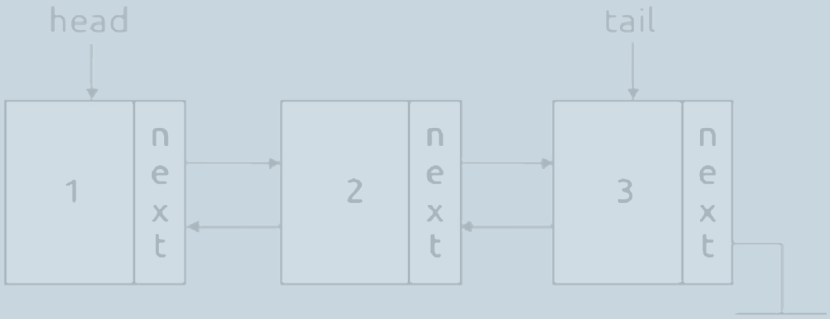
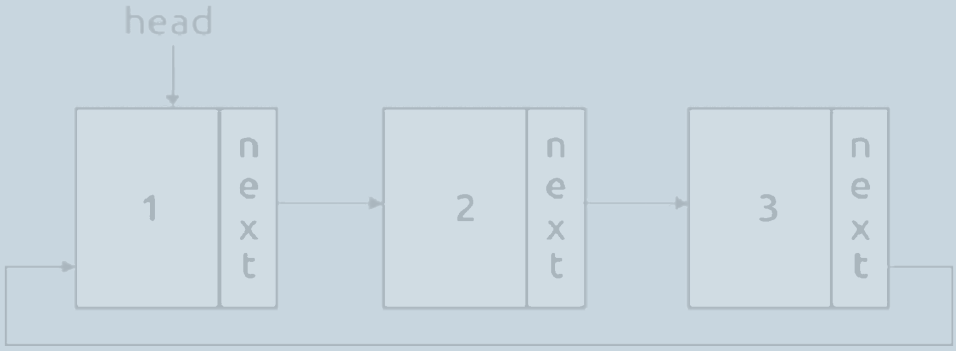
UNIVERZITA SV. CYRILA A METODA V TRNAVE
FAKULTA PRÍRODNÝCH VIED
Ústav počítačových technológií a informatiky

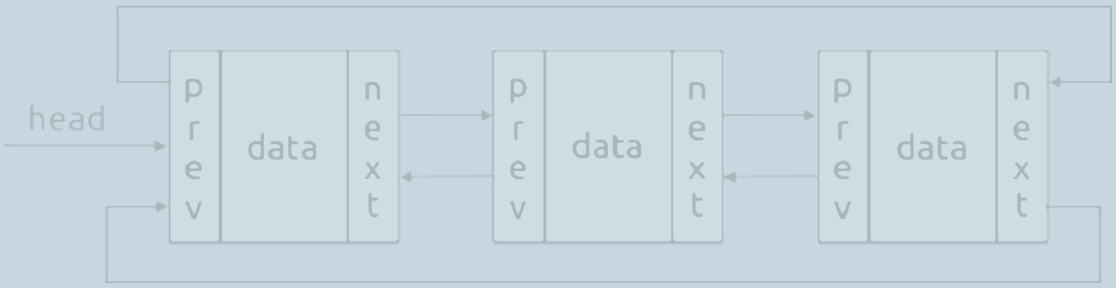


DÁTOVÉ ŠTRUKTÚRY. ABSTRAKTNÉ DÁTOVÉ TYPY.

Mierne pokročilé štruktúrované programovanie v jazyku C



Jana Jurinová



UNIVERZITA SV. CYRILA A METODA V TRNAVE

FAKULTA PRÍRODNÝCH VIED

Ústav počítačových technológií a informatiky

DÁTOVÉ ŠTRUKTÚRY. ABSTRAKTNÉ DÁTOVÉ TYPY. MIERNE POKROČILÉ ŠTRUKTÚROVANÉ PROGRAMOVANIE V JAZYKU C

Jana Jurinová

Trnava, 2023

Autor:

doc. Ing. Jana Jurinová, PhD.

Recenzenti:

prof. Ing. Pavel Važan, PhD.

Ing. Lukáš Herout, Ph.D.

Vysokoškolská učebnica bola schválená Edičnou radou Univerzity sv. Cyrila a Metoda v Trnave a vedením Fakulty prírodných vied UCM.

Vydané s podporou grantu KEGA 017UCM-4/2022 „Vývoj interaktívneho e-kurzu s využitím „SMART“ technológií na rozvoj algoritmického myslenia a programátorských zručností“.

Za odbornú a jazykovú stránku tejto vysokoškolskej učebnice zodpovedá autorka.

Rukopis neprešiel redakčnou ani jazykovou úpravou.

Vydavateľ: Univerzita sv. Cyrila a Metoda v Trnave, 2023

1. vydanie

ISBN 978-80-572-0401-5

Predhovor

Táto učebnica je určená predovšetkým študentom prvého a druhého ročníka študijného programu aplikovaná informatika, žiakom stredných škôl, ktorí chcú rozvíjať svoje vedomosti individuálne, ale aj všetkým tým, ktorí chcú preniknúť hlbšie do problematiky algoritmizácie a programovania v jazyku *C*. Učebnica vznikala aj ako súčasť e-kurzu, ktorý je jedným z cieľov projektu KEGA 017UCM-4/2022 s názvom: „Vývoj interaktívneho e-kurzu s využitím 'SMART' technológií na rozvoj algoritmického myslenia a programátorských zručností". Učebnica ponúka úvod do problematiky dátových štruktúr a abstraktných dátových typov a súvisiacich skutočností. V tejto učebnici venujeme pozornosť týmto dátovým štruktúram: statické jednorozmerné (*one-dimensional array*) a dvojrozmerné pole (*two-dimensional array*), dynamické jednorozmerné a dvojrozmerné pole, zreťazený zoznam, zásobník (*stack*) a front (*queue*). Tomu zodpovedá aj štruktúra a radenie jednotlivých kapitol.

Táto učebnica nezahŕňa kompletnú problematiku potrebnú k programovaniu v jazyku *C*. Preto odporúčame čitateľom, ktorí nemajú osvojené základné vedomosti a koncepty ohľadne algoritmizácie a programovania zamerané najmä na kľúčové poznatky, základné riadiace konštrukcie, jednoduché a štruktúrované dátové typy, prácu s funkciami, ako aj dynamickými premennými, aby siahli najprv po inej literatúre. Chýbajúce poznatky si môže záujemca doštudovať aj v prvom a druhom diele učebníc „Algoritmizácia a základy štruktúrovaného programovania v jazyku C. 1. diel" a „Algoritmizácia a základy štruktúrovaného programovania v jazyku C. 2. diel" vydaných autorkou v rokoch 2020 a 2021, ktoré sú voľne prístupné na: https://www.ucm.sk/sk/ucebne-texty-k-stiahnutiu/.

Učebnica vznikla zo súboru prednáškových materiálov, poznámok, skúseností, výskumných šetrení a spätnej väzby z praktických cvičení pri výučbe predmetov „algoritmy a dátové štruktúry I a II" a „programovanie I a II" študentov aplikovanej informatiky na bakalárskom stupni na Fakulte prírodných vied Univerzity sv. Cyrila a Metoda v Trnave. Spôsob výučby týchto predmetov sa vyvíjal v mnohých smeroch, čiastočne s cieľom zamerať sa na vedomostné zázemie prichádzajúcich študentov (nerozvinuté formálne zručnosti v predmetnej oblasti). V dôsledku toho boli témy starostlivo vybrané a zoskupené. Snahou nebolo vytvoriť encyklopedickú publikáciu, ale publikáciu, ktorá by rozvíjala nadobudnuté a osvojené vedomosti žiakov stredných škôl.

Učebnica vychádza v elektronickej podobe na základe dvoch faktorov. Učebnica je súčasťou e–kurzu, ktorý je dostupný online. Druhým dôvodom je snaha uľahčiť čitateľom prácu, preto ako súčasť učebnice poskytujeme zdrojové kódy všetkých riešených príkladov, vďaka ktorým

si môže čitateľ okamžite overiť ich funkčnosť a priamo s nimi pracovať. Odporúčame čitateľom, aby s týmito príkladmi experimentovali, dotvárali ich, rozširovali, modifikovali, pretože len aktívnym programovaním sa naučia a osvoja si požadované vedomosti a zručnosti. V každom kóde je vždy čo vylepšiť, prípadne zaujať iný postoj k riešeniu. Kódy programov je možné automaticky otvoriť v požadovanom softvéri po kliknutí na link, ktorý predstavuje jeho označenie. Pri uvádzaní komentárov v programe sú tieto výhradne uvádzané bez diakritiky. Treba zdôrazniť aj zastúpenie anglického jazyka vo výrazových prostriedkoch používaných v programovaní. Pre programátora je znalosť anglického jazyka prakticky nevyhnutná, aspoň na úrovni čítania a porozumenia dokumentácií. Preto väčšina slovenských kľúčových termínov použitých v učebnici je pre lepšie pochopenie interpretovaná aj v anglickom jazyku.

Ak by ste po štúdiu obsahu celej učebnice mali otázky, prípadne ak by ste objavili nejakú chybu, alebo by Vám nejaké informácie v tejto učebnici chýbali, neváhajte ma kontaktovať na jana.jurinova@ucm.sk.

Rada by som na tomto mieste poďakovala doc. Ing. Michalovi Čerňanskému, PhD., prof. Ing. Pavelovi Važanovi, PhD. a Ing. Lukášovi Heroutovi, Ph.D. za ich prínosné odborné komentáre, poznámky a poznatky pri tvorbe a recenzovaní tejto učebnice. Ich skúsenosti, postrehy a pripomienky prispeli k skvalitneniu informácií v tejto učebnici.

Október 2023 doc. Ing. Jana Jurinová, PhD., autorka učebnice

Obsah

1 Úvod do dátových typov *(data types)* a dátových štruktúr (*data structures*) v jazyku *C*

Dátové typy a dátové štruktúry sú dva základné pojmy v jazyku *C* a na prvý pohľad veľmi podobné, ale majú odlišné účely a charakteristiky. Základnú klasifikáciu dátových štruktúr môžeme kategorizovať do dvoch skupín:

- jednoduché dátové typy (*primitive data structures*),
- dátové štruktúry (*non-primitive data structures*).

Dátové typy definujú základné typy premenných, ktoré môžete používať v jazyku *C*. Jednoduché dátové typy (často v anglickej terminológii označované ako *primitive data types/structures*, ale tiež ako *simple data types*) sú definované v jazyku *C* a predstavujú atomické (ďalej už nedeliteľné) dátové typy, ktoré slúžia pre uloženie jednej hodnoty (*single value*). Napríklad pre ukladanie celočíselných hodnôt sa zvyčajne používa dátový typ *int*. Pre ukladanie reálnych čísiel, dátový typ *float* alebo *double*. Pre uloženie znaku, dátový typ *char*. Pre uloženie adresy premennej sa používa pointer, t. j. pointer ukazuje na adresu v pamäti, kde je uložená konkrétna hodnota.

Každý dátový typ definuje, aké hodnoty môžete do danej premennej uložiť. Tiež určujú, akým spôsobom sú dáta reprezentované v pamäti (akú veľkosť zaberajú) a aké operácie môžete s nimi vykonávať. Štruktúrované (niekedy označované aj ako zložené) dátové typy (*composite data types*) na rozdiel od jednoduchých (primitívnych) typov sú zložené z jednoduchých dátových typov alebo ich kombinácií. Môžu byť homogénne – všetky prvky sú rovnakého typu. Predstaviteľom homogénneho dátového typu je pole. Pole je sekvenčné (jeden prvok za druhým) zoskupenie prvkov. Alebo môžu byť heterogénne, napr. štruktúra a union, ktoré reprezentujú objekt zložený z dátových položiek rôzneho dátového typu ležiacich v bloku pamäte, pričom je zabezpečený koncept, ako pristupovať k jednotlivým členom štruktúrovaného dátového typu. Sú reprezentované jediným identifikátorom (premennou). Zoskupenie dát do logických celkov sprehľadňuje výsledný kód. Napríklad v prípade potreby uchovávania väčšieho počtu prvkov heterogénneho štruktúrovaného dátového typu *struct*, môžeme deklarovať pole štruktúr. Samotné pole týchto štruktúr je však opätovne len homogénnym dátovým typom.

Základný rozdiel medzi jednoduchými a štruktúrovanými dátovými typmi spočíva v tom, že jednoduché dátové typy definujú jednoduché typy premenných, zatiaľ čo štruktúrované dátové

typy umožňujú programátorovi definovať vlastné zložené typy, ktoré kombinujú viacero rôznych dátových typov do jedného objektu.

Dátové štruktúry (často v anglickej terminológii označované ako *non-primitive data structure*) sú organizované spôsoby ukladania a manipulácie s dátami v počítačoch. Sú to základné stavebné prvky programovania, ktoré umožňujú programátorom efektívne ukladať, vyhľadávať, aktualizovať a spracovávať dáta. Dátové štruktúry sú dôležité pre efektívne riešenie rôznych úloh a problémov v programovaní. Každá dátová štruktúra má svoje vlastnosti a použitie, a programátori si vyberajú tú, ktorá najlepšie vyhovuje konkrétnemu problému alebo úlohe, ktorú riešia. Porozumenie dátovým štruktúram je kľúčové pre efektívne programovanie a optimalizáciu výkonu programov. V jazyku *C* je možné implementovať rôzne dátové štruktúry pomocou štruktúr (*structs*) a dynamických alokačných funkcií ako *malloc()* a *calloc()*. Pre uvoľnenie alokovaného priestoru sa používa funkcia *free()*. V jazyku *C* musíte vytvoriť štruktúry, ktoré reprezentujú dátové štruktúry, a potom pracovať s nimi pomocou rôznych operácií, ako sú pridávanie, vyhľadávanie, aktualizácia a odstraňovanie prvkov, podľa konkrétnej dátovej štruktúry a úlohy, ktorú riešite. Pri implementácii dynamických dátových štruktúr, musíte byť opatrní pri práci s pamäťou a správne ju alokovať a uvoľňovať, aby ste zabránili únikom pamäte (tzv. *memory leaks*) a chybám v behu programu.

Existuje mnoho rôznych typov dátových štruktúr. Ich základná klasifikácia na základe ich štruktúry a usporiadania dát je členená do dvoch skupín:

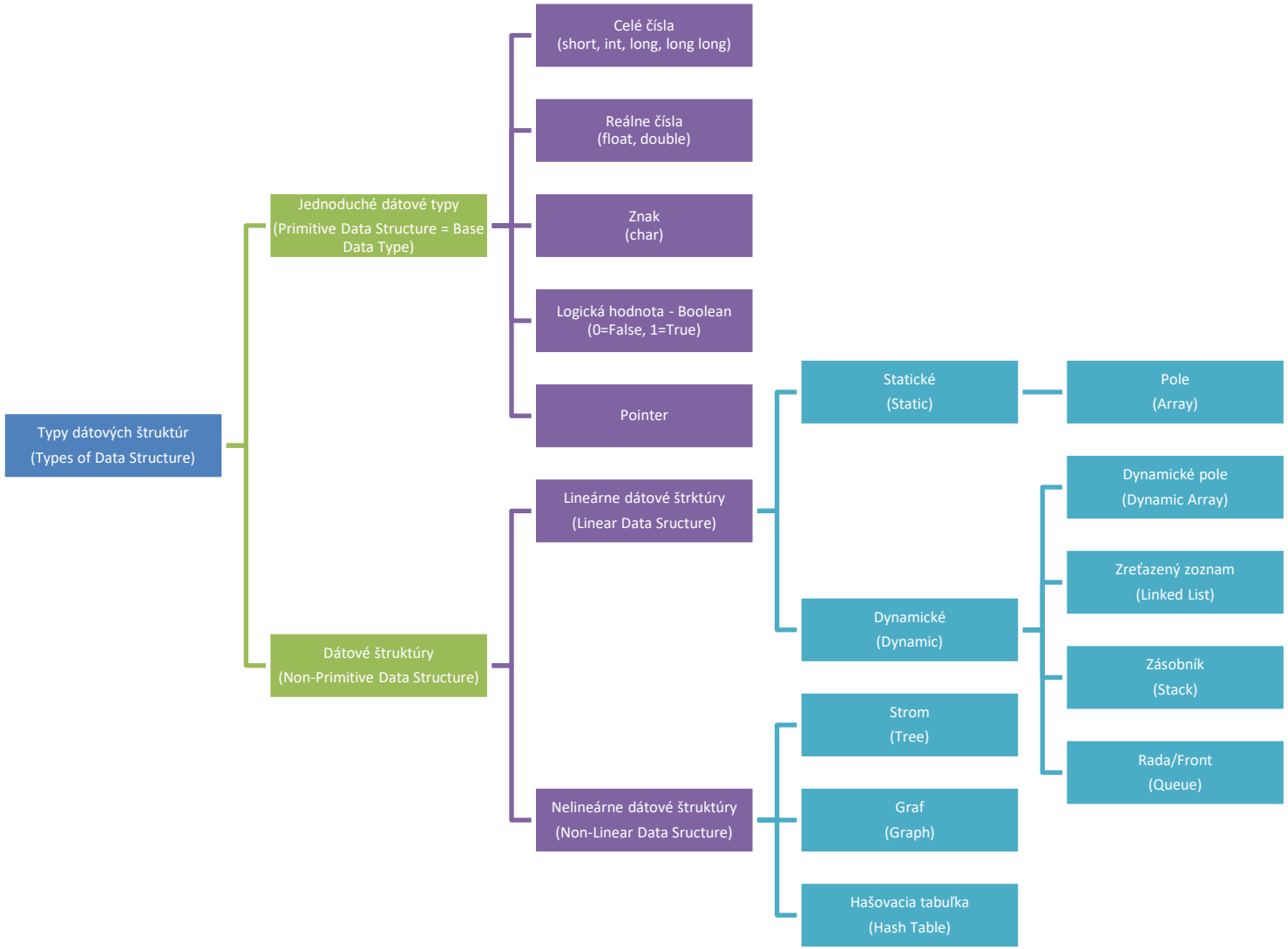
- Lineárne dátové štruktúry (*linear data structures*)
    - Statické dátové štruktúry (*static data structures*)
        - Pole (*array*)
    - Dynamické dátové štruktúry (*dynamic data structures*)
        - Dynamické pole (*dynamic array*)
        - Zreťazený zoznam (*linked list*)
        - Zásobník (*stack*)
        - Front (*queue*)
- Nelineárne dátové štruktúry (*non-linear data structures*)
    - Strom (*tree*)
    - Graf (*graph*)
    - Hašovacia tabuľka (*hash table*)

Lineárne dátové štruktúry v jazyku *C* sú dátové štruktúry, ktoré organizujú dáta do postupnosti, kde každý prvok má predchodcu a nasledovníka (s výnimkou prvého a posledného prvku). Statické dátové štruktúry majú pevnú veľkosť, ktorá je nemenná počas behu programu, na rozdiel od dynamických dátových štruktúr, kde je možné počas behu programu ich veľkosť modifikovať.

- **Pole** (*array*) je jednou zo základných lineárnych dátových štruktúr v jazyku *C*. Prvky sú uložené v pamäti za sebou a majú rovnaký dátový typ. Prístup k prvkom poľa sa uskutočňuje cez index. Ak je veľkosť poľa určená pri deklarácii, táto sa nemôže meniť počas behu programu. Vtedy hovoríme o statickom poli. Dynamické pole môže meniť svoju veľkosť počas behu programu. V niektorých literatúrach je táto dátová štruktúra označovaná ako zoznam (*list*).
- **Zreťazený zoznam** (*linked list*) je dynamická lineárna dátová štruktúra. Každý prvok, nazývaný uzol, obsahuje dáta a odkaz na nasledujúci uzol. V takomto prípade ide o jednosmerne zreťazený zoznam. Existujú rôzne typy zreťazených zoznamov, vrátane jednosmerných, obojsmerných a kruhových (cyklických) zoznamov. Z toho vyplýva, že zreťazený zoznam môžeme prechádzať iba postupne, t. j. prechádzať prvok po prvku. Na rozdiel od poľa, prvky zreťazeného zoznamu nemusia ležať v pamäti za sebou. Z toho vyplýva možnosť jednoducho dynamicky pridávať a odstraňovať uzly zreťazeného zoznamu.
- **Zásobník** (*stack*) je lineárna dátová štruktúra fungujúca na princípe "*last in, first out*" (LIFO), t.j. prvky vložené ako posledné, budú vybraté (odstránené) ako prvé. Inými slovami, prvky sú pridávané a odstraňované z vrcholu zásobníka, t. j. v opačnom poradí ako boli vložené.
- **Front** (*queue*) je lineárna dátová štruktúra fungujúca na princípe "*first in, first out*" (FIFO), t. j. prvky vložené ako prvé, budú vybraté (odstránené) ako prvé. Inými slovami, prvky sa pridávajú na koniec fronty a odoberajú sa z jej začiatku.

Nelineárne dátové štruktúry v jazyku *C* sú dátové štruktúry, ktoré organizujú dáta do zložitejších vzťahov, ktoré nevyhovujú lineárnej postupnosti. Tieto štruktúry sa používajú na reprezentáciu vzájomných vzťahov medzi dátami, ktoré môžu byť hierarchické, sieťové alebo úplne náhodné.

- **Strom** (*tree*) je nelineárna dátová štruktúra, ktorá sa skladá z uzlov spojených hranami. Každý uzol má minimálne jedného potomka, ale môže mať aj viacero potomkov (detí), ale iba jedného rodiča (okrem koreňového uzla). Uzol, ktorý nemá potomkov sa nazýva list. Stromy sa často používajú na reprezentáciu hierarchických dát. Rozoznávame viacero typov stromov, ale jedným z najčastejšie používaných je binárny vyhľadávací strom (*binary search tree*) alebo hromada (*heap*). Hromada = halda je stromová štruktúra. V hromade je každý prvok väčší alebo rovný svojim potomkom. To sa nazýva vlastnosť hromady (*heap property*) a delí sa na dve formy:
    - Max-hromada: Všetky uzly majú väčšiu hodnotu alebo rovnakú, ako ich potomkovia. Najväčší prvok je v koreni stromu.
    - Min-hromada: Všetky uzly majú menšiu hodnotu alebo rovnakú, ako ich potomkovia. Najmenší prvok je v koreni stromu.
- **Graf** (*graph*) je nelineárna dátová štruktúra, ktorá pozostáva z vrcholov a hrán, ktoré spájajú tieto vrcholy. V grafoch môže existovať viacero typov vzťahov medzi vrcholmi, a to aj cyklické vzťahy na rozdiel od dátovej štruktúry strom. Grafy sa používajú na reprezentáciu rôznych vzťahov a sieťových štruktúr.
- **Hašovacia tabuľka**, známa aj pod označením hašovacia mapa (*hash table*/*hash map*) je nelineárna dátová štruktúra, ktorá umožňuje rýchle vyhľadávanie hodnôt na základe kľúčov. Používa sa hašovacia funkcia na mapovanie kľúčov na indexy v tabuľke. Hašovacie tabuľky sú často využívané na rýchle vyhľadávanie a indexovanie dát, pretože operácia hľadania, vloženia a vymazania prvku sa vyznačuje konštantnou asymptotickou časovou zložitosťou O(1).



Obr. 1 Klasifikácia typov dátových štruktúr.

1.1 Jednorozmerné statické pole (*one-dimensional static array)*

Herout (2010) definuje jednorozmerné pole, ako štruktúrovaný dátový typ zložený z prvkov rovnakého typu, prístupný pod spoločným identifikátorom, tzv. homogénny štruktúrovaný dátový typ. Prvky poľa sú vždy rovnakého typu, pričom typ môže byť jedným z jednoduchých (*int*, *char*, *float*, *double*), ale aj štruktúrovaných (štruktúra, union, pole a pod.). Pozíciu položky v poli určuje index. Prvky poľa sa v jazyku *C* vždy indexujú od 0. Indexy poľa sa nikdy automaticky v jazyku *C* nekontrolujú. Názov poľa je zároveň smerníkom (ukazovateľom) na prvý prvok poľa. **Syntax:**

```
TYP pole[pocet];        // pole je pocet-prvkové pole typu TYP
```

príklad staticky alokuje blok pamäte pre *pocet* prvkov typu *TYP*, pričom rozsah indexov je od *0* po *pocet − 1*. Ak predpokladáme, že hodnota *pocet* je známa v čase prekladu, tak by to mal byť konštantný výraz – často sa preto používa konštanta. Ak je pole definované týmto spôsobom, ide o statické pole.

Avšak hodnota *pocet* nemusí byť vždy známa v čase prekladu, t. j. v prípade ak *pocet* nie je konštantný výraz, ale premenná. Vtedy ide o tzv. *variable lenght array*, kde sa pamäť pre pole alokuje o veľkosti premennej *pocet*, avšak až počas spustenia programu.

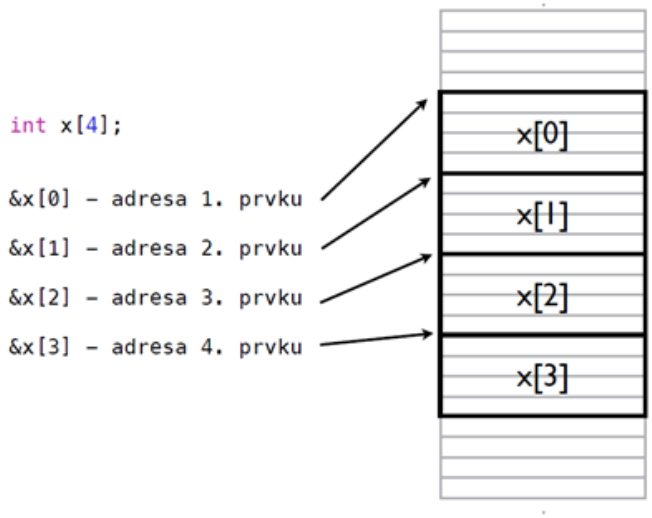
Pole možno inicializovať pri deklarácii. To nie je to isté ako priradenie hodnoty. Definícia inicializovaného poľa nemusí obsahovať veľkosť poľa (prekladač ho dopočíta z inicializácie). Pri rozdielnom počte inicializátorov a veľkosti poľa platí:

- ak veľkosť poľa > počet prvkov: inicializuje sa len začiatok, zvyšok ostáva nedefinovaný,
- ak veľkosť poľa < počet prvkov: chyba (príliš veľa inicializátorov).

Príklad:

```
int pole1[5] = { 2, 4, 6, 8, 10 };     // inicializovane 5 prvkove pole
int pole2[] = { 2, 4, 6, 8, 10 };      // inicializovane 5 prvkove pole
int pole3[5] = { 0 };                  // inicializovane 5 prvkove pole, prvky su 0
int pole4[10] = { 2, 4, 6, 8, 10 };    // poslednych 5 miest sa inicalizuje na 0
int pole5[4] = { 2, 4, 6, 8, 10 };     // chyba, nezmesti sa to tam
float pole6[5] = { 1.2, 2.5, 8.4, 6.0, 4.7 };
double pole7[] = { 1.2, 2.5, 8.4, 6.0, 4.7 };
```

Jednorozmerné pole sa dá inak nazvať vektor (*premenna(a1, a2, a3, ...)*). Graficky môžeme pole v pamäti reprezentovať, ako jeden súvislý blok pamäte, ako ilustruje obrázok 2.



Obr. 2 Jednorozmerné pole v pamäti.

Skutočnú veľkosť, ktorú zaberá pole v pamäti vieme zistiť v jazyku *C* pomocou operátora *sizeof()*, ktorý vracia veľkosť v bajtoch. V tomto prípade sa *sizeof(x)* vyhodnotí na **16 B**.

Jednorozmerné statické pole môžeme inicializovať viacerými spôsobmi:

- Inicializáciou jednotlivých prvkov poľa
- Inicializačným zoznamom – v deklaračnom príkaze môžeme vynechať špecifikáciu rozsahu poľa
- Cyklom
- Inicializáciu poľa môžeme prenechať používateľovi programu

```c
#include <stdio.h>
int main(void)
{
  int pole[5];
  pole[0] = 1;
  pole[1] = 2;
  pole[2] = 3;
  pole[3] = 4;
  pole[4] = 5;
  return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
  int pole[5] = {1, 2, 3, 4, 5};
  return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
  int pole[5];
  int i;
  for(i = 0; i <= 4; i++)
    pole[i] = i+1;
  return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
  int i, j, pole[5];
  for(i = 0; i <= 4; ++i)
   {
   printf("Zadajte %d. cislo: ", i+1);
   scanf("%d", &pole[i]);
   }
return 0;
}
```

1.2 Dvojrozmerné statické pole (*two-dimensional static array)*

Dvojrozmerné pole sa dá prirovnať k tabuľke alebo k matici. Tabuľka aj matica je špecifikovaná riadkami a stĺpcami. Preto dva rozmery, ktoré sa zadávajú pri deklarácii dvojrozmerného poľa, sa dajú interpretovať ako riadok a stĺpec. Syntax deklarácie dvojrozmerného poľa:

```
TYP pole[riadky][stlpce];
```

Technicky je možná aj deklarácia:

```
TYP pole[stlpce][riadky];
```

Záleží od používateľa ako sa na danú maticu pozrie. Odporúčame prvý spôsob, avšak v prípade ak by používateľ zvolil druhý, odporúčame zachovávať rovnaký spôsob práce s maticami v celom programe. Je viac než nevhodné meniť koncept v rámci jedného programu.

Inicializácia viacrozmerného, t. j. aj dvojrozmerného poľa môže byť realizovaná podobne ako pri jednorozmernom poli. Riadky môžu/nemusia byť v samostatnej zátvorke. Nepovinný je len prvý rozmer. Druhý a ďalšie (v prípade viacrozmerných polí) musia mať určenú veľkosť. Príklady:

```c
int matica[2][3] = { 5,2,7,8,6,3 };
int matica[2][3] = { {5,2,7}, {8,6,3} };
// pocet riadkov sa urci podla poctu inicializatorov
int matica[][3] = { {5,2,7}, {8,6,3}, {5,7,6}, {2,8,7} };
```

Dvojrozmerné statické pole môžeme inicializovať podobnými spôsobmi, akými sme inicializovali jednorozmerné statické pole:

- Inicializáciou jednotlivých prvkov poľa
- Inicializačným zoznamom
- Vnoreným cyklom

```c
#include <stdio.h>
int main(void)
{
  int pole[2][3];
  pole[0][0] = 1;
  pole[0][1] = 2;
  pole[0][2] = 3;
  pole[1][0] = 4;
  pole[1][1] = 5;
  pole[1][2] = 6;
  return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
  int pole[2][2] = {
     {1, 2, 3},
     {4, 5, 6} };
  return 0;
}
```

```c
#include <stdio.h>
int main(void)
{
  int pole[2][3];
  int i, j, x = 1;
  for(i = 0; i < 2; i++) {
   for(j = 0; j < 3; j++){
       pole[i][j] = x++;
   }
}
  return 0;
}
```

Pozn. autora: V prípade načítavania prvkov pomocou vnoreného cyklu si môžeme všimnúť, že prvky matice sa načítavajú po riadkoch. Vhodnou zmenou je možné realizovať načítavanie aj po stĺpcoch. Ako som uviedla vyššie, technicky to možné je, ale nie je to konvenčné riešenie. Ak sa však autor tak rozhodne, treba dodržiavať túto konvenciu v celom programe, t. j. nemiešať pri načítavaní prvkov matice prístup, že raz sa budú načítavať po riadkoch a druhýkrát po stĺpcoch. Je zrejmé, že naplnenia poľa je možné ponechať aj používateľovi.

1.3 Dynamické lineárne dátové štruktúry

Dynamické údajové štruktúry sú štruktúry, ktoré sa vytvárajú a rozširujú za behu programu, na rozdiel od statických údajových štruktúr, ktoré sa deklarujú a alokujú počas kompilácie. Tieto dynamické štruktúry sa spravujú pomocou alokácie pamäte za behu programu, najčastejšie pomocou funkcií, ktoré slúžia na správu pamäte, ako je napríklad *malloc()* (alokácia pamäte) a *free()* (uvoľnenie pamäte) v jazyku *C*. Dôležité je pamätať na to, že pri používaní dynamických údajových štruktúr je potrebné správne spravovať pamäť, najmä uvoľňovať ju po jej použití, aby sa predišlo únikom pamäte (*memory leaks*) a iným chybám spojených s pamäťou.

Dynamické údajové štruktúry umožňujú prispôsobiteľnosť a flexibilitu programu, pretože ich veľkosť a štruktúra môže byť určená počas behu programu v závislosti od vstupných dát alebo iných faktorov. To môže byť užitočné, ak potrebujeme pracovať s veľmi veľkými dátovými množinami, ktoré nemusia byť známe vopred, alebo ak sa môžu meniť počas vykonávania programu.

Vzhľadom na to, že ukazovatele sú nevyhnutným prostriedkom aj pre implementáciu dynamických údajových štruktúr, venujeme im v tejto učebnici náležitú pozornosť. Pri spracovaní sa opierame najmä o zdroje ako Belan (2011), Herout (2010), Palmarová (2003) a Horovčák a Podlubný (1997). Správnym a vhodným používaním ukazovateľov môžeme riešenie mnohých, niekedy na prvý pohľad komplikovaných problémov zjednodušiť a zefektívniť. V niektorých situáciách sa použitie ukazovateľa dokonca vyžaduje a nemožno ho obísť.

Všetky používané premenné môžeme podľa doby ich trvania kategorizovať do dvoch skupín:

- **Statické premenné**

  Vznikajú vyhradením miesta kompilátorom počas prekladu a trvajú po celý čas vykonávania programu. Sú to premenné, ktoré majú presne definovanú štruktúru, ktorá je nemenná počas celého výpočtu. To znamená, že majú presne stanovený rozsah hodnôt, ktoré táto premenná môže nadobúdať a tiež majú stanovený pevný počet pamäťových miest potrebných na ich reprezentáciu.

- **Dynamické premenné**

  Vznikajú a zanikajú podľa potreby počas vykonávania programu. Sú to premenné, ktorých štruktúra sa počas výpočtu mení. Avšak, elementárne zložky týchto štruktúr sú na určitom stupni detailnosti statické. Nedefinujú sa deklaráciou, ale vzniknú aj zaniknú počas vykonávania sa programu pomocou špeciálnych príkazov (v jazyku *C* zvyčajne pomocou štandardných knižničných funkcií *malloc()* a *free()*). Dynamické premenné teda nemožno zaviesť v úseku deklarácií/definícií, a zabezpečiť prístup k ich hodnotám pomocou ich identifikátorov. Aby bolo možné sa k nim dostať, zavádza sa dátový typ ukazovateľ (smerník, pointer), ktorý ukazuje na premennú, ktorá je uložená v pamäti.

Pointer (smerník = ukazovateľ) je premenná, v ktorej je uložená adresa pamäte a na tejto adrese sa až nachádza príslušný objekt (hodnota premennej, prvok poľa, inštancia štruktúry a pod.) (Herout, 2010). Smerník je ako šípka, ktorá ukazuje niekam do pamäte. Okrem toho, kam smerník ukazuje, treba mu väčšinou určiť aj to, aký typ premennej tam môže očakávať. Takže sú smerníky, ktoré sú typu *int* pretože ukazujú na premenné typu *int*, sú smerníky typu *float*, typu *char* a pod. Ďalšou dôležitou skutočnosťou je, že pri pointeroch musíme pracovať len s pamäťou, ktorá je naša. V jazyku *C* nič túto skutočnosť nekontroluje.

Deklarácia ukazovateľa

Realizuje sa podobne, ako deklarácia inej premennej, ale použije sa znak hviezdičky '*'. Je nutné určiť typ, na ktorý ukazovateľ môže ukazovať. Často sa používa pred identifikátorom takejto premennej prefix *p_*, zdôrazňujúci skutočnosť, že ide o premennú typu pointer, napr.:

```
double *p_d;          // ukazovatel na double
int *p_i;             // ukazovatel na int
char *p_c;            // ukazovatel na char
```

Deklarácia spolu s viacerými premennými

```
int *p_i, p_j;        // casta chyba, len p_i je pointer, premenna p_j je typu int
```

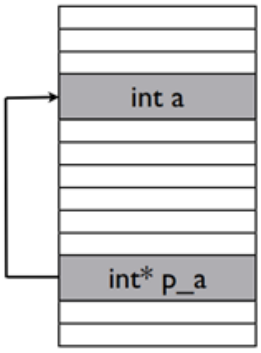
Inicializácia ukazovateľa

Inicializácia ukazovateľa *p_i* adresou premennej *i* sa realizuje pomocou referenčného operátora '&'. V prípade, ak by sme pointer neinicializovali môžeme hovoriť o tzv. zablúdenom pointery (*wild pointer*).

```
int a, *p_a = &a;      // pri deklaracii
```

alebo

```
int a, *p_a;
p_a = &a;              // v programe
```



Obr. 3 Grafická reprezentácia pointera a jeho inicializácie.

Práca s ukazovateľmi

Ak pointer *p_a* je inicializovaný premennou *a*, tak hodnotu premennej *a* pomocou pointera môžeme vypísať:

```
int a, *p_a = &a;
a = 3;
printf("Ukazovatel p_n ukazuje na hodnotu %d.", *p_a);
```

Z uvedeného príkladu je zrejmé, že operátor referencie je '&' a dereferencovania je '*'.

Ak pointer *p_a* nie je pri deklarácii aj inicializovaný, ukazuje na náhodné miesto v pamäti.

```
int *p_a;
printf("%d", *p_a);  // chyba - nevieme kam p_a ukazuje
```

Ak do *p_a* zapíšeme literál, určili sme adresu kam *p_a* ukazuje. Takéto priradenie nie je samozrejme správne. Pretože vieme, že adresa v pamäti je reprezentovaná v 16-kovej (hexadecimálnej) číselnej sústave a tiež, že priraďovať napevno pamäťové úseky nie je vhodné.

```
int *p_a;
p_a = 10;       // chyba - p_a ukazuje na miesto v pamati s adresou 10
```

Príklad nižšie ilustruje rozdielne priradenia pri práci s pointermi:

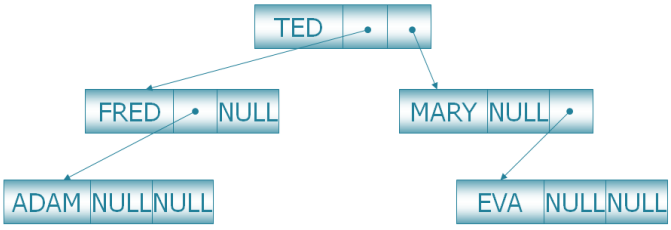
```
int m = 2, n = 4, *p1 = &m, *p2 = &n;
p1 = p2;        // kopirovanie adresy z p2 do p1
*p1 = *p2;      // kopirovanie hodnoty z pamate kam ukazuje p2 do pamate, kam ukazuje p1
```

Príklad nižšie ilustruje konverziu pointerov:

```
char znak = 'A', *p_znak = &znak;
int i, *p_i = &i;
p_znak = p_i;              // nevhodne pouzitie
p_znak = (char *) p_i;     // odporucane pouzitie
```

Príkladom použitia smerníkov v reálnej praxi, môže byť napríklad potreba reprezentácie rodokmeňovej štruktúry. Táto bude reprezentovaná prostredníctvom jednotlivých, pravdepodobne nespojitých záznamov. Pre každú osobu bude existovať práve jeden záznam, reprezentovaný napríklad štruktúrou *osoba*. Tieto osoby sú v rámci štruktúry pospájané pomocou adries priradených položkám *otec* a *matka*. Túto situáciu možno najlepšie graficky zobraziť pomocou šípok, t. j. smerníkov. Pre smerník, ktorý nikam neukazuje sa používa hodnota *NULL* – prázdny smerník. V tomto prípade to reprezentuje koniec dátovej štruktúry.

```
typedef struct osoba
{
    char meno[20];
    struct osoba *otec;
    struct osoba *matka;
}OSOBA;
```



Obr. 4 Reprezentácia rodokmeňovej štruktúry.

1.3.1 Dynamické prideľovanie a navracanie pamäte

Najpoužívanejšou funkciou pre prideľovanie pamäte v jazyku *C* je funkcia, deklarovaná v *stdlib.h*, ktorej prototyp je:

```
void *malloc(size_t size)
```
ktorá:

- Alokuje súvislý blok pamäte požadovanej veľkosti (*size_t* = *unsigned int*) (tento parameter udáva, koľko bajtov chceme alokovať) a vráti generický ukazovateľ na tento blok. Tento pointer je vhodné pretypovať na pointer zodpovedajúceho typu.
- Pamäť sa alokuje na halde (hromada, heap), nie na zásobníku, až počas vykonávania sa programu.
- Ak nie je v pamäti dosť miesta pre pridelenie požadovanej časti, vracia funkcia hodnotu *NULL*. Pri každom prideľovaní pamäti, sa odporúča testovať návratovú hodnotu na *NULL* a nespoliehať sa na fakt, že pamäti je dostatok.

Ukážka použitia *malloc()* vrátane reakcie na prípadný neúspech:

```
int *p_i;
if( (p_i = (int *) malloc(1000)) == NULL)
{
    printf("Nedostatok pamate. Program konci.\n");
    exit(1);
}
```

Pozn. autora: V tomto prípade išlo o deklaráciu bloku o veľkosti 1 000 B. Napriek tomu, že syntakticky je táto alokácia správna, v praxi bežne nepoužívame alokáciu priestoru pomocou pevne stanovej veľkosti, ale využívame operátor *sizeof()*. Toto použitie si ukážeme neskôr. V prípade, ak by nebolo dostatok pamäte, dôjde k ukončeniu programu pomocou funkcie *exit()*, predtým je však používateľ náležite informovaný o tom, čo sa deje.

Uvoľnenie (vrátenie pamäti) je opačná akcia než prideľovanie. Platí, že nepotrebnú pamäť je dobré okamžite vrátiť a nečakať až na koniec programu. Pre uvoľnenie pamäte sa používa funkcia `void free(void *)`, ktorej parametrom je pointer na typ *void*, ktorý ukazuje na začiatok skôr prideleného bloku pamäte. Dôležité je, že *free()* nemení hodnotu svojho parametru. To znamená, že pointer stále ukazuje na to isté miesto v pamäti, takže je možné s touto pamäťou ďalej pracovať, aj keď už programu nepatrí. To môže spôsobiť veľa problémov, preto po príkaze:

```
free((void *) p_i);
```

je vhodné uviesť aj príkaz:

```
p_i = NULL;
```

čím zabránime možnému prístupu do uvoľnenej pamäte, ako aj zabránime problému, ktorý by mohol nastať pri opakovanom aplikovaní funkcie *free()* na ten istý smerník za sebou bez pridelenia novej pamäte.

1.3.1.1 Únik pamäte, tzv. memory leak

Pri dynamickom prideľovaní a uvoľňovaní pamäte často dochádza k nasledovným chybám:

Majme deklaráciu dvoch pointerov:

```
char *p_c;
int *p_i;
```

Potom príkaz:

```
*p_c= 'a';
```

nie je úplne korektný, pretože *p_c* ukazuje niekam do pamäte, ktorú nemáme pridelenú. Pred týmto príkazom je teda nutné uviesť príkaz, aby sme alokovali dostatočný priestor pamäte:

```
p_c = malloc(1);
```

Aby sme dodržiavali odporúčania, je vhodné uviesť explicitnú typovú konverziu:

```
p_c = (char *) malloc(1);
```

Aby sme dodržiavali všetky odporúčania, je treba ešte zistiť, či sa požadovanú pamäť podarilo priradiť:

```
if(((char *)p_c = malloc(1)) == NULL)
{
        printf("Nieje volna pamat. \n");
        return;
}
```

Ak chceme následne alokovať pamäť o veľkosti 20 B prístupného pomocou rovnakého pointera *p_c*, tak príkaz:

```
p_c = (char*) malloc(20);
```

nie je úplne vhodný pretože takto sme stratili pointer na skôr alokovanú pamäť (v ktorej je znak 'a'). Túto pamäť sa už nikdy nepodarí uvoľniť a do konca programu bude znak 'a' visieť niekde v pamäti. Preto pred každou ďalšou alokáciou je potrebné pamäť uvoľniť:

```
free(p_c);
```

Ak potrebujeme alokovať pamäť pre uloženie napr. *char* hodnoty, tak je najvhodnejší nasledovný príkaz:

```
p_i = (int *) malloc(sizeof(char));
```

t. j. využitie operátora *sizeof()*.

Ak používame definíciu vlastných typov s využitím operátora *typedef*, je možné, že pri alokovaní pamäte sa stretneme aj s nasledujúcim spôsobom alokovania:

```
typedef int * P_INT;
P_INT p_i;
p_i = (P_INT) malloc(sizeof(int));
```

alebo

```
p_i = (P_INT) malloc(sizeof(*p_i));
```

1.3.1.2 Funkcia calloc()

Okrem funkcie *malloc()* môžeme pamäť dynamicky alokovať aj prostredníctvom funkcie *calloc()*. Je deklarovaná v *stdlib.h* a jej prototyp vyzerá takto:

`void* calloc (size_t pocet, size_t velkost)`, kde:

- *pocet* predstavuje počet prvkov dynamického poľa,
- *velkost* je nominálna alokačná kapacita jedného prvku dynamického poľa (v bajtoch).

Ak je volanie funkcie *calloc()* úspešné, funkcia alokuje pamäťový blok pre dynamické pole, pričom jednotlivé prvky poľa inicializuje na 0. Ak nie je úspešné, funkcia vráti nulový smerník (*NULL*). Návratovou hodnotou funkcie *calloc()* je generický smerník (*void* *) ukazujúci na začiatok dynamicky alokovanej pamäte. Použitie funkcie *calloc()* ilustruje nasledovný fragment kódu, ktorý alokuje priestor pre *n* prvkov typu *int*:

```
...
printf("Kolko cisel chces zadat? ");
scanf("%d", &n);
p = (int*) calloc(n, sizeof(int));
if(p == NULL){
    printf("Nedostatok pamatovych prostriedkov.\n");
    exit(1);
}
...
```

1.3.1.3 Príklad – dynamické prideľovanie a uvoľňovanie pamäte

Zadanie problému: Vytvorte program, ktorý pomocou funkcie nájde maximálny prvok zadaného poľa. Funkciu navrhnite tak, aby bolo možné zisťovať maximálny prvok aj z častí daného poľa.

Pozn. autora: Môžete si všimnúť že zadanie problému nehovorí o tom, či pracujeme so statickým, alebo dynamickým poľom. Je tomu tak z dôvodu, že ak si uvedomujeme skutočnosť, že by sme mali s pamäťou narábať hospodárne, nikto nám nemusí exaktne hovoriť, že je vhodné pracovať s dynamickým poľom. V tomto prípade sme využili pre alokovanie pamäte pre pole konštantu *N*, ale

ide len o ilustráciu. Je zrejmé, že konštanta môže byť nahradená premennou určujúcou veľkosť poľa od používateľa.

Objasnenie príkladu autorom:

V tomto príklade pracujeme s dynamickým poľom reálnych prvkov, ktoré reprezentujeme premennou *p_pole* deklarovanou na riadku 9, ako pointer na dátový typ *double*. Veľkosť poľa určuje konštanta *N,* definovaná na riadku 3. Táto je využitá pri dynamickej alokácii pamäte, ktorá sa realizuje na riadkoch 13 – 18 spolu s testovaním, či sa podarilo pamäť alokovať. V prípade ak nie, program je ukončený, pretože nemáme s akým blokom pamäte pracovať, ktorý by predstavoval pole. Do úvahy prichádza aj použitie funkcie *calloc(),* ale v tomto prípade je to zbytočné, keďže obsah tohto poľa je napĺňaný používateľom a nezáleží na tom, aký bude obsah poľa po alokácii. Celé pole napĺňame na riadkoch 21 – 25 od používateľa. Pre prístup k prvkom poľa využívame pointerovú aritmetiku, t. j. k prvkom poľa pristupujeme cez pointer, ktorý postupne navyšujeme o hodnotu premennej *i*, čím zabezpečíme postupný prechod cez všetky prvky poľa. Na riadkoch 28 – 31 vypisujeme obsah poľa, čo môžeme chápať ako kontrolu, či pole obsahuje prvky, ako sme zadali. Môžete si všimnúť, že tu sme využili indexovú notáciu pre prístup k prvkom poľa. Dôvodom je len ilustrácia jednotlivých možností. Volanie funkcie *max()*, ktorá je jadrom tohto programu je obsiahnuté na riadku 33 v rámci funkcie *printf().* Takto nie je nutné deklarovať žiadnu pomocnú premennú do ktorej by sme uložili výsledok funkcie *max()*. Vzhľadom na odovzdávané argumenty môžeme sledovať, že sa snažíme hľadať maximum len na úseku poľa. Buď môžeme opätovne využiť pointerovú aritmetiku, alebo odovzdať priamo adresu prvku ležiaceho na konkrétnom indexe, čím sa určí hranica prehľadávanej časti poľa. Pred ukončením samotného programu, na riadkoch 36 a 37 je realizované uvoľnenie pamäte pre pole.

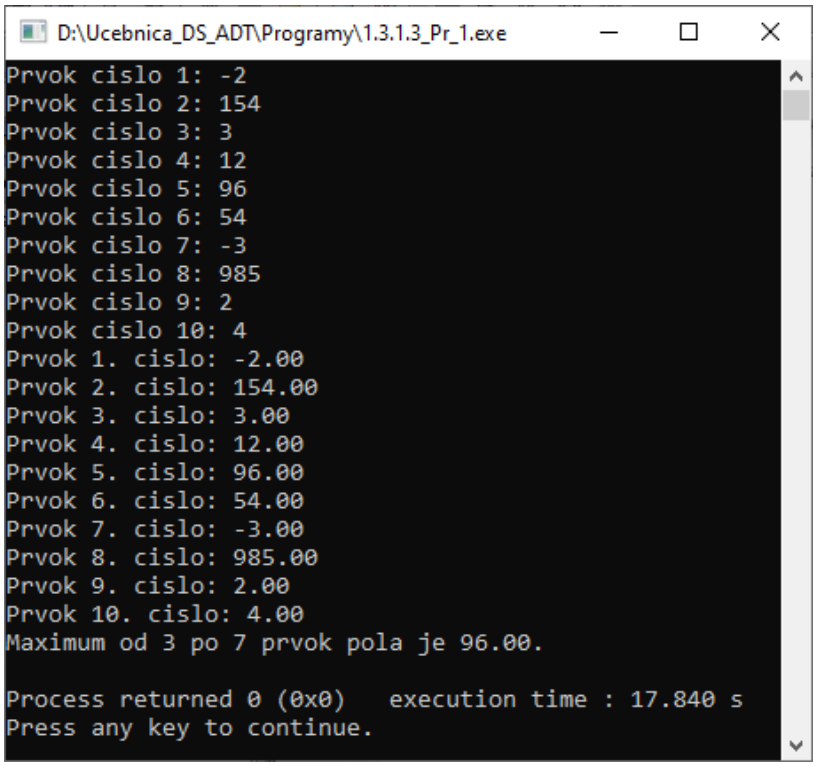
Samotná funkcia pre nájdenie výskytu maximálneho prvku v poli je navrhnutá tak, aby bola schopná pracovať s rôznymi úsekmi poľa. Preto parametre funkcie odovzdané odkazom sú adresy prvkov, ktoré predstavujú začiatok a koniec úseku poľa na ktorom sa má hľadať maximum. Algoritmus hľadania extrému je postavený na myšlienke, že prvý prvok prehľadávaného bloku považujeme za maximum a v cykle prehľadávame zvyšok poľa, či sa tam nenachádza prvok väčší, ako náš predpoklad. V prípade ak áno, aktualizujeme hodnotu premennej *maximum*. V tomto príklade pracujeme s hodnotou maxima. Samozrejme, že v prípade najmä budúceho rozšírenia programu a zisťovania napríklad počtu výskytov maxím a pod. by bolo vhodnejšie si zapamätávať index, na ktorom maximum leží, a nie jeho hodnotu. Pre posuv po prvkoch obdobne používame pointerovú

aritmetiku. Využívame operáciu pripočítavania celého čísla k pointeru, ako aj porovnávanie dvoch pointerov. V prípade ak čitateľ má isté rezervy v kontexte pointerovej aritmetiky, odporúčame aby siahol po našej učebnici (2. diel, kapitola 4.3) resp. po inom zdroji.

Jedna z možných implementácií 1.3.1.3_Pr_1.c:

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10

double max(double *zaciatok_pola, double *koniec_pola);     // deklaracia funkcie

int main(void)
{
    double *p_pole;    // deklaracia pointra reprezentujuceho pole
    int i;

    // dynamicka alokacia pola
    p_pole = (double *) malloc(N*sizeof(double));
    if(p_pole == NULL)
    {
        printf("Pole sa nepodarilo vytvorit. Nedostatok pamate. Program konci.\n");
        return 1;
    }

    // naplnenie pola
    for(i = 0; i < N; i++)
    {
        printf("Prvok cislo %d: ", i + 1);
        scanf("%lf",(p_pole + i));
    }

    // vypis pola
    for(i = 0; i < N; i++)
    {
        printf("Prvok %d. cislo: %.2lf\n", i + 1, p_pole[i]);
    }

    printf("Maximum od 3 po 7 prvok pola je %0.2lf. \n", max(p_pole + 2, &p_pole[7]));

    // uvolnenie alokovanej pamate
    free(p_pole);
    p_pole = NULL;
    return 0;
}

// definicia funkcie pre najdenie maxima
double max(double *zaciatok_pola, double *koniec_pola)
{
    double maximum = *zaciatok_pola, *p_pom;    // inicializacia max na prvy prvok pola

    // prehladavame uz len zvysok pola
    for (p_pom = zaciatok_pola + 1; p_pom < koniec_pola; p_pom++)
```

```
48      {
49          if (*p_pom > maximum)
50              maximum = *p_pom;
51      }
52      return maximum;
53  }
```



Obr. 5 Konzolový výstup programu 1.3.1.3_Pr_1.c.

1.3.1.4 Dynamická realokácia

Realokácia znamená zmenu veľkosti (zväčšenie alebo zmenšenie) dynamicky alokovaného pamäťového bloku, ktorý bol pridelený alokačnými funkciami *malloc()* resp. *calloc()*. Realokáciu dynamicky alokovanej pamäte uskutočňuje funkcia *realloc()*, ktorá je deklarovaná v *stdlib.h*. Prototyp funkcie *realloc()* vyzerá takto:

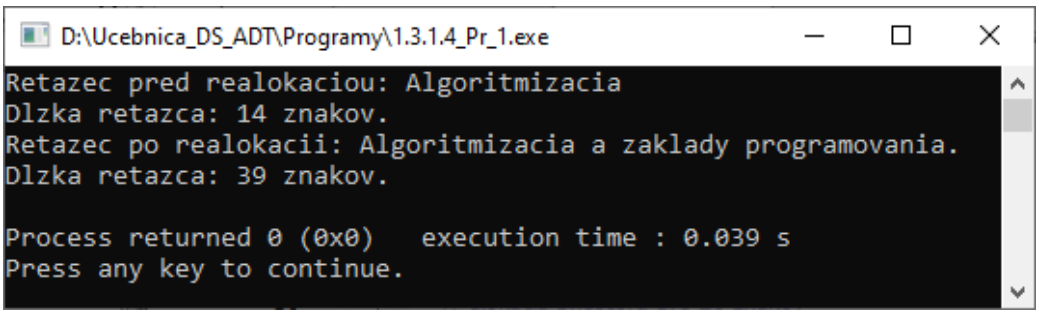
`void * realloc(void * PamatovyBlok, size_t velkost)`, kde:

- *PamatovyBlok* je generický smerník na už dynamicky alokovanú pamäťovú oblasť.
- *velkost* predstavuje novú alokačnú kapacitu pamäťového bloku v bajtoch.

Ak je dynamická realokácia úspešná, návratovou hodnotou funkcie *realloc()* je generický smerník na realokovaný pamäťový blok. Ak nie je dostatok dodatočného pamäťového priestoru, funkcia *realloc()* vracia nulový smerník (*NULL*).

Ukážka použitia funkcie *realloc()* na príklade 1.3.1.4_Pr_1.c:

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

int main (void)
{
    // deklaracia pointra na char
    char * p;

    // alokacia priestoru pre 20 znakov
    if((p = (char *) malloc(20*sizeof(char))) == NULL)
    {
        printf("Nedostatok pamate. Program konci.\n");
        return 1;
    }

    // naplenie priestoru retazcom
    strcpy(p, "Algoritmizacia");
    printf("Retazec pred realokaciou: %s\n", p);
    printf("Dlzka retazca: %d znakov.\n", strlen(p));

    // realokacia priestoru pre 40 znakov
    if((p = (char *) realloc(p, 40)) == NULL)
    {
        printf("Nedostatok pamate. Program konci.\n");
        free(p);    // uvolnenie predtym alokovanej pamate
        p = NULL;
        return 1;
    }

    // doplnenie existujuceho retazca o dalsi
    strcat(p, " a zaklady programovania.");
    printf("Retazec po realokacii: %s\n", p);
    printf("Dlzka retazca: %d znakov.\n", strlen(p));

    free(p);
    p = NULL;

    return 0;
}
```



Obr. 6 Konzolový výstup programu 1.3.1.4_Pr_1.c.

1.3.2 Jednorozmerné dynamické pole

Chápeme ním dynamickú údajovú štruktúru – kontajner (úložisko dát), ktorá automaticky prispôsobuje veľkosť poľa v prípade pokusu o vloženie nového prvku pri zaplnenej kapacite. Zväčšenie kapacity dynamického poľa môže byť vykonané pre viac položiek, alebo iba jednu položku. Zväčšovanie kapacity pre jednu položku je málo efektívne, keďže je nutné vykonávať túto operáciu pri každom pridaní novej položky – pričom na pozadí dochádza k prekopírovaniu celého obsahu poľa. Z dôvodu šetrenia pamäťou, pri odstraňovaní položiek môže byť zabezpečené uvoľnenie prebytočnej pamäte. Záleží však od kontextu, ako sa pole využíva, pretože ak sa využíva, ako kontajner pevnej veľkosti len málokedy dochádza k zmenšeniu jeho kapacity.

Jedným z variantov pre šetrenie pamäte je ten, že pri vkladaní nových položiek sa cyklicky prechádza pole a hľadá sa voľné – prázdne miesto. Ak také neexistuje, až vtedy dochádza k zväčšeniu kapacity poľa. Takáto implementácia je však možná len v prípade, ak vieme pri odstraňovaní prvkov určiť jednoznačný identifikátor, ktorý signalizuje, že dané miesto je prázdne. Tento spôsob nie je príliš využívaný. Pri odstraňovaní prvku častejšie dochádza k posuvu prvkov ležiacich za odstraňovaným prvkom, t. j. posuv doľava.

Z uvedeného vyplýva, že najčastejšími operáciami nad dynamickým poľom sú:

- Vyhraď pamäť pre oblasť nového poľa. Kapacita (veľkosť) poľa môže byť určená používateľom, alebo môže byť inicializovaná priamo programátorom na východiskovú hodnotou, ktorá logicky vyplynula zo špecifikácie problému. Niekedy sa kapacita a veľkosť chápu ako jeden a ten istý údaj, niekedy však možno chápať kapacitu (*capacity*) ako počiatočnú veľkosť úložiska a veľkosť (*size*), ako reálny počet prvkov v poli.
- Vloženie položky na pozíciu *pozicia* v poli. Najjednoduchšie pridávanie je na koniec poľa. V takomto prípade pomocnú premennú *pozicia* inicializujeme väčšinou hodnotou 0 a po každom úspešnom vložení prvku ju inkrementujeme. Prípadne ju môžeme inicializovať hodnotou −1 a inkrementovať pred vložením prvku.
- Ak je aktuálny počet prvkov + 1 > kapacita poľa => zväčši kapacitu poľa. Ako sme naznačili vyššie, funkcionalita zväčšovania kapacity je úplne závislá od špecifikácie problému.
- Odstránenie položky z poľa na pozícii *index*, alebo inej (určenej kľúčovou hodnotou). Operácia odstraňovania v poli je značne komplikovaná, pretože vyžaduje reorganizáciu položiek. V niektorých prípadoch môže viesť aj k zmene kapacity poľa. Jedným z univerzálnych spôsobov riešenia je reorganizácia položiek pomocou operácie:

- Urob posun položiek v poli od pozície *index*, prípadne od kľúčovej položky (t. j. tá ktorá sa ide odstraňovať) na pozíciu o 1 menšiu, t. j. posun vľavo. Ak sme však realizovali vkladanie cez pomocnú premennú *pozicia*, treba túto premennú následne dekrementovať, aby sme neporušili funkcionalitu vkladania prvkov. Iná situácia by samozrejme bola v prípade, ak by sme proces vkladania jednotlivých položiek riešili cyklickým prehľadávaním voľných miest.

- V prípade budovania usporiadaného dynamického poľa sa často stretáme s operáciou:
  - Urob posun položiek v poli od pozície *index* na pozíciu o 1 väčšiu, t. j. posun vpravo, z dôvodu vytvorenia miesta pre vkladaný prvok, aby sme budovali usporiadané pole.

Výhodou dynamického poľa oproti statickému je, že kapacita poľa nie je ohraničená pri jeho vytvorení. Nevýhodou statického poľa oproti dynamickému je tiež skutočnosť, že sa preň rezervuje pamäť, ktorá nemusí byť nikdy využitá.

Medzi problémové operácie nad dynamickým poľom zaraďujeme:

- Vkladanie položky na ľubovoľnú pozíciu inú než začiatok a koniec poľa – nutná reorganizácia položiek poľa, t. j. posun vľavo/vpravo.
- Odstránenie položky z ľubovoľnej pozície inej ako je začiatok a koniec poľa – vznik voľných miest v poli, resp. reorganizácia poľa.

1.3.2.1 Príklad – dynamické neusporiadané pole

Zadanie problému: Vytvorte program, ktorý bude poskytovať možnosť ukladania celých čísiel do dynamického poľa. Tento bude slúžiť ako kontajner, čiže jeho kapacita, ktorá bude na začiatku zadaná používateľom, sa používateľom už meniť nebude. V prípade potreby vloženia prvku pri nedostatočnej kapacite sa táto automaticky zväčší (zdvojnásobí).

Pozn. autora: Medzi základné operácie, ktoré sa pri použití kontajnerov používajú, patrí vloženie prvku, odstránenie prvku, zobrazenie obsahu, vyhľadanie konkrétnej položky, ale aj možnosť zistenia, či je kontajner plný, prázdny, aká je jeho kapacita a pod. V ilustračnom príklade som implementovala len niektoré funkcionality. Ostatné funkcionality, respektíve vylepšenie existujúcich ponechávame na čitateľa.

Objasnenie príkladu autorom:

V tomto príklade pracujeme s dynamickým poľom celočíselných prvkov, ktorý reprezentujeme premennou *p_pole* deklarovanou na riadku 14, ako pointer na dátový typ *int*. Pole budeme plniť postupne od začiatku a na signalizáciu práve obsadzovaného miesta deklarujeme premennú *pozicia*, ktorú inicializujeme hodnotou 0. Ako vieme, prvý prvok v poli sa nachádza na indexe 0. Po každom vložení budeme túto premennú inkrementovať.

Určenie veľkosti tohto kontajnera nechávame na používateľovi (riadky 17 – 21). Načítanie veľkosti poľa ošetrujeme pomocou cyklu s podmienkou na konci, tak aby používateľ mohol zadať len kladnú celočíselnú hodnotu. Následne sa na riadkoch 23 – 27 postaráme o alokovanie pamäte pomocou funkcie *malloc()* o danej veľkosti a otestujeme či sa podarilo pamäť alokovať. Do úvahy prichádza aj použitie funkcie *calloc()*, ale v tomto prípade je to zbytočné, keďže obsah tohto kontajnera bude napĺňaný používateľom a nezáleží na tom, aký bude obsah poľa po alokácii. Na riadkoch 29 – 79 je implementovaná ponuka pre používateľa, aby mohol interaktívne s programom komunikovať.

```
23      if((p_pole = (int *) malloc(sizeof(int) * velkost)) == NULL)
24      {
25          printf("Nedostatok pamate. Program konci.\n");
26          return 1;
27      }
```

Pre vloženie prvku sme definovali procedúru *vlozPrvok()*, ktorá má štyri parametre. Prvý parameter, odovzdaný hodnotu reprezentuje hodnotu prvku, ktorú chceme do poľa vložiť. Ostatné parametre sú odovzdané odkazom, pretože chceme, aby sa všetky zmeny realizované v procedúre prejavili aj na skutočných parametroch odovzdaných procedúre. Parameter *pozicia* signalizuje index na ktorý ideme prvok vložiť. V main-e sme túto premennú definovali. Ako sme uviedli, po každom vložení budeme túto premennú inkrementovať, aby vždy signalizovala správne miesto, kde chceme prvok vkladať. Tretí parameter reprezentuje pole do ktorého prvok chceme pridať. Ako posledný parameter tejto procedúry je *velkost*, ktorá reprezentuje veľkosť poľa. V prípade pokusu o vloženie prvku do už zaplneného kontajnera, riešime túto situáciu zväčšením priestoru dvojnásobne pomocou funkcie *realloc()*. V tele procedúry ako prvé otestujeme, či je miesto v kontajneri. To zistíme porovnaním hodnoty premennej *pozicia*, ktorá reprezentuje reálny počet prvkov v poli, a premennej *velkost*, ktorá reprezentuje jeho maximálnu veľkosť (riadok 92). V prípade ak je v poli miesto, vložíme prvok na index určený premennou *pozicia* a túto inkrementujeme (riadok 94). V opačnom prípade sa postaráme o zväčšenie priestoru na dvojnásobok pôvodnej veľkosti. Vtedy je potrebné modifikovať hodnotu v premennej *velkost* (riadok 105), aby bola zachovaná správna funkčnosť ďalších operácií

s kontajnerom a tiež sa postarať o vloženie prvku a inkrementáciu premennej *pozicia* (riadok 106). O jednotlivých stavoch používateľa patrične informujeme. Volanie procedúry je na riadku 43 v tvare:

```
    vlozPrvok(prvok, &pozicia, p_pole, &velkost);
```

```
89  // vlozenie prvku,ak sa presiahne velkost pola, tak sa tato automaticky zdvojnasobi
90  void vlozPrvok(int prvok, int *pozicia, int *p_pole, int *velkost)
91  {
92      if(*pozicia < *velkost)
93      {
94          p_pole[(*pozicia)++] = prvok;
95      }
96      else
97      {
98          if((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
99          {
100             printf("Nedostatok pamate. Program konci.\n");
101             return;
102         }
103         else
104         {
105             (*velkost) *= 2;
106             p_pole[(*pozicia)++] = prvok;
107             printf("Prvok bol vlozeny po zvacseny kontajnera.\n");
108         }
109     }
110 }
```

Pre odstránenie prvku sme definovali funkciu, ktorá sa postará o odstránenie prvku na základe posuvu prvkov, keďže žiadna iná možnosť odstraňovania v tomto prípade neprichádza do úvahy. Definovali sme pole pre všetky celočíselné hodnoty, takže neexistuje prvok, ktorý by sme vedeli vylúčiť zo zdrojových dát kontajnera a ktorý by sme mohli použiť na signalizáciu prázdneho miesta. Takáto implementácia má v najhoršom prípade lineárnu asymptotickú časovú zložitosť O(n). Cez návratovú hodnotu funkcie signalizujeme stav, či sa prvok podarilo alebo nepodarilo odstrániť. Na jej základe v main-e informujeme používateľa (riadky 52 a 53). Je logické, že operáciu odstraňovania a volanie príslušnej funkcie realizujeme len v prípade ak sa v poli nejaké prvky nachádzajú. Na overenie tohto stavu by sme mohli definovať funkciu *isEmpty()*. My sme zvolili jednoduchú kontrolu na základe obsahu premennej *pozicia*. Ak je hodnota premennej *pozicia* rovná 0, tak to signalizuje stav, že je pole prázdne (riadky 46 a 47). Funkcia *odstranPrvok()* preberá tri parametre. Prvý parameter reprezentuje hodnotu prvku, ktorý chceme odstrániť. Ako druhý parameter odovzdávame odkazom hodnotu premennej *pozicia*, aby sme po odstránení túto mohli dekrementovať a zmeny sa aplikovali aj na skutočnú premennú. Ako posledný parameter odovzdávame pole. V tele funkcie použitím cyklu *for* prechádzame postupne prvky poľa od začiatku až po hodnotu premennej *pozicia*. Nemá zmysel

pole prehľadávať až po koniec, pretože premenná *pozicia* reprezentuje reálny počet vložených prvkov do poľa, ktorý sa môže líšiť vzhľadom na maximálnu možnú veľkosť poľa. Na riadku 118 testujeme zhodu prvku v poli s hodnotou prvku, ktorý chceme odstrániť. Ak sa zhoda nájde, pomocou *for* cyklu, ktorý začína na indexe odstraňovaného prvku a končí poslednou zaplnenou pozíciou sa postaráme o posun všetkých prvkov ležiacich za odstraňovaným prvkom smerom doľava. Tým dôjde k prepísaniu hodnoty odstraňovaného prvku a posunu zvyšných prvkov. Nezabudneme dekrementovať premennú *pozicia* (riadok 124). Je zrejmé, že takouto implementáciou odstraňujeme prvý výskyt daného prvku. V prípade, ak by sme chceli odstrániť všetky prvky s touto hodnotou, je nutná modifikácia konceptu tejto funkcie.

```
112 // funkcia odstranuje prvy vyskyt zadaneho prvku, ak nedojde k odstranovaniu vrati 0
113 int odstranPrvok(int prvok, int *pozicia, int *p_pole)
114 {
115     int i, j;
116     for(i = 0; i < *pozicia; i++)
117     {
118         if(prvok == p_pole[i])
119         {
120             for(j = i; j < *pozicia; j++)
121             {
122                 p_pole[j] = p_pole[j + 1];
123             }
124             (*pozicia)--;
125             printf("Prvy vyskyt prvku %d bol odstraneny.\n", prvok);
126             return 1;
127         }
128     }
129     return 0;
130 }
```

Pre vypísanie obsahu poľa sme definovali procedúru *vypisPola()*, ktorá ako parametre preberá pole a hodnotu premennej *pozicia*. Tá je tentokrát odovzdaná hodnotou, pretože slúži len na určenie konca cyklu, ktorým sa staráme o vypísanie obsahu kontajnera. Keďže sme pole alokovali pomocou funkcie *malloc()* a nemáme teda vyčistený daný pamäťový priestor, v prípade ak by sme cyklus nastavili až po hodnotu veľkosti poľa, mohlo by dôjsť k vypísaniu klamlivého, resp. nezmyselného obsahu, ktorý nie je reálnym obsahom kontajnera. Pre prístup k prvom poľa sme využili pointerovú aritmetiku, ktorú je možné nahradiť indexovou notáciou.

```
132 void vypisPola(int *p_pole, int pozicia)
133 {
134     int i;
135     printf("Prvky pola: ");
136     for(i = 0; i < pozicia; i++)
137         printf("%d ", *(p_pole + i));
138     printf("\n");
139 }
```

Program zahŕňa aj funkciu pre sekvenčné vyhľadávanie. Ide o jednoduchý algoritmus vyhľadávania, ktorý slúži na nájdenie konkrétneho prvku v poli postupným prehľadávaním jednotlivých prvkov. Tento algoritmus porovnáva hľadanú hodnotu s každým prvkom v poli, pokiaľ nenájde zhodu. Ak sa zhoda nájde, funkcia vráti index tohto prvku. Ak hľadaná hodnota nie je v poli, sekvenčné vyhľadávanie prejde celé pole a nedosiahne zhodu. V takomto prípade tento stav signalizujeme hodnotou -1. Tento algoritmus je jednoduchý na implementáciu, avšak jeho asymptotická časová zložitosť je O(n), kde "*n*" je veľkosť poľa. To znamená, že čas vyhľadávania závisí od počtu prvkov v poli a pri veľkých poliach môže byť sekvenčné vyhľadávanie pomalé. Používanie sekvenčného vyhľadávania sa odporúča pre malé polia alebo v prípadoch, keď je vysoká pravdepodobnosť, že hľadaná hodnota sa nachádza na začiatku poľa. Pri veľkých poliach alebo častých vyhľadávaniach sa odporúča použiť efektívnejšie algoritmy, ako napríklad binárne vyhľadávanie. To však vyžaduje usporiadané pole. Z uvedeného je zrejmé, že funkcia pre svoje fungovanie potrebuje tri parametre. Potrebuje poznať pole v ktorom daný prvok hľadáme, jeho hodnotu, ako aj premennú *pozicia*, ktorou určujeme pokiaľ má cyklus prehľadávať.

```
141  // vrati -1 ak sa prvok nenajde, inak vrati jeho index
142  int sekvencneVyhladanie(int *p_pole, int pozicia, int prvok)
143  {
144      int i;
145      for(i = 0; i < pozicia; i++)
146      {
147          if (p_pole[i] == prvok)
148              return i;
149      }
150      return -1;
151  }
```

Na riadkoch 82 a 83 je realizované uvoľnenie pamäte pre pole pred samotným ukončením programu.

Jedna z možných implementácií 1.3.2.1_Pr_1.c:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   // deklaracie funkcii
5   void vlozPrvok(int prvok, int *pozicia, int *p_pole, int *velkost);
6   int odstranPrvok(int prvok, int *pozicia, int *p_pole);
7   void vypisPola(int *p_pole, int pozicia);
8   int sekvencneVyhladanie(int *p_pole, int pozicia, int prvok);
9
10  int main (void)
11  {
12      // deklaracia pointra reprezentujuceho pole a dalsich premennych
13      int pozicia = 0;
```

```
14      int *p_pole, velkost, volba, odpoved = 0, prvok, i, index;
15
16      // alokacia priestoru (kapacity pola) na zaklade volby pouzivatela
17      do
18      {
19          printf("Ake velke ulozisko si zelas vytvorit?: \n");
20          scanf("%d", &velkost);
21      } while(velkost <= 0);
22
23      if((p_pole = (int *) malloc(sizeof(int) * velkost)) == NULL)
24      {
25          printf("Nedostatok pamate. Program konci.\n");
26          return 1;
27      }
28      // menu
29      do
30      {
31          printf("1 - vlozenie prvku do pola.\n");
32          printf("2 - odstranenie prvku z pola.\n");
33          printf("3 - vypisanie obsahu pola.\n");
34          printf("4 - hladanie v poli.\n");
35          printf("5 - koniec programu.\n");
36
37          scanf("%d", &volba);
38          switch(volba)
39          {
40          case 1:
41              printf("Aky prvok chces vlozit do kontajnera?:\n");
42              scanf("%d", &prvok);
43              vlozPrvok(prvok, &pozicia, p_pole, &velkost);
44              break;
45          case 2:
46              if(pozicia == 0)
47                  printf("Tvoj kontajner je prazdny, nemas co odstranovat.\n");
48              else
49              {
50                  printf("Aky prvok chces odstranit z kontajnera?:\n");
51                  scanf("%d", &prvok);
52                  if(odstranPrvok(prvok, &pozicia, p_pole) == 0)
53                      printf("Taky prvok sa v kontajnery nenachadza.\n");
54              }
55              break;
56          case 3:
57              if(pozicia == 0)
58                  printf("Tvoj kontajner je prazdny, nemam co vypisat.\n");
59              else
60                  vypisPola(p_pole, pozicia);
61              break;
62          case 4:
63              if(pozicia == 0)
64                  printf("Tvoj kontajner je prazdny, nemam kde vyhladavat.\n");
65              else
66              {
67                  printf("Aky prvok chces vyhladat v kontajnery?:\n");
68                  scanf("%d", &prvok);
```

```
69                  if((index = sekvencneVyhladanie(p_pole, pozicia, prvok)) == -1)
70                      printf("Taky prvok sa v kontajnery nenachadza.\n");
71                  else
72                      printf("Index hladaneho prvku %d je: %d.\n",prvok, index);
73              }
74              break;
75          default:
76              odpoved = 1;
77          }
78      }
79      while(!odpoved);
80
81      // uvolnenie alokovanej oblasti
82      free(p_pole);
83      p_pole = NULL;
84
85      return 0;
86  }
87
88  // definicie funkcii
89  // vlozenie prvku,ak sa presiahne velkost pola, tak sa tato automaticky zdvojnasobi
90  void vlozPrvok(int prvok, int *pozicia, int *p_pole, int *velkost)
91  {
92      if(*pozicia < *velkost)
93      {
94          p_pole[(*pozicia)++] = prvok;
95      }
96      else
97      {
98          if((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
99          {
100             printf("Nedostatok pamate. Program konci.\n");
101             return;
102         }
103         else
104         {
105             (*velkost) *= 2;
106             p_pole[(*pozicia)++] = prvok;
107             printf("Prvok bol vlozeny po zvacseny kontajnera.\n");
108         }
109     }
110 }
111
112 // funkcia odstranuje prvy vyskyt zadaneho prvku, ak nedojde k odstranovaniu vrati 0
113 int odstranPrvok(int prvok, int *pozicia, int *p_pole)
114 {
115     int i, j;
116     for(i = 0; i < *pozicia; i++)
117     {
118         if(prvok == p_pole[i])
119         {
120             for(j = i; j < *pozicia; j++)
121             {
122                 p_pole[j] = p_pole[j + 1];
123             }
```

```
124             (*pozicia)--;
125             printf("Prvy vyskyt prvku %d bol odstraneny.\n", prvok);
126             return 1;
127         }
128     }
129     return 0;
130 }
131
132 void vypisPola(int *p_pole, int pozicia)
133 {
134     int i;
135     printf("Prvky pola: ");
136     for(i = 0; i < pozicia; i++)
137         printf("%d ", *(p_pole + i));
138     printf("\n");
139 }
140
141 // vrati -1 ak sa prvok nenajde, inak vrati jeho index
142 int sekvencneVyhladanie(int *p_pole, int pozicia, int prvok)
143 {
144     int i;
145     for(i = 0; i < pozicia; i++)
146     {
147         if (p_pole[i] == prvok)
148             return i;
149     }
150     return -1;
151 }
```

Pozn. autora: Vzhľadom na to, že pracujeme s neusporiadaným poľom, nemôžeme využiť vyhľadávací algoritmus s nižšou časovou zložitosťou, ktorým je binárne vyhľadávanie. Jeho časová zložitosť je logaritmická *O(log n)*, na rozdiel od sekvenčného vyhľadávania, ktoré pracuje s lineárnou časovou zložitosťou *O(n)*. Binárne vyhľadávanie je teda veľmi efektívne a rýchle, najmä pre veľké usporiadané polia.

1.3.2.2 Príklad – dynamické usporiadané pole

Zadanie problému: Vytvorte program, ktorý bude poskytovať možnosť usporiadaného ukladania celých čísiel do dynamického poľa. Tento bude slúžiť ako kontajner. To znamená, že jeho kapacita, ktorá bude na začiatku zadaná používateľom, sa používateľom už meniť nebude. V prípade potreby vloženia prvku pri nedostatočnej kapacite sa táto automaticky zväčší (zdvojnásobí).

Pozn. autora: Voči príkladu v kapitole 1.3.2.1 nastala úprava funkcie pre vkladanie prvku, ako aj zmena funkcie pre vyhľadanie prvku. V tomto prípade sme implementovali efektívnejšie binárne vyhľadávanie.

Objasnenie príkladu autorom:

Binárne vyhľadávanie v poli využíva vlastnosť usporiadania prvkov v poli a postupne zužuje vyhľadávací rozsah až kým nájde hľadanú hodnotu. Začína sa určením stredu, pre ktorý je potrebné určiť začiatočný a koncový index vyhľadávacieho rozsahu. Začiatočný index je zvyčajne 0 a koncový index je dĺžka poľa mínus 1. Výpočet stredového indexu sa určí podľa vzťahu (začiatočný index + koncový index) / 2 (riadok 144). Následne sa porovná hodnota na stredovom indexe s hľadanou hodnotou (riadok 145). Ak sa hodnoty zhodujú, vyhľadávanie sa ukončí a stredový index je pozíciou hľadanej hodnoty v poli (riadok 146). Ak je hodnota na stredovom indexe menšia ako hľadaná hodnota, vyhľadávanie pokračuje v pravom podpoli (stredový index + 1 až koncový index), riadok 148. Ak je hodnota na stredovom indexe väčšia ako hľadaná hodnota, vyhľadávanie pokračuje v ľavom podpoli (začiatočný index až stredový index - 1), riadok 149. Tento proces sa opakuje až kým sa nájde hľadaná hodnota alebo sa vyhľadávací rozsah zúži na jednoprvkový interval.

```
139  // vrati -1 ak sa prvok nenajde, inak vrati jeho index
140  int binarneVyhladanie(int *p_pole, int sh, int hh, int kluc)
141  {
142      if (hh < sh)
143          return -1;
144      int stred = (sh + hh) / 2;  // int stred = sh + (hh - sh) / 2;
145      if (kluc == p_pole[stred])
146          return stred;
147      if (kluc > p_pole[stred])
148          return binarneVyhladanie(p_pole, (stred + 1), hh, kluc);
149      return binarneVyhladanie(p_pole, sh, (stred - 1), kluc);
150  }
```

Kontajner budujeme ako usporiadaný. Z tohto dôvodu došlo k zmene implementácie funkcie *vlozPrvokUsporiadane()*. Skôr ako prvok vložíme do kontajnera, musí dôjsť k nájdeniu správneho miesta, kde má byť tento prvok vložený. Ako jeden z parametrov je funkcii odovzdaná hodnota premennej *pozicia*, ktorá signalizuje prázdne miesto v poli. Pomocou cyklu *for* začíname prehľadávať priestor v poli od tejto pozície zníženej o 1 (posledná reálne obsadená hodnota v poli) a postupujeme kým sme neprešli všetky prvky v poli, alebo kým platí podmienka, že vkladaný prvok je menší ako prvok v poli. Kým táto podmienka platí, realizujeme posun prvkov doprava. Tým vytvoríme miesto pre vloženie nového prvku (riadok 105). Po vložení prvku inkrementujeme premennú *pozicia* (riadok 107).

```
91  void vlozPrvokUsporiadane(int prvok, int *pozicia, int *p_pole, int *velkost)
92  {
93      int i, j;
94      if(*pozicia >= *velkost)
95      {
96          if((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
```

```
97          {
98              printf("Nedostatok pamate. Program konci.\n");
99              return;
100         }
101         printf("Prvok bol vlozeny po zvacseny kontajnera.\n");
102     }
103
104     for(i = (*pozicia) - 1; (i >= 0 && p_pole[i] > prvok); i--)
105         p_pole[i + 1] = p_pole[i];
106     p_pole[i + 1] = prvok;
107     (*pozicia)++;
108 }
```

Implementácia zvyšných častí programu ostala rovnaká, ako v príklade v podkapitole 1.3.2.1.

Jedna z možných implementácií 1.3.2.2_Pr_1.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // deklaracie funkcii
5  void vlozPrvokUsporiadane(int prvok, int *pozicia, int *p_pole, int *velkost);
6  int odstranPrvok(int prvok, int *pozicia, int *p_pole);
7  void vypisPola(int *p_pole, int pozicia);
8  int binarneVyhladanie(int *p_pole, int sh, int hh, int kluc);
9
10 int main (void)
11 {
12     // deklaracia pointra reprezentujuceho pole a dalsich premennych
13     int pozicia = 0;
14     int * p_pole, velkost, volba, odpoved = 0, prvok, i, index;
15
16     // alokacia priestoru (kapacity pola) na zaklade volby pouzivatela
17     do
18     {
19         printf("Ake velke ulozisko si zelas vytvorit?: \n");
20         scanf("%d", &velkost);
21     }
22     while(velkost <= 0);
23
24     if((p_pole = (int *) malloc(sizeof(int) * velkost)) == NULL)
25     {
26         printf("Nedostatok pamate. Program konci.\n");
27         return 1;
28     }
29     // menu
30     do
31     {
32         printf("1 - vlozenie prvku do pola.\n");
33         printf("2 - odstranenie prvku z pola.\n");
34         printf("3 - vypisanie obsahu pola.\n");
35         printf("4 - hladanie v poli.\n");
36         printf("5 - koniec programu.\n");
37
```

```
38          scanf("%d", &volba);
39          switch(volba)
40          {
41          case 1:
42              printf("Aky prvok chces vlozit do kontajnera?:\n");
43              scanf("%d", &prvok);
44              vlozPrvokUsporiadane(prvok, &pozicia, p_pole, &velkost);
45              break;
46          case 2:
47              if(pozicia == 0)
48                  printf("Tvoj kontajner je prazdny, nemas co odstranovat.\n");
49              else
50              {
51                  printf("Aky prvok chces odstranit z kontajnera?:\n");
52                  scanf("%d", &prvok);
53                  if(odstranPrvok(prvok, &pozicia, p_pole) == 0)
54                      printf("Taky prvok sa v kontajnery nenachadza.\n");
55              }
56              break;
57          case 3:
58              if(pozicia == 0)
59                  printf("Tvoj kontajner je prazdny, nemam co vypisat.\n");
60              else
61                  vypisPola(p_pole, pozicia);
62              break;
63          case 4:
64              if(pozicia == 0)
65                  printf("Tvoj kontajner je prazdny, nemam kde vyhladavat.\n");
66              else
67              {
68                  printf("Aky prvok chces vyhladat v kontajnery?:\n");
69                  scanf("%d", &prvok);
70                  if((index = binarneVyhladanie(p_pole,0, velkost, prvok)) == -1)
71                      printf("Taky prvok sa v kontajnery nenachadza.\n");
72                  else
73                      printf("Index prveho vyskytu hladaneho prvku %d je: %d.\n",prvok, index);
74              }
75              break;
76          default:
77              odpoved = 1;
78          }
79      }
80      while(!odpoved);
81
82      // uvolnenie alokovanej oblasti
83      free(p_pole);
84      p_pole = NULL;
85
86      return 0;
87  }
88
89  // definicie funkcii
90   // vlozenie prvku na spravne miesto,ak sa presiahne velkost pola, tak sa tato automaticky
zdvojnasobi
```

```
91  void vlozPrvokUsporiadane(int prvok, int *pozicia, int *p_pole, int *velkost)
92  {
93      int i, j;
94      if(*pozicia >= *velkost)
95      {
96          if((p_pole = (int *) realloc(p_pole, (sizeof(int) * (*velkost) * 2))) == NULL)
97          {
98              printf("Nedostatok pamate. Program konci.\n");
99              return;
100         }
101         printf("Prvok bol vlozeny po zvacseny kontajnera.\n");
102     }
103
104     for(i = (*pozicia) - 1; (i >= 0 && p_pole[i] > prvok); i--)
105         p_pole[i + 1] = p_pole[i];
106     p_pole[i + 1] = prvok;
107     (*pozicia)++;
108 }
109
110 // funkcia odstranuje prvy vyskyt zadaneho prvku, ak nedojde k odstranovaniu vrati 0
111 int odstranPrvok(int prvok, int *pozicia, int *p_pole)
112 {
113     int i, j;
114     for(i = 0; i < *pozicia; i++)
115     {
116         if(prvok == p_pole[i])
117         {
118             for(j = i; j < *pozicia; j++)
119             {
120                 p_pole[j]=p_pole[j + 1];
121             }
122             (*pozicia)--;
123             printf("Prvy vyskyt prvku %d bol odstraneny.\n", prvok);
124             return 1;
125         }
126     }
127     return 0;
128 }
129
130 void vypisPola(int *p_pole, int pozicia)
131 {
132     int i;
133     printf("Prvky pola: ");
134     for(i = 0; i < pozicia; i++)
135         printf("%d ", *(p_pole + i));
136     printf("\n");
137 }
138
139 // vrati -1 ak sa prvok nenajde, inak vrati jeho index
140 int binarneVyhladanie(int *p_pole, int sh, int hh, int kluc)
141 {
142     if (hh < sh)
143         return -1;
144     int stred = (sh + hh) / 2;  // int stred = sh + (hh - sh) / 2;
145     if (kluc == p_pole[stred])
```

```
146             return stred;
147         if (kluc > p_pole[stred])
148             return binarneVyhladanie(p_pole, (stred + 1), hh, kluc);
149         return binarneVyhladanie(p_pole, sh, (stred - 1), kluc);
150     }
```

1.3.2.3 Príklad – dynamické pole, vkladanie a odoberanie prvkov z ľubovoľnej pozície

Zadanie problému:

Vytvorte program, ktorý bude poskytovať možnosť vkladania a odstraňovania celých čísiel na/z ľubovoľnej pozície dynamického poľa. V prípade potreby vloženia prvku pri nedostatočnej kapacite sa táto automaticky zväčší (zdvojnásobí).

Objasnenie autora:

V tomto príklade definujeme vlastný dátový typ, ktorý bude reprezentovať dynamické pole s jeho základnými vlastnosťami, ktorými sú veľkosť (*capacity*) a počet (*size*) – reálny počet prvkov v poli. K tomu využijeme štruktúru (riadky 5 – 10). Vďaka zavedeniu takéhoto dátového typu je možné v main-e deklarovať ľubovoľný počet dynamických polí a nezávisle s nimi pracovať.

```
4   // definicia typu pre dynamicke pole
5   typedef struct
6   {
7       int * pole;
8       int pocet;
9       int velkost;
10  } dynamickePole;
```

V implementácii sme sa obmedzili len na základné operácie: inicializácia (vytvorenie kontajnera a inicializácia potrebných premenných), vloženie prvku na konkrétnu pozíciu, odstránenie prvku z konkrétnej pozície a uvoľnenie pamäte pre kontajner.

Inicializáciu kontajnera realizujeme pomocou funkcie *init()*, ktorej formálnymi parametrami sú *array*, ktoré reprezentuje pole s ktorým pracujeme. Tento parameter je odovzdávaný odkazom z dôvodu, že chceme aby všetky zmeny, ktoré sa vo funkcii zrealizujú, boli trvalé, t. j. aby ovplyvnili skutočný parameter odovzdaný do funkcie. Parameter *velkost*, ktorý reprezentuje kapacitu poľa, je parameter odovzdaný hodnotou, keďže nie je našim úmyslom ho nijako modifikovať a je ho potrebné funkcii len odovzdať pre jej správnu funkčnosť. V tele funkcie sa postaráme o alokovanie priestoru na základe parametra *velkost* a inicializujeme premenné *pocet* a *velkost* poľa s ktorým pracujeme. *Pocet* reprezentuje reálny počet vložených prvkov, preto ho inicializujeme na hodnotu 0 (riadky 79 – 89).

```
79  void init(dynamickePole * array, int velkost)
80  {
81      array->pole = (int*)malloc(velkost * sizeof(int));
82      if (array->pole == NULL)
83      {
84          printf("Nedostatok pamate. Program konci.\n");
85          exit(1);
86      }
87      array->pocet = 0;
88      array->velkost = velkost;
89  }
```

Pre vloženie prvku na konkrétnu pozíciu využijeme funkciu *vlozenie()*, ktorá má tri formálne parametre. *Array* identifikuje pole s ktorým pracujeme, *prvok* reprezentuje hodnotu, ktorú do kontajnera vkladáme a *pozicia* predstavuje index v poli, na ktorý chceme daný prvok vložiť. Ako prvé ošetríme korektnosť pozície. Táto nesmie byť menšia ako 0. V opačnom prípade zareaguje chybovým hlásením. Veľkosť poľa neobmedzujeme, iba kontrolujeme a v prípade, ak už nie je dostatok miesta (čo identifikujeme jednoduchým overením obsahu premennej *pocet* a *velkost*), zväčšíme alokovanú pamäť dvojnásobne pomocou funkcie *realloc()* (riadky 101 – 110). Pri nedostatku pamäte sa funkcia ukončí bez vloženia prvku. V opačnom prípade sa postaráme o vloženie prvku na požadovanú pozíciu tak, že v prípade, ak vkladáme prvok na obsadené miesto, postaráme sa o posun potrebných prvkov v kontajneri doprava (riadky 113 – 116). Následne môžeme daný prvok vložiť na požadovanú pozíciu a nezabudneme inkrementovať premennú *pocet*, ktorá reprezentuje reálny počet v kontajneri.

```
92  void vlozenie(dynamickePole* array, int prvok, int pozicia)
93  {
94      if (pozicia < 0)
95      {
96          printf("Neplatna pozicia.\n");
97          return;
98      }
99
100     // ak je pole plne, zvysime jeho kapacitu
101     if (array->pocet == array->velkost)
102     {
103         array->velkost *= 2;
104         array->pole = (int*)realloc(array->pole, array->velkost * sizeof(int));
105         if (array->pole == NULL)
106         {
107             printf("Nedostatok pamate. Funkcia konci.\n");
108             return;
109         }
110     }
```

Pri funkcii na odstraňovanie prvku z konkrétnej pozície postupujeme analogicky. Overíme, či ide o korektnú pozíciu. Táto nemôže byť logicky menšia ako 0, ale ani väčšia alebo rovná obsahu

premennej *pocet*, ktorá hovorí o počte prvkov v kontajneri. V prípade ak sa za odstraňovaním prvkom nachádzajú ďalšie prvky postaráme sa o ich posuv vľavo (riadky 133 – 136), čím zabezpečíme aj prepísanie odstraňovaného prvku, a tým simulujeme odstránenie v poli. Nezabudneme dekrementovať premennú *pocet*. Pre hospodárne využívanie pamäte sme sa v tejto funkcii rozhodli implementovať skutočnosť, že v prípade ak je reálny počet prvkov v kontajneri menší, nanajvýš rovný polovičke veľkosti kontajnera, tak tento priestor zmenšíme na polovicu (riadky 141 – 149). Samozrejme, nezabudneme aktualizovať hodnotou premennej *velkost*.

```
124  void odstranenie(dynamickePole* array, int pozicia)
125  {
126      if (pozicia < 0 || pozicia >= array->pocet)
127      {
128          printf("Neplatna pozicia.\n");
129          return;
130      }
131
132      // posunutie prvkov vlavo od odstranovanej pozÄ-cie
133      for (int i = pozicia; i < array->pocet - 1; i++)
134      {
135          array->pole[i] = array->pole[i + 1];
136      }
137
138      array->pocet--;
139
140      // ak je pocet prvkov v poli mensi ako polovica jeho kapacity, znizime kapacitu
141      if (array->pocet <= array->velkost / 2)
142      {
143          array->pole = (int*)realloc(array->pole, array->velkost/2 * sizeof(int));
144          if (array->pole == NULL)
145          {
146              printf("Nepodarilo sa zmensit velkost pola. Tato ostala bezo zmeny.\n");
147              return;
148          }
149          array->velkost /= 2;
150      }
151  }
```

Pre uvoľnenie pamäte kontajnera sme implementovali funkciu *uvolnenie()*, ktorá má len jeden parameter a tým je pole, ktoré chceme uvoľniť. Pomocou funkcie *free()* sa postaráme o uvoľnenie pamäťového priestoru a nastavíme hodnoty premenných *pocet* a *velkost* na 0. Vzhľadom k tomu, že korektne inicializujeme tieto hodnoty pre správnu funkčnosť kontajnera vo funkcii *init()*, nebolo by vynechanie týchto nastavení kritické. Avšak z pohľadu konceptu a jednoduchej čitateľnosti a zrozumiteľnosti kódu je to viac ako vhodné (riadky 156 – 159).

```
154 void uvolnenie(dynamickePole* array)
155 {
156     free(array->pole);
157     array->pole = NULL;
158     array->pocet = 0;
159     array->velkost = 0;
160 }
```

Jedna z možných implementácií 1.3.2.3_Pr_1.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // definicia typu pre dynamicke pole
5  typedef struct
6  {
7      int * pole;
8      int pocet;
9      int velkost;
10 } dynamickePole;
11
12 //deklaracie funkcii
13 void init(dynamickePole * array, int velkost);
14 void vlozenie(dynamickePole* array, int prvok, int pozicia);
15 void odstranenie(dynamickePole* array, int pozicia);
16 void uvolnenie(dynamickePole* array);
17
18
19 int main(void)
20 {
21     dynamickePole pole1, pole2;
22     int velkost1 = 4;
23     int velkost2 = 3;
24     init(&pole1, velkost1);
25     init(&pole2, velkost2);
26
27     // testovacie operacii na poli1
28     vlozenie(&pole1, 1, 0);      // vlozenie prvku 1 na poziciu 0
29     vlozenie(&pole1, 2, 0);      // vlozenie prvku 2 na poziciu 0
30     vlozenie(&pole1, 3, 2);      // vlozenie prvku 3 na poziciu 2
31
32     printf("Obsah po vlozeni prvkov: ");
33     for (int i = 0; i < pole1.pocet; i++)
34     {
35         printf("%d ", pole1.pole[i]);
36     }
37     printf("\n");
38
39     odstranenie(&pole1, 1);      // odstranenie prvku na pozicii 1
40
41     printf("Obsah po odstraneni prvku: ");
42     for (int i = 0; i < pole1.pocet; i++)
43     {
44         printf("%d ", pole1.pole[i]);
```

```
45      }
46      printf("\n");
47
48      uvolnenie(&pole1);  // uvolnenie pamate
49
50      // testovacie operacii na poli2
51      vlozenie(&pole2, 1, 0);      // vlozenie prvku 1 na poziciu 0
52      vlozenie(&pole2, 2, 1);      // vlozenie prvku 2 na poziciu 1
53      vlozenie(&pole2, 3, 2);      // vlozenie prvku 3 na poziciu 2
54      vlozenie(&pole2, 4, 3);      // vlozenie prvku 4 na poziciu 3
55      vlozenie(&pole2, 5, -2);     // vlozenie prvku 5 na poziciu -2
56
57      printf("Obsah po vlozeni prvkov: ");
58      for (int i = 0; i < pole2.pocet; i++)
59      {
60          printf("%d ", pole2.pole[i]);
61      }
62      printf("\n");
63
64      odstranenie(&pole2, 0);      // odstranenie prvku na pozicii 0
65      odstranenie(&pole2, 5);      // odstranenie prvku na pozicii 5
66
67      printf("Obsah po odstraneni prvku: ");
68      for (int i = 0; i < pole2.pocet; i++)
69      {
70          printf("%d ", pole2.pole[i]);
71      }
72      printf("\n");
73
74      uvolnenie(&pole2);  // uvolnenie pamate
75      return 0;
76  }
77
78  // inicializacia dynamickeho pola
79  void init(dynamickePole * array, int velkost)
80  {
81      array->pole = (int*)malloc(velkost * sizeof(int));
82      if (array->pole == NULL)
83      {
84          printf("Nedostatok pamate. Program konci.\n");
85          exit(1);
86      }
87      array->pocet = 0;
88      array->velkost = velkost;
89  }
90
91  // vlozenie prvku na urcenu poziciu
92  void vlozenie(dynamickePole* array, int prvok, int pozicia)
93  {
94      if (pozicia < 0)
95      {
96          printf("Neplatna pozicia.\n");
97          return;
98      }
99
```

```
100     // ak je pole plne, zvysime jeho kapacitu
101     if (array->pocet == array->velkost)
102     {
103         array->velkost *= 2;
104         array->pole = (int*)realloc(array->pole, array->velkost * sizeof(int));
105         if (array->pole == NULL)
106         {
107             printf("Nedostatok pamate. Funkcia konci.\n");
108             return;
109         }
110     }
111 
112     // posunutie prvkov napravo od vlozenej pozicie
113     for (int i = array->pocet; i > pozicia; i--)
114     {
115         array->pole[i] = array->pole[i - 1];
116     }
117 
118     // vlozenie prvku na urcenu poziciu
119     array->pole[pozicia] = prvok;
120     array->pocet++;
121 }
122 
123 // odstranenie prvku na urcenej pozicii
124 void odstranenie(dynamickePole* array, int pozicia)
125 {
126     if (pozicia < 0 || pozicia >= array->pocet)
127     {
128         printf("Neplatna pozicia.\n");
129         return;
130     }
131 
132     // posunutie prvkov vlavo od odstranovanej pozÄ-cie
133     for (int i = pozicia; i < array->pocet - 1; i++)
134     {
135         array->pole[i] = array->pole[i + 1];
136     }
137 
138     array->pocet--;
139 
140     // ak je pocet prvkov v poli mensi ako polovica jeho kapacity, znizime kapacitu
141     if (array->pocet <= array->velkost / 2)
142     {
143         array->pole = (int*)realloc(array->pole, array->velkost/2 * sizeof(int));
144         if (array->pole == NULL)
145         {
146             printf("Nepodarilo sa zmensit velkost pola. Tato ostala bezo zmeny.\n");
147             return;
148         }
149         array->velkost /= 2;
150     }
151 }
152 
153 // uvolnenie pamate pouzivanej dynamickym polom
154 void uvolnenie(dynamickePole* array)
```

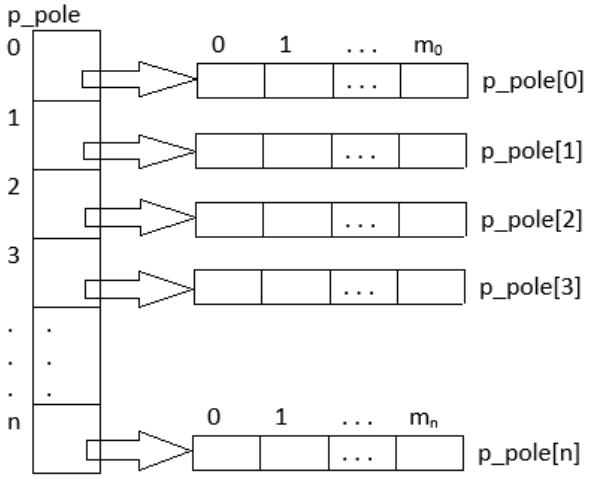
```
155  {
156      free(array->pole);
157      array->pole = NULL;
158      array->pocet = 0;
159      array->velkost = 0;
160  }
```

1.3.3 Dvojrozmerné dynamické pole

Uvažujme dynamické prideľovanie pamäte pre dvojrozmerné pole deklarované nasledovne:

```
int **p_pole;
```

Grafická reprezentácia takéhoto poľa môže byť reprezentovaná nasledovne (veľkosť vertikálneho poľa odpovedá počtu riadku, veľkosť horizontálnych polí počtu stĺpcov):



Obr. 7 Dvojrozmerné rovnomerné dynamické pole.

Algoritmus vytvorenia dvojrozmerného dynamického poľa je nasledujúci:

1. Deklarácia konštánt určujúcich počet riadkov a stĺpcov. Samozrejme, je možné rozmery načítať priamo od používateľa, resp. zo súboru:

   ```
   #define SIZE_X 4
   #define SIZE_Y 5
   ```

2. Deklarácia dvojitého pointera predstavujúceho dvojrozmerné pole a deklarácia pomocnej premennej:

   ```
   int **p_pole, i;
   ```

3. Alokovanie (vyhradenie) pamäte pre pole smerníkov. Toto zodpovedá počtu riadkov. Nezabúdajme na testovanie, či sa podarilo pamäť alokovať:

   ```
   p_pole = (int **) malloc(sizeof(int *) * SIZE_X);
   if(p_pole == NULL) exit(0);      // chyba
   ```

4. Alokovanie (vyhradenie) pamäte pre jednotlivé vektory poľa smerníkov. Toto zodpovedá počtu stĺpcov:

```
for(i = 0; i < SIZE_X; i++)
{
    p_pole[i] = (int *) malloc(sizeof(int) * SIZE_Y);
    if(p_pole[i] == NULL) exit(0);
}
```

Algoritmus uvoľnenia dvojrozmerného dynamického poľa je nasledujúci:

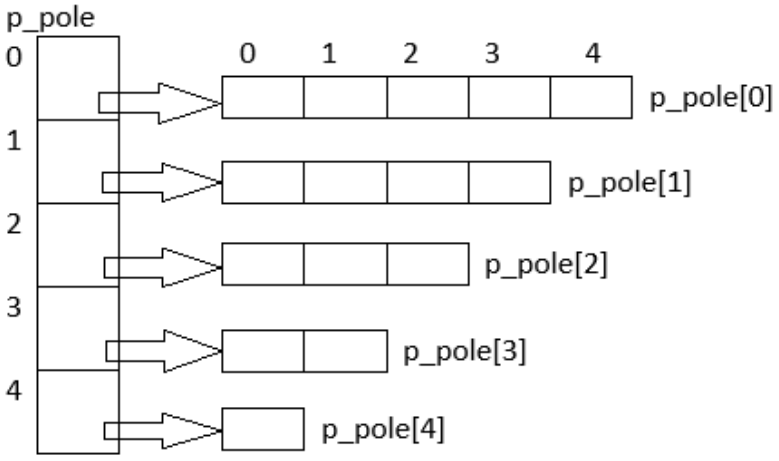
1. Uvoľnenie pamäte pre jednotlivé vektory poľa

```
for (i = 0; i < SIZE_X; i++)
{
    free(p_pole[i]);
}
```

2. Uvoľnenie pamäte pre pole smerníkov na vektory

```
free(p_pole);
p_pole = NULL;
```

Obdobným spôsobom je možné definovať aj dvojrozmerné pole s nerovnomernou štruktúrou, napríklad v tvare:



Obr. 8 Dvojrozmerné nerovnomerné dynamické pole.

1.3.3.1 Príklad – dynamické a statické dvojrozmerné pole

Zadanie problému: Vytvorte program, ktorý bude ilustrovať prácu s dynamickým aj so statickým dvojrozmerným poľom.

Jedna z možných implementácií 1.3.3.1_Pr_1.c:

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE_X 3
#define SIZE_Y 4

// deklaracie funkcii
int** alokuj_pole(int m, int n);
void uvolni_pole(int** a, int m);
void vypis_pole(int** a, int m, int n);

int main(void)
{
    // staticky alokovane 2D pole
    printf("Staticky alokovane pole:\n");

    int a[SIZE_X][SIZE_Y];

    // naplnenie pola
    for (int i = 0; i < SIZE_X; i++)
        for (int j = 0; j < SIZE_Y; j++)
            a[i][j] = i + 1;

    // vypis pola
    for (int i = 0; i < SIZE_X; i++)
    {
        for (int j = 0; j < SIZE_Y; j++)
            printf("a[%d][%d]=%d\t", i, j, a[i][j]);
        printf("\n");
    }

    // dynamicky alokovane 2D pole cez funkcie
    printf("\nDynamicky alokovane pole:\n");

    int** b = alokuj_pole(SIZE_X, SIZE_Y);

    // naplnenie pola
    for (int i = 0; i < SIZE_X; i++)
        for (int j = 0; j < SIZE_Y; j++)
            b[i][j] = i + 1;

    vypis_pole(b, SIZE_X, SIZE_Y);

    uvolni_pole(b, SIZE_X);

    return 0;
}
```

```
48
49  // definicie funkcii
50  int** alokuj_pole(int m, int n)
51  {
52      int** b;
53
54      if ((b = (int**) malloc(m * sizeof(int*))) == NULL)
55      {
56          printf("Nedostatok pamate. Program konci.\n");
57          exit(1);
58      }
59
60      for (int i = 0; i < m; i++)
61          if((b[i] = (int*) malloc(n * sizeof(int))) == NULL)
62          {
63              printf("Nedostatok pamate. Program konci.\n");
64              uvolni_pole(b, i);
65              exit(1);
66          }
67      return b;
68
69  }
70
71  void vypis_pole(int** a, int m, int n)
72  {
73      for (int i = 0; i < m; i++)
74      {
75          for (int j = 0; j < n; j++)
76              printf("a[%d][%d]=%d\t", i, j, a[i][j]);
77          printf("\n");
78      }
79  }
80
81  void uvolni_pole(int** a, int m)
82  {
83      for (int i = 0; i < m; i++)
84          free(a[i]);
85      free(a);
86      a = NULL;
87  }
```

```
C:\Programy\4.7.1_Pr_1.exe
Staticky alokovane pole:
a[0][0]=1       a[0][1]=1       a[0][2]=1       a[0][3]=1
a[1][0]=2       a[1][1]=2       a[1][2]=2       a[1][3]=2
a[2][0]=3       a[2][1]=3       a[2][2]=3       a[2][3]=3

Dynamicky alokovane pole:
a[0][0]=1       a[0][1]=1       a[0][2]=1       a[0][3]=1
a[1][0]=2       a[1][1]=2       a[1][2]=2       a[1][3]=2
a[2][0]=3       a[2][1]=3       a[2][2]=3       a[2][3]=3

Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

Obr. 9 Konzolový výstup programu 1.3.3.1_Pr_1.c.

Pozn. autora: Funkcia *alokuj_pole()* zohľadňuje aj takú skutočnosť, že ak by nebol dostatok priestorových nárokov počas priebežného alokovania pamäte pre jednotlivé riadky dvojrozmerného poľa, príde k uvoľneniu aj tohto čiastočne alokovaného priestoru – zabezpečené volaním funkcie na riadku č. 64.

1.3.4 Zreťazený zoznam = *linked list*

Zreťazený zoznam (lineárny zreťazený zoznam = *linked list*) je kontajner určený k ukladaniu dát v prípade, ak vopred nevieme určiť, aká bude jeho končená veľkosť. Spôsob vkladania a odoberania prvkov je na vývojárovi. Tým sa líši od zásobníka a fronty, kde je spôsob manipulácie s prvkami pevne určený. V podstate ide o nekonečne dlhú dátovú štruktúru, t. j. počet prvkov ktorý do neho vkladáme je obmedzený len veľkosťou dostupnej pamäte. Často sú zreťazené zoznamy preferované pred použitím poľa, a to z dôvodu, že operácie vkladania a vyberania prvkov zoznamu sú relatívne efektívne a jednoduché na implementáciu.

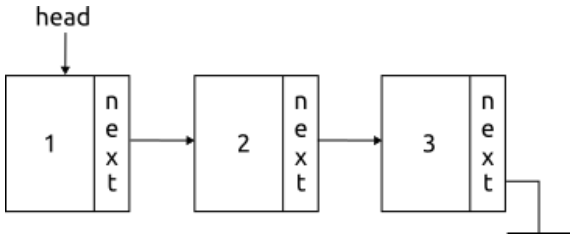
Základnou stavebnou jednotkou zreťazeného zoznamu je *uzol* (*node*), ktorého minimálna reprezentácia (kostra uzla) obsahuje dátovú časť (reprezentuje obsah uzla) a ukazovateľ na nasledujúci uzol (*nasledovník* = *next*). Je zrejmé, že jednotlivé časti uzla sú rôzneho dátového typu. Preto je vhodným dátovým typom používaným pre definíciu uzla heterogénny dátový typ štruktúra. Kostra (štruktúra) uzla sa môže meniť v závislosti od typu zoznamu.

Z dôvodu, že sa pamäť pre uzly alokuje dynamicky, nemusia byť jednotlivé uzly v pamäti počítača bezprostredne vedľa seba, ale môžu sa v pamäti nachádzať na rôznych miestach. Tieto sú však navzájom prepojené a tvoria lineárny zreťazený zoznam. To je principiálny rozdiel od poľa, kde sa alokuje jeden súvislý blok pamäte pre celé pole.

Rozlišujeme:

- **Jednosmerne zreťazený zoznam (*singly linked list*)**
  Je základným variantom tejto dátovej štruktúry. Jednotlivé uzly obsahujú okrem dát iba ukazovateľ na ďalší uzol (prvok), tzv. *nasledovník* (*next*). Posledný prvok ukazuje na *NULL*, keďže nemá nasledovníka a tým reprezentuje, kde dátová štruktúra končí. V takomto zozname je umožnený prechod len jedným smerom. Samotná štruktúra je reprezentovaná ukazovateľom na prvý uzol, ktorý sa najčastejšie označuje *hlava* (*head*) a reprezentuje začiatok dátovej štruktúry. V tomto variante sa nové uzly pridávajú na začiatok zoznamu z dôvodu dosiahnutia konštantnej časovej zložitosti tejto operácie. V takomto prípade stačí

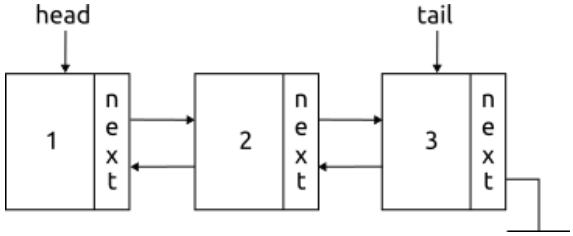
len vytvoriť nový uzol a vložiť ho na začiatok zoznamu. Ak by sme chceli uzol pridať na koniec, vyžadovalo by si to prechod cez všetky existujúce uzly, pretože zreťazený zoznam neumožňuje náhodný prístup k prvkom cez indexy ako pole, čo by spôsobilo lineárnu časovú zložitosť. Z tohto dôvodu sa často pri implementáciách udržiava aj ukazovateľ na posledný uzol, tzv. *chvost* (*tail*). Vďaka tomu je potom možné pridávať prvky aj na koniec s konštantnou časovou zložitosťou.



Obr. 10 Vizuálna reprezentácia jednosmerne zreťazeného zoznamu.

- **Obojsmerne zreťazený zoznam (*doubly linked list*)**

  V takomto type zreťazeného zoznamu obsahujú uzly okrem ukazovateľa na ďalší prvok (*nasledovník* = *next*) aj ukazovateľ na predchádzajúci prvok, tzv. *predchodcu* (*previous* = *prev*). Takáto skutočnosť mierne skomplikuje samotnú implementáciu, avšak na úkor vyššej flexibility, pretože je možné prechádzať zreťazeným zoznamom v oboch smeroch a zjednodušia sa aj viaceré implementácie operácií nad zreťazeným zoznamom.

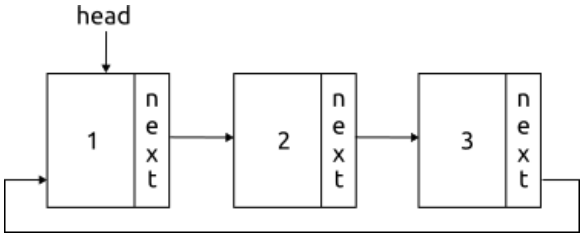


Obr. 11 Vizuálna reprezentácia obojsmerne zreťazeného zoznamu.

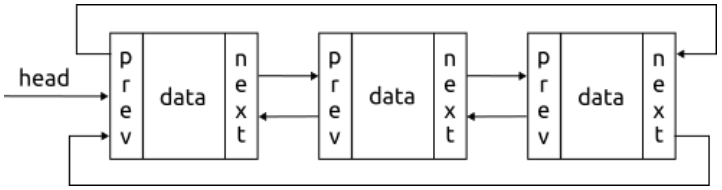
- **Kruhový zreťazený zoznam (*circular linked list*)**

  Je zreťazený zoznam, ktorý rozširuje klasický zreťazený zoznam (jednosmerný aj obojsmerný) tak, že posledný prvok v zreťazenom zozname odkazuje na prvý prvok, vytvárajúc tak kruhový cyklus. V podstate ide o zreťazený zoznam, ktorého nasledovník posledného uzla nemá hodnotu *NULL*, ale odkazuje späť na svoj počiatočný prvok. Táto dátová štruktúra umožňuje jednoduchý prístup k prvkom na začiatku aj na konci zoznamu v prípade obojsmerne zreťazeného zoznamu, pretože prvý prvok je ľahko dostupný z nasledovníka posledného prvku a posledný prvok sa nachádza priamo pred prvým prvkom.

Okrem toho prvý prvok je vždy prístupný cez pointer ukazujúci na začiatok, tzv. *head*. Kruhový zreťazený zoznam je užitočný v prípadoch, keď je potrebné prechádzať zoznam v cykle, napríklad pri implementácii kruhových front (*circular queues*) alebo cyklických bufferov. Taktiež je užitočný pre efektívne riešenie problémov, ktoré vyžadujú cyklické správanie, ako napríklad cyklický algoritmus na spracovanie dát. Pri práci s kruhovým zreťazeným zoznamom je potrebné dávať pozor na správne nastavenie ukazovateľov a riadenie operácií, aby sa zabránilo nekonečnému cyklu.



Obr. 12 Vizuálna reprezentácia jednosmerne zreťazeného kruhového zoznamu.



Obr. 13 Vizuálna reprezentácia obojsmerne zreťazeného kruhového zoznamu.

Problematická operácia mazania, resp. vloženia prvku na ľubovoľné miesto v dátovej štruktúre zreťazený zoznam je riešená veľmi jednoducho, na rozdiel od riešenia, ktoré sme používali pri poli. Stačí sa postarať o správne nastavenie hodnôt ukazovateľov jednotlivých uzlov, tzv. prepojenie, väzba medzi uzlami.

Typické použitie:

- Lineárne zreťazené zoznamy sú vhodné pre dynamické zoznamy, kde sa prvky môžu ľahko pridávať alebo odstraňovať, čím vieme jednoducho dynamicky meniť veľkosť dátovej štruktúry, na rozdiel od poľa. Táto flexibilita je užitočná pri implementácii mnohých algoritmov a štruktúr dát, ako napríklad zásobníky, fronty, ale aj zložitejších dátové štruktúry, ako stromy, grafy a mnoho ďalších.
- Lineárne zreťazené zoznamy sa používajú na spracovanie a manipuláciu s reťazcami. Každý znak reťazca môže byť reprezentovaný ako prvok v zreťazenom zozname. To umožňuje vkladanie, odstraňovanie a vyhľadávanie znakov v reťazci.

- *Undo/Redo* operácie: Lineárne zreťazené zoznamy sa často využívajú na implementáciu funkcií *Undo* (vrátenie) a *Redo* (opakovanie). Každá vykonaná operácia sa pridáva do zoznamu *Undo*, a ak sa vyžaduje vrátenie k predchádzajúcemu stavu, operácie sa vykonávajú v opačnom smere.
- Niektoré *garbage collection* algoritmy, ktoré sa používajú pre automatickú správu pamäte používajú zreťazené zoznamy na sledovanie alokovaných pamäťových blokov, ktoré je potrebné uvoľniť, keď sa už nepoužívajú.
- V systémoch plánovania úloh možno zreťazené zoznamy použiť na riadenie fronty úloh alebo úloh, ktoré je potrebné vykonať v špecifickom poradí.
- V kompilátoroch a interpretoch sa zreťazené zoznamy používajú na implementáciu tabuliek symbolov pre premenné, funkcie a identifikátory v zdrojovom kóde.
- Zreťazené zoznamy možno použiť na implementáciu zoznamov skladieb v prehrávačoch médií, kde je každá skladba alebo mediálny súbor reprezentovaný ako uzol a používatelia môžu skladby jednoducho pridávať, odstraňovať alebo meniť ich usporiadanie.
- Textové editory často používajú zreťazené zoznamy na reprezentáciu a manipuláciu s textom, pretože umožňujú efektívne vkladanie a odstraňovanie znakov alebo riadkov.

1.3.4.1 Príklad – obojsmerne zreťazený zoznam

Zadanie problému:

Vytvorte program, ktorý bude implementovať kontajner pomocou obojsmerne zreťazeného zoznamu. Kontajner bude slúžiť na uchovávanie informácii o študentoch. Tento bude umožňovať pridanie prvku, pridanie prvku pred a za konkrétny prvok na základe jeho obsahu, nájdenie konkrétneho prvku, zmenu údajov konkrétneho prvku, výpis obsahu, odstránenie konkrétneho prvku a odstránenie/uvoľnenie celého kontajnera.

Objasnenie príkladu autorom:

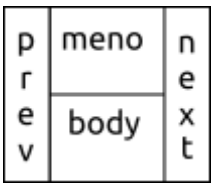
Ide o ilustráciu zoznamu študentov a ich bodového hodnotenia. Každý uzol predstavuje jeden záznam, ktorý má štyri položky: meno, body, ukazovateľ na nasledujúci uzol a na predchádzajúci uzol v zreťazenom zozname. Implementácia pomocou obojstranne zreťazeného zoznamu voči jednosmerne zreťazenému zoznamu zjednodušuje princíp napríklad pri vkladaní prvkov pred konkrétny prvok. Ak by sme využili jednosmerný zreťazený zoznam, potrebovali by sme

pri pridávaní prvku pred konkrétny prvok ostať pred týmto prvkom, aby sme vedeli vytvoriť potrebné prepojenia. V prípade obojstranne zreťazeného zoznamu, kde uzol uchováva odkaz na predchodcu, ako aj nasledovníka, stačí vyhľadať konkrétny uzol, pred ktorý chceme daný uzol pridať, pretože od neho sa vieme dostať aj na predchodcu aj nasledovníka.

Definujme nový dátový typ, ktorý sme nazvali *LISTnode*:

```
 7  typedef struct node
 8  {
 9      char meno[30];
10      int body;
11      struct node *next;
12      struct node *prev;
13  } LISTnode;
```

Prvá položka, ktorá je deklarovaná ako reťazec znakov slúži na uchovanie mena a priezviska študenta. Samozrejme, je vhodnejšie deklarovať dve samostatné položky, kde jedna by uchovávala meno a druhá priezvisko, aby sme vedeli efektívne narábať s týmito údajmi. Túto skutočnosť zanedbávame z dôvodu, že nám nejde o ilustráciu efektívnej práce s obsahom zreťazeného zoznamu, ale o ilustráciu základných operácií nad obojsmerne zreťazeným zoznamom, ako dátovej štruktúry. Druhá položka *body* predstavuje bodové hodnotenie študenta. Tretia a štvrtá položka predstavuje odkaz na nasledovníka a predchodcu. Takáto definícia, ktorá odkazuje sama na seba (*self-referential structure*) je možná len v prípade, ak využijeme spôsob definovania pomenovanej štruktúry. Pre pomenovanie štruktúry sa zvykne voliť meno, ktoré jednoznačne reprezentuje daný objekt, pričom identifikátor pre nový dátový typ sa zvyčajne volí podobný tomuto pomenovaniu, ale môže byť aj rovnaký. Štruktúru sme preto pomenovali *node* (uzol) a dátový typ *LISTnode*, keďže predstavuje uzol údajovej štruktúry zreťazený zoznam (*linked list*). Každý prvok v zreťazenom zozname bude tohto typu.



Obr. 14 Vizuálna reprezentácia kostry uzla v zreťazenom zozname.

Pozn. autora: Prepojenie jednotlivých uzlov v zreťazenom zozname vizuálne reprezentujeme pomocou šípky ⟶ a to, že uzol už neukazuje na ďalší uzol, t. j. ukazovateľ nadobúda hodnotu *NULL* symbolom ⏊.

Pre prácu s dynamickou štruktúrou by postačovalo deklarovať pointer, ktorý by ukazoval na začiatok zreťazeného zoznamu. My však definujeme dva ukazovatele, *head*, ktorý ukazuje na prvý prvok zreťazeného zoznamu a *tail*, ktorý ukazuje na posledný prvok zreťazeného zoznamu. Je to z dôvodu, že najprirodzenejším spôsobom pridávania nových prvkov do zreťazeného zoznamu je ich pridávať na koniec. V takomto prípade bude položka nasledovníka vždy nastavená na hodnotu *NULL* a je potrebné sa postarať už len o prepojenia medzi posledným a novým prvkom. Ak by sme neudržiavali *tail*, vyžadovalo by si to prechod všetkým uzlami zreťazeného zoznamu, až kým by ukazovateľ na nasledovníka nemal hodnotu *NULL*. To by spôsobilo lineárnu asymptotickú časovú zložitosť O(n). V tom prípade je vhodnejšie zvoliť pridávanie prvkov na začiatok zreťazeného zoznamu. V prípade udržiavania *tail*, vieme nový prvok na koniec pridať s konštantnou časovou zložitosťou O(1).

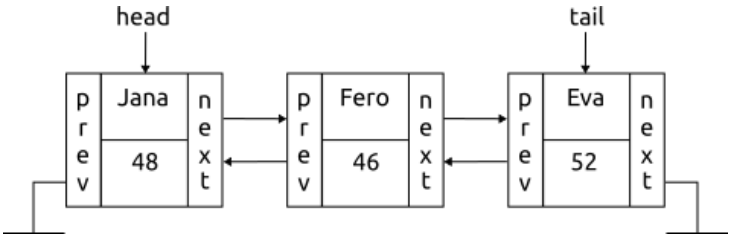
Deklaráciu týchto pointerov môžeme nájsť v main-e na riadku 29:

```
29      LISTnode *head, *tail;
```

Vytvorenie nového zreťazeného zoznamu, ktorý je na začiatku prázdny, reprezentujeme priradením hodnoty *NULL* do týchto pointerov:

```
32      head = tail = NULL;
```

1.3.4.2 Práca s uzlami v zreťazenom zozname



Obr. 15 Vizuálna reprezentácia obojstranne zreťazeného zoznamu.

Ak by sme chceli upraviť body pre študenta Fero, mohli by sme to zrealizovať príkazom:

```
head->next->body = 56;
```

head ukazuje na prvý prvok zoznamu, avšak v jeho časti *next* je uložený odkaz na druhý uzol, ktorý predstavuje záznam pre študenta Fera. Jednoduchým zreťazeným uzlov sa vieme k tomuto uzlu dostať. Našim cieľom je zmodifikovať hodnotu bodov a preto pristupujeme k položke *body*. Jednoduchým príkazom priradenia aktualizujeme túto hodnotu. Takýmto spôsobom je možné sa pohybovať po celej dátovej štruktúre. Samozrejme, pri existencii väčšieho množstva prvkov v zozname nebude takýto spôsob efektívny (zhoršuje sa čitateľnosť kódu) a budeme na prechod po uzloch používať cyklus. T. j.:

`head`	je ukazovateľ na prvý uzol zoznamu
`head->meno, head->body`	predstavujú jednotlivé položky prvého uzla zoznamu
`head->next`	predstavuje adresu ďalšieho uzla zoznamu
`head->prev`	predstavuje adresu predchádzajúceho uzla, ktorý je v prípade prvého prvku zoznamu nastavený na hodnotu *NULL*

1.3.4.3 Funkcia pre testovanie či je zreťazený zoznam prázdny

Jedna zo základných operácií nad dátovou štruktúrou je zistenie či je kontajner prázdny. Takejto funkcii postačí poznať začiatok zreťazeného zoznamu a keďže neočakávame žiadne zmeny na skutočnom zreťazenom zozname pod vplyvom tejto funkcie, stačí, ak je tento parameter odovzdaný hodnotou. V prípade práce s lineárnym zreťazeným zoznamom nám stačí overiť či hodnota pointera, ktorý predstavuje ukazovateľ na prvý prvok je rovná *NULL*. Na základe toho signalizujeme, či je zreťazený zoznam prázdny – funkcia vráti hodnotu 1.

Jedna z možných implementácií:

```
89  int prazdny(LISTnode *h)
90  {
91      /* vrati "true=1" ak je zoznam prazdny */
92      return(h == NULL);
93  }
```

1.3.4.4 Funkcia pre zobrazenie obsahu zreťazeného zoznamu

Funkciu pre zobrazenie obsahu navrhneme ako procedúru, ktorá v prípade ak zreťazený zoznam nie je prázdny, zobrazí jeho obsah. Takejto procedúre postačí poznať začiatok zreťazeného zoznamu, a keďže neočakávame žiadne zmeny na skutočnom zreťazenom zozname pod vplyvom tejto funkcie, stačí, ak je tento parameter odovzdaný hodnotou. Na testovanie, či nie je zreťazený zoznam prázdny, využijeme definovanú funkciu *prazdny()* (riadok 152). V opačnom prípade informujeme používateľa, že zreťazený zoznam je prázdny (riadok 166). Pri prehľadávaní zreťazeného zoznamu postupujeme tak, že prechádzame uzol po uzle pomocou ukazovateľa dovtedy, kým tento nemá hodnotu *NULL*, t. j. kým sme nenarazili na posledný uzol. V tele cyklu sa postaráme o výpis obsahu (riadky 158 a 159) a posun na ďalší uzol (riadok 160). Vhodnou modifikáciou konceptu procedúry vieme v prípade obojsmerne zreťazeného zoznamu prechádzať uzly aj v opačnom poradí, t. j. postarať sa o výpis obsahu od posledného prvku.

Pozn. autora: Volanie funkcie *getchar()* na riadkoch 161 a 168 je uvedené z dôvodu, aby sme stihli zaregistrovať zobrazený obsah zreťazeného zoznamu. Funkcia čaká na stlačenie ľubovoľného znaku, v opačnom prípade môže byť zobrazenie obsahu také rýchle, že ho nestihneme zaregistrovať.

Volanie funkcie v tele programu má podobu:

```
vypis(head);
```

Ako skutočný parameter odovzdávame začiatok zreťazeného zoznamu, t. j. pointer *head*. Ide o volanie hodnotou, a preto formálny parameter funkcie je deklarovaný ako jednoduchý pointer.

Jedna z možných implementácií:

```
150  void vypis(LISTnode *p)
151  {
152      if (!(prazdny(p)))
153      {
154          printf("VYPIS\n");
155
156          while (p != NULL)
157          {
158              printf("\nMeno: %s", p->meno);
159              printf("\nBody: %d \n",p->body);
160              p = p->next;     //posun na dalsi prvok
161              getchar();
162          }
163      }
164      else
165      {
166          printf("Zoznam neobsahuje ziadny prvok.\n");
167      }
168      getchar();
169  }
```

1.3.4.5 Funkcia pre vyhľadanie konkrétneho prvku

Funkcia pre vyhľadanie konkrétneho prvku v zreťazenom zozname preberá práve jeden parameter, a tým je začiatok zreťazeného zoznamu, odovzdaný hodnotou. Návratovou hodnotou funkcie bude v prípade úspechu priamo adresa daného uzla, v prípade neúspechu *NULL*. Z tohto dôvodu je návratová hodnota definovaná ako pointer na *LISTnode*. Na testovanie, či nie je zreťazený zoznam prázdny, využijeme definovanú funkciu *prazdny()* (riadok 201). V opačnom prípade informujeme používateľa, že zreťazený zoznam je prázdny (riadok 220 a 221). Skôr ako začneme zreťazený zoznam prehľadávať, vypýtame si od používateľa meno študenta, ktorého chce vyhľadať. Vzhľadom na jednoduchosť nášho zreťazeného zoznamu je to jediná logická hodnota na základe ktorej môžeme vyhľadávanie simulovať. Pri prehľadávaní zreťazeného zoznamu postupujeme tak, že prechádzame

uzol po uzle pomocou ukazovateľa dovtedy, kým tento nemá hodnotu *NULL*, t. j. kým sme nenarazili na posledný uzol alebo kým nenastane zhoda s menom hľadaného študenta. Na to sme využili vstavanú funkciu *strcmp()*, ktorá lexikograficky porovnáva dva reťazce a v prípade úspechu vráti hodnotu 0. V opačnom prípade vracia hodnotu rôznu od 0. V tele cyklu sa staráme len o posun na ďalší uzol (riadok 210). Vrátenie adresy nájdeného uzla, resp. hodnoty *NULL* realizujeme pomocou príkazu *return* na riadku 225. Obdobne platí, že vhodnou modifikáciou konceptu funkcie by sme mohli realizovať vyhľadávanie odzadu zreťazeného zoznamu.

Pozn. autora: Pre jednoduchosť implementácie sme položku uzla pre meno študenta deklarovali ako statické pole o veľkosti 30. V prípade použitia funkcie *gets()* pri zadaní reťazca dlhšieho ako 30 znakov hrozí zápis mimo alokovanej pamäte (napr. riadok 206). Tento problém je možné riešiť buď, zabezpečením načítania reťazca ľubovoľnej dĺžky, alebo zabrániť zápisu mimo pamäte, použitím funkcie *fgets()*. Funkcia *fgets()* pomocou druhého parametra dokáže obmedziť načítanie reťazca na požadovaný rozsah. V našom prípade je táto hodnota 30 znakov. To samozrejme nerieši problém úplne. V takomto prípade nemusí nastať načítanie celého reťazca, ktorý používateľ zadal. Načítanie pomocou funkcie *fgets()* by vyzeralo:

```
fgets(m, 30, stdin);
```

Na riadku 205 môžeme pozorovať volanie funkcie *fflush(stdin)*, ktorá sa má postarať o vyčistenie vstupného buffera. Čistenie buffera je vhodné realizovať pred každým formátovaným načítaním, ale hlavne pred načítavaním znaku/znakov. Z dôvodu, že každá hodnota zadaná používateľom do konzoly ako vstup je potvrdená znakom entera, ostáva tento znak po načítaní obsahu napr. celočíselnej premennej vo vstupnom bufferi. V prípade následného čítania znaku, sa načíta tento znak z buffera, čoho reálnym dôsledkom prejavu môže byť skutočnosť, že používateľovi nebude umožnené zadanie požadovaného vstupu programom. Funkcia *fflush()* nie je štandardnou funkciou a nemusí byť funkčná na každom operačnom systéme. Riešenie tohto problému, ktoré je závislé od operačného systému ponechávame na čitateľovi.

Volanie funkcie *najdi()* v tele programu má podobu:

```
najdi(head);
```

Ako skutočný parameter odovzdávame začiatok zreťazeného zoznamu, t. j. pointer *head*. Ide o volanie hodnotou a preto formálny parameter funkcie je deklarovaný, ako jednoduchý pointer.

Jedna z možných implementácií:

```
195  LISTnode *najdi(LISTnode *p)
196  /*
197     vrati pointer na prvok, ktory vyhovuje poziadavke
198     resp. NULL ak je zoznam prazdny alebo sa taky prvok v zozname nenachadza
199  */
200  {
201      if (!(prazdny(p)))
202      {
203          char m[30];
204          printf("Hladaj studenta: ");
205          fflush(stdin);
206          gets(m);
207
208          while ((p !=  NULL) && (strcmp(p->meno, m)))
209          {
210              p = p->next;
211          }
212
213          if (p ==  NULL)
214              printf("Nenasiel sa takyto zaznam!");
215          else
216              printf("Hladany student sa nasiel! Ma %d bodov\n",p->body);
217      } //if prazdny
218      else
219      {
220          printf("Zoznam je prazdny.\n");
221          printf("Najprv musis pridat nejaky prvok, aby si ho mohol vyhladat.\n");
222      }
223
224      getchar();
225      return(p);
226  }
```

1.3.4.6 Funkcia pre zmenu údajov konkrétneho prvku

Častou operáciou pri práci s kontajnermi môže byť aj nutnosť editácie údajov v dátovej časti uzla. Pre tento účel sme navrhli procedúru *zmena()*, ktorá v prípade ak zreťazený zoznam nie je prázdny, sa postará o vyhľadanie uzla, ktorého dáta chceme zmeniť, a na základe rozhodnutia používateľa umožní editovať obsah oboch položiek dátovej časti. Takejto procedúre postačí poznať začiatok zreťazeného zoznamu, ktorý je odovzdaný hodnotou. Na testovanie či nie je zoznam prázdny využijeme definovanú funkciu *prazdny()* (riadok 232). V opačnom prípade informujeme používateľa, že zreťazený zoznam je prázdny (riadky 263 a 264). Pre vyhľadanie uzla využijeme funkciu *najdi()*. Ako vieme, v prípade neúspechu vracia táto funkcia hodnotu *NULL*, preto na riadku 236 testujeme, či sme vôbec uzol, ktorého dáta chceme editovať, našli. Ak áno, postupne dávame používateľovi

na základe jeho odpovede možnosť výberu, ktorú položku chce editovať. Vo funkcii sa nesústreďujeme na ošetrenie korektnosti zadávaných údajov. Toto doplnenie nechávame čitateľovi.

Volanie funkcie v tele programu má podobu:

```
zmena(head);
```

Ako skutočný parameter odovzdávame začiatok zreťazeného zoznamu, t. j. pointer *head*. Ide o volanie hodnotou, a preto formálny parameter funkcie je deklarovaný ako jednoduchý pointer. Aj v tomto prípade postačuje odovzdanie hodnotou, pretože nemodifikujeme zreťazený zoznam, ale editujeme obsah v uzli, ktorý bol už vytvorený.

Pozn. autora: V tejto funkcii, sme ani v jednom prípade nevyužili na čistenie vstupného bufferu funkciu *fflush(stdin)*. Je to z dôvodu, ako sme uviedli vyššie, že nejde o štandardnú funkciu. Namiesto toho sme si jednoduchým cyklom bez tela, ktorý prechádza vstupný buffer znak po znaku, tento vyčistili. Ďalšiu skutočnosť v tomto kontexte si môžete všimnúť na riadkoch 239 a 249, kde očakávame načítanie znaku „A alebo N“ od používateľa. Aj v tomto prípade by bolo vhodné pred načítaním vyčistiť vstupný buffer. Namiesto toho sme pred použitým formátom *%c* použili znak medzery. To umožní preskočenie všetkých netlačiteľných znakov, ktoré sa môžu nachádzať vo vstupnom prúde. Takéto ošetrenie by v prípade korektného zadávania údajov používateľom malo byť dostačujúce. Nejde však o ideálne riešenie.

Predstavte si situáciu, že pred vykonaním riadku 239 mohol používateľ cez konzolu zadať nejaký údaj a tento potvrdil znakom entera. Požadované údaje sa uložili do príslušnej premennej, avšak znak enteru ostal visieť vo vstupnom prúde. Ak by sme znak medzery pred formátom nepoužili, tak by sa do premennej *volba* načítal práve tento enter, keďže ide o znak. Správanie programu by bolo také, že používateľovi by ani nebolo umožnené zadať svoju odpoveď a program by sa zastavil až na riadku 249, keďže podmienka na riadku 240 by sa nesplnila. Ak by sa vo vstupnom prúde nachádzali aj tlačiteľné znaky, využitie medzery pred formátom by nebolo dostačujúce. Vtedy by správanie programu mohlo byť stochastické (nepredvídateľné) v závislosti od interakcie používateľa s programom. Preto by bolo potrebné vyčistiť vstupný prúd.

Jedna z možných implementácií:

```
229  void zmena(LISTnode *h)
230  {
231      char volba;
232      if (!(prazdny(h)))
233      {
234          LISTnode *ten = najdi(h);    // najde prislusny prvok
235
```

```
236         if (ten != NULL)   // zmena udajov
237         {
238             printf("\n\nChes zmenit meno? A/N \n");
239             scanf(" %c",&volba);
240             if (volba == 'A' || volba == 'a')
241             {
242                 printf("\nZadaj nove meno: ");
243                 while (getchar() != '\n')
244                     ;
245                 gets(ten->meno);
246                 printf("\nNove meno bolo ulozene. ");
247             }
248             printf("\nChes zmenit body? A/N \n");
249             scanf(" %c",&volba);
250             if (volba == 'A' || volba == 'a')
251             {
252                 printf("Zadaj novy pocet bodov: ");
253                 scanf("%d",&(ten->body));
254                 printf("\nNovy pocet bodov bol ulozeny. ");
255                 getchar();
256             }
257         }
258         while (getchar() != '\n')
259             ;
260     } //if prazdny
261     else
262     {
263         printf("Zoznam je prazdny.\n");
264         printf("Najprv musis pridat nejaky prvok, aby si ho mohol modifikovat.\n");
265     }
266     getchar();
267 }
```

1.3.4.7 Uvoľnenie zreťazeného zoznamu z pamäte

Uzly zreťazeného zoznamu vznikajú dynamicky a preto je na vývojárovi, aby sa postaral o uvoľnenie alokovaného priestoru v prípade, ak už nepotrebuje túto pamäť. Pre tento účel sme definovali procedúru *zrus()*, ktorá preberá dva parametre. Začiatok zreťazeného zoznamu a jeho koniec. Oba parametre sú odovzdané odkazom, keďže očakávame, že sa všetky zmeny vykonané procedúrou aplikujú na skutočný zreťazený zoznam. Po overení či sa v zreťazenom zozname nachádzajú nejaké prvky (riadok 175), prechádzame zreťazeným zoznamom uzol po uzle a pomocou funkcie *free()* sa staráme o uvoľnenie pamäťového priestoru. Na riadku 183 pointer na posledný prvok naplníme obsahom pointera na prvý prvok, ktorý nadobúda hodnotu *NULL*. Tým signalizujeme, že zreťazený zoznam je prázdny.

Volanie funkcie v tele programu má podobu:

```
zrus(&head, &tail);
```

Ako skutočný parameter odovzdávame adresu pointerov *head* a *tail*. Z toho dôvodu musia byť formálne parametre funkcie deklarované ako dvojité pointery.

Jedna z možných implementácií:

```
172  void zrus(LISTnode **h, LISTnode **t)
173  {
174      LISTnode *p;           //pomocny pointer
175      if (!(prazdny(*h)))
176      {
177          while  (*h != NULL)
178          {
179              p = *h;
180              (*h) = (*h)->next;
181              free((void*)p);
182          }
183          *t = *h;     // ukazovatele h, t nastavime na NULL
184
185          printf("\nZoznam je zruseny!\n");
186      }
187      else
188      {
189          printf("Zoznam neobsahuje ziadny prvok. Nie je co odstranovat.\n");
190      }
191      getchar();
192  }
```

1.3.4.8 Pridanie prvku do zreťazeného zoznamu

Pridanie nového prvku do zreťazeného zoznamu je možné realizovať viacerými spôsobmi. Medzi ten najjednoduchší spôsob patrí pridanie na začiatok, alebo na koniec zoznamu. Mierne zložitejší koncept predstavuje pridanie prvku za/pred konkrétny prvok.

1.3.4.8.1 Pridanie prvku na koniec zoznamu

Ak nie je špecifikované pravidlo, ktoré by určovalo akým spôsob sa budú nové prvky pridávať do zreťazeného zoznamu, volí sa buď forma pridania na začiatok, alebo na koniec. V našej implementácii sme zvolili formu pridávania na koniec. Funkcia *pridaj()* obsahuje dva formálne parametre, ktoré sú odovzdané odkazom, pretože chceme aby sa zmeny realizované vo funkcii odzrkadlili v zreťazenom zozname, ktorý sme vytvorili v main-e.

Volanie funkcie v tele programu má podobu:

```
pridaj(&head, &tail);
```

Ako skutočný parameter odovzdávame adresu pointerov *head* a *tail*. Z toho dôvodu musia byť formálne parametre funkcie deklarované ako dvojité pointery. Vo funkcii riešime dva stavy, ktoré môžu nastať:

- pridanie prvku do prázdneho zreťazeného zoznamu,
- pridanie prvku do zreťazeného zoznamu, ktorý už obsahuje nejaké prvky.

Našou snahou je aj ilustrovať rôzny spôsob práce s dvojitým a jednoduchým pointerom. Z tohto dôvodu, ak je zreťazený zoznam prázdny (otestujeme či je obsah dvojitého pointera *h*, ktorý predstavuje začiatok zoznamu rovný *NULL*), alokujeme pamäť pre nový uzol priamo do tohto pointera (riadok 101). Ak táto podmienka neplatí, nový uzol vytvárame do jednoduchého pomocného pointera *p* (riadok 124). Po otestovaní úspešnej alokácie sa postaráme o naplnenie dátovej časti uzla.

Na riadkoch 109 a 132 dochádza k načítavaniu mena študenta pomocou funkcie *gets()*. Ako sme objasnili v podkapitole 1.3.4.5, toto nie je ideálna implementácia. Ponechávame na čitateľovi vylepšenie tejto časti.

Pri načítaní bodového hodnotenia študenta sme pre ošetrenie korektnej hodnoty z pohľadu dátového typu využili návratovú hodnotu funkcie *scanf()*. Pri popise tejto štandardnej funkcie je uvedené, že funkcia vráti počet úspešne prečítaných položiek, v opačnom prípade vráti *EOF*. Na základe uvedeného by sme mohli predpokladať, že v prípade úspechu, t. j. v tomto prípade načítania jednej celočíselnej hodnoty vráti hodnotu 1. A pri pokuse vloženia desatinnej hodnoty vráti hodnotu rôznu od 1. V skutočnosti však tomu tak nie je. Funkcia *scanf()* umožňuje formátované načítanie a v prípade zadania reálnej hodnoty používateľom sa zoberie len jej celočíselná hodnota, zvyšná časť desatinného čísla ostáva vo vstupnom buffery. Funkcia *scanf()* aj v tomto prípade vráti hodnotu 1. Takéto ošetrenie nám teda zabezpečí bezchybný chod len v prípade, ak by sa používateľ snažil vložiť do tejto premennej znak alebo reťazec. Celé načítanie je zapuzdrené do cyklu s podmienkou na začiatku. Preto je nutné inicializovať pomocnú premennú *pom* na akúkoľvek celočíselnú hodnotu rôznu od 1. My sme ju inicializovali na hodnotu 0. Tomuto ťažkopádnemu konceptu, ktorý označujeme za nečisté programovanie, s ktorým sa pri riešeniach od študentov veľmi často stretáme, sa vieme vyhnúť použitím cyklu s podmienkou na konci, ktorý je pre túto situáciu vhodnejší. Je to preto, že očakávame minimálne jedno načítanie hodnoty, ktoré predstavuje bodové hodnotenie študenta. V takom prípade nie je ani nutné inicializovať pomocnú premennú *pom*, a načítanie premennej spolu s ošetrením by vyzeralo:

```
do
{
  printf("Pocet bodov: ");
  fflush(stdin);
  pom = scanf("%d", &(temp->body));
}
while(pom != 1);
```

Tomuto ošetreniu však chýba ošetrenie z pohľadu obsahu načítavanej hodnoty. Je logické predpokladať, že bodové hodnotenie musí obsahovať kladnú hodnotu. Aj keď nie vždy to je pravda. Záleží od nastaveného systému hodnotenia študentov. My však zvažujeme teraz tento predpoklad.

Preto by sme mali upraviť podmienku podmieneného načítavania takto:

```
do
{
  printf("Pocet bodov: ");
  fflush(stdin);
  pom = scanf("%d", &(temp->body));
}
while(pom != 1 || temp->body <= 0);
```

Ako sme uviedli v úvode, našim cieľom je sústrediť sa na prácu so zreťazeným zoznamom a preto ďalšie vylepšenia takýchto a obdobných skutočností ponechávame na čitateľa.

Po úspešnom vytvorení a naplnení uzla pridávaného do zreťazeného zoznamu nám ostáva postarať sa o korektné vzájomné prepojenia. V prípade ak ide o pridanie uzla do prázdneho zreťazeného zoznamu, stačí ak ukazovateľa na predchodcu a nasledovníka nastavíme na *NULL* (riadky 116 a 117) a adresu tohto uzla uložíme do pointera predstavujúceho začiatok, ako aj koniec zreťazeného zoznamu. V tomto prípade stačilo už len hodnotu pointera *h* uložiť do pointera *t*, keďže sme uzol vytvorili priamo do pointera *h* (riadok 118).

Ak v zreťazenom zozname už nejaké uzly existujú, pridávame nový uzol na koniec. Uzol sme si vytvorili do pomocného pointera *p*, deklarovaného na riadku 123. Po naplnení uzla nastavíme nasledovníka na hodnotu *NULL* (riadok 139). Predchodcu reprezentuje posledný uzol zreťazeného zoznamu, ktorého adresa je uložená v pointeri *t* (riadok 140). Upravíme nasledovníka tohto uzla, ktorým sa stáva novo pridaný uzol (riadok 141). Ako posledný krok je nutné aktualizovať chvost zreťazeného zoznamu, ktorý teraz reprezentuje novo pridaný uzol (riadok 142). Poradie jednotlivých krokov je niekedy bezvýznamné, niekedy mu však musíme venovať patričnú pozornosť, aby sme neprišli o údaje, ktoré potrebujeme. Napríklad v prípade, ak by sme ako prvé aktualizovali chvost zreťazeného zoznamu na nový uzol, prišli by sme tak o adresu posledného prvku v zreťazenom zozname a nevedeli by sme zrealizovať jednotlivé prepojenia. Sekvencia príkazov je v tomto prípade veľmi dôležitá.

Na riadkoch 145 a 146 je uvedené čistenie vstupného buffera s využitím cyklu bez tela. Tieto riadky je možné nahradiť volaním funkcie *fflush(stdin)*, ktorá však nie je štandardnou funkciou a nemusí byť funkčná na každom operačnom systéme, ako sme už uviedli v podkapitole 1.3.4.5. Riešenie tohto problému ponechávame na čitateľovi.

Jedna z možných implementácií:

```
 96  void pridaj(LISTnode **h, LISTnode **t)
 97  {
 98      int pom = 0;
 99      if (*h == NULL)     //pridanie prveho prvku do prazdneho zoznamu
100      {
101          if((*h = (LISTnode *) malloc(sizeof(LISTnode))) == NULL)
102          {
103              printf("Malo pamate.");
104          }
105          else
106          {
107              printf("Meno studenta: ");
108              fflush(stdin);
109              gets((*h)->meno);
110              while(pom != 1)
111              {
112                  printf("Pocet bodov: ");
113                  fflush(stdin);
114                  pom=scanf("%d", &((*h)->body));
115              }
116              (*h)->next = NULL;
117              (*h)->prev = NULL;
118              *t = *h;
119          }
120      }
121      else               //pridanie dalsich prvkov
122      {
123          LISTnode *p;
124          if ((p = (LISTnode *)malloc(sizeof(LISTnode)))==NULL)
125          {
126              printf("Malo pamate.");
127          }
128          else
129          {
130              printf("Meno studenta:");
131              fflush(stdin);
132              gets(p->meno);
133              while(pom != 1)
134              {
135                  printf("Pocet bodov: ");
136                  fflush(stdin);
137                  pom = scanf("%d", &(p->body));
138              }
139              p->next = NULL;
140              p->prev = (*t);
141              (*t)->next = p;
```

```
142             (*t) = p;
143         }
144     }
145     while (getchar() != '\n') /* vyprazdnenie buffera */
146         ;
147 }
```

Vo funkcii vyššie sme sa snažili ilustrovať rozdielnu prácu s jednoduchým a dvojitým pointerom. V skutočnosti môžeme túto istú funkcionalitu riešiť aj bez použitia pomocného pointera *p*, ktorý je možné v tomto prípade nahradiť `(*t)->next`, t. j. nový uzol vytvoríme priamo ako nasledovníka posledného uzla. Následne je však potrebné upraviť prácu s danými pointermi tak, ako ilustruje kód nižšie. Je viac menej na vývojárovi, ktorý koncept mu je bližší.

```
121     else    //pridanie dalsich prvkov
122     {
123         if (((*t)->next = (LISTnode *)malloc(sizeof(LISTnode)))==NULL)
124         {
125             printf("Malo pamate.");
126         }
127         else
128         {
129             printf("Meno studenta:");
130             gets((*t)->next->meno);
131             while(pom != 1)
132             {
133                 printf("Pocet bodov: ");
134                 fflush(stdin);
135                 pom = scanf("%d", &((*t)->next->body));
136             }
137             (*t)->next->prev = (*t);
138             (*t) = (*t)->next;
139             (*t)->next = NULL;
140         }
141     }
```

Pri zohľadnení skutočnosti, že v prípade pridania uzla pôjde vždy o vytvorenie uzla a jeho naplnenie, mohli by sme koncept funkcionality funkcie *pridaj()* modifikovať spôsobom, aby sme nepodporovali redundantnosť (nadbytočnosť) kódu vzhľadom na tieto skutočnosti, t. j. neriešili vytvorenie a naplnenie uzla pri pridávaní do prázdneho zreťazeného zoznamu, ako aj pri pridávaní prvku do zreťazeného zoznamu s prvkami. Namiesto toho by sme uzol vytvorili do pomocného pointera, napr. *temp* a cez neho ho aj naplnili. Správne nastavenie väzieb medzi uzlami by sme riešili v jednotlivých stavoch, ktoré môžu nastať. V takomto prípade by sme koncept funkcie modifikovali tak, ako ilustruje kód nižšie:

```
 96  void pridaj(LISTnode **h, LISTnode **t)
 97  {
 98      int pom=0;
 99      LISTnode * temp = (LISTnode *) malloc(sizeof(LISTnode));
100      if(temp == NULL)
101      {
102          printf("Malo pamate.");
103          return;
104      }
105
106      printf("Meno studenta: ");
107      fflush(stdin);
108      fgets(temp->meno, 30, stdin);
109
110      do
111      {
112          printf("Pocet bodov: ");
113          fflush(stdin);
114          pom=scanf("%d", &(temp->body));
115      }
116      while(pom!=1 || temp->body <=0);
117
118      if (*h == NULL)     //pridanie prveho prvku do prazdneho zoznamu
119      {
120          (*h) = temp;
121          (*h)->next = NULL;
122          (*h)->prev = NULL;
123          *t = *h;
124      }
125      else               //pridanie dalsich prvkov
126      {
127          temp->next = NULL;
128          temp->prev = (*t);
129          (*t)->next = temp;
130          (*t) = temp;
131      }
132      while (getchar() != '\n') // vyprazdnenie buffera
133          ;
134  }
```

1.3.4.8.2 Pridanie prvku pred konkrétny prvok

V určitých situáciách, napr. v prípade budovania zreťazeného zoznamu podľa určitého pravidla, môže byť nutné zabezpečiť operáciu vkladania prvku pred konkrétny prvok. Funkcia *pridaj_presne_pred()* obsahuje dva formálne parametre, ktoré sú odovzdané odkazom.

Volanie funkcie v tele programu má podobu:

```
pridaj_presne_pred(&head, &tail);
```

Ako skutočný parameter odovzdávame adresu pointerov *head* a *tail*. Vo funkcii riešime dva stavy, ktoré môžu nastať a je ich nutné implementovať rôzne:

- pridanie prvku na začiatok zreťazeného zoznamu,
- pridanie prvku pred akýkoľvek ďalší prvok zreťazeného zoznamu.

Po overení či zreťazený zoznam obsahuje nejaké prvky (riadok 273) sa postaráme o vyhľadanie prvku, pred ktorý chceme nový prvok vložiť (riadok 275). Po otestovaní či sa nám prvok podarilo nájsť, testujeme, či je tento prvok (jeho adresa) rôzny od prvého prvku. Ak nie je táto podmienka splnená, ide o pridanie nového prvku na začiatok zreťazeného zoznamu a vykoná sa časť kódu na riadkoch 300 – 314. V tomto prípade bude predchodca tohto prvku nastavený na hodnotu *NULL*, nasledovníkom bude uzol, ktorého adresa je uložená v pointery *ten*. Postaráme sa o prepojenie novo pridaného uzla s prvým prvkom, ktorý sa stane jeho predchodcom. Na záver sa postaráme o aktualizáciu pointera *h*, ktorý predstavuje začiatok zreťazeného zoznamu.

V prípade pridania nového uzla pred akýkoľvek prvok zreťazeného zoznamu, vrátane posledného je potrebné zabezpečiť korektne štyri väzby medzi danými uzlami. Aby sme neprišli o ukazovateľ na uzol nachádzajúci sa pred uzlom prístupným cez pointer *ten*, postaráme sa najprv o tieto väzby. Následne nastavíme väzby medzi novo pridaným prvkom, a prvkom pred ktorý tento vkladáme (riadky 292 –295).

Pozn. autora: Kód je možné optimalizovať a redukovať duplicitu kódu spôsobom, ako bolo vysvetlené v podkapitole 1.3.4.8.1. Respektíve môže dôjsť k zmene konceptu návrhu funkcií. Ak by sme mali funkciu, ktorá zabezpečuje pridanie prvku na začiatok zreťazeného zoznamu, mohli by sme využiť jej volanie vo vnútri tejto funkcie.

Jedna z možných implementácií:

```
270  void pridaj_presne_pred(LISTnode **h, LISTnode **t)
271  {
272      LISTnode *p;
273      if (!(prazdny(*h)))
274      {
275          LISTnode *ten = najdi((*h));   //najde prislusny prvok
276
277          if( ten != NULL)
278          {
279              if (ten != (*h))   //ak nepridavam na zaciatok
280              {
281                  if(( p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
282                  {
283                      printf("Malo pamate.");
284                  }
```

```
285                 else
286                 {
287                     printf("Meno noveho studneta:");
288                     fflush(stdin);
289                     gets(p->meno);
290                     printf("Pocet bodov noveho studenta: ");
291                     scanf("%d",&(p->body));
292                     ten->prev->next = p;
293                     p->prev = ten->prev;
294                     p->next = ten;
295                     ten->prev = p;
296                 }
297             }
298             else          //pridavam na zaciatok
299             {
300                 if(( p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
301                 {
302                     printf("Malo pamate.");
303                 }
304                 else
305                 {
306                     printf("Meno noveho studneta:");
307                     fflush(stdin);
308                     gets(p->meno);
309                     printf("Pocet bodov noveho studenta: ");
310                     scanf("%d",&(p->body));
311                     ten->prev = p;
312                     p->prev = NULL;
313                     p->next = ten;
314                     (*h) = p;
315                 }
316             }
317         } //if
318         while (getchar() != '\n') // vyprazdnenie buffera
319             ;
320     } //if prazdny
321     else
322     {
323         printf("Zoznam je prazdny.\n");
324         printf("Najprv musis pridat nejaky prvok, aby si mohol pridavat uzly pred neho.\n");
325     }
326     getchar();
327 }
```

1.3.4.8.3 Pridanie prvku za konkrétny prvok

Pre prípad pridania prvku za konkrétny prvok sme definovali funkciu *pridaj_presne_za()*, ktorá obsahuje dva formálne parametre, ktoré sú odovzdané odkazom.

Volanie funkcie v tele programu má podobu:

```
pridaj_presne_za(&head, &tail);
```

Ako skutočný parameter odovzdávame adresu pointerov *head* a *tail*. Vo funkcii riešime dva stavy, ktoré môžu nastať a je ich nutné implementovať rôzne:

- pridanie prvku na koniec zreťazeného zoznamu,
- pridanie prvku za akýkoľvek iný prvok zreťazeného zoznamu.

Po overení, či zreťazený zoznam obsahuje nejaké prvky (riadok 333), postaráme sa o vyhľadanie prvku, za ktorý chceme nový prvok vložiť (riadok 335). Po otestovaní, či sa nám prvok podarilo nájsť, testujeme, či sa tento prvok (jeho adresa) líši od posledného prvku. Ak nie je táto podmienka splnená, ide o pridanie nového prvku na koniec zreťazeného zoznamu a vykoná sa časť kódu na riadkoch 359 – 368. Nový uzol predstavuje nasledovníka posledného uzla, prístupného cez pointer *ten* (riadok 364). V tomto prípade bude nasledovník pridávaného uzlu nastavený na hodnotu *NULL* (riadok 365), predchodcom bude uzol, ktorého adresa je uložená v pointeri *ten* (riadok 366). Na záver sa postaráme o aktualizáciu pointera *t*, ktorý predstavuje koniec zreťazeného zoznamu.

V prípade pridania nového uzla za akýkoľvek iný prvok zreťazeného zoznamu ako je koniec, vrátane prvého, je potrebné zabezpečiť korektne štyri väzby medzi danými uzlami. Aby sme neprišli o ukazovateľ na uzol nachádzajúci sa za uzlom prístupným cez pointer *ten*, postaráme sa najprv o tieto väzby (riadky 351 a 352). Následne nastavíme väzby medzi novo pridaným prvkom a prvkom za ktorý tento vkladáme (riadky 353 a 354). Na riadkoch 355 a 368 voláme funkciu *vypis()*, ktorá zobrazí obsah daného zreťazeného zoznamu, ako formu kontroly korektnosti implementovanej operácie.

Pozn. autora: Kód je možné optimalizovať a redukovať duplicitu kódu spôsobom, ako bolo vysvetlené v podkapitole 1.3.4.8.1. Respektíve môže dôjsť k zmene konceptu návrhu funkcií. Ak by sme mali funkciu, ktorá zabezpečuje pridanie prvku na koniec zreťazeného zoznamu, mohli by sme využiť jej volanie vo vnútri tejto funkcie.

Jedna z možných implementácií:

```
330  void pridaj_presne_za(LISTnode **h, LISTnode **t)
331  {
332      LISTnode *p;
333      if (!(prazdny(*h)))
334      {
335          LISTnode *ten = najdi((*h));   // najde prislusny prvok
336          if(ten != NULL)
337          {
338              if((p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
339              {
340                  printf("Malo pamate.");
341              }
```

```
342             else
343             {
344                 if (ten != (*t))    //ak nepridavam na koniec
345                 {
346                     printf("Meno noveho studneta:");
347                     fflush(stdin);
348                     gets(p->meno);
349                     printf("Pocet bodov noveho studenta: ");
350                     scanf("%d",&(p->body));
351                     ten->next->prev = p;
352                     p->next = ten->next;
353                     p->prev = ten;
354                     ten->next = p;
355                     vypis((*h));
356                 }
357                 else
358                 {
359                     printf("Meno noveho studneta:");
360                     fflush(stdin);
361                     gets(p->meno);
362                     printf("Pocet bodov noveho studenta: ");
363                     scanf("%d",&(p->body));
364                     ten->next = p;
365                     p->next = NULL;
366                     p->prev = ten;
367                     (*t) = p;
368                     vypis((*h));
369                 }
370             } //else
371         } // if
372         while (getchar() != '\n') // vyprazdnenie buffera
373             ;
374     } //if prazdny
375     else
376     {
377         printf("Zoznam je prazdny.\n");
378         printf("Najprv musis pridat nejaky prvok, aby si mohol pridavat uzly za neho.\n");
379     }
380     getchar();
381 }
```

1.3.4.9 Odstránenie prvku zo zreťazeného zoznamu

Pri odstraňovaní prvku zo zreťazeného zoznamu je potrebné zvažovať päť rôznych situácií:

- zreťazený zoznam neobsahuje prvky – nemáme čo odstraňovať,
- ak zreťazený zoznam obsahuje práve jeden prvok – po odstránení prvku musí *head* aj *tail* ostať nastavené na *NULL*,
- ak odstraňujeme prvý prvok zreťazeného zoznamu,

- ak odstraňujeme posledný prvok zreťazeného zoznamu,
- ak odstraňujeme prvok vo vnútri zreťazeného zoznamu.

Funkcia *odstran_presne()* obsahuje dva formálne parametre, ktoré sú odovzdané odkazom, pretože chceme, aby sa zmeny realizované vo funkcii odzrkadlili v zreťazenom zozname, ktorý sme vytvorili v main-e.

Volanie funkcie v tele programu má podobu:

```
odstran_presne(&head, &tail);
```

Ako skutočný parameter odovzdávame adresu pointerov *head* a *tail*. Z toho dôvodu musia byť formálne parametre funkcie deklarované ako dvojité pointery. Volanie funkcie realizujeme aj na konci programu (riadok 81), pretože sa nikdy nespoliehame na používateľa, že sa postará o uvoľnenie zreťazeného zoznamu pred ukončením práce s programom.

V tele funkcie, ako prvé pomocou funkcie *prazdny()* testujeme, či sa v zreťazenom zozname nachádzajú nejaké prvky. Ak nie, zareagujeme používateľovi výpisom. V opačnom prípade sa postaráme o vyhľadanie uzla, ktorý chceme odstrániť (riadok 388). Testujeme či sme daný uzol našli a pokračujeme kontrolou, či je tento uzol rôzny od prvého a posledného prvku. Ak je tomu tak, ide o odstraňovanie uzla z vnútra zreťazeného zoznamu a je potrebné sa postarať o prepojenie predchodcu a nasledovníka odstraňovaného uzla (riadky 395 a 396). Ešte pred uvoľnením samotného uzla výpisom informujeme používateľa, aký uzol odstraňuje. Uzol uvoľníme pomocou funkcie *free()*.

Ak ide o odstránenie prvého uzla, skôr ako začneme nastavovať správne väzby, skontrolujeme, či sa v zreťazenom zozname nenachádza len jeden prvok. Túto skutočnosť testujeme na základe skúmania hodnoty nasledovníka tohto uzla. Ak je hodnota *NULL*, vieme, že v zreťazenom zozname je len jeden prvok, a v takomto prípade uvoľníme daný uzol (riadok 405), a pointery *h* a *t* nastavíme na hodnotu *NULL* (riadok 406) a funkciu ukončíme (riadok 407). V opačnom prípade je potrebné sa postarať o aktualizáciu pointera *h*, ktorý predstavuje ukazovateľ na prvý prvok zreťazeného zoznamu (riadok 409). Predchodca tohto uzla bude prirodzene nastavený na hodnotu *NULL* (410). Ostatné operácie sú obdobné, ako v prvom prípade.

Ak nejde o odstraňovanie prvého prvku a vylúčili sme aj možnosť, že nejde o odstraňovanie z vnútra zreťazeného zoznamu, môžeme predpokladať, že ide o odstránenie posledného prvku. V takomto prípade je potrebné sa postarať o aktualizáciu pointera *t* na predchodcu odstraňovaného uzla (riadok 417). Ukazovateľ na nasledovníka tohto uzla nastavíme na hodnotu *NULL* (riadok 418).

Jedna z možných implementácií:

```
384 void odstran_presne(LISTnode **h, LISTnode **t)
385 {
386     if (!(prazdny(*h)))
387     {
388         LISTnode *ten = najdi((*h));    // najde prislusny prvok
389 
390         //ak bol prvok ktory sa ma odstranit najdeny
391         if( ten != NULL)
392         {
393             if (ten != (*h) && ten != (*t))     //ak neodstranujem zo zaciatku ani z konca
394             {
395                 ten->prev->next = ten->next;
396                 ten->next->prev = ten->prev;
397                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
398                 getchar();
399                 free((void *)ten);
400             }
401             else if (ten == (*h))       //ak odstranujem zo zaciatku
402             {
403                 if (ten->next == NULL) //ak je v zozname prave jeden prvok
404                 {
405                      free((void *)ten);
406                      (*h) = (*t) = NULL;
407                      return;
408                 }
409                 (*h) = ten->next;
410                 ten->next->prev = NULL;
411                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
412                 getchar();
413                 free((void *)ten);
414             }
415             else    //ak odstranujem z konca
416             {
417                 (*t) = ten->prev;
418                 ten->prev->next = NULL;
419                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
420                 getchar();
421                 free((void *)ten);
422             }
423         } //if
424     } //if prazdny
425     else
426     {
427         printf("Zoznam je prazdny.\n");
428         printf("Najprv musis pridat nejaky prvok, aby si ho mohol odstranovat.\n");
429     }
430     getchar();
431 }
```

Jedna z možných implementácií celého programu 1.3.4.10_Pr_1.c:

```c
/* pripojenie potrebnych hlavickovych suborov */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* definicia noveho typu */
typedef struct node
{
    char meno[30];
    int body;
    struct node *next;
    struct node *prev;
} LISTnode;

/* deklaracie funkcii */
int prazdny(LISTnode *h);
void pridaj(LISTnode **h, LISTnode **t);
void vypis(LISTnode *p);
void zrus(LISTnode **h, LISTnode **t);
LISTnode *najdi(LISTnode *p);
void zmena(LISTnode *h);
void pridaj_presne_pred(LISTnode **h, LISTnode **t);
void pridaj_presne_za(LISTnode **h, LISTnode **t);
void odstran_presne(LISTnode **h, LISTnode **t);


int main(void)
{
    LISTnode *head, *tail;
    int volba;

    head = tail = NULL;

    do
    {
        system("cls");
        printf("\n***OBOJSMERNY LINEARNY ZOZNAM***\n\n");
        printf("Pridaj prvok.......................... 1\n");
        printf("Vloz prvok pred iny prvok............. 2\n");
        printf("Vloz prvok za iny prvok............... 3\n");
        printf("Najdi prvok........................... 4\n");
        printf("Zmen konkretny prvok.................. 5\n");
        printf("Vypis cely obsah...................... 6\n");
        printf("Odstran konkretny prvok .............. 7\n");
        printf("Zrus cely zoznam...................... 8\n");
        printf("Koniec................................ k\n\n");

        printf("\nCo ideme robit? Zvol cislo z ponuky: \n\n");
        switch (volba = getchar())
        {
        case '1':
            pridaj(&head, &tail);
            break;
```

```
54          case '2':
55              pridaj_presne_pred(&head, &tail);
56              break;
57          case '3':
58              pridaj_presne_za(&head, &tail);
59              break;
60          case '4':
61              najdi(head);
62              break;
63          case '5':
64              zmena(head);
65              break;
66          case '6':
67              vypis(head);
68              break;
69          case '7':
70              odstran_presne(&head, &tail);
71              break;
72          case '8':
73              zrus(&head, &tail);
74              break;
75          default:
76              break;
77          }
78      }
79      while (volba != 'k');
80
81      zrus(&head, &tail);
82      system("pause");
83      return(0);
84  }
85
86  /* definicie funkcii */
87
88  /* overenie ci je zoznam prazdny */
89  int prazdny(LISTnode *h)
90  {
91      /* vrati "true=1" ak je zoznam prazdny */
92      return(h == NULL);
93  }
94
95  /* prida prvok na koniec ak sa v zozname uz nejake uzly nachadzaju */
96  void pridaj(LISTnode **h, LISTnode **t)
97  {
98      int pom = 0;
99      if (*h == NULL)     //pridanie prveho prvku do prazdneho zoznamu
100     {
101         if((*h = (LISTnode *) malloc(sizeof(LISTnode))) == NULL)
102         {
103             printf("Malo pamate.");
104         }
105         else
106         {
107             printf("Meno studenta: ");
108             fflush(stdin);
```

```
109             gets((*h)->meno);
110             while(pom != 1)
111             {
112                 printf("Pocet bodov: ");
113                 fflush(stdin);
114                 pom=scanf("%d", &((*h)->body));
115             }
116             (*h)->next = NULL;
117             (*h)->prev = NULL;
118             *t = *h;
119         }
120     }
121     else              //pridanie dalsich prvkov
122     {
123         LISTnode *p;
124         if ((p = (LISTnode *)malloc(sizeof(LISTnode)))==NULL)
125         {
126             printf("Malo pamate.");
127         }
128         else
129         {
130             printf("Meno studenta:");
131             fflush(stdin);
132             gets(p->meno);
133             while(pom != 1)
134             {
135                 printf("Pocet bodov: ");
136                 fflush(stdin);
137                 pom = scanf("%d", &(p->body));
138             }
139             p->next = NULL;
140             p->prev = (*t);
141             (*t)->next = p;
142             (*t) = p;
143         }
144     }
145     while (getchar() != '\n') /* vyprazdnenie buffera */
146         ;
147 }
148
149 /* vypisanie obsahu zoznamu */
150 void vypis(LISTnode *p)
151 {
152     if (!(prazdny(p)))
153     {
154         printf("VYPIS\n");
155
156         while (p != NULL)
157         {
158             printf("\nMeno: %s", p->meno);
159             printf("\nBody: %d \n",p->body);
160             p = p->next;     //posun na dalsi prvok
161             getchar();
162         }
163     }
```

```
164     else
165     {
166         printf("Zoznam neobsahuje ziadny prvok.\n");
167     }
168     getchar();
169 }
170 
171 /* zrusenie zoznamu */
172 void zrus(LISTnode **h, LISTnode **t)
173 {
174     LISTnode *p;           //pomocny pointer
175     if (!(prazdny(*h)))
176     {
177         while  (*h != NULL)
178         {
179             p = *h;
180             (*h) = (*h)->next;
181             free((void*)p);
182         }
183         *t = *h;     // ukazovatele h, t nastavime na NULL
184 
185         printf("\nZoznam je zruseny!\n");
186     }
187     else
188     {
189         printf("Zoznam neobsahuje ziadny prvok. Nie je co odstranovat.\n");
190     }
191     getchar();
192 }
193 
194 /* vyhladanie konkretneho uzla */
195 LISTnode *najdi(LISTnode *p)
196 /*
197    vrati pointer na prvok, ktory vyhovuje poziadavke
198    resp. NULL ak je zoznam prazdny alebo sa taky prvok v zozname nenachadza
199 */
200 {
201     if (!(prazdny(p)))
202     {
203         char m[30];
204         printf("Hladaj studenta: ");
205         fflush(stdin);
206         gets(m);
207 
208         while ((p !=  NULL) && (strcmp(p->meno, m)))
209         {
210             p = p->next;
211         }
212 
213         if (p ==  NULL)
214             printf("Nenasiel sa takyto zaznam!");
215         else
216             printf("Hladany student sa nasiel! Ma %d bodov\n",p->body);
217     } //if prazdny
218     else
```

```
219     {
220         printf("Zoznam je prazdny.\n");
221         printf("Najprv musis pridat nejaky prvok, aby si ho mohol vyhladat.\n");
222     }
223 
224     getchar();
225     return(p);
226 }
227 
228 /* zmena udajov konkretneho uzla */
229 void zmena(LISTnode *h)
230 {
231     char volba;
232     if (!(prazdny(h)))
233     {
234         LISTnode *ten = najdi(h);    // najde prislusny prvok
235 
236         if (ten != NULL)  // zmena udajov
237         {
238             printf("\n\nChes zmenit meno? A/N \n");
239             scanf(" %c",&volba);
240             if (volba == 'A' || volba == 'a')
241             {
242                 printf("\nZadaj nove meno: ");
243                 while (getchar() != '\n')
244                     ;
245                 gets(ten->meno);
246                 printf("\nNove meno bolo ulozene. ");
247             }
248             printf("\nChes zmenit body? A/N \n");
249             scanf(" %c",&volba);
250             if (volba == 'A' || volba == 'a')
251             {
252                 printf("Zadaj novy pocet bodov: ");
253                 scanf("%d",&(ten->body));
254                 printf("\nNovy pocet bodov bol ulozeny. ");
255                 getchar();
256             }
257         }
258         while (getchar() != '\n')
259             ;
260     } //if prazdny
261     else
262     {
263         printf("Zoznam je prazdny.\n");
264         printf("Najprv musis pridat nejaky prvok, aby si ho mohol modifikovat.\n");
265     }
266     getchar();
267 }
268 
269 /* pridanie prvku pred konkretny prvok */
270 void pridaj_presne_pred(LISTnode **h, LISTnode **t)
271 {
272     LISTnode *p;
273     if (!(prazdny(*h)))
```

```
274     {
275         LISTnode *ten = najdi((*h));   //najde prislusny prvok
276 
277         if( ten != NULL)
278         {
279             if (ten != (*h))    //ak nepridavam na zaciatok
280             {
281                 if(( p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
282                 {
283                     printf("Malo pamate.");
284                 }
285                 else
286                 {
287                     printf("Meno noveho studneta:");
288                     fflush(stdin);
289                     gets(p->meno);
290                     printf("Pocet bodov noveho studenta: ");
291                     scanf("%d",&(p->body));
292                     ten->prev->next = p;
293                     p->prev = ten->prev;
294                     p->next = ten;
295                     ten->prev = p;
296                 }
297             }
298             else          //pridavam na zaciatok
299             {
300                 if(( p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
301                 {
302                     printf("Malo pamate.");
303                 }
304                 else
305                 {
306                     printf("Meno noveho studneta:");
307                     fflush(stdin);
308                     gets(p->meno);
309                     printf("Pocet bodov noveho studenta: ");
310                     scanf("%d",&(p->body));
311                     ten->prev = p;
312                     p->prev = NULL;
313                     p->next = ten;
314                     (*h) = p;
315                 }
316             }
317         } //if
318         while (getchar() != '\n') // vyprazdnenie buffera
319             ;
320     } //if prazdny
321     else
322     {
323         printf("Zoznam je prazdny.\n");
324         printf("Najprv musis pridat nejaky prvok, aby si mohol pridavat uzly pred neho.\n");
325     }
326     getchar();
327 }
328 
```

```
329 /* pridanie prvku za konkretny prvok */
330 void pridaj_presne_za(LISTnode **h, LISTnode **t)
331 {
332     LISTnode *p;
333     if (!(prazdny(*h)))
334     {
335         LISTnode *ten = najdi((*h));   // najde prislusny prvok
336         if(ten != NULL)
337         {
338             if((p = (LISTnode *)malloc(sizeof(LISTnode))) == NULL)
339             {
340                 printf("Malo pamate.");
341             }
342             else
343             {
344                 if (ten != (*t))    //ak nepridavam na koniec
345                 {
346                     printf("Meno noveho studneta:");
347                     fflush(stdin);
348                     gets(p->meno);
349                     printf("Pocet bodov noveho studenta: ");
350                     scanf("%d",&(p->body));
351                     ten->next->prev = p;
352                     p->next = ten->next;
353                     p->prev = ten;
354                     ten->next = p;
355                     vypis((*h));
356                 }
357                 else
358                 {
359                     printf("Meno noveho studneta:");
360                     fflush(stdin);
361                     gets(p->meno);
362                     printf("Pocet bodov noveho studenta: ");
363                     scanf("%d",&(p->body));
364                     ten->next = p;
365                     p->next = NULL;
366                     p->prev = ten;
367                     (*t) = p;
368                     vypis((*h));
369                 }
370             } //else
371         } // if
372         while (getchar() != '\n') // vyprazdnenie buffera
373             ;
374     } //if prazdny
375     else
376     {
377         printf("Zoznam je prazdny.\n");
378         printf("Najprv musis pridat nejaky prvok, aby si mohol pridavat uzly za neho.\n");
379     }
380     getchar();
381 }
382
383 /* odstranenie konkretneho prvku */
```

```
384 void odstran_presne(LISTnode **h, LISTnode **t)
385 {
386     if (!(prazdny(*h)))
387     {
388         LISTnode *ten = najdi((*h));   // najde prislusny prvok
389 
390         //ak bol prvok ktory sa ma odstranit najdeny
391         if( ten != NULL)
392         {
393             if (ten != (*h) && ten != (*t))      //ak neodstranujem zo zaciatku ani z konca
394             {
395                 ten->prev->next = ten->next;
396                 ten->next->prev = ten->prev;
397                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
398                 getchar();
399                 free((void *)ten);
400             }
401             else if (ten == (*h))       //ak odstranujem zo zaciatku
402             {
403                 if (ten->next == NULL) //ak je v zozname prave jeden prvok
404                 {
405                      free((void *)ten);
406                      (*h) = (*t) = NULL;
407                      return;
408                 }
409                 (*h) = ten->next;
410                 ten->next->prev = NULL;
411                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
412                 getchar();
413                 free((void *)ten);
414             }
415             else    //ak odstranujem z konca
416             {
417                 (*t) = ten->prev;
418                 ten->prev->next = NULL;
419                 printf("Odstranujes studenta: %s s poctom bodov %d.\n",ten->meno, ten->body);
420                 getchar();
421                 free((void *)ten);
422             }
423         } //if
424     } //if prazdny
425     else
426     {
427         printf("Zoznam je prazdny.\n");
428         printf("Najprv musis pridat nejaky prvok, aby si ho mohol odstranovat.\n");
429     }
430     getchar();
431 }
432 
```

1.3.4.10 Iný pohľad na budovanie predstaveného konceptu práce so zreťazeným zoznamom

Viaceré vyššie programovacie jazyky obmedzujú priamu prácu s pointermi z dôvodu bezpečnosti a zabezpečenia stability aplikácií. Napríklad, ak chceme v jazyku *C#* pracovať priamo s pointermi, je potrebné použiť špeciálne prvky a konštrukcie jazyka – musíme označiť zdrojový kód ako „*unsafe*“. Je dôležité si uvedomiť, že práca s pointermi v *C#* je obmedzená, a nie je odporúčaná v bežných prípadoch, pretože je zdrojom častých chýb, ak vývojár dôkladne nerozumie pointerom a práci s nimi. Pointerové operácie sú rizikové a môžu viesť k chybám a nestabilite aplikácií. Väčšina programov v *C#* môže byť implementovaná bez použitia pointerov pomocou bezpečných a robustných dátových štruktúr a konštruktov jazyka *C#*. Preto sa v *C#* odporúča obmedziť používanie pointerov na prípady, ktoré vyžadujú najvyššiu výkonnosť a úzko spolupracujú s nízkoúrovňovým systémom, ako napríklad pri práci s natívnymi knižnicami alebo hardvérom.

V jazyku Java nie je priamo podporovaná práca s pointermi na rovnakej úrovni ako v jazyku *C* a *C#*. Jazyk Java bol navrhnutý s dôrazom na bezpečnosť a jednoduchosť programovania, a preto je manipulácia s pointermi obmedzená a riadená na vyššej úrovni. Java podobne ako *C#* poskytuje automatické riadenie pamäte pomocou *Garbage Collectoru*, čo znamená, že programátor nemusí explicitne spravovať alokáciu a dealokáciu pamäte. Taktiež je zabudovaná bezpečnostná vrstva, ktorá zabráni prístupu k neoprávneným pamäťovým oblastiam. Namiesto použitia pointerov tak ako v *C#*, aj v Jave sa programátor zameriava na prácu s referenciami na objekty. Referencie slúžia na odkazovanie sa na objekty v pamäti, ale nemajú priamy prístup k ich fyzickej adrese. Programátor pracuje s referenciami a používa ich na manipuláciu s objektami a volanie ich metód.

Rovnako ako v *C#*, aj v jazyku Java existuje aj tzv. "*weak reference*" (slabá referencia), ktorá poskytuje možnosť sledovať životnosť objektov. Slabé referencie nezabraňujú zberu odpadu objektov (*garbage collection*), na ktoré ukazujú. To je užitočné pri situáciách, keď chceme sledovať objekty, ale nechceme ich udržiavať nažive, ak na ne nie sú žiadne silné odkazy.

Keď je potrebné pracovať s nízkoúrovňovými operáciami alebo zabezpečiť priamy prístup k pamäti, je možné použiť tzv. "*native methods*" (natívne metódy). Tieto metódy sú implementované v iných jazykoch, ako napríklad C alebo C++, a môžu byť volané z Javy. Avšak aj pri použití natívnych metód je bezpečnosť a správne riadenie pamäte dôležitou oblasťou, ktorú je potrebné zohľadniť.

Aj programátori jazyka C v kontexte práce s pointermi sú opatrní. Napríklad sa snažia vyhýbať kombinácii dvojitých hviezdičiek, čo je potrebné napríklad pri deklarácii dvojitého pointera. Robia

to však najmä z dôvodu prehľadnosti kódu. Tento koncept ilustrujeme na tej istej funkcionalite, ktorú sme predstavili v príklade 1.3.4.1. Obmedzíme sa však len na funkcie pridania, zobrazenia obsahu a uvoľnenia zreťazeného zoznamu z pamäte. Okrem definovania vlastného typu pre uzol (riadky 9 – 15) si definujeme aj vlastný typ *p_LISTnode* (riadok 7), ktorý reprezentuje pointer na tento uzol.

Pozn. autora: Na riadkoch 13 a 14 už môžeme vidieť použitie tohto typu a skrytie jednej hviezdičky. Zápis uvedený v komentári na riadku 13 je identický so zápisom deklarácie uvedeným pred ním.

Jedna z možných implementácií:

```
6  /* definicia novych typov a struktur */
7  typedef struct node * p_LISTnode;
8
9  typedef struct node
10 {
11     char meno[30];
12     int body;
13     p_LISTnode next;     //struct node *next;
14     p_LISTnode prev;
15 } LISTnode;
```

V predchádzajúcom príklade sme pracovali len s jedným zreťazeným zoznamom a tento sme reprezentovali deklaráciou pointerov na začiatok a koniec zreťazeného zoznamu priamo v hlavnom tele programu:

```
29     LISTnode *head, *tail;
```

V tomto prípade sme pre túto deklaráciu využili nový, používateľom definovaný dátový typ *p_LIST* (riadok 17), ktorý reprezentuje pointer na pomenovanú štruktúru *LIST* (riadky 19 –23). V tejto sme zapuzdrili deklaráciu pointerov *head* a *tail*.

```
17 typedef struct LIST * p_LIST;
18
19 struct LIST
20 {
21     p_LISTnode head;
22     p_LISTnode tail;
23 };
```

Pri takomto koncepte môžeme v tele programu deklarovať viacero zreťazených zoznamov takýmto spôsobom:

```
p_LIST L1, p_LIST L2;
```

V našom príklade sme využili definíciu (riadky 34 a 35), čo znamená, že ihneď pri deklarácii zreťazených zoznamov sme ich inicializovali pomocou funkcie *LISTinit()*, ktorej cieľom je nastaviť pointery *head* a *tail* na hodnotu *NULL*. V predchádzajúcom príklade sme túto inicializáciu spravili priamo v main-e.

```
34      p_LIST L1 = LISTinit();
35      p_LIST L2 = LISTinit();
```

Jedna z možných implementácií funkcie *LISTinit()*:

```
65  p_LIST LISTinit()
66  {
67      p_LIST l = (p_LIST) malloc(sizeof(*l));
68      l->head = NULL;
69      l->tail = NULL;
70      return l;
71  }
```

Pozn. autora: V tejto implementácii sme z dôvodu stručnosti, ako aj možnosti modifikácie priamo čitateľom opomenuli testovanie alokovania pamäte. Doplnenie ponechávame čitateľovi.

Koncept funkcionality pridania prvku do zreťazeného zoznamu sme nechali rovnaký, ako v predchádzajúcom príklade. Ak je zreťazený zoznam prázdny, postaráme sa o pridanie prvku, ktorý bude predstavovať tak začiatok, ako aj koniec. V prípade, že sa v zreťazenom zozname už nejaké prvky nachádzajú, nový prvok pridáme na koniec.

Volanie funkcie *pridaj()* v tele programu má podobu:

```
pridaj(L1);
pridaj(L2);
```

Ako skutočný parameter odovzdávame *L1* alebo *L2*, ktoré sú pointerom na štruktúru *LIST*. Samotné položky tejto štruktúry sú pointerom na uzol *LISTnode*. Z definície funkcie je zrejmé, že formálny parameter, ktorý v skutočnosti je typu dvojitého pointera, pri definovaní dátového typu už neobsahuje žiadny znak *. Je tomu vďaka zapuzdreniu a definícii typov, ktoré sme zrealizovali. Samozrejme že sa mierne mení syntax príkazov pristupujúcich k uzlom a k ich jednotlivých položkám. Napríklad v prípade vloženia prvého prvku do zreťazeného zoznamu je našou snahou meno študenta uložiť do položky *meno* prvého uzla, ktorý je prístupný cez parameter funkcie *l*, položku *head* štruktúry *LIST* a položku *meno* štruktúry *node* (riadok 85).

Jedna z možných implementácií:

```
79  void pridaj(p_LIST l)
80  {
81      if (l->head == NULL)     //pridanie prveho prvku do prazdneho zoznamu
82      {
83          l->head =(LISTnode *) malloc(sizeof(LISTnode));
84          printf("Meno studenta: ");
85          gets((l->head)->meno);
86          printf("Pocet bodov: ");
87          scanf("%d", &((l->head)->body));
88          (l->head)->next = NULL;
89          (l->head)->prev = NULL;
90          l->tail = l->head;
91      }
```

```
92      else     //pridanie dalsich prvkov
93      {
94          (l->tail)->next = (LISTnode *)malloc(sizeof(LISTnode));
95          printf("Meno studenta:");
96          gets(l->tail->next->meno);
97          printf("Pocet bodov: ");
98          scanf("%d", &(l->tail->next->body));
99          l->tail->next->next = NULL;
100         l->tail->prev = l->tail;
101         l->tail = l->tail->next;
102     }
103     while (getchar() != '\n') // vyprazdnenie buffera
104         ;
105 }
```

Jedna z možných implementácií celého programu 1.3.4.11.1_Pr_1.c:

```
1   /* pripojenie potrebnych hlavickovych suborov */
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5
6   /* definicia novych typov a struktur */
7   typedef struct node * p_LISTnode;
8
9   typedef struct node
10  {
11      char meno[30];
12      int body;
13      p_LISTnode next;      //struct node *next;
14      p_LISTnode prev;
15  } LISTnode;
16
17  typedef struct LIST * p_LIST;
18
19  struct LIST
20  {
21      p_LISTnode head;
22      p_LISTnode tail;
23  };
24
25  /* deklaracie funkcii */
26  p_LIST LISTinit();
27  int prazdny(p_LIST l);
28  void pridaj(p_LIST l);
29  void vypis(p_LIST l);
30  void zrus(p_LIST l);
31
32  int main(void)
33  {
34      p_LIST L1 = LISTinit();
35      p_LIST L2 = LISTinit();
36
37      // Test pridania prvkov do dvoch vytvorenych zoznamov
38      pridaj(L1);
```

```
39      pridaj(L1);
40      pridaj(L1);
41
42      pridaj(L2);
43      pridaj(L2);
44
45      // Vypisanie obsahu oboch zoznamov
46      printf("Vypis obsahu 1. linearneho zoznamu: \n");
47      vypis(L1);
48      printf("Vypis obsahu 2. linearneho zoznamu: \n");
49      vypis(L2);
50
51      // Pridanie dalsieho prvku do prveho zoznamu
52      pridaj(L1);
53      printf("Vypis obsahu 1. linearneho zoznamu: \n");
54
55      // Vypisanie obsahu prveho zoznamu
56      vypis(L1);
57
58      // Uvolnenei zoznamov
59      zrus(L1);
60      zrus(L2);
61
62      return(0);
63  }
64
65  p_LIST LISTinit()
66  {
67      p_LIST l = (p_LIST) malloc(sizeof(*l));
68      l->head = NULL;
69      l->tail = NULL;
70      return l;
71  }
72
73  int prazdny(p_LIST l)
74  {
75      /* vrati "true=1" ak je zoznam prazdny */
76      return(l->head == NULL);
77  }
78
79  void pridaj(p_LIST l)
80  {
81      if (l->head == NULL)    //pridanie prveho prvku do prazdneho zoznamu
82      {
83          l->head =(LISTnode *) malloc(sizeof(LISTnode));
84          printf("Meno studenta: ");
85          gets((l->head)->meno);
86          printf("Pocet bodov: ");
87          scanf("%d", &((l->head)->body));
88          (l->head)->next = NULL;
89          (l->head)->prev = NULL;
90          l->tail = l->head;
91      }
92      else     //pridanie dalsich prvkov
93      {
```

```
94          (l->tail)->next = (LISTnode *)malloc(sizeof(LISTnode));
95          printf("Meno studenta:");
96          gets(l->tail->next->meno);
97          printf("Pocet bodov: ");
98          scanf("%d", &(l->tail->next->body));
99          l->tail->next->next = NULL;
100         l->tail->prev = l->tail;
101         l->tail = l->tail->next;
102     }
103     while (getchar() != '\n') // vyprazdnenie buffera
104         ;
105 }
106
107 /* vypisanie obsahu zoznamu */
108 void vypis(p_LIST l)
109 {
110     printf("VYPIS\n");
111
112     p_LISTnode p=l->head;
113
114     while (p != NULL)
115     {
116         printf("\nmeno: %s", p->meno);
117         printf("\nbody: %d \n",p->body);
118         p = p->next;      //posun na dalsi prvok
119     }
120     getchar();
121 }
122
123 /* zrusenie zoznamu */
124 void zrus(p_LIST l)
125 {
126     p_LISTnode p;           //pomocny pointer
127
128     while  (l->head != NULL)
129     {
130         p = l->head;
131         l->head = l->head->next;
132         free((void*)p);
133     }
134     l->tail = l->head;    // ukazovatele h, t nastavime na NULL
135
136     printf("\nZoznam je zruseny!");
137     getchar();
138 }
```

1.4 Príklady pre samostatné precvičovanie vedomostí

Pri všetkých úlohách sa snažte implementovať čo najviac operácií, ktoré je možné s dátovými štruktúrami realizovať.

1. Vytvorte program, ktorý bude implementovať zásobník pomocou poľa, ako aj zreťazeného zoznamu. Zamyslite sa či je vhodnejšie použiť jednosmerne zreťazený zoznam alebo obojsmerne zreťazený zoznam.
2. Vytvorte program, ktorý bude implementovať front pomocou poľa, ako aj zreťazeného zoznamu. Zamyslite sa, či je vhodnejšie použiť jednosmerne zreťazený zoznam, obojsmerne zreťazený zoznam, resp. čo prinesie implementácia pomocou cyklického zreťazeného zoznamu.
3. Vytvorte program, ktorý bude implementovať jednosmerne zreťazený zoznam. Okrem základných operácií, implementujte operáciu vkladania prvku pred a za konkrétny prvok, ako aj mazanie konkrétneho prvku. Jednu implementáciu realizujte s udržiavaním *tail* a jednu bez. Určte časové zložitosti implementovaných operácií a porovnajte ich s implementáciou operácií pri obojstranne zreťazenom zozname.

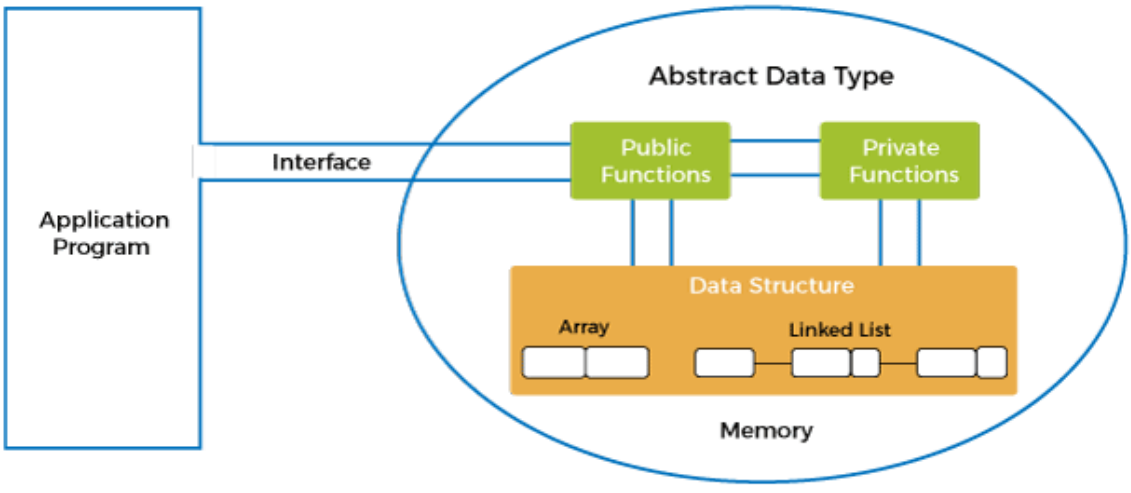
2 Abstraktné dátové typy

Abstraktné dátové typy (*ADT - Abstract Data Types*) sú matematickým a programátorským konceptom, ktorý umožňuje zapuzdrovať dáta a operácie nad nimi do jednotného celku, tzv. zapuzdrenie = *encapsulation*, bez zverejňovania ich konkrétnej implementácie za účelom ukladania objektov (dát), tzv. abstrakcia =*abstraction*. ADT poskytuje rozhranie, ktoré definuje správanie a operácie, ktoré môžu byť vykonávané s dátami, ale nezaoberá sa detailmi, aké dátové štruktúry alebo algoritmy sú potrebné na ich realizáciu. Týmto spôsobom ADT oddeľuje rozhranie od implementácie a umožňuje rôznym implementáciám poskytnúť rovnaké rozhranie, t. j. konkrétny abstraktný dátový typ môže byť implementovaný pomocou rôznych dátových štruktúr. Dátová štruktúra je teda konkrétna implementácia abstraktného dátového typu.

ADT definuje dva hlavné prvky:

- Dátové typy: určujú údaje s ktorými ADT pracuje.
- Operácie: určujú, aké akcie môžu byť vykonávané s dátami ADT. To zahŕňa operácie ako vkladanie, odstránenie, vyhľadávanie, získavanie dát, zmena dát, atď.

Nižšie uvedený obrázok znázorňuje model ADT. V modeli ADT existujú dva typy modelov, t. j. verejné funkcie a súkromné funkcie. Model ADT obsahuje aj dátové štruktúry, ktoré používame. V tomto modeli sa vykoná ako prvé zapuzdrenie, t. j. všetky dáta sú zabalené do jednej jednotky. Potom sa vykoná abstrakcia, čo znamená, že ukážeme operácie, ktoré je možné vykonať s dátovou štruktúrou.



Obr. 16 Vizuálna reprezentácia abstraktného dátového typu (javaTpoint, 2022).

Skúsme pochopiť ADT na príklade z reálneho sveta. Ak uvažujeme o smartfóne, tak údaje ako:

- 4 GB RAM
- Snapdragon 2.2ghz processor
- 5 inch LCD screen
- Dual camera
- Android 8.0

môžeme chápať ako dáta s tým, že na smartfóne môžeme vykonávať operácie ako:

- telefonovanie – *call()*
- posielanie správ – *message()*
- fotografovanie – *photo()*
- tvorba videa – *video()*
- a pod.

Používanie abstraktných dátových typov (ADT) prináša niekoľko výhod, ktoré robia programovanie efektívnejším, flexibilnejším a udržateľnejším:

- **Abstrakcia (*abstraction*)**: ADT umožňuje abstrahovať od implementácie. To znamená, že programátor nemusí poznať a zaujímať sa o detaily implementácie dátových štruktúr. To zjednodušuje prácu s komplexnými štruktúrami a znižuje riziko chýb spojených s priamym prístupom k dátam.
- **Zapuzdrenie (*encapsulation*)**: ADT poskytujú spôsob na zapuzdrenie dát a operácií do jedného celku, čo uľahčuje správu a úpravu dátovej štruktúry.
- **Modulárnosť (*modularity*)**: ADT je možné kombinovať s inými ADT na vytvorenie zložitejších dátových štruktúr, čo môže zvýšiť flexibilitu a modularitu v programovaní.
- **Flexibilita (*flexibility*)**: Vďaka oddeleniu rozhrania a implementácie môže byť ADT implementovaný rôznymi spôsobmi. Týmto spôsobom môže byť implementácia ADT zmenená bez ovplyvnenia zvyšku kódu, ktorý ju používa. To uľahčuje údržbu a znižuje náklady na rozvoj softvéru. Programátor môže vybrať optimálnu implementáciu podľa konkrétnych požiadaviek a výkonnostných kritérií. To umožňuje lepšie prispôsobenie programu konkrétnym potrebám. Táto vlastnosť je často označovaná aj ako nezávislosť dátovej štruktúry (*data structure independence*).

- **Bezpečnosť a ochrana**: ADT môže definovať operácie, ktoré sú dostupné pre manipuláciu s dátami. Ak implementácia ADT zaručuje, že iba tieto operácie sú povolené, môže to zlepšiť bezpečnosť programu a znížiť riziko nebezpečných chýb. Táto vlastnosť je často označovaná aj ako skrytie informácií (*information hiding*).
- **Znovupoužiteľnosť (*code reusability*)**: Keď sú dáta a operácie zapuzdrené a reprezentované ako ADT, môžu byť jednoducho použité v rôznych častiach programu a aj v iných projektoch. To zvyšuje znovupoužiteľnosť kódu a zrýchľuje vývoj nových aplikácií.

Všetky tieto výhody vedú k lepšej organizácii kódu, zlepšujú jeho udržateľnosť (*maintainability*) a znižujú riziko chýb pri vývoji softvéru. Vo všeobecnosti môžeme konštatovať, že výhody často prevyšujú nevýhody používania ADT. Medzi nevýhody môžeme považovať, že implementácia ADT môže zvýšiť réžiu z hľadiska pamäte a spracovania, čo môže ovplyvniť výkon. Používanie ADT si vyžaduje znalosť ich implementácie a používania, čo si môže vyžadovať čas a námahu na učenie.

Implementácia dátovej štruktúry, ktorá predstavuje abstraktný dátový typ môže byť realizovaná rôznymi spôsobmi. Napríklad zásobník môže byť implementovaný pomocou jednoduchého statického poľa alebo dynamického poľa, lineárneho zreťazeného zoznamu, obojstranne zreťazeného zoznamu a pod. Používateľ dátovej štruktúry pracuje len s definovanými operáciami a nemusí sa starať o to, ako sú dáta uložené a aká je konkrétna implementácia operácií. Toto oddelenie medzi rozhraním a implementáciou umožňuje zmenu implementácie dátovej štruktúry bez ovplyvnenia existujúceho kódu, čo zvyšuje modularitu a znovupoužiteľnosť kódu.

Mnohé programovacie jazyky obsahujú knižnice s preddefinovanými (vstavanými) dátovými štruktúrami (často označované aj ako kontajnery/kolekcie), ktoré môže programátor použiť na ukladanie údajov. Knižnica STL (*Standard Template Library*) rozširuje možnosti štandardnej knižnice jazyka C++. STL poskytuje sadu generických algoritmov, kontajnerov a funkcií na prácu s dátami. STL poskytuje rôzne typy kontajnerov, ktoré uľahčujú ukladanie a správu dát. Medzi najbežnejšie kontajnery patria dynamické pole (*vector* alebo *deque* – je *vector* doplnený o možnosť vkladania/mazania zo začiatku poľa) a zoznam (*list* – ktorý je implementovaný pomocou obojstranne zreťazeného zoznamu). Tieto kontajnery sú označované ako sekvenčné. Vektor je vlastne dynamické pole, ktoré môže obsahovať prvky ľubovoľného typu. Rozmery poľa sa môžu meniť podľa potreby, t. j. aj po počiatočnej alokácii. Prístup k prvkom vektora je rovnaký ako k prvkom štandardného poľa. Vyznačuje sa rýchlym prístupom k prvkom, avšak vkladanie a vymazávanie konkrétnych prvkov je pomalé. *Deque*, tiež nazývaná ako obojstranný front sa podobá vektoru. Rozdiel je najmä vo vkladaní

prvkov. Do vektora sa nové prvky pridávajú len na koniec, a ak by sme chceli vložiť prvok inde, musíme celý zvyšok poľa posúvať. Do obojstrannej fronty môžeme prvky pridávať aj na začiatok, bez toho, aby sa celé pole posúvalo. Ďalšie rozdiely sú v spôsobe alokácie a dealokácie pamäte. Jeho použitie volíme v prípade ak budeme veľakrát vkladať (alebo vyberať) prvky zo začiatku alebo konca. Jeho výhodou je rýchle vkladanie na začiatok, ako aj rýchly náhodný prístup k prvkom. Nevýhodou je pomalé vkladanie a mazanie vo vnútri a časovo náročnejšia dealokácia ako pri vektore. Pri zozname (*list*) ide o dátovú štruktúru, kde každý prvok okrem hodnoty obsahuje aj smerník na nasledujúci a predchádzajúci prvok. Prvky sú navzájom previazané, a ku konkrétnemu prvku sa môžeme dostať len postupne od začiatku alebo od konca. Nie je možný priamy náhodný prístup k prvkom. Zvyčajne ho používame, ak nepotrebujeme k prvkom pristupovať náhodne, ale skôr je dôležité ich poradie. T. j. v prípade, ak vždy spracovávame všetky prvky a potrebujeme rýchlo vkladať alebo mazať prvky.

Medzi kontajnerové adaptéry patria: zásobník (*stack*) a front (*queue*). Front je dátová štruktúra typu FIFO, čo znamená, že pridávať sa dá len na koniec a odoberať len zo začiatku. Nie je možný priamy náhodný prístup k prvkom. Jeho použitie volíme v prípade, keď potrebujeme postupne spracovávať dáta, ktoré „čakajú“ na spracovanie v rade. Výhodou je, že ide o jednoduchú dátovú štruktúru, a ako nevýhodu je možné chápať jeho obmedzenú funkčnosť. Zásobník je štruktúra typu LIFO, čo znamená že pridávať prvky sa dá len na koniec a odoberať takisto len z konca. Nie je možný priamy náhodný prístup k prvkom. Jeho použitie volímeT ak potrebujeme odkladať dáta, zatiaľ čo robíme niečo iné, a po ukončení tejto činnosti ich chceme spracovávať, pričom môžeme začať aj posledným uloženým. Výhodou je, že ide o jednoduchú dátovú štruktúru, a ako nevýhodu je možné chápať jeho obmedzenú funkčnosť.

Ako asociatívne kontajnery sú označované multimnožina (vrece, bag, *multiset*), množina (*set*), tabuľka (*table*), mapa (*map*) a ďalšie. Množina je usporiadaná dátová štruktúra, ktorá vylučuje duplicitu prvkov. To znamená, že hodnoty sú usporiadané podľa nejakého pravidla, ktoré je možné definovať a žiadna hodnota sa v štruktúre nevyskytuje dvakrát. Podobne ako zásobník alebo fronta, ani množina neumožňuje priamy náhodný prístup k prvkom. Jej použitie volíme v prípade ak potrebujeme usporiadané hodnoty s vylúčením duplicity. Multimnožina je množina, ktorá umožňuje duplicitu. Takisto ako množina má svoje prvky usporiadané podľa nejakého pravidla. Mapa je ďalší utriedený dátový typ, ktorý však namiesto samotných prvkov ukladá dvojice prvkov, tzv. kľúč a samotnú hodnotu prvku. Dvojice v mape sú utriedené podľa kľúča, triediace kritérium sa dá definovať. Pri vkladaní sa nový prvok automaticky zaradí na správne miesto. Mapa nepovoľuje

duplicitu kľúčov, hodnoty sa môžu opakovať. Používa sa kedykoľvek potrebujeme uchovať dvojice prvkov zoradené podľa nejakého kritéria. Jej výhodou je automatické zoradenie hodnôt podľa kľúča, uchovávanie dvojíc, zabraňovanie nejednoznačnosti kľúčov. Nevýhodou je nemožnosť náhodného prístupu.

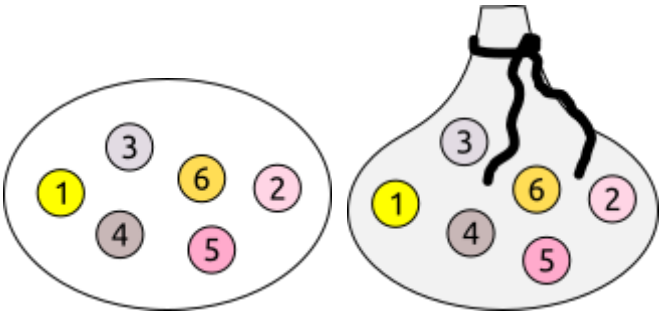
Všetky tieto kontajnery majú rôzne časové a priestorové zložitosti operácií, ako aj usporiadanie údajov realizované v rámci dátovej štruktúry. Schopnosť naprogramovať si vlastný ADT môže byť užitočná z dôvodu, že niekedy potrebujeme konkrétne špecifické správanie, respektíve sa nechceme spoliehať na kód tretej strany. Základom je však rozumieť jednotlivým ADT, aby pri ich používaní ste s nimi narábali efektívne a vedeli, ktorý kontajner kedy zvoliť pre riešenie daného problému.

Ako sme uviedli vyššie, ADT poskytuje rozhranie, ktoré definuje správanie a operácie, ktoré môžu byť vykonávané s dátami. Medzi základné operácie radíme:

- vytvorenie prázdneho kontajnera, prípadne inicializácia premenných potrebných k implementácii,
- vloženie prvku do kontajneru,
- odobratie prvku z kontajneru,
- zistenie či je kontajner prázdny,
- zistenie či je kontajner plný,
- reálny počet prvkov v kontajneri,
- prehľadávanie kontajnera, zobrazenie obsahu kontajnera.

2.1 Množina = *set*

Množina je dátovou štruktúrou, ktorá negarantuje poradie prvkov (*unordered*), ktoré obsahuje. To poskytuje istú voľnosť vývojárovi pri jeho implementácii. Prvok s rovnakou hodnotou sa v množine môže vyskytovať len raz, na rozdiel od multimnožiny, kde sa prvky s rovnakou hodnotou môžu vyskytovať viackrát.



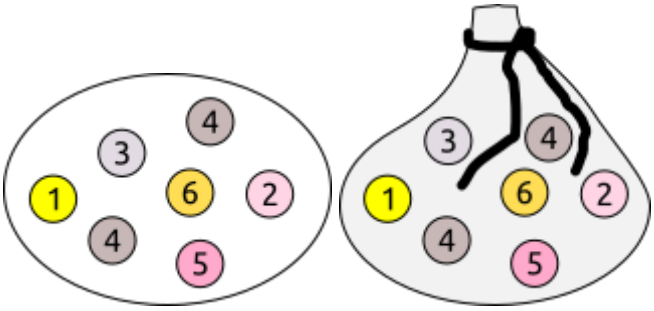
Obr. 17 Vizuálna reprezentácia množiny.

Množinu je možné implementovať rôznymi spôsobmi:

- Pomocou poľa (*array*). Keďže množina negarantuje poradie prvkov, je možné problematickú operáciu mazania prvku realizovať s konštantnou časovou zložitosťou O(1), a to tak, že sa mazaný prvok nahradí posledným prvkom uloženým v množine. V opačnom prípade by sme museli realizovať posuv prvkov, čo vedie k lineárnej časovej zložitosti. Stále sme však obmedzení veľkosťou poľa, ktorú v prípade použitia dynamického poľa vieme riešiť realokáciou. To však vyžaduje permanentnú kontrolu tejto skutočnosti pri každom vložení prvku (t. j. či je dostatok miesta v množine), okrem kontrolovania, či sa prvok s danou hodnotou už v množine nenachádza.
- Pomocou zoznamu (*list*), bez zohľadňovania poradia v ktorom prvky do množiny, vkladáme a len kontrolujeme, či sa náhodou prvok s danou hodnotou už v množine nenachádza. Ide o najjednoduchšiu implementáciu, ktorej neefektivita spočíva v tom, že v prípade hľadania konkrétneho prvku môže byť nutné prejsť celú množinu, čo spôsobí lineárnu asymptotickú časovú zložitosť O(n) obdobne, ako je tomu aj pri implementácii pomocou poľa.
- Pomocou hašovacej tabuľky (*hash table*). Ide o efektívnu implementáciu množiny, ktorá umožňuje prístup k položkám s konštantnou časovou zložitosťou. Táto problematika je však nad rámec obsahu tejto učebnice.

2.2 Multimnožina = *multiset*

Multimnožina, alebo často nazývaná aj ako množina s opakovaním, je dátová štruktúra vychádzajúca z množiny, ktorá taktiež negarantuje poradie prvkov, ale prvky s rovnakou hodnotou sa môžu opakovať.



Obr. 18 Vizuálna reprezentácia multimnožiny.

Multimnožinu je možné implementovať rôznymi spôsobmi, podobne ako množinu:

- Pomocou poľa (*array*), avšak odpadá kontrola pri vkladaní prvku, či sa prvok s danou hodnotou už v multimnožine nachádza, keďže tu je opakovanie prvkov s tou istou hodnotou dovolené.
- Pomocou zoznamu (*listu*), bez zohľadňovania poradia. Ide o najjednoduchšiu implementáciu, ktorej neefektivita spočíva v množstve použitej pamäte v prípade, ak sa prvky s rovnakou hodnotou často opakujú.
- Pomocou dvojice zoznamov, kde prvý obsahuje hodnoty a druhý zaznamenáva počty ich výskytov. Tento postup je efektívnejší, ale stále vyžaduje pomerne náročnú operáciu prechodu zoznamom pre prístup k jednotlivým položkám (lineárna asymptotická časová zložitosť O(n)).
- Pomocou hašovacej tabuľky. Ide o efektívnu implementáciu multimnožiny, ktorá umožňuje prístup k položkám s očakávanou konštantnou časovou zložitosťou. Táto problematika je však nad rámec obsahu tejto učebnice.

2.3 Lineárne abstraktné dátové typy

Slovo lineárny v kontexte dátových štruktúr hovorí o tom, že akýkoľvek prvok dátovej štruktúry má vždy maximálne jedného predchodcu, ako aj nasledovníka. Podľa spôsobu vkladania, resp. vyberania položiek z dátovej štruktúry rozoznávame tri typy lineárnych dátových štruktúr. Ide o zoznam, zásobník a frontu.

2.3.1 Zoznam = *list*

Zoznam je abstraktný dátový typ, ktorý opisuje lineárnu kolekciu dátových položiek v určitom poradí tak, že každý prvok zaberá špecifickú pozíciu v zozname. Poradie môže byť abecedné alebo číselné. Takýto zoznam je často označovaný ako usporiadaný (*ordered list*), alebo to môže byť len poradie, v ktorom boli pridané prvky do zoznamu. Rozlišujeme aj neusporiadaný zoznam (*unordered list*), kde prvky nemusia byť usporiadané podľa nejakého pravidla, respektíve toto pravidlo si môže určiť vývojár. V literatúre sa môžete stretnúť aj s označením *indexed list*, ktorý zabezpečuje prístup k prvkom použitím indexu. Na rozdiel od množiny, prvky zoznamu nemusia byť jedinečné. Zoznam je abstraktný dátový typ, ktorý sa najčastejšie implementuje pomocou dynamického poľa alebo zreťazeného zoznamu. Avšak poznáme aj statickú implementácia zoznamu. Medzi základné operácie patrí vytvorenie (*create*), pridanie prvku (*add*), odstránenie prvku (*delete*) a prechod zoznamom (*traverse*).

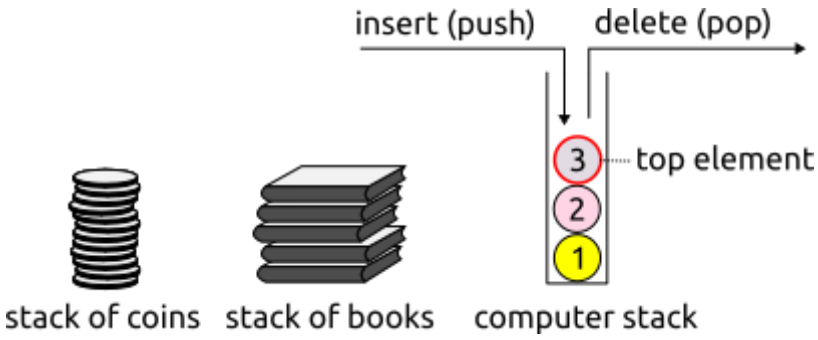
Časové zložitosti (*time complexity*) operácií nad dátovou štruktúrou zoznam (*list*) implementovanej pomocou poľa:

- Prístup k prvkom (*access*) – O(1) = konštantná časová zložitosť. Prvky poľa sú prístupne cez svoj index.
- Vyhľadávanie v dátovej štruktúre (*search*) – O(n) = lineárna časová zložitosť. Hľadanie konkrétneho výskytu prvku v poli v najhoršom prípade vyžaduje prehľadať celú dátovú štruktúru, z čoho vyplýva lineárna časová zložitosť, kde *n* predstavuje veľkosť poľa.
- Vloženie/vymazanie prvku z poľa (*insert/delete*) – O(n) = lineárna časová zložitosť. Operácii vloženia či vymazania prvku v prípade budovania usporiadaného poľa podľa určitého kritéria môže predchádzať operácia vyhľadávania. V určitých prípadoch môže nastať posuv prvkov pre vytvorenie požadovaného miesta pre vloženie prvku, respektíve odstránenie prvku, čo spôsobuje lineárnu časovú zložitosť. V prípade, že nám nezáleží na usporiadaní prvkov v poli, je možné, že operácia vloženia/vymazania bude mať konštantnú časovú zložitosť.

2.3.2 Zásobník = *stack*

Zásobník je homogénna dátová štruktúra. Pri práci so zásobníkom sú k dispozícii operácie ako "*push*" (vloženie prvku na vrchol zásobníka), "*pop*" (odstránenie prvku z vrcholu zásobníka), "*top*" (získanie hodnoty vrcholu zásobníka), "*isEmpty*" (overenie, či je zásobník prázdny), "*isFull*" (overenie, či je zásobník plný), a pod. Je zrejmé, že ide o dátovú štruktúru, ktorá umožňuje prístup iba k prvku, ktorý sa nachádza na vrchole zásobníka. Obdobne aj vkladanie prvkov je možné len na vrchol zásobníka.

Vizuálna reprezentácia zásobníka je znázornená na obrázku 19. Ako príklad reálnej predstavy sa často ilustruje zásobník ako kopa mincí, kníh a pod.



Obr. 19 Vizuálna reprezentácia zásobníka.

Zásobník je možné implementovať pomocou poľa alebo zreťazeného zoznamu. Pri použití zreťazeného zoznamu dochádza k eliminácii problémových operácií pri mazaní prvkov, keďže v zreťazenom zozname každý uzol vzniká dynamicky, ako samostatný prvok.

Časové zložitosti (*time complexity*) operácií nad dátovou štruktúrou:

- Prístup k prvkom (*access*) – O(n) = lineárna časová zložitosť. Prístup k prvkom v zásobníku v najhoršom prípade vyžaduje výber (odstránenie) jedného prvku po druhom, kde *n* predstavuje počet prvkov v zásobníku.
- Vyhľadávanie v dátovej štruktúre (*search*) – O(n) = lineárna časová zložitosť. Hľadanie konkrétneho výskytu prvku v zásobníku v najhoršom prípade vyžaduje výber (odstránenie) jedného prvku po druhom a jeho porovnanie s cieľom, kde *n* predstavuje počet prvkov v zásobníku.
- Vloženie/vymazanie prvku (*insert/delete*) – O(1) = konštantná časová zložitosť. Operácia vloženia/vymazania prvku v zásobníku sa dá vždy spraviť s konštantnou časovou zložitosťou.

Typické použitie:

- Zásobník je využívaný implicitne vo všetkých rekurzívnych algoritmoch. Keď je funkcia volaná, jej stav (vrátane parametrov a iných premenných) sa ukladá na zásobník. Po ukončení funkcie sa stav obnovuje zo zásobníka, čo umožňuje správne vykonávanie rekurzívnych volaní.
- Používa sa pre ukladanie dát v priebehu výpočtu. Pri vyhodnocovaní matematických výrazov sa môže použiť zásobník na ukladanie operátorov a operandov. Tento postup je známy ako "zásobníkové vyhodnocovanie" (*stack-based evaluation*) a zabezpečuje správne poradie vykonávania operácií.
- Zásobník môže byť využitý v kontexte správy pamäte. V niektorých programovacích jazykoch sa zásobník používa na správu lokálnych premenných a volanie funkcií. Každý blok kódu (napríklad funkcia alebo pri objektovo-orientovanom programovaní metóda) má svoj vlastný rámec na zásobníku, v ktorom sa uchovávajú jeho lokálne premenné a návratová adresa.
- Pri vykonávaní algoritmov. V rôznych algoritmických problémoch sa zásobník používa na ukladanie stavov a sledovanie prehľadávania priestoru riešení. Typickým príkladom je prehľadávanie dátových štruktúr (strom, graf) do hĺbky (*Depth-First Search*), algoritmus prehľadávania v grafikách (*Scanline algorithm*) a mnoho ďalších.
- *Undo/Redo* operácie: Pri implementácii funkcionality "*Undo*" a "*Redo*" v textových editoroch alebo grafických programoch sa často používa zásobník. Každá vykonaná akcia (napríklad pridanie, odstránenie alebo zmena) sa ukladá na zásobník, čo umožňuje vrátiť sa späť (*undo*) alebo opätovne vykonať (*redo*) predchádzajúce operácie.
- Zásobník sa často používa na kontrolu vyváženosti symbolov, ako sú zátvorky, zložené zátvorky a iné v texte. Pri spracovaní sa symboly vkladajú a odoberajú zo zásobníka a kontroluje sa ich správne vyváženie a platnosť.
- Náboje v pištoli fungujú na princípe zásobníku.

2.3.2.1 Príklad – implementácia zásobníka pomocou poľa

Zadanie problému:

Vytvorte program, ktorý bude implementovať zásobník pomocou poľa ako abstraktný dátový typ. Kontajner bude slúžiť na uchovávanie informácií, ktoré pre jednoduchosť ilustrujeme ukladaním znakov. Tento bude umožňovať vytvorenie zásobníka, pridanie prvku, odstránenie prvku, zistenie počtu prvkov v zásobníku, prístup k vrcholu zásobníka, zobrazenie obsahu zásobníka a odstránenie/uvoľnenie celého kontajnera. Našou snahou je zabezpečiť aj priamy náhodný prístup k prvkom.

Objasnenie príkladu autorom:

Na riadkoch 4 a 5 deklarujeme dva vlastné dátové typy. *item_type* reprezentuje jednoduchý dátový typ *char*, ako ilustráciu, že typom prvku v kontajneri môže byť čokoľvek iné. *result_type*, ktorý reprezentuje jednoduchý dátový typ *int*, používame pre označenie návratovej hodnoty funkcií, ktorého význam má byť, či funkcia skončila v poriadku, alebo nie. Konštanty RESULT_OK a RESULT_ERROR definované na riadkoch 7 a 8 využívame pre návrat príznaku, či funkcia skončila v poriadku, alebo nie.

```
4  typedef char item_type;
5  typedef int result_type;
6
7  #define RESULT_OK 0
8  #define RESULT_ERROR 1
```

Pre implementáciu kontajnera typu zásobník, ako prvé definujeme dátovú štruktúru *simple_array_type*, pomocou ktorej budeme zásobník implementovať. O poli evidujeme položku *items*, ktorá bude reprezentovať samotné prvky kontajnera, *capacity*, ktorá predstavuje veľkosť dátovej štruktúry a *count*, ktorá udržiava reálny počet prvkov v poli. Pre toto sme definovali vlastný dátový typ *simple_array_type* na riadkoch 11 – 16.

```
10 // simple array
11 typedef struct simple_array_type
12 {
13     item_type* items;
14     int capacity;
15     int count;
16 } simple_array_type;
```

Funkciu, ktorá sa postará o vznik dátovej štruktúry pole sme pomenovali *simple_array_create()*. Tá preberá jeden parameter, ktorý reprezentuje veľkosť dátovej štruktúry. Ako prvé alokujeme pamäť pre samotné pole (riadok 20). Ak sa pamäť nepodarí alokovať, funkciu ukončíme a vrátime hodnotu *NULL*. V prípade úspechu pokračujeme alokáciu pamäte pre samotné prvky poľa podľa parametra

capacity a dátového typu položiek (v našom prípade *char*) (riadok 24). Položku poľa *capacity* inicializujeme hodnotou tohto parametra (riadok 28) a *count* nastavujeme na hodnotu 0, keďže pole je zatiaľ prázdne (riadok 29). Návratovou hodnotou funkcie je samotné pole.

```
18  simple_array_type* simple_array_create(int capacity)
19  {
20      simple_array_type* array = malloc(sizeof(simple_array_type));
21      if (!array)
22          return NULL;
23
24      array->items = malloc(sizeof(char)*capacity);
25      if (!array->items)
26          return NULL;
27
28      array->capacity = capacity;
29      array->count = 0;
30
31      return array;
32  }
```

Funkciu pre uvoľnenie alokovanej pamäte sme nazvali *simple_array_destroy()*. Preberá jeden parameter, ktorým je dátová štruktúra pole. V tele funkcie sa najprv postaráme o uvoľnenie pamäte reprezentujúcej pole prvkov dátovej štruktúry (riadok 36) a následne uvoľníme pamäť reprezentujúcu samotnú dátovú štruktúru pole (riadok 37).

```
34  void simple_array_destroy(simple_array_type *array)
35  {
36      free(array->items);
37      free(array);
38  }
```

Funkcia, ktorá vráti počet prvkov uložených v poli sa nazýva *simple_array_count()*. Logicky preberá jeden parameter, ktorým je pole. Počet prvkov poľa reprezentuje premenná *count*, ktorej hodnota je z funkcie vrátená cez návratovú hodnotu.

```
40  result_type simple_array_count(simple_array_type *array)
41  {
42      return array->count;
43  }
```

Pre pridanie prvku do dátovej štruktúry sme definovali funkciu *simple_array_add_item()*, ktorá ako parameter preberá pole a hodnotu prvku, ktorý chceme do neho vložiť. Ako prvé zisťujeme či je v poli ešte miesto na základe porovnania obsahu premenných *count* a *capacity* (riadok 47). Ak tomu tak nie je, funkciu ukončíme s chybou (riadok 49). V opačnom prípade na pozíciu určenou premennou *count*, ktorá reprezentuje index prázdneho miesta v poli, vložíme prvok (riadok 52). Premennú *count* inkrementujeme o 1 (riadok 53) a funkciu ukončujeme s príznakom, že všetko prebehlo úspešne (riadok 55).

```
45  result_type simple_array_add_item(simple_array_type *array, item_type item)
46  {
47      if (array->count == array->capacity)
48      {
49          return RESULT_ERROR;
50      }
51
52      array->items[array->count] = item;
53      array->count++;
54
55      return RESULT_OK;
56  }
```

Pre prístup k prvku na konkrétnom indexe definujeme funkciu *simple_array_get_item_at_index()*. Táto preberá tri parametre: pole, index prvku, na ktorom sa nachádza prvok, ktorý chceme získať a parameter *item*, ktorý v sebe nesie hodnotu prvku. Pre tento parameter by bolo možné využiť návratovú hodnotu funkcie, avšak dodržujeme jednotný koncept návrhu funkcií, t. j. cez návratovú hodnotu funkcie vraciame príznak, či funkcia skončila v poriadku, alebo nie. Preto je nutné tento parameter odovzdať odkazom, aby sme s hodnotou prvku mohli v main-e pracovať. V tele funkcie ako prvé testujeme či index prvku, ktorý chceme získať, je v rozsahu nášho zaplneného poľa (riadok 60). Ak tomu tak nie je, vraciame príznak neúspechu funkcie pomocou konštanty *RESULT_ERROR* (riadok 62). V opačnom prípade do parametra *item* uložíme hodnotu prvku na indexe danom parametrom *index* (riadok 65) a funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 67).

```
58  result_type simple_array_get_item_at_index(simple_array_type *array, int index, item_type *item)
59  {
60      if (index < 0 || index >= array->count)
61      {
62          return RESULT_ERROR;
63      }
64
65      *item = array->items[index];
66
67      return RESULT_OK;
68  }
```

V prípade potreby editácie hodnoty prvku v poli sme definovali funkciu *simple_array_set_item_at_index()*, ktorá preberá tri parametre. Parameter, ktorý reprezentuje pole, index prvku, ktorý chceme editovať a novú hodnotu tohto prvku. Ak je index prvku väčší, nanajvýš rovný hodnote položky *count*, ktorá reprezentuje najbližší voľný index, na ktorý môžeme vložiť nový prvok, tak funkciu ukončujeme s neúspechom (riadky 72 –75). Inak sa postaráme o prepísanie hodnoty prvku ležiacom na indexe určenom parametrom *index* novou hodnotou, ktorá je uložená

v parametri *item* (riadok 77) a funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 79).

```
70  result_type simple_array_set_item_at_index(simple_array_type *array, int index, item_type item)
71  {
72      if (index >= array->count)
73      {
74          return RESULT_ERROR;
75      }
76
77      array->items[index] = item;
78
79      return RESULT_OK;
80  }
```

V prípade potreby odstránenia hodnoty prvku v poli sme definovali funkciu *simple_array_remove_item_at_index()*, ktorá preberá tri parametre. Parameter, ktorý reprezentuje pole, index prvku, ktorý chceme odstrániť a parameter *item*, ktorý v sebe nesie hodnotu odstraňovaného prvku. Ak je index odstraňovaného prvku mimo rozsah zaplneného poľa (riadok 84), funkciu ukončujeme s neúspechom (riadok 86). Inak sa postaráme o uloženie hodnoty prvku do parametra *item* (riadok 89). Pomocou cyklu s presne definovaným počtom opakovaní sa postaráme o posuv prvkov ležiacich za odstraňovaným prvkom smerom doľava od pozície odstraňovaného prvku (riadky 91 – 94). Tým simulujeme mazanie v poli, keďže pamäť pre pole je alokovaná ako jeden súvislý blok. Je dôležité nezabudnúť dekrementovať hodnotu premennej *count* o 1 (riadok 95), ktorá signalizuje voľné miesto v poli. Funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 97).

```
82  result_type simple_array_remove_item_at_index(simple_array_type *array, int index, item_type
*item)
83  {
84      if (index < 0 || index >= array->count)
85      {
86          return RESULT_ERROR;
87      }
88
89      *item = array->items[index];
90
91      for (int i=index; i<array->count-1; i++)
92      {
93          array->items[i] = array->items[i+1];
94      }
95      array->count--;
96
97      return RESULT_OK;
98  }
```

Pre vypísanie obsahu dátovej štruktúry sme definovali funkciu *simple_array_print()*, ktorá preberá jeden parameter, ktorým je pole. V tele funkcie sa postaráme o výpis obsahu premenných *count*

a *capacity*, ktoré reprezentujú reálny počet prvkov v poli a samotnú veľkosť poľa (riadok 102). Pomocou cyklu s presne definovaným počtom opakovaní, ktorý beží podľa počtu prvkov uložených v poli, sa postaráme o vypísanie hodnôt prvkov (riadky 103 a 104).

```
100  void simple_array_print(simple_array_type *array)
101  {
102      printf("count: %d, capacity: %d, items: [", array->count, array->capacity);
103      for (int i=0; i<array->count; i++)
104          printf("%c", array->items[i]);
105      printf("]");
106  }
```

Samotný dátový typ reprezentujúci kontajner typu zásobník definujeme na riadkoch 109 – 112, pomocou štruktúry, ktorá obsahuje práve jednu položku a tou je nami definované pole typu *simple_array_type*.

```
108  // stack using simple array
109  typedef struct stack_type
110  {
111      simple_array_type* array;
112  } stack_type;
```

Funkciu, ktorá sa postará o vznik dátovej štruktúry zásobník sme pomenovali *stack_create()*. Tá preberá jeden parameter, ktorý reprezentuje veľkosť dátovej štruktúry. Ako prvé alokujeme pamäť pre samotnú dátovú štruktúru (riadok 116). Ak sa pamäť nepodarí alokovať, funkciu ukončíme a vrátime hodnotu *NULL*. V prípade úspechu pokračujeme alokáciou pamäte pre samotné prvky poľa podľa parametra *capacity* pomocou definovanej funkcie *simple_array_create()* (riadok 120). Ak sa pamäť nepodarí alokovať, funkciu ukončíme a vrátime hodnotu *NULL*. V opačnom prípade je návratovou hodnotou funkcie adresa dátovej štruktúry, ktorá reprezentuje zásobník (riadok 124).

```
114  stack_type* stack_create(int capacity)
115  {
116      stack_type* stack = malloc(sizeof(stack_type));
117      if (!stack)
118          return NULL;
119
120      stack->array = simple_array_create(capacity);
121      if (!stack->array)
122          return NULL;
123
124      return stack;
125  }
```

Funkciu pre uvoľnenie alokovanej pamäte sme nazvali *stack_destroy()*. Preberá jeden parameter, ktorým je dátová štruktúra zásobník. V tele funkcie sa najprv postaráme o uvoľnenie pamäte reprezentujúcej pole prvkov dátovej štruktúry pomocou definovanej funkcie *simple_array_destroy()*

(riadok 129) a následne uvoľníme pamäť reprezentujúcu samotnú dátovú štruktúru zásobníka (riadok 130).

```
127  void stack_destroy(stack_type* stack)
128  {
129      simple_array_destroy(stack->array);
130      free(stack);
131  }
```

Funkcia, ktorá vráti počet prvkov uložených v zásobníku sa nazýva *stack_count()*. Logicky preberá jeden parameter, ktorým je zásobník. Počet prvkov zásobníka zistíme zavolaním funkcie *simple_array_count()*, ktorej je ako parameter odovzdaná položka zásobníka s názvom *array*. Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_count()*.

```
133  result_type stack_count(stack_type* stack)
134  {
135      return simple_array_count(stack->array);
136  }
```

Pre pridanie prvku do dátovej štruktúry sme definovali funkciu *stack_push()*. Tá ako parameter preberá zásobník a hodnotu prvku, ktorý chceme do neho vložiť. Pre samotné vloženie prvku do zásobníka využívame definovanú funkciu *simple_array_add_item()*, ktorej sú odovzdané dva argumenty. Pole, cez ktoré je zásobník implementovaný a hodnota samotného prvku. Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_push()* (riadok 140).

```
138  result_type stack_push(stack_type* stack, item_type item)
139  {
140      return simple_array_add_item(stack->array, item);
141  }
```

Zásobník je dátová štruktúra, ktorá vyberá/odstraňuje prvky z vrcholu zásobníka, t. j. prvok, ktorý bol do zásobníka vložený ako posledný, bude ako prvý odstránený. Z toho dôvodu vo funkcii *stack_pop()* ako prvé zistíme počet prvkov nachádzajúcich sa v zásobníku pomocou funkcie *simple_array_count()* a túto hodnotu uložíme do lokálnej premennej *count* (riadok 145). Následne prvok zo zásobníka odstránime pomocou funkcie *simple_array_remove_item_at_index()*, pričom samotný index tohto prvku je definovaný premennou *count* zníženou o 1, keďže premenná *count* pri našej implementácii signalizuje voľné miesto v poli prvkov (riadok 146). Parameter funkcie *item* je odovzdaný odkazom, aby sme mohli s hodnotou odstraňovaného prvku v main–e pracovať. Napríklad, v našom programe ju vypisujeme (riadky 216, 225, 241 a 250). Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_pop()*.

```
143  result_type stack_pop(stack_type* stack, item_type *item)
144  {
145      int count = simple_array_count(stack->array);
146      return simple_array_remove_item_at_index(stack->array, count-1, item);
147  }
```

Zásobník ako dátová štruktúra definuje aj funkciu pre prístup k prvku na vrchole zásobníka. V tomto prípade nejde o odstránenie tohto prvku, ale len o jeho sprístupnenie. Pre tento účel sme definovali funkciu *stack_top()*, ktorá preberá dva parametre. Parameter reprezentujúci zásobník a parameter, do ktorého je uložená hodnota tohto prvku. Obidva sú odovzdané odkazom. Ako prvé zistíme počet prvkov nachádzajúcich sa v zásobníku pomocou funkcie *simple_array_count()* a túto hodnotu uložíme do lokálnej premennej *count* (riadok 151). Následne prvok zo zásobníka sprístupnime pomocou funkcie *simple_array_get_item_at_index()*, pričom samotný index tohto prvku je definovaný premennou *count* zníženou o 1, keďže premenná *count* pri našej implementácii signalizuje voľné miesto v poli prvkov (riadok 146). Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_top()*.

```
149  result_type stack_top(stack_type* stack, item_type *item)
150  {
151      int count = simple_array_count(stack->array);
152      return simple_array_get_item_at_index(stack->array, count-1, item);
153  }
```

Pre vypísanie obsahu dátovej štruktúry sme definovali funkciu *stack_print()*, ktorá preberá jeden parameter, ktorým je zásobník. V tele funkcie sa postaráme o výpis obsahu zásobníka pomocou definovanej funkcie *simple_array_print()* (riadok 157). Lokálna premenná *top* je deklarovaná vo funkcii na riadku 160 pre sprístupnenie prvku v zásobníku, ktorý sa nachádza na jeho vrchole. Pre prístup k tomuto prvku sme využili definovanú funkciu *stack_top()* (riadok 160) a podľa návratovej hodnoty tejto funkcie, ktorú ukladáme do lokálnej premennej *res*, reagujeme buď vypísaním hodnoty tohto prvku, ak zásobník nejaké prvky obsahuje, alebo výpisom „*stack is empty*“ ak je zásobník prázdny (riadky 161 –164).

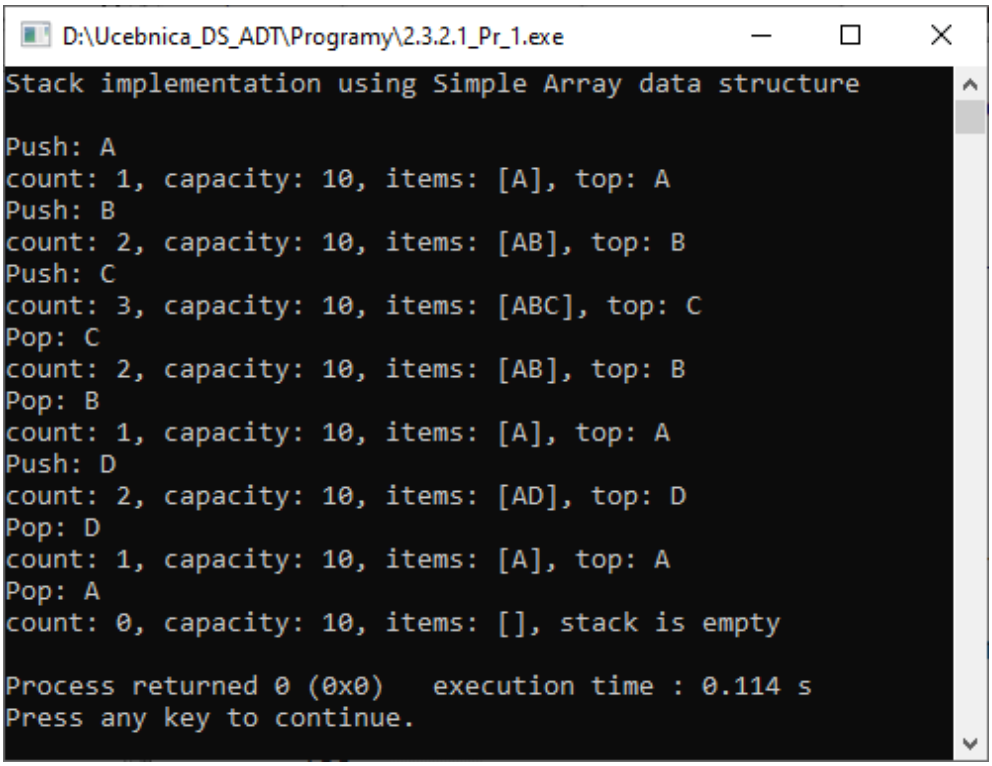
```
155  void stack_print(stack_type *stack)
156  {
157      simple_array_print(stack->array);
158
159      item_type top;
160      result_type res = stack_top(stack, &top);
161      if (res)
162          printf(", stack is empty");
163      else
164          printf(", top: %c", top);
165  }
```

Samotné telo programu je vytvorené minimalisticky pre otestovanie hlavných funkcionalít. Postaráme sa o vytvorenie zásobníka o veľkosti 10 (riadok 173), do ktorého vkladáme prvky a tieto aj vyberáme. Tieto operácie sa striedajú pre lepšie pochopenie, ako dátová štruktúra zásobník funguje. Nezabúdame na testovanie návratových hodnôt funkcií, ktoré môžu signalizovať rôzne stavy

v kontexte manipulácie so zásobníkom. Pre spätnú väzbu pre používateľa zobrazujeme aj meniaci sa obsah zásobníka pomocou volania funkcie *stack_print()*. Na riadku 254 sa staráme o uvoľnenie samotnej dátovej štruktúry.

```
167 // main
168 int main(void)
169 {
170     printf("Stack implementation using Simple Array data structure\n\n");
171 
172     // create
173     stack_type* stack = stack_create(10);
174     if (!stack)
175     {
176         printf("Create stack error\n");
177         return 1;
178     }
179 
180     item_type item;
181     result_type res;
182     int count;
183 
184     // push 'A'
185     item = 'A';
186     printf("Push: %c\n", item);
187     res = stack_push(stack, item);
188     if (res)
189         printf("Push to stack error\n");
190     stack_print(stack);
191     printf("\n");
192 
193     // push 'B'
194     item = 'B';
195     printf("Push: %c\n", item);
196     res = stack_push(stack, item);
197     if (res)
198         printf("Push to stack error\n");
199     stack_print(stack);
200     printf("\n");
201 
202     // push 'C'
203     item = 'C';
204     printf("Push: %c\n", item);
205     res = stack_push(stack, item);
206     if (res)
207         printf("Push to stack error\n");
208     stack_print(stack);
209     printf("\n");
210 
211 
212     // pop item
213     res = stack_pop(stack, &item);
214     if (res)
215         printf("Pop from stack error\n");
```

```
216     printf("Pop: %c\n", item);
217     stack_print(stack);
218     printf("\n");
219 
220     // pop item
221     res = stack_pop(stack, &item);
222     if (res)
223         printf("Pop from stack error\n");
224     printf("Pop: %c\n", item);
225     stack_print(stack);
226     printf("\n");
227 
228     // push 'D'
229     item = 'D';
230     printf("Push: %c\n", item);
231     res = stack_push(stack, item);
232     if (res)
233         printf("Push to stack error\n");
234     stack_print(stack);
235     printf("\n");
236 
237     // pop item
238     res = stack_pop(stack, &item);
239     if (res)
240         printf("Pop from stack error\n");
241     printf("Pop: %c\n", item);
242     stack_print(stack);
243     printf("\n");
244 
245     // pop item
246     res = stack_pop(stack, &item);
247     if (res)
248         printf("Pop from stack error\n");
249     printf("Pop: %c\n", item);
250     stack_print(stack);
251     printf("\n");
252 
253     // destroy
254     stack_destroy(stack);
255 
256     return 0;
257 }
```



Obr. 20 Konzolový výstup programu 2.3.2.1_Pr_1.c.

Jedna z možných implementácií celého programu 2.3.2.1_Pr_1.c.

2.3.2.2 Príklad – implementácia zásobníka pomocou listu

Zadanie problému:

Vytvorte program, ktorý bude implementovať zásobník pomocou zoznamu (*listu*) ako abstraktný dátový typ. Pod dátovou štruktúrou zoznam (*list*) rozumieme dynamické pole, ktorého veľkosť po naplnení sa vždy zdvojnásobuje. Na rozdiel od poľa, *list* je dynamická dátová štruktúra, môžeme ho teda meniť za behu, pridávať, uberať, pýtať sa na jeho veľkosť a pod. Kontajner bude slúžiť na uchovávanie informácií, ktoré pre jednoduchosť ilustrujeme ukladaním znakov. Tento bude umožňovať vytvorenie zásobníka, pridanie prvku, odstránenie prvku, zistenie počtu prvkov v zásobníku, prístup k vrcholu zásobníka, zobrazenie obsahu zásobníka a odstránenie/uvoľnenie celého kontajnera. Našou snahou je zabezpečiť aj priamy náhodný prístup k prvkom.

Objasnenie príkladu autorom:

Kontajner bude disponovať rovnakou funkcionalitou, ako bolo objasnené v príklade v kapitole 2.3.2.1. Preto sa v tejto časti obmedzíme najmä na vysvetlenie rozdielov v implementáciách. Okrem zmien v názvoch funkcií a dátových štruktúr, sme definovali funkciu *dynamic_array_resize()*, ktorá ako parameter preberá samotný zoznam, t. j. dynamické pole. Účelom tejto funkcie je zdvojnásobiť veľkosť samotného zoznamu. Preto na riadku 48 definujeme lokálnu premennú *capacity*, ktorej veľkosť nastavíme ako dvojnásobok veľkosti už existujúceho zoznamu (riadok 48). Na základe tejto novej veľkosti sa postaráme o zväčšenie existujúceho pamäťového bloku pomocou funkcie *realloc()* (riadok 50). Obdobne ako aj funkcia *malloc()*, aj funkcia *realloc()* v prípade neúspechu vráti hodnotu nulového pointera. Preto na riadku 51 testujeme túto skutočnosť a v prípade, ak sa nepodarilo zväčšiť alokovaný priestor, funkciu ukončujeme s príznakom neúspechu. Ak proces zväčšenia priestoru bol úspešný, postaráme sa o aktualizáciu položky *items* na nový blok pamäte (riadok 53) a aktualizáciu veľkosti nového zoznamu (riadok 63) a funkciu ukončíme s príznakom úspechu.

```
46  result_type dynamic_array_resize(dynamic_array_type *array)
47  {
48      int capacity = array->capacity * 2;
49
50      item_type* items = realloc(array->items, sizeof(char)*capacity);
51      if (!items) return RESULT_ERROR;
52
53      array->items = items;
54      array->capacity = capacity;
55
56      return RESULT_OK;
57  }
```

K miernej modifikácii dochádza aj vo funkcii *dynamic_array_add_item()*, ktorá ako parameter preberá zoznam a hodnotu prvku, ktorý chceme do neho vložiť. Ako prvé zisťujeme na základe porovnania obsahu premenných *count* a *capacity* (riadok 61), či je v zozname ešte miesto. Ak tomu tak nie je, zavoláme funkciu *dynamic_array_resize()*, ktorá má za cieľ dvojnásobne zväčšiť veľkosť pôvodného zoznamu. Ak by táto funkcia skončila neúspechom, ukončíme aj funkciu *dynamic_array_add_item()* s príznakom neúspechu (riadky 63 a 64). V opačnom prípade na pozíciu určenú premennou *count* reprezentujúcu index prázdneho miesta v zozname vložíme prvok (riadok 67). Premennú *count* inkrementujeme o 1 (riadok 68) a funkciu ukončujeme s príznakom, že všetko prebehlo úspešne (riadok 70).

```
59  result_type dynamic_array_add_item(dynamic_array_type *array, item_type item)
60  {
61      if (array->count == array->capacity)
62      {
63          if (dynamic_array_resize(array))
64              return RESULT_ERROR;
65      }
66
67      array->items[array->count] = item;
68      array->count++;
69
70      return RESULT_OK;
71  }
```

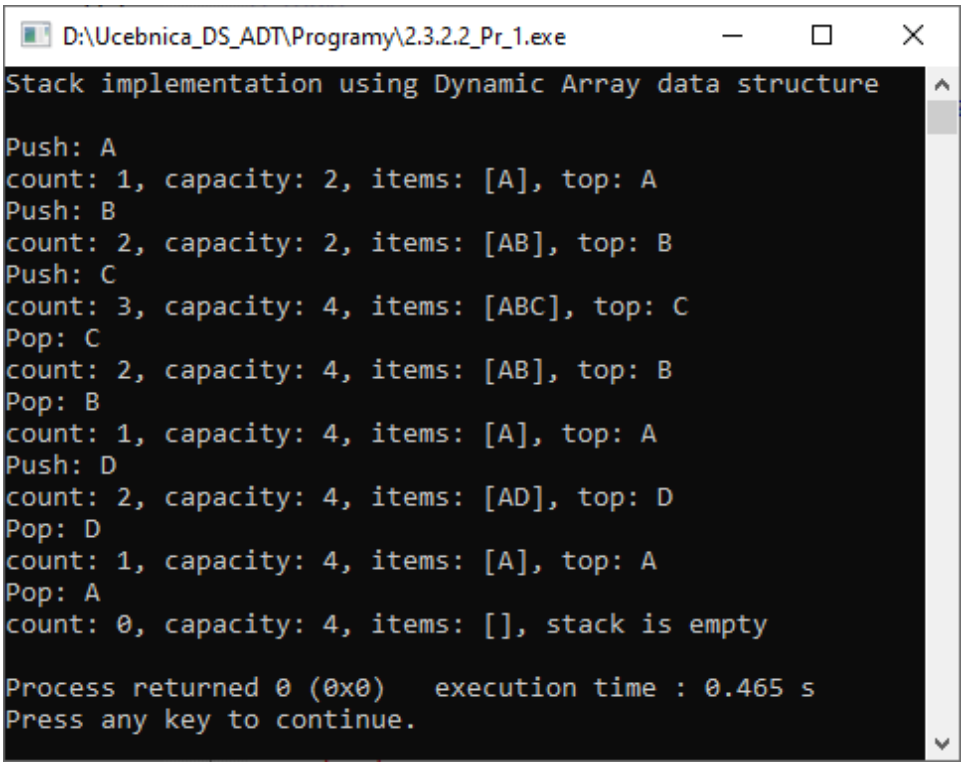
Modifikácia v implementácii, ale nie v logike funkcie, nastala aj pri funkcii *dynamic_array_remove_item_at_index()*. Úlohou tejto funkcie je odstránenia hodnoty prvku v zozname na konkrétnom indexe. Funkcia preberá tri parametre. Parameter, ktorý reprezentuje zoznam, index prvku, ktorý chceme odstrániť a parameter *item*, ktorý v sebe nesie hodnotu odstraňovaného prvku. Ak je index odstraňovaného prvku mimo rozsah zaplneného zoznamu (riadok 95), funkciu ukončujeme s neúspechom (riadok 96). Inak sa postaráme o uloženie hodnoty prvku do parametra *item* (riadok 98). Pre posuv prvkov ležiacich za odstraňovaným prvkom smerom doľava od pozície odstraňovaného prvku sme predtým použili cyklus s presne definovaným počtom opakovaní. V tomto prípade sme využili funkciu *memmove()* definovanú v knižnici *string.h*, ktorá má za cieľ prekopírovať všetky prvky ležiace za odstraňovaním prvkom na miesto určené jeho indexom. Počet týchto prvkov zistíme na základe vyhodnotenia výrazu: reálny počet prvkov – index odstraňovaného prvku – 1 (riadok 100). Je dôležité nezabudnúť dekrementovať hodnotu premennej *count* o 1 (riadok 102), ktorá signalizuje voľné miesto v zozname. Funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 104).

```
93  result_type dynamic_array_remove_item_at_index(dynamic_array_type *array, int index, item_type
*item)
94  {
95      if (index < 0 || index >= array->count)
96          return RESULT_ERROR;
97
98      *item = array->items[index];
99
100     memmove(array->items+index, array->items+index+1, sizeof(char) * (array->count-index-1));
101
102     array->count--;
103
104     return RESULT_OK;
105 }
```

Zvyšná funkcionalita je postavená na obdobnom princípe. Samotné telo programu je vytvorené identicky. Jediný rozdiel je na riadku 180, kde dochádza k vytvoreniu dátovej štruktúry zásobník o veľkosti 2. Predtým sme vytvárali zásobník o veľkosti 10, pretože jeho veľkosť sme nevedeli modifikovať. V tejto implementácii, kde využívame zoznam (dynamické pole) sa po zaplnení kapacity zásobníka, táto zväčší dvojnásobne.

```
174 // main
175 int main(void)
176 {
177     printf("Stack implementation using Dynamic Array data structure\n\n");
178 
179     // create
180     stack_type* stack = stack_create(2);
181     if (!stack)
182     {
183         printf("Create stack error\n");
184         return 1;
185     }
186 
187     item_type item;
188     result_type res;
189     int count;
190 
191     // push 'A'
192     item = 'A';
193     printf("Push: %c\n", item);
194     res = stack_push(stack, item);
195     if (res)
196         printf("Push to stack error\n");
197     stack_print(stack);
198     printf("\n");
199 
200     // push 'B'
201     item = 'B';
202     printf("Push: %c\n", item);
203     res = stack_push(stack, item);
204     if (res)
205         printf("Push to stack error\n");
206     stack_print(stack);
207     printf("\n");
208 
209     // push 'C'
210     item = 'C';
211     printf("Push: %c\n", item);
212     res = stack_push(stack, item);
213     if (res)
214         printf("Push to stack error\n");
215     stack_print(stack);
216     printf("\n");
217 
218 
219     // pop item
```

```
220     res = stack_pop(stack, &item);
221     if (res)
222         printf("Pop from stack error\n");
223     printf("Pop: %c\n", item);
224     stack_print(stack);
225     printf("\n");
226 
227     // pop item
228     res = stack_pop(stack, &item);
229     if (res)
230         printf("Pop from stack error\n");
231     printf("Pop: %c\n", item);
232     stack_print(stack);
233     printf("\n");
234 
235     // push 'D'
236     item = 'D';
237     printf("Push: %c\n", item);
238     res = stack_push(stack, item);
239     if (res)
240         printf("Push to stack error\n");
241     stack_print(stack);
242     printf("\n");
243 
244     // pop item
245     res = stack_pop(stack, &item);
246     if (res)
247         printf("Pop from stack error\n");
248     printf("Pop: %c\n", item);
249     stack_print(stack);
250     printf("\n");
251 
252     // pop item
253     res = stack_pop(stack, &item);
254     if (res)
255         printf("Pop from stack error\n");
256     printf("Pop: %c\n", item);
257     stack_print(stack);
258     printf("\n");
259 
260     // destroy
261     stack_destroy(stack);
262 
263     return 0;
264 }
```



Obr. 21 Konzolový výstup programu 2.3.2.2_Pr_1.c.

Jedna z možných implementácií celého programu 2.3.2.2_Pr_1.c.

2.3.2.3 Príklad – implementácia zásobníka pomocou jednostranne zreťazeného zoznamu

Zadanie problému:

Vytvorte program, ktorý bude implementovať zásobník pomocou jednostranne zreťazeného zoznamu ako abstraktný dátový typ. Kontajner bude slúžiť na uchovávanie informácií, ktoré pre jednoduchosť ilustrujeme ukladaním znakov. Tento bude umožňovať vytvorenie zásobníka, pridanie prvku, odstránenie prvku, zistenie počtu prvkov v zásobníku, prístup k vrcholu zásobníka, zobrazenie obsahu zásobníka a odstránenie/uvoľnenie celého kontajnera. Našou snahou je zabezpečiť aj priamy náhodný prístup k prvkom.

Objasnenie príkladu autorom:

Na riadkoch 5 a 6 deklarujeme dva vlastné dátové typy. *item_type* reprezentuje jednoduchý dátový typ *char*, ako ilustráciu, že typom prvku v kontajneri môže byť čokoľvek iné. *result_type*, ktorý reprezentuje jednoduchý dátový typ *int* používame pre označenie návratovej hodnoty funkcií, ktorých význam má byť, či funkcia skončila v poriadku, alebo nie. Konštanty RESULT_OK

a RESULT_ERROR definované na riadkoch 7 a 8 využívame pre návrat príznaku, či funkcia skončila v poriadku, alebo nie.

```
5  typedef char item_type;
6  typedef int result_type;
7
8  #define RESULT_OK 0
9  #define RESULT_ERROR 1
```

Pre implementáciu kontajnera typu zásobník ako prvé definujeme dátovú štruktúru, ktorá reprezentuje uzol jednostranne zreťazeného zoznamu *linked_list_node_type*. Pre uzol jednostranne zreťazeného zoznamu evidujeme položku *next*, ktorá bude reprezentovať ukazovateľ na nasledovníka a *item*, reprezentujúci dátovú časť uzla, v ktorej budeme udržiavať hodnotu typu *item_type* (riadky 11 – 16).

```
11 // linked list
12 typedef struct linked_list_node_type
13 {
14     struct linked_list_node_type* next;
15     item_type item;
16 } linked_list_node_type;
```

Samotnú dátovú štruktúru jednostranne zreťazeného zoznamu, ktorú reprezentujeme uchovávaním hlavy aj chvosta zoznamu spolu s evidovaním reálneho počtu prvkov, definujeme na riadkoch 18 – 23. Pre toto sme definovali vlastný dátový typ *linked_list_type*.

```
18 typedef struct linked_list_type
19 {
20     linked_list_node_type* head;
21     linked_list_node_type* tail;
22     int count;
23 } linked_list_type;
```

Funkciu, ktorá sa postará o vznik dátovej štruktúry jednostranne zreťazeného zoznamu, sme pomenovali *linked_list_create()*. Táto nepreberá žiadny parameter. Je to z dôvodu, že lineárny zreťazený zoznam môžeme chápať ako nekonečne dlhú dátovú štruktúru, ktorá je obmedzená len veľkosťou voľnej pamäte. Ako prvé alokujeme pamäť pre samotný zreťazený zoznam (riadok 27). Ak sa pamäť nepodarí alokovať, funkciu ukončíme a vrátime hodnotu *NULL*. V prípade úspechu pokračujeme inicializáciou pointerov predstavujúcich začiatok a koniec zreťazeného zoznamu. Tieto nastavujeme na *NULL*, pretože zreťazený zoznam je zatiaľ prázdny (riadky 31 a 32). Logicky, položku *count* nastavujeme na hodnotu 0 (riadok 33). Návratovou hodnotou funkcie je samotný zreťazený zoznam (riadok 35).

```
25  linked_list_type* linked_list_create()
26  {
27      linked_list_type* list = malloc(sizeof(linked_list_type));
28      if (!list)
29          return NULL;
30
31      list->head = NULL;
32      list->tail = NULL;
33      list->count = 0;
34
35      return list;
36  }
```

Funkciu pre uvoľnenie alokovanej pamäte sme nazvali *linked_list_destroy()*. Preberá jeden parameter, ktorým je dátová štruktúra zreťazeného zoznamu. V tele funkcie sa najprv postaráme o uvoľnenie pamäte pre jednotlivé uzly. Na to sme využili cyklus s podmienkou na začiatku, ktorý sa vykonáva až kým neprídeme na koniec zreťazeného zoznamu. V tele cyklu sme definovali lokálnu premennú *node*, do ktorej si zapamätávame adresu nasledovníka práve uvoľňovaného uzla (riadok 42). Ten bude novou hlavou zreťazeného zoznamu. Pomocou funkcie *free()* uvoľníme uzol, ktorý je prístupný cez pointer *head*. Následne len aktualizujeme začiatok zreťazeného zoznamu hodnotou uzla, ktorú sme si zapamätali v premennej *node* (riadok 44). Ako posledné sa postaráme o uvoľnenie pamäte reprezentujúcej samotnú dátovú štruktúru zreťazeného zoznamu (riadok 46).

```
38  void linked_list_destroy(linked_list_type *list)
39  {
40      while (list->head)
41      {
42          linked_list_node_type* node = node->next;
43          free(list->head);
44          list->head = node;
45      }
46      free(list);
47  }
```

Funkcia, ktorá vráti počet prvkov uložených v zreťazenom zozname sa nazýva *linked_list_count()*. Logicky preberá jeden parameter, ktorým je zreťazený zoznam. Počet prvkov zreťazeného zoznamu reprezentuje premenná *count*, ktorá je z funkcie vrátená cez návratovú hodnotu.

```
49  result_type linked_list_count(linked_list_type *list)
50  {
51      return list->count;
52  }
```

Pre pridanie prvku do dátovej štruktúry sme definovali funkciu *linked_list_add_item()*. Tá ako parameter preberá zreťazený zoznam a hodnotu prvku, ktorý chceme do neho vložiť. Ako prvé alokujeme miesto pre nový uzol (riadok 56). Ak sa nepodarí pamäť alokovať, funkciu ukončujeme s príznakom chyby pomocou konštanty *RESULT_ERROR* (riadky 57 a 58). V opačnom prípade

do dátovej časti uzla vložíme hodnotu prvku (riadok 60), časť nasledovníka nastavíme na hodnotu *NULL* (riadok 61) a zvýšime premennú *count* o 1 (riadok 62), ktorá predstavuje reálny počet prvkov v zreťazenom zozname. Pri vkladaní prvkov do zreťazeného zoznamu zvažujeme dve situácie, ktoré môžu nastať. Buď ide o situáciu vkladania uzlu do prázdneho zreťazeného zoznamu, alebo do zoznamu, kde sa už nejaké uzly nachádzajú. Prvú situáciu zohľadňuje podmienka na riadku 64. V tomto prípade sa nový uzol stáva začiatkom aj koncom zreťazeného zoznamu. V opačnom prípade nový uzol pridávame na koniec zreťazeného zoznamu (riadok 70) a nezabudneme aktualizovať chvost zreťazeného zoznamu na novo vložený uzol (riadok 71). Funkciu ukončujeme s príznakom, že všetko prebehlo úspešne (riadok 74).

```
54  result_type linked_list_add_item(linked_list_type *list, item_type item)
55  {
56      linked_list_node_type* node = malloc(sizeof(linked_list_node_type));
57      if (!node)
58          return RESULT_ERROR;
59
60      node->item = item;
61      node->next = NULL;
62      list->count++;
63
64      if (!list->head)
65      {
66          list->head = list->tail = node;
67      }
68      else
69      {
70          list->tail->next = node;
71          list->tail = list->tail->next;
72      }
73
74      return RESULT_OK;
75  }
```

Pole je dátová štruktúra v ktorej sú prvky prístupné cez indexy. To však neplatí pre zreťazený zoznam. Napriek tomu môžeme definovať funkciu, ktorá zabezpečí prístup k prvku na konkrétnom indexe, na základe predpokladu, že prvý uzol bude pre nás predstavovať prvok s indexom 0. Pre tento účel definujeme funkciu *linked_list_node_at_index()*. Táto preberá dva parametre: zreťazený zoznam a index prvku, ktorého adresu vraciame cez návratovú hodnotu funkcie. V tele funkcie ako prvé testujeme či index prvku ktorý chceme získať je v rozsahu nášho zaplneného zreťazeného zoznamu (riadok 79). Ak tomu tak nie je, vraciame príznak neúspechu funkcie pomocou nulového pointera (riadok 80). V opačnom prípade testujeme, či hodnota indexu nepredstavuje posledný prvok zreťazeného zoznamu. Obsah premennej *count*, ktorú udržiavame v zreťazenom zozname, reprezentuje reálny počet prvok zreťazeného zoznamu. Za predpokladu, že uzly indexujeme tak, ako

sa indexujú tieto v poli, t. j. od nuly, hodnotu premennej *count* musíme znížiť o 1 (riadok 82). V prípade vyhodnotenia tejto podmienky ako pravdivej, funkciu ukončujeme a vrátime adresu tohto uzla uloženého v chvoste zreťazeného zoznamu. V opačnom prípade musíme zreťazený zoznam prechádzať postupne, kým sa nedostaneme na požadovaný prvok. Pre tento účel sme definovali lokálnu premennú *node*, ktorú sme inicializovali hlavou zreťazeného zoznamu. Za využitia cyklu s podmienkou na začiatku, zabezpečujeme posuv na ďalší prvok až kým to je potrebné. Ako podmienku sme využili priamo hodnotu *index*, ktorú po každej iterácii dekrementuje o 1 pomocou postfixového zápisu. Je tomu tak z dôvodu, že najprv sa vyhodnotí, či už sa nenachádzame na požadovanom uzle určenom indexom (nezabúdame, že akákoľvek hodnota rôzna od 0 pre jazyk *C* reprezentuje logickú pravdu), a ak by nie, tak dôjde k dekrementácii obsahu parametra *index*. Obdobne funkciu ukončujeme tým, že vrátime adresu tohto uzla (riadok 89).

```
77  linked_list_node_type* linked_list_node_at_index(linked_list_type *list, int index)
78  {
79      if (index < 0 || index >= list->count)
80          return NULL;
81
82      if (index == list->count-1)
83          return list->tail;
84
85      linked_list_node_type* node = list->head;
86      while (index--)
87          node = node->next;
88
89      return node;
90  }
```

Pozn. autora: Dôvod návrhu tejto funkcie, ako aj nasledovných troch, je jednotný návrh implementácie zásobníka, ako abstraktného dátového typu, ktorý v kapitole 2.3.2 predstavujeme pomocou troch rôznych implementácií. Nie je to teda jediná možnosť, ako implementovať zásobník pomocou jednosmerne zreťazeného zoznamu.

Pre prístup k prvku na konkrétnom indexe definujeme funkciu *linked_list_get_item_at_index()*. Táto preberá tri parametre: zreťazený zoznam, index prvku, na ktorom sa nachádza prvok, ktorý chceme získať a parameter *item*, ktorý v sebe nesie hodnotu prvku. Pre tento parameter by bolo možné využiť návratovú hodnotu funkcie, avšak dodržujeme jednotný koncept návrhu funkcií, t. j. cez návratovú hodnotu funkcie vraciame príznak či funkcia skončila v poriadku, alebo nie. Preto je nutné tento parameter odovzdať odkazom, aby sme s hodnotou prvku mohli v main-e pracovať. V tele funkcie využívame skôr definovanú funkciu *linked_list_node_at_index()*. Ak funkcia nájde požadovaný prvok, vráti jeho adresu, v opačnom prípade vráti *NULL*. Tento ukladáme do lokálnej premennej *node*

(riadok 94). Preto na riadku 95 testujeme túto skutočnosť. Ak sa uzol nepodarilo nájsť, funkciu ukončujeme s príznakom neúspechu (riadok 96). V opačnom prípade do parametra *item* uložíme hodnotu uzlu na indexe danom parametrom *index*, ktorého adresu máme uloženú v lokálnej premennej *node* (riadok 98) a funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 100).

```
 92  result_type linked_list_get_item_at_index(linked_list_type *list, int index, item_type *item)
 93  {
 94      linked_list_node_type* node = linked_list_node_at_index(list, index);
 95      if (!node)
 96          return RESULT_ERROR;
 97
 98      *item = node->item;
 99
100      return RESULT_OK;
101  }
```

V prípade potreby editácie hodnoty prvku uloženej v uzli zreťazeného zoznamu sme definovali funkciu *linked_list_set_item_at_index()*, ktorá preberá tri parametre. Parameter, ktorý reprezentuje zreťazený zoznam, index prvku, ktorý chceme editovať a novú hodnotu tohto prvku. Opätovne, za využitia skôr definovanej funkcie *linked_list_node_at_index()* nájdeme požadovaný prvok. Tento ukladáme do lokálnej premennej *node* (riadok 105). Ak sa uzol nepodarilo nájsť, funkciu ukončujeme neúspechom (riadok 107). Inak sa postaráme o prepísanie hodnoty prvku identifikovanom parametrom *index* novou hodnotou, ktorá je uložená v parametri *item* (riadok 109) a funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 111).

```
103  result_type linked_list_set_item_at_index(linked_list_type *list, int index, item_type item)
104  {
105      linked_list_node_type* node = linked_list_node_at_index(list, index);
106      if (!node)
107          return RESULT_ERROR;
108
109      node->item = item;
110
111      return RESULT_OK;
112  }
```

V prípade potreby odstránenia uzla zreťazeného zoznamu sme definovali funkciu *linked_list_remove_item_at_index()*, ktorá preberá tri parametre. Parameter, ktorý reprezentuje zreťazený zoznam, index reprezentujúci uzol, ktorý chceme odstrániť a parameter *item*, ktorý v sebe nesie hodnotu odstraňovaného uzla. Ak je index, ktorý predstavuje odstraňovaný uzol mimo rozsahu počtu prvkov zreťazeného zoznamu (riadok 116), funkciu ukončujeme s neúspechom (riadok 117). Inak testujeme, či nejde o prvý uzol zreťazeného zoznamu, čo indikuje hodnota parametra *index* rovná 0 (riadok 119). Proces odstraňovania prvého uzla zreťazeného zoznamu je mierne odlišný

od odstraňovania ostatných uzlov. Vtedy, okrem iného, je potrebné modifikovať hlavu zreťazeného zoznamu. Ak ide o odstraňovanie prvého uzla, tak si do pomocného pointera označeného ako *node* zapamätáme jeho adresu (riadok 121). Následne sa postaráme o modifikáciu hlavy zreťazeného zoznamu. Novou hlavou bude nasledovník práve odstraňovaného uzla (riadok 122). Uloženie hodnoty dátovej časti uzla do parametra *item* realizujeme na riadku 123. Teraz už môžeme daný uzol odstrániť (riadok 124). Ak by zreťazený zoznam obsahoval práve jeden prvok, čo indikuje situácia že obsah pointera *head* (po jeho úprave) je rovný *NULL*, nastavujeme aj chvost zreťazeného zoznamu na *NULL* (riadok 125). V opačnom prípade, t. j. ak ide o odstránenie uzla rôzneho od prvého, tak pomocou funkcie *linked_list_node_at_index()* si nájdeme predchodcu odstraňovaného uzla. Tento ukladáme do lokálnej premennej *node_previous* (riadok 129). Je to z dôvodu, že využívame jednosmerne zreťazený zoznam a v takom prípade si uzly uchovávajú informáciu len o nasledovníkoch, t. j. nevieme sa z odstraňovaného uzla dostať na jeho predchodcu. Pri odstraňovaní uzla nachádzajúceho sa medzi dvoma uzlami sa musíme postarať o ich vzájomné prepojenie. Ak by sme nepoznali adresu predchodcu odstraňovaného uzla, nemali by sme ako toto prepojenie zrealizovať. Do lokálnej premennej *node* si uložíme adresu odstraňovaného uzla (riadok 130). Na riadku 131 sa staráme o vytvorenie prepojenia medzi predchodcom a nasledovníkom odstraňovaného uzla. Uloženie hodnoty dátovej časti uzla do parametra *item* realizujeme na riadku 132. Teraz už môžeme daný uzol odstrániť (riadok 133). Ak by išlo o odstraňovanie posledného uzla, je nutné aktualizovať chvost zreťazeného zoznamu na predchodcu tohto uzla, ktorého adresu máme uloženú v lokálnej premennej *node_previous* (riadky 134 a 135). Je dôležité nezabudnúť dekrementovať hodnotu premennej *count* o 1 (riadok 138). Funkciu ukončíme s príznakom úspechu, pomocou konštanty *RESULT_OK* (riadok 140).

```
114 result_type linked_list_remove_item_at_index(linked_list_type *list, int index, item_type *item)
115 {
116     if (index < 0 || index >= list->count)
117         return RESULT_ERROR;
118
119     if (index == 0)
120     {
121         linked_list_node_type* node = list->head;
122         list->head = node->next;
123         *item = node->item;
124         free(node);
125         if (!list->head) list->tail = NULL;
126     }
127     else
128     {
129         linked_list_node_type* node_previous = linked_list_node_at_index(list, index-1);
130         linked_list_node_type* node = node_previous->next;
```

```
131             node_previous->next = node->next;
132             *item = node->item;
133             free(node);
134             if (!node_previous->next)
135                 list->tail = node_previous;
136         }
137 
138         list->count--;
139 
140         return RESULT_OK;
141     }
```

Pre vypísanie obsahu dátovej štruktúry sme definovali funkciu *linked_list_print()* preberajúcu jeden parameter, ktorým je zreťazený zoznam. V tele funkcie sa postaráme o výpis obsahu premennej *count*, ktorá reprezentuje reálny počet prvkov v zreťazenom zozname (riadok 145). Pomocou cyklu s podmienku na začiatku, ktorý beží až kým lokálna premenná *node* definovaná na riadku 146 nadobudne hodnotu *NULL*. V tele cyklu sa postaráme o vypísanie hodnôt jednotlivých uzlov (riadok 149) a posuv na ďalší uzol (riadok 150).

```
143 void linked_list_print(linked_list_type *list)
144 {
145     printf("count: %d, items: [", list->count);
146     linked_list_node_type* node = list->head;
147     while (node)
148     {
149         printf("%c", node->item);
150         node = node->next;
151     }
152     printf("]");
153 }
```

Samotný dátový typ reprezentujúci kontajner typu zásobník definujeme na riadkoch 155 – 159, pomocou štruktúry, ktorá obsahuje práve jednu položku a tou je nami definovaný jednosmerne zreťazený zoznam typu *linked_list_type*.

```
155 // stack using linked list
156 typedef struct stack_type
157 {
158     linked_list_type* list;
159 } stack_type;
```

Funkciu, ktorá sa postará o vznik dátovej štruktúry zásobník sme pomenovali *stack_create()*. Tá nemá žiadny parameter, pretože zreťazený zoznam je kvázi nekonečná dátová štruktúra. Ako prvé alokujeme pamäť pre samotnú dátovú štruktúru (riadok 163). Ak sa pamäť nepodarí alokovať, funkciu ukončíme a vrátime hodnotu *NULL*. V prípade úspechu pokračujeme vytvorením zreťazeného zoznamu pomocou definovanej funkcie *linked_list_create()* (riadok 167). Ak by táto funkcia skončila neúspechom, t. j. funkcia vráti hodnotu *NULL*, funkciu ukončíme a vrátime hodnotu

NULL. V opačnom prípade je návratovou hodnotou funkcie adresa dátovej štruktúry, ktorá reprezentuje zásobník (riadok 171).

```
161  stack_type* stack_create()
162  {
163      stack_type* stack = malloc(sizeof(stack_type));
164      if (!stack)
165          return NULL;
166
167      stack->list = linked_list_create();
168      if (!stack->list)
169          return NULL;
170
171      return stack;
172  }
```

Funkciu pre uvoľnenie alokovanej pamäte sme nazvali *stack_destroy()*. Preberá jeden parameter, ktorým je dátová štruktúra zásobník. V tele funkcie sa najprv postaráme o uvoľnenie pamäte reprezentujúcej zreťazený zoznam pomocou definovanej funkcie *linked_list_destroy()* (riadok 176) a následne uvoľníme pamäť reprezentujúcu samotnú dátovú štruktúru zásobníka (riadok 177).

```
174  void stack_destroy(stack_type* stack)
175  {
176      linked_list_destroy(stack->list);
177      free(stack);
178  }
```

Funkcia, ktorá vráti počet prvkov uložených v zásobníku sa nazýva *stack_count()*. Logicky preberá jeden parameter, ktorým je zásobník. Počet prvkov zásobníka zistíme zavolaním funkcie *linked_list_count()*, ktorej je ako parameter odovzdaná položka zásobníka s názvom *list*. Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_count()*.

```
180  result_type stack_count(stack_type* stack)
181  {
182      return linked_list_count(stack->list);
183  }
```

Pre pridanie prvku do dátovej štruktúry sme definovali funkciu *stack_push()*. Tá ako parameter preberá zásobník a hodnotu prvku, ktorý chceme do neho vložiť. Pre samotné vloženie prvku do zásobníka využívame definovanú funkciu *linked_list_add_item()*, ktorej sú odovzdané argumenty reprezentujúce zreťazený zoznam, cez ktorý je zásobník implementovaný a hodnota samotného prvku. Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_push()* (riadok 187).

```
185  result_type stack_push(stack_type* stack, item_type item)
186  {
187      return linked_list_add_item(stack->list, item);
188  }
```

Zásobník je dátová štruktúra, ktorá vyberá/odstraňuje prvky z vrcholu zásobníka, t. j. prvok, ktorý bol do zásobníka vložený ako posledný, bude ako prvý odstránený. Z toho dôvodu vo funkcii *stack_pop()* ako prvé zistíme počet prvkov nachádzajúcich sa v zásobníku pomocou funkcie *linked_list_count()* a túto hodnotu uložíme do lokálnej premennej *count* (riadok 192). Následne prvok zo zásobníka odstránime pomocou funkcie *linked_list_remove_item_at_index()*, pričom samotný index tohto prvku je definovaný premennou *count* zníženou o 1, keďže premenná *count* pri našej implementácii reprezentuje reálny počet prvkov v zreťazenom zozname (riadok 193). Parameter funkcie *item* je odovzdaný odkazom, aby sme mohli s hodnotou odstraňovaného prvku v main–e pracovať, napríklad v našom programe ju vypisujeme (riadky 263, 271, 288 a 296). Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_pop()*.

```
190  result_type stack_pop(stack_type* stack, item_type *item)
191  {
192      int count = linked_list_count(stack->list);
193      return linked_list_remove_item_at_index(stack->list, count-1, item);
194  }
```

Zásobník, ako dátová štruktúra definuje aj funkciu pre prístup k prvku na vrchole zásobníka. V tomto prípade nejde o odstránenie tohto prvku, ale len o jeho sprístupnenie. Pre tento účel sme definovali funkciu *stack_top()*, ktorá preberá dva parametre. Parameter, ktorý reprezentuje zásobník a parameter do ktorého je uložená hodnota tohto prvku, obidva sú odovzdané odkazom. Ako prvé zistíme počet prvkov nachádzajúcich sa v zásobníku pomocou funkcie *linked_list_count()* a túto hodnotu uložíme do lokálnej premennej *count* (riadok 198). Následne prvok zo zásobníka sprístupnime pomocou funkcie *linked_list_get_item_at_index()*, pričom samotný index tohto prvku je definovaný premennou *count* zníženou o 1, keďže premenná *count* pri našej implementácii reprezentuje reálny počet prvkov v zreťazenom zozname (riadok 199). Návratová hodnota tejto funkcie je zároveň vrátená cez návratovú hodnotu funkcie *stack_top()*.

```
196  result_type stack_top(stack_type* stack, item_type *item)
197  {
198      int count = linked_list_count(stack->list);
199      return linked_list_get_item_at_index(stack->list, count-1, item);
200  }
```

Pre vypísanie obsahu dátovej štruktúry sme definovali funkciu *stack_print()*. Tá preberá jeden parameter, ktorým je zásobník. V tele funkcie sa postaráme o výpis obsahu zásobníka pomocou definovanej funkcie *linked_list_print()* (riadok 204). Lokálna premenná *top* je deklarovaná vo funkcii na riadku 206 pre sprístupnenie prvku v zásobníku, ktorý sa nachádza na jeho vrchole. Pre prístup k tomuto prvku sme využili definovanú funkciu *stack_top()* (riadok 207) a podľa návratovej hodnoty tejto funkcie, ktorú ukladáme do lokálnej premennej *res* reagujeme buď vypísaním hodnoty tohto

prvku, ak zásobník nejaké prvky obsahuje, alebo výpisom „*stack is empty*“ v prípade, ak je zásobník prázdny (riadky 208 –212).

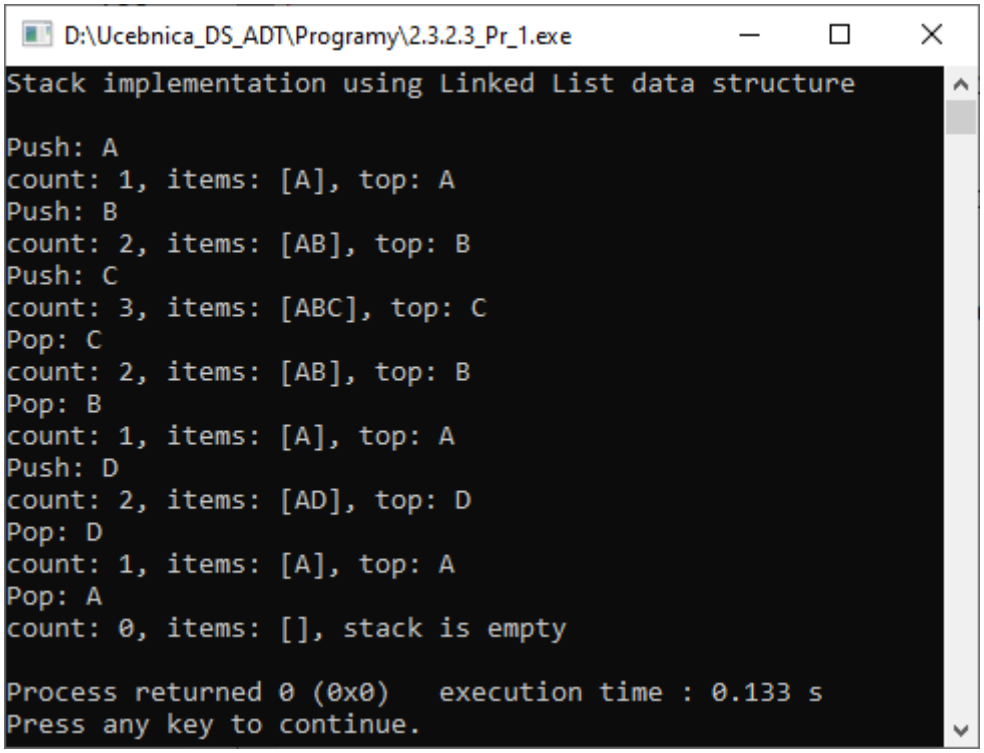
```
202 void stack_print(stack_type *stack)
203 {
204     linked_list_print(stack->list);
205 
206     item_type top;
207     result_type res = stack_top(stack, &top);
208     if (res)
209         printf(", stack is empty");
210     else
211         printf(", top: %c", top);
212 }
```

Samotné telo programu je vytvorené identicky ako v predchádzajúcich dvoch príkladoch. Jediný rozdiel je na riadku 220, kde dochádza k vytvoreniu dátovej štruktúry zásobník bez definovania veľkosti. Predtým sme vytvárali zásobník o veľkosti 10, pretože jeho veľkosť sme nevedeli modifikovať. V druhom príklade sme definovali zásobník o veľkosti 2, aby sme mohli ilustrovať, že jeho veľkosť sa automaticky po zaplnení zdvojnásobuje. V tejto implementácii, kde využívame jednosmerne zreťazený zoznam vôbec nie je nutné špecifikovať veľkosť.

```
214 // main
215 int main(void)
216 {
217     printf("Stack implementation using Linked List data structure\n\n");
218 
219     // create
220     stack_type* stack = stack_create();
221     if (!stack)
222     {
223         printf("Create stack error\n");
224         return 1;
225     }
226 
227     item_type item;
228     result_type res;
229     int count;
230 
231     // push 'A'
232     item = 'A';
233     printf("Push: %c\n", item);
234     res = stack_push(stack, item);
235     if (res)
236         printf("Push to stack error\n");
237     stack_print(stack);
238     printf("\n");
239 
240     // push 'B'
241     item = 'B';
242     printf("Push: %c\n", item);
```

```
243     res = stack_push(stack, item);
244     if (res)
245         printf("Push to stack error\n");
246     stack_print(stack);
247     printf("\n");
248 
249     // push 'C'
250     item = 'C';
251     printf("Push: %c\n", item);
252     res = stack_push(stack, item);
253     if (res)
254         printf("Push to stack error\n");
255     stack_print(stack);
256     printf("\n");
257 
258 
259     // pop item
260     res = stack_pop(stack, &item);
261     if (res)
262         printf("Pop from stack error\n");
263     printf("Pop: %c\n", item);
264     stack_print(stack);
265     printf("\n");
266 
267     // pop item
268     res = stack_pop(stack, &item);
269     if (res)
270         printf("Pop from stack error\n");
271     printf("Pop: %c\n", item);
272     stack_print(stack);
273     printf("\n");
274 
275     // push 'D'
276     item = 'D';
277     printf("Push: %c\n", item);
278     res = stack_push(stack, item);
279     if (res)
280         printf("Push to stack error\n");
281     stack_print(stack);
282     printf("\n");
283 
284     // pop item
285     res = stack_pop(stack, &item);
286     if (res)
287         printf("Pop from stack error\n");
288     printf("Pop: %c\n", item);
289     stack_print(stack);
290     printf("\n");
291 
292     // pop item
293     res = stack_pop(stack, &item);
294     if (res)
295         printf("Pop from stack error\n");
296     printf("Pop: %c\n", item);
297     stack_print(stack);
```

```
298     printf("\n");
299
300     // destroy
301     stack_destroy(stack);
302
303     return 0;
304 }
```



Obr. 22 Konzolový výstup programu 2.3.2.3_Pr_1.c.

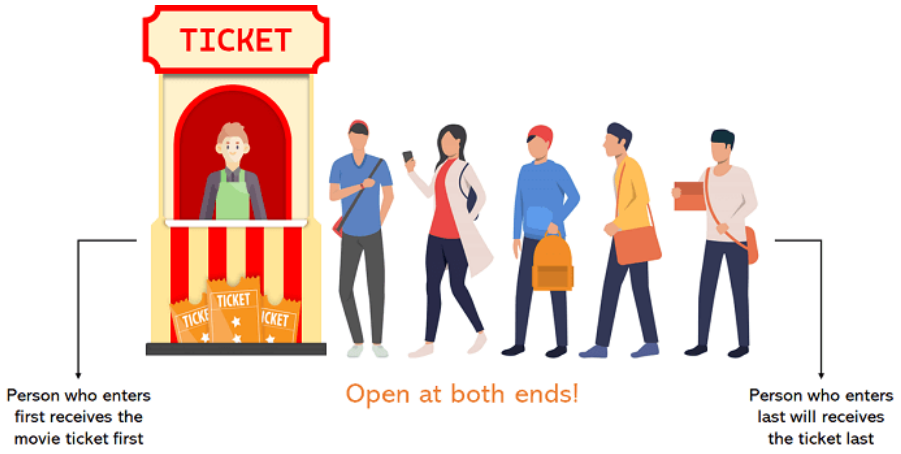
Jedna z možných implementácií celého programu 2.3.2.3_Pr_1.c.

2.3.3 Front = *queue*

Front je homogénna lineárna dátová štruktúra fungujúca na princípe *FIFO*, t. j. použijeme ju v prípade, ak potrebujeme prvky spracovávať v takom poradí, ako boli do štruktúry vložené. Umožňuje prístup iba k prvku na začiatku, pričom prvky sú vkladané na koniec fronty. Pri práci s frontom sú k dispozícii operácie ako "*put*" (vloženie prvku na koniec fronty), "*get*" (odstránenie prvku zo začiatku fronty), "*front*" (získanie prvku zo začiatku fronty, bez jeho odstránenia), a "*isEmpty*" (overenie, či je front prázdny) a pod.

Podobne ako zásobník, je možné aj front implementovať pomocou poľa alebo zreťazeného zoznamu. Pri použití zreťazeného zoznamu dochádza k eliminácii problémových operácií pri mazaní prvkov, keďže v zreťazenom zozname každý uzol vzniká dynamicky ako samostatný prvok.

Ako príklad reálnej predstavy sa často ilustruje front ako rad zákazníkov v obchode.



Obr. 23 Vizuálna reprezentácia fronty.

Dve najbežnejšie operácie vykonávané vo fronte sú *Enqueue()* a *Dequeue()*. *Enqueue()* pridá prvok do fronty, zatiaľ čo *Dequeue()* z nej odstráni prvok.

Časové zložitosti (*time complexity*) operácií nad dátovou štruktúrou:

- Prístup k prvkom (*access*) – O(n) = lineárna časová zložitosť. Prístup k prvkom vo fronte v najhoršom prípade vyžaduje výber (odstránenie) jedného prvku po druhom, kde *n* predstavuje počet prvkov vo fronte.
- Vyhľadávanie v dátovej štruktúre (*search*) – O(n) = lineárna časová zložitosť. Hľadanie konkrétneho výskytu prvku vo fronte v najhoršom prípade vyžaduje výber (odstránenie) jedného prvku po druhom a jeho porovnanie s cieľom, kde *n* predstavuje počet prvkov vo fronte.
- Vloženie/vymazanie prvku (*insert/delete*) – O(1) = konštantná časová zložitosť. Operácia vloženia/vymazania prvku vo fronte sa dá vždy spraviť s konštantnou časovou zložitosťou.

Typické použitie:

- Front sa často používa na správu úloh alebo požiadaviek, ktoré majú byť vykonané postupne. Každá úloha alebo požiadavka sa vkladá na koniec fronty a vykonávajú sa v poradí, v akom boli prijaté. Príkladom môže byť front úloh pre plánovač úloh v operačnom systéme (*Task Management*). Iným príkladom môže byť spravovanie správ (*Message Queuing*). Správy sa vkladajú do fronty a následne sa doručujú správcom alebo príjemcom. Toto umožňuje asynchrónne spracovanie správ a zabezpečuje ich správne doručenie v poradí. V sieťových systémoch ako sú serverové farmy alebo webové servery sa fronty používajú na správu

čakajúcich požiadaviek od klientov. Požiadavky sa zaradia do fronty a následne sú spracovávané podľa definovaných pravidiel, čím sa zabezpečuje správne a efektívne rozdeľovanie záťaže.

- Fronty sa často využívajú pri spracovaní udalostí (*Event Processing*), ako sú udalosti z používateľského rozhrania, systémové udalosti alebo udalosti z komunikačných kanálov. Udalosti sa vkladajú na koniec fronty a následne sa spracovávajú v poradí, v akom boli prijaté.
- Front môže slúžiť ako synchronizačný mechanizmus medzi viacerými procesmi alebo vláknami. Napríklad v rámci paralelného spracovania úloh môžu procesy vkladať úlohy do fronty a následne ich ďalšie procesy môžu vytiahnuť a spracovať.
- Fronty sa používajú v rôznych algoritmoch vyhľadávania a prehľadávania, ako je napríklad prehľadávanie do šírky (*Breadth-first Search*). Prvky sa postupne pridávajú do fronty a následne sa spracovávajú v poradí, v akom boli vložené.
- Fronty sa často používajú v rámci systémov, kde je potrebné riadiť rad zákazníkov. Napríklad v bankách, obchodoch, letiskách sa fronty používajú na usporiadanie prichádzajúcich zákazníkov a riadenie ich obsluhy v poradí, v akom prichádzajú, tzv. *Customer Queue*.
- Front je využitý aj pri riadení tlačiarní a spracovaní tlačových úloh. Dokumenty, ktoré majú byť vytlačené, sa vkladajú do fronty. Tlačové úlohy sú spracovávané postupne v poradí, v akom boli vložené do fronty. Ak máte viacero tlačiarní zdieľaných medzi viacerými používateľmi alebo systémami, môže byť použitá na riadenie prístupu k tlačiarňam. Tlačové žiadosti sa vkladajú do fronty a následne sa spracovávajú v poradí. To zabezpečuje, že tlačové úlohy sú spracované spravodlivo a súčasne to zabraňuje nadmernému zaťaženiu tlačiarní.
- V komunikačných systémoch, ako sú e-mailové servery alebo systémy správ, sa fronty používajú na spracovanie prichádzajúcich správ. Správy sa vkladajú do fronty a následne sú spracovávané a doručované príjemcom v poradí, v akom boli prijaté.

2.4 Príklady pre samostatné precvičovanie vedomostí

Pri všetkých úlohách sa snažte implementovať čo najviac operácií, ktoré je možné s abstraktnými dátovým typmi realizovať.

1. Vytvorte program, ktorý bude implementovať množinu pomocou poľa (môžete zvoliť implementáciu pomocou statického, vhodnejšie pomocou dynamického poľa).
2. Vytvorte program, ktorý bude implementovať množinu pomocou zreťazeného zoznamu (môžete zvoliť implementáciu pomocou jednosmerne zreťazeného zoznamu, ako aj pomocou obojsmerne zreťazeného zoznamu).
3. Vytvorte program, ktorý bude implementovať multimmnožinu pomocou poľa (môžete zvoliť implementáciu pomocou statického, vhodnejšie pomocou dynamického poľa).
4. Vytvorte program, ktorý bude implementovať multimnožinu pomocou zreťazeného zoznamu (môžete zvoliť implementáciu pomocou jednosmerne zreťazeného zoznamu, ako aj pomocou obojsmerne zreťazeného zoznamu).
5. Vytvorte program, ktorý bude implementovať multimnožinu pomocou dvojice lineárnych zreťazených zoznamov (jeden zoznam bude udržiavať hodnoty prvkov, druhý bude mapovať ich výskyt).
6. Vytvorte program, ktorý bude implementovať zásobník pomocou obojstranne zreťazeného zoznamu.
7. Vytvorte program, ktorý bude implementovať front pomocou poľa (môžete zvoliť implementáciu pomocou statického, vhodnejšie pomocou dynamického poľa).
8. Vytvorte program, ktorý bude implementovať front pomocou zreťazeného zoznamu (môžete zvoliť implementáciu pomocou jednosmerne zreťazeného zoznamu, ako aj pomocou obojsmerne zreťazeného zoznamu).

3 Zoznam použitej literatúry a informačných zdrojov

ANWAR, A. 2021. Seven (7) Essential Data Structures for a Coding Interview and associated common questions. [online]. [cit. 18.8.2022]. Dostupné na: ⟨https://towardsdatascience.com/seven-7-essential-data-structures-for-a-coding-interview-and-associated-common-questions-72ceb644290⟩.

BELAN, Anino, 2011. Kurz Jazyk C. Učebný text pre kvartu a kvintu osemročného gymnázia. 2. vydanie. [online]. [cit. 26.7.2019]. Dostupné na: ⟨https://docplayer.gr/24355101-Kurz-jazyka-c-ucebny-text-pre-kvartu-a-kvintu-osemrocneho-gymnazia.html⟩.

GEEKSFORGEEKS. 2019. Is sizeof for a struct equal to the sum of sizeof of each member? [online]. [cit. 18.5.2020]. Dostupné na: ⟨https://www.geeksforgeeks.org/is-sizeof-for-a-struct-equal-to-the-sum-of-sizeof-of-each-member/⟩.

HEROUT, Pavel, 2010. Učebnice jazyka C 1. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-383-8.

HEROUT, Pavel, 2010. Učebnice jazyka C 2. díl. České Budějovice : Kopp nakladatelství. ISBN 978-80-7232-367-8.

HOROVČÁK, Pavel a Igor PODLUBNÝ, 1997. Úvod do programovania v jazyku C. [online]. [cit. 26.7.2019]. Dostupné na: ⟨http://people.tuke.sk/igor.podlubny/C/⟩.

LOSHIN, D. TechTarget. Definition data structures. © 2005–2021. [online]. [cit. 26.7.2022]. Dostupné na: ⟨https://www.techtarget.com/searchdatamanagement/definition/data-structure⟩.

PALMÁROVÁ, Viera, 2003. Dynamické údajové štruktúry. [online]. [cit. 18.7.2020]. Dostupné na: ⟨http://cec.truni.sk/stoffov/dynamicke-udajove-struktury/start.html⟩.

Vivien, L. 2022. Data structures and Java collections. [online]. [cit. 18.8.2022]. Dostupné na: ⟨https://www.lavivienpost.com/data-structures-and-java-collections/⟩.

WIKIPÉDIA, 2020. Matica. © 2020. [online]. [cit. 26.7.2020]. Dostupné na: ⟨https://sk.wikipedia.org/wiki/Matica_(matematika)⟩.

WRÓBLEVSKI, Piotr, 2004. Algoritmy. Datové struktury a programovací techniky. Brno : Computer Press. ISBN 80-251-0343-9.

Názov:	Dátové štruktúry. Abstraktné dátové typy. Mierne pokročilé štruktúrované programovanie v jazyku C. (vysokoškolská učebnica)
Autor:	doc. Ing. Jana Jurinová, PhD.
Recenzenti:	prof. Ing. Pavel Važan, PhD. Ing. Lukáš Herout, Ph.D.
Vydavateľ:	Univerzita sv. Cyrila a Metoda v Trnave
Rok:	2023
Rozsah:	128 strán/ 7,12 AH
Grafická úprava obálky:	doc. Ing. Jana Jurinová, PhD.
Forma vydania:	online

https://www.ucm.sk/download/Ucebnica_DS_ADT.pdf?s=NjA6YTFlODFlNTg6ZDoxOjM1Y2JiYSAg