8. Abstraktní datový typ - zásobník, fronta

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přehled témat

- Část 1 – Pole v dynamické paměti
  - 2D pole v dynamické paměti
  - Struktura s ukazatelem na pole
- Část 2 – Abstraktní datový typ
  - Zásobník
  - Fronta
  - Množina

Datové struktury

Na datové struktury se lze dívat ze dvou pohledů:

- Z pohledu vnitřní realizace
  - **pole** – posloupnost proměnných stejného typu,
  - **záznam** – struct, skupina proměnných různého typu,
  - **objekt** – datovou strukturu tvoří pouze atributy instancí, atributy třídy jsou globální proměnné pouze formálně přidružené k dané třídě, proto se ukládají obvykle nezávisle na instancích třídy,
  - **řetězec** – string, posloupnost znaků.
- Podle logiky přístupu k jednotlivým položkám
  - **strom**,
  - **seznam**,
  - **množina**,
  - **zásobník**,
  - **fronta**.

Dva pohledy na data

Abstraktní

- operace, které budu s daty provádět
- co musí operace splňovat
- například množina: ulož, najdi, vymaž
- tento předmět

Výhody: snadný vývoj, jednodušší přemýšlení o problémech

Riziko: svádí k ignorování efektivity

Implementační

- jak jsou data uložena v paměti
- jak jsou operace implementovány
- například binární vyhledávací strom

Další kritéria klasifikace datových struktur

- Přístup k položkám struktury
  - **sekvenční** – např. jednosměrný seznam, uspořádaná posloupnost položek, kde každá položka obsahuje odkaz na následující položku,
  - **přímý** – např. dynamické pole, položky jsou přístupné pomocí indexů.
- Organizace dat
  - **lineární** – v řadě za sebou, pole, seznam, zásobník a fronta,
  - **nelineární** – binární strom.
- Fyzické vymezení prostoru
  - **statické struktury** – mají svou velikost přesně a neměně určenou v okamžiku překladu zdrojového kódu,
  - **dynamické struktury** – mohou měnit počet položek za běhu programu - seznam, dynamické pole.

Část I

Pole v dynamické paměti

I. Pole v dynamické paměti

Umístění pole v paměti

Pole alokované v zásobníku (stack)

- Má neměnnou velikost
- Sám zásobník je omezen co do velikosti
- Je lokální proměnná, při neopatrnosti zanikne
- V zásobníku je dobrá správa paměti (= bez starostí)

Pole v dynamické paměti (heap/halda)

- V tomto segmemtu paměti existuje až do explicitní dealokace
- Správa pamět přiděluje, ale nedealokuje
- Dochází k fragmentaci (stejný segment je dostupný více procesům)
- Lze měnit velikost pole - platí i pro 2D pole?

2D pole na haldě jako ekvivalent pole v zásobníku

- Souvislý blok paměti
- Řádky jsou datového typu ukazatel na N-prvkové pole

```
1  /* datovy typ popisujici radek */
2  int (*p)[3];
4  /* alokujeme souvisly blok 6 prvku */
5  /* bloky jsou usporadany jako dve pole int (*)[3] */
6  p = calloc (6, sizeof(int));
8  /* dealokace jednoho bloku */
9  free(p);
```

- Lze přidávat řádky
- Nelze měnit velikost řádků

2D pole jako pole ukazatelů na řádky

- Pole ukazatelů na jednotlivé řádky pole

```
1  /* pole ukazatelu v zasobniku */
2  int *p[2];
4  /* radky 2D pole na halde */
5  p[0] = calloc (3, sizeof(int));
6  p[1] = calloc (3, sizeof(int));
8  /* dealokace obou radku */
9  free(p[0]);
10 free(p[1]);
```

- Nelze měnit počet řádků
- Délka řádků může být různá a mohly by být i různého datového typu. **Jak?**

2D pole celé na haldě

- Ukazatel na ukazatel

```
1  /* pole ukazatelu na halde */
2  int ** p;
3  p = malloc (2*sizeof(int *));
5  /* jednotlive radky */
6  p[0] = calloc (3, sizeof(int));
7  p[1] = calloc (3, sizeof(int));
9  /* dealokovat je treba nejprve radky a pak pole ukazatelu */
10 free(p[0]);
11 free(p[1]);
12 free(p);
```

- Lze měnit počet řádků
- Délka řádků může být různá

I. Pole v dynamické paměti

2D pole v dynamické paměti

Struktura s ukazatelem na pole

Struktura s ukazatelem na pole

- Téměř vždy při práci s polem potřebujeme informaci o alokované velikosti
- Velmi často tuto informaci z různých důvodů nemáme
- Řešením může být struktura, která obsahuje metadata pole

```
1  typedef struct
2  {
3    int delka;
4    int *data;
5  } pole;
```

```
1  void init (pole * x, int y) {
2    x->delka = velikost;
3    x->data = calloc (y ,sizeof(int));
4  }
6  int main() {
7    pole a;
8    init (&a, 5);
9  }
```

Co zde chybí?

Část II

Abstraktní datový typ

Datové struktury a abstraktní datový typ

Datová struktura (typ) je množina dat a operací s těmito daty.
Abstraktní datový typ formálně definuje data a operace s nimi.

- Množina druhů dat (hodnot) a příslušných operací
  - jsou přesně specifikovány a to nezávisle na konkrétní implementaci
- Definujeme rozhraní a operace, které rozhraní poskytuje
  - Konstruktor vracející odkaz (na strukturu nebo objekt)

    Procedurální i objektově orientovaný přístup.
  - Operace, které akceptují odkaz na argument (data)
  - Operace, které mají přesně definovaný účinek na data
- Nejpoužívanější abstraktní datové typy:
  - Zásobník (Stack)
  - Fronta (Queue)
  - Pole (Array)
  - Tabulka (Table)
  - Seznam (List)
  - Strom (Tree)
  - Množina (Set)

Abstraktní datový typ

- Počet datových položek může být
  - Neměnný – statický datový typ
    - počet položek je konstantní
    - např. pole, řetězec, struktura
  - Proměnný – dynamický datový typ
    - počet položek se mění v závislosti na provedené operaci (vložení / vyjmutí položky)
- Typ položek (dat):
  - Homogenní – všechny položky jsou stejného typu
  - Nehomogenní – položky mohou být různého typu
- Existence bezprostředního následníka
  - Lineární – existuje bezprostrední následník prvku, např. pole, fronta, seznam, ...
  - Nelineární – neexistuje přímý jednoznačný následník, např. strom

II. Abstraktní datový typ

Zásobník

Fronta

Množina

Zásobník

- Zásobník je dynamická datová struktura umožnující vkládání a odebírání hodnot tak, že naposledy vložená hodnota se odebere jako první

LIFO – Last In, First Out

- Základní operace:
  - `push()` – vložení hodnoty na vrchol zásobníku
  - `pop()` – odebrání hodnoty z vrcholu zásobníku
  - `empty()` – test na prázdnost zásobníku
- Další operace nad zásobníkem mohou být
  - `top()` / `peek()` – čtení hodnoty z vrcholu zásobníku
  - `search()` – vrátí pozici prvku v zásobníku (pokud tam je)
  - `size()` – aktuální počet prvků v zásobníku (zpravidla není potřeba)

Zásobník – rozhraní

- Zásobník můžeme definovat rozhraním (funkcemi), bez konkrétní implementace

```
int stack_push(void *value, void **stack);
void * stack_pop(void **stack);
int stack_empty(void **stack);
void * stack_peek(void **stack);
void stack_init(void **stack); // inicializace ADT
void stack_delete(void **stack); // smazání ADT
void stack_free(void **stack); // uvolnění paměti
```

- V tomto případě používáme obecný zápis s ukazatelem typu `void`
- Je plně v režii programátora (uživatele) implementace, aby zajistil správné chování programu
  - Alokaci proměnných a položek vkládaných do zásobníku
  - A také následné uvolnění paměti
- Do zásobníku můžeme dávat rozdílné typy, musíme však zajistit jejich správnou interpretaci

Zásobník – implementace

- Součástí ADT není volba konkrétní implementace.
- Zásobník lze implementovat např.
  - Polem fixní velikosti (definujeme chování při zaplnění)
  - Polem s měnitelnou velikostí (realokace)
  - Spojovým seznamem

- Struktura popisující vlastnosti zásobníku

```
1 typedef struct {
2   void **stack; // array of void pointers
3   int count;
4 } stack_t;
```

- Pomocné funkce pro inicializaci a uvolnění paměti

```
1 void stack_init(stack_t **stack);
2 void stack_delete(stack_t **stack);
3 void stack_free(stack_t *stack);
```

- Základní operace se zásobníkem

```
1 int stack_empty(const stack_t *stack);
2 int stack_push(void *value, stack_t *stack);
3 void* stack_pop(stack_t *stack);
4 void* stack_peek(const stack_t *stack);
```

- Inicializace struktury popisující zásobník
- Maximální velikost zásobníku definuje hodnota makra

```
1  #ifndef STACK_SIZE
2  #define STACK_SIZE 5
3  #endif
5  #define STACK_OK    0
6  #define STACK_MEMFAIL 100
8  void stack_init (stack_t **stack) {
9    *stack = (stack_t*) malloc(sizeof(stack_t));
10   (*stack)->stack = (void**) malloc(sizeof(void*)*STACK_SIZE);
11   (*stack)->count = 0;
12 }
```

- Vyprázdnění zásobníku

```
1  void stack_free(stack_t *stack) {
2    while(!stack_is_empty (stack)) {
3      void *value = stack_pop (stack);
4      free (value);
5    }
6  }
```

- Uvolnění paměti alokované zásobníkem

```
1  void stack_delete(stack_t **stack) {
2    stack_free(*stack);
3    free((*stack)->stack);
4    free(*stack);
5    *stack = NULL;
6  }
```

- Pokud je v zásobníku volné místo, vložím položku

```
1  int stack_push(void *value, stack_t *stack) {
2    int ret = STACK_OK;
3    if (stack->count < MAX_SIZE) {
4      stack->stack[stack->count++] = value;
5    } else {
6      ret = STACK_MEMFAIL;
7    }
8    return ret;
9  }
```

- Pokud jsou v zásobníku položky, vyjmi poslední vloženou (destruktivní)

```
1  void * stack_pop(stack_t *stack) {
2    return stack->count > 0 ? stack->stack[--(stack->count)] : NULL;
3  }
```

- Vrať položku na vrcholu zásobníku (nedestruktivní)

```
1  void* stack_peek (const stack_t *stack) {
2    return stack_empty(stack) ? NULL : stack->stack[stack->count-1];
3  }
```

- Pokud je hloubka zásobníku nenulová, vrať kladné číslo (= `True`)

```
1  int stack_empty(const stack_t *stack) {
2    return stack->count == 0;
3  }
```

```
1  int * val;
2  stack_t * zasobnik;
3  stack_init(&zasobnik);
5  val = malloc(sizeof(int));
6  *val = 10;
7  stack_push(val, zasobnik);
9  val = malloc(sizeof(int));
10 *val = 20;
11 stack_push(val, zasobnik);
13 while(stack_empty(zasobnik) == 0) {
14   val = stack_pop(zasobnik);
15   printf("-> %i\n", *val);
16 }
18 stack_delete(&zasobnik);
```

8.2 Převod z dekadické soustavy

```
1  #define BASE 16
2  char chars[] = "0123456789ABCDEF";
4  stack_t * zasobnik;
5  stack_init(&zasobnik);
6  int dek = 101;  // cislo, ktere budeme prevadet
8  do {
9    int * val = malloc(sizeof(int));
10   *val = dek % BASE;
11   stack_push(val, zasobnik);
12 } while (dek /= BASE, dek > 0);
14 while(stack_empty(zasobnik) == 0) {
15   printf("%c", chars[*((int *)stack_pop(zasobnik))]);
16 }
18 stack_delete(&zasobnik);
```

II. Abstraktní datový typ

Zásobník

Fronta

Množina

Fronta

- Dynamická datová struktura, kde se odebírají prvky v tom pořadí, v jakém byly vloženy

FIFO - First In, First Out

- Implementace
  - Pole
    - Pamatujeme si pozici začátku a konce fronty v poli
    - Pozice cyklicky rotují (modulo velikost pole)
  - Spojový seznam
    - Pamatujeme si ukazatel na začátek a konec fronty
    - Přidáváme na začátek (head) a odebíráme z konce
    - Přidáváme na konec a odebíráme ze začátku (head)
- Z hlediska vnějšího (ADT) chování fronty na vnitřní implementaci nezáleží.

Operace fronty

- Základní operace nad frontou jsou vlastně identické jako pro zásobník
  - `push()` – vložení prvku na konec fronty
  - `pop()` – vyjmutí prvku z čela fronty
  - `isEmpty()` – test na prázdnost fronty
- Další operace mohou být
  - `peek()` – čtení hodnoty z čela fronty
  - `size()` – vrátí aktuální počet prvků ve frontě
- Hlavní rozdíl je v operacích `pop()` a `peek()`, které vracejí nejdříve vložený prvek do fronty.

Na rozdíl od zásobníku, u kterého je to poslední vložený prvek.

Fronta – definice rozhraní

```
1  typedef struct {
2    ...
3  } queue_t;
5  void queue_delete (queue_t **queue);
6  void queue_free (queue_t *queue);
7  void queue_init (queue_t **queue);
8  int queue_push (void *value, queue_t *queue);
9  void* queue_pop (queue_t *queue);
10 int queue_is_empty (const queue_t *queue);
11 void* queue_peek (const queue_t *queue);
```

8.3 Fronta – implementace polem

- Implementace velmi podobná zásobníku v poli
- Zásadní změna ve funkci `queue_push()`

```
1  int queue_push (void *value, queue_t *queue) {
2    int ret = QUEUE_OK;
3    if (queue->count < MAX_QUEUE_SIZE) {
4      queue->queue[queue->end] = value;
5      queue->end = (queue->end + 1) % MAX_QUEUE_SIZE;
6      queue->count += 1;
7    } else {
8      ret = QUEUE_MEMFAIL;
9    }
10   return ret;
11 }
```

- Ukládáme na konec (proměnná `end`), která odkazuje na další volné místo (pokud `count < QUEUE_SIZE`)

- Funkce `queue_pop()` vrací hodnotu na indexu `start` tak jako metoda `queue_peek()`

```
1  void* queue_pop (queue_t *queue) {
2    void* ret = NULL;
3    if (queue->count > 0) {
4      ret = queue->queue[queue->start];
5      queue->start = (queue->start + 1) % MAX_QUEUE_SIZE;
6      queue->count -= 1;
7    }
8    return ret;
9  }
11 void* queue_peek (const queue_t *queue) {
12   return queue_is_empty(queue) ? NULL : queue->queue[queue->start];
13 }
```

II. Abstraktní datový typ

Zásobník

Fronta

Množina

Množina

- Množina je datová struktura umožnující vkládání a odebírání hodnot tak, že každá hodnota se ve struktuře vyskytuje pouze jednou
- Základní operace:
  - `add()` – vložení hodnoty do množiny
  - `delete()` – odebrání hodnoty z množiny
  - `contains()` – test přítomnosti hodnoty v množině
- Další operace nad zásobníkem mohou být
  - `size()` – aktuální počet prvků v množině (zpravidla není potřeba)

Rozptylovací tabulka (hash table)

Rozptylovací tabulka – implementace množiny / asociativního pole

\+ velmi rychlé vkládání i hledání, $O(1)$

– neudržuje uspořádání (hledání maxima/minima)

– méně efektivní využití paměti

Co je to hash?

- *hash* – rozemlít, rozsekat, sekané maso, haše, . . . hašiš
- *hash function* – rozptylovací/transformační/hašovací/hešovací/ funkce:
  objekt → celé číslo
- *hash / fingerprint* – haš/heš, otisk

Základní myšlenky a vlastnosti

- pole **m** přihrádek *(slots)* pro ukládání položek.
- položka *(item)* = klíč *(key)* + hodnota *(value)*
- klíč je unikátní
- **rozptylovací funkce** *(hash function)*: φ: klíč $\rightarrow$ číslo přihrádky $0 \ldots m-1$
- více položek v jedné přihrádce = **kolize** *(collision/clash)*
- operace jsou rychlé, protože
  - víme, v které přihrádce hledat
  - v každé přihrádce je jen omezený počet položek

Příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m =$ `x % m`

Vložíme čísla

x	54	26	93	17	77	31
$\varphi(x)$	10	4	5	6	0	9

Vznikne tabulka a určíme indexy

0	1	2	3	4	5	6	7	8	9	10
77				26	93	17			31	54

Relativní naplnění: $\lambda = 6/11 \approx 0.54$

Rozptylovací funkce – (hash function)

Nutné vlastnosti

- *Stejné* klíče musí mít stejný otisk – $x = y \Rightarrow \varphi(x) = \varphi(y)$
- Neměnnost / nenáhodnost / konstantnost / opakovatelnost

Požadované vlastnosti

- Rychlost výpočtu
- *Různé* klíče mají mít pokud možno různý otisk – $x \neq y \Rightarrow$ velká $P\big[\varphi(x) \neq \varphi(y)\big]$
  - každý klíč jiný otisk = *perfect hashing*
  - rovnoměrné využití všech přihrádek
  - pravděpodobnost zvolení konkrétní přihrádky $1/m$ (i pro strukturované vstupy)
  - malé množství kolizí

Kvalitu lze ověřit experimentálně.
Souvislost s kryptografií a náhodnými čísly.

Velikost rozptylovací tabulky

- Vhodná velikost je prvočíselná – např. $11, 103, 1009 \ldots$
  - Jinak riziko kolizí pokud $\varphi(x) \in \{k, 2k, 3k, \ldots\}$
- Dynamická realokace:
  - pokud se tabulka naplní ($\lambda > \lambda_{\mathsf{max}}$) — vytvoříme větší tabulku ($m' \approx 2m$)
  - pokud se tabulka vyprázdní ($\lambda < \lambda_{\mathsf{min}}$) — vytvoříme menší tabulku ($m' \approx m/2$)

Možné hodnoty $m_0 = 11$, $\lambda_{\mathsf{max}} = 0.75$, $\lambda_{\mathsf{min}} = 0.25$.

Řešení kolizí

Co když dvě položky mají stejný otisk?

Zřetězení

- Každá přihrádka je seznam *(nebo pole)*.
- Zaplnění λ může být > 1.

Otevřené adresování

- Kapacita přihrádky je 1.
- Pokud je přihrádka $m_0 = \varphi(x)$ obsazená, zkusíme jinou $(m_1, m_2, \ldots)$
  - Lineární zkoušení *(linear probing)* – zkusíme $m_i = m_0 + i$.
  - Kvadratické zkoušení *(quadratic probing)* – zkusíme $m_i = m_0 + ai^2 + bi$, např. $a = 1$, $b = 0$.
  - Dvojité rozptylování *(double hashing)* – zkusíme $m_i = m_0 + i\psi(x)$.
- Menší režie než zřetězení.
- Zaplnění λ nesmí být velké (≈ 0.7).
- Rozptylovací funkce nesmí vytvářet shluky.

Otevřené adresování – příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 31:

0	1	2	3	4	5	6	7	8	9	10
77				26	93	17			31	54

Otevřené adresování – příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 44:

0	1	2	3	4	5	6	7	8	9	10
77	44			26	93	17			31	54

Otevřené adresování – příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 55:

0	1	2	3	4	5	6	7	8	9	10
77	44	55		26	93	17			31	54

Otevřené adresování – příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 20:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

Otevřené adresování – příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 20:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

‘Prázdné’ položky = speciální hodnota.

Implementace např. *Problem Solving with Algorithms and Data Structures*
https://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html

Mazání položek

- Zřetězení — smažeme ze seznamu přihrádky.
- Otevřené adresování — smazané položky označíme speciální hodnotou ‘přeskoč’.
- Mazání často není potřeba