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 1

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 údajové štruktúry 1

Garant predmetu:

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

Autori:

RNDr. Michal Winczer, PhD.
KZVI FMFI UK, Bratislava
RNDr. František Galčík, PhD.
ÚINF PF UPJŠ, Košice

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

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: dvojrozmerným poľom, zoznamom, zásobníkom a radom. Ukážeme základné myšlienky tvorby jednoduchých algoritmov s prihliadnutím na vedomosti, ktoré získali na základe absolvovania modulov Programovanie 1 až 9.

Obsah

Úvod

Milá čitateľka, milý čitateľ, v učebných materiáloch Programovanie 1 až 9 ste sa zoznámili so základmi programovania v programovacom jazyku Pascal v prostredí Lazarus. Programovací jazyk slúži na komunikáciu človeka s počítačom. Dá sa to veľmi zjednodušene prirovnať k ľudskému jazyku, ktorým chceme komunikovať s inými ľuďmi. Všetci vieme, že jazykov, ktorými komunikujú ľudia je veľa, pritom je v zásade jedno o aký jazyk ide, ľudia v ňom dokážu vyjadriť svoje myšlienky (samozrejme v materskom jazyku dokonalejšie ako v cudzom). Ovládanie jazyka ale nehovorí vôbec nič o tom, ako rozmýšľame a ako budeme formulovať naše myšlienky, je to iba nástroj na ich vyjadrovanie (do istej miery nás limituje slovná zásoba). Podobne je to aj s programovacími jazykmi. Je ich tiež veľmi veľa a z pohľadu bežného programátora je celkom jedno o aký programovací jazyk ide, pretože je to len nástroj, ktorým vyjadruje svoje myšlienky tak, aby im porozumel počítač. Existujú aj špecializované programovacie jazyky, ktorých analógiou by v ľudskej reči mohli byť jazyky rôznych špecialistov (lekárov, matematikov, chemikov a pod.). Týmito špeciálnymi programovacími jazykmi sa nebudeme v tomto texte zaoberať.

Čo sa týka výrazových prostriedkov, sú programovacie jazyky oveľa jednoduchšie a obmedzujúcejšie ako jazyky, ktoré používajú na komunikáciu ľudia. Čo je však veľmi dôležité, programovacie jazyky sú o na rozdiel od prirodzených jazykov celkom **jednoznačné**.

Základnými prostriedkami, ktorými sa v programovacom jazyku vyjadrujeme sú príkazy a údajové (dátové) štruktúry. Prostredníctvom nich formulujeme v reči, ktorej počítač rozumie, čo chceme a akým spôsobom chceme, aby to počítač vypočítal. Tieto postupy na riešenie konkrétnych úloh, zapísané v programovacom jazyku, nazývame programy. Program počítač vie automaticky zrealizovať. Niklaus Wirth, autor jazyka Pascal napísal knihu, ktorá vyšla aj v slovenčine [1] a jej anglický názov to vyjadruje veľmi výstižne: Algorithms + Data Structures = Programs.

Knihu N. Wirtha vrelo odporúčame všetkým, ktorý to myslia s programovaním vážne.

Tvorba algoritmov a použitie údajových štruktúr patrí medzi nevyhnutné predpoklady tvorby prakticky použiteľných programov. Podobne ako v ľudskej reči je posúdenie vyjadrenia určitej myšlienky možné hodnotiť z pohľadu zrozumiteľnosti, stručnosti, slušnosti, krásy a ďalších hľadísk len subjektívne, aj vyjadrenie nejakého postupu v programovacom jazyku môžeme posúdiť do značnej miery tiež len subjektívne, je tu ale niekoľko dôležitých odlišností: objektívne môžeme posúdiť **efektívnosť** konkrétneho programu (t.j. koľko operácií počítač musí vykonať, aby vyriešil konkrétny prípad úlohy) a tiež **správnosť** programu (či rieši ozaj tú úlohu, ktorú chceme vyriešiť).

V predkladaných dvoch moduloch sa nebudeme snažiť čitateľa naučiť ako tvoriť algoritmy ani neposkytneme prehľad existujúcich údajových štruktúr. (Ani v predmete slovenčina nás neučia ako napísať dobrý sloh. Slohy sa iba píšu a zvyčajne čím viac kníh žiak prečíta tým lepšie sa vie vyjadrovať). Pokúsime sa aspoň trochu pootvoriť dvere do *umenia* tvorby algoritmov a používania údajových štruktúr. Čitateľom želáme veľa trpezlivosti pri čítaní cudzích a písaní vlastných programov. Autori nepoznajú inú cestu vedúcu k zdokonaleniu sa v programovaní.

Tvorba programov patrí z právneho hľadiska do rovnakej kategórie ako písanie literárnych diel. Ide teda jednoznačne o umenie.

Cieľ modulu

Po absolvovaní tohto modulu sa od účastníka vzdelávania očakáva, že

- bude vedieť použiť dvojrozmerné pole, pri riešení jednoduchých úloh,
- bude vedieť, že okrem tých údajových štruktúr, s ktorými sa zoznámil v moduloch Programovanie, existujú aj ďalšie údajové štruktúry,
- bude vedieť, že jeden problém sa dá vyriešiť rôznymi spôsobmi, ktoré sa líšia spôsobom použitia údajových štruktúr.

Vstupné vedomosti

Požadované prerekvizity

Všetky moduly Programovanie 1 až 9.

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,
- 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.

Dvojrozmerné pole

1. Obrázok ako dvojrozmerné pole farieb

V moduloch Programovanie 1-9 sme sa postupne zoznámili s komponentom **TImage** (kresliaca plocha). Tento komponent zobrazuje rastrový (bitmapový) obrázok a navyše ho aj umožňuje kresliť použitím príkazov pre podkomponent **Canvas**. Ak si rastrový obrázok dostatočne priblížime, uvidíme, že v skutočnosti sa skladá z malých farebných štvorčekov - **pixelov**. Na kresliacu plochu sa teda môžeme pozerať ako na dvojrozmernú mriežku s malými, rôzne zafarbenými políčkami. Farby týchto políčok sme doposiaľ menili pomocou kresliacich príkazov ako **Rectangle**, **LineTo**, či **Ellipse**. Každý z týchto príkazov zmenil farbu jedného alebo mnohých bodov v kresliacej ploche podľa toho, čo mal nakresliť. Farbu jednotlivých políčok rastrového obrázka však vieme meniť aj priamo. Nasledujúce priradenie ukazuje priamu zmenu farby pixelu so súradnicami *[X: 50, Y:100]* v kresliacej ploche **Image1** na červenú farbu.

Obrazový prvok alebo **pixel** (skratka z angl. picture element - obrazový prvok; po anglicky aj pel/PEL), je najmenšia jednotka digitálnej rastrovej (bitmapovej) grafiky. Predstavuje jeden svietiaci bod na monitore, resp. jeden bod obrázku zadaný svojou farbou.

```
Image1.Canvas.Pixels[50, 100] := clRed;
```

Ako sa môžeme dovtípiť z príkladu, prvé z čísel v hranatých zátvorkách určuje x-ovú súradnicu a druhé y-ovú súradnicu pixelu. Hodnota, ktorú môžeme takto priradiť pixelu kresliacej plochy, musí byť typu **TColor**. Pripomeňme si základné fakty o súradniciach v kresliacej ploche:

- aktuálne rozmery kresliacej plochy vieme zistiť pomocou vlastností **Width** a **Height** komponentu **TImage** (napr. **Image1.Width, Image1.Height**),
- pixel (bod) v ľavom hornom rohu má súradnice *[X:0, Y:0]*, v pravom dolnom rohu je pixel so súradnicami *[X: Width-1, Y: Height-1]*, t.j. v obrázku so šírkou 300 pixelov a výškou 200 pixelov má pixel v pravom dolnom rohu súradnice *[X:299, Y:199]*.

Vyskúšajte aplikáciu **Lupa** (**Magnifier**) na zväčšenie obsahu obrazovky.

Pozrime sa, ako by sme dosiahli zmenu farby 50 náhodných pixelov na modrú farbu.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X, Y: Integer;
  I: Integer;
begin
  for I := 1 to 50 do
  begin
    X := Random(Image1.Width);
    Y := Random(Image1.Height);
    Image1.Canvas.Pixels[X, Y] := clBlue;
  end;
end;
```

Pripomeňme, že funkcia **Random(N)** vráti náhodné celé číslo od 0 po N-1.

Oveľa krajším príkladom je vlastná verzia príkazu **FillRect** na nakreslenie vyplneného obdĺžnika. Naša verzia tejto procedúry bude mať 5 parametrov (x-ová súradnica ľavého horného rohu, y-ová súradnica ľavého horného rohu, šírka, výška a farba výplne obdĺžnika).

```
procedure VyplnenyObdlznik(X, Y, S, V: Integer; F: TColor);
var
  XX, YY: Integer;
begin
  for XX := X to X+S-1 do
    for YY := Y to Y+V-1 do
      if (XX >= 0) and (XX < Form1.Image1.Width) and
           (YY >= 0) and (YY < Form1.Image1.Height) then
        Form1.Image1.Canvas.Pixels[XX, YY] := F;
end;
```

Z globálnych procedúr a funkcií pristupujeme ku komponentom formulára tak, že pridáme **Form1.** pred ich meno.

Všimnime si podmienku, ktorá sa vyhodnocuje pred samotným priradením farby pixelu. Podobne, ako pri práci s poľom sme si dávali veľký pozor na to, aby sme nepristúpili k neexistujúcemu prvku poľa, aj tu si dávame pozor na to, aby sme nemenili farbu neexistujúceho pixelu.

Už vieme, že vlastnú farbu je možné v Lazarus-e namiešať pomocou funkcie **RGBToColor.** Naopak pomocou funkcií **Red, Green** a **Blue** dokážeme získať jednotlivé farebné zložky farby.

```
var
  R, G, B: Integer;
  F: TColor;
begin
  F := clMagenta;
  R := Red(F);
  G := Green(F);
  B := Blue(F);
end;
```

Pomocou **Pixels** dokážeme konkrétnemu pixelu nielen zmeniť farbu, ale aj zistiť jeho aktuálnu farbu.

```
var Farba: TColor;
begin
  Farba := Image1.Canvas.Pixels[50, 100];
  //...
end;
```

Znalosť farby pixelu môžeme využiť pri rôznych „klikacích“ aktivitách.

Aktivita 1.	Vytvorte program, ktorý pri kliknutí do kresliacej plochy na červený pixel (bod) nakreslí červený kruh s polomerom 15 a so stredom v bode kliknutia. Pridajte do formulára tlačidlo, ktoré nakreslí jeden náhodný červený kruh s polomerom 20.

Pozrime sa, ako by mohla vyzerať procedúra obsluhujúca udalosť **OnMouseDown**.

```
procedure TForm1.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Image1.Canvas.Pixels[X, Y] = clRed then
  begin
    Image1.Canvas.Brush.Color := clRed;
    Image1.Canvas.Pen.Color := clRed;
    Image1.Canvas.Ellipse(X-15, Y-15, X+15, Y+15);
  end;
end;
```

Prácu s pixelmi rastrovej grafiky si môžete precvičiť aj na niektorej z týchto úloh:

Úloha 1.	Spočítajte celkový počet pixelov červenej farby v kresliacej ploche. Poznamenajme, že z počtu pixelov by sme vedeli odvodiť (keby sme poznali rozmery či obsah jedného pixelu), aký približný obsah majú nakreslené červené útvary.
Úloha 2.	Do formulára pridajte tlačidlo, ktoré všetky body červenej farby prefarbí na rovnakú, náhodne zvolenú farbu.

Klikaním do kresliacej plochy v predchádzajúcom príklade vieme akurát tak prilepiť nový kruh k už existujúcemu červenému útvaru. Postupným klikaním nám vznikne nejaký (stále väčší a väčší) červený útvar. Bolo by pekné, ak by sme nakreslenému útvaru pridali obrys, aby jeho hranice bolo lepšie vidieť. Ako ale nakresliť obrys takého komplikovaného útvaru? Namiesto kreslenia obrysových čiar len zmeníme farbu bodov po obvode červeného útvaru na čiernu. Presnejšie každý nečervený bod, ktorý je susedom červeného bodu, prefarbíme na čierno.

Nižšie uvedené riešenie na nakreslenie obrysu spočíva v tom, že pre každý pixel kresliacej plochy (prechod dvoma vnorenými cyklami) zisťujeme, či niektorý z jeho susedných pixelov nie je náhodou červenej farby. Ak je farba skúmaného pixelu rôzna od červenej a zároveň sme zistili, že má červeného suseda, zmeníme jeho farbu na čiernu. Podobne, ako v úlohe na nakreslenie vyplneného obdĺžnika, využívame podmienený príkaz, aby sme sa nepýtali na farbu neexistujúceho pixelu kresliacej plochy. Táto podmienka má význam pri zisťovaní prítomnosti suseda červenej farby u pixelov na okraji kresliacej plochy.

```
procedure TForm1.Button4Click(Sender: TObject);
var
  X, Y: Integer;
  CS: Boolean;
begin
  for X := 0 to Image1.Width-1 do
    for Y := 0 to Image1.Height-1 do
    begin
      CS := false;

      if X > 0 then
        CS := CS or (Image1.Canvas.Pixels[X-1, Y] = clRed);
      if X < Image1.Width-1 then
        CS := CS or (Image1.Canvas.Pixels[X+1, Y] = clRed);
      if Y > 0 then
        CS := CS or (Image1.Canvas.Pixels[X, Y-1] = clRed);
      if Y < Image1.Height-1 then
        CS := CS or (Image1.Canvas.Pixels[X, Y+1] = clRed);

      if CS and (Image1.Canvas.Pixels[X, Y] <> clRed) then
        Image1.Canvas.Pixels[X, Y] := clBlack;
    end;
end;
```

Premenná **CS** (skratka pre Červený Sused) uchováva informáciu, či sme už pre aktuálne analyzovaný pixel so súradnicami *[X, Y]* našli nejakého červeného suseda.

Pri riešení využívame, že `CS := A or B or C;` vieme nahradiť postupnosťou priradení:

```
CS := false;
CS := CS or A;
CS := CS or B;
CS := CS or C;
```

Ďalšie úlohy na precvičenie

Zadanie 1.	Načítajte do kresliacej plochy obrázok zo súboru: `Image1.Picture.LoadFromFile('obrazok.bmp');` Naprogramujte procedúru **Sietka**, ktorá každý druhý pixel v riadku zmení na biely (vyrobí dierku). Jednotlivé dierky v za sebou idúcich riadkoch nech sú voči sebe posunuté o 1 pixel.
Zadanie 2.	Vymaľujte kresliacu plochu tak, aby pixel vzdialený *r* (po zaokrúhlení na celé číslo) od stredu kresliacej plochy mal farbu `RGBToColor(r, r, r)`. Ak je *r* väčšie ako 255, použite hodnotu 255. Experimentujte s rôznymi variáciami parametrov pre funkciu **RGBToColor** (napr. `RGBToColor(r, r div 2, r mod 10)`)
Zadanie 3.	Naprogramujte procedúru, ktorá horizontálne preklopí obsah kresliacej plochy (vymení sa poradie pixelových riadkov v kresliacej ploche).
Zadanie 4.	Naprogramujte procedúru **KruhSDierou(X, Y, R1, R2, Farba)**, ktorá nakreslí vyplnený kruh s vonkajším polomerom **R1** a polomerom „diery“ (nevyplňovaná časť) **R2**. Návod: Použite postup podobný postupu v Zadaní 2.
Zadanie 5.	Naprogramujte procedúru na rastrovanie (vyštvorčekovanie) obdĺžnikovej časti kresliacej plochy tak, ako sa používa v TV na zakrytie nezverejniteľných častí obrazu. Pri rastrovaní sa oblasť rovnomerne rozdelí na malé štvorčeky. Farba každého štvorčeka sa vypočíta ako priemerná farba pixelov v tomto štvorčeku (priemerná farba vznikne spriemerovaním farebných zložiek farieb pixelov).

2. Deklarácia dvojrozmerných polí

Zo štruktúrovaných typov sme sa už stretli s typom (jednorozmerné) pole. Pripomeňme si, čo už vieme o jednorozmerných poliach:

- všetky prvky poľa sú rovnakého typu,
- k prvkom poľa pristupujeme pomocou indexu – zapisujeme ho do [ ] zátvoriek,
- indexy prvkov poľa sú určené intervalom nejakého ordinálneho typu
- zvyčajne si pole predstavujeme ako jednoriadkovú tabuľku, ktorá má očíslované prvky od 1 do počet prvkov.

Ordinálny typ je taký typ, ktorého hodnoty možno lineárne usporiadať a pre každú hodnotu vieme určiť jej bezprostredného predchodcu a nasledovníka (vieme použiť **Inc** a **Dec**) Ordinálnymi typmi v jazyku Pascal sú napríklad **Integer, Boolean, Char** a **Byte**. Typ **Real** nie je ordinálnym typom. (Aká hodnota by bola predchodcom reálnej hodnoty 3.14 ?)

```
var Teploty: array[1..7] of Real;
```

1	2	3	4	5	6	7
18.4	19.2	20.5	17.6	18.9	18.5	22.5

V reálnom živote a v mnohých počítačových aplikáciách sa však oveľa častejšie stretávame s tabuľkami, ktoré majú viac než len jeden riadok. Na vytvorenie viacriadkovej tabuľky využívame dvojrozmerné polia – ďalší zo štruktúrovaných typov jazyka Pascal. Voláme ich dvojrozmerné preto, že (tak ako tabuľka) majú dva rozmery – počet riadkov (výšku) a počet stĺpcov (šírku). Pozrime sa na to, ako zadeklarovať štruktúrovanú premennú typu dvojrozmerné pole. Nazvime ju **Tabulka** a bude sa skladať z 3 riadkov a 7 stĺpcov indexovaných celými číslami od 1.

```
var Tabulka: array[1..3, 1..7] of Real;
```

Tento zápis hovorí, že v premennej **Tabulka** budú riadky indexované číslami od 1 po 3 (prvý interval v deklarácii) a stĺpce indexmi od 1 po 7 (druhý interval v deklarácii). Všetky prvky dvojrozmerného poľa **Tabulka** budú reálneho typu (**Real**), celkový počet prvkov bude 3 x 7 = 21.

	1	2	3	4	5	6	7
1	18.6	19.0	20.5	19.7	21.0	20.4	18.6
2	30.8	29.2	28.7	30.2	30.5	31.6	30.5
3	27.4	28.8	27.0	27.3	28.3	29.0	28.7

Ak vo voľbách prekladača máme zapnutú kontrolu rozsahu, pri prístupe k neexistujúcemu prvku poľa dostaneme chybovú správu **„Range check error“.** Kontrolu rozsahu sa odporúča mať zapnutú (viď Programovanie 7) .

Ak chceme pristúpiť k nejakému prvku dvojrozmerného poľa, jeden index, ako v prípade jednorozmerného poľa, už nestačí. Potrebujeme dva indexy: index riadka (prvý index) a index stĺpca (druhý index), ktoré jednoznačne identifikujú prvok v dvojrozmernom poli. Oba indexy musia byť vždy v príslušnom rozsahu pre riadky, resp. stĺpce určenom pri deklarovaní poľa. Napríklad prvok **Tabulka[2, 4]** má hodnotu *30.2*, prvok **Tabulka[3, 7]** má hodnotu *28.7*. Naopak prvok **Tabulka[5, 4]** v poli **Tabulka** neexistuje, pretože pre prvý index sú platné len hodnoty 1 až 3.

S prvkami dvojrozmerného poľa sme už pracovali v predchádzajúcej časti venovanej rastrovej grafike. Pracovali sme s dvojrozmerným poľom **Pixels**, ktorého prvky boli typu **TColor**. Všimnime si ale jeden podstatný rozdiel. V poli **Pixels** sme ako prvý index uvádzali x-ovú súradnicu, čo je analógia stĺpca, a ako druhý index y-ovú

súradnicu pixelu, ktorá zodpovedá riadku pixelu v kresliacej ploche. Teda presne naopak, ako to uvádzame teraz. V skutočnosti v Pascale nezáleží na tom, čo vnímame ako riadok a čo ako stĺpec. V Pascale uvažujeme vždy len dvojicu indexov príslušného rozsahu, ktorá identifikuje jeden z prvkov poľa. Vplyvom matematiky (v maticiach je prvý index riadok a druhý index stĺpec) a tiež vplyvom toho, ako sú jednotlivé prvky poľa uložené v pamäti počítača, zaužívalo sa, že prvý index označuje riadok a druhý index stĺpec. Ak budeme hovoriť o dvojrozmernom poli s rozmermi **m x n**, myslíme tým pole s **m** riadkami a **n** stĺpcami. U poľa **Pixels** je to naopak preto, že pri práci s grafikou sa zaužívalo ako prvú hodnotu uvádzať x-ovú súradnicu a ako druhú hodnotu y-ovú súradnicu.

Vo zvyšných častiach tejto tematickej jednotky si na praktických príkladoch ukážeme, ako vhodne deklarovať a používať dvojrozmerné polia.

3. Dvojrozmerné pole ako tabuľka

Jedno zo zadaní, ktoré sa riešilo v module Programovanie 7, spočívalo v tom, že sme pri pohybe myšou mali zbierať body, v ktorých sa nachádza kurzor myši. Robili sme záznam trajektórie pohybu myši. Keďže každý bod (poloha kurzora myši) je popísaný svojou x-ovou a y-ovou súradnicou, vytvorili sme dve (veľké) polia – jedno pre zaznamenanie x-ových súradníc, druhé pre zaznamenanie y-ových súradníc bodov. Okrem toho sme použili ďalšiu premennú, v ktorej sme si pamätali, koľko bodov sme načítali, resp. koľko prvých údajov v poli sú platné súradnice bodov.

```
var
  XX, YY: array[1..10000] of Integer;
  Pocet: Integer = 0;
```

Na tieto dve polia sa môžeme pozerať aj ako na dva stĺpce veľkej tabuľky bodov, v ktorej sa v i-tom riadku nachádzajú informácie o i-tom bode. Takúto tabuľku s veľkým počtom riadkov a dvoma stĺpcami vieme pomocou dvojrozmerného poľa vytvoriť takto:

```
var
  Body: array[1..10000, 1..2] of Integer;
  Pocet: Integer;
```

Prvý stĺpec v každom riadku uchováva x-ovú súradnicu bodu a druhý stĺpec y-ovú súradnicu, t.j. i-ty bod má x-ovú súradnicu **Body[I][1]** a y-ovú súradnicu **Body[I][2]**. Premenná **Pocet** určuje, koľko bodov sme zozbierali, resp. že riadky s indexmi 1 až **Pocet** (**Pocet <= 10000**) uchovávajú údaje o týchto zozbieraných bodoch.

Dvojrozmerné pole **Body** pre hodnotu **Pocet = 5.**

	1	*2*
1	100	50
2	132	67
3	143	89
4	156	132
5	148	134
6	?	?
7	?	?
...	?	?
10000	?	?

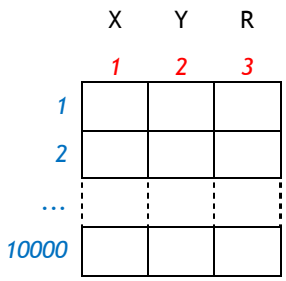
Pamätajme si: Ak vytvoríme akúkoľvek premennú, tak jej počiatočný obsah je nedefinovaný (môže tam byť čokoľvek). Existujú aj výnimky, ale na tie sa nespoliehame.

Projekt Padajúce bodky

Aktivita 1.

Vytvorte program zobrazujúci padajúce červené bodky. Nová bodka sa vytvára kliknutím do kresliacej plochy a jej počiatočný polomer je 6. Ak klikneme na už existujúcu bodku, jej polomer sa zväčší o 2. Každých 200 milisekúnd sa všetky bodky posunú (spadnú) smerom nadol o 4 (pixely). Ak pri páde nadol vyjde bodka z kresliacej plochy, objaví v hornej časti kresliacej plochy.

Zo zadania je na prvý pohľad jasné, že vo formulári budeme potrebovať jednu kresliacu plochu (**TImage**) a jeden časovač (**TTimer**) s periódou tikania 200 milisekúnd. Zo zadania vyplýva, že o každej bodke si potrebujeme pamätať súradnice jej stredu a jej aktuálny polomer – teda celkovo 3 celé čísla. Na uloženie týchto informácií využijeme globálne dvojrozmerné pole **Bodky** typu **n x 3** (n riadkov a 3 stĺpce) a jednu globálnu celočíselnú premennú **PocetBodiek** uchovávajúcu aktuálny počet bodiek „padajúcich“ v kresliacej ploche. Budeme predpokladať, že nikdy viac ako 10000 bodiek nevznikne.




```
var
  Bodky: array[1..10000, 1..3] of Integer;
  PocetBodiek: Integer = 0;
```

Každý riadok poľa **Bodky** uchováva informáciu o jednej bodke: na stĺpcových indexoch 1 a 2 sú to súradnice stredu bodky a na indexe 3 je to jej aktuálny polomer.

Celú prácu na projekte si rozložme do niekoľkých procedúr a funkcií s jasne definovanou funkcionalitou. Je dobrou praxou oddeľovať prácu s dátami a vykresľovanie. Preto aj my si na nakreslenie bodiek vytvoríme samostatné globálne procedúry: **NakresliBodku**, ktorá nakreslí do kresliacej plochy indexom (poradím v poli) určenú bodku, a **NakresliBodky**, ktorá s využitím procedúry **NakresliBodku** nakreslí všetky bodky do kresliacej plochy.

```
procedure NakresliBodku(Index: Integer);
var
  X, Y, R: Integer;
begin
  X := Bodky[Index, 1];
  Y := Bodky[Index, 2];
  R := Bodky[Index, 3];
  Form1.Image1.Canvas.Brush.Color := clRed;
  Form1.Image1.Canvas.Ellipse(X-R, Y-R, X+R, Y+R);
end;

procedure NakresliBodky;
var
  I: Integer;
begin
  Form1.Image1.Canvas.Brush.Color := clWhite;
  Form1.Image1.Canvas.FillRect(Form1.Image1.ClientRect);
  for I := 1 to PocetBodiek do
    NakresliBodku(I);
end;
```

Vykresliť bodky podľa obsahu poľa **Bodky** by sme už vedeli, ostáva nám vytvoriť vhodnú procedúru na pridanie novej bodky do poľa.

Procedúra **PridajBodku** dostáva ako parametre súradnice stredu pridávanej bodky.

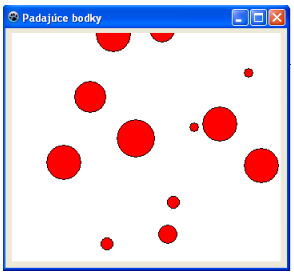
Pripomeňme, že iniciálny polomer novej bodky je 6.

```
procedure PridajBodku(X, Y: Integer);
begin
  Inc(PocetBodiek);
  Bodky[PocetBodiek, 1] := X;
  Bodky[PocetBodiek, 2] := Y;
  Bodky[PocetBodiek, 3] := 6;
end;
```

Posledná vec, ktorá sa nám pri programovaní obsluhy udalostí zíde, je funkcia, ktorá pre zadané súradnice bodu kliknutia v kresliacej ploche vráti poradové číslo (index) tej bodky, na ktorú sa kliklo.

```
function IndexKliknutej(X, Y: Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to PocetBodiek do
    if Sqrt(Sqr(Bodky[I, 1] - X) + Sqr(Bodky[I, 2] - Y))
         <= Bodky[I, 3] then
      Result := I;
end;
```

Funkcia **IndexKliknutej** pracuje tak, že sa pre každú „padajúcu“ bodku v kresliacej ploche pýtame, či sa bod so súradnicami *[X, Y]* nenachádza od stredu tejto bodky menej než je jej polomer. V prípade pozitívnej odpovede si uložíme index tejto bodky - je to bodka, na ktorú sa kliklo. Všimnime si, ako funkcia **IndexKliknutej** vráti informáciu, že sa nekliklo na žiadnu bodku. Iniciálna hodnota premennej **Result** je 0. Ak sa nenájde bodka, v ktorej by skúmaný bod ležal (je blízko od stredu bodky), zmena hodnoty v premennej **Result** sa nikdy nevykoná. Vtedy sa vráti hodnota 0. V opačnom prípade sa do premennej **Result** uloží index bodky, čo je číslo väčšie alebo rovné ako 1, a to sa vráti. Volajúci funkcie **IndexKliknutej** tak podľa vrátenej hodnoty vie ľahko zistiť, či sa kliklo na niektorú z bodiek a ak áno, tak vie na ktorú.



Teraz môžeme pristúpiť k naprogramovaniu obsluhy udalosti zatlačenia tlačidla myši nad kresliacou plochou. Vďaka pripraveným globálnym procedúram a funkciám je to jednoduché: ak sa kliklo na bodku, zväčšíme jej polomer a ak nie, tak pridáme novú bodku. V oboch prípadoch na záver prekreslíme kresliacu plochu na základe aktuálneho stavu poľa **Bodky**.

```
procedure TForm1.Image1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var Index: Integer;
begin
  Index := IndexKliknutej(X, Y);

  if Index = 0 then
    PridajBodku(X, Y)
  else
    Bodky[Index, 3] := Bodky[Index, 3] + 2;

  NakresliBodky;
end;
```

Obsluha udalosti časovača môže vyzerať napríklad takto:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  I: Integer;
begin
    for I := 1 to PocetBodiek do
    if Bodky[I, 2] >= Image1.Height + Bodky[I, 3] then
      Bodky[I, 2] := -Bodky[I, 3]
    else
      Bodky[I, 2] := Bodky[I, 2] + 4;

  NakresliBodky;
end;
```

V tejto procedúre prejdeme for-cyklom všetky riadky prislúchajúce „padajúcim“ bodkám a v každom riadku zvýšime hodnotu v druhom stĺpci (y-ová súradnica) o 4. Ak je táto hodnota príliš veľká (vypadlo sa z dolnej strany), nastaví sa táto hodnota tak, aby sa bodka začala vynárať z hornej strany kresliacej plochy.

Úloha 1.

Upravte program tak, aby pri zväčšení bodky za istú hraničnú veľkosť (napr. polomer 30) bodka zmizla (praskla). Na odstránenie bodky v poli **Bodky** vytvorte a použite procedúru **OdstranBodku**:

```
procedure OdstranBodku(Index: Integer);
```

Variant 1: Riadky idúce za odstraňovaným riadkom v poli **Bodky** posunieme o jeden riadok vyššie a znížime počet bodiek o 1.

Variant 2: Na miesto odstraňovaného riadku skopírujete aktuálne posledný riadok a zníženie počet bodiek o 1. Pozor, týmto spôsobom sa zmení poradie vykresľovania bodiek.

Projekt Teploty

Dvojrozmerné pole môžeme použiť kedykoľvek, keď potrebujeme uchovávať údaje vo forme tabuľky, ktorej jednotlivé políčka (podobné „excelovským bunkám“) sú rovnakého typu. Príkladom takejto tabuľky môže byť aj tabuľka meraní teplôt, pričom každý deň experimentu je teplota meraná ráno, na poludnie a večer, t.j. trikrát denne. V takejto tabuľke každému dňu zodpovedá jeden riadok. Jednotlivým typom merania zodpovedajú stĺpce.

```
var Teploty: array[1..100, 1..3] of Real;
```

Ak nepoznáme dopredu z koľkých dní budeme spracovávať meranie (čo je asi najčastejší prípad), použijeme „fintu“ z predchádzajúcich príkladov. Vytvoríme si pole s dostatočne veľkým počtom riadkov a v ďalšej celočíselnej premennej si budeme pamätať, koľko riadkov (dní) máme v poli skutočne uložených.

Aktivita 2.

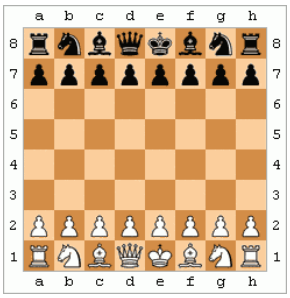
Pozrite si pripravený projekt **Teploty**. Program pri spustení načítava tabuľku meraní z textového súboru do dvojrozmerného poľa. Na základe obsahu poľa sa potom zobrazí stĺpcový graf meraní teplôt.

Doprogramujte v programe funkciu **PriemernaTeplota,** ktorá pre zadaný deň (zadaný indexom riadka, ktorý prislúcha danému dňu) vráti priemernú teplotu v tento deň.

Riešenie

```
function PriemernaTeplota(Idx: Integer): Real;
var
  I: Integer;
begin
  Result := 0;
  for I := 1 to 3 do
    Result := Result + Teploty[Idx, I];

  Result := Result / 3;
end;
```

4. Dvojrozmerné pole ako hracia plocha



V moduloch Programovanie 1-9 sme maľovali do kresliacej plochy kadejaké šachovnice a mriežky. Pri kliknutí do plochy sme dokresľovali aj rôzne farebné krúžky, či iné geometrické útvary. Odtiaľ je len malý krôčik k naprogramovaniu jednoduchej doskovej hry (piškvorky, reverzy, klobásky, míny, ...), či dokonca aj hry ako je skákaná alebo dáma. Nakresliť hraciu plochu i hracie kamene (figúrky) už dokážeme. Avšak skôr či neskôr by sme natrafili na situáciu, kedy by sme potrebovali rozhodnúť, či nejaký ťah je povolený (v prípade piškvoriek, či políčko je voľné), alebo či aktuálne rozloženie hracích kameňov/figúrok znamená výhru pre niektorého z hráčov. Na to ale potrebujeme vedieť, ktoré políčka hracej dosky sú obsadené akými figúrkami. Riešením je uchovávať si stav hracej dosky niekde v pamäti - v premenných. Keďže typické hracie pole doskovej hry je štvorcová mriežka, **dvojrozmerné pole typu n x n je ideálna voľba.**

Projekt Piškvorky 3 x 3

Naprogramovanie jednoduchej doskovej hry nie je až také náročné, ako by sa mohlo na prvý pohľad zdať. No prv, než sa pustíme do jej programovania, si musíme ujasniť, ako chceme uchovávať stav hry. V prípade doskových hier na to zvyčajne využijeme dvojrozmerné pole, ktoré má presne také rozmery ako hracia doska. Akého typu však majú byť prvky tohto poľa? To záleží od konkrétnej hry. Pozrime sa bližšie na hru piškvorky s hracou plochou 3 x 3. Je to hra pre 2 hráčov. Ľubovoľné políčko hracej plochy môže byť buď prázdne alebo obsadené hracím kameňom jedného z dvoch hráčov. Celkovo tak dostávame, že políčko môže byť v jednom z troch stavov:

- prázdne políčko, ktoré budeme označovať číslom 0,
- políčko obsadené hráčom 1, ktoré budeme označovať číslom 1,
- políčko obsadené hráčom 2, ktoré budeme označovať číslom 2.

Vhodným typom pre prvky poľa je teda ľubovoľný celočíselný typ. Stav hracej plochy si môžeme uložiť v takto deklarovanom dvojrozmernom poli:

```
var HraciaPlocha: array[1..3, 1..3] of Integer;
```

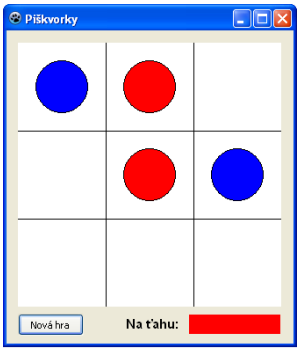
Stav hracej plochy (obsadenosť políčok hracími kameňmi) však nepopisuje stav hry úplne. Musíme si ešte pamätať, ktorý hráč je na ťahu. Túto informáciu si môžeme uložiť v globálnej celočíselnej premennej uchovávajúcej číslo hráča (hodnota 1 alebo 2).

```
var NaTahu: Integer = 1;
```

Niekedy sa nám tiež hodí informácia o tom, či hra ešte stále beží, t.j. či sa nedosiahol stav hry, v ktorom je už určený víťaz. Túto informáciu môžeme uložiť v globálnej premennej typu **Boolean** (hra beží: **true**/**false**). Ďalšou možnosťou je využiť premennú **NaTahu** a v prípade, že hra už nebeží, uložiť do nej špeciálnu hodnotu, ktorá nás bude o tomto informovať (napr. číslo 0).

Aktivita 1.

Pozrite si projekt **Piskvorky** - jednoduché piškvorky pre dvoch hráčov na hracej ploche 3 x 3.



Ako je v poli uchovaná informácia o obsadenosti políčka hracej plochy? Ako sa zisťuje v obsluhe udalosti **OnMouseDown**, na ktoré políčko sme klikli? Ako je vyriešené, aby sa nemohlo kliknúť na už obsadené políčko? Do akých procedúr a funkcií je rozložená funkcionalita programu? Čo tieto podprogramy robia?

Upravte funkciu **OverVyhru** tak, aby vrátila **true** práve vtedy, keď aktuálny stav hry (rozloženie kameňov) je výherné pre jedného z hráčov. V prípade, že aktuálne rozloženie kameňov znamená remízu, funkcia nech vráti **false**.

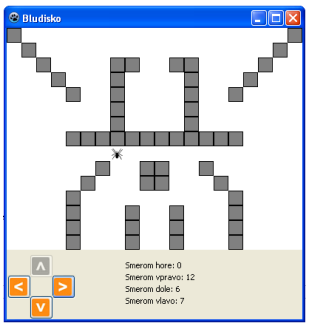
Na to, aby sme overili výhernosť rozmiestnenia kameňov v hracej ploche, potrebujeme overiť, že v aspoň jednom riadku alebo v aspoň jednom stĺpci alebo aspoň na jednej z dvoch uhlopriečok sú len kamene rovnakej farby. V našej hre piškvorky 3 x 3 máme celkovo 8 trojíc políčok, z ktorých každú treba skontrolovať. Pozrime sa, ako by vyzerala podmienka, ktorá overí, či i-ty riadok hracej plochy je výherný pre niektorého z hráčov, t.j. všetky políčka tohto riadka sú nenulové a majú rovnakú hodnotu.

```
(HraciaPlocha[I, 1] <> 0) and
(HraciaPlocha[I, 1] = HraciaPlocha[I, 2]) and
(HraciaPlocha[I, 2] = HraciaPlocha[I, 3])
```

Podmienka sa skladá z (konjunkcie) 3 testov na rovnosť, resp. nerovnosť prvkov poľa. Prvý test overuje, či prvý prvok v riadku je nenulový, t.j. či je obsadený nejakým kameňom. Nasledujúce dva testy overujú, či prvý a druhý prvok v riadku, resp. druhý a tretí prvok v riadku sú rovnaké. Tieto dva testy vyplývajú z toho, že ak chceme overiť, či tri premenné obsahujú rovnakú hodnotu, stačí overiť, či sú po dvojiciach rovnaké. Matematicky: $a = b = c \Leftrightarrow a = b \wedge b = c$. Prvý test v podmienke je veľmi dôležitý. Ak by tam nebol, podmienka by bola splnená aj v prípade, kedy by všetky prvky v riadku boli nulové, t.j. políčka v hracej ploche by boli neobsadené. Podobným spôsobom môžeme overiť aj ďalšie zo „skúmaných“ trojíc prvkov poľa:

```
function OverVyhru: Boolean;
var
  I, J: Integer;
begin
  Result := false;

  for I := 1 to 3 do
    if (HraciaPlocha[I, 1] <> 0) and
       (HraciaPlocha[I, 1] = HraciaPlocha[I, 2]) and
       (HraciaPlocha[I, 2] = HraciaPlocha[I, 3]) then
      Result := true;

  for I := 1 to 3 do
    if (HraciaPlocha[1, I] <> 0) and
       (HraciaPlocha[1, I] = HraciaPlocha[2, I]) and
       (HraciaPlocha[2, I] = HraciaPlocha[3, I]) then
      Result := true;

  if (HraciaPlocha[2, 2] <> 0) and
     (HraciaPlocha[1, 1] = HraciaPlocha[2, 2]) and
     (HraciaPlocha[2, 2] = HraciaPlocha[3, 3]) then
    Result := true;

  if (HraciaPlocha[2, 2] <> 0) and
     (HraciaPlocha[1, 3] = HraciaPlocha[2, 2]) and
     (HraciaPlocha[2, 2] = HraciaPlocha[3, 1]) then
    Result := true;
end;
```

Projekt Bludisko



Dvojrozmerné pole môžeme využiť aj na uchovanie mapy bludiska. Bludisko sa v tomto prípade chápe ako obdĺžniková mriežka, v ktorej na jednotlivých políčkach je alebo nič (voľné políčko), alebo je tam prekážka. V bludisku máme panáčika, ktorý sa v ňom môže pohybovať. Každý krok panáčika môže byť v jednom zo štyroch smerov: nahor, vpravo, nadol a vľavo. Panáčik sa v bludisku samozrejme môže nachádzať len na voľných políčkach (cez stenu prejsť nedokáže), resp. vie vstúpiť len na voľné políčka. V pripravenom projekte **Bludisko** panáčik v tvare pavúka po bludisku dokáže len bezcieľne blúdiť. Isto si ale vieme predstaviť aj zaujímavejšiu aktivitu: hľadanie východu z bludiska, či zbieranie predmetov v bludisku. A to všetko trebárs doplnené potvorkami, ktoré ho naháňajú a s ktorými sa nesmie stretnúť na spoločnom políčku (na jednom mieste).

Aktivita 2.

Prezrite si projekt **Bludisko** simulujúci panáčika v bludisku. Aké procedúry a funkcie sú v ňom využité? Všimnite si, akým spôsobom si panáčik pamätá smer, v ktorom je natočený.

Pozrime sa, ako je uložené bludisko a ako stav (pozícia a natočenie) panáčika.

```
var
  Bludisko: array[1..15, 1..20] of Boolean;
  PanacikR, PanacikS: Integer;
  Smer: Integer;
```

Na uloženie bludiska používame dvojrozmerné pole **Bludisko** prvkov logického typu. Rozmery bludiska sú 15 riadkov krát 20 stĺpcov. Logická hodnota uchováva, či je dané políčko obsadené prekážkou (**true**) alebo je voľné (**false**). Aktuálnu pozíciu

panáčika uchovávame v celočíselných premenných **PanacikR** (index riadka políčka, na ktorom sa nachádza) a **PanacikS** (index stĺpca políčka, na ktorom sa nachádza). Posledná dôležitá premenná je premenná **Smer**, v ktorej uchovávame aktuálne natočenie panáčika (kvôli vykresľovaniu). Táto premenná môže nadobúdať jednu zo štyroch hodnôt: 1-sever/nahor, 2-východ/vpravo, 3-juh/nadol, 4-západ/vpravo.

Aktivita 3.

V projekte **Bludisko** implementujte funkciu **DaSaPohnut**, ktorá pre zadaný smer (číslo od 1 po 4) vráti, či sa panáčik vie pohnúť daným smerom. Panáčik sa vie pohnúť daným smerom, ak sa cieľové políčko kroku nachádza v hracej ploche a zároveň neobsahuje žiadnu prekážku (je voľné).

```
function DaSaPohnut(Smerom: Integer):Boolean;
```

Určite rýchlo prídeme na riešenie zadania. Najprv overíme, či susedné políčko v zadanom smere vôbec existuje, t.j. či jeho prvý (riadkový) index je v rozsahu od 1 po 15 a druhý (stĺpcový) index v rozsahu od 1 po 20. Ak políčko existuje, overíme, či je voľné. Na zistenie súradníc susedného políčka by sme asi použili 4 podmienené príkazy **if**, prípadne príkaz **case**. Každý z týchto štyroch podmienených príkazov (pre každý možný smer pohybu jeden) by mohol vyzerať nejako takto:

```
if Smerom = 1 then
begin
  SusednyR := PanacikR - 1;
  SusednyS := PanacikS;
end;
```

Ukážme si teraz iný spôsob ako vyriešiť úlohu zo zadania. To, čo potrebujeme spraviť hneď na začiatku, je nejako vypočítať indexy (súradnice) susedného políčka, na ktoré by sme sa presunuli, ak by sme spravili krok v zadanom smere.

[R - 1, S - 1]	[R - 1, S + 0]	[R - 1, S + 1]
[R + 0, S - 1]	[R + 0, S + 0]	[R + 0, S + 1]
[R + 1, S - 1]	[R + 1, S + 0]	[R + 1, S + 1]

Všimnime si, že pri posunutí smerom nahor sa druhý (stĺpcový) index nemení. Nový stĺpcový index sme akoby dostali tak, že sme k aktuálnemu indexu prirátali číslo 0. Pri posune nahor sa mení len prvý (riadkový) index a to tak, že sa jeho hodnota zníži o jedna. Na túto zmenu sa môžeme pozrieť aj tak, že sme k aktuálnej hodnote riadkového indexu prirátali číslo -1. Posun smerom nahor teda vieme charakterizovať dvojicou čísel *[R: -1, S:0]*. Táto dvojica je akýmsi „vektorom posunutia“ - zmeny indexov. Podobnými dvojicami čísel vieme charakterizovať aj posun zvyšnými smermi: vpravo dvojicou *[R: 0, S: 1]*, nadol dvojicou *[R: 1, S: 0]* a vľavo dvojicou *[R: 0, S: -1]*. Uvedomme si, že ak sa panáčik nachádza na políčku *[R, S]*, po posune charakterizovanom dvojicou *[ΔR, ΔS]* sa bude nachádzať na políčku *[R + ΔR, S + ΔS]*. Jednotlivé dvojice charakterizujúce posunutia si môžeme uložiť v globálnom inicializovanom poli.

```
var Posuny: array[1..4, 1..2] of Integer =
                          ((-1, 0), (0, 1), (1, 0), (0, -1));
```

Pole **Posuny** je dvojrozmerné pole na uloženie tabuľky so 4 riadkami a 2 stĺpcami. Každý riadok tejto tabuľky obsahuje dvojicu charakterizujúcu posunutie v jednom zo smerov. Konkrétne, posunutie v smere **S** je charakterizované zmenou riadkového indexu **Posuny[S][1]** a zmenou stĺpcového indexu **Posuny[S][2]**. S využitím tohto postupu môžeme elegantne zapísať funkciu **DaSaPohnut** takto:

	ΔR	*ΔS*
	1	*2*
⇧ *1*	-1	0
⇨ *2*	0	1
⇩ *3*	1	0
⇦ *4*	0	-1

```
function DaSaPohnut(Smerom: Integer): Boolean;
var
  SusednyR, SusednyS: Integer;
begin
  SusednyR := PanacikR + Posuny[Smerom][1];
  SusednyS := PanacikS + Posuny[Smerom][2];

  if (SusednyS <= 0) or (SusednyS >= 21) or
       (SusednyR <= 0) or (SusednyR >= 16) then
    Result := false
  else
    Result := (Bludisko[SusednyR, SusednyS] = false);
end;
```

Pozrime sa na ešte krajšie využitie „vektorov posunutia“ pre jednotlivé smery.

Aktivita 4.

Naprogramujte funkciu **NajviacVSmere**, ktorá pre zadaný smer (číslo od 1 po 4) vráti, koľko najviac krokov vie panáčik spraviť v zadanom smere.

Keďže asi často budeme musieť v rôznych situáciách testovať, či nejaké indexy sú platnými indexmi niektorého z políčok bludiska, vytvorme si funkciu **PlatnePolicko**, ktorá nám takýto test zrealizuje.

```
function PlatnePolicko(R, S: Integer): Boolean;
begin
  Result := (R >= 1) and (R <= 15) and (S >= 1) and (S <= 20);
end;
```

A teraz sa pozrime, ako by mohla vyzerať implementácie funkcie **NajviacVSmere.**

```
function NajviacVSmere(Smerom: Integer): Integer;
var
  PozR, PozS: Integer;
begin
  PozR := PanacikR;
  PozS := PanacikS;

  Result := 0;
  while PlatnePolicko(PozR, PozS) and
          (Bludisko[PozR, PozS] = false) do
  begin
    PozR := PozR + Posuny[Smerom][1];
    PozS := PozS + Posuny[Smerom][2];
    Inc(Result);
  end;
  Dec(Result);
end;
```

Základná myšlienka implementácie funkcie **NajviacVSmere** spočíva v postupnom simulovaní krokov panáčika v zadanom smere. Používame dve pomocné premenné **PozR** a **PozS**, ktoré uchovávajú aktuálnu pozíciu simulovaného panáčika. Panáčika posúvame v zadanom smere dovtedy, kým sa nachádzame na políčku, ktoré je v bludisku a zároveň je voľné (nie je na ňom prekážka). Posunutie panáčika je vykonané v každej iterácii **while** cyklu. Okrem toho premennú **Result** používame ako počítadlo počtu vykonaných krokov. Uvedomme si, že **while** cyklus sa skončí práve vtedy, keď sa dostaneme mimo plochu bludiska alebo na obsadené políčko. Avšak posledný krok vedúci na takéto „zlé“ políčko ostane už zarátaný v premennej **Result**. Preto po skončení **while** cyklu znížime premennú **Result** o 1, aby sme tento posledný krok nezapočítavali.

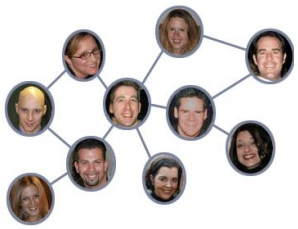
Hlavná výhoda využitia poľa (tabuľky) s „vektormi posunutia“ pre jednotlivé typy

pohybov spočíva v ušetrení potenciálne veľkého počtu podmienených príkazov. Pri bludisku to boli 4 smery. V rovnakom počte smerov (vodorovne, zvislo, šikmo v smere uhlopriečky doľava a šikmo v smere uhlopriečky doprava) musíme prehľadávať hraciu plochu pri hre piškvorky **n x n** na to, aby sme zistili, či aktuálne rozloženie kameňov je výherné. Pri riešení osemsmerovky však už musíme uvažovať všetkých 8 smerov. Dokonca aj pohyby figúrok v hre šach je možné vyjadriť pomocou poľa s „vektormi posunutia“.

5. Dvojrozmerné pole ako relácia

V praktických počítačových aplikáciách neraz potrebujeme zachytiť vzťah medzi objektmi reálneho sveta. Napríklad v sociálnych sieťach (Facebook, LinkedIn, ...) je nutné uchovať informáciu o tom, kto je s kým kamarát (napr. že Janko je kamarát s Jožkom). V GPS navigátore či elektronickom mýtnom systéme o tom, aká je dĺžka priamej cesty medzi mestami (napr. že z Bratislavy do Trnavy je priama cesta dlhá 50 km). V informačnom systéme vysokej školy potrebujeme zachytiť informácie o prerekvizitách (napr. modul Programovanie 9 musí byť absolvovaný pred zapísaním modulu Algoritmy a údajové štruktúry 1). Aj v tomto prípade sú dvojrozmerné polia typu **n x n** veľmi nápomocné, pretože nám dovoľujú veľmi jednoducho zachytiť rôzne vzťahy medzi objektmi (matematici tomu hovoria relácie).

Projekt Sociálna sieť



Sociálne siete slúžia na udržiavanie kontaktov a väzieb medzi používateľmi siete. Aby používateľ sociálnej siete v nej nebol osamotený, udržiava si v nej zoznam svojich kamarátov – iných používateľov tej istej sociálnej siete. Byť kamarátom je obojstranný vzťah medzi používateľmi. To znamená, že ak používateľ Janko je kamarátom používateľa Jožko, tak aj Jožko je kamarátom Janka. Môžeme povedať, že Jožko je v kamarátskom vzťahu s Jankom. Je jasné, že webová aplikácia sociálnej siete si nejako musí uchovávať informáciu o tom, kto je s kým kamarát. Ak sa zamyslíme, určite nájdeme viacero spôsobov a údajových štruktúr, ktoré by boli pre takýto účel vyhovujúce. My si teraz predstavíme najjednoduchší, no zároveň pamäťovo (a niekedy aj časovo) najmenej efektívny spôsob. Predpokladajme, že v sociálnej sieti máme zaregistrovaných **n** používateľov. Ak si vyberieme ľubovoľných dvoch používateľov siete, mali by sme vedieť povedať, či sú kamaráti – t.j. či existuje medzi nimi kamarátsky vzťah. Kamarátsky vzťah medzi dvoma používateľmi buď existuje alebo neexistuje – je to teda logická hodnota, ktorú vieme uložiť v premennej typu **Boolean**. S využitím dvojrozmerného poľa s políčkami typu **Boolean** a s rozmermi **n x n** vieme uložiť údaje o kamarátstvach tak, že políčko v i-tom riadku a j-tom stĺpci bude hovoriť, či medzi i-tym a j-tym používateľom je (alebo nie je) kamarátsky vzťah.

```
var Kamaratstvo: array[1..100, 1..100] of Boolean;
```

Teda, ak by sme na políčku **Kamaratstvo[4, 10]** našli hodnotu **true**, tak vieme, že štvrtý používateľ a desiaty používateľ systému sú kamaráti. Ak by sme našli hodnotu **false**, vieme že kamaráti nie sú. Možno sa nám nepáči, že hovoríme o štvrtom používateľovi a nie o Jožkovi. Opäť kvôli jednoduchosti je lepšie stotožniť používateľov s číslami. Môžu to byť trebárs poradové čísla podľa poradia registrácie v sociálnej sieti. Navyše typ **String** nie je ordinálnym typom a tak ho ani nemôžeme použiť ako index v poliach. Ak by sme chceli priradiť k číslam mená, môžeme to spraviť pomocným poľom **Pouzivatelia**, ktoré by uchovávalo mená používateľov v nejakom poradí. Čiže v prvku **Pouzivatelia[4]** nájdeme meno štvrtého používateľa.

Skúste nakresliť, ako by sa vzťahy o kamarátstvach z vyššie načrtnutej sociálnej siete uložili v dvojrozmernom poli **Kamaratstvo**.

```
var Pouzivatelia: array[1..100] of String;
```

V prípade, že počet používateľov sociálnej siete sa môže meniť (nie je dopredu známy), použijeme už starú známu fintu s celočíselnou premennou uchovávajúcou skutočný počet používateľov, resp. počet prvkov v poli s platným obsahom.

```
var PocetOsob: Integer = 0;
```

Aktivita 1.

Pozrite si projekt **Kamarati**. Skúste rozanalyzovať, ako pracuje procedúra na načítanie údajov o kamarátskych vzťahoch z textového súboru. Aký je formát vstupných údajov v textovom súbore?

Keďže kamarátstva v sociálnej sieti sú vždy obojstranné, v našom dvojrozmernom poli **Kamaratstvo** musí pre ľubovoľné indexy **I** a **J** platiť, že **Kamaratstvo[I, J] = Kamaratstvo[J, I]**. Na to si musíme dávať pozor hlavne pri pridávaní kamarátstiev a pri načítavaní údajov o kamarátstvach. Pole **Kamaratstvo** musíme vždy modifikovať tak, aby táto vlastnosť obojstrannosti ostala zachovaná (viď procedúra **PridajKamaratstvo**, ktorá do poľa uloží, že medzi osobami je kamarátsky vzťah). Matematici takéto obojstranné vzťahy nazývajú symetrické relácie.

```
procedure PridajKamaratstvo(Osoba1, Osoba2: Integer);
begin
  if (Osoba1 >= 1) and (Osoba1 <= PocetOsob) and
       (Osoba2 >= 1) and (Osoba2 <= PocetOsob) then
  begin
    Kamaratstvo[Osoba1, Osoba2] := true;
    Kamaratstvo[Osoba2, Osoba1] := true;
  end;
end;
```

Aktivita 2.

Naprogramujte funkciu **PocetKamaratov**, ktorá vráti počet kamarátov zadanej osoby. Naprogramujte aj funkciu **PocetSpoluKamaratov** tak, aby vrátila počet osôb, ktoré sa kamarátia s oboma zadanými osobami.

Pozrime sa, ako by mohli tieto procedúry vyzerať. V oboch prípadoch myšlienka riešenia spočíva v tom, že „skúšame“ všetkých používateľov (for-cyklus) a pre každého z nich sa pýtame, či spĺňa skúmanú vlastnosť (je kamarátom jedného, resp. oboch zadaných používateľov). Ak používateľ spĺňa skúmanú vlastnosť, zvýšime počítadlo nájdených používateľov (premennú **Result**).

```
function PocetKamaratov(Osoba: Integer): Integer;
var
  I: Integer;
begin
  if (Osoba <= 0) or (Osoba > PocetOsob) then
  begin
    Result := -1;
    Exit;
  end;

  Result := 0;
  for I := 1 to PocetOsob do
    if Kamaratstvo[Osoba, I] then
      Inc(Result);
end;

function PocetSpoluKamaratov(Osoba1,Osoba2: Integer): Integer;
var
  I: Integer;
begin
  if (Osoba1 <= 0) or (Osoba1 > PocetOsob) or
       (Osoba2 <= 0) or (Osoba2 > PocetOsob) then
  begin
```

```
    Result := -1;
    Exit;
  end;

  Result := 0;
  for I := 1 to PocetOsob do
    if Kamaratstvo[Osoba1,I] and Kamaratstvo[I,Osoba2] then
       Inc(Result);
end;
```

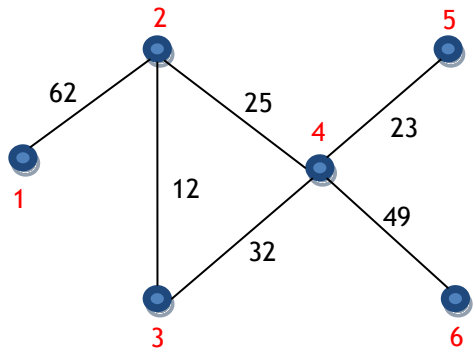
Projekt Cestná sieť

Vzťah byť kamarátom, tak ako sme ho použili v predchádzajúcich príkladoch, nie je ohodnotený - nehodnotíme kvalitu vzťahu. Niekedy však potrebujeme zachytiť aj kvalitu vzťahu - relácie. Napríklad informácia o tom, že medzi mestom **A** a mestom **B** existuje priama cesta, nemusí v konkrétnej reálnej situácii stačiť. Zvyčajne nás pri takejto ceste zaujíma aj jej dĺžka, či to, za aký čas ju vieme prejsť. Kým v prechádzajúcom príklade nám na charakterizovanie vzťahu medzi osobami stačilo políčko typu **Boolean**, na vyjadrenie kvality vzťahu už potrebujeme použiť v dvojrozmernom poli políčka typu **Integer** alebo **Real.** V nasledujúcich príkladoch sme sa rozhodli pre celočíselný typ **Integer**, t.j. naše priame cesty môžu mať len celočíselnú dĺžku. Podobne, ako v projekte o sociálnej sieti, budeme namiesto názvov miest používať len ich číselné označenia. Teda naša cestná sieť bude tvorená mestami 1, 2, 3,… Deklarácia poľa na uloženie údajov o priamych cestných spojeniach (o cestnej sieti) môže vyzerať napríklad takto:

```
const MaxMiest = 100;
var Cesty: array[1..MaxMiest, 1..MaxMiest] of Integer;
```

Políčko **Cesty[I, J]** v dvojrozmernom poli **Cesty** obsahuje dĺžku priamej cesty medzi mestami **I** a **J.** Keďže cesty v našej sieti sú obojsmerné, pre hodnoty uložené v poli **Cesty** musí platiť, že **Cesty[I, J] = Cesty[J, I]**. To, že **Cesty[I, J]** obsahuje dĺžku priamej cesty medzi **I** a **J** už vieme. Ale čo uložíme do **Cesty[I, J],** ak medzi mestami **I** a **J** neexistuje priama cesta? Jedna možnosť je vytvoriť si ďalšie dvojrozmerné pole logických hodnôt, v ktorom by sme uchovali informáciu, či medzi mestami existuje priama cesta. Druhá, pamäťovo efektívnejšia možnosť je využiť hodnotu, o ktorej vieme, že nikdy nemôže byť dĺžkou cesty. Táto špeciálna hodnota by bola indikátorom toho, že priama cesta medzi zadanými dvoma mestami neexistuje. Dobrá voľba pre takúto špeciálnu hodnotu je napríklad číslo -1. Asi cesty zápornej dĺžky v našej cestnej sieti nemáme a ani nebudeme mať.

	1	2	3	4	5	6
1	-1	62	-1	-1	-1	-1
2	62	-1	12	25	-1	-1
3	-1	12	-1	32	-1	-1
4	-1	25	32	-1	23	49
5	-1	-1	-1	23	-1	-1
6	-1	-1	-1	49	-1	-1



Matematickým modelom (zovšeobecnením) pre cestné siete sú tzv. **grafy**. Grafy sa skladajú z **vrcholov** (mestá) a **hrán** (priame cesty). Dĺžka priamej cesty medzi mestami sa v grafovej terminológii nazýva cena hrany alebo **ohodnotenie hrany**. Grafy sú intenzívne skúmané v matematike aj informatike. Táto oblasť sa nazýva **teória grafov**.

Dĺžka cestného spojenia určeného postupnosťou *1, 2, 4, 3, (-1)* z predchádzajúceho príkladu je: ***62*** *(1, 2)* + ***25*** *(2, 4)* + + ***32*** *(4, 3)* = ***119***

Aktivita 3.

Pozrite si projekt **Cesty**, ktorý z textového súboru načíta informácie o priamych spojeniach medzi mestami. Všimnite si, ako je v poli zachytená informácia o tom, že medzi mestami nie je cesta, či to, ako sa načítava cestná sieť z textového súboru. Pozrite si funkciu **NajblizsieMesta**, ktorá nájde najkratšiu priamu cestu v cestnej sieti.

Naprogramujte funkciu **DlzkaCesty** tak, aby vrátila dĺžku cesty, ktorá prechádza zadanými mestami. Ak taká cesta neexistuje, funkcia nech vráti *-1*. Cesta je určená postupnosťou miest v parametri typu pole ako postupnosť čísel ukončená číslom -1.

Parametrom funkcie **DlzkaCesty** je postupnosť čísel v poli. My už vieme, že ak pole má byť parametrom procedúry alebo funkcie, najprv si definujeme vlastný „poľový“ typ a ten využijeme na označenie typu parametra funkcie.

```
type TCesta = array[1..MaxMiest+1] of Integer;
```

To, čo máme podľa zadania zistiť, je súčasne overiť existenciu nepriameho cestného spojenia a spočítať jeho dĺžku. Základný postup je jednoduchý. V postupnosti cestného spojenia budeme postupne vyberať dvojice za sebou idúcich miest (čísel). Pre každú vybranú dvojicu miest overíme, či medzi nimi existuje priama cesta. Ak neexistuje, okamžite ukončíme výpočet vrátením hodnoty -1. V opačnom prípade pripočítame dĺžku priameho spojenia do premennej **Result**, v ktorej si budeme pamätať, akú dĺžku sme doposiaľ prešli po priamych cestách medzi mestami.

```
function DlzkaCesty(Cesta: TCesta): Integer;
var
  I: Integer;
begin
  Result := 0;
  I := 1;
  while Cesta[I] <> -1 do
  begin
    if I >= 2 then
    begin
      if Cesty[Cesta[I-1], Cesta[I]] >= 0 then
        Result := Result + Cesty[Cesta[I-1], Cesta[I]]
      else
      begin
        Result := -1;
        Exit;
      end;
    end;

    Inc(I);
  end;
end;
```

Cestné siete a kamarátske vzťahy boli v oboch príkladoch obojsmerné – symetrické. Dvojrozmerné polia však vieme využiť aj na zachytenie nesymetrických vzťahov medzi objektmi. Ak by **Cesta[I, J]** vyjadrovala čas na prejdenie priamej cesty z mesta **I** do **J**, tak hodnota **Cesta[J, I]** sa môže líšiť od **Cesta[I, J]** (napr. z mesta **I** do mesta **J** sa ide do kopca a tak cesta trvá dlhšie než cesta z **J** do **I**, ktorá ide z kopca). V prípade nesymetrických relácií je preto veľmi dôležité, ktorý index zadáme ako prvý.

Čo sme sa naučili

Prvky poľa môžu byť umiestnené nie len v jednom riadku za sebou (jednorozmerné pole), ale aj vo forme obdĺžnikovej tabuľky (dvojrozmerné pole). Dvojrozmerné pole ako ďalší zo štruktúrovaných typov ponúka rôzne možnosti použitia: uloženie tabuliek, uloženie stavu hracej dosky, či uloženie vzťahov (relácií) medzi objektmi. Naučili sme sa aj základné algoritmy a „finty“, ktoré sa využívajú pri práci s dvojrozmerným poľom: prechod prvkami poľa, použitie „vektorov“ posunutia, či zakódovanie neexistencie vzťahu medzi objektmi pomocou špeciálnej hodnoty.

Úlohy na precvičenie

Zadanie 1.	Navrhnite, ako by ste uložili výsledky písomiek pre ďalšie spracovanie, ak viete, že každá písomka sa skladá z rovnakého počtu úloh. Navrhnite a naprogramujte rôzne funkcie počítajúce štatistické informácie o písomkách (najhoršia písomka, najlepšia písomka, úloha s najlepším priemerným výsledkom,…). Ako by ste ku každej písomke priradili aj meno žiaka, ktorý ju písal?
Zadanie 2.	Naprogramujte procedúru, ktorá vygeneruje iniciálne rozloženie hracích kameňov pre hru dáma. Naprogramujte aj procedúru, ktorá aktuálne rozloženie kameňov v hracej ploche (v dvojrozmernom poli) vykreslí do kresliacej plochy.
Zadanie 3.	Pridajte do hry piškvorky automatického protihráča, ktorý náhodne obsadí nejaké voľné políčko. Porozmýšľajte o lepších stratégiách s cieľom naprogramovať inteligentného protihráča.
Zadanie 4.	Naprogramujte funkciu, ktorá overí, či dvojrozmerné pole celých čísel s rozmermi *9 x 9* je korektným riešením sudoku.
Zadanie 5.	Do programu o cestnej sieti doprogramujte funkciu, ktorá overí, či cestná sieť spĺňa tzv. trojuholníkovú nerovnosť: pre akúkoľvek trojicu miest **A, B** a **C**, ktoré sú navzájom prepojené priamymi cestami platí, že dĺžka priamej cesty z **A** do **B** je menšia alebo rovná než dĺžka cesty z **A** do **B** cez mesto **C**.
Zadanie 6.	Vytvorte program, ktorý z textového súboru načíta zadanie osemsmerovky (mriežka s písmenami a hľadané slová) a vyrieši ho.

Zoznam

V rámci vzdelávania ste sa už stretli so zoznamom v module Programovanie 0 – programovanie v Imagine. V Imagine je zoznam základná štruktúra údajov. Z modulov Programovanie 1 až 9 poznáte jednorozmerné pole. V tejto časti porovnáme obe štruktúry údajov.

Štruktúra údajov zoznam je ľubovoľná postupnosť prvkov, pričom vieme pridávať a odoberať prvky za alebo pred určeným prvkom. Príklad číselného zoznamu:

Tie cifry som už niekde videl....

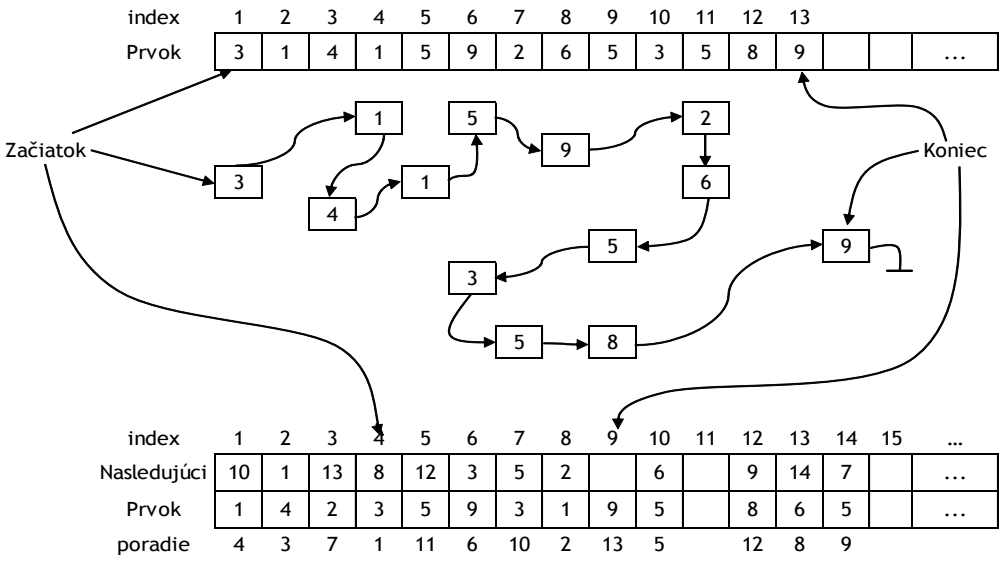
```
3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5
```

Úloha 1.

Vykonajte s predchádzajúcim zoznamom nasledujúce operácie:
a) za desiaty prvok vložte prvok 10
b) pred prvý prvok vložte prvok 0
c) na koniec vložte prvok -1
d) vyhoďte tridsiaty tretí prvok
Pokúste sa použiť čo najviac spôsobov reprezentácie zoznamu z obr.1

Na obr. 1 sú znázornené tri reprezentácie tej istej štruktúry údajov – trinásť prvkového zoznamu, pomocou jednorozmerného poľa, dvojrozmerného poľa a spájaným zoznamom.

Niekedy nás na úrade pošlú z kancelárie č. K do kancelárie č. L a odtiaľ do kancelárie č. M, atď. Musíme vlastne prejsť zoznam kancelárií. Vopred nevieme v akom poradí kancelárie budeme navštevovať. Vieme iba , ktorá je prvá a kam máme ísť ďalej sa dozvieme vždy až v práve navštívenej kancelárii.



Obr. 1: Znázornenie reprezentovania zoznamu (zhora nadol) jednorozmerným poľom, spájaným zoznamom a dvojrozmerným poľom.

Aktivita 1.

Diskutujte o tom, v čom sa na obr. 1 štruktúry údajov od seba odlišujú. Skúste sa zamyslieť na dôsledkami týchto odlišností.

Ak ste vykonali 1. úlohu a 1. aktivitu, pravdepodobne ste prišli na to, že spôsob reprezentácie zoznamu priamo vplýva na to, ako „šikovne“ vieme jednotlivé operácie realizovať. Skôr než sa budeme podrobnejšie venovať skúmaniu ako reprezentácia ovplyvňuje realizáciu pozrime sa kde sa zoznam v praxi vyskytuje.

Príklad 1.

CD-čka alebo DVD-čka na poličke tvoria zoznam. Keď chceme pridať nové, môžeme ho dať na začiatok, na koniec, alebo ho vsunúť medzi niektoré dve (za alebo pred niektoré).

Príklad 2.	Okná na ploche sú organizované ako zoznam. Keď na ploche už máme nejaké okná a vytvoríme nové okno, zvyčajne sa pridá pred posledné aktívne okno. Môžeme postupne prechádzať po jednotlivých oknách a vybrať, ktoré bude aktívne.

Poradie okien si môžeme pozrieť stlačením ALT+TAB, keď podržíme ALT, opätovné stlačenie TAB nás posúva na ďalšie okno.

Príklad 3.	Súbory na disku sú uložené ako zoznamy. Súbory nie sú uložené v jednom súvislom úseku, ale zvyčajne vo viacerých menších úsekoch pevnej veľkosti. Je to vymyslené tak, že vieme, kde sa nachádza prvý úsek. Každý úsek vie, kde sa na disku nachádza ďalší úsek. Táto stratégia umožňuje rozumne využiť priestor na disku. Čítanie z disku je oproti čítaniu z pamäte asi 10^6 pomalšie, preto je výhodné čítať z disku väčšie úseky naraz. Keďže nevieme odhadnúť dopredu veľkosti súborov, je dobre ich rozdeliť na menšie úseky rovnakej veľkosti, ktoré sa čítajú celé naraz.

Aha! Tá čudná akcia „defragmentácia disku“ vlastne prehadzuje úseky, v ktorom je súbor uložený tak, aby sa dal súbor rýchlo čítať.

Na obr. 1 sme zobrazili niekoľko spôsobov ako sa dá zoznam realizovať. Na prvý pohľad sa to môže zdať ako zbytočné, ale v skutočnosti to je celkom bežná situácia, ktorú musíme riešiť, vždy keď chceme niečo naprogramovať a potrebujeme pri tom nejakú zložitejšiu štruktúru údajov. Pri rozhodovaní sa akú štruktúru vybrať a ako ju realizovať by sme mali zobrať do úvahy, aké údaje v nej budeme ukladať, čo s nimi potrebujeme robiť, koľko je údajov, koľko operácií s údajmi budeme robiť a prípadne ďalšie kritériá, ako napríklad či sa bude výsledná aplikácia používať jednorázovo alebo opakovane mnohokrát, koľko času máme na jej naprogramovanie a pod. V nasledujúcom sa pokúsime ilustrovať na príkladoch výhody a nevýhody konkrétnej realizácie.

A iste si viete vymyslieť mnoho ďalších.

Toto je akési abstraktné...

Aktivita 2.	Vytvorte si 100 lístkov a očíslujte ich ľubovoľne číslami od 1 po 1000. Tieto lístky budeme používať vo viacerých nasledujúcich príkladoch, takže si ich odložte.
Aktivita 3.	Vyberte z lístkov náhodne desať. Rozložte si ich pred seba do radu zľava doprava. Lístky môžete položiť na papier a nad každý si napísať jeho pozíciu 1., 2., atď. Napríklad takto: 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 17 3 523 87 125 638 348 272 13 859 a) Z nevyložených lístkov jeden vyberte a vymeňte ho s lístkom na 4. pozícii. b) Na pozíciách s prvočíselnými hodnotami, t.j. 2., 3., 5. a 7. vymeňte lístky za ľubovoľné iné z ešte nepoužitých. c) Rozdeľte sa na dvojice a vymyslite svojmu susedovi nejakú jednoduchú úlohu, ako má lístky pomeniť. Napíšte ju a dajte susedovi. Navzájom si skontrolujte vaše riešenia.
Aktivita 4.	Rozmyslite si podrobne, čo znamená: „vsunúť“ na niektorú pozíciu lístok a „odstrániť“ lístok z niektorej pozície.

Aktivita 5.

a) „Posuňte“ lístky doľava, teda lístok z 2. pozície dajte na 1. pozíciu, lístok z 3. pozície na 2. pozíciu, atď. až po lístok z 10. pozície, ktorý dáte na 9. pozíciu. Na 10. pozíciu dajte lístok z doteraz nepoužitých.

b) „Posuňte“ lístky doprava, teda lístok z 1. pozície dajte na 2. pozíciu, lístok z 2. pozície na 3. pozíciu, atď. až po lístok z 9. pozície, ktorý dáte na 10. pozíciu. Na 1. pozíciu dajte lístok z doteraz nepoužitých.

Diskutujte v čom sa líšili postupy riešenia týchto úloh. Dalo sa v obidvoch prípadoch zrealizovať priamočiaro posunutie tak, ako to bolo napísané v texte úlohy? Zdôvodnite, prečo to bolo tak.

Operácie v poli:
- čítanie prvku na danej pozícii
- zápis prvku na danú pozíciu

Operácie v zozname:
- pridanie/odobranie prvku na začiatok/koniec
- pridanie/odobranie prvku pred/za daný prvok
- zistenie či je prvok v zozname
- zistenie či je zoznam prázdny

Aj nový mobil má prázdny zoznam kontaktov.

Všetky predchádzajúce aktivity boli na prácu s jednorozmerným poľom. Pozície lístkov sú vlastne indexy prvkov a číslo na lístku, ktorý je na pozícii *p*, je hodnota prvku poľa s indexom *p*. Výhoda poľa je v tom, že počítač vie ľahko pristúpiť k ľubovoľnému prvku poľa podľa jeho indexu (pozície). Keď ale chceme do poľa nejaký prvok vsunúť na danú pozíciu, alebo ho z poľa odstrániť, musíme všetky prvky poľa, ktoré sú za danou pozíciou, prípadne prvkom premiestniť (posunúť).

Aký je rozdiel medzi poľom a zoznamom?

Už sme spomínali, že v poli vieme pristupovať k hodnote prvku na *i*-tej pozícii, vieme ju prečítať, alebo zmeniť (zapísať). Všetky zložitejšie činnosti v poli musíme navrhnúť len s využitím uvedených dvoch elementárnych operácií.

V zozname sú elementárne operácie *iné* ako v poli. Pripomíname, že zoznam je postupnosť nejakých prvkov. Najjednoduchšie operácie v zozname sú zvyčajne pridanie prvku na začiatok zoznamu, alebo na koniec zoznamu. Odstránenie prvku zo začiatku zoznamu, alebo z konca zoznamu. Pridanie prvku pred, alebo za daný prvok a odobratie prvku, ktorý je pred, alebo za daným prvkom. Vieme tiež zistiť, či sú v zozname nejaké prvky, alebo nie. Keď v zozname nie sú žiadne prvky hovoríme, že je prázdny. Pre prvok zo zoznamu zvyčajne vieme, ktorý je nasledujúci (a niekedy aj predchádzajúci) prvok zoznamu.

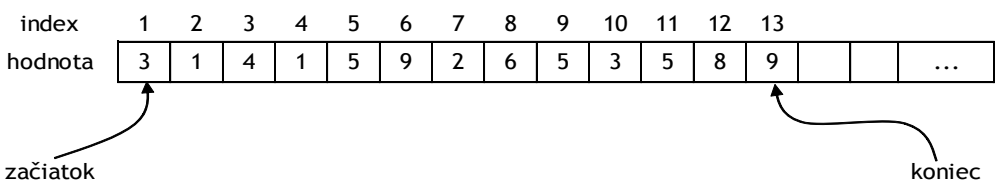
Asi ste si už všimli, že sme zatiaľ nikde nehovorili o tom, ako sa jednorozmerné pole, alebo zoznam realizujú. Nie je to preto, že by sme na to zabudli, ale preto, že to, aké elementárne operácie údajová štruktúra poskytuje a ako ich realizujeme, sú dve rôzne veci.

Ako realizovať zoznam?

Ako sa realizuje jednorozmerné pole, sa obyčajne nerozoberá, pretože samotná pamäť počítača tvorí jedno veľké jednorozmerné pole. Takže jednorozmerné pole je realizované „zadarmo“ priamo hardvérom počítača.

Realizácia jednorozmerným poľom

Pozrime sa teda, ako by sme vedeli prakticky zrealizovať operácie v údajovej štruktúre zoznam. Asi prvé, čo nám príde na um, je využiť na realizáciu jednorozmerné pole. To znamená prvky zoznamu uložiť jeden za druhým v jednorozmernom poli, ako je to znázornené na obr. 2.:



Obr. 2: Realizácia zoznamu jednorozmerným poľom

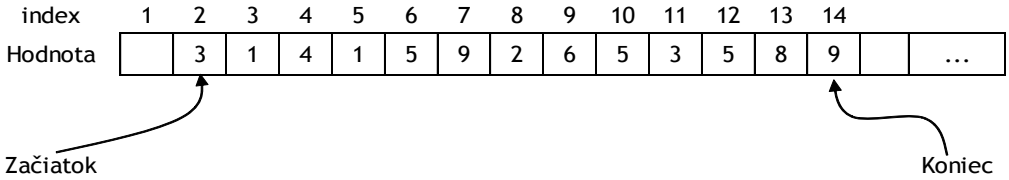
Všimnite si, že si okrem samotných prvkov, uložených v poli na indexoch 1 až 13, si pamätáme aj index prvého a posledného prvku. Je to preto, aby sme ľahko mohli, pridávať prvky na začiatok, alebo na koniec zoznamu. Uvedieme, ako by sme mohli zapísať príslušné deklarácie typov a premenných:

Index prvého prvku, potrebujeme len vtedy, keď prvý prvok zoznamu nie je v prvom prvku poľa.

```
type
  //budeme predpokladať, že zoznam bude mať najviac 100 prvkov
  TVelkostZoznamu = 1..100;
  THodnota = integer;
var
  Zaciatok, Koniec : integer;
  Hodnota : array[TVelkostZoznamu] of THodnota;
```

Z obr. 2 vidíme, že pridať prvok na koniec zoznamu vieme veľmi jednoducho, stačí zvýšiť hodnotu premennej **Koniec** o 1 a nový prvok zapísať do **Hodnota[Koniec]**. Pridať prvok na začiatok zoznamu, ale znamená, že si najprv musíme vytvoriť v poli miesto kam ho môžeme zapísať. Teda všetky prvky zoznamu uložené v poli **Hodnota** musíme posunúť o jedno miesto „doprava“ (obr. 3), porovnajte s obr. 2.

Pozornému čitateľovi iste neuniklo, že predpokladáme, že zoznam ešte nie je plný, t.j. že v ňom už nebolo 100 prvkov. Túto kontrolu musíme spraviť vždy, keď chceme posúvať prvky v poli doprava, t.j. posúvať **koniec.**



Obr. 3: Prvky zoznamu posunuté v poli o jedno miesto doprava

A teraz môžeme zmenšiť hodnotu premennej **Zaciatok** o 1 a nový prvok zapísať do **Hodnota[Začiatok]**.

Úloha 2.

Skúste analogicky opísať postup

a) pridávania a

b) odoberania

prvku do/zo zoznamu, keď to nemá byť ani na začiatku ani na konci, ale niekde uprostred zoznamu.

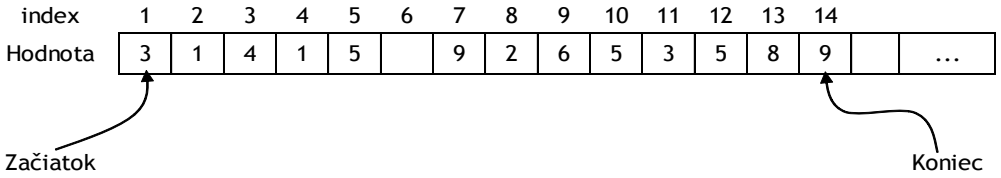
Namiesto práce, ktorú musí počítač vykonať na vyriešenie konkrétneho problému (úlohy), v informatike hovoríme o *zložitosti algoritmu*.

Zamyslime sa, koľko práce musel počítač vynaložiť pri vykonávaní vyššie opísaných činností – pridávanie a odoberanie prvku do zoznamu. *Ako meriame prácu počítača?* Zvyčajne sa to robí tak, že spočítame, koľko elementárnych inštrukcií musí počítač vykonať, aby zrealizoval danú úlohu. *Ktoré sú elementárne inštrukcie?* Na tom sa môžeme, a aj musíme, dohodnúť pred tým, než sa budeme usilovať určiť potrebnú prácu počítača. Pochopiteľne je dobré, keď za elementárne inštrukcie zoberieme také, ktoré budú čo najlepšie vystihovať nevyhnutnú prácu. Napríklad v našom prípade je praktické všímať si koľkokrát meníme hodnotu prvkov poľa.

Ľahko vidíme, že pri pridávaní prvku na začiatok zoznamu musíme posunúť všetky prvky zoznamu. Takže celkovo potrebujeme na vykonanie tejto operácie toľko priradení, koľko je prvkov v zozname. Hovoríme, že zložitosť tejto operácie je priamo úmerná dĺžke zoznamu (keďže úmernosť je v tomto prípade lineárna, hovoríme aj o zložitosti tejto operácie, že závisí lineárne od dĺžky zoznamu). Keď pridávame prvok na koniec zoznamu stačí nám jedno priradenie do poľa. V tomto prípade nezávisí počet priradení od počtu prvkov v zozname, vtedy vravíme, že je zložitosť konštantná (vykonáme iba konštantný počet operácií).

Hodnota y závisí od x *lineárne*, keď $y = a.x + b$, pre nejaké konštanty a, b.

Čo keď pridávame prvok doprostred zoznamu? Vtedy musíme preň vytvoriť miesto a posunúť všetky prvky, ktoré majú byť za ním o jedno miesto doprava (obr. 4, porovnajte s obr.2).



Obr. 4: Vytvorenie miesta pre pridanie prvku uprostred zoznamu.

Vidíme, že počet prvkov, ktoré v poli musíme posunúť, závisí od toho, za ktorý prvok zoznamu chceme nový prvok vsunúť. Teda počet priradení, ktorým meriame potrebnú prácu je niekde medzi 1 a počtom prvkov zoznamu.

Keď pre štruktúru údajov potrebujeme na vykonanie nejakej operácie prácu úmernú počtu prvkov, ktoré sú v štruktúre zapamätané, nie je to najhoršie, ale ani veľmi šikovné. Skúste odhadnúť koľko priradení by sme v najhoršom museli vykonať, keby sme chceli do zoznamu realizovanom v jednorozmernom poli pridať 1000000 prvkov na

- začiatok zoznamu,
- koniec zoznamu

Ako je to, keď chceme prvok zo zoznamu odstrániť? Predpokladajme, že vieme index v poli, kde je prvok uložený. Je očividné, že po odstránení prvku nemôže niekde uprostred pola ostať prázdne miesto (napr. situácia ako na obr.4). Analogicky ako pri vkladaní musíme po odstránení prvku z poľa všetky prvky, ktoré sú v poli uložené za ním posunúť o jedno miesto doľava. Teda aj potrebná práca bude rovnaká ako pri vkladaní prvku do zoznamu.

Úloha 3.

Navrhnite a napíšte procedúru (procedúry), ktorá bude realizovať

a) pridanie prvku do zoznamu,

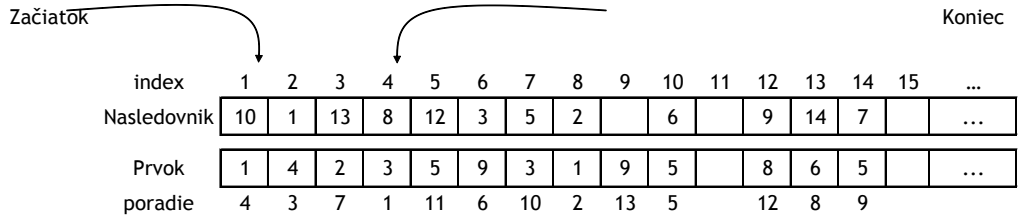
b) odobratie prvku zo zoznamu.

Porovnajte si svoje návrhy a realizácie (spoločne alebo v dvojiciach).

Realizácia dvomi poľami

V predchádzajúcom sme videli, že ak zoznam realizujeme v jednorozmernom poli tak, že dáme prvky do poľa jeden za druhým, tak ako sú v zozname, bude pridávanie a odoberanie prvkov zoznamu vyžadovať prácu závisiacu lineárne od počtu prvkov v zozname. Dá sa to urobiť aj lepšie? Skúsme sa zamyslieť, čo zapríčinilo, že sme potrebovali toľko operácií. Museli sme posúvať prvky zoznamu uložené za prvkom, ktorý sme vkladali, alebo odoberali zo zoznamu. Vieme sa zbaviť posúvania prvkov? Ako to urobiť?

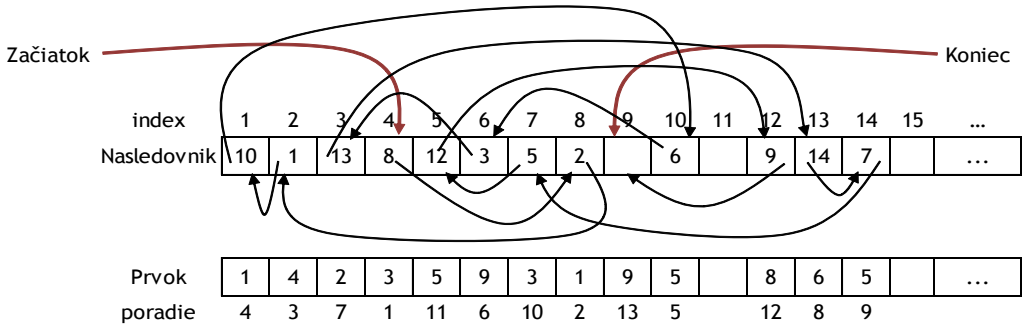
Prečo sme museli prvky posúvať? Aby sme vytvorili príp. odstránili voľné miesto v poli, ktoré bolo treba na umiestnenie nového prvku, alebo vzniklo po odobratí prvku. Keby sme nový prvok zapísali do ľubovoľného miesta v poli, stratili by sme informáciu, za ktorým prvkom v zozname má byť! Riešením je zapamätať si pre každý prvok zoznamu, ktorý je nasledujúci prvok zoznamu. Realizovať to môžeme napríklad tak, že pre každý prvok si budeme pamätať index, kde je v poli uložený za ním nasledujúci prvok zoznamu. K poľu **Prvok** pridáme pole **Nasledovnik**, pričom v **Nasledovnik[i]** je uložený index, kde je nasledovník prvku **Prvok[i]** v poli **Prvok** (obr. 5).



Obr. 5: Realizácia zoznamu poľom prvkov a poľom indexov ich nasledovníkov.

Na obrázku 6 ilustrujeme pomocou šípiek nasledovníkov jednotlivých prvkov zoznamu, ktorý sme zobrazili na obr. 5. Všimnite si, že prvý prvok zoznamu je

Prvok[4], posledný prvok zoznamu je **Prvok[9]**, **Prvok[11]** neobsahuje žiaden prvok zoznamu.



Obr. 6: Znázornenie nasledovníkov prvkov

To vyzerá celkom ako zamotané špagety ...

Je to riadne neprehľadné. Má to vôbec nejakú výhodu oproti predchádzajúcemu spôsobu? Pozrime sa ako by sme pri takejto realizácii zoznamu vedeli vykonávať pridávanie a odoberanie prvkov do zoznamu.

Aktivita 6.	Vytvorte dvojice. Napíšte si nejaký 10 prvkový zoznam, zapíšte ho do poľa **Prvky**, tak aby prvky zoznamu neboli v poli **Prvky** za sebou. Doplňte zodpovedajúco hodnoty pola **Nasledovnik** a navzájom si skontrolujte, či zoznam v poli **Prvky** zodpovedá vášmu pôvodnému zoznamu.
Úloha 4.	Pokúste sa navrhnúť postup vykonávania pridávania/odoberania prvku do/zo zoznamu realizovaného poľami **Prvok** a **Nasledovnik**. Diskutujte o problémoch, ktoré sa pri tom vyskytli.

Opísané štruktúry by sme zadefinovali napríklad takto:

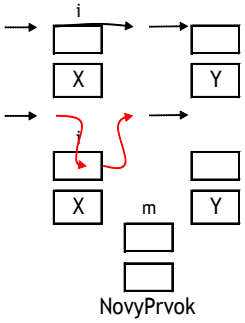
```
type
  //budeme predpokladať, že zoznam bude mať najviac 100 prvkov
  TVelkostZoznamu = 1..100;
  THodnota = Integer;
var
  Zaciatok, Koniec : Integer;
  Nasledovnik : array[TVelkostZoznamu] of Integer;
  Prvok : array[TVelkostZoznamu] of THodnota;
```

Keď vieme, za ktorý prvok zoznamu chceme pridať prvok, napríklad nech je to **Prvok[i],** tak nový prvok zapíšeme na neobsadené miesto, nech je to napríklad **Prvok[m]** a ešte musíme pozorne upraviť pole **Nasledovnik**, aby nasledovník prvku **Prvok[i]** bol index pridaného prvku, teda **m** a nasledovník pridaného prvku bol pôvodný nasledovník prvku **Prvok[i]**. V programe by sme to mohli zapísať:

Potvrdenie „zákona o zachovaní obtiažnosti“.

```
Prvok[m] := NovyPrvok;
Nasledovnik[m] := Nasledovnik[i];
Nasledovnik[i] := m;
```

Pridanie prvku:



Iste sa pýtate: „ako nájdeme v poli **Prvok** voľné miesto, kam môžeme zapísať novú hodnotu?“ Napríklad tak, že voľné miesto bude označené tým, že v poli **Nasledovnik** bude hodnota -1, teda napríklad **Nasledovník[m]** bude mať hodnotu -1. Čo to znamená v praxi, že na začiatku behu programu budú všetky hodnoty v poli **Nasledovník** rovné -1 a keď budeme pridávať do zoznamu nový prvok, musíme nájsť napríklad najmenší index **n**, pre ktorý **Nasledovnik[n]=-1**. To ale znamená, že sme si vôbec nepomohli, lebo, síce nový prvok do zoznamu vieme zaradiť rýchlo, ale

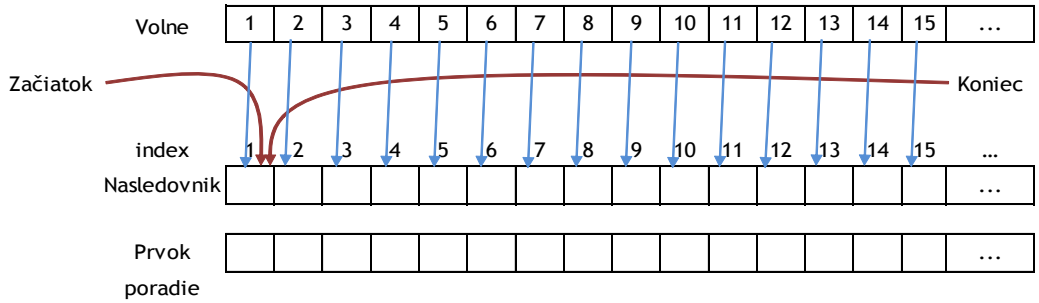
Takže už nepočítame len počet vykonaných priradení, ale aj počet vykonaných porovnaní...

na to, aby sme našli voľné miesto, potrebujeme skontrolovať v najhoršom prípade toľko hodnôt prvkov v poli **Nasledovník**, koľko je prvkov v zozname. Ak by sa nám vyhľadávanie voľného miesta podarilo urýchliť, mala by takáto realizácia predsa len praktický zmysel. Vieme to urobiť? Našťastie áno.

Úloha 5.

Navrhnite ako upraviť realizáciu zoznamu, aby sme vedeli voľné miesto nájsť vykonaním konštantného počtu operácií.

Jedno riešenie napríklad je zobrať si na pomoc ďalšie pole **Volne**, ktoré bude obsahovať pozície voľných miest v poli **Prvok** (zodpovedajúca pozícia v poli **Nasledovnik** bude tiež voľná). Na začiatku by to vyzeralo napríklad ako na obr. 7.



*Obr. 7: Evidencia voľných miest v poli **Volne***

To, že vopred si nevieme predstaviť všetky detaily hľadaného riešenia a stále sa objavujú ďalšie a ďalšie problémy, ktoré musíme vyriešiť, je viac menej typická situácia, ktorá sa prirodzene vyskytuje v procese hľadania riešenia každého trochu zložitejšieho problému (aj programátorského). Takže treba zachovať pokoj, nič zvláštne to nie je a postupne treba všetky detaily vyriešiť.

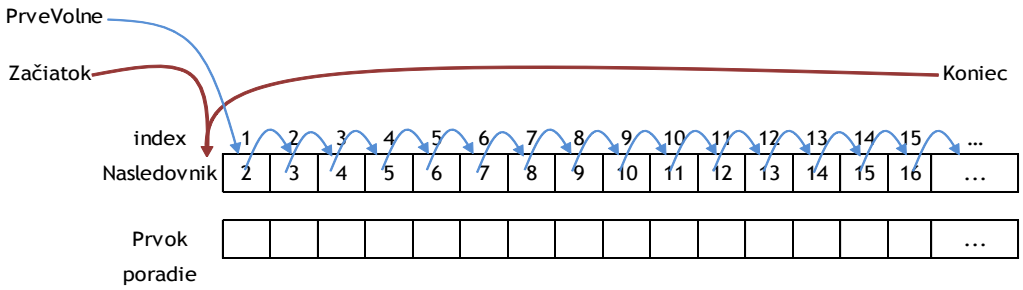
Čo sa bude diať, keď chceme do zoznamu pridať prvok? V poli **Volne** máme informáciu o tom, kde je voľné miesto pre uloženie prvku. Informácie o voľných miestach musia byť v poli **Volne** rýchlo prístupné, najprirodzenejšie je keď budú uložené na po sebe idúcich pozíciách, začínajúc od prvej. Počas práce s poľom **Volne** by sme hodnoty v ňom nemali posúvať (prečo?). Budeme si pamätať pozíciu v poli **Volne**, od ktorej sú informácie o voľných miestach uložené - v premennej **PrvaVolna**. Pri pridávaní prvku, tento uložíme do poľa **Prvok** na pozíciu, ktorú zoberieme z **Volne[PrveVolne]** a **PrveVolne** zvýšime o 1. Toto ozaj vieme robiť rýchlo.

Ďalšia operácia je odstránenie prvku zo zoznamu. Je to veľmi podobné ako pridávanie prvku. Keď chceme odstrániť prvok zoznamu za prvkom **Prvok[i]** (nech je na pozícii **m**), musíme vykonať nasledujúce operácie:

```
m := Nasledovnik[i];
Nasledovnik[i] := Nasledovnik[m];
PrveVolne := PrveVolne - 1;
Volne[PrveVolne] := m;
```

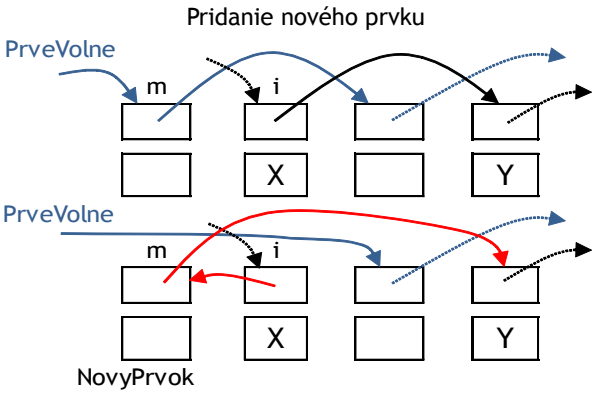
Úloha 6.

Slovne vysvetlite význam predchádzajúcich príkazov. Nakreslite obrázky analogické obr. 7 pred a po odstránení prvku.



*Obr. 8: Evidencia voľných (modré) miest v poli **Nasledovník***

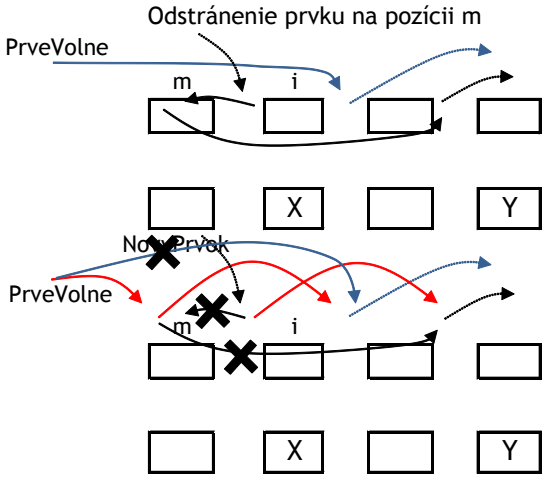
Aj keď majú dnešné počítače veľa pamäte, potrebujeme skutočne pole **Volne**? Predsa voľné miesta môžu tiež tvoriť zoznam! Takže si ich pozície vieme pamätať v tom istom poli **Nasledovník** ako aj prvky zoznamu. **PrveVolne** bude pozícia prvého prvku v zozname voľných. Na začiatku budú všetky pozície voľné (obr.8), takže budeme mať iba zoznam voľných miest. Keď do zoznamu budeme pridávať prvok zo zoznamu voľných si vyberieme prvý prvok, ktorý nám určí voľnú pozíciu v poli **Prvok** (aj **Nasledovnik**), kam môžeme prvok pridať. Prvý prvok zo zoznamu voľných prvkom je vlastne to miesto kam zapíšeme pridávaný prvok. **PrveVolne** „posunieme“ na pozíciu druhého prvku v zozname voľných (ktorý sa tým stane prvým prvkom tohto zoznamu). Zapísané v programovacom jazyku:




```
// predpokladáme, že pridávame za i-ty prvok v zozname
m := PrveVolne;  // pozícia prvého voľného
PrveVolne := Nasledovnik[PrveVolne];  // posunieme PrveVolne
Prvok[m] := NovyPrvok;
Nasledovnik[m] := Nasledovnik[i];
Nasledovnik[i] := m;
```

Úloha 6.	Nakreslite si situáciu po každom priradení v predchádzajúcom programe
Úloha 7.	Ďalší detail, na ktorý nesmieme zabudnúť, je pridávanie prvého prvku v zozname a pridávanie doprostred zoznamu (za i-ty prvok v zozname). Doplňte program aj o tento prípad.

Ešte si všimnime, ako by sa bez použitia poľa **Volne** odberali prvky zo zoznamu. Predpokladajme, že chceme odstrániť prvok zoznamu za prvkom **Prvok[i]**. Hlavná myšlienka riešenia je že odstránený prvok musíme pridať do zoznamu voľných prvkov a **Nasledovnik[i]** upraviť, tak aby odteraz bol nasledovníkom prvku **Prvok[i]** prvok zoznamu nasledujúci za odstráneným prvkom. Vyriešenie detailov vyžaduje trocha pozornosti, pretože jednotlivé činnosti musíme robiť v správnom poradí. Zápis v programovacom jazyku je za úlohou 8.



Úloha 8.	Nakreslite si situáciu po každom priradení v nasledujúcom programe.

```
// predpokladáme, že odoberáme za i-tym prvkom v zozname
m := Nasledovnik[i];  // pozícia odstraňovaného prvku
Nasledovnik[i] := Nasledovnik[m];  // odstránenie prvku zo zoznamu
Nasledovnik[m] := PrveVolne; //odstránený prvok dáme na začiatok
                             //zoznamu voľných prvkov
PrveVolne := m;
```

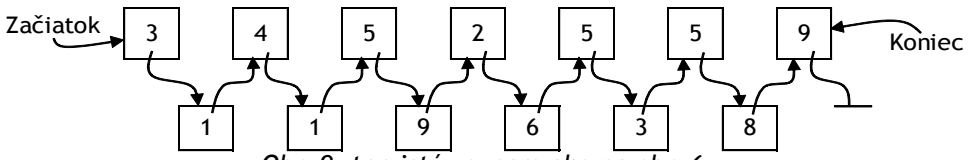
Po tých všetkých „špagetových“ obrázkoch iste oceníte znázorňovanie prvkov zoznamu v tvare „vláčika“ - obdĺžnikov predstavujúcich prvky zoznamu a pospájaných šípkami, ktoré znázorňujú, ktorý prvok je v zozname za ktorým. Šípky

majú znázorňovať aj to, z ktorého prvku zoznamu v programe vieme rýchlo prejsť na ktorý.

Úloha 9.	Predstavte si, že v zozname ste na prvku **Prvok[i]**. Navrhnite postup ako pridať do zoznamu prvok pred **Prvok[i]**, prípadne predchádzajúci prvok zo zoznamu odstrániť.

Prekreslime obr. 6 pomocou „vláčikovým“ spôsobom:

Značka označuje, že za týmto prvkom už nie je ďalší.



Obr. 9: ten istý zoznam ako na obr. 6.

Takáto reprezentácia sa nazýva *spájaný zoznam*. Keď ste sa dočítali až sem a poctivo ste spravili aj všetky cvičenia, asi ľahko uveríte, že realizovať údajovú štruktúru zoznam nie je celkom jednoduché. Kreslenie šípok vyžadovalo veľa pozornosti a aj napriek sústredeniu sa, veľa krát sme sa pomýlili. Uvedený spôsob realizácie má ale svoje výhody, preto sa v praxi často používa (a nielen pri zoznamoch). Údajové štruktúry, ktoré sú takýmto spôsobom realizované sa zvyknú nazývať *dynamické*. Aby sa minimalizovala možnosť chýb, moderné programovacie jazyky poskytujú konštrukcie, ktoré uvedené realizácie údajových štruktúr pred programátorom ukrývajú. Programátor môže využívať výhody, ktoré takýto spôsob realizácie údajových štruktúr poskytuje, ale nemôže svojou nepozornosťou urobiť také chyby, ktoré sa často vyskytujú. Ďalšia výhoda dynamických údajových štruktúr je, že nemusíme vopred vedieť aké budú veľké, jediný obmedzujúci faktor je zvyčajne kapacita voľnej pamäte počítača.

Napríklad programovací jazyk Java.

Tak má, či nemá posledný prvok?

Úloha 10.	Čo by sa stalo, keby „posledný“ prvok v zozname mal ako svojho nasledovníka prvý prvok? Viete si predstaviť nejakú situáciu, v ktorej by bola takáto údajová štruktúra užitočná?
Úloha 11.	Ako by ste riešili úlohu, keď máte pre dve čísla m a n zistiť celú časť podielu m/n, zistiť periódu a predperiódu desatinného zápisu. Napríklad, pre $m = 3$ a $n = 280$ je podiel $0{,}010714285$, kde červenou je označená celá časť, modrou predperióda a čiernou perióda podielu

Čo sme sa naučili

Že údajová štruktúra je charakterizovaná operáciami, ktoré sa pomocou nej dajú realizovať. Že jednu údajovú štruktúru môžeme realizovať viacerými spôsobmi, ktoré určujú množstvo práce, ktorú počítač potrebuje na vykonanie jednotlivých operácií v údajovej štruktúre.

Rad a zásobník

Rad a zásobník sú štruktúry údajov, s ktorými sa často stretávame v praxi. V úrade, banke, u lekára čakáme v rade. Zásobník je trocha ukrytejší: niekedy spracovávame prichádzajúce požiadavky tak, že najskôr sa usilujeme vybaviť poslednú ktorá prišla. V digitálnom fotoaparáte si často môžete pozrieť poslednú vytvorenú fotografiu ako prvú, predposlednú ako druhú, atď. Vagóny na slepej koľaji tvoria zásobník: vagón ktorý prišiel na slepú koľaj ako posledný, odíde ako prvý a pod.

Príklad zásobníka tanierov.

Rad aj zásobník sú dôležité štruktúry údajov. Rad umožňuje napríklad ľahko vyriešiť úlohu ako nájsť cestu z bludiska. Zásobník je užitočný keby sme chceli napísať napríklad program kalkulačka a mnoho iných programov, používa sa napríklad vždy keď v programe zavoláme nejakú funkciu alebo procedúru, bez toho, že by sme o tom vedeli.

Zásobník

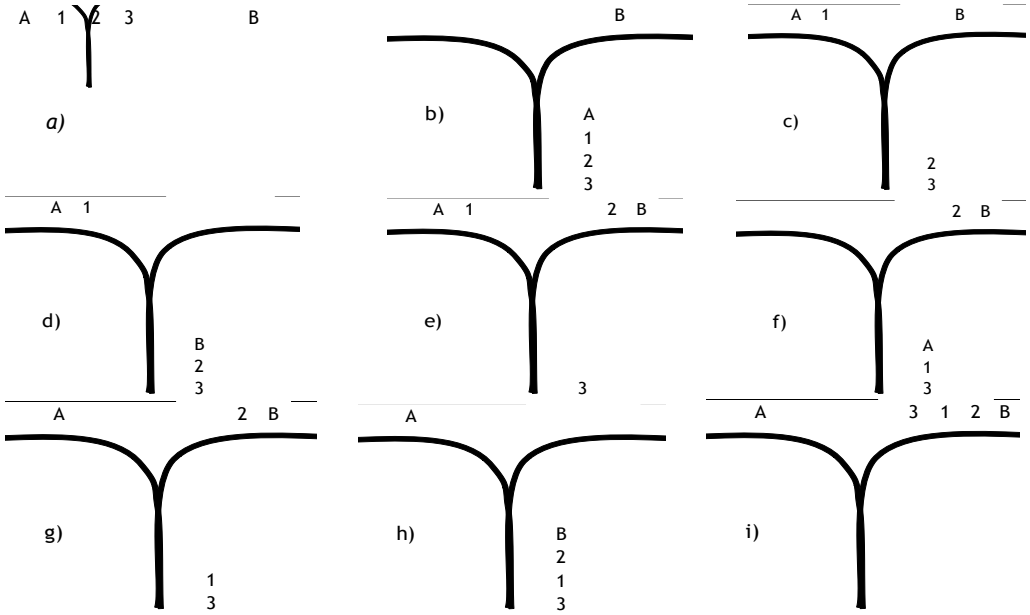
Aktivita 1.

Zoberieme si tri kúsky papiera očíslované 1, 2 a 3. Budú predstavovať čísla vagónov, ktoré sú zoradené za sebou. Predstavme si, že súpravu vagónov tlačí lokomotíva A (obr. 1.a) a celá súprava môže zájsť na slepú koľaj tvaru Y (obr. 1.b), kde zo súpravy môžeme odpojiť/pripojiť ľubovoľné vagóny. Lokomotíva A sa zo zvyšnými (aj nijakými) vagónmi vráti na koľaj odkiaľ prišla (obr. 1.c). Lokomotíva B, na druhom ramene Y, môže zo slepej koľaje odtiahnuť na druhé rameno ľubovoľný počet vagónov (obr. 1.d a e). Obe lokomotívy sa môžu do spoločnej slepej koľaje vrátiť neobmedzený počet krát a pripájať odpájať ľubovoľné vagóny.

Aké poradia vagónov môže lokomotíva ťahať?

Vedeli by ste povedať aké poradie vagónov lokomotíva B nikdy nemôže ťahať?

Posledný dnu, prvý von.



Obr. 1. a) Počiatočná situácia, lokomotíva A s troma vagónmi. b) Lokomotíva A dotlačila vagóny na slepú koľaj. c) Lokomotíva A sa vrátila s vagónom č.1 a na slepej koľaji nechala vagóny 2 a 3. d) Lokomotíva B prišla na slepú koľaj. e) B odtiahla zo slepej koľaje vagón č. 2. f) A dotlačila na slepú koľaj vagón č. 1. g) A sa vrátila bez vagónov. h) B dotlačila na slepú koľaj vagón č. 2. i) B odtiahla zo slepej koľaje vagóny zoradené v poradí 2 1 3.

Aktivita 2.

Ako Aktivita 1, ale zoberte súpravu zloženú s piatich vagónov.

Viete dosiahnuť aby lokomotíva B ťahala vagóny v poradí

a) 1 2 5 3 4

b) 5 4 3 2 1

c) 1 2 3 5 4

Veď to poznáme, aj CD média predsa predávajú v zásobníkovom balení, v ktorom sú CD bez obalov uložené na kolíku. Vybrať sa dá iba vrchne CD a pridať sa dá len na vrch.

Podobne ako údajová štruktúra zoznam, ktorú sme charakterizovali operáciami, ktoré poskytuje urobíme to aj v prípade zásobníka. Do zásobníka môžeme pridať prvok, odobrať prvok, pričom vždy odoberáme naposledy pridaný prvok a môžeme sa pýtať, či je zásobník prázdny a zistiť hodnotu prvku na vrchu zásobníka.

Operácie v zásobníku:

- pridanie prvku
- odobranie naposledy pridaného prvku
- zistenie, či je zásobník prázdny
- zistenie hodnoty prvku na vrchu zásobníka

Zásobník sa dá realizovať jednoduchým spôsobom v jednorozmernom poli.

```
const
  //budeme predpokladať, že zásobník bude mať najviac 100 prvkov
  MaxVelkostZasobnika = 100;
type
  TVelkostZasobnika = 1..MaxVelkostZasobnika;
  THodnota = Integer;
var
  Vrch : Integer;
  Zasobnik : array[TVelkostZasobníka] of THodnota;
```

Premenná **Vrch** bude určovať pozíciu naposledy pridaného prvku do zásobníka. Prvky zásobníka sa budú ukladať v poli **Zasobnik** od nižších pozícií smerom k vyšším. Keď bude zásobník prázdny, bude mať **Vrch** hodnotu 0. Takže operáciu, či je zásobník prázdny realizujeme jednoduchým testom:

```
function JePrazdnyZasobnik : Boolean;
begin
  Result := Vrch = 0;
end;
```

Operáciu pridania do zásobníka môžeme realizovať ako funkciu, ktorá vráti **True**, keď sa operácia pridania podarí. Môže sa aj nepodariť? Nepodarí sa napríklad vtedy, keď je už zásobník plný.

```
function PridajDoZasobnika(Prvok : THodnota) : Boolean;
begin
  Result := Vrch < MaxVelkostZasobnika;
  if Result then begin
    Inc(Vrch);
    Zasobnik[Vrch] := Prvok
  end;
end;
```

V prípade, že sa do zásobníka ešte dá pridať ďalší prvok, hodnota premennej **Vrch** sa zvýši o 1 a pridávaný prvok sa zapíše do poľa **Zasobnik** na pozíciu **Vrch**.

Úloha 1.

Napíšte funkciu **JePlnyZasobnik**, ktorá vráti hodnotu **True**, keď je zásobník plný, inak vráti hodnotu **False**.

Odobratie prvku zo zásobníka je rovnako jednoduché. Budeme predpokladať, že z prázdneho zásobníka sa nebudeme nikdy snažiť vyberať prvok. Pred volaním **VyberZoZasobnika** vždy zistíme funkciou **JePrazdnyZasobnik**, či nie je prázdny.

```
function VyberZoZasobnika : THodnota;
begin
  Result := Zasobnik[Vrch];
  Dec(Vrch);
end;
```

Uvedená realizácia údajovej štruktúry zásobník bola veľmi jednoduchá. Samozrejme má svoje nevýhody. Pomerne obmedzujúce je, že veľkosť zásobníka je určená konštantou **MaxVelkostZasobnika**. V praxi ale najčastejšie nevieme koľko prvkov do zásobníka budeme dávať, preto je prakticky zaujímavejšia realizácia zásobníka spájaným zoznamom. Čo sa týka náročnosti na prácu počítača, všetky uvedené operácie so zásobníkom sa dajú robiť na konštantný počet operácií (nezávisí od počtu prvkov v zásobníku).

Zdá sa, že predsa len má ten spájaný zoznam aj nejaké praktické využitie...

Rad

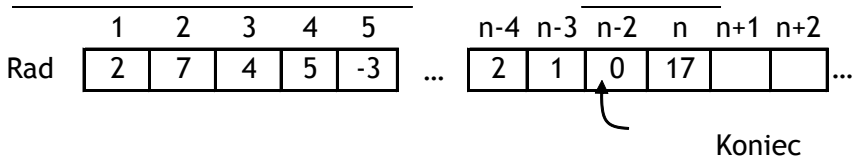
Ako sme už v úvode spomínali, rad je pomerne bežný aj v každodennom živote. Čím sa rad líši od zásobníka? Operáciami, ktoré poskytuje. Rad je postupnosť prvkov, pričom prvky môžeme pridávať a odoberať, na rozdiel od zásobníka, z oboch jeho koncov (začiatku aj konca). Samozrejme vieme zistiť, či je rad prázdny a hodnoty prvkov na oboch jeho koncoch.

Rad poznáme napríklad z ambulancie lekára: posledný kto príde sa zaradí na koniec radu čakajúcich pacientov a k lekárovi ide ten, kto prišiel najskôr.

Realizovať rad vieme v jednorozmernom poli. Najjednoduchšia realizácia radu v jednorozmernom poli bude zachovávať začiatok radu vždy na pozícii 1 (obr. 2.). Teda pri odstránení prvku zo začiatku budeme musieť posunúť všetky prvky v rade o jedno miesto. Ak uvážime, že táto operácia môže byť dosť častá, celkovo vyžaduje takáto realizácia veľa práce počítača. Dá sa to aj lepšie? Našťastie áno.

Prvý dnu, prvý von.

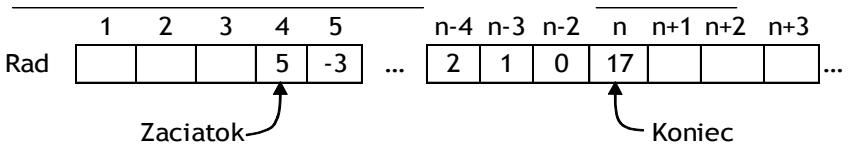
Operácie v rade:
- pridanie prvku na začiatok
- pridanie prvku na koniec
- odobranie prvku zo začiatku
- odobranie prvku z konca
- zistenie, či je rad prázdny
- zistenie hodnoty prvku na začiatku
- zistenie hodnoty prvku na konci



Obr. 2. Rad so začiatkom na pozícii 1a premenlivým koncom.

Keď netrváme na tom, že začiatok radu musí byť vždy na prvej pozícii, ale budeme ho posúvať vždy keď odoberieme prvok zozačiatku radu dostaneme šikovnejšiu realizáciu (obr. 3). Odpadne nevyhnutnosť posúvania všetkých prvkov v rade a odobratie prvku zo začiatku už bude vyžadovať len konštantný počet operácií.

```
const
  //budeme predpokladať, že zásobník bude mať najviac 100 prvkov
  MaxVelkostRadu = 100;
type
  TVelkostRadu = 1..MaxVelkostRadu;
  THodnota = Integer;
var
  Zaciatok, Koniec : Integer;
  Rad : array[TVelkostRadu] of THodnota;
```



Obr. 3. Rad so premenlivým začiatkom aj koncom.

Tento spôsob je výhodnejší než by sa na prvý pohľad mohlo zdať. Rad totiž môže v poli putovať oboma smermi podľa toho, ako pribúdajú resp. ubúdajú prvky z jeho

oboch koncov. Treba ale dať pozor pri realizácii operácií pridávania a odoberania prvku, pretože sa môže stať, že niektorá z premenných **Zaciatok,** alebo **Koniec** „vybehne“ mimo rozsah pozícií poľa. Pri takejto realizácii je výhodné, aby po pozícii **MaxVelkostRadu** nasledovala pozícia 1 a aj opačne pred pozíciou 1 bola pozícia **MaxVelkostRadu** (môžeme si to predstaviť, že prvky radu **Rad** sú v kruhu).

Úloha 2.

Zrealizujete operácie údajovej štruktúry rad. Použite realizáciu s posuvným začiatkom aj koncom radu. Vyberte si aspoň jednu operáciu pridania prvku, jednu operáciu odobratia prvku a test, či je rad prázdny.

Keď úlohu vyriešite, rozdeľte sa na dvojice, vysvetlite vaše riešenie druhému a diskutujte o svojich riešeniach.

Realizácia radu v jednorozmernom poli má tú istú nevýhodu, ako mala aj realizácia zásobníka, že počet prvkov v rade je obmedzený počtom prvkov v poli **Rad**. Rovnako ako pri zásobníku, tento nedostatok sa dá odstrániť využitím spájaného zoznamu.

S použitím radu ste sa stretli už v [4], kde ste sa zoznámili s jednoduchým prehľadávaním grafov a hľadaním najlacnejšej cesty v grafe.

Úloha 3.

Zopakujte si hľadanie najkratšej cesty v grafe a diskutujte o tom, ako by sa tento postup dal zrealizovať využitím údajovej štruktúry rad.

Pomocou radu sa dá hľadať aj cesta z bludiska, ktoré sme spomínali v časti o dvojrozmernom poli. Keď stojíme niekde v neznámom bludisku. Počiatočnú pozíciu vložíme na začiatok radu **R**. Potom opakujeme nasledujúcu činnosť pokým rad nie je prázdny, alebo sme nenašli východ z bludiska: Z konca radu **R** odoberieme pozíciu, označíme ju ako navštívenú a rozšírime sa všade, kam sa z nej dá rozšíriť (t.j. na s ňou susediace pozície, kde sme ešte neboli a dá sa na ne ísť – nie je tam prekážka, napr. stena a pod.) a všetky pozície kam sa dalo rozšíriť, pridáme na začiatok radu. Dovolíme si to zapísať schematicky nasledujúcim spôsobom:

Názvy procedúr a funkcií naznačujú čo robia.

```
Pozicia := PoziciaVchodu;
PridajNaZaciatokRadu(Pozicia);
Pozicia := VyberZKoncaRadu;
repeat
  OznacPoziciuAkoNavstivenu(Pozicia);
  for SusedPozicie zo VsetciSusedia(Pozicia) do begin
    if Volny(SusedPozicie) and Nenavstiveny(SusedPozicie) then
      PridajNaZaciatokRadu(SusedPozicie)
  end;
  Pozicia := VyberZKoncaRadu;
until (PrazdnyRad or JeVychod(Pozicia));
if JeVychod(Pozicia) then NasliSmeVychod
else VychodNeexistuje
```

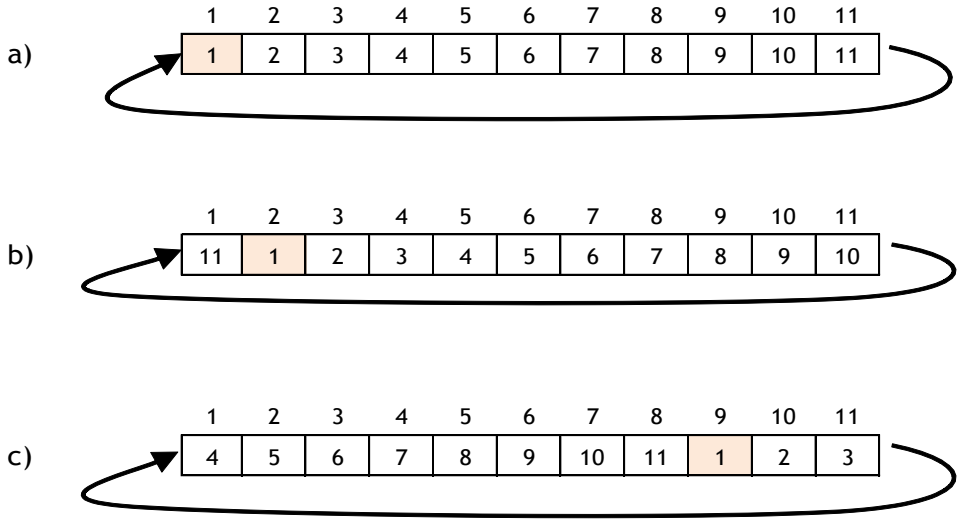
Čo sme sa naučili

Zoznámili sme sa s údajovou štruktúrou zásobník a rad. Ukázali sme si aké operácie poskytujú a ako sa dajú jednoduchým spôsobom realizovať.

Úloha s posúvaním jednorozmerného poľa

Keď zadáme žiakom úlohu typu: Napíšte program, ktorý ... .
Ak má mať výsledný program aspoň desať riadkov, s istotou dostaneme takmer toľko rôznych riešení, koľko je žiakov.
Uvedením viacerých riešení chceme ukázať, že aj jednoduchá úloha má veľký počet rôznych riešení.

V tejto časti uvedieme niekoľko riešení jednoduchej algoritmickej úlohy v jednorozmernom poli. Úlohou bude posunúť „cyklicky“ prvky poľa o k prvkov doprava (obr. 1.). Pri každom z prezentovaných riešení sa zamyslíme, koľko práce počítač musí vykonať a aj koľko pri tom bude potrebovať pamäti. Na základe týchto kritérií porovnáme predložené riešenia.



Obr. 1. a) Jedenásť prvkové pole; b) Pole z a) sme cyklicky posunuli o jedno miesto doprava. Všimnite si, že prvok z pozície 11 sa dostal na pozíciu 1; c) Pole z a) sme posunuli o 8 miest doprava (alebo, čo je to isté, o 3 miesta doľava, alebo čo je to isté o -3 prvky doprava).

Uf. Posúvať o -3 miesta doprava namiesto o 3 doľava? Nie je to priveľmi matematické? Ale v zadaní mame len posúvanie doprava ...

Úloha 1.

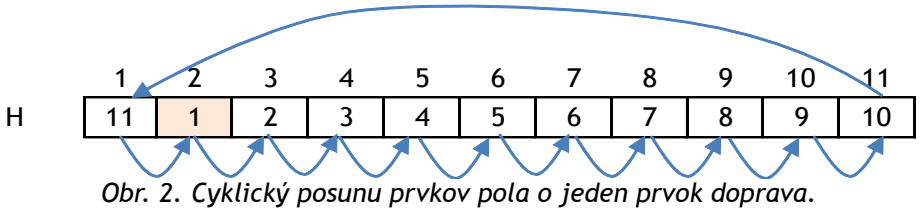
Navrhnite postup na riešenie uvedenej úlohy. Snažte sa ho rozpracovať tak podrobne, ako viete. Vedeli by ste Vaše riešenie zapísať v programovacom jazyku?

1. riešenie

Ak by sa ukázalo, že pole na obr. 1 je priveľké, smelo môžeme zobrať menšie.

Použime najprv odskúšaný postup a vyriešme úlohu pre pole konkrétnej veľkosti a s konkrétnou, čo najmenšou hodnotou k. Skúsme pole z obr. 1. a čo najmenšiu hodnotu k. Pre $k = 0$, netreba prvky v poli posúvať a riešením je priamo vstupné pole. Pre $k = 1$, ide o posun o jeden prvok doprava a výsledok je na obr. 1 b). Keď je $k > 1$, t.j. posun o k miest doprava, vieme úlohu vyriešiť opakovaním posunu o jedno miesto k-krát po sebe. Takže všeobecná úloha sa nám zmenila na riešenie konkrétneho prípadu - cyklického posunu o jeden prvok doprava. To je celkom jednoduchá úloha. Skúsme ju vyriešiť.

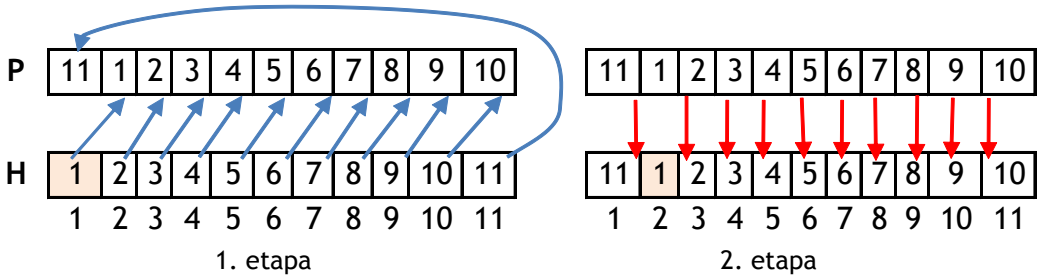
Pozrime sa, čo presnejšie znamená posunúť cyklicky pole o jedno miesto doprava. Znázornili sme to na nasledujúcom obrázku:



Obr. 2. Cyklický posunu prvkov pola o jeden prvok doprava.

Vidíme, že **H[1]** sa presunie na miesto prvku **H[2]**. Na to, aby sme to mohli vykonať bez toho, aby sme prepísali hodnotu prvku **H[2]**, musíme prvok **H[2]** najprv posunúť na miesto prvku **H[3]**, atď. Zdá sa, že najmenej náročné na riešenie technických detailov bude použitie pomocného pola **P**, rovnakej veľkosti ako pole **H**. Riešenie

úlohy rozdelíme na dve etapy. V prvej prepíšeme prvky poľa **H** do pola **P** tak, aby už boli na výslednej pozícii ako po cyklickom posunutí. V druhej etape prvky pomocného poľa **P** prepíšeme naspäť do poľa **H**. Na obr. 3 sú znázornené obe etapy.

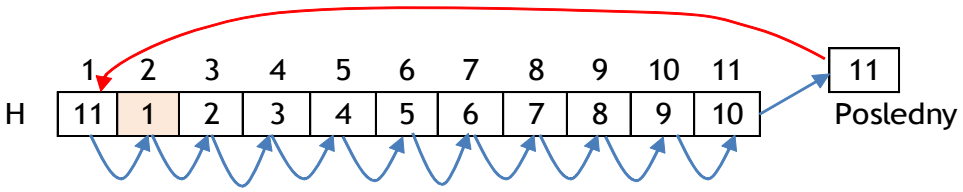


Obr. 3. Posun o jeden prvok s použitím pomocného poľa. Modrou farbou je znázornená 1. etapa a červenou 2. etapa

Riešenie posúvaním o jeden prvok s pomocným poľom:
priradení: $2kn$
pamäť: n

Zamyslime sa, koľko práce vyžaduje takéto riešenie. Každý prvok poľa **P** dvakrát prepíšeme. Najprv do poľa **H** a potom nazad do poľa **P**. Ak predpokladáme, že pole **P** má n prvkov, riešenie vyžaduje $2n$ priradení. Koľko potrebuje pamäti? Okrem niekoľkých premenných (pomocné premenné, napr. v cykle), ktorých počet nezávisí od počtu prvkov, potrebujeme pomocné pole **P** veľkosti n. Keď pole posúvame o k miest, musíme vykonať celý hore uvedený postup k krát.

Nešlo by to predsa len aj bez pomocného poľa **P**? Uvedomme si čo bola prekážka posúvania prvkov priamo v poli **P**. Keď sme chceli zapísať napríklad na miesto i prvok, ktorý sa tam dostal posunom z predchádzajúceho miesta, museli sme už mať z miesta i prvok presunutý, atď. až sa dostaneme opäť k miestu i. Tento „začarovaný“ kruh prerušíme jednoducho. Niektorý prvok, napríklad **H[n]**, si odložíme do premennej **Posledny**. Teraz môžeme prvok **H[n-1]** presunúť do **H[n]**, **H[n-2]** do **H[n-1]**, atď, až **H[1]** do **H[2]** a na koniec prvok **H[n]** odložený v premennej **Posledny** do **H[1]** (obr 4.).



Obr. 4. Posun o jeden prvok bez pomocného poľa.

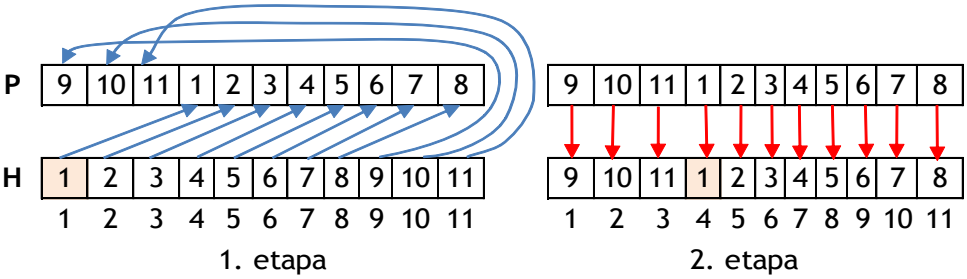
Riešenie posúvaním o jeden prvok bez pomocného poľa:
priradení: $2k(n+1)$
pamäť: konštantný počet

Koľko teraz potrebujeme priradení? Prvok **H[n]** priraďujeme dvakrát a všetky ostatné prvky poľa **H** raz. Spolu $n+1$ priradení a nepoužili sme pomocné pole. Pri posune o k miest postup opakujeme k krát.

2. riešenie

Pri 1. riešení s pomocným poľom nám asi okamžite prišlo na um, že prvky poľa **H** môžeme do poľa **P** v prvej etape zapisovať priamo na ich definitívnu pozíciu, ktorú majú mať v poli **H**. Potom ich už len prepíšeme z poľa **P** do poľa **H** (obr. 5).

Nemôže byť výsledkom rovno pole **P**? Ušetrili by sme tak n priradení pri prepisovaní **P** do **H**.



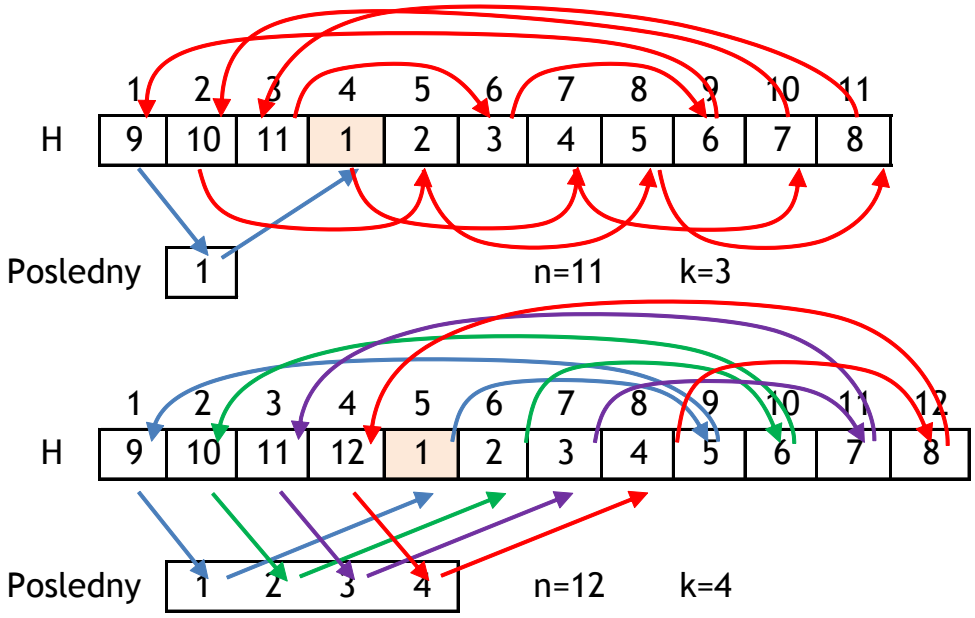
Obr. 5. Posun o $k = 3$ prvkov s použitím pomocného poľa. Modrou farbou je znázornená 1. etapa a červenou 2. etapa

Všimnite si, že oproti 1. riešeniu teraz nemusíme posúvanie opakovať. Vystačíme s $2n$ priradeniami a pomocnou pamäťou veľkosti n (pole **P**).

Riešenie posúvaním o k prvkov s pomocným poľom:
priradení: $2n$
pamäť: n

3. riešenie

Nedal by sa podobný trik, aký sme urobili v 2. riešení, použiť aj na úpravu toho spôsobu z 1. riešenia, v ktorom sme nepoužili pomocné pole? Teda snažiť sa presunúť prvok **H[i]** rovno na miesto $i + k$ v poli **H**. Pripomeňme si, že v 2. riešení sme si pri posúvaní odložili „posledný“ prvok do premennej **Posledny**. Keď ale posúvame o k prvkov, nie je vôbec očividné, ktorý prvok je „posledný“. Prvky posúvame predsa cyklicky, takže si môžeme pokojne odložiť prvok **H[i]** a na uvoľnené miesto dať prvok ktorý sa tam ma posunutím dostať. Tým sa nám uvoľní ďalšie miesto, na ktoré môžeme posunúť ďalší prvok, atď. až pokým sa nevrátime k pozícii i, kedy presunieme prvok odložený v premennej **Posledny**.



Obr. 6. Dva prípady posúvania poľa.

Na obr. 6. sú dva prípady posúvania. Všimnite si, že v prípade $n = 11$ a $k = 3$ sa nám podarilo všetky prvky poľa **H** „obísť“ na jeden raz. Do premennej **Posledny** sme si odložili hodnotu prvého prvku a červené šípky ilustrujú, ktorý prvok sa kam posunul. V prípade $n = 12$ a $k = 4$ sa nám nepodarilo „obísť“ všetky prvky na jeden raz. Prvý raz sme do premennej **Posledny** dali prvý prvok a modré šípky ilustrujú ktorý prvok sa kam posunul. Vidíme, že k prvému prvku sme sa dostali už pri treťom posúvaní. Takže do premennej sme museli odložiť postupne ešte aj druhý, tretí a štvrtý prvok. Jednotlivé posúvania sú znázornené rôznymi farbami, zelenou, fialovou a červenou. Závisí počet vzniknutých „cyklov“ nejako od hodnôt n a k? Áno, počet cyklov je $nsd(n,k)$ a ich dĺžka je $n/nsd(n,k)$. Koľko práce musí počítač pri tomto riešení vykonať? Všimnite si, že počet potrebných priradení je taký istý, ako počet šípok, lebo každá šípka znázorňuje jedno priradenie. Počet šípok zrátame ľahko, je ich toľko, koľko je „cyklov“ krát počet šípok v jednom cykle. Počet cyklov poznáme a počet šípok v cykle je o jedna viac, ako je jeho dĺžka. Teda spolu máme počet priradení (predpokladáme, že $n > k$):

$$nsd(n,k) \cdot \left(\frac{n}{nsd\,(n,k)} + 1\right) = n + nsd(n,k) \leq n + \frac{n}{2} = \frac{3}{2}n.$$

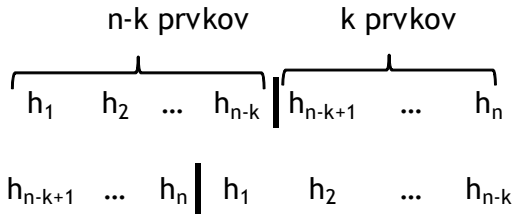
Riešenie posúvaním o k prvkov bez pomocného poľa:
priradení: $\frac{3}{2}n$
pamäť: konštantná

Pamäte potrebujeme okrem poľa **H** len konštantne veľa (nebude to závisieť od n ani od k). (Keby sme úlohu chceli naprogramovať začali by sa vynárať ďalšie detaily: aby sme určili počet cyklov, museli by sme vyrátať $nsd(n,k)$. Do potrebnej práce sme ale výpočet nsd nezapočítali. Asi je zrejme, že táto bude závisieť od n a k. Počet priradení potrebných na výpočet $nsd(n,k)$, keď $n > k$ je úmerný $log(n)$.)

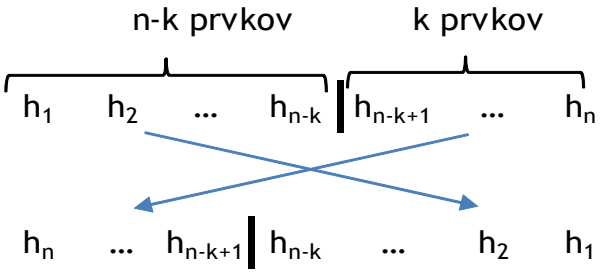
4. riešenie

Ďalší príklad toho, že šikovné označenie nám môže pomôcť.

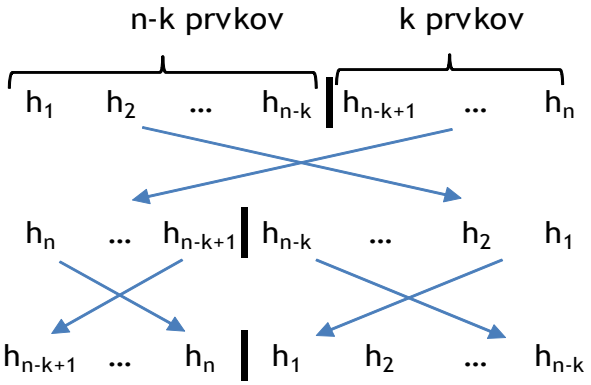
Skúsme si napísať ako vyzerá pole na začiatku a ako po cyklickom posune o k prvkov doprava. Namiesto **H[i]**, budeme písať len h_i.



Keď porovnáme počiatočnú situáciu zo situáciou po posunutí, zbadáme, že posledných k prvkov sa premiestnilo na začiatok a prvých $n-k$ prvkov zase na koniec poľa. To vieme ale dosiahnuť ľahko keď prvky v poli „zrkadlovo otočíme“, prvý vymeníme s posledným, druhý s predposledným, atď. Pozrime sa, čo vlastne takto dostaneme. Zrkadlové otočenie znázorníme šípkami.



Veď je to skoro to čo chceme, len prvú aj druhú časť treba tiež zrkadlovo otočiť! Takže dostaneme:



A sme hotoví. Koľko práce počítač vykoná pri tomto riešení? Koľko priradení sa vykoná pri riešení? Riešenie sme poskladali pomocou „operácie“ zrkadlového otočenia úseku poľa, určime teda koľko priradení sa vykoná pri jednom zrkadlovom otočení p prvkového úseku poľa. Na výmenu dvoch prvkov potrebujeme tri priradenia (dá sa to urobiť aj na dve) a v p prvkovom úseku pola je $p/2$ dvojíc. Spolu teda potrebujeme na zrkadlové otočenie p prvkového úseku poľa $3p/2$ priradení. Celková dĺžka úsekov ktoré otáčame je $2n$, takže spolu vykonáme $3(2n)/2 = 3n$ priradení. Okrem poľa **H** potrebujeme len pamäť konštantnej veľkosti.

Riešenie zrkadlovým otáčaním
priradení: $3n$
pamäť: konštantná

Čo sme sa naučili

V každom prípade, každý problém, ktorý vyriešime, rozšíri našu schopnosť riešiť ďalšie problémy v budúcnosti.

Že aj jednoduchá úloha môže mať viacero riešení. Jednotlivé riešenia sa môžu líšiť tým, koľko práce počítač potrebuje aby úlohu vyriešil a tým koľko pamäte potrebuje. Niekedy je ťažké rozhodnúť ktoré riešenie je lepšie, niektoré sa ľahšie naprogramuje, iné môže byť krajšie (vtipnejšie).

Čo sme sa naučili v tomto module

Zhrnutie

Tento modul zoznámil s

- dvojrozmerným poľom
- zoznamom
- radom a zásobníkom

Jednoduchou formou priblížil tvorbu jednoduchých algoritmov.

Preverenie výstupných vedomostí

Účastník vzdelávania vie použiť dvojrozmerné pole na naprogramovanie jednoduchej hry, ktorú budú hrať prostredníctvom počítača dvaja hráči-ľudia.

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

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 1

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 978-80-8118-034-7