Lineární spojový seznam a jeho užití

Michal Žemlička

1 Využití paměti počítače „očima“ aplikace

Při jistém zjednodušení můžeme dělit paměť počítače na dvě základní části:

1. operační paměť – rychlá vnitřní paměť počítače, kde má zejména právě zpracovávané programy a jejich data;
2. vnější paměti – pomalejší úložiště umožňující programy a data uchovávat dlouhodobě.

Zde se budeme věnovat pouze operační paměti. Ta dovoluje přistupovat k libovolně malým kouskům paměti velmi rychle. Také je ale třeba počítat s tím, že její obsah je udržován pouze pokud počítač běží, resp. pokud běží aplikace, jíž je daná část paměti přidělena.

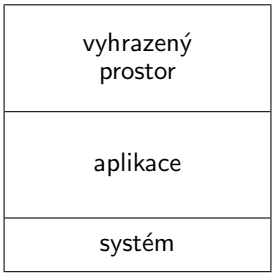
Na operační paměť můžeme pohlížet jako na pole bitů (jednotlivých jedniček a nul) dané šířky (zpravidla mocnina 2 – dnes většinou 64, ale najdou se i počítače s 32, 16, 8, nebo 128 a více) a dané délky (opět mocnina 2 – např. 2^{28}, 2^{16}, ...). Často je nejmenším přímo adresovatelným kouskem skupina 8 bitů (označovaná jako *bajt* – z angl. *byte*). Menší části bývají adresovatelné tak, že je určen bajt, a v něm se určuje poloha a délka dané části.

Tam, kde šířka paměti (resp. přímo zpracovatelného *slova*) je větší než 1 bajt, může být přístup ke do paměti omezen tak, že se smí pracovat je s částmi zarovanými podle velikosti do slova (tedy kdy dvoubajtové kousky smějí začínat jen na sudých adresách, čtyřbajtové jen na adresách dělitelných čtyřmi, atd.). I v případě, že to není zakázáno, umístění dat odporující výše uvedenému doporučení může v důsledku zanemnat, že program poběží pomaleji (bude třeba k datům přistoupit nadvakrát). Proto se může stát, že pokud si necháme určit velikost záznamu, nemusí vždy odpovídat jen součtu jednotlivých položek – překladač jen rozmístil data tak, aby vyhověl výše uvedeným omezením.

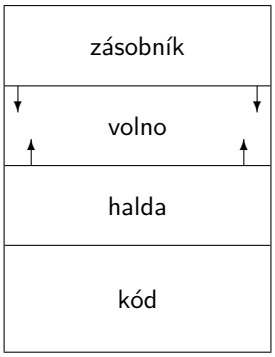
Dnešní systémy zpravidla umožňují aplikacím, aby si mohly udržovat pohled na paměť, jako by byly v počítači samy – tedy pokud se nejedná o aplikace, které jsou od počátku koncipovány na spolupráci s jinými aplikacemi, pak jsou k dispozici mechanismy, jak takové aplikace mohou sdílet zdroje.

Aplikace tak vidí paměť rozdělenou na část, která je jí dostupná a na zbytek adresového prostoru. Aplikaci nedostupné části operační paměti jsou určeny zejména systému a jeho rozšířením – viz obr. 1.

Prostor vymezný aplikaci je dále dělen na části. Rozlišuje se kód a data. Ta jdou dále členěna na statická, automatická a na vyžádání (*halda*[1]). Statická data tvoří globální proměnné a lokální proměnné označené některým z prostředků jazyka označené jako statické. Automatická data tvoří běžné lokální proměnné podprogramů a metod. S nimi jsou ukládány i informace potřebné pro práci s podprogramy (parametry, někdy i návratové hodnoty, ale i adresy, kam je třeba se vrátit). Poslední vymezenou oblastí je prostor pro data „na vyžádání“ – halda. Místo v této oblasti je přidělováno až za běhu programu. O něj si programátor říká přímo (pomocí new), případně nepřímo (některé předpřipravené podprogramy či metody mohou o paměť zde žádat samy). Rozložení částí paměti je patrné z obr. 2.



Obrázek 1: Typické umístění aplikace v adresovém prostoru



Obrázek 2: Typické rozdělení paměti aplikace

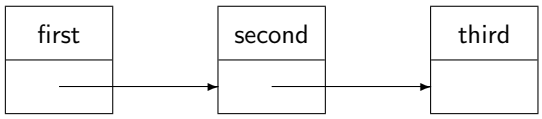
2 K čemu jsou lineární spojové seznamy

Lineární spojový seznam je datová struktura s různým využitím.

[1] Pozor, zde se nejedná o datovou strukturu halda, ale o prostor, kde jsou data uložena „jak to vyšlo“.

3 Představa

Lineární spojový seznam je tvořen posloupností uzlů obsahujících data. Každý uzel obsahuje jak uživatelská data, tak i pomocný údaj – odkaz na další prvek, případně informaci, že tento je poslední (formou speciálního odkazu).



Obrázek 3: Lineární spojový seznam s třemi prvky

Na lineární spojový seznam můžeme pohlížet jako na lineární i jako na rekurzivní strukturu. Lze tak operace se tout strukturou provádět buď rekurzivně, nebo sekvenčně.

Některé operace nad lineárním spojovým seznamem jsou jednoduché a přímočaré. Na to abychom zjistili, zda je seznam prázdný, stačí, abychom odkaz na řetízek uzlů porovnali se speciální hodnotou `None`:

Chceme-li lineární spojový seznam implementovat, bude se hodit deklarace uzlu. Při vytváření nového uzlu si necháváme otevřenou možnost určit jak obsah uzlu, tak i následující uzel:

```
class Uzel:
    """ uzel jednosmerného lineárního
        spojového seznamu """

    def __init__(self, prvek=None, dalsi=None):
        self.info = prvek
        self.next = dalsi
```

Nyní se můžeme pokusit deklarovat i seznam (zatím jen s konstruktorem):

```
class Seznam:
    """ lineární spojový seznam """

    def __init__(self):
        self.seznam = None
```

Pro ladění je výhodné, je-li jednou z prvních metod tisk, kdy procházíme seznamem prvek po prvku a tiskneme tam nalezenou hodnotu:

```
    def tisk():
        print('[ ',end='')
        kde_jsem = seznam    # odkaz na
     # cely retizek
        while kde_jsem != None:
            print(kde_jsem.info,end=' ')
            kde_jsem = kde_jsem.next
        print(']')
```

Asi nejjednodušší metodou je test prázdnosti seznamu:

```
def jePrazdny():
    return  seznam == None
```

Máme-li vložit prvek na začátek seznamu, stačí požádat o nový uzel, vložit do něj určená data a odkaz na původní řetízek uzlů a následně uchovat odkaz na tento nový uzel jako odkaz na celý řetízek uzlů reprezentující seznam (obr. 4).

Při implementaci můžeme s výhodou využít konstruktoru s parametry, čímž se kód může výrazně zjednodušit:

```
def vloz_na_zacatek(x):
    seznam = Uzel(x,seznam)
```

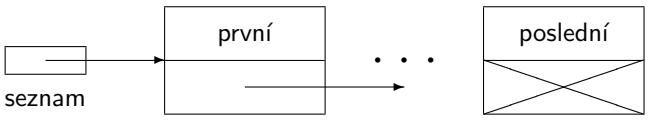
Příkladem může být vkládání prvku na konec seznamu: můžme jej řešit rekurzivně, nebo sekvenčně. Při rekurzivním řešíme prázdný seznam, a neprázdný seznam. V prvním případě vytvoříme prvek a vrátíme jej jako seznam, zatímco ve druhém případě zavoláme vložení téhož prvku na „další“. Při sekvenčním průchodu je také třeba rozlišit, zda dostaneme prázdný seznam, či seznam, který již nějaké uzly obsahuje.

Při vkládání je třeba měnit ukazatel na danou strukturu. To lze řešit voláním odkazem (je-li k dispozici), nebo tak, že podprogram vrací aktualizovaný odkaz, ktarý pak uložíme.

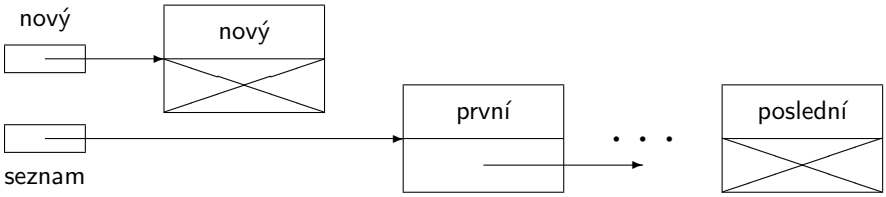
Zkuste si navrhnout sami, kontrolní řešení vyvěsím, až uvidím vaše pokusy.

Zkuste si nadefinovat třídu LinkedList s těmito operacemi:

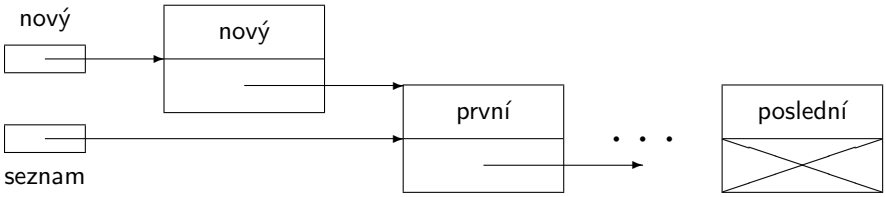
- vytvoření (`__init__`)
- vypsání celého seznamu (print)
- vložení prvku na začátek (insert)
- odebrání prvku ze začátku seznamu (fetch)
- vložení prvku na konec (extend)
- odebrání prvního výskytu prvku s danou hodnotou (removefirst)
- je seznam prázdný? (isempty)



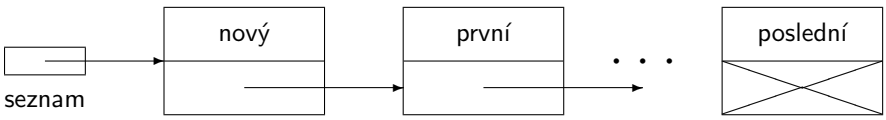
a) původní seznam



b) nový prvek a původní seznam



c) nový prvek ukazuje na původní seznam



d) lineární spojový seznam s novým prvkem na začátku

Obrázek 4: Vkládání prvku na začátek lineárního spojového seznamu