9. Spojové struktury. Abstraktní datový typ.

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přehled témat

- Část 1 – Abstraktní datový typ
  - Datové struktury a abstraktní datový typ
  - Zásobník
  - Fronta
- Část 2 – Spojové struktury
  - Spojový seznam
  - Kruhový spojový seznam
  - Obousměrný spojový seznam

Část I

Abstraktní datový typ

I. Abstraktní datový typ

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

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

I. Abstraktní datový typ

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_is_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_is_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);
```

- `stack_init()` inicializuje strukturu
- Maximální velikost zásobníku definuje hodnota makra

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

- `stack_free()` uvolní paměť vložených položek v 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  }
```

- `stack_delete()` kompletně uvolní paměť alokovanou zásobníkem

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

```
1  int stack_push (void *value, stack_t *stack) {
2    int ret = STACK_OK;
3    if (stack->count < MAX_STACK_SIZE) {
4      stack->stack[stack->count++] = value;
5    } else {
6      ret = STACK_MEMFAIL;
7    }
8    return ret;
9  }
11 void* stack_pop (stack_t *stack) {
12   return stack->count > 0 ? stack->stack[--(stack->count)] : NULL;
13 }
```

```
1  void* stack_peek (const stack_t *stack) {
2    return stack_is_empty (stack) ? NULL : stack->stack[stack->count -
       1];
3  }
5  int stack_is_empty (const stack_t *stack) {
6    return stack->count == 0;
7  }
```

I. Abstraktní datový typ

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);
```

9.2 Fronta – implementace polem 1/2

- 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`)

9.2 Fronta – implementace polem 2/2

- 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 }
```

Část II

Spojový seznam

II. Spojový seznam

Ukládání datových struktur do paměti

- V programech je velmi běžný požadavek na uchování seznamu (množiny) prvků (proměnných/struktur)
- Základní kolekce je pole
  - Jedná se o kolekci položek (proměnných) stejného typu
  - \+ Umožňuje jednoduchý přístup k položkám indexací prvku

    Položky jsou stejného typu (velikosti).
  - – Velikost pole je určena při vytvoření pole
    - Velikost (maximální velikost) musí být známa v době vytváření
    - Změna velikost v podstatě není přímo možná

      Nutné nové vytvoření (alokace paměti), resp. realloc.
    - Využití pouze malé části pole je mrháním paměti
  - V případě řazení pole přesouváme položky
    - Vložení prvku a vyjmutí prvku vyžaduje kopírování

Seznam

- Seznam (proměnných nebo objektů) patří mezi základní datové struktury

Základní ADT – Abstract Data Type

- Seznam zpravidla nabízí sadu základních operací:
  - `insert()` – vložení prvku
  - `remove()` – odebrání prvku
  - `index_of()` – vyhledání prvku
  - `size()` – aktuální počet prvku v seznamu
- Implementace seznamu může být různá:
  - Pole
    - Indexování je velmi rychlé
    - Vložení prvku na konkrétní pozici může být pomalé

Nová alokace a kopírování.

  - Spojové seznamy

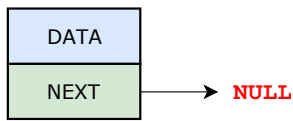
Spojový seznam

- Datová struktura realizující seznam dynamické délky
- Každý prvek seznamu obsahuje
  - Datovou část (hodnota proměnné / objekt / ukazatel na data)
  - Odkaz (ukazatel) na další prvek v seznamu

NULL v případě posledního prvku seznamu.

- První prvek seznamu se zpravidla označuje jako **head** nebo **start**.

Realizujeme jej jako ukazatel odkazující na první prvek seznamu.



Základní operace se spojovým seznamem

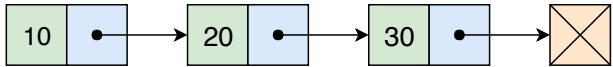
- Vložení prvku
  - Předchozí prvek odkazuje na nový prvek
  - Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje

  Obousměrný spojový seznam.
- Odebrání prvku
  - Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
  - Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek
- Základní implementací spojového seznamu je tzv.

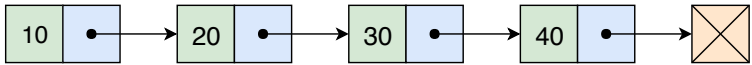
jednosměrný spojový seznam

Jednosměrný spojový seznam

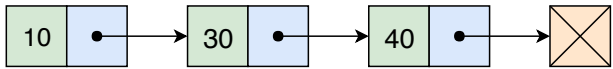
- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 40 na konec seznamu



- Odebrání prvku 20 ze seznamu



1. Nejdříve sekvenčně najdeme prvek s hodnotou 30

   Prvek, na který odkazuje NEXT odebíraného prvku.

2. Následně vyjmeme a napojíme prvek 10 na prvek 30

   Hodnotu NEXT prvku 10 nastavíme na adresu prvku 30.

Implementace spojového seznamu

- Seznam tvoří struktura prvku
  - Vlastní data prvku
  - Odkaz (ukazatel) na další prvek
- Vlastní seznam
  - Ukazatel na první prvek `head`,
  - nebo vlastní struktura pro seznam
- Příklad struktur pro uložení spojového seznamu celých čísel

Obecně mohou obsahovat libovolná data.

```c
/* polozka */
typedef struct entry
{
  int value;
  struct entry *next;
} entry_t;

entry_t *head = NULL;
```

```c
/* spojova struktura */
typedef struct
{
  entry_t *head;
  entry_t *tail;
  int counter; // pocet prvku
} linked_list_t;
```

Přidání prvku – příklad

1. Vytvoříme nový prvek (10) seznamu a uložíme odkaz v `head`

```
1 head = (entry_t*)malloc(sizeof(entry_t));
2 head->value = 10;
3 head->next = NULL;
```

2. Další prvek (20) přidáme propojením s aktuálně 1. prvkem

```
1 entry_t *new = (entry_t*)malloc(sizeof(entry_t));
2 new->value = 13;
3 new->next = head;
```

3. a aktualizací proměnné `head`

```
1 head = new;
```

- Stále máme přístup na všechny prvky přes `head` a `head->next`
- Inicializace položek prvku je důležitá
  - Hodnota `head == NULL` indikuje prázdný seznam
  - Hodnota `entry->next == NULL` indikuje poslední prvek seznamu

Spojový seznam – push()

- Přidání prvku na začátek implementujeme ve funkci `push()`
- Předáváme adresu, kde je uložen odkaz na start seznamu
  - `head` je ukazatel, předáváme adresu proměnné, tj. `&head`, parametr je ukazatel na ukazatel.

```
1  void push(int value, entry_t **head)
2  {
3    // add new entry at front
4    entry_t *new = (entry_t*)malloc(sizeof(entry_t));
5    new->value = value; // set data
6    if (*head == NULL)  // first entry in the list
7      new->next = NULL; // reset the next
8    else
9      new->next = *head;
10   *head = new;         // update the head
11 }
```

Spojový seznam – pop()

- Odebrání prvního prvku ze seznamu implementujeme ve funkci `pop()`

```
1  int pop(entry_t **head)
2  {
3    // linked list must be non-empty
4    assert(head != NULL && *head != NULL);
5    entry_t *prev_head = *head; // save the current head
6    int ret = prev_head->value;
7    *head = prev_head->next; // will be set to NULL if
8    // the last item is popped
9    free(prev_head); // relase memory of the popped entry
10   return ret;
11 }
```

Spojový seznam – size()

- Zjištění počtu prvků v seznamu vyžaduje projít seznam až k zarážce `NULL`, tj. položka `next` je `NULL`
- Proměnnou `cur` používáme jako **kurzor** pro procházení seznamu

```
1  int size(const entry_t *const head)
2  { // const - we do not attempt to modify the list
3    int counter = 0;
4    const entry_t *cur = head;
5    while (cur) { // or cur != NULL
6      cur = cur->next;
7      counter += 1;
8    }
9    return counter;
10 }
```

- Pro zjištění počtu prvků v seznamu musíme projít kompletní seznam, tj. `n` položek

Spojový seznam – back()

- Vrácení hodnoty posledního prvku ze seznamu

```
1  int back(const entry_t *const head)
2  {
3    const entry_t *end = head;
4    while (end && end->next) { // 1st test list is not empty
5      end = end->next;
6    }
7    assert(end); //do not allow calling back on empty list
8    return end->value;
9  }
```

- Pro vrácení hodnoty posledního prvku v seznamu musíme projít všechny položky seznamu

Spojový seznam – print()

- Procházení seznamu

```
1  void print(const entry_t *const head)
2  {
3    const entry_t *cur = head; // set the cursor to head
4    while (cur != NULL) {
5      printf("%i%s", cur->value, cur->next ? " " : "\n");
6      cur = cur->next; // move in the linked list
7    }
8  }
```

- Použijeme konstantní ukazatel na konstantní proměnnou, neboť seznam pouze procházíme a nemodifikujeme

Z hlavičky funkce je zřejmé, že vstupní strukturu nemodifikujeme.

Spojový seznam – vložení prvku

- Vložení do seznamu:
  - na začátek – modifikujeme proměnnou `head` (funkce `push()`)
  - na konec – modifikujeme proměnnou posledního prvku a nastavujeme nový konec `tail` (funkce `pushEnd()`)
  - obecně – potřebujeme prvek (`entry`), za který chceme nový prvek (`new_entry`) vložit

```
1  entry_t *new = (entry_t*)malloc(sizeof(entry_t));
2  // nastaveni hodnoty
3  new->value = value;
4  //propojeni s nasledujicim
5  new->next = entry->next;
6  //propojeni entry
7  entry->next = new;
```

- Do seznamu můžeme chtít prvek vložit na konkrétní pozici, tj. podle indexu v seznamu

Zajímavou variantou je vkládání podle velikosti – insert sort.

II. Spojový seznam

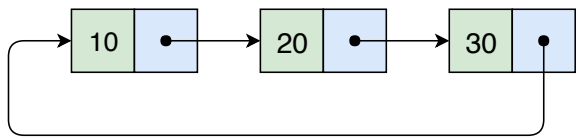
Spojový seznam

Kruhový spojový seznam

Obousměrný spojový seznam

Kruhový spojový seznam

- Položka next posledního prvku může odkazovat na první prvek
- Tak vznikne kruhový spojový seznam
- Při přidání prvku na začátek je nutné aktualizovat hodnotu položky next posledního prvku



II. Spojový seznam

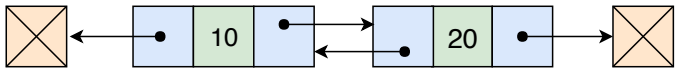
Spojový seznam

Kruhový spojový seznam

Obousměrný spojový seznam

Obousměrný spojový seznam

- Každý prvek obsahuje odkaz na následující a předchozí položku v seznamu, položky `prev` a `next`
- První prvek má nastavenu položku `prev` na hodnotu `NULL`
- Poslední prvek má next nastavenu na `NULL`
- Příklad obousměrného seznamu celých čísel



9.3 Zásobník – implementace spojovým seznamem 1/4

- Definujeme strukturu `stack_entry_t` pro položku seznamu

```
1  typedef struct entry {
2    void *value; //ukazatel na hodnotu vlozeného prvku
3    struct entry *next;
4  } stack_entry_t;
```

- Struktura zásobníku `stack_t` obsahuje pouze ukazatel na `head`

```
1  typedef struct {
2    stack_entry_t *head;
3  } stack_t;
```

- Inicializace tak pouze alokuje strukturu `stack_t`

```
1  void stack_init (stack_t **stack) {
2    *stack = (stack_t *)malloc(sizeof(stack_t));
3    (*stack)->head = NULL;
4  }
```

- Při vkládání prvku `push()` alokujeme položku spojového seznamu

```
1  int stack_push (void *value, stack_t *stack) {
2    int ret = STACK_OK;
3    stack_entry_t *new_entry = (stack_entry_t *) malloc(sizeof(
       stack_entry_t));
4    if (new_entry) {
5      new_entry->value = value;
6      new_entry->next = stack->head;
7      stack->head = new_entry;
8    } else {
9      ret = STACK_MEMFAIL;
10   }
11   return ret;
12 }
```

- Při vyjmutí prvku funkcí `pop()` paměť uvolňujeme

```
1  void* stack_pop (stack_t *stack) {
2    void *ret = NULL;
3    if (stack->head) {
4      ret = stack->head->value; //retrive the value
5      stack_entry_t *tmp = stack->head;
6      stack->head = stack->head->next;
7      free (tmp);  // release stack_entry_t
8    }
9    return ret;
10 }
```

- Implementace `stack_is_empty()` a `stack_peek()` je triviální

```
1  int stack_is_empty (const stack_t *stack) {
2    return stack->head == 0;
3  }
5  void* stack_peek (const stack_t *stack) {
6    return stack_is_empty (stack) ? NULL : stack->head->value;
7  }
8
```

- Použití je identické jako v předchozím případě
  - Výhoda spojového seznamu proti implementaci v poli je v (téměř) neomezené kapacitě zásobníku

    Jsem omezen pouze dostupnou pamětí.
  - Nevýhodou spojového seznamu je větší paměťová režie

9.4 Fronta – implementace spojovým seznamem 1/4

- Spojový seznam s udržováním začátku `head` a konce `end` seznamu
- Strategie vkládání a odebírání prvků
  - Vložením prvku do fronty `queue_push()` dáme prvek na konec seznamu `end`
  - Odebrání prvku z fronty `queue_pop()` vezmeme prvek z počátku seznamu `head`
  - Nemusíme tak lineárně procházet seznam a aktualizovat `end` při odebrání prvku z fronty

```
1  typedef struct entry {
2    void *value;
3    struct entry *next;
4  } queue_entry_t;
6  typedef struct {
7    queue_entry_t *head;
8    queue_entry_t *end;
9  } queue_t;
```

```
1  void queue_init(queue_t **
       queue) {
2    *queue = (queue_t*)malloc
       (sizeof(queue_t));
3    (*queue)->head = NULL;
4    (*queue)->end = NULL;
5  }
```

```
1  int queue_push (void *value, queue_t *queue) {
2    int ret = QUEUE_OK;
3    queue_entry_t *new_entry = malloc (sizeof(queue_entry_t));
4    if (new_entry) { // fill the new_entry
5      new_entry->value = value;
6      new_entry->next = NULL;
7      if (queue->end) { // if queue has end
8        queue->end->next = new_entry; // link new_entry
9      } else { // queue is empty
10       queue->head = new_entry; // update head as well
11     }
12     queue->end = new_entry; // set new_entry as end
13   } else
14     ret = QUEUE_MEMFAIL;
15   return ret;
16 }
```

```
1  void* queue_pop (queue_t *queue) {
2    void *ret = NULL;
3    if (queue->head) { // having at least one entry
4      ret = queue->head->value; //retrive the value
5      queue_entry_t *tmp = queue->head;
6      queue->head = queue->head->next;
7      free (tmp);  // release queue_entry_t
8      if (queue->head == NULL) { // update end if last
9        queue->end = NULL;       // entry has been
10     }                          // popped
11   }
12   return ret;
13 }
```

```
1  int queue_is_empty (const queue_t *queue) {
2    return queue->head == 0;
3  }
5  void* queue_peek (const queue_t *queue) {
6    return queue_is_empty (queue) ? NULL : queue->head->value;
7  }
```