Dynamické datové struktury

Pavel Töpfer, KSVI MFF UK Praha

OBSAH

LITERATURA

Různé algoritmy pro práci s dynamickými datovými strukturami naleznete vedle tohoto textu také v učebnicích

Pavel Töpfer: Algoritmy a programovací techniky, Prometheus Praha 1995, 2. vyd. 2007

Niklaus Wirth: Algoritmy a štruktúry údajov, Alfa Bratislava 1988

1. Způsoby alokace paměti

1.1 Základní druhy proměnných

Proměnná slouží k uchovávání dat a dalších pomocných informací v programech po dobu výpočtu. Každá proměnná má nějaké označení, pomocí něhož se na ni příkazy programu odvolávají. U běžných proměnných deklarovaných jak v hlavním programu, tak i v procedurách a funkcích používáme k tomuto označení **jméno proměnné**, které má podobu tzv. **identifikátoru** (ve standardním Pascalu je to posloupnost písmen a číslic začínající písmenem, v Turbo Pascalu se navíc v identifikátoru připouštějí znaky '_'). Jména všech používaných proměnných jsou zavedena v úseku deklarací proměnných za klíčovým slovem *var*. V deklaracích uvádíme také typ každé proměnné. **Typ proměnné** určuje, jaké hodnoty se do ní mohou ukládat. Tím je zároveň stanoveno, jak velký úsek paměti je třeba této proměnné vyhradit. Tak třeba proměnná typu *char* určená k uložení jednoho znaku vystačí s jedním bytem paměti, zatímco pole o sto znacích (např. *array [1..100] of char*) potřebuje pochopitelně bytů sto. Zanedlouho se seznámíme s tzv. dynamicky alokovanými proměnnými, na které se v programu neodvoláváme přiděleným jménem, ale přímo jejich adresou v paměti počítače. Jak uvidíme, také každá dynamicky alokovaná proměnná má pevně stanovenou velikost, kolik místa zabírá v paměti. Její velikost bývá většinou určena typem ukládaných dat stejně, jako je tomu u „obyčejných" proměnných, v Turbo Pascalu je kromě toho možné ponechat určení velikosti dynamicky alokované proměnné až na dobu výpočtu programu.

Již jsme naznačili, že se v programech setkáváme s několika způsoby přidělování paměti pro proměnné (s tzv. **alokací paměti**). Jednotlivé způsoby alokace si nyní rozebereme podrobněji. V principu můžeme rozlišit tři základní metody: statická alokace, alokace na zásobníku a dynamická alokace na haldě.

Nejjednodušší je **statická alokace paměti**, která spočívá v pevném umístění proměnných do paměti na celou dobu výpočtu. V Pascalu se používá pro proměnné hlavního programu. Tyto proměnné musí existovat po celou dobu výpočtu programu, a proto se jim paměť přiděluje „napevno" při spuštění programu. Velikost paměťového prostoru určeného pro staticky alokované proměnné může být v závislosti na konkrétní implementaci jazyka nějak omezena. V Turbo Pascalu je toto omezení dáno způsobem adresování paměti na počítači typu PC. Všechny staticky alokované proměnné se umisťují do jednoho datového segmentu, takže jejich souhrnné paměťové nároky nemohou překročit 64 KB. Výhodou statické alokace je její jednoduchost, přístup k takovýmto proměnným z programu je nejsnadnější a nejrychlejší. Naopak nevýhodou statické alokace je skutečnost, že tyto proměnné zabírají místo v paměti po celou dobu výpočtu programu. To pochopitelně nevadí u proměnných hlavního programu, které po celou dobu výpočtu tak jako tak musí neustále existovat. Staticky alokovat proměnné procedur či funkcí by však bylo zbytečným plýtváním místa v paměti a navíc by zcela znemožnilo provádět rekurzivní volání. (Tak tomu bylo například v implementacích programovacího jazyka Fortran.) Další nevýhodou statické alokace je již zmíněný omezený prostor pro proměnné tohoto druhu.

Druhou metodou přidělování paměti proměnným je jejich **alokace na zásobníku**. Tento způsob alokace je prováděn za běhu programu vždy v okamžiku, když jsou příslušné proměnné potřeba. Je využíván pro lokální proměnné procedur a funkcí. V paměti počítače je vyhrazena oblast, v níž je umístěn tzv. systémový zásobník. Ten je na začátku výpočtu prázdný. Kdykoliv je během výpočtu programu zavolána nějaká procedura nebo funkce, je na vrchol zásobníku umístěn její aktivační záznam. Aktivační záznam obsahuje prostor pro umístění všech lokálních proměnných a parametrů volaného podprogramu a vedle toho ještě další technické údaje, které jsou potřebné například k zajištění přístupu ke globálním proměnným a také k pozdějšímu správnému ukončení podprogramu (obsahuje mimo jiné návratovou adresu, což je místo v kódu programu, odkud byl podprogram zavolán). Po ukončení výpočtu procedury či funkce je její aktivační záznam ze zásobníku odstraněn.

Tím zároveň zaniknou lokální proměnné této procedury (funkce) a uvolní na zásobníku místo pro pozdější využití při dalším volání nějakého podprogramu.

Výhodou alokace na zásobníku je dobré hospodaření s paměťovým prostorem. V každém okamžiku výpočtu je přidělena paměť pouze těm proměnným, které skutečně potřebujeme, tzn. jenom proměnným deklarovaným v právě rozpočítaných podprogramech. Pokud například volá hlavní program během svého výpočtu střídavě procedury *A, B, C*, pak se lokální proměnné těchto tří procedur střídají na stejném místě v paměti. Alokace proměnných na zásobníku také umožňuje provádět rekurzivní volání procedur a funkcí. V tomto případě se na zásobníku jednoduše objeví více aktivačních záznamů odpovídajících témuž podprogramu (každý záznam odpovídá jednomu zavolání tohoto podprogramu), což je plně v souladu s tím, že každý rekurzivní exemplář procedury nebo funkce musí mít svoji vlastní kompletní sadu lokálních proměnných a parametrů.

Podobně, jako bývá omezen celkový prostor pro staticky alokované proměnné, může omezovat konkrétní implementace také velikost paměti vyhrazené pro systémový zásobník. V případě Turbo Pascalu to je standardně pouhých 16 KB s možností rozšíření maximálně až na 64 KB. Požadovanou velikost zásobníku je možné nastavit pomocí menu v integrovaném prostředí Turbo Pascalu nebo přímo ve zdrojovém textu programu direktivou překladače {$M}.

Třetí základní metodou přidělování paměti proměnným je **dynamická alokace na haldě**. Po úvodním krátkém opakování se tak konečně dostáváme k hlavnímu tématu tohoto textu. V názvu uvedená „dynamičnost“ znamená, že na rozdíl od předchozích druhů proměnných stačí až v průběhu výpočtu určit (na základě vstupních dat, dosavadního průběhu výpočtu, hodnot mezivýsledků apod.), kolik a jak velkých dynamických proměnných budeme potřebovat. Z tohoto důvodu nemohou mít dynamicky alokované proměnné předem přidělena ani svá jména, tyto proměnné vůbec nedeklarujeme, neboť předem nevíme, kolik jich bude při výpočtu zapotřebí. Na dynamicky alokované proměnné se odkazujeme přímo jejich adresou v paměti počítače. Program musí během výpočtu sám požádat o přidělení každé dynamické proměnné. Pokud systém této žádosti může vyhovět, tzn. pokud existuje v paměti volné místo požadované velikosti, přidělí proměnné vhodné místo v paměti a programu sdělí její paměťovou adresu. Podobně se musí program sám postarat o uvolnění paměti, kterou zabírají jeho již nepotřebné dynamické proměnné. Program tedy předá systému adresu uvolňované dynamicky alokované proměnné a její velikost a systém se již postará o skutečné uvolnění paměti.

Dynamicky alokované proměnné jsou umisťovány do vyhrazené paměťové oblasti nazývané **halda** (anglicky heap). Na začátku výpočtu je halda prázdná a jednotlivé žádosti programu o dynamickou alokaci paměti jsou realizovány jedna za druhou. Uvolňování paměti však neprobíhá nutně přesně v opačném pořadí, jako tomu bylo v případě zásobníku. Zatímco v zásobníku je obsazen v každém okamžiku souvislý úsek paměti, na haldě mohou vznikat při postupném uvolňování jednotlivých dynamických proměnných „díry“. Řešení tohoto problému - tzv. **fragmentace paměti** - je záležitostí systému. Správce haldy se snaží spojovat sousedící uvolněné úseky do větších celků a nově přicházející žádosti o přidělení paměti uspokojovat tak, aby se opětovně co nejlépe využila uvolněná místa uvnitř haldy.

Uvolňování již nepotřebných dynamicky alokovaných proměnných není v principu nezbytné, pokud máme jistotu, že celkové paměťové požadavky vznesené programem během výpočtu nepřekročí maximální kapacitu haldy. Při ukončení výpočtu programu je pak samozřejmě celá halda automaticky zrušena. Nemáme-li však tuto jistotu, budeme se při psaní programu držet zásady „slušného chování“ a budeme vracet všechno, co jsme si půjčili, jakmile již půjčenou věc (v tomto případě dynamicky alokovanou proměnnou) nepotřebujeme.

Umístění haldy v paměti počítače a její celková velikost opět závisejí na konkrétním překladači a na velikosti volné paměti počítače. V některých systémech se používala metoda, že po umístění prováděného binárního kódu (tzn. přeloženého programu) a statických proměnných do paměti počítače se zbývající volná paměť využila společně na zásobník a na haldu. Každá z těchto dvou struktur byla vytvářena od jednoho konce volného úseku paměti a během výpočtu rostla směrem

k opačnému konci. Výpočet probíhal korektně, pokud se vrchol zásobníku a vrchol haldy nestřetly. Výhodou tohoto uspořádání je, že jednotným způsobem poskytuje maximální dostupnou paměť jak programům s velkými nároky na zásobník a malými na haldu, tak i programům s malým zásobníkem a velkou haldou, ale dokonce i programům, u nichž je během výpočtu v některém okamžiku velký zásobník a v jiném velká halda. Způsob adresování na osobním počítači typu PC však toto rozložení jednoduše neumožňuje. V Turbo Pascalu je proto zásobník umístěn v samostatném paměťovém segmentu velikosti maximálně 64 KB a haldě je věnována veškerá další zbývající volná paměť. Maximální velikost haldy je tedy samozřejmě také omezena, zpravidla je však paměť určená pro dynamicky alokované proměnné několikanásobně větší než paměť pro staticky alokované proměnné a pro zásobník. To je také jedna z vedlejších výhod použití dynamicky alokovaných proměnných v Turbo Pascalu.

Hlavní přednost dynamických proměnných však spočívá v něčem jiném - právě v jejich dynamičnosti. Umožňují nám vytvářet **dynamické datové struktury**, jejichž velikost není známa předem v době psaní programu či v době překladu, ale závisí až na konkrétních vstupních datech zadaných při spuštění přeloženého programu. Práci s dynamickými datovými strukturami bude věnována největší část této učebnice (kap. 2 a 3).

1.2 Dynamicky alokované proměnné v Pascalu

Obecné principy práce s dynamicky alokovanými proměnnými uvedené v předchozí kapitole nyní přeneseme do syntaxe jazyka Turbo Pascal a ukážeme si, jak se s dynamickými proměnnými pracuje při psaní programu. Již víme, že na jednotlivé dynamicky alokované proměnné se odkazujeme jejich paměťovými adresami. Abychom si však tyto adresy mohli někde v programu ukládat, potřebujeme mít k dispozici proměnné vhodného typu. Proto byl zaveden datový typ „adresa v paměti“, který se běžně nazývá **ukazatel**, neboť proměnná tohoto typu „ukazuje“ na jisté místo v paměti počítače. Ukazatel v Pascalu bývá obvykle spojen s datovým typem, na který je oprávněn ukazovat. Můžeme si tedy deklarovat například ukazatel na proměnnou typu *integer*, ukazatel na pole obsahující tisíc znaků indexované od 1 do 1000 apod. Podle normy jazyka Pascal jsou dokonce všechny ukazatele vázány na konkrétní typ dynamicky alokované proměnné. V kap. 1.3 se dočtete, že Turbo Pascal zná navíc i obecnější ukazatele, které nejsou spojeny s žádným odkazovaným datovým typem. Zatím ale zůstaneme u běžnějších ukazatelů spojených s typem a ukážeme si, jak se s nimi pracuje. Ačkoliv všechny proměnné typu ukazatel obsahují paměťovou adresu a ukládají se stejným způsobem, překladač jazyka dbá na typovou kontrolu a nedovolí nám dosadit do proměnné typu „ukazatel na *integer*“ třeba hodnotu proměnné typu „ukazatel na *real*“.

Chceme-li v programu pracovat s dynamicky alokovanou proměnnou typu *T* (kde *T* může být libovolný typ, ať standardní, nebo častěji v programu definovaný), musíme si nejprve deklarovat pomocnou proměnnou typu „ukazatel na *T*“. Příslušnou deklaraci můžeme zapsat takto (pomocnou proměnnou jsme zde pojmenovali *P*):

```
type  T = ..... ;                { libovolný typ }
      UkazatelNaT = ^T;          { typ „ukazatel na T“ }
var   P: UkazatelNaT;            { proměnné typu ukazatel }
```

Můžeme použít také zkrácený zápis ve tvaru

```
var   P: ^T;
```

jak je to obvyklé i u proměnných jiných typů.

Proměnná *P* sama je tedy „obyčejná“ proměnná hlavního programu nebo nějakého podprogramu a jako taková je alokována staticky nebo na zásobníku společně se všemi ostatními proměnnými deklarovanými v sekci *var*.

Žádost o dynamické **vytvoření proměnné** typu *T* má podobu volání standardní procedury **new.** Tato procedura má jeden parametr typu ukazatel předávaný odkazem. Parametr slouží pouze jako výstupní, po provedení procedury *new* v něm volající program obdrží adresu přiděleného paměťového místa. Při volání procedury *new* nemusíme (a ani nemůžeme) explicitně udávat, o jak velký úsek dynamicky přidělované paměti žádáme. Tuto velikost si odvodí překladač sám na základě toho, jaký ukazatel stojí na místě parametru. V našem případě tedy zapíšeme v programu jednoduchý příkaz

```
new(P)
```

Překladač ví, že proměnná *P* je typu „ukazatel na *T*" a zná také velikost paměti potřebné pro proměnnou typu *T*. Volání *new(P)* proto správně přeloží na žádost systému o přidělení odpovídajícího úseku paměti v prostoru na haldě.

Po úspěšném vykonání příkazu *new(P)* již můžeme s dynamicky vytvořenou proměnnou typu *T* normálně pracovat. Pracujeme s ní úplně stejně jako s každou jinou proměnnou typu *T* - můžeme do ní dosazovat hodnotu přiřazovacím příkazem nebo čtením ze vstupu, můžeme používat její hodnotu ve výrazech nebo ji vypisovat na výstup příkazem *write*. Pokud je typ *T* strukturovaný, přistupujeme k jeho složkám obvyklým způsobem, tzn. indexováním v případě pole a pomocí tečkové notace (příp. příkazem vnoření *with*) v případě záznamu. Jedinou odlišností od běžných proměnných je skutečnost, že dynamicky vytvořená proměnná nemá jméno ve tvaru identifikátoru, jak jsme zvyklí. Místo toho se na ni **odkazujeme pomocí její paměťové adresy**, kterou máme uloženu v příslušné proměnné typu ukazatel. Obsahuje-li ukazatel *P* adresu dynamicky alokované proměnné, potom zápis *P^* představuje tuto proměnnou typu *T*. Znamená tedy „to, kam právě ukazuje *P*".

Příklady:

1. Je-li *P* typu ^*integer*, po provedení *new(P)* můžeme psát třeba

```
P^:=111; write(P^);
```

2. Je-li *P* typu ^*T*, kde *T=array[1..100] of char*, můžeme po *new(P)* psát

```
P^[1]:='A'; P^[2]:=P^[1];
```

3. Je-li *P* typu ^*T*, kde *T=record A,B: integer end*, můžeme po *new(P)* psát

```
read(P^.A); P^.B:=2; write(P^.A + P^.B);
```

Připomeňme si ještě, že bezprostředně po provedení příkazu *new(P)* není definována hodnota proměnné *P^*. Tato proměnná je pouze „vytvořena", je jí přiděleno místo v paměti, není však nijak inicializována. Pokud bychom se pokusili třeba vypsat její hodnotu dříve, než do ní něco dosadíme, můžeme dostat zcela libovolný náhodný výsledek (podle toho, co zrovna zůstalo v příslušném místě paměti z dřívějších výpočtů).

Mezi proměnnými typu ukazatel, které jsou kvalifikovány pro stejný datový typ, je možné provádět **dosazení.** Pokud *P* a *Q* jsou dva ukazatelé téhož typu a pokud *Q* právě ukazuje na nějakou dynamicky alokovanou proměnnou, pak příkaz *P:=Q* způsobí, že na tutéž dynamickou proměnnou začne ukazovat také *P*. Neprovede se žádná nová dynamická alokace, nýbrž pouze přesměrování ukazatele *P*. Proměnná typu ukazatel tedy může nabýt hodnoty dvěma způsoby: buď provedením procedury *new* (pak ukazuje na nově alokovanou proměnnou) nebo dosazením (v tom případě začne ukazovat na nějakou již dříve alokovanou proměnou). V obou případech se samozřejmě ztratí původní hodnota proměnné *P*. Musíme proto dát pozor, abychom zcela nepřišli o přístup k nějaké dynamicky alokované proměnné. Jestliže na nějakou dynamickou proměnnou ukazuje pouze ukazatel *P* a my do *P* dosadíme jinou hodnotu, pak se proměnná, na níž ukazatel *P* dosud ukazoval, stane nedostupnou. Nemáme již žádný prostředek, jak se dostat k její hodnotě, a navíc tato nedostupná proměnná zůstane až do konce výpočtu umístěna v paměti, kde bude zbytečně zabírat místo.

Při dosazování rozlišujte mezi příkazy *P := Q* a *P^ := Q^*. První z nich znamená přesměrování ukazatele *P* tam, kam právě ukazuje *Q*. Před jeho provedením by ukazatel *P* buď neměl ukazovat na

žádnou dynamickou proměnnou, nebo jen na takovou, na niž máme v programu ještě nějaký jiný odkaz, takže ztráta dosavadní hodnoty uložené v proměnné *P* nezpůsobí nedostupnost nějaké dynamicky alokované proměnné. Naproti tomu příkaz *P^ := Q^* představuje zkopírování hodnoty mezi dvěma dynamicky alokovanými proměnnými. Před jeho provedením musí ukazatelé *P* a *Q* ukazovat každý na jednu dynamickou proměnnou a proměnná *Q^* by měla mít již definovanou hodnotu. Tato hodnota se uvedeným příkazem zkopíruje do proměnné *P^* a přepíše její dosavadní hodnotu. Obsah ukazatelů *P, Q* se však nezmění.

Hodnoty proměnných typu ukazatel můžeme také **porovnávat** pomocí relačních operátorů. Podobně jako v případě dosazování lze mezi sebou porovnávat pouze dvojici ukazatelů téhož typu. Mezi ukazateli je povoleno použít porovnání na rovnost (binární relační operátor =) a nerovnost (operátor <>), ostatní relační operátory jazyka Pascal (jako třeba <, <=) nemají v případě ukazatelů smysl. Podmínka *P = Q* je splněna právě tehdy, ukazují-li ukazatelé *P, Q* na tutéž dynamicky alokovanou proměnnou (tzn. proměnné *P, Q* mají stejnou hodnotu). Porovnání hodnot dvou ukazatelů uplatníme například v podmíněném příkazu

```
if P = Q then ...
```

nebo v podmínce cyklu

```
while P <> Q do ...
```

K **uvolnění** již nepotřebné dynamicky alokované proměnné používáme standardní proceduru **dispose.** Voláme ji s jedním parametrem typu ukazatel, tedy například *dispose(P)*. Překladač přeloží toto volání na pokyn systému, že paměťová oblast na haldě umístěná od adresy *P* je volná k dalšímu použití. Velikost uvolňované paměti určí překladač podle typu ukazatele *P*. Proměnnou, kterou jsme jednou uvolnili, nesmíme v programu nadále používat. Místo v paměti, kde byla umístěna, může být totiž vzápětí využito k uložení jiné proměnné po provedení dalšího příkazu *new*. Musíme si na to dávat pozor zejména v případě, ukazuje-li na tutéž dynamicky alokovanou proměnnou více ukazatelů. Jsou-li například proměnné *P, Q* stejného typu ukazatel, je korektní posloupnost příkazů

```
new(P);          { nová dynamická alokace }
Q:=P;            { přesměrování Q na tutéž proměnnou }
dispose{Q};      { uvolnění této proměnné }
```

Bezprostředně poté již v programu nesmíme pracovat s proměnnou *P^*, ačkoliv jsme právě provedli její alokaci *new(P)*, ale neprovedli jsme její uvolnění příkazem *dispose(P)*. Proměnná však byla zrušena příkazem *dispose(Q)*, když ukazatel *Q* ukazoval na stejné místo v paměti jako *P*. Později budeme moci v programu opět pracovat s proměnnou *P^* za předpokladu, že nově nadefinujeme hodnotu ukazatele *P* dosazením nebo provedením *new(P)* - tedy až *P* bude opět ukazovat na řádně přidělenou dynamicky alokovanou proměnou.

Každá proměnná typu ukazatel může nabývat speciální hodnoty **nil.** Konstanta *nil* patří mezi klíčová slova jazyka Pascal a představuje hodnotu „neukazuje nikam“. Vnitřně je reprezentována takovou adresou, jaká se na haldě nikdy nemůže vyskytnout (obvykle adresou 0). Hodnotu *nil* dosadíme do proměnné příkazem tvaru

```
P := nil;
```

V programu můžeme také testovat, zda nabyl některý ukazatel této speciální hodnoty, tedy například v podmínce

```
if P = nil then ...
```

nebo v cyklu

```
while P <> nil do ...
```

Dosazováním hodnoty *nil* do každého ukazatele, který právě nikam neukazuje, se může programátor částečně chránit proti některým vlastním chybám. Má-li totiž proměnná *P* hodnotu *nil*, jakýkoli pokus

o práci s dynamickou proměnnou *P^* způsobí běhovou chybu programu a upozorní nás na závadu v programu. Pokud však ukazatel *P* neukazuje na žádnou dynamicky alokovanou proměnnou a nedosadíme do něj konstantu *nil*, bude jistě obsahovat nějakou náhodnou hodnotu z dřívějška. Není-li to náhodou zrovna vnitřní kód konstanty *nil*, chybný pokus o práci s proměnnou *P^* v tomto případě nezpůsobí běhovou chybu, ale může vést k velmi „divokému" výpočtu. Obsah proměnné *P* je chápán jako adresa dynamicky alokované proměnné a s obsahem paměti na této adrese je provedena požadovaná operace. Důsledkem toho může být třeba přepsání hodnoty nějaké úplně jiné proměnné. Pečlivý programátor se ovšem těmto chybám může vyhnout i bez použití konstanty *nil*. Jakmile však začneme z jednotlivých dynamicky alokovaných proměnných vytvářet dynamické datové struktury, bez konstanty *nil* se neobejdeme (viz kap. 2 a 3).

Pro uvolňování již nepotřebných dynamických proměnných můžeme v Pascalu místo procedury *dispose* použít ještě jeden méně obvyklý mechanismus představovaný dvojicí standardních procedur **mark** a **release**. Tento mechanismus nám umožňuje pracovat s haldou podobným způsobem jako se zásobníkem. Pro správnou činnost však nesmíme kombinovat v jednom programu příkazy *release* s voláním procedury *dispose*, která naopak vytváří v haldě „díry". V každém programu bychom tedy měli používat buď pouze proceduru *dispose*, nebo pouze dvojici procedur *mark* - *release*. Proceduru *mark* voláme s jedním parametrem typu ukazatel (libovolného druhu). Tento parametr je předáván odkazem a je pouze výstupní. Provedením příkazu *mark(P)* se do proměnné *P* zaznamená adresa, kam až je halda zaplněna. Další příkazy *new* vedou k alokaci proměnných na haldě nad tuto zaznamenanou hladinu. Následné volání *release(P)* způsobí, že je najednou uvolněna veškerá paměť na haldě nad hladinou zaznamenanou v ukazateli *P*. To znamená, že jsou uvolněny všechny dynamicky alokované proměnné, které byly vytvořeny od posledního definování hodnoty proměnné *P*. Přitom pro tento účel bývá hodnota *P* stanovena typicky voláním procedury *mark(P)*, ale můžeme ji nastavit také provedením *new(P)*, případně i přímým dosazením do *P*.

1.3 Rozšíření Turbo Pascalu

Implementace jazyka Turbo Pascal přinesla do standardního Pascalu celou řadu doplnění a rozšíření. Některé změny se týkají i možností alokace paměti pro proměnné.

Do jazyka byl zaveden standardní datový typ **pointer** představující obecný ukazatel, který není vázán na žádny typ, na nějž by byl oprávněn ukazovat. Proměnné typu *pointer* mohou ukazovat kamkoliv do paměti, na proměnnou libovolného typu. Nesmíme je ale používat jako skutečné parametry při volání procedur *new* nebo *dispose*, neboť překladač by nedokázal vygenerovat správnou žádost na systém. Nevěděl by totiž, o kolik bytů dynamické paměti žádáme, resp. kolik bytů paměti uvolňujeme. Ve spojení s proměnnými typu pointer se však nabízí nový mechanismus přístupu do haldy, a to procedury **GetMem** a **FreeMem**. Procedura *GetMem* vyvolává žádost o dynamické přidělení paměti dané velikosti. Její první parametr typu *pointer* je předáván odkazem, je výstupní a slouží stejně jako parametr procedury *new* k převzetí adresy, kde se nachází přidělená paměťová oblast. Druhý parametr je celočíselný, je naopak vstupní a určuje velikost požadované paměti v bytech. Místo uvedení typu nově vytvářené dynamické proměnné tedy přímo specifikujeme její velikost. Procedura *FreeMem* se analogicky používá k uvolňování již nepotřebné paměti. Její první parametr typu *pointer* opět určuje adresu uvolňované proměnné a druhý celočíselný parametr udává její velikost v bytech.

Jestliže v programu použijeme proceduru *GetMem* k dynamickému získání paměťového prostoru, musíme se ještě postarat o to, abychom s takto přidělenou dynamickou proměnnou mohli vůbec nějak rozumně pracovat. Je-li totiž proměnná *P* typu *pointer*, pak dynamická proměnná *P^* nemá definován typ a není proto možné manipulovat s ní pomocí běžných aritmetických operátorů a procedur jazyka Pascal. Požadovaný typ jí však můžeme dočasně přidělit pomocí tzv. **přetypování**. Jméno obecného ukazatele *P* uzavřeme do závorek a před závorku napíšeme jméno příslušného ukazatelového typu, například *UkazatelNaT(P)*, kde *UkazatelNaT = ^T*. Tím sdělíme překladači, že v tomto místě programu nemá chápat proměnnou *P* jako obecný *pointer*, ale jako ukazatel kvalifikovaný pro

odkazování na proměnné typu *T*. Zápis *UkazatelNaT(P)^* potom představuje dynamickou proměnnou získanou příkazem *GetMem(P,n),* s níž nyní můžeme pracovat jako s proměnnou typu *T*. Druhou, o něco pohodlnější cestou je nechat dosadit adresu dynamicky přidělené proměnné přímo do typovaného ukazatele, např. voláním *GetMem(P,n)* pro proměnnou *P* typu *UkazatelNaT*. V programu pak můžeme normálně pracovat s proměnnou *P^* jako s proměnou typu *T*.

Užitečnou ukázkou využití právě popsaného mechanismu je realizace **dynamického pole** v Turbo Pacalu. Jazyk Pascal (podobně jako řada dalších programovacích jazyků) normálně povoluje deklarovat pouze pole pevné velikosti. Velikost pole musí být známa již v době překladu, aby mohl překladač generovat efektivní binární kód. Všechny proměnné hlavního programu nebo procedury či funkce se umisťují do souvislého bloku za sebou a je-li pevně dána velikost každé z nich, zná překladač také relativní adresu (vzdálenost) každé z proměnných od začátku tohoto bloku. Povinnost používat v programu pouze pole předem dané velikosti se týká nejen statických proměnných hlavního programu a proměnných procedur a funkcí alokovaných na zásobníku, ale také dynamicky alokovaných polí získaných procedurou *new*. Neznáme-li předem při psaní programu přesný počet údajů uložených v poli, může nám být uvedené omezení nepříjemné. Znamená pro nás obvykle nutnost deklarovat pole větší (velikost určíme podle horního odhadu počtu ukládaných dat) a pak z něj využívat v programu jenom část podle skutečného rozsahu dat. Jinou možností řešení je nahradit jednorozměrné pole dynamicky alokovaným lineárním spojovým seznamem, čímž ovšem ztrácíme výhodu přímého přístupu k prvkům pomocí indexů (viz kap 2). Konečně další možností je vytvořit dynamicky alokované pole potřebné velikosti až za běhu programu pomocí procedury *GetMem*. Můžeme postupovat třeba takto:

```
program DynamickePole;
{realizace dynamického pole na haldě -
 přiklad: jednorozměrné pole celých čísel}

type  Pole = array[1..maxint] of integer;
      Uk = ^Pole;

var   P : Uk;
      N : integer;   {potřebný počet prvků pole}
      V : integer;   {potřebná velikost alokace paměti}
      I : integer;

begin
N := 200;      {libovolná hodnota, kterou získáme na základě
                vstupních dat a předchozího výpočtu}
V := N * SizeOf(integer);
GetMem(P,V);       {vytvoření pole o N položkách}
for I:=1 to N do
  P^[I] := I;      {libovolná práce s prvky pole P^}
for I:=1 to N do
  write(P^[I]:4);
writeln;
FreeMem(P,V);      {uvolnění paměti - zrušení pole}
end.
```

Pro práci s dynamicky alokovanými proměnnými nám jazyk Turbo Pascal nabízí ještě dvě pomocné funkce nazvané **MemAvail** a **MaxAvail**. Obě jsou bez parametrů a vracejí celočíselnou hodnotu. Funkce *MemAvail* udává v každém okamžiku velikost volné paměti na haldě, funkce *MaxAvail* určuje velikost největšího souvislého bloku volné paměti na haldě. Říká nám tedy, jakou největší proměnnou

lze v tuto chvíli dynamicky přidělit pomocí procedury *new* nebo *GetMem*. Obě funkce udávají svůj výsledek v bytech.

Další rozšíření jazyka Turbo Pascal se již netýkají přímo dynamicky alokovaných proměnných, ale mají vztah k alokaci proměnných vůbec. Z obecnějšího pohledu na přidělování paměti jsou zajímavé ještě **inicializované proměnné**, které se v Turbo Pascalu nazývají typované konstanty. Název typované konstanty je však poněkud zavádějící, neboť ve skutečnosti to nejsou žádné konstanty, ale proměnné s přiřazenou počáteční hodnotou, kterou lze později během výpočtu měnit. Je proto také dosti matoucí, že se nedeklarují jako proměnné za klíčovým slovem *var*, nýbrž v úseku definic konstant *const*. Syntaxe deklarace vypadá následovně:

```
const X: T = H;
```

kde *X* je jméno deklarované inicializované proměnné, *T* je její typ a *H* je její hodnota na začátku výpočtu. Typ *T* může být jak jednoduchý, tak i strukturovaný, tedy například pole nebo záznam. V každém případě musí být také hodnota *H* odpovídajícího typu *T*. Složky pole zapisujeme do závorek a oddělujeme čárkou, položky záznamu uvádíme rovněž v závorce vždy se jménem příslušné položky a oddělené středníkem. Nejlépe si deklaraci inicializovaných proměnných předvedeme na několika jednoduchých příkladech:

```
const N: integer = 100;

type  Vektor = array[1..10] of integer;
const V: Vektor = (4,7,-9,-2,1,0,0,0,4,1);

type  Bod = record X, Y: real end;
const B: Bod = (X: 3.3; Y: 4.1);
```

Ještě jednou si zopakujme, že po zahájení výpočtu můžeme s inicializovanou proměnnou pracovat stejně jako s každou jinou proměnnou - nejen že můžeme používat její hodnotu ve výrazech, ale můžeme ji i libovolně měnit.

Inicializované proměnné deklarované v hlavním programu jsou jednoduše alokovány staticky v datovém segmentu společně s ostatními proměnnými hlavního programu. Velice zvláštní postavení z hlediska paměťové alokace však mají inicializované proměnné deklarované lokálně v procedurách a funkcích. Při ukončení výpočtu podprogramu si tyto proměnné uchovávají svoji poslední hodnotu do příštího volání téhož podprogramu. To znamená, že až bude v programu tatáž procedura znovu zavolána, nebude takováto proměnná opětovně inicializována hodnotou uvedenou v její deklaraci *const*, ale nalezneme v ni hodnotu, která tam zůstala od předchozího volání. Inicializace proměnné hodnotou zapsanou v její deklaraci *const* se tedy provede pouze jednou při spuštění programu. (Takovéto proměnné jsou programátoři v jazyce C zvyklí označovat paměťovou třídou „static“.)

Vše si můžeme ukázat na jednoduchém příkladu:

```
program Inic_Prom;
  procedure S;
  const J: byte = 1;
  begin J:=J+1; writeln(J) end;
begin S; S; S end.
```

Při každém zavolání procedury *S* je hodnota proměnné *J* zvýšena o 1 a svoji hodnotu si uchová do příštího volání, takže program vypíše na výstup postupně čísla 2, 3 a 4.

Inicializované proměnné deklarované jako lokální v podprogramu nemohou být alokovány na zásobníku společně s ostatními lokálními proměnnými. Po ukončení výpočtu podprogramu je totiž

jeho aktivační záznam ze zásobníku odstraněn a tím bychom ztratili i poslední hodnotu inicializované proměnné. Tyto proměnné musí proto být alokovány staticky. Překladač ale zachází s jejich identifikátory jako s lokálními symboly, takže nejsou dostupné mimo podprogram, v němž byly deklarovány.

Pro úplnost zbývá dodat, jak je to s **počátečními hodnotami** obyčejných proměnných. Norma jazyka Pascal o nich nic nepředpokládá a také Turbo Pascal až do verze 6 včetně je nijak nedefinoval. Je proto dobré zvyknout si, že počáteční hodnoty proměnných nejsou definovány a že dříve, než proměnnou použijeme k výpočtu, musíme do ní nejprve nějakou hodnotu sami dosadit (přiřazovacím příkazem nebo ze vstupu voláním procedury *read*). Pokud bychom pracovali v programu s nedefinovanou hodnotou proměnné, nedojde při výpočtu k žádné běhové chybě. Za její hodnotu se ale vezme to, co náhodou zbylo z předchozích výpočtů v tom místě paměti, kam bylo tato proměnná alokována. Žádné rozumné výsledky tedy v takovém případě nemůžeme očekávat a dokonce při každém spuštění téhož programu můžeme obdržet jiné výsledky podle toho, co zrovna kde v paměti zůstalo za hodnoty. Až Turbo Pascal 7.0 zavedl automatickou inicializaci všech staticky alokovaných proměnných hlavního programu na hodnotu 0, proměnných alokovaných na zásobníku a na haldě se však inicializace na nulu netýká.

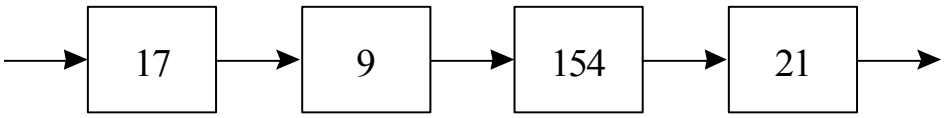
2. Lineární spojové seznamy

2.1 Operace s lineárními spojovými seznamy

Dynamicky alokované proměnné se v programech málokdy objevují izolovaně, většinou je používáme k vytváření **dynamických datových struktur**. Dynamická struktura je soustava několika dynamických proměnných, mezi nimiž jsou vybudovány vzájemné odkazy pomocí ukazatelů. Dynamicky alokované proměnné vázané do dynamických datových struktur bývají zpravidla typu záznam. Některé položky těchto záznamů použijeme k uložení těch dat a informací, které chceme ve struktuře zobrazit, další jedna nebo více položek každého záznamu je typu ukazatel a slouží k propojení jednotlivých proměnných do struktury. Počet propojených záznamů, tzn. velikost výsledné dynamické datové struktury, není předem dán a může záviset na vstupních datech a na průběhu předchozího výpočtu programu. Celá struktura je vytvářena dynamicky až za běhu programu a může se během výpočtu měnit. Této dynamičnosti můžeme při psaní programu výhodně využívat, ale musíme za ni také něco zaplatit. Oproti klasickým datovým strukturám je práce s dynamickými strukturami o něco pomalejší (každé přidělení či uvolnění dynamicky alokované proměnné vyžaduje vykonat ne zcela triviální akci) a bývá i paměťově náročnější (část přiděleného paměťového prostoru neslouží k uložení dat, ale k umístění ukazatelů udržujících pohromadě celou strukturu).

Dynamická datová struktura může být tvořena buď záznamy téhož typu (viz kap. 2 a 3.1), nebo se v ní mohou kombinovat záznamy více různých typů (viz kap. 3.2). Nejjednodušším případem dynamické struktury je **lineární spojový seznam**, který je tvořen jednou řadou po sobě jdoucích uzlů téhož typu. Každý prvek lineárního spojového seznamu je záznam obsahující jednak vlastní uložená data libovolného typu, jednak jeden ukazatel na další prvek seznamu:

```
type  Uk = ^Uzel;
      Uzel = record
                 Hodnota: T;      {uložená data}
                 Dalsi: Uk        {propojení uzlů}
             end;
```



Všechny prvky lineárního spojového seznamu jsou alokovány dynamicky na haldě pomocí procedury *new*. Na první z nich však musí ukazovat nějaký ukazatel statický (deklarovaný v programu pomocí *var*), abychom měli v programu ke spojovému seznamu vůbec nějaký přístup. Tento ukazatel nám slouží při práci se seznamem jako vstupní bod, k dalším uzlům seznamu se již dostáváme postupným průchodem pomocí ukazatelů, jimiž je seznam propojen (položka *Dalsi*). Na rozdíl od pole tedy nemáme k dispozici přímý přístup k jednotlivým prvkům pomocí indexu. Protože délka spojového seznamu se může během výpočtu měnit, je velmi důležité zajistit to, abychom byli schopni kdykoliv rozpoznat poslední prvek seznamu. Jeho ukazatel musí mít proto hodnotu *nil*.

Ukážeme si, jak se provádějí základní operace s lineárními spojovými seznamy. Pro jednoduchost budeme do jednotlivých uzlů seznamu ukládat pouze hodnoty typu *integer*. První potřebnou akcí je **vytvoření nového seznamu**. Následující procedury vytvářejí lineární spojový seznam, v němž budou uložena všechna celá čísla zadaná na vstupu. Pokud nám nezáleží na pořadí čísel v seznamu, je snadnější stavět seznam odzadu a vložit do něj tedy čísla v opačném pořadí, než v jakém byla zadána na vstupu (procedura *Vytvor1*). Jestliže ale požadujeme, aby v seznamu zůstalo zachováno pořadí

čísel ze vstupu, musíme vytvářet seznam odpředu a nové uzly zapojovat na jeho konec (procedura *Vytvor2*). V tomto případě se vyplatí udržovat si průběžně pomocný ukazatel na dosud poslední prvek vytvářeného seznamu.

```
procedure Vytvor1(var P: Uk);
{ve výstupním parametru P procedura vrací odkaz na začátek
 vytvořeného lineárního spojového seznamu}
var Z: Uk;
begin
P := nil;                {prázdný seznam}
while not eof do         {dokud není konec vstupních dat}
  begin
  new(Z);
  read(Z^.Hodnota);
  Z^.Dalsi := P;
  P := Z
  end
end; {procedure Vytvor1}
```

```
procedure Vytvor2(var P: Uk);
{ve výstupním parametru P procedura vrací odkaz na začátek
 vytvořeného lineárního spojového seznamu}
var Z, K: Uk;
begin
if eof then
  P := nil               {prázdný seznam}
else
  begin
  new(P);                {první prvek seznamu}
  read(P^.Hodnota);
  K := P;                {prozatím je to i konec}
  while not eof do       {dokud není konec vstupních dat}
    begin
    new(Z);
    read(Z^.Hodnota);
    K^.Dalsi := Z;
    K := Z
    end;
  K^.Dalsi := nil
  end
end; {procedure Vytvor2}
```

Máme-li seznam vytvořen, potřebujeme se naučit **procházet** jím a provádět v jeho uzlech nějaké operace s uloženými daty. Pro jednoduchost budeme v následujícím příkladě pouze vypisovat hodnoty uložené v uzlech seznamu, volání procedury *write* bychom samozřejmě mohli nahradit nějakou jinou akcí. Důležitým momentem v následující proceduře *Pruchod* je rozpoznání konce seznamu pomocí konstanty *nil*.

```
procedure Pruchod(P: Uk);
begin
while P <> nil do
```

```
  begin
  writeln(P^.Hodnota);
  P := P^.Dalsi
  end;
end; {procedure Pruchod}
```

Častou operací je také **vyhledání hodnoty** v seznamu. Následující procedura *Hledej* dostává ve vstupních parametrech odkaz na začátek zkoumaného seznamu a hledanou hodnotu, ve výstupním parametru vrací ukazatel na první prvek seznamu obsahující zadanou hodnotu. Pokud hledaná hodnota není v seznamu vůbec obsažena, bude mít výstupní parametr hodnotu *nil*.

```
procedure Hledej(P: Uk; H: integer; var Z: Uk);
{v seznamu P hledá hodnotu H, výsledek vrací v Z}
var Hledat: boolean;
begin
Hledat := P <> nil;
while Hledat do
  if P^.Hodnota = H then Hledat := false        {nalezeno}
  else
    begin
    P := P^.Dalsi;                              {další prvek}
    if P = nil then Hledat := false             {není tam}
    end;
Z := P;          {výsledek: Z^.Hodnota = H nebo Z = nil}
end; {procedure Hledej}
```

Vedle pouhého vyhledání prvku obsahujícího zadanou hodnotu můžeme také potřebovat prvek s touto hodnotou **ze seznamu vyřadit**. V takovém případě není vhodné použít výše uvedenou proceduru *Hledej*, neboť ta nám vrací odkaz na vybraný prvek, nikoliv však na jeho předchůdce. Při vyřazování takto nalezeného prvku ze seznamu bychom pak museli projít seznamem znovu od začátku, nalézt předchůdce vypouštěného prvku a jemu přepojit ukazatel *Dalsi*. Šikovnější je naprogramovat rovnou proceduru, která vše zvládne při jednom průchodu seznamem. Procedura *Vyrad* má stejnou strukturu parametrů jako procedura *Hledej*. Liší se pouze tím, že v případě úspěšného nalezení hodnoty *H* první prvek obsahující tuto hodnotu ze seznamu vyřadí a v parametru *Z* vrátí odkaz na něj. Povšimněte si ještě, že na rozdíl od procedury *Hledej* musí být první parametr *P* předáván odkazem. To proto, že první prvek s hodnotou *H* může stát na samém začátku seznamu a po jeho vyřazení by se tedy začátek seznamu změnil.

```
procedure Vyrad(var P: Uk; H: integer; var Z: Uk);
{v seznamu P hledá hodnotu H, první uzel s touto hodnotou
 vyřadí a odkaz na něj vrací v Z}
var Hledat: boolean;
    Q, R: Uk;
begin
if P = nil then Z := nil
else if P^.Hodnota = H then
  begin Z := P; P := P^.Dalsi; Z^.Dalsi := nil end
else
  begin
  Q := P^.Dalsi;        {aktuální prvek}
```

```
  R := P;                {předchůdce}
  Hledat := Q <> nil;
  while Hledat do
    if Q^.Hodnota = H then
      begin
      Z := Q;
      R^.Dalsi := Q^.Dalsi;
      Q^.Dalsi := nil;
      Hledat := false
      end
    else
      begin
      R := Q;
      Q := Q^.Dalsi;
      if Q = nil then
        begin
        Hledat := false;
        Z := nil
        end
      end
  end
end; {procedure Vyrad}
```

Jako ukázku práce s lineárními spojovými seznamy si naprogramujeme ještě jednu velmi podobnou úlohu. Úkolem nyní bude v daném lineárním spojovém seznamu vyhledat všechny prvky obsahující zadanou hodnotu a vyřadit je ze seznamu. V tomto případě budeme chtít vyřazené prvky rovnou zrušit.

```
procedure VyradVsechny(var P: Uk; H: integer);
{ze seznamu P vypustí všechny uzly s hodnotou H}
var Hledat: boolean;
    Q, R: Uk;
begin
Hledat := P <> nil;
while Hledat do    {vypustit všechny H na začátku seznamu}
  if P^.Hodnota = H then
    begin
    R := P;
    P := P^.Dalsi;
    dispose(R);
    if P = nil then Hledat := false
    end
  else
    Hledat := false;
if P <> nil then
  begin
  Q := P^.Dalsi;         {aktuální prvek}
  R := P;                {předchůdce}
  Hledat := Q <> nil;
  while Hledat do
    if Q^.Hodnota = H then
      begin
      R^.Dalsi := Q^.Dalsi;
```

```
      dispose(Q);
      Q := R^.Dalsi;
      end
    else
      begin
      R := Q;
      Q := Q^.Dalsi;
      if Q = nil then Hledat := false
      end
  end
end; {procedure VyradVsechny}
```

Velmi snadnou úlohou je **spojit** dva lineární spojové seznamy za sebe. Máme-li dány ukazatele na začátky obou seznamů, musíme nejprve projít pomocným ukazatelem na konec prvního seznamu a za následníka posledního prvku prvního seznamu prohlásit první prvek druhého seznamu.

```
procedure Spoj(var P, Q: Uk);
{na konec seznamu P připojí seznam Q}
var R: Uk;     {poslední prvek seznamu P}
begin
if P = nil then
  P := Q
else
  begin
  R := P;
  while R^.Dalsi <> nil do R := R^.Dalsi;
  R^.Dalsi := Q
  end;
Q := nil
end; {procedure Spoj}
```

Někdy můžeme naopak chtít **rozdělit** prvky jednoho spojového seznamu do více seznamů. Kritériem pro rozdělování bude hodnota uložená v prvcích seznamu. Jako příklad zvolíme úlohu rozdělit prvky daného lineárního spojového seznamu do dvou seznamů podle znaménka jejich hodnoty - do jednoho seznamu zařadíme prvky se zápornou hodnotou, do druhého prvky s hodnotou nezápornou. Pro jednoduchost budeme předpokládat, že na pořadí prvků ve výsledných seznamech nezáleží. V tom případě bude pro nás snadnější vytvářet oba seznamy odzadu (srovnej s procedurami *Vytvor1*, *Vytvor2*).

```
procedure Rozdel(P: Uk; var Zapor, Nezapor: Uk);
{prvky seznamu P rozdělí na záporné a nezáporné
do dvou lineárních spojových seznamů}
var Q: Uk;     {zařazovaný prvek}
begin
Zapor := nil;
Nezapor := nil;
while P <> nil do
  begin
  Q := P;
  P := P^.Dalsi;
  if Q^.Hodnota < 0 then
```

```
    begin Q^.Dalsi := Zapor; Zapor := Q end
  else
    begin Q^.Dalsi := Nezapor; Nezapor := Q end
  end
end; {procedure Rozdel}
```

Poněkud speciálnější, ale občas užitečnou operací je **obrácení pořadí prvků** v lineárním spojovém seznamu. Například ze seznamu obsahujícího po řadě prvky s čísly 10, 20, 30, 40, 50 chceme získat seznam tvořený stejnými prvky, ale v pořadí 50, 40, 30, 20, 10. Postup obracení si můžeme nejlépe představit tak, že původní seznam prvek po prvku rozebíráme a z jednotlivých uzlů sestavujeme odzadu výsledný obrácený seznam. V každém kroku tedy odpojíme jeden prvek ze začátku starého seznamu a přepojíme ho na začátek nového seznamu. Jedná se vlastně o naprosto stejný postup, jaký jsme použili v předcházející proceduře *Rozdel*, jenom se neprovádí žádné rozdělování a všechny odpojené prvky se zařazují do téhož výsledného seznamu.

```
procedure Obrat(var P: Uk);
{obrátí pořadí prvků v seznamu}
var N: Uk;      {nový seznam}
    R: Uk;      {přepojovaný prvek}
begin
N := nil;
while P <> nil do
  begin
  R := P;               {první prvek ze starého seznamu}
  P := P^.Dalsi;        {posunout začátek starého seznamu}
  R^.Dalsi := N;        {zapojit na začátek nového seznamu}
  N := R                {změnit začátek nového seznamu}
  end;
P := N;                 {vrací začátek nového seznamu}
end; {procedure Obrat}
```

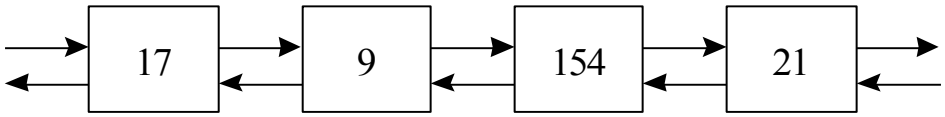
Po ukončení práce s lineárním spojovým seznamem bychom ho měli **zrušit** a uvolnit tak místo v paměti, které zabíral. Seznam musíme rušit postupně prvek po prvku, jak ukazuje procedura *Zrus*.

```
procedure Zrus(var P: Uk);
{zruší lineární spojový seznam prvek po prvku}
var Z: Uk;          {rušený prvek}
begin
while P <> nil do
  begin
  Z := P;
  P := P^.Dalsi;
  dispose(Z)
  end
end; {procedure Zrus}
```

2.2 Druhy lineárních spojových seznamů

Dosud jsme pracovali s lineárním spojovým seznamem v nejjednodušší možné podobě. Takový seznam bývá označován také jako jednosměrný, neboť se v něm můžeme pohybovat pomocí ukazatelů pouze jedním směrem. V některých případech se nám může hodit sestrojit lineární spojový seznam **obousměrný**. V něm obsahuje každý uzel dva ukazatele, a to na svého předchůdce a na následníka:

```
type  Uk = ^Uzel;
      Uzel = record
                 Hodnota: T;      {uložená data}
                 Pred, Nasl: Uk   {propojení uzlů}
             end;
```



Obousměrné seznamy nejen že umožňují průchod datovou strukturou oběma směry, ale usnadňují nám i provádění celé řady dalších operací. Jako příklad si můžeme uvést vypuštění jednoho prvku ze seznamu, známe-li ukazatel *P* ukazující na vypouštěný prvek. V případě jednosměrného seznamu musíme nejprve vyhledat předchůdce vypouštěného prvku, což může být dost pracné a znamená to projít pomocným ukazatelem *Q* celým seznamem od začátku až k prvku *P*^ (předpokládejme nyní pro jednoduchost, že *P*^ není prvním prvkem seznamu):

```
Q := Zacatek;
while Q^.Dalsi <> P do Q := Q^.Dalsi;
Q^.Dalsi := P^.Dalsi;
dispose(P);
```

V případě obousměrného seznamu není třeba procházet celým seznamem, stačí jen vhodně přepojit dva ukazatele. Pro zjednodušení budeme opět předpokládat, že *P*^ není krajním prvkem seznamu, tento speciální případ by bylo třeba ošetřit zvlášť:

```
P^.Pred^.Nasl := P^.Nasl;
P^.Nasl^.Pred := P^.Pred;
dispose(P);
```

Pro úplnost dodejme, že i v případě jednosměrného spojového seznamu je možné vypouštět zadaný prvek *P*^ bez procházení celým seznamem, ovšem za cenu kopírování veškerých dat uložených v uzlu seznamu (a tato data mohou být značně rozsáhlá a kopírování tudíž zdlouhavé). K vyřešení úkolu použijeme malý trik. Ze seznamu nevypustíme prvek *P*^, ale jeho následníka, a přitom do prvku *P*^ vložíme data uložená původně ve vypuštěném následníkovi prvku *P*^. Tento trik je samozřejmě možné použít jen tehdy, pokud prvek *P*^ nějakého následníka má, tzn. pokud není posledním prvkem seznamu.

```
Q := P^.Dalsi;  {Q je následníkem P}
P^ := Q^;       {zkopíruje se celý obsah - data i ukazatel}
dispose(Q);     {místo P^ zrušíme Q^}
```

Použití obousměrného seznamu má oproti jednosměrnému ovšem také určité nevýhody, takže v každé konkrétní úloze musíme zvažovat, která datová struktura bude pro její řešení nejvhodnější. Jednou nevýhodou je větší paměťová náročnost, v každém uzlu seznamu je třeba vyhradit prostor pro druhý ukazatel. Pokud bychom tedy pracovali v Turbo Pascalu například s obousměrným seznamem znaků, každý uzel seznamu bude zabírat v paměti devět bytů, z čehož pouhý jeden byte slouží k uložení vlastní informace (jednoho znaku), zatímco zbývajících osm bytů potřebujeme na uschování dvou ukazatelů (tedy na „režii“ seznamu). Druhou nevýhodou je pak zpomalení některých operací se seznamy - při jakékoliv změně ve struktuře je třeba místo jednoho ukazatele přepojovat dva.

V některých situacích můžeme výhodně využít ještě další druhy lineárních spojových seznamů. Pro pohodlné přidávání a ubírání prvků seznamu někdy používáme tzv. **seznam s hlavou.** Hlava je jeden zvláštní prvek na samém začátku seznamu, v němž nejsou uložena žádná data (nebo jeho datovou položku můžeme využít nějakým jiným vhodným způsobem). Prázdný seznam potom není reprezentován ukazatelem na *nil* jako u obyčejných seznamů, nýbrž ukazatelem na hlavu, jejíž položka *Dalsi* má hodnotu *nil*. A k čemu je to vlastně dobré? Přidáváme-li do lineárního spojového seznamu nový prvek vždy na jeho konec, musíme v případě obyčejného seznamu bez hlavy stále rozlišovat případy, kdy je seznam ještě prázdný a kdy již nějaké prvky obsahuje. V každém z těchto případů se vložení nového prvku provádí odlišně. V prvním případě se na nový prvek nasměruje ukazatel na začátek seznamu a také ukazatel na dosud poslední prvek seznamu, ve druhém případě se nový prvek připojí za dosud poslední prvek seznamu a stane se sám posledním prvkem. Seznam je pro úlohy tohoto typu dobré reprezentovat nejen ukazatelem na jeho začátek, ale také na jeho konec:

```
type  Seznam = record
                 Z: Uk;     {začátek seznamu}
                 K: Uk;     {poslední prvek seznamu}
               end;
```

Nový prázdný seznam *S* založíme příkazem `S.Z := nil`

```
procedure Pridej1(var S: Seznam; P: Uk);
{do seznamu S přidá prvek P}
begin
if S.Z = nil then       {S je dosud prázdný}
  begin S.Z := P; S.K := P; S.K^.Dalsi := nil end
else    {připojí P za dosud poslední prvek v S}
  begin S.K^.Dalsi := P; S.K := P; S.K^.Dalsi := nil end
end; {procedure Pridej1}
```

Naproti tomu u seznamů s hlavou se nové prvky přidávají mnohem jednodušeji, jednotně do prázdného i neprázdného seznamu. Nový prázdný seznam *S* s hlavou vytvoříme dvojicí příkazů

```
new(S.Z);       {hlava}
S.K := S.Z;     {hlava je poslední}
```

```
procedure Pridej2(var S: Seznam; P: Uk);
{do seznamu s hlavou S přidá prvek P}
begin
  S.K^.Dalsi := P; S.K := P; S.K^.Dalsi := nil
end; {procedure Pridej2}
```

Obdobně při vypouštění prvků ze seznamu musíme u obyčejných lineárních spojových seznamů rozlišovat, zda to náhodou nebyl v pořadí první (případně jediný) prvek seznamu a v tom případě provést odlišnou akci, zatímco vypuštění libovolného prvku ze seznamu s hlavou probíhá jednotným způsobem.

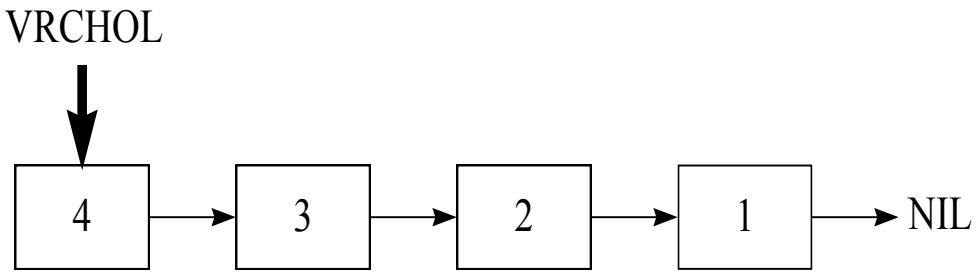
Dalším typem lineárních spojových seznamů jsou **seznamy cyklické**. U nich neukazuje poslední prvek seznamu na *nil*, ale místo toho je jeho ukazatel nasměrován na začátek seznamu, tedy na první prvek. Celý seznam je tak „stočen do kruhu“ a umožňuje nám navštívit postupně všechny prvky počínaje v libovolném z nich.

Jednotlivé modifikace seznamů je možné podle potřeby různě **kombinovat**, takže můžeme v programu pracovat třeba s obousměrným cyklickým seznamem s hlavou. Ten umožňuje pomocí ukazatelů uložených v hlavě snadný přístup k oběma koncům seznamu, průchod seznamem oběma směry, nabízí jednotný způsob vkládání nových prvků na libovolný z konců seznamu (nebo také jinam do seznamu), a to i v případě prázdného seznamu, snadno z něj můžeme také vypustit zvolený prvek, a sice jednotným způsobem bez ohledu na to, zda je to prvek koncový nebo ne.

2.3 Příklady použití lineárních seznamů

Lineární spojové seznamy využíváme v programech k uložení dat zejména tehdy, neznáme-li předem přesný počet ukládaných záznamů. Někdy potřebujeme zachovat pořadí, ve kterém byly do seznamu jednotlivé prvky vloženy, a se seznamem pracovat jak se zásobníkem nebo s frontou.

Zásobník je lineární seznam prvků zachovávající pořadí, v němž jsou prvky přidávány na jeden jeho konec (tzv. vrchol zásobníku) a jsou odebírány z téhož konce. Při každém odebírání prvku je tedy ze zásobníku odstraněn ten záznam, který je nejnovější, tzn. který je tam uložen nejkratší dobu. Zásobník můžeme snadno reprezentovat obyčejným lineárním spojovým seznamem. Musíme jen dát pozor na směr, v jakém budou prvky v seznamu navzájem propojeny. Vždy je totiž mnohem snadnější přidávat a ubírat prvky na začátku lineárního spojového seznamu než na jeho konci. Začátek seznamu bude proto představovat vrchol zásobníku, nejstarší prvek bude umístěn na samém konci seznamu. Každý prvek zásobníku bude obsahovat ukazatel na toho, kdo byl zařazen do zásobníku bezprostředně před ním (tedy na „starší“ prvek).



Procedury pro manipulaci se zásobníkem jsou velmi jednoduché. Předvedeme si je na příkladě zásobníku celých čísel.

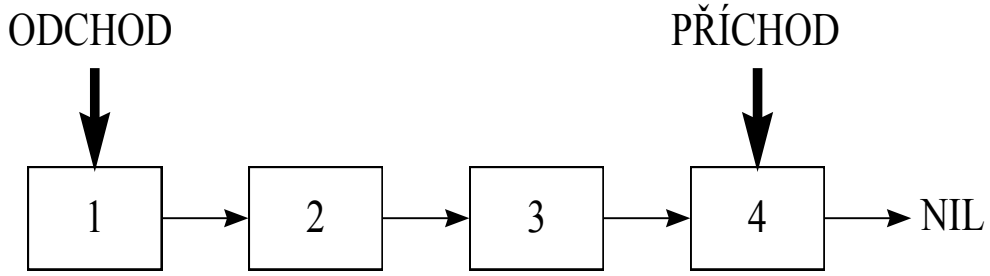
```
type  UkZasobnik = ^Zasobnik;
      Zasobnik = record
                   Cislo: integer;
                   Starsi: UkZasobnik
                 end;
```

```
procedure DoZasobniku(var Z: UkZasobnik; C: integer);
{do zásobníku Z přidá číslo C}
var P: UkZasobnik;
begin
new(P);
P^.Cislo := C;
P^.Starsi := Z;
Z := P
end; {procedure DoZasobniku}

function ZeZasobniku(var Z: UkZasobnik): integer;
{vrací jedno číslo odebrané ze zásobníku;
je-li zásobník prázdný, funkce by neměla být volána,
vrací v tom případě hodnotu maxint}
var P: UkZasobnik;
begin
if Z = nil then
  ZeZasobniku := maxint
else
  begin
  ZeZasobniku := Z^.Cislo;
  P := Z;
  Z := Z^.Starsi;
  dispose(P)
  end
end; {function ZeZasobniku}

function PrazdnyZasobnik(Z: UkZasobnik): boolean;
{vrací informaci, zda je zásobník Z prázdný}
begin
PrazdnyZasobnik := Z = nil
end; {function PrazdnyZasobnik}
```

Fronta je také seznam zachovávající pořadí vložených údajů. Prvky jsou však přidávány na jeden konec seznamu („příchod“) a odebírány jsou z opačného konce („odchod“) podobně, jako je tomu u běžné fronty zákazníků třeba někde v prodejně. Při každém odebírání je tedy z fronty odstraněn vždy nejstarší záznam, ten, co v ní čeká nejdéle. Rovněž k programové realizaci fronty vystačíme s obyčejným lineárním spojovým seznamem. Musíme si ale ještě pečlivěji rozmyslet směr propojení jeho prvků. Již víme, že na začátku lineárního spojového seznamu se prvky velmi snadno přidávají i ubírají. Na konec seznamu je možné snadno přidávat nové prvky za předpokladu, že si budeme průběžně udržovat nejen ukazatel na začátek, ale také na momentální konec seznamu (viz procedura *Pridej1* v kap. 2.2). Pouze odebírání posledního prvku seznamu je akce komplikovanější, k její realizaci nám nepomůže ani pomocný ukazatel na konec seznamu. Museli bychom vždy projít celým seznamem, vyhledat jeho předposlední prvek, a pak teprve můžeme poslední prvek ze seznamu odpojit. Frontu proto postavíme tak, že příchod nových prvků bude na konci spojového seznamu (a budeme si udržovat pomocný ukazatel na dosud poslední prvek ve frontě), zatímco odchod bude na začátku seznamu. Každý prvek ve frontě obsahuje ukazatel na toho, kdo byl do fronty zařazen bezprostředně po něm (tedy na „mladší“ prvek).



Základní operace s frontou si opět předvedeme na příkladě fronty celých čísel.

```
type  UkPrvekFronty = ^PrvekFronty;
      PrvekFronty = record
                      Cislo: integer;
                      Mladsi: UkPrvekFronty
                    end;
      Fronta = record
                 Zacatek, Konec: UkPrvekFronty
               end;

procedure DoFronty(var F: Fronta; C: integer);
{do fronty F přidá číslo C}
var P: UkPrvekFronty;
begin
new(P);
P^.Cislo := C;
if F.Zacatek = nil then          {prázdná fronta}
  begin F.Zacatek := P; F.Konec := P end
else
  begin F.Konec^.Mladsi := P; F.Konec := P end
end; {procedure DoFronty}

function ZFronty(var F: Fronta): integer;
{vrací jedno číslo odebrané z fronty;
je-li fronta prázdná, funkce by neměla být volána,
vrací v tom případě hodnotu maxint}
var P: UkPrvekFronty;
begin
if F.Zacatek = nil then
  ZFronty := maxint     {nic tam není}
else
  begin
  ZFronty:= F.Zacatek^.Cislo;
  P := F.Zacatek;
  if F.Zacatek = F.Konec then
    F.Zacatek := nil    {nic tam nezbylo}
  else
    F.Zacatek := F.Zacatek^.Mladsi;
  dispose(P)
  end
end; {function ZFronty}
```

```
function PrazdnaFronta(F: Fronta): boolean;
{vrací informaci, zda je fronta F prázdná}
begin
PrazdnaFronta:= F.Zacatek = nil
end; {function PrazdnaFronta}
```

Jako příklad zcela odlišného použití lineárních spojových seznamů si můžeme ukázat práci s tzv. **dlouhými čísly**. Pro jednoduchost se omezíme na celá čísla bez znaménka. Potřebujeme navrhnout vhodnou reprezentaci mnohaciferných celých čísel a naprogramovat základní aritmetické operace (např. sčítání). Maximální počet cifer v číslech není předem znám, takže použití pole nemusí být příliš vhodné. Použijeme proto lineární spojový seznam a desítkový zápis čísla rozložíme po cifrách nebo po skupinách cifer do jednotlivých uzlů seznamu. Rozklad na jednotlivé cifry je názornější a o něco snadněji se programuje, pro praktické použití je však paměťově značně neúsporný a operace s čísly jsou také zbytečně pomalejší, než je tomu v případě rozkladu na větší skupiny cifer. Nám však nyní dobře poslouží pro svou jednoduchost k vysvětlení principu práce s dlouhými čísly.

Jednotlivé cifry mohou být v prvcích lineárního spojového seznamu uloženy buď v podobě znakové (jako typ *char*), nebo jako číselné hodnoty (typ *integer*, příp. úspornější *byte*). Druhá možnost je lepší k provádění výpočtů, abychom nemuseli číslice stále převádět ze znakové podoby do číselné a zpět. Dále musíme zvolit směr, jakým bude spojový seznam cifer zřetězen. Pro většinu operací se nám bude spíše hodit směr od cifer nejnižšího řádu (jednotek) k cifrám vyšších řádů. Po cifrách odzadu s přenosem do vyšších řádů se provádí například sčítání, odčítání nebo násobení, odpředu se zase lépe provádějí operace vstupu a výstupu a také dělení. Některé operace, jako např. porovnání dvou čísel, se dají stejně dobře provádět jak při průchodu čísly odzadu, tak zepředu. Budeme-li chtít vykonávat s čísly všechny zmíněné operace, můžeme buď spojový seznam podle potřeby převracet (tzn. měnit směr zřetězení prvků - viz procedura *Obrat* v kap. 2.1), nebo použijeme pro reprezentaci čísel lineární spojový seznam obousměrný (ušetříme si tím práce s převracením pořadí prvků v seznamu, ale zase spotřebujeme pro každé číslo větší prostor v paměti).

Pro ilustraci si ukážeme alespoň dvě jednoduché operace s dlouhými čísly, a to porovnání a sčítání. Budeme pracovat s čísly uloženými v podobě jednosměrného lineárního spojového seznamu, jehož každý prvek obsahuje jednu cifru v číselné podobě a uzly jsou zřetězeny ve směru od cifer nejnižšího řádu k vyšším řádům. Ukazatel na celý seznam reprezentující dlouhé celé číslo tedy ukazuje na prvek obsahující cifru v řádu jednotek.

```
type Uk = ^Uzel;
     Uzel = record
              Cifra: byte;
              Dalsi: Uk
            end;
```

Při porovnávání, které z dvou čísel je větší, rozhoduje v první řadě jejich počet cifer - číslo s více ciframi je zřejmě větší. V případě stejného počtu cifer musíme nalézt nejvyšší řád, v němž se cifry obou srovnávaných čísel liší. Porovnání těchto cifer pak určuje, které z čísel je větší. Existují tři možné výsledky, jak může porovnání čísel *A, B* dopadnout: $A = B$, $A > B$, $A < B$. Použijeme tedy funkci typu *byte* a zvolíme vhodný způsob, jak tyto tři možné výsledky zakódovat (v následujícím příkladě jsou zvolili pro jednotlivé výsledky porovnání funkční hodnoty po řadě 0, 1, 2).

```
function Porovnani(A, B: Uk): byte;
{porovnání velikosti dvou dlouhých čísel A, B;
```

```
výsledek: 0 když A = B, 1 když A > B, 2 když A < B}
var V: byte;
begin
V := 0;
while (A <> nil) and (B <> nil) do
  begin
  if A^.Cifra > B^.Cifra then V := 1
  else if A^.Cifra < B^.Cifra then V := 2;
  A := A^.Dalsi;
  B := B^.Dalsi
  end;
if A <> nil then V := 1
else if B <> nil then V := 2;
Porovnani := V
end; {function Porovnani}
```

Při sčítání dvou čísel postupujeme stejně, jako když na papíře pod sebou sčítáme dvě víceciferná čísla. Sečteme tedy vždy cifry obou čísel ze stejného řádu a nesmíme zapomenout na započítání případného přenosu z nižšího řádu.

```
procedure Soucet(A, B: Uk; var C: Uk);
{sčítá dlouhá čísla A, B, zachová je beze změn,
vytvoří nový výsledný seznam C se součtem hodnot A+B}
var K: Uk;   {ukazatel na poslední uzel v C}
    Prenos: byte;
begin
new(C);
K := C;
C^.Cifra := (A^.Cifra + B^.Cifra) mod 10;
Prenos := (A^.Cifra + B^.Cifra) div 10;
A := A^.Dalsi;
B := B^.Dalsi;
while (A <> nil) and (B <> nil) do
  begin
  new(K^.Dalsi);
  K := K^.Dalsi;
  K^.Cifra := (A^.Cifra + B^.Cifra + Prenos) mod 10;
  Prenos := (A^.Cifra + B^.Cifra + Prenos) div 10;
  A := A^.Dalsi;
  B := B^.Dalsi;
  end;
while A <> nil do
  begin
  new(K^.Dalsi);
  K := K^.Dalsi;
  K^.Cifra := (A^.Cifra + Prenos) mod 10;
  Prenos := (A^.Cifra + Prenos) div 10;
  A := A^.Dalsi;
  end;
while B <> nil do
  begin
  new(K^.Dalsi);
  K := K^.Dalsi;
```

```
  K^.Cifra := (B^.Cifra + Prenos) mod 10;
  Prenos := (B^.Cifra + Prenos) div 10;
  B := B^.Dalsi;
  end;
if Prenos > 0 then
  begin
  new(K^.Dalsi);
  K := K^.Dalsi;
  K^.Cifra := Prenos
  end;
K^.Dalsi := nil
end; {procedure Soucet}
```

3. Nelineární dynamické struktury

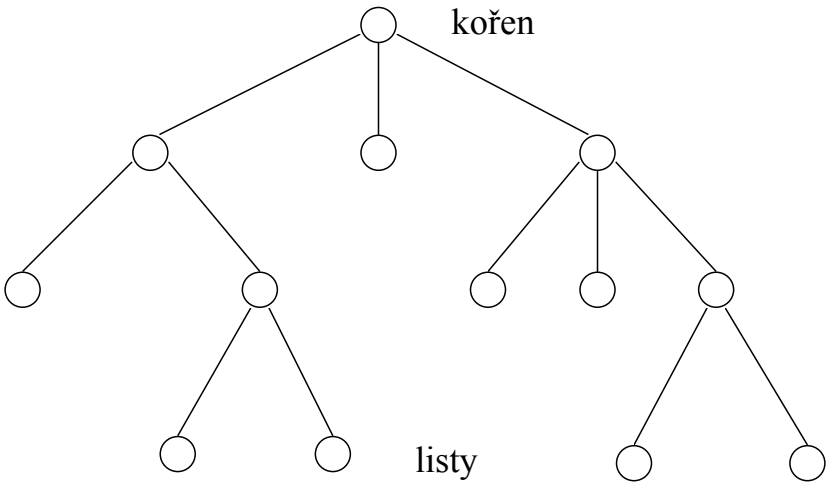
V mnoha programech nevystačíme s lineárními spojovými seznamy, potřebujeme používat i různé komplikovanější dynamické datové struktury. Nejpoužívanější nelineární strukturou je strom, občas využijeme třeba seznam seznamů, dynamickou reprezentaci obecného grafu nebo i různé složitější struktury.

Některé jednodušší dynamické struktury jsou celé tvořeny **uzly téhož typu**. K nim patří jednosměrné i obousměrné lineární spojové seznamy, s nimiž jsme se seznámili již v kap. 2, nebo třeba také binární stromy (kap. 3.1). Ve složitějších dynamických datových strukturách bývají vzájemně provázány **uzly více typů** (např. v seznamu lineárních seznamů nebo v dynamické reprezentaci grafu - viz kap. 3.2).

3.1 Stromy

Pojem **strom** se v matematice používá pro označení zvláštního druhu grafu, totiž souvislého grafu bez cyklů. Stromy jsou spolu s ostatními grafy předmětem zkoumání matematické disciplíny zvané teorie grafů. My zde však nebudeme uvádět žádné přesné formální definice z tohoto oboru a pro naše potřeby plně vystačíme s názorným popisem a intuitivním chápáním všech potřebných pojmů. Pro úplnost jen dodejme, že název strom budeme nadále pro jednoduchost používat k označení struktury, které se v teorii grafů někdy odborně říká zakořeněný nebo také orientovaný strom, zatímco samotný pojem strom tam slouží k označení poněkud obecnější struktury.

Strom je tvořen soustavou uzlů provázaných podle určitých pravidel. Výchozím bodem stromu je jeden význačný uzel zvaný **kořen**. Kořen má několik následníků, každý z nich má zase své následníky, ti mají své vlastní následníky, atd. Maximální počet následníků libovolného uzlu není obecně nijak omezen. Některé uzly nemají žádného následníka a nazývají se **listy**. Vzdálenost od kořene k nejvzdálenějšímu listu označujeme jako **hloubku** stromu.



Pro označování vzájemné polohy uzlů ve stromě někdy používáme slova představující příbuzenské vztahy. O následnících uzlu říkáme, že to jsou jeho synové, a naopak předchůdcem uzlu ve stromu je jeho otec. Synům téhož otce pak můžeme říkat bratři. Tato příbuzenská označení vycházejí z jednoho reálného praktického využití stromové struktury - k zachycení rodokmenu. V tomto případě je „strom rodu“ postaven od prapředka (ten je uložen v kořeni stromu) k nejmladším dětem (v listech). Můžeme ho ale vystavět i obráceně tak, že do kořene stromu umístíme člověka, jehož rodokmen do minulosti chceme zachytit. Následníci kořene budou představovat jeho rodiče, následníci následníků prarodiče

(tj. rodiče rodičů), atd., až v listech stromu budou zaznamenáni nejvzdálenější předkové, kterých se dopátráme. Všimněte si, že v tomto druhém případě má každý uzel stromu nejvýše dva následníky, neboť každý člověk má jednoho otce a jednu matku (alespoň pokud uvažujeme pouze skutečné biologické rodiče). Stromům s touto vlastností říkáme binární stromy. Budeme se jimi zabývat samostatně, neboť jsou velmi časté a mají dokonce větší praktické uplatnění než stromy obecné s neomezeným počtem následníků v každém uzlu.

Binární strom můžeme v programu reprezentovat dynamickou datovou strukturou tvořenou uzly následujícího typu:

```
type  Uk = ^Uzel;
      Uzel = record
               Hodnota: T;        {uložená data}
               L, P: Uk           {levý a pravý následník}
             end;
```

Podobně jako u lineárních spojových seznamů budou všechny uzly binárního stromu alokovány dynamicky pomocí procedury *new*. Statický ukazatel (deklarovaný v programu pomocí *var*) bude obsahovat odkaz na kořen stromu a bude nám při práci se stromem sloužit jako jediný vstupní bod. Ke všem ostatním uzlům stromu se dostaneme od kořene pomocí ukazatelů *L*, *P* obsažených v uzlech. Pro správné procházení stromem je důležité, aby ukazatel *L* resp. *P* v uzlu stromu obsahoval hodnotu *nil*, pokud příslušný levý resp. pravý následník uzlu neexistuje. To je obdobné opatření jako použití hodnoty *nil* na ukončení lineárního spojového seznamu.

V případě programové realizace **obecného stromu** musíme vyřešit problém, že předem neznáme počet následníků jednotlivých uzlů. Pokud by bylo dáno omezení počtu následníků platné pro všechny uzly, mohli bychom do každého uzlu umístit pole ukazatelů na následníky uzlu:

```
type  Uk = ^Uzel;
      Uzel = record
               Hodnota: T;                          {uložená data}
               Nasl: array[1..MaxNasl] of Uk        {následníci}
             end;
```

Toto řešení je však použitelné pouze pro dosti malé hodnoty *MaxNasl*, jinak by se ve většině uzlů zřejmě příliš plýtvalo pamětí (nepoužité prvky pole *Nasl* s hodnotu *nil*). Nabízí se proto řešení použít v každém uzlu stromu místo pole pomocný spojový seznam. V uzlu stromu bude nyní uložen jeden ukazatel na začátek tohoto seznamu a seznam bude přesně tak dlouhý, kolik následníků uzel stromu má. Je-li uzel stromu listem, bude mít jeho položka *Sez* přímo hodnotu *nil* (seznam následníků je prázdný):

```
type  Uk = ^Uzel;
      UkSez = ^Seznam;

      Uzel = record                 {uzel stromu}
               Hodnota: T;          {uložená data}
               Sez: UkSez           {seznam následníků}
             end;

      Seznam = record               {uzel pomocného seznamu}
                 Nasl: Uk;          {následník uzlu stromu}
```

```
    Dalsi: UkSez    {propojeni v seznamu}
  end;
```

Poprvé se tak setkáváme s dynamickou datovou strukturou, ve které budou provázány dynamicky alokované proměnné **dvou odlišných typů** - jednak záznamy typu *Uzel* představující uzly stromu, jednak záznamy typu *Seznam* reprezentující vlastně jednotlivé vztahy otec-syn ve stromě.

Stromová datová struktura je svým charakterem **rekurzivní**. Rekurzivní popis stromu říká, že strom je tvořen kořenem, z něhož vede několik (v krajním případě žádný, v případě binárního stromu nejvýše dva) odkazů na podstromy, z nichž každý je opět stromem. Na této rekurzivní představě je postavena většina algoritmů pro práci se stromy.

Nejjednodušší operací je **průchod stromem**. Chceme projít celým stromem a v každém uzlu provést nějakou operaci s uloženými daty. Pro jednoduchost zde budeme například pouze vypisovat uložené hodnoty. Předpokládejme ještě, že nám nezáleží na pořadí, v jakém jednotlivé uzly stromu procházíme. Myšlenka algoritmu průchodu vychází ze zmíněné rekurzivní definice stromu: zpracovat data ve všech uzlech stromu znamená zpracovat data uložená v kořenu a dále zpracovat obdobným způsobem (rekurzívně) data obsažená ve všech podstromech kořene. Ukážeme si řešení zvlášť pro binární a zvlášť pro obecné stromy. Využijeme přitom výše popsané datové reprezentace stromů.

```
procedure Pruchod1(K: Uk);
{průchod binárním stromem s vypsáním hodnot všech uzlů,
parametr K je ukazatel na kořen stromu;
zcela obecné řešení - připouští se i možnost K = nil}
begin
if K <> nil then
  begin
  write(K^.Hodnota);
  Pruchod1(K^.L);
  Pruchod1(K^.P)
  end
end; {procedure Pruchod1}

procedure Pruchod2(K: Uk);
{průchod binárním stromem s vypsáním hodnot všech uzlů,
parametr K je ukazatel na kořen stromu;
za předpokladu, že K <> nil, lze ve srovnání s procedurou
Pruchod1 ušetřit zhruba polovinu rekurzivních volání}
begin
write(K^.Hodnota);
if K^.L <> nil then Pruchod2(K^.L);
if K^.P <> nil then Pruchod2(K^.P)
end; {procedure Pruchod2}

procedure Pruchod3(K: Uk);
{průchod obecným stromem s vypsáním hodnot všech uzlů,
parametr K je ukazatel na kořen stromu;
předpokládá se volání s hodnotou K <> nil}
var S: UkSez;
begin
write(K^.Hodnota);
S := K^.Sez;
while S <> nil do
```

```
  begin
  Pruchod3(S^.Nasl);
  S := S^.Dalsi
  end
end; {procedure Pruchod3}
```

Můžete se zamyslet nad tím, jak lze ovlivnit pořadí zpracování jednotlivých uzlů stromu tím, že zaměníme pořadí příkazů v procedurách *Pruchod1, Pruchod2, Pruchod3*. Pokud totiž při vstupu do uzlu napřed zpracujeme třeba všechny jeho následníky a pak teprve data uložená v uzlu samotném, budou nakonec zase zpracována data ze všech uzlů stromu, ale v jiném uspořádání.

Další základní operací je **vyhledání hodnoty** ve stromě. Budeme postupovat obdobně jako při procházení celým stromem. Pokud však narazíme na hledanou hodnotu, procházení ihned ukončíme a odkaz na příslušný uzel vrátíme jako výsledek hledání. Není-li hledaná hodnota ve stromě vůbec obsažena, procedura prohledá celý strom a pak ve výstupním parametru vrátí hodnotu *nil*. Pro jednoduchost budeme předpokládat, že v každém uzlu stromu je uloženo jedno celé číslo. Ve skutečnosti mohou být samozřejmě data obsažená ve stromu mnohem komplikovanější a my bychom pak vyhledávali uzel podle jedné zvolené položky (zvané obvykle klíč).

```
procedure Hledej1(K: Uk; H: integer; var Z: Uk);
{v binárním stromu s kořenem K hledá hodnotu H,
v parametru Z vrací odkaz na nalezený uzel, resp. nil;
předpokládá se volání procedury s hodnotou K <> nil}
begin
if K^.Hodnota = H then Z := K
else
  begin
  Z := nil;
  if K^.L <> nil then Hledej1(K^.L, H, Z);
  if (Z = nil) and (K^.P <> nil) then Hledej1(K^.P, H, Z)
  end
end; {procedure Hledej1}

procedure Hledej2(K: Uk; H: integer; var Z: Uk);
{v obecném stromu s kořenem K hledá hodnotu H,
v parametru Z vrací odkaz na nalezený uzel, resp. nil;
předpokládá se volání procedury s hodnotou K <> nil}
var S: UkSez;
begin
if K^.Hodnota = H then Z := K
else
  begin
  Z := nil;
  S := K^.Sez;
  while (Z = nil) and (S <> nil) do
    begin
    Hledej2(S^.Nasl, H, Z);
    S := S^.Dalsi
    end
  end
end; {procedure Hledej2}
```

Předchozí procedury *Hledej1* a *Hledej2* ukazují, že jsou-li data rozložena v uzlech stromu neuspořádaně, vyhledání zvolené hodnoty může být dosti pracné a pomalé. V krajním případě musíme projít celý strom. Stromovou strukturu však můžeme využít i k mnohem šikovnějšímu a efektivnějšímu ukládání a vyhledávání dat, pokud jednotlivé hodnoty rozmístíme do uzlů stromu podle určitých pravidel. Výslednou datovou strukturu nazýváme **binární vyhledávací strom**. Je to binární strom, v němž pro každý uzel *U* platí, že všechny hodnoty uložené v jeho levém podstromu jsou menší než hodnota uzlu *U* a všechny hodnoty uložené v jeho pravém podstromu jsou naopak větší než hodnota uzlu *U*. Budeme-li hledat jistou hodnotu *H* v binárním vyhledávacím stromu, nemusíme procházet celým stromem, nýbrž stačí pouze jednou sestoupit od kořene k listu stromu. Hodnotu *H* porovnáme nejprve s hodnotou obsaženou v kořeni stromu. Je-li náhodou stejná, máme nalezen výsledný uzel. Pokud je *H* menší, stačí pokračovat v hledání v levém podstromu, naopak je-li *H* větší, budeme prohledávat už jen pravý podstrom. Stejně pokračujeme tak dlouho, až buď ve stromě najdeme hodnotu *H*, nebo dojdeme až do uzlu, odkud podle uvedených pravidel nemůžeme pokračovat dále (uzel nemá patřičného následníka).

```
procedure Hledej3(K: Uk; H: integer; var Z: Uk);
{v binárním vyhledávacím stromu s kořenem K hledá hodnotu H,
v parametru Z vrací odkaz na nalezený uzel, resp. nil;
předpokládá se volání procedury s hodnotou K <> nil}
begin
if H = K^.Hodnota then Z := K
else if H < K^.Hodnota then
  if K^.L <> nil then Hledej3(K^.L, H, Z)
                 else Z := nil
else {H > K^.Hodnota}
  if K^.P <> nil then Hledej3(K^.P, H, Z)
                 else Z := nil
end; {procedure Hledej3}
```

Operaci vyhledání hodnoty v binárním vyhledávacím stromu můžeme snadno zapsat také bez použití rekurze. Díky uspořádání hodnot ve stromě vystačíme s jednoduchým cyklem, který realizuje sestup pomocného ukazatele po hladinách stromu od kořene směrem k listům.

```
procedure Hledej4(K: Uk; H: integer; var Z: Uk);
{v binárním vyhledávacím stromu s kořenem K hledá hodnotu H,
v parametru Z vrací odkaz na nalezený uzel, resp. nil}
var Sestup: boolean;
begin
Sestup := K <> nil;
while Sestup do
  begin
  if H = K^.Hodnota then Sestup := false
  else if H < K^.Hodnota then K := K^.L
  else {H > K^.Hodnota} K := K^.P;
  if K = nil then Sestup := false
  end;
Z := K;
end; {procedure Hledej4}
```

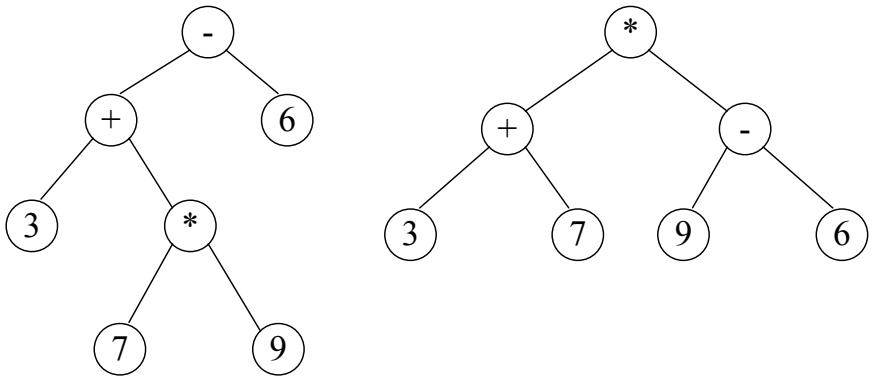
Časová náročnost vyhledávání v binárním vyhledávacím stromu (tj. počet provedených porovnání hodnot) je v nejhorším případě rovna hloubce stromu (tzn. počtu jeho hladin). Hloubka binárního stromu o *N* vrcholech sice může být v extrémním případě degenerovaného stromu až *N*, v nejlepším případě, když je strom vyvážený, je však pouze log_2N a rovněž v průměrném případě se od log_2N příliš neliší. Vyhledávání tedy probíhá ve většině případů velmi rychle.

Také operace související s aktualizací obsahu binárního vyhledávacího stromu, tzn. přidávání nových hodnot do stromu a odstraňování hodnot, probíhají stejně rychle. Vyžadují provést nejvýše jeden sestup od kořene stromu k listu, což je ve stromu o *N* uzlech práce úměrná zpravidla log_2N. Příslušné algoritmy naleznete vysvětlené a naprogramované např. v učebnicích [1] nebo [3].

Další zajímavou oblastí využití binárních stromů je jejich použití pro **reprezentaci aritmetických výrazů.** Výraz je v nejjednodušší podobě tvořen operandy (čísly), binárními operátory (aritmetickými znaménky +, -, *, /) a závorkami. Pravidla pro vyhodnocování výrazů jednoznačně určují, v jakém pořadí se budou jednotlivé operace provádět - rozhodují nejprve závorky, pak priority operátorů a v případě více operátorů téže priority se při vyhodnocování postupuje zápisem výrazu zleva doprava. Toto pořadí vyhodnocování můžeme zaznamenat do binárního stromu, a to dokonce bez použití závorek. V listech stromu budou uloženy operandy, v ostatních uzlech jsou operátory. List je tedy elementární podvýraz s hodnotou přímo určenou v něm uloženým operandem. Každý uzel s operátorem určuje, jakou aritmetickou operaci je třeba vykonat s výsledky podvýrazů reprezentovaných levým a pravým podstromem tohoto uzlu. Výslednou hodnotu výrazu získáme vyhodnocením operátoru uloženého v kořeni stromu.

Na příkladu dvou aritmetických výrazů a jejich reprezentace binárním stromem si můžeme ukázat, jak se změní struktura stromu, pokud výraz pozměníme pouhým přidáním závorek (a tím tedy změnou pořadí vyhodnocování):

3 + 7 * 9 - 6 (3 + 7) * (9 - 6)



Pro uložení aritmetického výrazu pomocí binárního stromu můžeme použít podobnou datovou strukturu, s jakou jsme se seznámili již v úvodu kap. 3.1. Musíme jen ošetřit skutečnost, že v listech stromu je uloženo číslo, zatímco v ostatních uzlech se nachází znaménko. Pro uzly stromu použijeme proto variantní záznam, přičemž v listech nebudeme potřebovat ani ukazatele na následníky.

```
type  Uk = ^Uzel;
      Uzel = record
                case List: boolean of
                true:  (Cislo: real);
```

```
      false: (Znamenko: char; L, P: Uk)
  end;
```

Máme-li aritmetický výraz reprezentován binárním stromem, je velmi snadné spočítat jeho hodnotu. Rekurzivní procedura na **vyhodnocení výrazu** je založena na myšlence, že stačí vyhodnotit levý a pravý podstrom a s jejich výsledky pak provést poslední operaci uloženou v kořeni stromu. Stejně postupujeme i při vyhodnocování dílčích podvýrazů (podstromů).

```
function Vyhodnot(K: Uk): real;
{počítá hodnotu výrazu reprezentovaného binárním stromem,
K je ukazatel na kořen stromu}
begin
with K^ do
  if List then Vyhodnot := Cislo
  else
    case Znamenko of
    '+': Vyhodnot := Vyhodnot(L) + Vyhodnot(P);
    '-': Vyhodnot := Vyhodnot(L) - Vyhodnot(P);
    '*': Vyhodnot := Vyhodnot(L) * Vyhodnot(P);
    '/': Vyhodnot := Vyhodnot(L) / Vyhodnot(P)
    end
end; {function Vyhodnot}
```

Daleko obtížnějším úkolem než právě popsané vyhodnocování je ovšem samotné sestrojení binárního stromu na základě zadaného výrazu. Příslušné algoritmy můžete nalézt například v učebnici [1].

Na závěr kapitoly věnované stromům ještě dodejme, že programová realizace stromu dynamickou datovou strukturou není jedinou možností. Známe-li předem maximální počet uzlů ve stromu (označme ho konstantou *MaxUzlu*) a maximální počet následníků každého uzlu (*MaxNasl*), můžeme strom uložit do pole. Dynamické ukazatele na následníky uzlů v něm jednoduše nahradí indexy v poli. Kořen stromu bude vždy reprezentován prvkem pole s indexem 1. Hodnotu ukazatele *nil* nahradí index následníka 0. Pro binární strom tedy použijeme pole typu

```
type  BinStrom = array[1..MaxUzlu] of
                   record
                     Hodnota: T;       {uložená data}
                     L, P: 0..MaxUzlu  {levý a pravý následník}
                   end;
```

zatímco pro obecný strom

```
type  Strom = array[1..MaxUzlu] of
                record
                  Hodnota: T;         {uložená data}
                  Nasl: array[1..MaxNasl] of 0..MaxUzlu
                end;
```

Princip všech algoritmů pro práci se stromy zůstane zachován zcela beze změny.

3.2 Složitější datové struktury

Některé úlohy vyžadují použít i komplikovanější datovou strukturu, než je pouhý lineární spojový seznam nebo strom. V mnoha úlohách se například pracuje s **grafy**. Opět se vyhneme přesné formální definici grafu a omezíme se na intuitivní představu, že graf je struktura tvořená uzly (tzv. vrcholy grafu) a spojnicemi mezi nimi (tzv. hrany grafu). V případě orientovaného grafu je každá hrana navíc orientována, tzn. je určen směr, odkud kam vede. U obyčejných neorientovaných grafů představuje každá hrana obousměrný spoj příslušných vrcholů. Pokud je graf ohodnocený, je navíc každé hraně přiřazeno ohodnocení (například číselný údaj o délce hrany). Jednoduchou představu, jak může vypadat graf, získáme pohledem na mapu - města představují vrcholy grafu a silnice, které je spojují, jsou jeho hrany (nesmějí se ovšem křížit ani větvit nikde mimo města).

Při programování grafových úloh zpravidla pracujeme s různými reprezentacemi grafů pomocí matic, tj. dvourozměrných polí. Některé možnosti uložení grafu v programu a vhodné způsoby naprogramování nejběžnějších grafových algoritmů lze nalézt v knize [1]. Neznáme-li však předem omezení na maximální počet vrcholů grafu, můžeme někdy využít i dynamickou reprezentaci grafu. Z hlediska struktury se vůbec neliší od uložení obecného stromu, které známe už z kap. 3.1.

```
type  Uk = ^Vrchol;
      UkHr = ^Hrana;

      Vrchol = record              {vrchol grafu}
                 Hodnota: T;       {uložená data}
                 Hrany: UkHr       {seznam hran vedoucích
                                    z vrcholu}
               end;

      Hrana = record               {uzel pomocného seznamu
                                   = orientovaná hrana}
                Kam: Uk;           {kam hrana vede}
                Dalsi: UkHr{propojeni v seznamu hran}
              end;
```

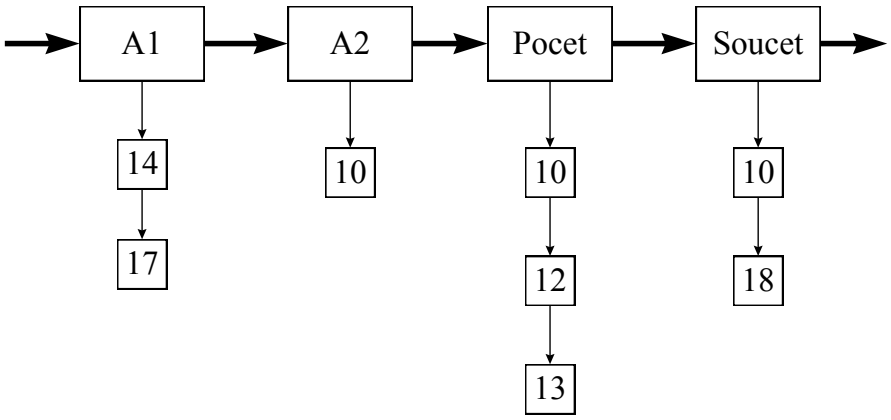
Hrany v této dynamické reprezentaci jsou vždy orientované. Pokud potřebujeme pracovat s neorientovaným grafem, nahradíme každou jeho neorientovanou hranu dvěma orientovanými hranami vedoucími mezi dvojicí vrcholů tam a zpět. V případě ohodnoceného grafu přibude do záznamu *Hrana* ještě další položka představující ohodnocení hrany.

Druhým praktickým příkladem složitějších dynamických datových struktur je **seznam lineárních seznamů**. Chtěli bychom například prozkoumat zdrojový text programu zapsaného v programovacím jazyce Pascal, nalézt v něm všechny identifikátory, vytvořit jejich seznam a ke každému z nich zaznamenat všechna čísla řádků, na nichž se v programu nachází. Výslednému seznamu identifikátorů s čísly řádků jejich výskytu v programu se říká tabulka křížových referencí (crossreference) a některé překladače ji poskytují pro snadnější orientaci v programu a bezpečnější provádění oprav ve zdrojovém textu. Předem neznáme ani počet identifikátorů, ani jak často se v textu programu budou opakovat, takže použití dynamické datové struktury bude zřejmě vhodnější než pole. Nejjednodušším řešením je vytvořit lineární spojový seznam uzlů obsahujících jednotlivé identifikátory. Kromě zápisu identifikátoru bude každý uzel obsahovat ještě ukazatel na spojový seznam prvků s příslušnými čísly řádků. Jedná se tedy opět o datovou strukturu tvořenou dynamicky alokovanými proměnnými dvou odlišných typů.

```
type  UkIdent = ^Ident;
      UkRadek = ^Radek;
```

```
Ident = record              {uzel s identifikátorem}
          Identifikator: string[MaxDelka];
          Radky: UkRadek;  {seznam výskytů}
          Dalsi: UkIdent   {další identifikátor}
        end;

Radek = record              {uzel v seznamu řádků}
          Cislo: word;      {číslo řádku}
          Dalsi: UkRadek    {další řádek s výskytem}
        end;
```

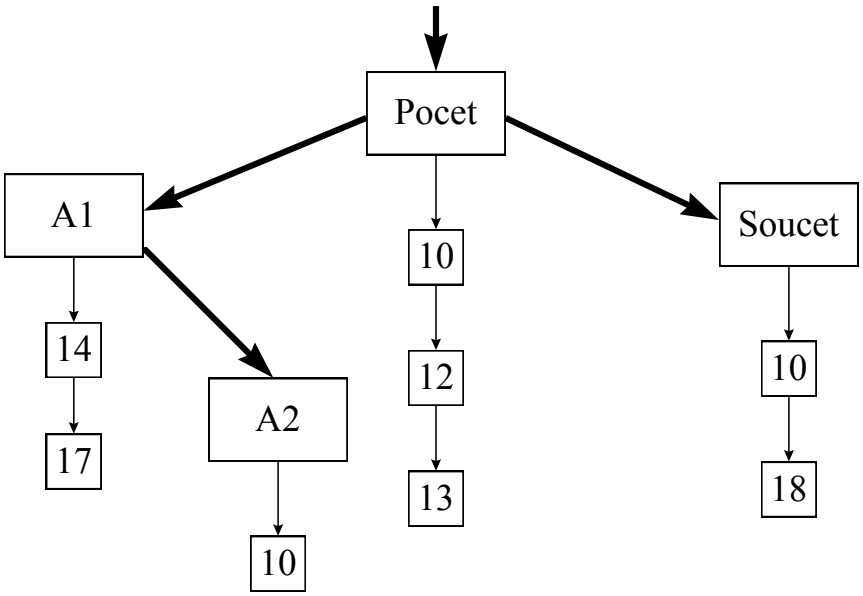


Budeme-li chtít zrychlit operace vyhledání identifikátoru, přidání dalšího identifikátoru a vkládání nového údaje o výskytu identifikátoru v programu, můžeme nahradit lineární seznam identifikátorů binárním vyhledávacím stromem (viz kap. 3.1). V uzlu stromu bude uložen vždy jeden identifikátor (podle této položky je vyhledávací strom uspořádán), ukazatel na lineární spojový seznam s čísly řádků jeho výskytu v programu a ukazatelé na levý a pravý podstrom, v nichž jsou uloženy další identifikátory. Výsledná datová struktura bude mít tedy tvar **binárního stromu s odkazem na lineární spojový seznam v každém uzlu**.

```
type  UkIdent = ^Ident;
      UkRadek = ^Radek;

      Ident = record              {uzel s identifikátorem}
                Identifikator: string[MaxDelka];
                Radky: UkRadek;  {seznam výskytů}
                L, P: UkIdent    {levý a pravý následník}
              end;

      Radek = record              {uzel v seznamu řádků}
                Cislo: word;      {číslo řádku}
                Dalsi: UkRadek    {další řádek s výskytem}
              end;
```



Složitější dynamická datová struktura nemusí vždy sloužit k zachycení struktury uložených dat. Někdy ji použijeme pouze **k realizaci zvoleného algoritmu**. Příkladem takového použití je **průchod binárním stromem „po vrstvách“**. Již v kap. 3.1 jsme se seznámili s algoritmem průchodu binárním stromem pomocí rekurze (procedury *Pruchod1* a *Pruchod2*). Nyní budeme chtít vypsat hodnoty uložené ve všech uzlech binárního stromu v pevně stanoveném pořadí, a to po jednotlivých hladinách stromu postupně od kořene k listům. Na každé hladině navíc máme při vypisování postupovat zleva doprava.

Pro řešení této úlohy doplníme dočasně binární strom s uloženými daty další dynamickou datovou strukturou, která bude s původním stromem provázána pomocí ukazatelů. Bude to lineární spojový seznam představující frontu (fronta viz kap. 2.3), v níž každý uzel obsahuje odkaz na jeden uzel stromu. Během procházení stromem budeme do fronty ukládat odkazy na uzly, jejichž hodnotu je ještě třeba vypsat. Vždy po vypsání hodnoty nějakého uzlu tento uzel z fronty vyřadíme a naopak do fronty zařadíme odkazy na jeho následníky. Způsob obsluhy fronty (zachovává se pořadí, tzn. vždy je obsloužen ten, kdo nejdéle čeká) zajišťuje, že jsou hodnoty z uzlů stromu vypisovány skutečně po vrstvách.

Reprezentace binárního stromu (viz též kap. 3.1):

```
type  Uk = ^Uzel;
      Uzel = record
                Hodnota: integer;
                L, P: Uk
             end;
```

Dynamická realizace fronty s odkazy na uzly stromu (viz též kap. 2.3):

```
type  UkPrvekFronty = ^PrvekFronty;

      PrvekFronty = record
                      Kdo: Uk;
```

```
                    Dalsi: UkPrvekFronty
                  end;

      Fronta = record
                 Zacatek, Konec: UkPrvekFronty
               end;


procedure PruchodPoVrstvach(K: Uk);
{průchod binárním stromem s vypsáním hodnot všech uzlů
po vrstvách,
parametr K je ukazatel na kořen stromu,
předpokládá se volání s hodnotou K <> nil}
var F: Fronta;
    U: Uk;      {zpracovávaný uzel stromu}
    Pom: UkPrvekFronty;
                {pomocný ukazatel pro vyřazování z fronty}
begin
new(F.Zacatek);
F.Konec := F.Zacatek;
F.Zacatek^.Kdo := K;          {ve frontě je jen kořen stromu}
while F.Zacatek <> nil do     {dokud není fronta prázdná}
  begin
  U := F.Zacatek^.Kdo;        {zpracovávaný uzel stromu}
  write(U^.Hodnota);          {vypsat jeho hodnotu}
  if U^.L <> nil then         {levý syn do fronty}
    begin
    new(F.Konec^.Dalsi);
    F.Konec := F.Konec^.Dalsi;
    F.Konec^.Kdo := U^.L
    end;
  if U^.P <> nil then         {pravý syn do fronty}
    begin
    new(F.Konec^.Dalsi);
    F.Konec := F.Konec^.Dalsi;
    F.Konec^.Kdo := U^.P
    end;
  Pom := F.Zacatek;
  if F.Zacatek = F.Konec then
    F.Zacatek := nil
  else
    F.Zacatek := F.Zacatek^.Dalsi;
  dispose(Pom)
  end
end; {procedure PruchodPoVrstvach}
```

4. Úlohy k procvičení

Většina úloh je formulována tak, že úkolem není napsat celý program, ale jen naprogramovat proceduru nebo funkci, která provádí uvedenou operaci s danou dynamickou strukturou.

Úvodní a zároveň nejrozsáhlejší část této kapitoly se týká práce s lineárními spojovými seznamy jakožto se základní a nejpoužívanější dynamickou datovou strukturou. Všechny uvedené úlohy předpokládají, že pracujeme s jednosměrným lineárním spojovým seznamem, který má v každém svém prvku jako klíčovou informaci uloženo jedno celé číslo (jen v několika úlohách je to jeden znak). Není-li řečeno jinak, nevylučuje se, že se čísla uložená v jednotlivých prvcích seznamu mohou opakovat, tj. více prvků seznamu může obsahovat stejné číslo. V zadání některých úloh, u nichž je tato skutečnost podstatná pro postup řešení, je to znovu ještě připomenuto. Úlohy je možné modifikovat tak, že změníme množství a druh informace uložené v prvcích seznamu. Zadání všech úloh je také možné změnit tak, že budeme pracovat s obousměrnými spojovými seznamy.

Závěr kapitoly obsahuje několik úloh na práci s binárními stromy a s binárními vyhledávacími stromy.

4.1 Lineární spojové seznamy

1. Napište proceduru, která vytvoří lineární spojový seznam celých čísel. Seznam bude mít 10 prvků s hodnotami po řadě 1, 2, 3,..., 10.

2. Napište proceduru, která vytvoří lineární spojový seznam celých čísel. Seznam bude mít 10 prvků s hodnotami, které procedura přečte ze vstupu. Pořadí čísel uložených v seznamu bude opačné, než v jakém byla čísla zadána na vstupu.

3. Napište proceduru, která vytvoří lineární spojový seznam celých čísel. Seznam bude mít 10 prvků s hodnotami, které procedura přečte ze vstupu. Pořadí čísel uložených v seznamu bude stejné, v jakém byla čísla zadána na vstupu.

4. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Napište funkci, která určí počet prvků seznamu obsahujících číslo dělitelné pěti. Např. pro seznam 1, 5, 10, 7, 5 bude výsledkem hodnota 3.

5. Napište proceduru, která z daného lineárního spojového seznamu celých čísel vynechá všechny prvky obsahující číslo dělitelné pěti. Např. ze seznamu 1, 5, 10, 7, 5 tak vznikne upravený seznam 1, 7.

6. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Jednotlivá čísla se mohou v seznamu opakovat. Napište funkci, která určí největší hodnotu čísla uloženého v daném seznamu. Např. pro seznam tvořený prvky s čísly 4, 8, 8, 4, 6, 8, 4 bude výsledkem hodnota 8.

7. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Jednotlivá čísla se mohou v seznamu opakovat. Napište funkci, která určí počet výskytů největší hodnoty čísla v daném seznamu. Např. pro seznam tvořený prvky s čísly 4, 8, 8, 4, 6, 8, 4 bude výsledkem číslo 3, neboť největší hodnota 8 je v seznamu obsažena třikrát.

8. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Jednotlivá čísla se mohou v seznamu opakovat. Napište proceduru, která z daného seznamu vynechá v pořadí první prvek obsahující největší ze všech čísel v seznamu. Např. ze seznamu 4, 8, 8, 4, 6, 8, 4 tak vznikne upravený seznam 4, 8, 4, 6, 8, 4.

9. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Jednotlivá čísla se mohou v seznamu opakovat. Napište proceduru, která z daného seznamu vynechá poslední prvek obsahující největší ze všech čísel v seznamu. Např. ze seznamu 4, 8, 8, 4, 6, 8, 4 tak vznikne upravený seznam 4, 8, 8, 4, 6, 4.

10. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Jednotlivá čísla se mohou v seznamu opakovat. Napište proceduru, která z daného seznamu vynechá všechny prvky obsahující největší ze všech čísel v seznamu. Např. ze seznamu 4, 8, 8, 4, 6, 8, 4 tak vznikne upravený seznam 4, 4, 6, 4.

11. Napište proceduru, která z daného lineárního spojového seznamu celých čísel vynechá všechny takové prvky, které obsahují stejné číslo jako prvek seznamu bezprostředně jim předcházející. Např. ze seznamu 1, 4, 4, 4, 5, 4, 9, 9 tak vznikne upravený seznam 1, 4, 5, 4, 9.

12. Napište proceduru, která z daného lineárního spojového seznamu celých čísel vynechá všechny takové prvky, které obsahují stejné číslo jako některý předcházející prvek seznamu. Výsledný seznam tedy bude obsahovat všechna čísla jako původní seznam, ale každé z nich pouze jednou. Např. ze seznamu 1, 4, 4, 4, 5, 4, 9, 9 tak vznikne upravený seznam 1, 4, 5, 9.

13. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Napište proceduru, která v daném lineárním spojovém seznamu všechny prvky zdvojí, tj. za každý prvek vloží jeden nový prvek se stejnou hodnotou. Např. ze seznamu 1, 2, 3, 4 tak vznikne upravený seznam 1, 1, 2, 2, 3, 3, 4, 4.

14. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Napište proceduru, která v daném lineárním spojovém seznamu zdvojí všechny prvky obsahující kladné číslo. To znamená, že za každý prvek s kladným číslem procedura vloží jeden nový prvek se stejnou hodnotou. Např. ze seznamu 1, -2, -3, 4 tak vznikne upravený seznam 1, 1, -2, -3, 4, 4.

15. V prvcích lineárního spojového seznamu jsou uložena celá čísla. Napište proceduru, která daný lineární spojový seznam zdvojí, tj. na konec seznamu připojí ještě kopii každého prvku seznamu se zachováním původního pořadí. Např. ze seznamu 1, 2, 3, 4 tak vznikne upravený seznam 1, 2, 3, 4, 1, 2, 3, 4.

16. V prvcích lineárního spojového seznamu jsou uložena navzájem různá celá čísla. Napište proceduru, která v seznamu vymění prvek obsahující nejmenší číslo s prvkem obsahujícím největší číslo. Výměna se musí provést přepojením prvků, není povoleno kopírovat uložené celočíselné hodnoty.

17. Napište proceduru, která přečte ze vstupního textového souboru posloupnost kladných celých čísel ukončenou nulou a vytvoří dva spojové seznamy. Tyto seznamy budou ve svých prvcích obsahovat přečtená kladná čísla (tj. všechna čísla ze vstupu kromě koncové nuly). V prvním seznamu budou uložena všechna přečtená lichá čísla ve stejném pořadí, v jakém byla uvedena na vstupu, ve druhém budou sudá čísla v opačném pořadí, než v jakém byla zadána na vstupu. Např. pro vstupní posloupnost čísel 11, 28, 24, 17, 99, 66, 60, 0 bude první seznam tvořen prvky s čísly v pořadí 11, 17, 99, druhý seznam bude mít tvar 60, 66, 24, 28.

18. Napište funkci typu boolean, která zjišťuje, zda jsou dané dva lineární spojové seznamy celých čísel stejné, tzn. zda jsou tvořeny stejnými prvky ve stejném pořadí.

19. Napište funkci typu boolean, která zjišťuje, zda jsou dané dva lineární spojové seznamy celých čísel (obecně neuspořádané) tvořeny stejnými prvky. Pro jednoduchost předpokládejte, že v žádném ze seznamů se čísla neopakují. Až funkce ukončí svoji práci, dané seznamy čísel nemusejí zůstat zachovány. Např. pro seznamy 5, 7, 9, 11 a 11, 5, 9, 7 bude správná odpověď znít true.

20. Napište proceduru, která daný spojový seznam kladných celých čísel rozdělí na *K* seznamů tak, že do každého z výsledných seznamů zařadí vždy všechna čísla dávající stejný zbytek při celočíselném dělení číslem *K*. Hodnota *K* je zadána parametrem, předem je znám její horní odhad (například 10). Např. pro vstupní seznam 1, 7, 5, 14, 2, 9, 6, 16, 4, 19, 8 a hodnotu *K*=3 vytvoří procedura následující tři seznamy: první obsahuje čísla dělitelná třemi a má tvar 9, 6, druhý seznam je tvořen čísly 1, 7, 16, 4, 19 (dávají při dělení třemi zbytek 1) a třetí vytvořený seznam obsahuje zbývající čísla 5, 14, 2, 8, která při dělení třemi dají zbytek 2.

21. Dva lineární spojové seznamy obsahují ve svých prvcích celá čísla. V prvním ze zadaných seznamů jsou všechna čísla navzájem různá. Napište proceduru, která v prvním seznamu najde prvek s největším číslem a tento prvek přeřadí na konec druhého seznamu. Např. pro vstupní seznamy s čísly 11, 70, 35 a 25, 99, 6, 18 vede požadovaná úprava k seznamům ve tvaru 11, 35 a 25, 99, 6, 18, 70.

22. Dva lineární spojové seznamy obsahují ve svých prvcích celá čísla. V prvním ze zadaných seznamů jsou všechna čísla navzájem různá. Napište proceduru, která v prvním seznamu najde prvek s největším číslem. Tento nalezený prvek potom procedura přeřadí na konec druhého seznamu, pokud ve druhém seznamu dosud není žádný prvek s tímto číslem. V opačném případě nalezený prvek z prvního seznamu vypustí a zruší. Např. pro vstupní seznamy s čísly 11, 70, 35 a 25, 99, 6, 18 vede požadovaná úprava k seznamům ve tvaru 11, 35 a 25, 99, 6, 18, 70. Naproti tomu pro seznamy 11, 70, 35 a 25, 70, 6, 18 bude mít po úpravě první seznam rovněž tvar 11, 35, ale druhý seznam zůstane beze změn (číslo 70 v něm již bylo).

23. Dva lineární spojové seznamy obsahují ve svých prvcích celá čísla. Oba seznamy jsou uspořádané vzestupně podle velikosti uložených čísel. Napište proceduru, která přeřadí prvek s největším číslem (tj. poslední prvek) z prvního seznamu do druhého seznamu tak, aby druhý seznam zůstal uspořádaný. Např. pro vstupní seznamy 11, 35, 70 a 22, 25, 91, 99 budou mít výsledné seznamy tvar 11, 35 a 22, 25, 70, 91, 99.

24. Dva lineární spojové seznamy obsahující ve svých prvcích celá čísla jsou uspořádány vzestupně podle velikosti. Napište proceduru, která přesune všechny jejich prvky do jednoho uspořádaného seznamu. Např. pro vstupní seznamy 11, 35, 70 a 22, 25, 91, 99 bude mít výsledný seznam tvar 11, 22, 25, 35, 70, 91, 99.

25. Napište proceduru, která obrací pořadí prvků v jednosměrném lineárním spojovém seznamu. Např. ze seznamu 2, 4, 6, 8, 10 tak vznikne seznam obsahující prvky v pořadí 10, 8, 6, 4, 2.

26. Množiny jsou reprezentovány jako jednosměrné spojové seznamy svých prvků (každý prvek množiny je v seznamu uložen právě jednou). Napište proceduru, která v této reprezentaci provádí operaci sjednocení dvou množin. Např. pro dva vstupní seznamy, z nichž jeden obsahuje (v libovolném pořadí) prvky 1, 3, 5, 6, 8 a druhý 2, 3, 4, 5, 6, bude výsledný seznam obsahovat (opět v libovolném pořadí) prvky 1, 2, 3, 4, 5, 6, 8.

27. Množiny jsou reprezentovány jako jednosměrné spojové seznamy svých prvků (každý prvek množiny je v seznamu uložen právě jednou). Napište proceduru, která v této reprezentaci provádí operaci průnik množin. Např. pro dva vstupní seznamy, z nichž jeden obsahuje (v libovolném pořadí) prvky 1, 3, 5, 6, 8 a druhý 2, 3, 4, 5, 6, bude výsledný seznam obsahovat (opět v libovolném pořadí) prvky 3, 5, 6.

28. Řešte předchozí dvě úlohy za předpokladu, že množina je reprezentována uspořádaným lineárním spojovým seznamem svých prvků.

29. Velké přirozené číslo je v programu reprezentováno jako lineární spojový seznam svých cifer. Napište proceduru nebo funkci pro sčítání takto reprezentovaných čísel.

30. Velké přirozené číslo je v programu reprezentováno jako lineární spojový seznam cifer. Napište proceduru nebo funkci pro porovnávání takto reprezentovaných čísel. Výsledkem práce procedury (funkce) bude informace, zda je první ze zadaných čísel menší nebo větší než druhé číslo, případně zda se obě čísla sobě rovnají.

31. Napište proceduru, která přečte ze vstupního textového souboru dvě velká kladná celá čísla (každé je zapsáno na jednom řádku souboru) a vytiskne jejich součet. Velikost čísel není nijak omezena.

32. Napište proceduru, která přečte ze vstupního textového souboru tři velká kladná celá čísla (každé je zapsáno na jednom řádku souboru) a vytiskne jejich součet. Velikost čísel není nijak omezena.

33. Napište funkci na přesný výpočet hodnoty *N!* (faktoriál) pro dané kladné celé číslo *N*. Výslednou hodnotu předejte ve vhodné reprezentaci pomocí lineárního spojového seznamu.

34. Napište funkci na přesný výpočet hodnoty 2^N (*N*-tá mocnina čísla 2) pro dané kladné celé číslo *N*. Výslednou hodnotu předejte ve vhodné reprezentaci pomocí lineárního spojového seznamu.

35. Napište proceduru, která vytvoří lineární spojový seznam celých čísel. Seznam bude mít 10 prvků s hodnotami, které procedura přečte ze vstupu. Ve vytvořeném seznamu budou čísla uspořádána vzestupně podle velikosti.

36. Je dán lineární spojový seznam obsahující ve svých prvcích celá čísla. Napište proceduru, která prvky seznamu uspořádá vzestupně podle hodnot uložených čísel. Není povoleno měnit celočíselné hodnoty uložené v jednotlivých prvcích seznamu, smíte pouze přepojovat celé prvky seznamu.

37. Znakové řetězce jsou reprezentovány jako jednosměrné spojové seznamy znaků. Napište proceduru, která v řetězci *A* nahradí první výskyt řetězce *B* řetězcem *C* (kde *A, B, C* jsou parametry procedury).

38. Znakové řetězce jsou reprezentovány jako jednosměrné spojové seznamy znaků. Napište proceduru, která z řetězce *A* vynechá poslední výskyt řetězce *B* (kde *A, B* jsou parametry procedury).

39. Znakové řetězce jsou reprezentovány jako jednosměrné spojové seznamy znaků. Napište proceduru, která daný řetězec doplní na dvojnásobnou délku zrcadlovým obrazem původního řetězce.

40. Znakové řetězce jsou reprezentovány jako jednosměrné spojové seznamy znaků. Napište funkci, jejíž funkční hodnotou je počet výskytů řetězce *A* v řetězci *B*. Jednotlivé výskyty řetězce *A* v řetězci *B* se mohou částečně překrývat. *A* a *B* jsou parametry funkce.

41. Mnohočlen je reprezentován pomocí lineárního spojového seznamu tak, že v každém prvku seznamu je uložena hodnota koeficientu a exponentu jednoho členu. Prvky seznamu jsou uspořádány sestupně podle hodnot exponentu. Členy s nulovým koeficientem se do seznamu neukládají. Napište proceduru, která přečte ze vstupu údaje o koeficientech a exponentech daného mnohočlenu a vytvoří jeho datovou reprezentaci.

42. Mnohočlen je reprezentován pomocí lineárního spojového seznamu tak, že v každém prvku seznamu je uložena hodnota koeficientu a exponentu jednoho členu. Prvky seznamu jsou uspořádány sestupně podle hodnot exponentu. Členy s nulovým koeficientem se do seznamu neukládají. Napište proceduru, která na základě uvedené reprezentace mnohočlenu tento mnohočlen vytiskne.

43. Mnohočlen je reprezentován pomocí lineárního spojového seznamu tak, že v každém prvku seznamu je uložena hodnota koeficientu a exponentu jednoho členu. Prvky seznamu jsou uspořádány sestupně podle hodnot exponentu. Členy s nulovým koeficientem se do seznamu neukládají. Napište proceduru na sčítání dvou mnohočlenů.

44. Mnohočlen je reprezentován pomocí lineárního spojového seznamu tak, že v každém prvku seznamu je uložena hodnota koeficientu a exponentu jednoho členu. Prvky seznamu jsou uspořádány sestupně podle hodnot exponentu. Členy s nulovým koeficientem se do seznamu neukládají. Napište proceduru na násobení dvou mnohočlenů.

4.2 Binární stromy

45. Je dán binární strom, který má v každém svém uzlu uloženo jedno celé číslo. Napište proceduru, která projde daným binárním stromem do šířky (tj. po vrstvách stromu) a při průchodu každým uzlem vytiskne hodnotu uloženého čísla.

46. Napište proceduru, která přečte ze vstupního textového souboru posloupnost (alespoň tří) kladných celých čísel ukončenou nulou a vytvoří binární vyhledávací strom, do kterého uloží všechna přečtená čísla.

47. Napište proceduru, která přečte ze vstupního textového souboru posloupnost (alespoň tří) kladných celých čísel ukončenou nulou a vytvoří binární vyhledávací strom, do kterého uloží všechna přečtená čísla kromě dvou největších z nich.

48. V binárním vyhledávacím stromu jsou uložena celá čísla. Napište proceduru, která z něj vypustí uzel s daným číslem, pokud tam je takové číslo vůbec uloženo. Není-li dané číslo ve stromě obsaženo, procedura ponechá binární vyhledávací strom beze změn. Parametry procedury jsou odkaz na kořen stromu a vypouštěné číslo.

49. V binárním vyhledávacím stromu jsou uložena celá čísla. Napište proceduru, která z něj vypustí všechny uzly s hodnotami vyššími, než je dané číslo *C*. Parametry procedury jsou odkaz na kořen stromu a číslo *C*.

50. V uzlech binárního vyhledávacího stromu jsou uložena celá čísla. Napište proceduru, která vypíše ohodnocení všech uzlů daného binárního vyhledávacího stromu v pořadí podle rostoucích hodnot klíčů.

51. V uzlech binárního vyhledávacího stromu jsou uložena celá čísla. Napište proceduru, která vypíše ohodnocení všech uzlů daného binárního vyhledávacího stromu v pořadí podle rostoucích hodnot klíčů. Nepoužívejte rekurzi.

52. Napište proceduru, která vytvoří dokonale vyvážený binární vyhledávací strom o N vrcholech (N je parametr), v němž budou vrcholy ohodnoceny čísly 1 až N. Dokonale vyvážený strom je takový, ve kterém se počet vrcholů v levém a pravém podstromu libovolného vrcholu liší nejvýše o jednu.

53. Množinu budeme reprezentovat jako binární vyhledávací strom, ve kterém jsou uloženy právě všechny prvky množiny. Napište proceduru, která v této reprezentaci provádí operaci sjednocení množin.

54. Množinu budeme reprezentovat jako binární vyhledávací strom, ve kterém jsou uloženy právě všechny prvky množiny. Napište proceduru, která v této reprezentaci provádí operaci průnik množin.

4.3 Návody k řešení úloh

1. Seznam můžete vytvářet buď od začátku s připojováním nových prvků na konec seznamu, nebo odzadu a nové prvky připojovat vždy na začátek seznamu. Druhá z uvedených možností je o něco jednodušší.

2. Čísla čtěte ze vstupu a hned je vkládejte do nově vytvářených prvků seznamu. Seznam vytvářejte odzadu, tj. nové prvky připojujte vždy na začátek seznamu.

3. Čísla čtěte ze vstupu a hned je vkládejte do nově vytvářených prvků seznamu. Seznam vytvářejte odpředu, tj. nové prvky připojujte vždy na konec seznamu. K tomu je výhodné udržovat si stále pomocný ukazatel na dosud poslední prvek vytvářeného seznamu. Vložení prvního čísla do seznamu se ale musí provést odlišným způsobem.

5. Vynechání prvku ze začátku spojového seznamu se provádí jinak než vynechání jiného prvku. Proto je vhodné nejprve ve zvláštním cyklu vypustit všechny prvky s číslem dělitelným pěti, které jsou umístěny na začátku seznamu.

8. Nejjednodušší postup řešení vyžaduje projít daným spojovým seznamem dvakrát. Při prvním průchodu se vyhledá vypouštěný prvek, při druhém průchodu se najde jeho předchůdce a patřičný prvek se ze seznamu vypustí. Lepší je řešení, které vystačí s jedním průchodem. Při postupném průchodu spojovým seznamem si v takovém případě stále musíte udržovat odkaz na předchůdce prvního takového prvku, který obsahuje dosud největší nalezené číslo.

9. Viz návod k předchozí úloze. Jediná změna spočívá v tom, že při postupném průchodu spojovým seznamem si stále musíte udržovat odkaz na předchůdce posledního takového prvku, který obsahuje dosud největší nalezené číslo.

10. Máte v podstatě dvě možnosti, jak řešit úlohu. Při průchodu daným spojovým seznamem si můžete vytvořit pomocný seznam odkazů na předchůdce všech takových prvků, které obsahují největší nalezené číslo. Pomocí tohoto pomocného seznamu potom můžete příslušné prvky vypustit. Přitom je třeba zvlášť ošetřit případ, že se největší z čísel nachází také v prvním prvku seznamu (tento prvek nemá předchůdce). Druhou možností je projít daný spojový seznam dvakrát. Při prvním průchodu pouze určíte největší číslo v seznamu, při druhém průchodu vypustíte všechny prvky seznamu, které ho obsahují. Druhý postup řešení může vést k o něco pomalejšímu výpočtu, ale lépe se programuje.

13. Vždy po vložení nového prvku do seznamu musíte dát pozor, abyste se pro další práci přesunuli až za něj, tzn. na další prvek původního seznamu.

14. Vždy po vložení nového prvku do seznamu musíte dát pozor, abyste se pro další práci přesunuli až za něj, tzn. na další prvek původního seznamu.

17. První seznam s lichými čísly bude vytvářen odpředu, druhý seznam se sudými čísly odzadu.

20. Procedura bude mít tři parametry: ukazatel na začátek výchozího spojového seznamu, celočíselný parametr *K* a pole ukazatelů na začátky vytvářených seznamů.

25. Ze začátku daného seznamu postupně odpojujte jednotlivé prvky a vytvářejte z nich nový seznam. Přesouvané prvky zapojujte vždy na začátek nově vytvářeného seznamu. Po rozebrání celého daného seznamu bude vytvořený výsledný seznam obsahovat přesně stejné prvky, ale v opačném pořadí, neboť byl vytvářen odzadu.

26. Na rozdíl od prostého spojení dvou seznamů zde musíte zabránit opakovanému výskytu téže hodnoty ve výsledném seznamu.

28. Narozdíl od předchozích úloh stačí jeden souběžný průchod oběma danými seznamy.

29. Pro potřeby sčítání je výhodné zvolit si takovou orientaci spojového seznamu, aby na jeho začátku byly cifry nejnižších řádů a na konci cifry nejvyšších řádů. Během sčítání pak nezapomeňte na přenosy do vyššího řádu. Pamatujte na to, že každé z čísel může mít jiný počet cifer.

30. Je možné použít spojové seznamy jak s orientací od nejnižších řádů čísla k nejvyšším (tj. s ciframi uloženými odzadu), tak i s orientací opačnou (tj. cifry uložené odpředu, od nejvyššího řádu k nejnižšímu). V obou případech stačí projít daná čísla pouze jednou.

31. Není-li znám žádný odhad omezující počet cifer sčítaných čísel, je třeba reprezentovat je pomocí spojových seznamů. V prvcích seznamů mohou být uloženy jednotlivé cifry nebo případně pro úsporu paměti skupiny cifer. Pro potřeby sčítání je výhodné zvolit si takovou orientaci spojového seznamu, aby na jeho začátku byly cifry nejnižších řádů a na konci cifry nejvyšších řádů. Během sčítání pak nezapomeňte na přenosy do vyššího řádu.

32. Použijte návod k předchozí úloze.

35. Nejlepší je zařazovat nové prvky se čtenými čísly do vytvářeného seznamu tak, aby byl seznam hned od začátku stále setříděný podle velikosti.

36. Použijte některý z běžných třídicích algoritmů. Nejlépe se zde hodí třídění přímým výběrem nebo přímé zatřiďování.

39. Úlohu je možné řešit jedním průchodem daným seznamem. Zrcadlový obraz řetězce si nejprve vytvořte do druhého seznamu a ten pak vcelku připojte na konec původního seznamu.

44. Násobte běžným způsobem každý člen jednoho činitele každým členem druhého činitele. Před vytvořením nového prvku pro uložení výsledku jednoho takovéhoto elementárního násobení musíte

zkontrolovat, zda ve vytvářeném seznamu, který bude reprezentovat součin mnohočlenů, není již obsažen prvek se stejným exponentem. V takovém případě nezařadíte do seznamu nový prvek, ale jen změníte hodnotu koeficientu v prvku s příslušným exponentem. (Navíc se po takové změně koeficientu může dokonce stát, že nová hodnota koeficientu bude nulová a celý člen budete muset z mnohočlenu odstranit.)

45. Při průchodu stromem si vytvářejte pomocnou frontu odkazů na jednotlivé uzly stromu (v pořadí "po vrstvách"). Frontu realizujte pomocí lineárního spojového seznamu. Každý prvek fronty bude obsahovat záznam tvořený dvěma ukazateli: jeden ukazuje na uzel stromu, druhý ukazatel slouží k propojení prvků ve frontě.

47. V pomocných proměnných si průběžně udržujte dvě dosud největší přečtená čísla. Do vytvářeného stromu vkládejte pouze ta čísla, která tam jistě patří a nebude třeba je později vypouštět.

50. Výpis hodnot uzlů v požadovaném pořadí podle velikosti získáte při běžném "středním" průchodu stromem (tzv. průchod inorder).

51. Použití rekurze nahradíte tím, že si sami naprogramujete zásobník.

52. Použijte rekurzivní proceduru. Vytvořte kořen stromu, uložte do něj hodnotu ((N+1) div 2) a dále vytvořte levý a pravý podstrom s hodnotami uzlů 1 až ((N+1) div 2)-1 v levém podstromu a s hodnotami ((N+1) div 2)+1 až N v pravém podstromu.