračujete rovnakým spôsobom ďalej. Mohla nastať jedna z nasledujúcich dvoch možností, buď ste trafili (knihu ste otvorili) za slovom, ktoré hľadáte, alebo pred ním. Podstatné je, že v oboch prípadoch už hľadáte len v menšej časti knihy. Zo skúsenosti viete, že týmto spôsobom slovo nájdete veľmi rýchlo. Pozrime sa podrobnejšie, prečo je tento postup taký rýchly a prečo vôbec funguje.

V predchádzajúcom opise sme použili termín odhadneme. Aby sme vedeli postup naprogramovať, musíme spresniť, čo odhadneme znamená. Trochu si zjednodušíme situáciu: namiesto odhadovania otvoríme knihu vždy v polovici. Presnejšie v polovici tej jej časti, v ktorej ešte hľadáme. Otvoriť niekde knihu, aby sme sa pozreli, čo je v nej, predstavuje v programovaní použitie údajovej štruktúry a pristúpenie k niektorému údaju v nej. Pre jednoduchosť uvažujme vyhľadávanie v zozname

To je zjednodušenie??? Mne sa zdá, že sme postup skomplikovali.

prvkov. Na obr. 1 sme videli, že pristúpiť k prvku uprostred zoznamu realizovaného poľom je rýchle, a naopak v prípade zoznamu realizovaného ako spájaný zoznam pomalé. Preto nikoho asi neprekvapí, že si vyberieme realizáciu poľom.



*Obr. 2. Priebeh binárneho vyhľadávania. Premenná **Prvy** je znázornená modrou, **Posledny** červenou šípkou. Vľavo je prípad úspešného vyhľadávania prvku s hodnotou 2 a vpravo neúspešného vyhľadávania prvku s hodnotou 3.*

Ďalší príklad toho, že dobré označenie je viac než polovica úspechu.

(Prvy + Posledny)/2 myslíme samozrejme celočíselné delenie, teda programátorsky (Prvy + Posledny) div 2

Urobíme ešte ďalšie zjednodušenie. Predpokladajme, že hľadáme prvok s hodnotou **X**, ktorý sa medzi prvkami zaručene vyskytuje a že všetky prvky sú rôzne. Do premennej **Prvy** uložme pozíciu prvého prvku a do premennej **Posledny** pozíciu posledného prvku, medzi ktorými hľadáme prvok s hodnotou **X.** Pozícia prvku ležiaceho uprostred medzi prvkami na pozíciách **Prvy** a **Posledny** je (**Prvy** + **Posledny**)/2, budeme si ju pamätať v premennej **Stred.** Takže hľadané **X** porovnáme s **Prvok[Stred]**, keď je **X** menší, hľadáme ho už len v prvej polovici (t.j. od pozície **Prvy** po pozíciu **Stred-1**), keď je **X** väčší, hľadáme ho v druhej polovici (t.j. od pozície **Stred+1** po pozíciu **Posledny**). Nesmieme zabudnúť aj na tretiu a poslednú možnosť, že nastala rovnosť, a teda **X** sa nachádza na pozícii **Stred**. Skončí tento proces vždy? Ak si uvedomíme, že v prípade, keď sme pozíciu s hodnotou **X** ešte nenašli, sa pri ďalšom hľadaní zmenší počet prvkov, medzi ktorými ho hľadáme. A keďže sme predpokladali, že taký prvok tam je, musíme naň natrafiť po istom (konečnom) počte zmenšení. Takže uvedený proces hľadania ozaj vždy skončí.

Zamyslime sa, koľko práce musí vykonať počítač pri tomto spôsobe vyhľadávania. Iste ste si všimli, že úsek poľa, v ktorom sa snažíme „lokalizovať“ hľadanú hodnotu, sa zmenšuje. Je zrejmé, že zmenšení nebude viac, než koľko je všetkých prvkov (prečo?). Usilovali sme sa, aby sme úsek poľa, kde hľadáme **X** zmenšoval rýchlejšie než vždy len o jeden prvok. Ako sa teda zmenšuje? Porovnáme veľkosť úseku poľa pred zmenšením a po ňom. Zmenšenie budeme robiť podľa nižšie uvedeného programu. Úsek môžeme zmenšiť dvomi spôsobmi: buď posunieme pozíciu **Posledny** alebo pozíciu **Prvy**. Dĺžka úseku je rozdiel pozície posledného a prvého prvku, t.j. **Posledny-Prvy+1**. Pre jednoduchosť zanedbáme v predchádzajúcom výraze **+1**. Predpokladajme, že meníme hodnotu **Prvy**, vtedy dostaneme

Podľa programu na nasledujúcej strane má ale najkratší úsek dĺžku 2. Takže ešte jedno skrátenie a bola by dĺžka jedna. A jedno skrátenie sa nepočíta, podobne ako pri dĺžke úseku.

$$\frac{Posledny - Prvy}{\frac{Prvy + Posledny}{2} - Prvy} = \frac{Posledny - Prvy}{\frac{Prvy + Posledny - 2Prvy}{2}} = \frac{2(Posledny - Prvy)}{Posledny - Prvy} = 2.$$

Teda zmenšený úsek je dvakrát kratší než predchádzajúci. Keď meníme **Posledny**, dostaneme rovnaký výsledok (skúste to spočítať). Koľko krát sa môže zmenšiť úsek, keď sa zmenšuje vždy na polovicu predchádzajúceho a najkratší úsek má dĺžku 1? Nech má pôvodne úsek dĺžku n, vypíšme si ako sa mení: n, $n/2$, $n/2^2$, $n/2^3$,... Aká najväčšia môže byť hodnota exponentu v menovateli, aby bola hodnota zlomku väčšia alebo rovná jednej? Hodnota exponentu je náš hľadaný počet zmenšení. Je to približne $\log_2 n$. Každé zmenšenie vyžaduje len zopár operácií, teda celková práca, ktorú vykoná počítač pri binárnom vyhľadávaní prvku medzi n prvkami, je približne $\log_2 n$ operácií.

Na podobný problém sme natrafili už v Matematike 2, keď sme zisťovali koľko bitov potrebujeme na zápis čísla n.

Všimnite si, že keď chceme podrobnejšie preskúmať, koľko práce bude počítač potrebovať pri realizácii binárneho vyhľadávania, musíme si zobrať konkrétny program na riešenie tejto úlohy. Potrebnú prácu počítača môžeme určiť jeho analyzovaním. Ale musíme si uvedomiť, že potrebnú prácu sme určili iba pre náš konkrétny program. Nehovorí nič o tom, že niekto nemôže napísať iný program, ktorý by na riešenie rovnakej úlohy potreboval menej práce! Ale na druhej strane aspoň vieme, že naše riešenie zaručene nebude potrebovať viac práce počítača. To nám často stačí na to, aby sme dve riešenia toho istého problému vedeli porovnať podľa množstva práce potrebnej na ich riešenie. Oveľa ťažšia je situácia v prípade opačnej úlohy, teda keby sme chceli ukázať, aká je minimálna potrebná práca počítača na riešenie konkrétneho problému. Inak povedané, že sa nikomu nepodarí napísať program, ktorý by tento problém riešil efektívnejšie. Určovanie minimálnej potrebnej práce na riešenie rôznych úloh je jednou z hlavných aktivít teoretických informatikov.

Aha! Teda keď naprogramujem riešenie nejakej úlohy a určím koľko práce bude potrebovať počítač na jeho vykonanie, už viem, že danú úlohu viem riešiť s najviac takouto prácou, ale možno aj s menšou.

Nasledujúca ukážka funkcie na binárne vyhľadávanie obsahuje v komentároch podmienky, ktoré sú splnené, keď sa program na danom mieste vykonáva. Predpokladáme, že hľadaný prvok **X** je menší než posledný prvok (zabezpečíme to napríklad tak, že na koniec zoznamu čísiel dáme napr. fiktívnu, dostatočne veľkú hodnotu). Prvky sa môžu aj opakovať a hľadaný prvok sa medzi nimi nemusí nachádzať, vtedy funkcia vráti hodnotu 0, inak vráti pozíciu hľadaného prvku.

```
var
  Prvok : array[1..MaxPocetPrvkov] of Integer;

function NajdiBinarne(X, Prvy, Posledny : Integer) : Integer;
// predpokladáme, že
//    1 <= Prvy < Posledny <= MaxPocetPrvkov
// a že
//    Prvok[1]<=Prvok[2]<=…<=Prvok[MaxPocetPrvkov]
// Označme Q podmienku:
//   (1 <= Prvy < MaxPocetPrvkov) and (X < Prvok[Prvy])
// Označme P podmienku:
//   (1 <= Prvy < Posledny < MaxPocetPrvkov) and
//   (Prvok[Prvy] <= X < Prvok[Posledny]) or Q
var
  Stred : integer;
begin
  // platí podmienka P
  while Prvy + 1 <> Posledny do begin
    Stred := (Prvy + Posledny) div 2;
    // Prvy < Stred < Posledny
    if Prvok[Stred] < X then
      Prvy := Stred
      // platí podmienka P
   else
      Posledny := Stred
      // platí podmienka P
 end;
 // (platí Prvok[Prvy] <= X < Prvok[Prvy + 1]) or Q
  if Prvok[Prvy] = X then
    Result := Prvy
  else
    Result := 0;
end;
```

Preklad do ľudskej reči:

Prvky sú usporiadané rastúco a **Prvy** je menší ako **Posledny.**

Podmienka Q je splnená, keď je **X** menší od všetkých prvkov (teda aj od najmenšieho).

Podmienka P hovorí, že **Prvy** musí byť menší než **Posledny** a hľadaný prvok **X** je niekde medzi prvým, vrátane, a posledným.

Po skončení cyklu musíme rozlíšiť, či sme **X** medzi prvok našli, alebo nie.

Aktivita 5.

Preštudujte predchádzajúci program. Vymyslite si 10 čísiel (môžete použiť aj lístky s číslami). Ručne simulujte činnosť programu. Zapisujte si hodnoty premenných **Prvy**, **Posledny** a **Stred**. Overujte pri tom aj platnosť podmienok v komentároch.

Triedenie

V predchádzajúcej časti sme videli, že keď bol zoznam usporiadaný a mohli sme rovnako ľahko (lacno) pristupovať k ľubovoľnému prvku zoznamu, vedeli sme šikovne vyhľadávať. Usporiadanie zoznamu prvkov sa ukazuje ako kľúčové. Prvky usporadúvame vždy podľa nejakého kritéria, pričom potrebujeme, aby sme vedeli porovnať ľubovoľné dva a rozhodnúť, ktorý je väčší, ktorý menší, alebo, že sú oba rovnako veľké.

Aktivita 6.	Z vašich 100 lístkov si vyberte náhodne 10 lístkov. Usporiadajte ich podľa veľkosti.
Aktivita 7.	Opíšte ako ste postupovali v predchádzajúcej aktivite. Postup zapíšte slovami.
Aktivita 8.	Zamiešajte všetky Vaše lístky a Váš postup aplikujte na všetkých 100 lístkov.
Aktivita 9.	Diskutujte o Vašich riešeniach, vysvetlite ich ostatným. Roztrieďte ich podľa spôsobu riešenia.

Ak ste vykonali predchádzajúce štyri aktivity, iste ste prišli na rôzne spôsoby triedenia lístkov. Pravdepodobne ste si tiež overili, že je zložité vysvetliť svoj postup druhému tak, aby ho pochopil do tej miery, aby ho bol schopný použiť. Keď má človek utriediť nejaké objekty, zvyčajne pri tom prirodzene využíva, že vidí súčasne viac objektov (nemusia to byť všetky) a vie z nich vybrať najmenší alebo najväčší, prípadne ich vie usporiadať. Počítač „vidí súčasne“ vždy iba dva objekty. Postup na triedenie pre počítač musí zohľadniť túto skutočnosť.

Počítač vie súčasne porovnať iba dve čísla.

V ďalšom si ukážeme dva jednoduché spôsoby triedenia (výberom a vsúvaním) prvkov uložených v poli. Zamyslíme sa aj nad tým, koľko práce počítača vyžadujú.

Triedenie výberom

Nazýva sa aj *MaxSort*.

Jednoduchý spôsob triedenia prvkov podľa veľkosti je založený na rovnako jednoduchej myšlienke:

1. Nájdeme a odstránime najväčší spomedzi prvkov,
2. odstránený prvok zaradíme na koniec radu už utriedených prvkov,
3. opakujeme body 1 a 2 dovtedy, pokiaľ zoznam prvkov obsahuje aspoň jeden prvok.

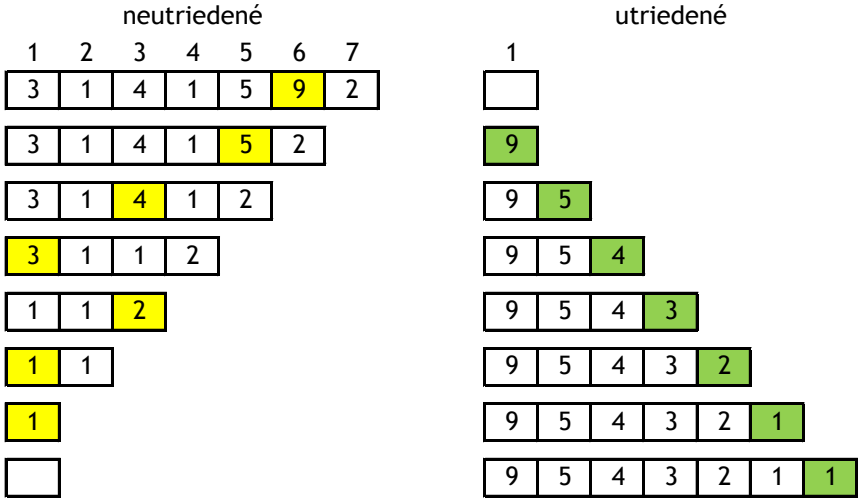
Úloha 2.	Čo by sa stalo, keby sme v predchádzajúcom opise namiesto najväčšieho prvku vyberali a odstraňovali najmenší?

Ja si myslím, že by sa nestalo nič.

Na obr. 3 ilustrujeme priebeh triedenia 7 prvkov.

Aktivita 10.	Diskutujte o postupe znázornenom na obr. 3. Vyberte si náhodne 10 lístkov s číslami a rozložte si ich pred seba na stôl do radu. Pokúste sa s nimi vykonávať rovnaký postup.

Pokúsme sa vyriešiť „detaily“ uvedeného postupu triedenia. Ako nájsť najväčší prvok spomedzi prvkov, čo znamená odstrániť a čo znamená pridať odstránený prvok na koniec radu už utriedených.

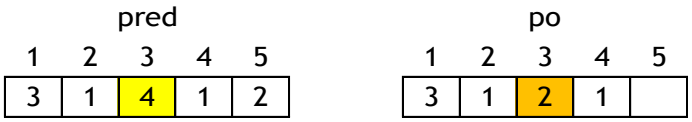


Obr. 3. Ilustrácia triedenia výberom. Žltou je vyznačený prvok, ktorý sa v danom kroku vyberie ako najväčší. Po jeho odstránení sa prvky za ním posunú o jedno miesto doľava. Zelenou je prvok, ktorý sa pridáva v danom kroku na koniec radu už utriedených čísiel.

Problém nájsť najväčší prvok, nie je nový, riešili sme ho už v predmete Programovanie. Odstrániť prvok z poľa, znamená posunúť všetky prvky za ním o jedno miesto doľava. Tým sme sa zaoberali v predmete Algoritmy a údajové štruktúry 1. Hovorili sme vtedy, že prvky sa oveľa rýchlejšie dajú odstrániť zo spájaného zoznamu, než z poľa. Prečo teda nepoužijeme namiesto poľa spájaný zoznam? Aké operácie vlastne potrebujeme? Prejsť cez všetky prvky, odstrániť vybraný prvok (môže to byť hociktorý prvok) a pridávať posledný prvok (za všetky predtým vybrané). Toto všetko sa dá použitím spájaného zoznamu vykonať s vynaložením rovnakej (prechádzanie prvkami a pridávanie prvku na koniec), alebo menšej práce (odoberanie prvku), ako s použitím poľa!

Vtip spočíva v tom, že pri odstránení vybraného prvku nemusíme posúvať všetky prvky za ním. Ako to, že nie? Neutriedené prvky môžu byť v ľubovoľnom poradí, takže nemusíme dodržať, že po odstránení prvku budú všetky prvky v takom poradí, ako boli pred jeho odstránením. Takže namiesto posúvania všetkých prvkov za odstráneným, miesto, ktoré vznikne jeho odstránením jednoducho „zaplátame“ tak, že naň dáme posledný prvok z tých, čo sú za ním (obr. 4.). Všimnite si, že ak bol odstránený prvok posledný, nemusíme robiť nič. Ak nebol posledný, existuje posledný prvok, ktorý dáme na jeho miesto.

Koniec koncov, veď ich utriedením ich poradie tak, či tak zmeníme.

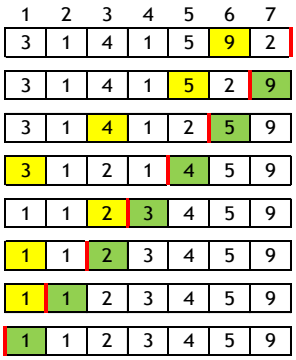


Obr. 4. Vľavo je žltou označený vybraný prvok, ktorý chceme odstrániť. Vpravo je situácia po jeho odstránení. Nahradili sme ho posledným prvkom s okrovou farbou.

Takže teraz už vieme realizovať aj v poli všetky potrebné operácie tak efektívne ako v spájanom zozname. Vybrané prvky pridávame vždy na koniec radu už predtým vybraných. Na prvý pohľad to vyzerá, že budeme potrebovať dve polia (ako na obr. 3.), jedno, kde budú neutriedené prvky a druhé kam budeme dávať utriedené. Pri pozornejšom pohľade si ale všimneme, že počet prvkov v neutriedenej časti sa zmenšuje a v utriedenej zasa rastie. Navyše, v oboch častiach je spolu stále rovnaký počet prvkov, takže teoreticky by sa mohli pomestiť do jedného poľa. Prakticky to zariadime tak, že neutriedená časť bude na začiatku poľa a za ňou bude utriedená.

Použiť pri triedení iba jedno pole je dôležité z praktického hľadiska. V praxi potrebujeme triediť často veľmi veľa prvkov, ktoré, ak sa nezmestia všetky naraz do operačnej pamäte počítača, musíme počas spracovávania čítať a zapisovať na disk, čo je viac než 10^5 krát pomalšie než spracovanie v pamäti.

Na obr. 5. Je znázornený priebeh triedenia, v ktorom sme spojili všetky predchádzajúce myšlienky.



Obr. 5. Ukážka triedenia výberom v jednom poli. Po zvislú červenú čiaru sú neutriedené prvky, za ňou sú už utriedené. Žltou je označený vybraný prvok (maximálny z ešte neutriedených). Zelenou je označený prvok, ktorý sme pridali ako posledný k už utriedeným prvkom. Všimnite si, že žlto označený prvok vždy vymeníme s prvkom tesne pred červenou čiarou (posledný neutriedený).

To si ale trúfame trochu priveľa, nie?

Koľko práce bude potrebovať počítač na realizáciu tohto spôsobu triedenia? V časti o binárnom vyhľadávaní sme hovorili, že to vieme iba keď si napíšeme konkrétny program. Už sme ale dosť skúsení na to, aby sme to skúsili aj bez konkrétneho programu, s tým, že si musíme vedieť predstaviť naprogramovanie príslušných detailov. Vidíme, že algoritmus pracuje v krokoch. V jednom kroku vyberieme maximálny prvok z doteraz neutriedených a vymeníme ho s posledným z ešte neutriedených. Na určenie maximálneho prvku potrebujeme prejsť všetky ešte neutriedené prvky. Počet potrebných operácií v jednom kroku teda bude priamo (lineárne) úmerný počtu neutriedených prvkov. Koľko krokov potrebujeme na utriedenie všetkých prvkov? V každom kroku spracujeme definitívne jeden prvok (počet ešte neutriedených prvkov klesne o 1 a počet utriedených sa zvýši o 1). Krokov je teda presne toľko, koľko je všetkých prvkov. Keď je prvkov n, v prvom kroku potrebujeme približne n operácií, v druhom $n-1$, atď. až v n-tom iba jednu. Keď to sčítame dostaneme

$$n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} \approx n^2.$$

Napríklad údaje o všetkých občanoch Slovenska.

Pri opísanom triedení výberom n prvkov, je počet operácií, ktoré potrebuje počítač vykonať približne n^2. Uvedomte si, že keď chceme triediť napríklad 6 miliónov prvkov, potrebných by bolo najmenej $(6 \cdot 10^6)^2 = 36 \cdot 10^{12}$ operácií. Ak zoberieme, že počítač je schopný vykonať približne 10^9 operácií za sekundu, trvalo by triedenie zhruba hodinu, čo je asi neakceptovateľné.

Triedime údaje Indov alebo Číňanov?

Úloha 3.

Zoberme opísaný algoritmus triedenia výberom a rýchlosť počítača ako v texte. Koľko prvkov by najviac vedel počítač utriediť, keby sme nechceli čakať dlhšie než jednu sekundu? Koľko by mu trvalo utriediť 10^9 údajov?

```
var
  Prvok : array[1..MaxPocetPrvkov] of Integer;

procedure TriedVyberomMax;
var
  I, J, Max : integer;
begin
  for I := MaxPocetPrvkov downto 1 do begin
    max := Prvok[1];
    for J := 2 to I do
      if Prvok[J] > Prvok[Max] then
        Max := J;
    Vymen(Max, I) // vymení vzájomne Prvok[max] a Prvok[i]
  end
end;
```

Premenná **Max** je pozícia doteraz najväčšieho prvku

Predchádzajúci program triedenia výberom maximálneho prvku obsahuje vnorený dvojitý cyklus. Premenná **i** vonkajšieho cyklu zodpovedá pozícii červenej čiarky (hranica medzi neutriedenou a utriedenou časťou) na obr. 5. Vnútorný cyklus hľadá

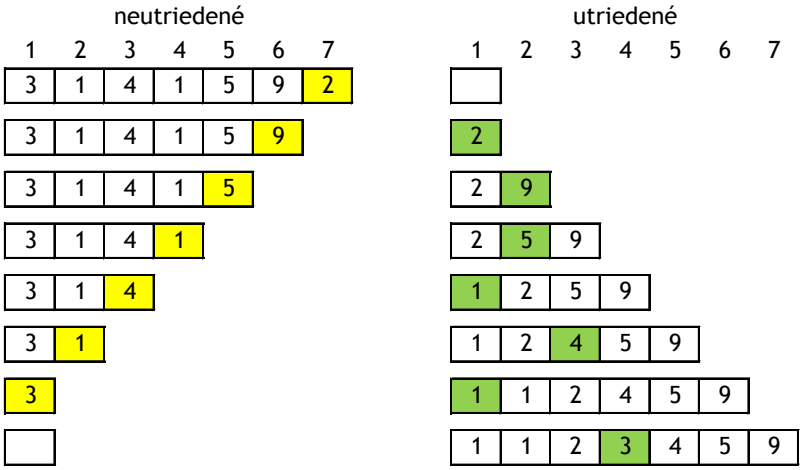
najväčší prvok v úseku po červenú čiarku (medzi ešte neutriedenými prvkami). Po skončení vnútorného cyklu je v premennej **Max** pozícia najväčšieho prvku v úseku poľa od 1. po i. pozíciu. Volaním procedúry **Vymen(Max, I)** vymeníme najväčší prvok v úseku s posledným.

Úloha 4.	Naprogramujte procedúru **Vymen**.
Aktivita 11.	Simulujte činnosť programu pomocou malého počtu lístkov (napr. desiatich). V prípade, že nájdete v programe chybu, pokúste sa ju opraviť.

Triedenie vsúvaním

Druhým jednoduchým algoritmom triedenia je triedenie vsúvaním. Pri triedení výberom sme vyberali spomedzi prvkov postupne najväčší, potom druhý najväčší, atď. až po najmenší prvok. Pri triedení vsúvaním budeme postupne vytvárať utriedený zoznam prvkov. Opäť si prvky rozdelíme na dve časti, utriedenú a neutriedenú. Na začiatku budú všetky prvky v neutriedenej časti a utriedená bude prázdna. Na rozdiel od predchádzajúceho spôsobu teraz vyberieme z neutriedenej časti ľubovoľný prvok a budeme ho vsúvať do už utriedenej časti.

Nazýva sa aj *Insertsort*.



Obr. 6. Ilustrácia triedenia vsúvaním. Žltou je prvok, ktorý sa vsúva medzi už utriedené prvky. Zelenou je označené miesto naposledy vsunutého prvku.

„Ľubovoľný“ prvok z neutriedenej časti vyberieme samozrejme tak, aby sme to vedeli čo najjednoduchšie zrealizovať. Napríklad prvý, alebo posledný prvok z tejto časti. Jadro problému bude ako ho vsunúť do utriedenej časti. Ale to je pre nás už známy problém, na ktorý sme už niekoľkokrát natrafili. Treba nájsť pozíciu prvku, za ktorý (pred ktorý) ho máme vsunúť. Aké operácie potrebujeme? Odobrať prvý (resp. posledný) prvok a vsunúť prvok do zoznamu. Videli sme, že potrebné operácie sa veľmi ľahko (a rýchlo) dajú realizovať spájaným zoznamom. Prečo sa teda snažíme použiť pole? V spájanom zozname vieme rýchlo pridať nový prvok, ale musíme vedieť, za ktorý prvok ho chceme pridať. Na vyhľadanie tohto prvku počítač potrebuje prácu (lineárne) úmernú počtu prvkov, ktoré sú v zozname pred ním. Celkovo potrebujeme množstvo práce (lineárne) úmerné počtu prvkov v utriedenom zozname, ktoré budú pred vloženým prvkom. V poli nič neušetríme, lebo na vyhľadanie miesta prvku budeme potrebovať v najhoršom čas (lineárne) úmerný počtu prvkov, ktoré sú pred ním a potom ešte budeme musieť vytvoriť pre vkladaný prvok voľnú pozíciu – teda všetky prvky pred ním (resp. za ním) posunúť o jednu pozíciu, čo tiež vyžaduje približne rovnako veľkú prácu ako pre vyhľadanie jeho miesta.

Prečo v najhoršom? Vari to ide aj rýchlejšie?

Úloha 5.

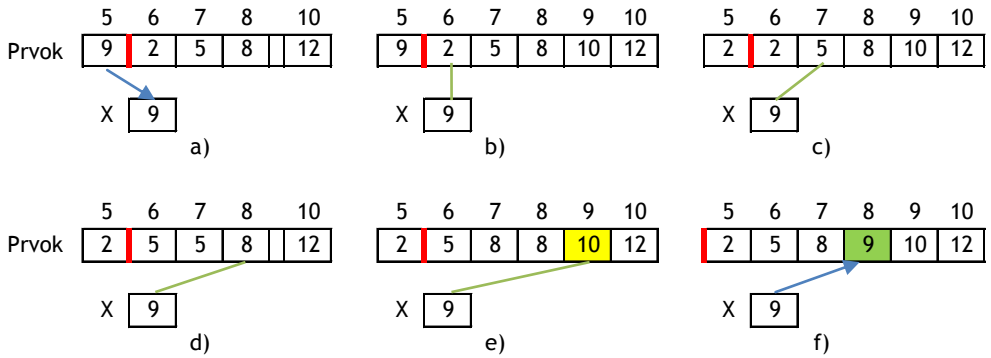
Má zmysel nahradiť hľadanie miesta kam treba vsunúť prvok binárnym vyhľadávaním? Zdôvodnite svoj názor.

Je to prirodzené, lebo posledný prvok z neutriedenej časti je najbližšie k už utriedeným.

Keď si uvedomíme, že v poli je hľadanie prvku a posúvanie realizované jednoduchým cyklom, teda, že obe činnosti majú dosť veľa spoločného, prídeme na myšlienku, že by stálo za úvahu pokúsiť sa ich šikovne spojiť a vykonávať súčasne. Príklad priebehu hľadania a posúvania je na obr. 7. Pred červenou čiarou sú neutriedené prvky a za ňou utriedené prvky. Do utriedenej časti budeme vsúvať posledný prvok z neutriedenej. Posledný prvok je na pozícii 5. Hodnotu **Prvok[5]** si najprv odložíme do premennej **X** (obr. 7a), a potom ho chceme vsunúť medzi prvky na pozíciách 6 až 10. Všimnite si, že **Prvok[6]<=Prvok[7]<=Prvok[8]<=Prvok[9]<=Prvok[10]**. Hľadáme takú pozíciu, aby platilo, že **Prvok[j]<=X<Prvok[j+1]**, teda, že prvok **X** máme vsunúť za prvok na pozícii **j**, v našom príklade sa **j** rovná 8. Pokým neurčíme hľadanú hodnotu j, postupne porovnávame, či platí, že **X >= Prvok[6]** (obr. 7b), ak áno vieme, že **j** už bude aspoň 7, teda, že **Prvok[6]** budeme musieť posunúť o jedno miesto doľava, na miesto **Prvok[5]**. Teraz oceníme, že **Prvok[5]** je voľný, pretože sme ho premiestnili do premennej **X**, takže **Prvok[6]** sem môžeme prepísať. Čím sme sa dostali do tej istej situácie, v akej sme boli pred chvíľou, ale teraz **X** vsúvame už len medzi prvky na pozíciách 7, 8, 9 a 10 (počet prvkov, kam vsúvame, sa zmenšil o jeden). Ak by platilo **X < Prvok[6]**, našli sme hľadané miesto, prvok **X** patrí na pozíciu 5 (platí **Prvok[5]<=X<Prvok[6]**). V našom príklade platí predchádzajúca možnosť, takže pozíciu **X** hľadáme ďalej (obr. 7c, 7d, 7e a 7f).

<= je tam preto, lebo prvky môžu mať rovnakú hodnotu.

Kvôli tomu, aby **Prvok[5]** ostal na mieste, sme robili takúto obchádzku???



Obr.7. Zlúčenie hľadania a posúvania. Za červenou čiarou je utriedená časť. Zelenou čiarou sú spojené prvky, ktoré sa porovnávajú. Žltou je hodnota, pre ktorú je prvý raz X menšie. Zelenou je označený vsunutý prvok.

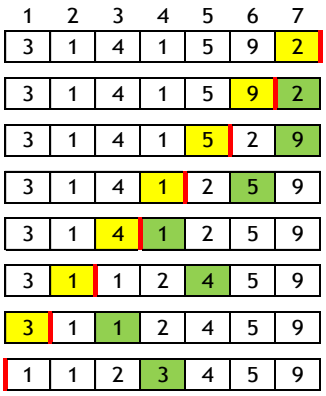
Nájdenie miesta kam máme vsunúť prvok a jeho vsunutie na toto miesto vyžaduje rovnaký počet priradení a porovnaní, koľko prvkov je v zozname pred vsunutým prvkom. Predpokladajme, že triedime n prvkov. Ak by sme mali takú smolu, že by sme každý prvok pridávali až na koniec zoznamu, prvý by sme pridali bez práce, druhý by vyžadoval 1 priradenie a porovnanie, ďalší po dve, atď. až po posledný, ktorý by vyžadoval $n-1$ priradení a porovnaní. Keď všetko spočítame, dostaneme, že na utriedenie zoznamu je v najhoršom prípade potrebných približne

$$1 + 2 + \cdots + (n-1) = \frac{n(n+1)}{2} \approx n^2$$

operácií. Použili sme termín „v najhoršom prípade“. Čo tým myslíme? V skutočnosti je málo pravdepodobné, že by sme všetky prvky pridávali vždy na koniec zoznamu utriedených. Uvedomte si, že to je iba v jedinom prípade, keď sú prvky usporiadané opačne ako chceme. Oveľa častejší je prípad, keď prvok vsunieme niekam „doprostred“ zoznamu už utriedených prvkov. Keď máme šťastie a prvky v zozname budú už na začiatku utriedené, vykonáme celkovo len toľko operácií, koľko je prvkov v zozname, teda n. To je výrazne lepšie oproti triedeniu výberom, kde sme vždy potrebovali približne n^2operácií. Tak isto bude triedenie vsúvaním oveľa lepšie v prípade, keď bude zoznam na začiatku „skoro“ utriedený.

Takže predsa je triedenie vsúvaním lepšie než triedenie výberom! Hoci v najhoršom prípade sú rovnako zlé.

Na obr. 8. je ilustrovaný priebeh triedenia vsúvaním v jednom poli.



Obr. 8. Ukážka triedenia vsúvaním v jednom poli. Po zvislú červenú čiaru sú neutriedené prvky, za ňou sú už utriedené. Žltou je označený prvok ktorý budeme vsúvať (posledný z ešte neutriedených). Zelenou je označený posledný vsunutý prvok do už utriedených prvkov.

Nasleduje procedúra **TriedVsuvanim**, ktorá realizuje vyššie opísané úvahy v programovacom jazyku.

```
var
  Prvok : array[1..MaxPocetPrvkov] of Integer;

procedure TriedVsuvanim;
var
  I, J, X : Integer;
begin
  for I := MaxPocetPrvkov - 1 downto 1 do begin
    X := Prvok[I]; // odložime si prvok, ktorý vsúvame
    J := I + 1;  // nastavíme sa na začiatok utriedenej časti
    while (J <= MaxPocetPrvkov) and (Prvok[J] <= X) do begin
      // platí predchádzajúca podmienka, ešte sme nenašli
      // miesto kam treba X vsunúť
      Prvok[J - 1] := Prvok[J]; // posunieme J-ty prvok doľava
      Inc(J); // a ideme na ďalší
    end;
    Prvok[J - 1] := X // vsunieme na pozíciu J-1
  end
end;
```

Všimnite si, že premenná **J** ukazuje vždy na pozíciu prvky, pred ktorý máme **X** vsunúť.

Úloha 6.	Využitím zarážky sa pokúste odstrániť z predchádzajúceho programu dvojitý test v cykle while.
Aktivita 12.	Skúste navrhnúť a naprogramovať triedenie vsúvaním tak, že v poli **Prvok** budú uložené najprv utriedené prvky a potom neutriedené (t.j. opačne, ako sme to prezentovali v texte)

Čo sme sa naučili

Zoznámili sme sa s vyhľadávaním v neutriedenom a utriedenom zozname prvkov. Vieme prečo sa v utriedenom zozname dá vyhľadávať rýchlejšie.

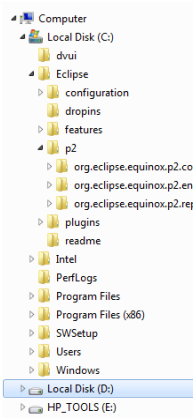
Poznáme dva jednoduché algoritmy triedenia prvkov v poli, triedenie výberom a vsúvaním.

Stromy

1. Stromové štruktúry okolo nás

Údaje o objektoch a veciach okolo nás sú v počítačoch uchovávané a organizované rôznymi spôsobmi a v rôznych formách. Každá z foriem ponúka isté možnosti, má svoje výhody aj nevýhody. Jednou z najčastejších foriem uloženia a prezentácie dát sú tabuľky. Tie sme sa v našich programoch naučili vytvárať skombinovaním záznamov a polí. Poznáme už aj spájané zoznamy, ktoré sú podobné poliam. Umožňujú oveľa efektívnejšie pridávanie a odoberanie prvkov, no žiaľ na úkor efektívnosti prístupu ku konkrétnemu prvku poľa. Polia aj spájané zoznamy majú spoločné to, že jednotlivé prvky - údaje sú uchované ako nejaká postupnosť. To však niekedy nestačí. Zoberme si taký súborový systém. Súbory na disku nemáme uložené pokope v jednom obrovskom priečinku (adresári) ako nejaký zoznam. Súbory na disku máme hierarchicky usporiadané do priečinkov. V priečinkoch môžeme mať okrem súborov opäť priečinky. A v nich môžu byť ďalšie a ďalšie priečinky. Aby sme sa v množstve súborov a priečinkov rýchlo zorientovali, využívame tzv. stromové zobrazenie.

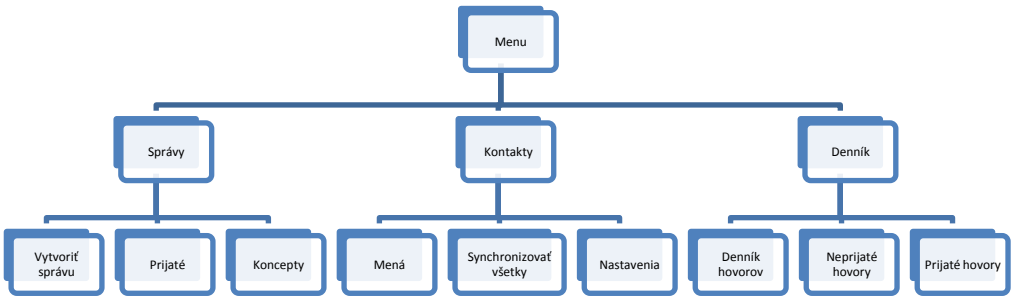
Aktivita 1.	Pozrite si v súbory a priečinky vo svojom počítači v stromovom zobrazení. V čom spočíva hierarchickosť štruktúry ?



Stromové usporiadanie nenájdeme len v organizácii súborov na disku. Spomeňme si na menu obrazovky v mobile, televízore alebo inom elektronickom zariadení. Menu je tiež organizované hierarchicky v stromovej štruktúre. Jednotlivé položky menu môžu priamo vykonať nejakú akciu alebo viesť na nejaké podmenu (obrazovku s menu) - na menu nižšej úrovne. Položky menu sú zvyčajne voliteľné aj číslom, vďaka čomu sa vieme rýchlo pohybovať v hierarchii menu. Tieto čísla predstavujú akúsi navigačnú cestu v stromovom usporiadaní.

Aktivita 2.	Skúste schematicky (a hierarchicky) zakresliť štruktúru menu vo svojom mobilnom telefóne. Stačí niekoľko položiek. Pre vybrané položky si zaznačte, aká postupnosť čísel (skratka) k nim vedie.

Veľmi malá časť menu v mobilnom telefóne by mohla schematicky vyzerať takto:

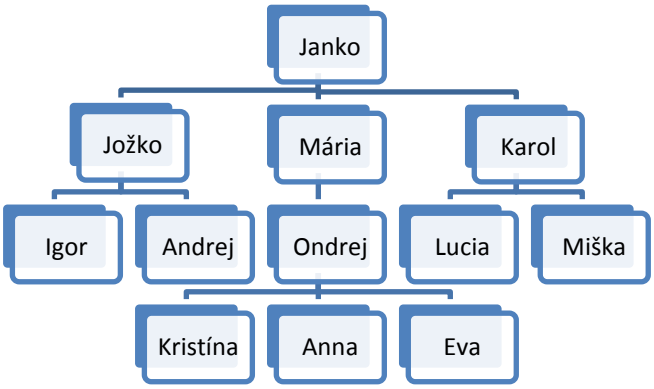


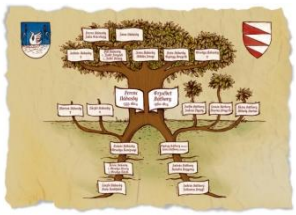
Ďalším miestom, kde môžeme nájsť tzv. „stromové“ usporiadanie, sú mapy webových sídiel. Pozrime sa na niektoré z nich a všimnime si stromové usporiadanie stránok. Jednotlivé úrovne predstavujú rozdelenie článkov na podskupiny:

- http://www.nbs.sk/sk/mapa-stranky
- http://new.sacr.sk/sacr/mapa-stranky/
- http://www.uniba.sk/index.php?id=8

2. Rodokmeň

Genealógia je oblasťou, v ktorej je „stromové“ usporiadanie údajov veľmi dôležité. Nie náhodou sú v tejto oblasti najčastejšie používané výrazy ako rodostrom alebo rodokmeň. Jednou z genealogických schém je strom potomkov osoby. Predpokladajme, že Janko mal deti Jožka, Máriu a Karola. Jožko mal deti Igora a Andreja. Mária mala len jedného syna Ondreja. Karol mal dve dcéry Luciu a Mišku. No a nakoniec Ondrej mal 3 dcéry: Kristínu, Annu a Evu. Potom Jankov strom potomkov môžeme schematicky znázorniť takto:





Aktivita 1.

Schematicky zakreslite strom potomkov niektorého zo svojich predkov (napr. pradedka). Pozrite si aj stromy potomkov niektorého z dávnych panovníkov:

- http://en.wikipedia.org/wiki/Habsburg_family_tree
- http://en.wikipedia.org/wiki/List_of_family_trees

Nás, ako informatikov, určite zaujíma, ako by sme vedeli takéto stromovo (hierarchicky) usporiadané údaje uložiť a použiť v našich programoch. Kľúčovou vlastnosťou v stromoch sú väzby (prepojenia) na potomkov - hierarchicky nižšie umiestnené údaje. S prepojeniami medzi údajmi sme sa už stretli pri spájaných zoznamoch. V nich pri každom prvku bola umiestená informácia o tom, kde nájsť priameho nasledovníka tohto prvku v zozname. Nie je preto nijako prekvapujúce, že aj stromové údaje vieme uložiť podobným spôsobom. Základný prvok v strome potomkov, ktorý by obsahoval záznam o jednej osobe, by mohol vyzerať takto:

```
const
  MaxDeti = 20;

type
  TOsobaVStrome = record
    Meno: String;
    PocetDeti: Integer;
    Deti: array[1..MaxDeti] of Integer;
  end;
```

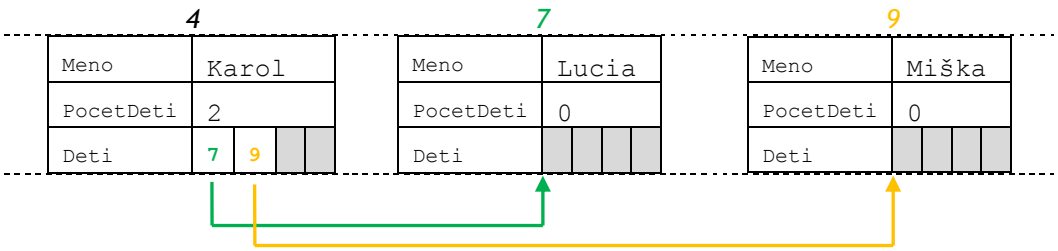
Predpokladáme, že žiadna osoba nemôže mať viac ako 20 detí. Ak by to bolo málo, zmeníme hodnotu konštanty **MaxDeti** na inú vhodnejšiu hodnotu.

Záznam každej z osôb obsahuje jej meno v položke **Meno**, informáciu o počte jej detí v položke **PocetDeti** a nejaké pole čísel **Deti**. Podľa názvu položky tušíme, že pole **Deti** bude obsahovať nejaké údaje o deťoch. Aké sú to údaje bude jasné, keď si povieme, ako uložiť všetky osoby v strome potomkov. Keďže osôb je veľa, na uloženie všetkých osôb použijeme pole záznamov.

```
type TVelkostStromu = 1..1000;
var OsobyStromu: array[TVelkostStromu] of TOsobaVStrome;
```

Záznamy sú v poli záznamov **OsobyStromu** uložené lineárne za sebou. V strome potomkov však potrebujeme uložiť záznamy hierarchicky. Na vrchole celej hierarchie stojí osoba, ktorej strom potomkov chceme zachytiť. Tento záznam

uložme v poli na indexe 1. Záznamy ďalších osôb budeme postupne pridávať do poľa záznamov. Na zachytenie hierarchie použijeme položku **Deti** v záznamoch osôb. Do tohto poľa si pre každú osobu poznačíme indexy záznamov (v poli záznamov) jej detí. Toto pole nám bude pri pohybe v stromovej hierarchii slúžiť ako navigácia k jednotlivým deťom osoby. Predpokladajme, že záznam Karola je uložený na indexe 4, záznam jeho dcéry Lucie na indexe 7 a dcéry Mišky na indexe 9. Potom v Karolovom zázname bude položka **Deti[1]** obsahovať číslo 7 (navigácia na záznam prvej dcéry) a položka **Deti[2]** číslo 9 (navigácia na záznam druhej dcéry):



Všimnime si, že pri takomto uložení prepojení (väzieb) medzi záznamami vôbec nezáleží, kde sa nachádzajú záznamy detí jednotlivých osôb. Keďže záznamy vytvárajú stromovú štruktúru, na to, aby sme sa vedeli dostať k ľubovoľnému záznamu, nám stačí vedieť, na ktorom indexe sa nachádza záznam osoby na vrchole hierarchie. My sme sa dohodli, že tento záznam bude na indexe 1. Ale pokojne si môžeme vytvoriť celočíselnú premennú, ktorá by uchovávala iný index záznamu na vrchole hierarchie (index záznamu osoby, ktorej strom potomkov je uchovaný v strome). Pripomeňme, že variant s premennou navigujúcou na začiatok údajovej štruktúry sme už použili pri spájaných zoznamoch - vtedy sme použili premennú **Zaciatok** uchovávajúcu index záznamu prvého prvku v zozname.

Aktivita 2.

Zakreslite, ako by ste uložili strom potomkov svojho predka (z predchádzajúcej aktivity) v poli záznamov typu **TOsobaVStrome**. Predpokladajte, že k dispozícii máte pole záznamov, ktoré má práve toľko prvkov, koľko osôb sa nachádza v nakreslenom strome potomkov.

Už vieme, ako zachytiť väzby medzi záznamami. Ostáva nám už len vyriešiť „manažovanie“ záznamov v poli pri pridávaní alebo odobraní osoby (záznamu) do, resp. zo stromu podobne, ako to bolo pri spájaných zoznamoch. Totiž pri pridávaní nového záznamu musíme nejako vedieť, ktorý záznam v poli záznamov je voľný a ktorý je už použitý na uchovanie nejakých údajov. Ak sa obmedzíme len na pridávanie nových osôb do stromu potomkov, vystačíme si s podobnou fintou ako pri poliach. V celočíselnej premennej **PocetOsob** si budeme pamätať, koľko prvých prvkov v poli záznamov je použitých. Pri požiadavke na pridanie nového záznamu zvýšime obsah premennej **PocetOsob** o 1 a ako nový záznam použijeme posledný platný záznam na indexe **PocetOsob**, t.j. prvok **OsobyStromu[PocetOsob].** Pozrime sa, ako by mohla vyzerať funkcia **PridajOsobu** na pridanie nového záznamu do stromu potomkov.

Všeobecnejší spôsob (s pridávaním aj odoberaním uzlov) „manažovania“ voľných záznamov si ukážeme neskôr pri binárnych vyhľadávacích stromoch.

```
function PridajOsobu(Rodic: Integer; Meno: String): Integer;
begin
  Inc(PocetOsob);
  OsobyStromu[PocetOsob].Meno := Meno;
  OsobyStromu[PocetOsob].PocetDeti := 0;

  Inc(OsobyStromu[Rodic].PocetDeti);
  OsobyStromu[Rodic].Deti[OsobyStromu[Rodic].PocetDeti]:=
                                                    PocetOsob;
  Result := PocetOsob;
end;
```

Funkcia **PridajOsobu** má dva parametre. Parameter **Rodic** obsahuje index záznamu tej osoby, ktorá má byť rodičom pridávanej osoby. Keďže strom potomkov je hierarchická štruktúra, je celkom prirodzené, že aj osoby do stromu budeme pridávať hierarchicky – najprv rodiča a až potom jeho deti. Druhým parametrom je parameter **Meno**, ktorý obsahuje meno osoby. Ak by sme chceli o osobe v strome potomkov uchovávať viac informácií, asi by sme ich pridali cez ďalšie parametre tejto funkcie, prípadne by sme použili nejaký záznamový typ, v ktorom by sme ich „doručili“ do vnútra funkcie **PridajOsobu**. Ako funkcia pracuje? Najprv pridáme do poľa záznamov nový platný záznam osoby a naplníme ho inicializačnými údajmi – položke **Meno** sa priradí meno pridávanej osoby a poznačí sa, že táto osoba nemá zatiaľ žiadne známe deti (položka **PocetDeti** sa nastaví na 0). Keďže pridávaná osoba je dieťaťom osoby, ktorej záznam je na indexe **Rodic**, musíme ešte v tomto zázname poznačiť, že táto osoba má ďalšie dieťa. Každá osoba uchováva indexy záznamov svojich detí v položke **Deti**, ktorá je typu jednorozmerné pole, a počet detí v položke **PocetDeti**. Preto záznamu na indexe **Rodic** zvýšime počet detí (položka **PocetDeti**) a následne na posledný platný index poľa **Deti** uložíme index záznamu práve pridávanej osoby. Nakoniec nastavíme ako návratovú hodnotu index záznamu pridávanej osoby. Cieľom je, aby funkcia vrátila index záznamu pridanej osoby. Môže sa to zdať zbytočné, pretože vieme, že táto hodnota je rovná hodnote v premennej **PocetOsob**. Ak by sme ale v budúcnosti zmenili mechanizmus, ako vyberáme nový „prázdny“ záznam na použitie, stačí nám zmenu spraviť vo funkcii **PridajOsobu.** Nemusíme potom hľadať, kde všade vo vytváranom programe sme predpokladali, že záznam naposledy pridanej osoby je uložený v prvku **OsobyStromu[PocetOsob]**.

Ďalším dôvodom pre hierarchické pridávanie je to, že v navrhnutej údajovej štruktúre si iba rodič pamätá indexy záznamov svojich detí. Dieťa nepozná index záznamu svojho rodiča. Dieťa nemôžeme do stromu pridať skôr ako jeho rodiča aj preto, že by sme nevedeli, kde do stromu potomkov ho máme umiestniť.

Aktivita 3.

Otvorte a prezrite si projekt **Rodokmen**, ktorý načíta rodokmeň zo súboru do poľa záznamov a zobrazí ho. Všimnite si, ako vyzerá funkcia na pridanie prvku do poľa (predpokladáme, že prvky do stromu budeme len vkladať), či to, v akom formáte sú uložené údaje v súbore.

Pozrime sa teraz, ako by sme v projekte **Rodokmen** doprogramovali obslužný kód zatlačenia tlačidla, ktorého stlačením sa vypíše tzv. panovnícka línia. Tá je tvorená prvorodenými deťmi panovníkov. Pre jednoduchosť predpokladajme, že deti osôb sú v strome potomkov uložené od najstaršieho po najmladšie. Zamyslime sa, čo to máme vlastne spraviť. Ak by sme to nechceli programovať, ale len „zrealizovať“, tak pravdepodobne spravíme nasledovné. Najprv sa presunieme do hierarchicky najvyššej osoby v strome – t.j. do osoby, ktorej strom potomkov uvažujeme. Vypíšeme meno tejto osoby. Potom sa budeme postupne posúvať zhora nadol v strome potomkov tak, že si vyberieme najľavejšie (najstaršie) dieťa. Toto budeme opakovať, kým sa dá. Takto zídeme až k osobe, ktorá je bez známych potomkov. No a samozrejme počas putovania v strome potomkov budeme vypisovať mená osôb, ktoré sme pri putovaní navštívili. Tento postup zapísaný v Pascale by mohol vyzerať napríklad takto:.

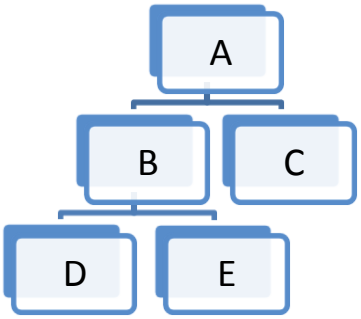
```
procedure TForm1.Button1Click(Sender: TObject);
var
  IndexOsoby: Integer;
begin
  Memo1.Lines.Clear;

  IndexOsoby := 1;
  while OsobyStromu[IndexOsoby].PocetDeti > 0 do
  begin
    Memo1.Lines.Add(OsobyStromu[IndexOsoby].Meno);
    IndexOsoby := OsobyStromu[IndexOsoby].Deti[1];
  end;

  Memo1.Lines.Add(OsobyStromu[IndexOsoby].Meno);
end;
```

Všimnime si, akým priradením je implementovaný „presun“ zo záznamu osoby na záznam jej najstaršieho dieťaťa.

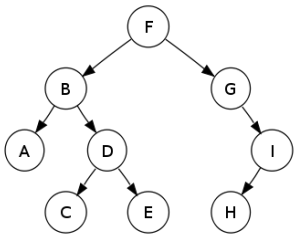
Premenná **IndexOsoby** slúži v procedúre na uchovanie toho, na záznam ktorej osoby v strome potomkov sa práve nachádzame. Na začiatku do nej priradíme index záznamu osoby, ktorá je v strome potomkov hierarchicky najvyššie. V našom prípade vzhľadom na dohodu je to číslo 1. Potom kým osoba, na záznam ktorej sa práve nachádzame, má nejaké deti, opakujeme to, že sa presunieme na najstaršie dieťa (ak existujú nejaké deti, existuje aj najstaršie dieťa, ktorého index záznamu je na indexe 1 v položke **Deti**).

3. Stromová terminológia

Aj okolo stromových štruktúr v informatike sa ustálila istá terminológia (inšpirovaná stromami potomkov a stromami v prírode).



Terminologicky si popíšeme strom znázornený na obrázku (schéme) vpravo. Jednotlivé „okienka“ (záznamy, osoby) stromu voláme **uzlami** alebo tiež vrcholmi stromu. Strom má celkom 5 uzlov označených písmenami **A**, **B**, **C**, **D** a **E**. Uzol, ktorý je v hierarchii najvyššie, sa nazýva **koreň** stromu. Na obrázku je to uzol označený písmenom **A**. Vzťahy medzi uzlami sú popisované „familiárnou“ terminológiou. Ak sa uzol **X** nachádza v hierarchii priamo pod iným uzlom **Y**, hovoríme, že uzol **X** je dieťaťom, resp. synom uzla **Y** a uzol **Y** je rodičom, resp. otcom uzla **X**. Na obrázku je uzol **A** rodičom uzlov **B** a **C**, uzly **B** a **C** sú deťmi uzla A. Podobne uzol **B** je rodičom uzlov **D** a **E** a uzly **D** a **E** sú deťmi uzla **B**. Uzly, ktoré nemajú žiadne deti sa nazývajú **listami** stromu. Na obrázku sú listami uzly **D**, **E** a **C**. Uzly, ktoré nie sú listami, nazývame **vnútornými uzlami** stromu. Ak je uzol dieťaťom jeho dieťaťa, hovoríme, že je vnukom uzla. Zovšeobecnením je výraz **potomok** uzla, ktorý označuje ľubovoľný uzol, ktorý je nejakým prapra...dieťaťom uzla. **Predok** uzla je každý uzol, ktorý je nejakým prapra...rodičom uzla. Na základe tejto definície môžeme tvrdiť, že každý uzol v strome je nejakým potomkom koreňa a zároveň koreň je predkom ľubovoľného uzla v strome.



Stromy vieme zakresliť rôznymi spôsobmi. Napríklad uzly stromu znázorníme krúžkami a hierarchické väzby šípkami.

Dôležitou charakteristikou stromu je aj jeho **výška.** Výška stromu je počet úrovní stromu znížený o 1. Koreň **A** má ako potomkov prvej generácie uzly **B** a **C** a ako potomkov druhej generácie uzly **D** a **E**. Výška stromu je tiež rovná počtu generácií potomkov koreňa. Zaujímavou vlastnosťou stromov je to, že ak si zoberieme akúkoľvek súvislú časť, tak je to opäť nejaký strom. Existujú dve definície, čo je to byť podstromom. Pre nás je zaujímavá tá „informatická“, ktorá hovorí, že podstrom stromu je taká časť stromu, ktorá sa skladá z nejakého uzla stromu a všetkých jeho potomkov. Podstrom s koreňom v uzle **B** je tvorený uzlami **B**, **D** a **E**.

Ako je možné vidieť na obrázkoch, aj kreslenie stromových štruktúr má svoje zvyklosti. Tie sú v informatike trochu „postavené na hlavu“: stromy sa kreslia s koreňom hore a listami dole.

4. Rozhodovacie stromy

V polovici deväťdesiatych rokov boli veľmi populárne knihy pre mládež, v ktorých sa čitateľ ako hlavný hrdina mohol rozhodnúť, čo spraví v aktuálnej situácii. Pre jednotlivé alternatívy bolo v knihe uvedené, na ktorej strane má čitateľ pokračovať v čítaní.

Už vieme, že menu (v mobilnom telefóne, v počítačovej aplikácií, atď.) má zvyčajne stromovú štruktúru. Výberom položiek v menu a potom neskôr v podmenu sa vlastne pohybujeme v strome volieb v smere od koreňa k listom. Listy sú uzly stromu, kde sa zrealizuje už konkrétna akcia. V každom uzle stromu, ktorý nie je listom, sa rozhodujeme, do ktorého potomka tohto uzla sa presunieme. Strom s ponukou menu je teda akýmsi rozhodovacím stromom. Špeciálnym typom rozhodovacích stromov sú binárne áno/nie rozhodovacie stromy. Každý vnútorný uzol stromu predstavuje aktivitu alebo otázku s jedným z dvoch možných výsledkov. Na základe výsledku aktivity, resp. odpovede na otázku sa rozhodujeme, či sa presunieme k „áno“ dieťaťu alebo „nie“ dieťaťu uzla. Naša rozhodovacia púť skončí vždy v liste, keďže takýto uzol nemá potomkov. List reprezentuje niektorý z možných výsledkov celého rozhodovacieho procesu – záverečné rozhodnutie.

Aktivita 1.

Vytvorte a nakreslite rozhodovací strom pre hru „Hádaj, na čo myslím“, v ktorej dopredu poznáte všetky možné „myslené“ pojmy: Požiadajte kolegu, aby povedal 8 pojmov, ktoré si môžete myslieť (napr. názvy krajín, predmety v miestnosti, významné osobnosti, …). K týmto pojmom navrhnite rozhodovací strom áno/nie otázok, ktoré vedú k odhaleniu každého z 8 pojmov.

Spolu s kolegom vyskúšajte rozhodovací strom zahraním si hry. V strome zakreslite postupnosť otázok, ktorá viedla k myslenému pojmu. Pozorujte, ako sa presúvate v rozhodovacom strome pri kladení otázok.

Aký je minimálny počet otázok, ktoré sa musia položiť, aby ste vedeli odhaliť ľubovoľný z 8 pojmov? Aký je minimálny počet uzlov takéhoto rozhodovacieho stromu? Ako je to v prípade, keď potrebujete odhaliť jeden z N pojmov?

V binárnych rozhodovacích stromoch má každý uzol buď dve deti alebo žiadne. Obmedzenie na počet detí uzla v strome znamená, že v porovnaní so stromom potomkov vieme uzly binárneho rozhodovacieho stromu uložiť v oveľa „kompaktnejších“ záznamoch. Základom je takáto štruktúra typu záznam na uloženie údajov jedného uzla stromu:

```
type
  TRozhodovaciUzol = record
    Hodnota: String;
    AnoPokracovanie, NiePokracovanie: Integer;
  end;
```

Pri použití takéhoto typu záznamu nám zíde na um otázka: ako uložiť informáciu, že daný uzol je už listom? T.j. ako zistiť, že reťazec v položke **Hodnota** je už odpoveď a nie ďalšia otázka? Môžeme sa inšpirovať spájanými zoznamami. V nich sme na uloženie toho, že prvok zoznamu nemá nasledovníka, použili špeciálnu „dohodnutú“ hodnotu 0. Využili sme to, že 0 nemôže byť platným indexom žiadneho záznamu v poli záznamov, v ktorom indexovanie políčok začína od 1. Dohodnime sa, že ak v zázname uzla máme uložené v položkách **AnoPokracovanie** a **NiePokracovanie** hodnotu 0, bude to označovať, že tento uzol je listom, t.j. nemá potomkov. V opačnom prípade do položiek **AnoPokracovanie** a **NiePokracovanie** uložíme indexy záznamov v poli záznamov, ktoré obsahujú údaje o príslušných uzloch (deťoch uzla) binárneho rozhodovacieho stromu.

Aktivita 2.

Nakreslite, ako by ste uložili rozhodovací strom s N uzlami z predchádzajúcej aktivity v poli záznamov:

```
var Otazky: array[1..N] of TRozhodovaciUzol;
```

Záznam s koreňom stromu (prvá otázka) uložte na indexe 1.

5. Binárne vyhľadávacie stromy

Strom, v ktorom má každý uzol nanajvýš dve deti, nazývame **binárny strom**. Vzhľadom na malý počet detí uzla sa zaužívalo neoznačovať ich ako prvé a druhé dieťa, ale ako **ľavé** a **pravé dieťa** uzla. Podstrom s koreňom v ľavom dieťati nazývame **ľavým podstromom** uzla a podstrom s koreňom v pravom dieťati **pravým podstromom** uzla. V binárnych rozhodovacích stromoch pre každý uzol stromu platilo, že mal buď dve deti alebo žiadne. V binárnych stromoch však pre každý uzol môže nastať jedna zo štyroch situácií: uzol nemá žiadne deti, uzol má len ľavé dieťa, uzol má len pravé dieťa a uzol má obe deti. Na uloženie binárnych stromov

v programoch používame záznamový typ podobný nasledovnému:

```
type
  TTypHodnoty = String;

  TBinUzol = record
    Hodnota: TTypHodnoty;
    Lavy, Pravy: Integer;
  end;
```

Typ **TTypHodnoty** je nami definovaný typ, ktorý určuje, akého typu sú hodnoty uložené v uzloch stromu. Položka **Hodnota** v záznamoch typu **TBinUzol** uchováva hodnotu uloženú v uzle. Položky **Lavy** a **Pravy** slúžia na uloženie „navigácie“ (indexov do poľa záznamov) k záznamu ľavého, resp. pravého dieťaťa uzla. Tak ako pri rozhodovacích stromoch, dohodnime sa, že hodnota 0 v položke **Lavy**, resp. **Pravy** bude znamenať, že uzol nemá ľavé, resp. pravé dieťa.

Binárny vyhľadávací strom sa v angličtine nazýva **binary search tree** (skratka BST).

Binárne stromy môžu mať veľa zaujímavých použití. Jedno z nich je využitie binárnych stromov ako **binárnych vyhľadávacích stromov** – skrátene BVS. Tieto stromy v sebe kombinujú dobré vlastnosti spájaných zoznamov, rozhodovacích stromov a binárneho vyhľadávania. BVS môžeme použiť iba na uloženie údajov, ktoré dokážeme navzájom nejakým spôsobom porovnávať – zvyčajne čísel, či reťazcov. Tak ako u zoznamov, aj od údajovej štruktúry strom vyžadujeme, aby umožňovala operácie pridania hodnoty, odobrania (odstránenia) hodnoty a zistenia, či zadaná hodnota je uložená v strome. Keďže prvky stromu nie sú uložené nejako usporiadanie, nemá zmysel hovoriť o poradí prvkov v strome. Vzhľadom na neexistenciu poradia hodnôt v strome, BVS sú použiteľné na uloženie množiny hodnôt. V množinách nás totiž nezaujíma poradie hodnôt v množine, ale len to, či zadaná hodnota patrí alebo nepatrí do množiny. Od množín taktiež vyžadujeme operácie pridania a odstránenia hodnoty (prvku) do resp. z množiny, ktoré BVS umožňujú. Keďže v množinách nemá zmysel mať uloženú rovnakú hodnotu viackrát, aj v BVS budeme zabezpečovať, aby každá hodnota v nich bola uložená len raz.

Okrem množín vieme BVS použiť aj na efektívnu implementáciu multimnožín (hodnota sa môže v množine vyskytovať viackrát), či slovníkov.

Pri BVS budeme potrebovať uzly do stromu nielen pridávať, ale aj odoberať. Pozrime sa, ako by sme naprogramovali „manažovanie“ poľa záznamov pre jednotlivé uzly stromu. Pripomeňme, že podobne ako u zoznamov si potrebujeme pamätať, ktoré záznamy v poli záznamov uchovávajú údaje uzlov stromu a ktoré sú voľné. Môžeme použiť nasledovné premenné:

```
const MaxUzol = 1000;
var
  Strom: array[1..MaxUzlov] of TBinUzol;
  Koren: Integer;
  Volne: array[1..MaxUzlov] of Integer;
  PocetVolnych: Integer;
```

V poli **Strom** si budeme uchovávať záznamy uzlov stromu. Premennú **Koren** využijeme na uloženie indexu toho záznamu v poli **Strom**, ktorý reprezentuje koreň BVS. Indexy voľných záznamov v poli **Strom** si uložíme v poli **Volne** vo forme zásobníka a premenná **PocetVolnych** bude uchovávať počet aktuálne uložených indexov voľných záznamov. Počiatočné naplnenie premenných správnymi hodnotami zabezpečí procedúra **Inicializacia**. Táto procedúra sa postará, že na začiatku budú všetky záznamy pridané medzi voľné a hodnota premennej **Koren** sa nastaví na 0. Hodnota 0 v premennej **Koren** bude znamenať, že BVS neobsahuje žiadne uzly.

Porozmýšľajte, ako by ste vedeli uchovať informáciu o voľných záznamoch bez použitia poľa **Volne**, resp. bez akýchkoľvek pomocných polí. Rada: Inšpirujte sa spájaným zoznamov voľných prvkov.

```
procedure Inicializacia;
var
  I: Integer;
begin
  Koren := 0;
  PocetVolnych := MaxUzlov;
  for I := 1 to MaxUzlov do
```

```
    Volne[I] := I;
end;
```

S využitím poľa **Volne** vieme napísať funkciu **PripravUzol** na pripravenie záznamu na uloženie uzla v strome a procedúru **UvolniUzol** na označenie, že záznam v poli záznamov už môže byť použitý na uloženie iného uzla stromu.

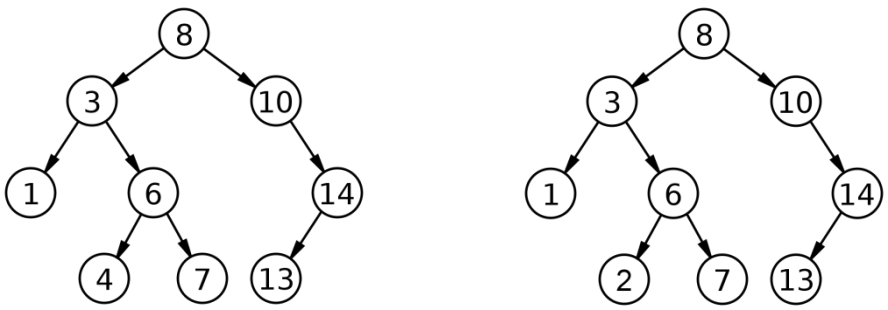
```
function PripravUzol(Hodnota: THodnota): Integer;
begin
  Result := Volne[PocetVolnych];
  Dec(PocetVolnych);

  Strom[Result].Hodnota := Hodnota;
  Strom[Result].Lavy := 0;
  Strom[Result].Pravy := 0;
end;

procedure UvolniUzol(IndexUzla: Integer);
begin
  Inc(PocetVolnych);
  Volne[PocetVolnych] := IndexUzla;
end;
```

Funkcia **PripravUzol** ako parameter dostane hodnotu, ktorá má byť uložená v pripravovanom uzle a vráti index záznamu v poli, ktorý sa vyhradil na uloženie údajov uzla. Procedúra **UvolniUzol** poznačí, že záznam na zadanom indexe už prestal byť používaný na uloženie údajov nejakého uzla a preto sa môže opätovne použiť.

Vráťme sa ale späť k BVS. Ešte sme si nepovedali, kedy je vlastne nejaký strom BVS. Binárnym vyhľadávacím stromom nazveme ľubovoľný binárny strom, v ktorom pre každý jeho vnútorný uzol platí, že hodnota v ňom uložená je:

- väčšia ako akákoľvek hodnota uložená v jeho ľavom podstrome,
- menšia ako akákoľvek hodnota uložená v jeho pravom podstrome.

Zdôraznime ešte raz, že túto vlastnosť musí spĺňať každý vnútorný uzol stromu, nie len niektoré.



Obrázok 1: Strom vľavo je binárny vyhľadávací strom. Strom vpravo nie je, pretože uzol s hodnotou 3 nespĺňa vlastnosť predpísanú pre všetky uzly BVS. V pravom podstrome uzla s hodnotou 3 je uzol s hodnotou 2, t.j. uzol s menšou hodnotou.

Aktivita 1.

Požiadajte kolegu, aby napísal 12 rôznych čísel od 1 po 100. Nakreslite binárny strom s 12 uzlami a do jeho uzlov uložte čísla, ktoré napísal kolega. Binárny strom vytvorte tak, aby bol navyše aj binárnym vyhľadávacím stromom. Snažte sa minimalizovať výšku vytvoreného stromu. Požiadajte kolegu, aby skontroloval, či nakreslený strom je skutočne BVS.

Prečo sú BVS také zaujímavé? Všimnime si, že ak chceme zistiť, či je nejaká hodnota uložená v strome, tak nemusíme prehľadávať celú údajovú štruktúru (celý strom) tak, ako to bolo pri použití spájaných zoznamov. Vďaka vlastnosti uzlov v BVS dokážeme určiť len na základe porovnania hľadanej hodnoty a hodnoty v uzle, v ktorom sa práve nachádzame, či pokračovať v hľadaní hodnoty v ľavom alebo v pravom podstrome. Ak je hľadaná hodnota menšia ako hodnota v uzle, v ktorom aktuálne sme, tak v hľadaní pokračujeme v ľavom podstrome, t.j. presunieme sa do ľavého dieťaťa aktuálneho uzla. V prípade, že hľadaná hodnota je väčšia, v hľadaní pokračujeme v pravom podstrome. Ilustrujme si proces hľadania hodnoty v BVS. Predpokladajme, že v BVS na obrázku 1 vľavo chceme zistiť, či obsahuje hodnotu 6. Začíname od koreňa stromu. V ňom je hodnota 8. Z vlastnosti BVS pre koreň vyplýva, že všetky hodnoty menšie ako 8 sú uložené v ľavom podstrome. V ďalšom kroku sa preto presunieme do ľavého dieťaťa koreňa, t.j. do uzla s hodnotou 3. Vlastnosť BVS pre uzol s hodnotou 3 hovorí, že hodnota 6, ak sa v strome nachádza, musí byť v jeho pravom podstrome. Presunieme sa preto do pravého dieťaťa uzla 3. Tu sa už ale dostávame k uzlu, ktorého hodnota je hľadaná hodnota. Hľadanie môžeme ukončiť. Čo sa stane, ak hľadáme hodnotu, ktorá sa v BVS nenachádza? Predpokladajme, že hľadáme hodnotu 9. Keďže $8 < 9$, z koreňa sa presúvame do pravého dieťaťa. Tento uzol má hodnotu 10. Teraz pre hľadanú hodnotu 9 a hodnotu uzla, v ktorom sa nachádzame, platí, že $9 < 10$. V hľadaní by sme mali teda pokračovať v ľavom podstrome. Avšak uzol, v ktorom sa nachádzame, nemá žiadne ľavé dieťa. Hľadanie nás doviedlo k neexistujúcemu uzlu. Hľadanie preto ukončujeme s výsledkom, že hodnotu sme nenašli. Všimnime si, že pri hľadaní hodnoty stále postupujeme vždy len smerom od koreňa nadol. Nikdy teda neurobíme viac krokov, ako je výška BVS. Práca vykonaná počítačom pri hľadaní hodnoty v BVS je v najhoršom prípade úmerná výške stromu. Z tohto dôvodu je minimalizácia výšky BVS veľmi dôležitá.

Aktivita 2.

Vyskúšajte si hľadanie rôznych hodnôt (uložených aj neuložených) v BVS z predchádzajúcej aktivity aplikovaním opísaného postupu.

Pozrime sa teraz, ako môžeme postup na nájdenie hodnoty v BVS zapísať v Pascale. Funkcia **JeVBVS** vráti **true** práve vtedy, keď sa parametrom zadaná hodnota nachádza v strome.

V premennej **Aktualny** si pamätáme index záznamu uzla, v ktorom sa práve nachádzame pri hľadaní hodnoty. Na začiatku je týmto indexom index koreňa. Kým je hodnota premennej **Aktualny** platným indexom záznamu, tak na základe porovnania hodnoty v uzle na indexe **Aktualny** a hľadanej hodnoty sa presunieme k „správnemu“ dieťaťu uzla. Hľadanie končíme, ak prídeme k neexistujúcemu uzlu (podmienka while-cyklu) alebo nájdeme uzol s hľadanou hodnotou (prvý if vo vnútri tela while-cyklu).

```
function JeVBVS(Hodnota: THodnota): Boolean;
var
  Aktualny: Integer;
begin
  Result := false;

  Aktualny := Koren;
  while Aktualny <> 0 do
  begin
    if Strom[Aktualny].Hodnota = Hodnota then
    begin
      Result := true;
      break;
    end;

    if Hodnota < Strom[Aktualny].Hodnota then
      Aktualny := Strom[Aktualny].Lavy
    else
      Aktualny := Strom[Aktualny].Pravy;
  end;
end;
```

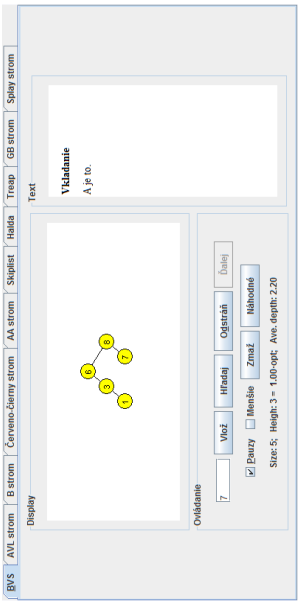
Ako pridáme novú hodnotu do BVS? Uvedomme si, že uzol s pridávanou hodnotou nemôžeme umiestniť na ľubovoľné miesto v strome, pretože by sa mohla narušiť vlastnosť BVS a binárny strom by už prestal viac byť BVS.

Aktivita 3.

Diskutujte a pokúste sa navrhnúť postup, ako pridať do BVS hodnotu, ktorá sa v ňom ešte nenachádza.

Vyskúšajte vizualizačný applet na stránke:
http://people.ksp.sk/~kuko/bak/index-sk.html

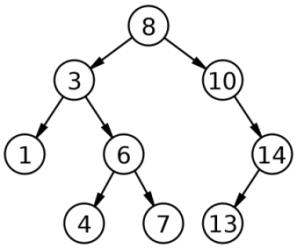
Odpozorujte a popíšte postup, ktorý je na pridanie hodnoty do BVS použitý v tomto vizualizačnom applete.



Úvahami alebo odpozorovaním z vizualizačného appletu môžeme prísť na spôsob, ako vložiť novú hodnotu do BVS tak, aby stále ostal BVS. Základná myšlienka je veľmi jednoduchá. Tvárime sa, že hodnotu do BVS nechceme vložiť, ale chceme zistiť, či sa v BVS nachádza. V každom kroku porovnávame vkladanú hodnotu s hodnotou v uzle, v ktorom práve sme. Počas tohto zostupu v BVS sa môže stať, že natrafíme na uzol s rovnakou hodnotou, ako je vkladaná hodnota. V tomto prípade môžeme vkladanie ukončiť, keďže takáto hodnota už v BVS je. V opačnom prípade nás porovnanie vždy naviguje k jednému zo synov. Keďže zostupovať sa nedá donekonečna, raz natrafíme na uzol, v ktorom nás „porovnávacia navigácia“ pošle k jeho neexistujúcemu dieťaťu. Pri vyhľadávaní je toto situácia, kedy prehlásime, že hľadaná hodnota sa v BVS nenachádza. Naopak pri vkladaní do stromu je toto situácia, kedy sme našli vhodné miesto na vloženie hodnoty. Nový uzol s vkladanou hodnotou vytvoríme na mieste neexistujúceho dieťaťa uzla, v ktorom sme hľadanie skončili. Všimnime si, že miesto pre vkladanú hodnotu sme vlastne hľadali tak, že sme sa v každom kroku presunuli do toho podstromu, do ktorého následné vloženie hodnoty neporuší vlastnosť BVS pre uzol, v ktorom sa práve nachádzame.

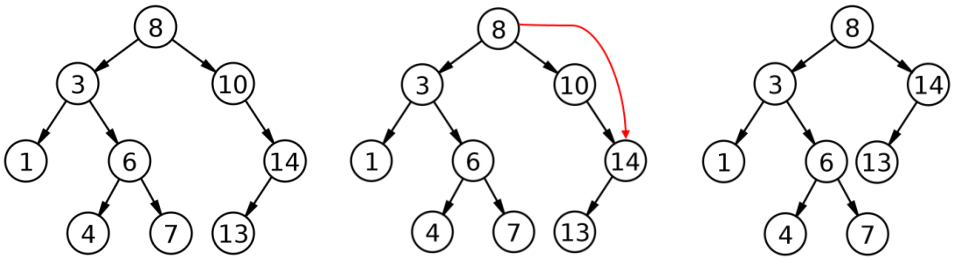
```
procedure PridajDoBVS(Hodnota: THodnota);
var
  Aktualny, RodicAktualneho: Integer;
  JeLavySyn: Boolean;
begin
  if Koren = 0 then
  begin
    Koren := PripravUzol(Hodnota);
    Exit;
  end;

  RodicAktualneho := 0;
  Aktualny := Koren;
  while Aktualny <> 0 do
  begin
    if Strom[Aktualny].Hodnota = Hodnota then
       Exit;

    RodicAktualneho := Aktualny;
    if Hodnota < Strom[Aktualny].Hodnota then
    begin
      Aktualny := Strom[Aktualny].Lavy;
      JeLavySyn := true;
    end
    else
    begin
      Aktualny := Strom[Aktualny].Pravy;
      JeLavySyn := false;
    end;
  end;

  if JeLavySyn then
    Strom[RodicAktualneho].Lavy := PripravUzol(Hodnota)
  else
    Strom[RodicAktualneho].Pravy := PripravUzol(Hodnota);
end;
```

Nahliadnime, ako je procedúra **PridajDoBVS** naprogramovaná. Ak je BVS prázdny, môžeme okamžite vytvoriť uzol s vkladanou hodnotou a spraviť ho koreňom BVS. Ak BVS nie je prázdny, najprv realizujeme akoby hľadanie vkladanej hodnoty. V premennej **Aktualny** si pamätáme index záznamu uzla, v ktorom sa práve nachádzame. Počas zostupu si navyše udržiavame v premennej **RodicAktualneho** index záznamu rodiča aktuálneho uzla a v premennej **JeLavySyn** informáciu, či uzol **Aktualny** je ľavým alebo pravým synom svojho rodiča. Hľadanie končí, keď nájdeme vkladanú hodnotu v BVS alebo keď aktuálny uzol je neexistujúcim uzlom (index jeho záznamu je 0). V druhom prípade, po skončení cyklu, vytvoríme uzol s vkladanou hodnotou. Keďže index záznamu rodičovského uzla máme uchovaný, ľahko v zázname rodiča zmeníme, že vytvorený uzol s vkladanou hodnotou je novým dieťaťom rodiča na mieste, ktoré pôvodne viedlo k neexistujúcemu dieťaťu. Poznamenajme, že pri vkladaní novej hodnoty do BVS je opäť práca vykonaná počítačom v najhoršom prípade úmerná výške stromu.



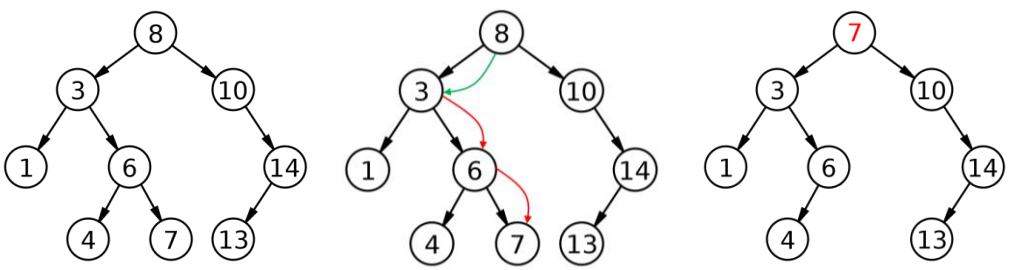
Binárny vyhľadávací strom, v ktorom uvažujeme rôzne scenáre odstránenia hodnoty.

Posledná operácia, ktorej sme sa nevenovali, je odstránenie hodnoty z BVS. Operáciu odstránenia hodnoty môžeme rozdeliť na niekoľko častí. Najprv je treba odstraňovanú hodnotu nájsť v strome. Na to vieme použiť algoritmus overujúci, či sa zadaná hodnota nachádza v BVS (použitý vo funkcii **JeVBVS**). Môžeme však nájdený uzol s odstraňovanou hodnotou len tak odstrániť? To záleží od situácie. Uvažujme BVS v poznámkovej časti. Ak je odstraňovaná hodnota 4, tak uzol s touto hodnotu jednoducho odstránime bez toho, aby sme niečo pokazili. Podobne môžeme odstrániť ľubovoľnú hodnotu v uzle, ktorý je listom. V prípade, že je odstraňovaná hodnota 10, nemôžeme uzol len tak odstrániť, pretože uzol má jedno dieťa a toto dieťa musí byť nejako pripojené k BVS. V situácii, keď má odstraňovaný uzol práve jedno dieťa, postupujeme tak, že v zázname detí jeho rodiča nahradíme odstraňovaný uzol dieťaťom odstraňovaného uzla (je len jediné, takže nemusíme vyberať) – viď obrázok 2.



Obrázok 2: Pri odstraňovaní uzla s hodnotou 10 nastavíme uzlu s hodnotou 8 ako dieťa (namiesto uzla s hodnotou 10) uzol s hodnotou 14.

Najkomplikovanejšia situácia vznikne vtedy, keď odstraňovaný uzol má obe deti – ľavé aj pravé. V tomto prípade namiesto odstraňovaného uzla odstránime iný uzol. Najprv v ľavom podstrome uzla s odstraňovanou hodnotou nájdeme uzol, ktorého hodnota je maximálna. Nájdené maximum si uložíme a uzol, ktorý ho obsahuje, odstránime z BVS. Nakoniec nájdené maximum v ľavom podstrome uložíme do uzla s odstraňovanou hodnotou, t.j. do uzla, ktorý sme pôvodne chceli odstrániť.



Obrázok 3: Odstránenie hodnoty 8 z uzla s oboma deťmi. Najprv nájdeme maximum v ľavom podstrome (kým sa dá, ideme smerom vpravo). Uzol s nájdeným maximom odstránime a jeho hodnotu uložíme do uzla, ktorý sme pôvodne chceli odstrániť.

Pri spomenutom postupe sa vynára otázka. Ako nájsť maximálnu hodnotu v ľavom podstrome uzla s odstraňovanou hodnotou? Pripomeňme, že tento uzol má obe deti a teda jeho ľavý podstrom nie je prázdny. Z vlastností BVS vyplýva, že ak chceme nájsť maximálnu hodnotu v strome, musíme sa pri zostupe v strome vybrať vždy do pravého podstromu (prečo?). Algoritmus je preto jednoduchý. Kým sa dá, posúvame sa vždy smerom k pravému dieťaťu uzla, v ktorom práve sme. Ak pravé dieťa uzla neexistuje, znamená to, že aktuálny uzol obsahuje maximálnu hodnotu. Predpokladajme, že sme uzol s maximálnou hodnotou našli. Podľa algoritmu (návodu) ho máme odstrániť – opäť ale tak, aby sme nenarušili vlastnosti BVS. Všimnime si, že nájdený uzol určite nemá pravé dieťa. Ak by ho totiž mal, určite by sme hľadanie maximálnej hodnoty neskončili u neho, ale pokračovali by sme v hľadaní. Uzol s nájdenou maximálnou hodnotou má preto buď ľavé dieťa alebo žiadne. Keďže uzol s maximom má nanajvýš jedno dieťa, na jeho odstránenie môžeme použiť postup pre jeden z dvoch prípadov, ktoré sme analyzovali už skôr (odstránenie uzla so žiadnym alebo jedným dieťaťom). Pozrime sa, ako by mohla vyzerať funkcia na odstránenie maximálnej hodnoty v ľavom podstrome zapísaná v Pascale.

```
function OdstranMaximumVlavo(KorenPodstromu: Integer):
  THodnota;
var
  Aktualny, RodicAktualneho: Integer;
begin
  RodicAktualneho := KorenPodstromu;
  Aktualny := Strom[RodicAktualneho].Lavy;

  while Strom[Aktualny].Pravy <> 0 do
  begin
    RodicAktualneho := Aktualny;
    Aktualny := Strom[Aktualny].Pravy;
  end;

  if Strom[RodicAktualneho].Lavy = Aktualny then
    Strom[RodicAktualneho].Lavy := Strom[Aktualny].Lavy
  else
    Strom[RodicAktualneho].Pravy := Strom[Aktualny].Lavy;

  Result := Strom[Aktualny].Hodnota;
  UvolniUzol(Aktualny);
end;
```

Funkcia **OdstranMaximumVlavo** dostane ako parameter index záznamu uzla, v ktorého ľavom podstrome chceme odstrániť maximálnu hodnotu, a vráti maximálnu hodnotu uloženú v odstránenom uzle.

Rozobrali sme všetky prípady, ktoré môžu nastať pri úprave BVS s cieľom odstrániť zadanú hodnotu z BVS. Aj v tomto prípade je práca vykonaná počítačom v najhoršom prípade úmerná výške BVS.

Ako by vyzerala procedúra na odstránenie hodnoty z BVS zapísaná v Pascale, je možné pozrieť si v projekte s názvom **BVS** v systéme Moodle projektu ĎVUI.

Algoritmy na pridávanie a odoberanie hodnôt (prvkov) do, resp. z binárneho vyhľadávacieho stromu patria k tým náročnejším. Ich náročnosť spočíva v tom, že žiadna zmena v strome nesmie narušiť vlastnosti BVS. Ako sme mali možnosť vidieť, práca vykonaná počítačom pri realizácii jednotlivých operácií je v najhoršom prípade úmerná aktuálnej výške BVS.

Aktivita 4.

Pre náhodne zvolených 10 čísel navrhnite postupnosť vloženia hodnôt do BVS tak, aby finálny BVS mal (a) najväčšiu možnú výšku, (b) najmenšiu možnú výšku.

Ak chceme minimalizovať prácu počítača, mali by sme robiť úpravy BVS takým spôsobom, aby jeho výška bola stále čo najmenšia. V reálnych programoch zvyčajne nepoznáme dopredu, aké hodnoty a v akom poradí budeme do BVS pridávať, resp. odoberať. Ak máme smolu, môže sa nám stať, že hodnoty budú prichádzať v takom poradí, že hĺbka BVS bude úmerná počtu hodnôt uložených v strome. Vtedy sme na tom rovnako, ako by sme na uloženie hodnôt použili oveľa jednoduchšiu údajovú

štruktúru zoznam. Našťastie sa vymyslelo veľa fínt a algoritmov ako zabezpečiť, aby BVS malo stále malú výšku – úmernú logaritmu počtu hodnôt v ňom uložených. V odbornej literatúre sa môžeme stretnúť s pojmami ako samovyvažovacie stromy, AVL stromy, či červeno-čierne stromy. Nám stačí vedieť, že sú to sofistikované údajové štruktúry podobné binárnym vyhľadávacím stromom, ktoré optimalizujú výšku stromu a tým umožňujú efektívne vykonávanie operácií pridávania, odoberania a testovania prítomnosti hodnôt.

Aktivita 5.	V applete z 3. aktivity si pozrite, ako pracujú niektoré zo samovyvažovacích stromov. Skúste odpozorovať ako pracujú a na základe pozorovania zadajte také operácie a hodnoty, aby ste prinútili strom mať čo najväčšiu výšku. Podarí sa to?

BVS a jeho modifikácie sú veľmi dôležitou údajovou štruktúrou. Dokonca možno povedať (nie je to pravda vždy, ale dosť často), že sa každým efektívnym algoritmom treba hľadať nejaký strom. Napríklad, vďaka stromom môžeme rýchlo vyhľadávať v databázach, či iných „úložiskách“ údajov.

Čo sme sa naučili

Údaje môžu byť uložené v rôznych formách. Na hierarchické uloženie údajov využívame údajovú štruktúru strom. Naučili sme sa, ako túto štruktúru uložiť v programoch prostredníctvom poľa záznamov. Predstavili sme si aj údajovú štruktúru binárny vyhľadávací strom, ktorá umožňuje veľmi efektívne implementovať základné množinové operácie: pridanie hodnoty, odstránenie hodnoty a otestovanie, či hodnota patrí do množiny.

Úlohy na precvičenie

Zadanie 1.	Navrhnite údajovú štruktúru na uloženie stromu predkov: pre každú osobu si okrem jej osobných informácií pamätáte jej otca a matku (ak sú známi). Vytvorte strom svojich predkov a nakreslite, ako by mohol byť uložený v poli záznamov.
Zadanie 2.	Navrhnite údajovú štruktúru na uloženie údajov o kolegoch v študijnej skupine s rýchlym vyhľadávaním podľa priezviska. Pre každú osobu treba uložiť priezvisko, meno, pohlavie, mesto a e-mailový kontakt. Informácie o kolegoch usporiadajte do binárneho vyhľadávacieho stromu tak, aby jeho výška bola čo najmenšia. Ako by ste strom uložili v poli záznamov?
Zadanie 3.	Naprogramujte funkciu, ktorá vráti minimálnu hodnotu uloženú v BVS.
Zadanie 4.	Formálne (matematicky) zdôvodnite, že strom s výškou h má aspoň $h + 1$ uzlov a nanajvýš $2^{h+1} - 1$uzlov. Čo vieme na základe toho povedať o výške stromu s *n* uzlami?

Revízia triedenia

V predchádzajúcej časti ste videli, že pomocou stromových štruktúr údajov sa dajú údaje organizovať tak, že pri mnohonásobnom vyhľadávaní, pridávaní a odoberaní údajov celkovo vykonáme oveľa menšiu prácu, než keby sme ich nepoužili a údaje uložili bez systému jeden s druhým.

Analogicky sa dá táto dobrá myšlienka použiť aj pri návrhu lepšieho algoritmu triedenia údajov. Táto téma svojou náročnosťou presahuje možnosti tohto materiálu a nebudeme sa jej venovať. V nasledujúcom texte si ukážeme veľmi šikovný spôsob triedenia údajov založený na inom princípe, ako boli algoritmy v kapitole Triedenie a vyhľadávanie, ktorý môžete úspešne využiť aj v bežnom živote. A na záver jeden netradičný spôsob triedenia, ktorý sa dá využiť v prípade, že triedime vhodné prvky.

Triedenie po cifrách

Nazýva sa aj *Radixsort*.

Pred tým než sa pustíme do teórie, budeme experimentovať. Je to dôležitá súčasť informatiky a netreba sa báť experimentovať s využitím počítača, alebo aj bez neho, ako to budeme robiť teraz my.

Aktivita 1.	Spomedzi kartičiek s číslami vyberte všetky obsahujúce iba jednociferné čísla. Roztrieďte ich na desať kôpok tak, že kartičky s rovnakým číslom budú na tej istej kope.
Aktivita 2.	Teraz vytrieďte kartičky s dvojcifernými číslami. Najprv ich roztrieďte na kôpky podľa hodnoty poslednej cifry – podľa cifry jednotiek (rovnaké cifry na tú istú kôpku). Teraz vzniknuté kôpky dajte všetky na seba tak, že a) Začnete kôpkou s cifrou 0, na ňu kôpku s cifrou 1, atď. až po kôpku s cifrou 9 b) Začnete kôpkou s cifrou 9, na ňu kôpku s cifrou 8, atď. až po kôpku s cifrou 0 - teda v opačnom poradí kôpok ako v a)
Úloha 1.	Prezrite si po sebe idúce kartičky aj v prípade a), aj v prípade b). Máte ohľadom poradia kartičiek nejakú hypotézu? Sformulujte ju.
Aktivita 3.	Analogicky ako Aktivita 2, ale kartičky, ktoré máte na kope ako výsledok činnosti Aktivity 2. rozložte na desať kôpok podľa hodnoty druhej predposlednej cifry – podľa cifry desiatok (rovnaké cifry na tú istú kôpku). Vzniknuté kôpky dajte na seba, predpokladáme, že na základe Aktivity 2 už viete v akom poradí.

Nie je tu tých experimentálnych úloh dajako priveľa?

Pozrime sa na predchádzajúce aktivity podrobnejšie. V prípade Aktivity 1 ste si iste všimli, že kartičky s jednocifernými číslami sú po rozložení na kôpky vlastne utriedené.

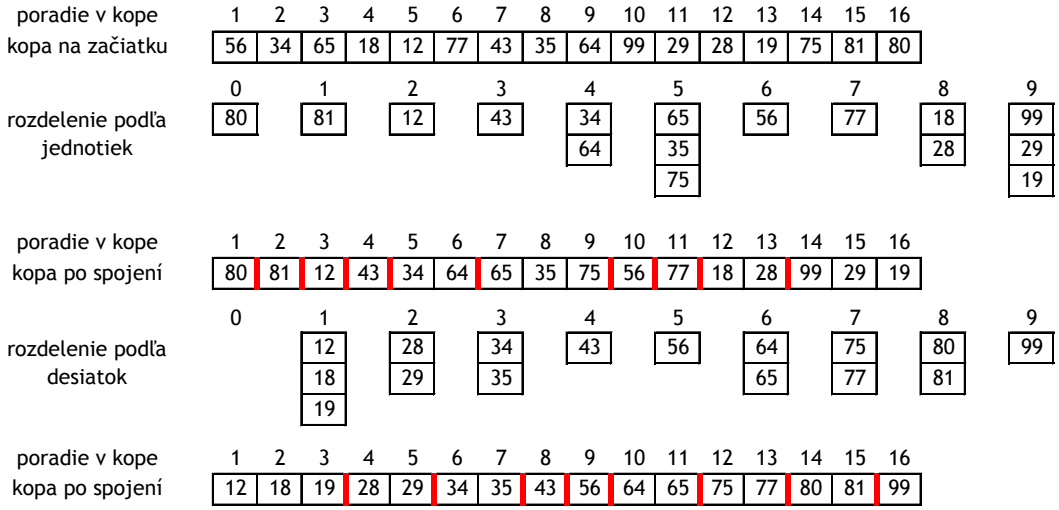
Keď ste v Aktivite 2 dali kôpky kartičiek na sebe podľa bodu a), možno ste dostali v Aktivite 3 kartičky utriedené podľa hodnoty na nich, od najmenšej po najväčšiu. Prečo „možno“? Nasledujúca aktivita objasní, že „možno“ je na mieste a nemuseli ste lístky dostať usporiadané podľa hodnôt.

Aktivita 4.

V Aktivite 3 ste mali pri rozdeľovaní na kôpky podľa druhej cifry (podľa desiatok) dve možnosti: buď ste kartičky vyberali z vrchu kopy, alebo zo spodku kopy. Vyskúšajte, čo sa stane v každej z oboch možností.

Také niečo sme už predsa videli niekde predtým.

Takže, keď kartičky dávame na kôpky a tie potom spojíme do jednej kopy, nemôžeme z nej v ďalšom brať ako prvé lístky, ktoré sme v predchádzajúcom dali na kôpky ako posledné, ale ako prvé musíme brať z výslednej kopy lístky, ktoré sme na kôpky dávali ako prvé. Ale to je predsa údajová štruktúra rad, v ktorej sme prvky pridávali na koniec a zo začiatku ich zasa odoberali. A ozaj bude postup opísaný v predchádzajúcich aktivitách vždy fungovať? Urobme si najprv malý príklad (obr. 1.)



Obr. 1. Priebeh triedenia po cifrách v prípade dvojciferných čísiel. Prvky z kopy odoberáme podľa poradia od 1. po 16. Do príslušného radu podľa cifry ich zaradíme vždy na koniec (na spodok). Spojenú kopu vytvoríme tak, že zapíšeme za seba rady pre jednotlivé cifry od nuly po deviatku. V spojenej kope sú prvky jednotlivých radov oddelené červenou čiarou.

Pri rozdeľovaní podľa desiatok by všetky jednociferné skončili v rade pre cifru 0.

Čo by sa stalo, keby sme mali na obr. 1. aj jednociferné čísla? Vlastne nič, pretože si ich môžeme pre účely tohto triedenia predstaviť ako dvojciferné s tým, že cifra na mieste desiatok by bola 0.

Teraz sa pokúsme odpovedať na otázku, či tento postup vždy funguje. Na základe riešenia Aktivít 1 až 4 a uvedeného príkladu sa budeme usilovať zovšeobecniť naše skúsenosti. Mohli sme si všimnúť, že postup prebieha v niekoľkých etapách. V každej etape rozdelíme celú kopu na 10 kôpok podľa hodnoty niektorej cifry (jednotky desiatky, stovky, tisícky,...). Nakoniec vytvoríme jednu kopu spojením všetkých kôpok za seba v poradí od kôpky pre cifru 0 až po cifru 9. Na obr. 1 vidíme po prvej etape, že v kope sú najprv všetky prvky s hodnotou jednotiek 0, potom s hodnotou 1, atď. až po prvky s hodnotou jednotiek 9. V druhej etape sú za sebou v jednotlivých radoch prvky usporiadané podľa posledných dvoch cifier.

To nie je prekvapujúce, veď kôpky sme presne tak za sebou spájali.

Úloha 2.

Presvedčte sa, že aj keď pridáte prvky s jednocifernou hodnotou, budú v druhej etape prvky v jednotlivých radoch usporiadané podľa posledných dvoch cifier.

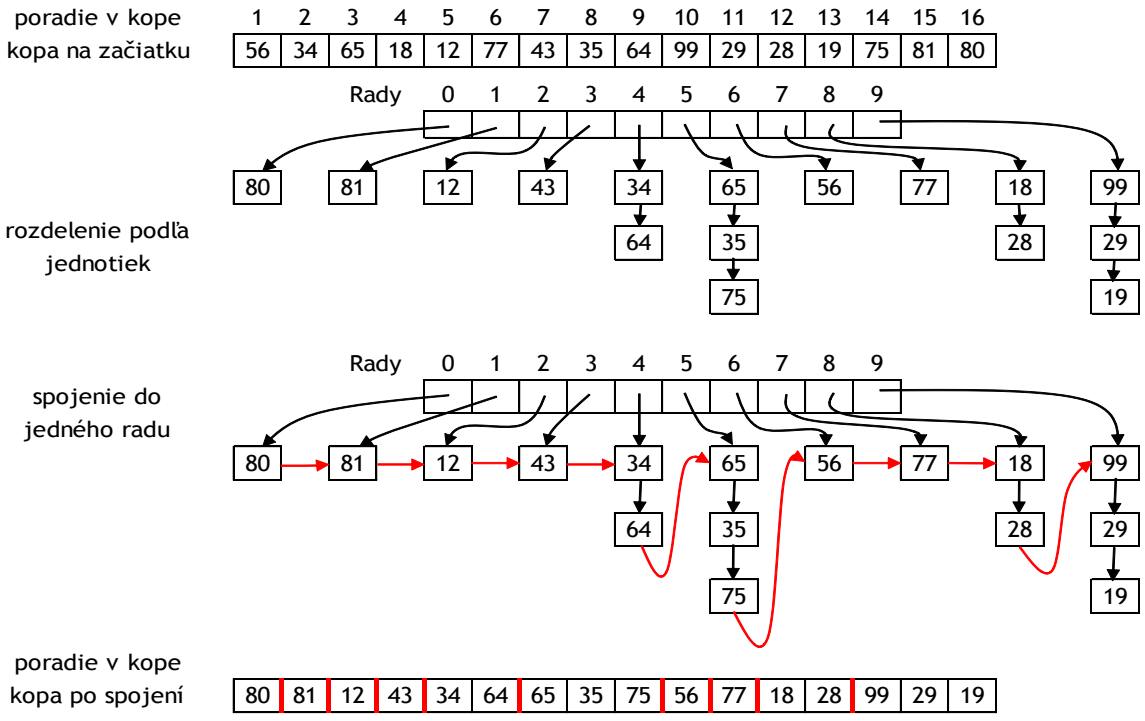
Zoberme si dve dvojciferné čísla $X = 10A + B$ a $Y = 10C + D$, kde A, B, C, D sú ich cifry. Nech $X < Y$. V prvej etape sme čísla rozdelili do skupín podľa jednotiek, t.j. podľa hodnoty B a D. Tvrdíme, že po druhej etape musia byť čísla vo výslednej kope v poradí najprv X a potom Y. Vzhľadom na to, že $X < Y$, buď $A < C$ a vtedy je X

v poradí pred Y, pretože vo výslednom poradí sú prvky radu pre druhú cifru rovnajúcu sa A pred prvkami radu pre druhú cifru C. Alebo nastane prípad $A = C$ a vtedy sú X aj Y v druhej etape v tom istom rade, ale vtedy musí platiť $B < D$ (prečo?), teda po prvej etape museli byť v želanom poradí. Túto úvahu môžeme rozšíriť na ľubovoľný počet cifier, takže uvedený postup naozaj vždy funguje.

Koľko práce vyžaduje tento postup? Vzhľadom na to, že sme pre tento problém neuviedli program, bude náš odhad založený len na tom, že vzhľadom na naše doterajšie programátorské skúsenosti si dokážeme potrebné programátorské detaily predstaviť. Predpokladajme, že týmto postupom chceme utriediť n k-ciferných čísiel. Je zrejmé, že na utriedenie potrebujeme k etáp. Koľko práce vyžaduje jedna etapa? V rámci i-tej etapy každý prvok odoberieme z kopy, ktorú realizujeme radom. Z kopy odoberáme vždy len z „vrchu“ (z jedného konca). To vieme urobiť s konštantným počtom operácií. Každý prvok dáme na koniec radu podľa i-tej cifry. Na zistenie i-tej cifry potrebujeme tiež len konštantný počet operácií. Rady budeme mať v desať prvkovom poli **Rady : array[0..9] of TRad**. Keď bude i-ta cifra napríklad c, prvok zaradíme na koniec radu **Rad[c]**. To tiež vieme zrealizovať s konštantným počtom operácií. Na záver etapy ešte musíme pospájať **Rady** do jedného radu (jednej kopy). Keď rady realizujeme spájanými zoznamami, vieme dva rady spojiť za seba znovu použitím konštantného počtu operácii, teda aj 10 radov bude vyžadovať počet operácii nezávislý od n. Keď to všetko sčítame, dostaneme, že celá etapa bude vyžadovať počet operácií (lineárne) úmerný počtu prvkov. Celý postup triedenia po cifrách teda vyžaduje počet operácií úmerný kn. To je ale výrazne menej, než bolo v prípade triedenia vsúvaním aj výberom. Je to preto, lebo neporovnávame vzájomne hodnoty prvkov, ale len ich cifier, pričom využívame, že hodnoty sú čísla.

A ja som si myslel, že najrýchlejšie triedenie n prvkov vyžaduje až $n \log_2 n$ operácií.

Na nasledujúcom obrázku je znázornený priebeh 1. etapy realizovaný pomocou poľa radov **Rady**.



Obr. 2. Znázornenie priebehu 1. etapy triedenia po cifrách. Kopa je realizovaná tiež radom. Po rozdelení do radov podľa cifier sa rady spoja opäť do jedného radu (červené šípky) a nasleduje ďalšia etapa.

Nazýva sa aj *Countsort*.

Triedenie počítaním

Predstavte si, že by ste mali utriediť podľa veľkosti veľa prvkov, medzi ktorými by ich bolo iba málo s rôznymi hodnotami. V takom prípade sa dá použiť nasledujúca myšlienka. Pre všetky rôzne prvky spočítame, koľkokrát sa ktorý z nich vyskytuje. Potom už vieme želaný výsledok – stačí vypísať príslušný počet prvkov. Ak navyše vieme prvky s rôznymi hodnotami rýchlo utriediť, dostávame zaujímavé riešenie úlohy triedenia.

Napríklad ak triedime 1000 čísiel z intervalu 1 až 10, použijeme hodnotu čísla ako index do poľa **Pocet**, v ktorom si budeme pamätať počet prvkov s touto hodnotou. Deklarácie poľa **Prvok** a **Pocet** môžu vyzerať nasledujúco:

```
Var
  Prvok : array[1..1000] of Integer;
  Pocet : array[1..10] of Integer;
```

Na začiatku musíme pole **Pocet** „vynulovať“. Výskyt prvku na pozícii **i** započítame vykonaním príkazu **Inc(Pocet[Prvok[i]])**. Keď takto spracujeme všetky prvky, v poli **Prvok** budeme mať ich počty v poli **Pocet**. Teraz už stačí len pole **Prvok** upraviť (prepísať) tak, aby bolo utriedené. Nasledujúci program poskytuje detaily realizácie.

```
const
  MinHodnota = 1;
  MaxHodnota = 10;
  MaxPocetPrvkov = 1000;
type
  Hodnota = MinHodnota..MaxHodnota;
var
  Prvok : array[1..MaxPocetPrvkov] of Integer;
  Pocet : array[1..10] of Integer;
  I, J : Integer;
begin
  for I := MinHodnota to MaxHodnota do
    Pocet[I] := 0;
  for I := 1 to MaxPocetPrvkov do
    Inc(Pocet[Prvok[I]]); // spočítame počty rôznych prvkov
  J := 0; // počet zapísaných utriedených prvkov
  for I := MinHodnota to MaxHodnota do // podľa veľkosti
    while Pocet[I] > 0 do begin // ešte nie sú všetky zapísané
      Inc(J);
      Prvok[J] := I;  // zapíšeme ďalší prvok
      Dec(Pocet[I])  // zmenšíme počet nezapísaných
    end
end;
```

Koľko operácií vyžaduje utriedenie n prvkov týmto spôsobom? Každý prvok spracúvame práve dvakrát. Raz keď počítame jeho výskyt a druhý raz, keď ho zapisujeme do utriedeného poľa. Teda celkovo tento spôsob vyžaduje počet operácií (lineárne) úmerný počtu prvkov n.

Čo sme sa naučili

Že existujú aj také algoritmy triedenia, ktoré nevyužívajú pri triedení vzájomné porovnávanie prvkov, ale priamo hodnoty prvkov.

Čo sme sa naučili v tomto module

Zhrnutie

Tento modul zoznámil s

- údajovou štruktúrou záznam a pole záznamov
- údajovou štruktúrou strom
- algoritmom jednoduchého a binárneho vyhľadávania v poli
- algoritmom triedenia vsúvaním a výberom najväčšieho prvku
- s ukážkou algoritmov triedenia, ktoré na triedenie namiesto porovnávania používajú hodnoty prvkov.

Jednoduchou formou priblížil tvorbu jednoduchých algoritmov.

Preverenie výstupných vedomostí

Účastník vzdelávania vie použiť údajovú štruktúru záznam, pole záznamov a strom a zoznámil sa s jednoduchými algoritmami triedenia a vyhľadávania v poli. Je schopný naprogramovať jednoduché úlohy využívajúce prebrané údajové štruktúry.

Literatúra a použité zdroje

Základné materiály:

[1] Wirth, N. (1988) *Algoritmy a Dátové štruktúry.* Bratislava: Alfa 1988.
[2] Blaho, A. (2006) *Informatika pre stredné školy, Programovanie v Delphi*, SPN, Bratislava
[3] Blaho, A., Salanci, Ľ. (2009) Programovanie 1 až 9, študijné materiály DVUI, ŠPÚ, Bratislava
[4] Czimmermann, P., Krajči, S. (2009) Matematika pre učiteľov informatiky 3, študijné materiály DVUI, ŠPÚ, Bratislava

Internetové zdroje:

Vizualizácie rôznych štruktúr údajov a algoritmov

[5] Kováč J. http://people.ksp.sk/~kuko/bak/big/
[6] Kučera, L. http://kam.mff.cuni.cz/~ludek/Algovision/Algovision.html

Iné zdroje:

[7] Dijkstra, E., W., Feijen, W., A Method of Programming, Addison-Wesley, 1988

Tento študijný materiál vznikol ako súčasť národného projektu Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika v rámci Aktivity „Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ“.

Autori © RNDr. Michal Winczer, PhD.
RNDr. František Galčík, PhD.

Názov Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Podnázov Algoritmy a údajové štruktúry 2

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti RNDr. Peter Gurský, PhD.
doc. RNDr. Gabriela Andrejková, CSc.

Počet strán 40

Náklad 300 ks

Prvé vydanie, Bratislava 2010

Vydal Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, v súčinnosti s Univerzitou Pavla Jozefa Šafárika v Košiciach, Univerzitou Komenského v Bratislave, Univerzitou Konštantína Filozofa v Nitre, Univerzitou Mateja Bela v Banskej Bystrici a Žilinskou univerzitou v Žiline

Vytlačil BRATIA SABOVCI, s r.o., Zvolen

ISBN 978-80-8118-039-2