Dynamický datový typ zásobník

1 Dynamický datový typ

Běžné datové typy, se kterými jsme doposud pracovali, popisují proměnné, které sice mohou měnit své hodnoty, ale ne svou velikost. Každá proměnná typu struktura má v průběhu života programu stále stejný počet složek. Stejně tak u pole máme po dobu života programu stejný počet prvků. Takovým typům se říká *statické datové typy*.

Dynamický datový typ se vyznačuje tím, že v průběhu života programu dynamicky mění počet svých prvků. Prvky do něj můžeme přidávat, ale také je z něj můžeme ubírat.

Častou implementací jsou struktury tvořené pomocí prvků provázaných ukazateli. Tento princip jsme si ukázali na minulém cvičení.

Takový datový typ si musíme vytvořit sami. Uděláme to tak, že deklarujeme datový typ pro jeden prvek a poté množinu operací – podprogramů, které budou s tímto datovým typem pracovat. V našem případě budeme chtít především prvek vyrobit a inicializovat, provázat s ostatními prvky (přidat) a vyvázat od ostatních prvků (odebrat).

Takovémuto datovému typu, který rozšiřuje nabídku zabudovaných typů, říkáme obecně *abstraktní datový typ - ADT*.[1]

2 Prvek

V tomto cvičení se budeme věnovat implementaci ADT *zásobník*. Prvky zásobníku budou tvořit lineární spojový seznam. Strukturu pro takový prvek jsme si už ukazovali minule. Prvky jsou provázané pomocí ukazatelů, takže každý prvek musí obsahovat složku typu ukazatel na další prvek stejného typu. Pro jednoduchost opět ponecháme složku nesoucí hodnotu jednoduchého typu. S ohledem na úlohu, kterou v rámci tohoto cvičení budete řešit, bude teď hodnota prvku typu `char`.

```
typedef struct _prvek Tprvek;

struct _prvek {
  char hodnota;
  Tprvek * dalsi;
};
```

Nový prvek se bude vždy alokovat dynamicky, vyrobíme si tedy funkci pro vyrábění a správnou inicializaci.

Správná inicializace prvků s ukazateli je velice důležitá, protože pokud se do dynamické datové struktury dostane neinicializovaný ukazatel (zde na další prvek), je to zaručený recept na katastrofu.

```
Tprvek* prvekNovy(char hodnota) {
  Tprvek* novy=malloc(sizeof(Tprvek));
  if (novy != NULL) {
    novy->hodnota = hodnota;
    novy->dalsi = NULL;
  }
  return novy;
}
```

Všimněte si, že funkce vrací ukazatel. V případě, že se alokace nepovede, vrací prázdný ukazatel NULL (protože jej vrací malloc). Kdekoli se tato funkce použije, musí se hned vzápětí otestovat, jestli je získaný ukazatel platný.

3 Zásobník

ADT zásobník vytvoříme pomocí datového typu struktura s ukazatelem na jeden konec seznamu prvků, tzv. vrchol. Dále pomocí množiny podprogramů, které s touto strukturou umožní uživateli pracovat.

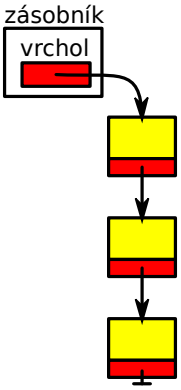
```
typedef struct {
  Tprvek* vrchol;
  int vyska;
} Tzasobnik;
```

Složka pro uchování výšky zásobníku (počtu prvků) není povinná, ale někdy se hodí a její implementace je triviální.

1 Dynamické datové typy jsou speciálním případem abstraktního datového typu. Jako abstraktní datový typ můžeme označit i typ, který vůbec není dynamický, ale je také dodatečně vyrobený. Příkladem může být třeba komplexní číslo realizované jako struktura s reálnou a imaginární složkou, plus množina funkcí umožňující práci s komplexními čísly.

Prvky se budou do zásobníku přes vrchol jak vkládat, tak i odebírat. Díky tomu zásobník realizuje svou charakteristickou vlastnost, vyjádřenou zkratkou LIFO (last in, first out). Zásobník *otáčí pořadí* prvků. Vybíráme je totiž v opačném pořadí, než v jakém byly vloženy. Zásobník se v algoritmech využívá častěji pro tuto charakteristickou vlastnost, nežli proto, že jde o dynamický datový typ a může tedy obsahovat velké množství prvků.

Graficky znázorněná představa zásobníku může vypadat takto:



3.1 Inicializace struktury

Na začátku je potřeba provést inicializaci samotné struktury. Během ní je klíčové nastavit ukazatel na vrchol na NULL, což bude tvořit prázdný seznam.

Inicializace struktury nemusí nutně znamenat dynamickou alokaci. Hlavička takové funkce může vypadat takto.

```
Tzasobnik zasInit(void);
```

Pak bude pravděpodobně nutné zásobník do dalších podprogramů předávat odkazem pomocí referenčního operátoru:

```
int main(void) {
  Tzasobnik z = zasInit();
  zasPush(&z, 'a');
  zasPush(&z, 'b');
  // ...
  return 0;
}
```

Druhá, preferovaná varianta bude *dynamicky alokovat* samotnou pomocnou strukturu. Budeme s ní tedy vždy pracovat přes ukazatel, který získáme při alokaci pomocí funkce `malloc`.

```
Tzasobnik* zasInitD(void){
  Tzasobnik* z = malloc(
                   sizeof(Tzasobnik));
  // + test + inicializace složek
  return z;
}
```

V takovém případě bude nutné tuto strukturu (typ Tzasobnik) uvolnit pomocí funkce `free` po odstranění všech prvků zásobníku na konci programu. Nesmíte na to zapomenout!

Do podprogramů budeme zásobník zase předávat odkazem, ale protože teď již půjde o ukazatel, nebude potřeba používat referenční operátor.

Je nutné si ale uvědomit, že zásobník nejde vyrobit jako proměnná typu ukazatel, aniž by byla provedena alokace. Pak by totiž nebyla vytvořena pomocná struktura, ale pouze proměnná pro (zřejmě neinicializovanou) adresu. Jakákoli práce s takovou proměnnou by téměř jistě vedla k havárii.

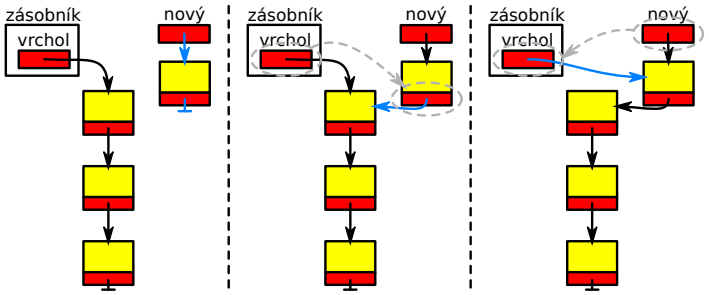
```
int main(void) {
  Tzasobnik* z; // tady se nealokuje!
  zasPush(z, 'a'); // tady to havaruje
  zasPush(z, 'b');
  // ...
return 0;
}
```

Použitelná a preferovaná verze:

```
int main(void) {
  Tzasobnik* z = zasInitD();
  zasPush(z, 'a');
  zasPush(z, 'b');
  // ...
  zasFree(z);
return 0;
}
```

3.2 Operace *push*

Tato operace vkládá novou hodnotu na vrchol zásobníku. Nejprve alokuje nový prvek a inicializuje jej zadanou hodnotou. Pak provede navázání vrcholu (nejvyššího prvku v seznamu) za nový prvek. Nakonec nastaví nový prvek jako nový vrchol.



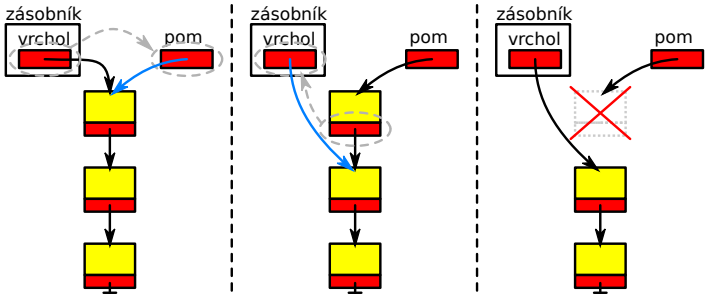
Protože operace může selhat, když se nepovede alokace, bude vracet logickou hodnotu, která půjde snadno testovat.

```
bool zasPush(Tzasobnik *z, char x);
```

Jednotlivé kroky této operace je potřeba dělat pečlivě ve správném pořadí. V opačném případě dojde snadno k tzv. podřezání větve, kdy ztratíte ukazatel na zbytek seznamu. Taková chyba je velmi častá a je zákeřná v tom, že vás na ní překladač nemůže upozornit varovným hlášením. Program totiž může být syntakticky naprosto v pořádku. Toto bude programátorská, ne syntaktická chyba.

3.3 Operace *pop*

Tato operace odebírá prvek z vrcholu zásobníku a vrací hodnotu, kterou nesl. V této operaci je potřeba nastavit pomocnou proměnnou typu ukazatel na původní vrchol zásobníku. Pak se provede přemostění, kdy složka vrchol začne ukazovat na následující prvek. Pak je teprve možné uvolnit prvek odkazovaný pomocnou proměnnou. Samozřejmě nesmíme zapomenout zkopírovat hodnotu nesenou tímto prvkem ještě před jeho uvolněním.



Operace se opět nemusí povést, proto bude vracet logickou funkční hodnotu. V tomto případě se operace nepovede, když se pokusíme získat hodnotu z prázdného zásobníku.

```
bool zasPop(Tzasobnik* z, char* x);
```

Všimněte si, že hodnota nesená odebíraným prvkem se teď vrací přes parametr předávaný odkazem.

3.4 Uvolnění zásobníku

Tato operace není nutná, pokud je zásobník vyráběn jako struktura (verze `zasInit`). Paměť na hromadě se uvolní prostým opakovaným voláním operace `zasPop`. Většina algoritmů, které používají zásobník navíc přirozeně skončí s prázdným zásobníkem.

U (preferované) varianty s dynamicky alokovanou pomocnou strukturou je nutné tuto strukturu nakonec také uvolnit. Musí se to ale dělat nad prázdným zásobníkem, jinak dojde k paměťovému úniku – už se nepůjde dostat ke zbylým prvkům seznamu.

4 Úloha – závorkový algoritmus

Že zásobník není samoúčelná struktura si ukážeme na úloze, kterou pomocí zásobníku implementujete velice snadno. Naproti tomu bez něj je prakticky neřešitelná.[2]

Nutnou podmínkou pro implementaci této úlohy je mít implementovaný funkční zásobník, jehož prvky nesou hodnoty typu char.

Zadání zní takto:

- Vytvoř program, který zpracuje soubor obsahující libovolný text, v němž se vyskytují závorky různých typů
  - kulaté (),
  - hranaté [],
  - složené {},
  - úhlové <>.
- Ověř, zda jsou závorky v textu
  - správně spárovány,
  - pár vždy začíná levou a končí pravou závorkou,
  - jsou správně vnořené, kde
    - (([()]{<>})) – tohle je ok,
    - [(]) – tohle je špatně.
- Použij pro řešení svou implementaci zásobníku a na konci programu vypiš hlášení o tom, zda jsou v zadaném souboru závorky v pořádku, nebo ne.

2 Tato úloha není jediná. Existuje celá skupina algoritmů, které jsou řešitelné pouze s pomocí zásobníku. Tato úloha je ale natolik typická, že je škoda si ji nevyzkoušet.