

9. Spojové struktury. Abstraktní datový typ.

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Spojové struktury
 - Spojový seznam
 - Spojový seznam
 - Vložení / odebrání prvku
 - Kruhový spojový seznam
 - Obousměrný spojový seznam
- Část 2 – Abstraktní datový typ
 - Datové struktury a abstraktní datový typ
 - Zásobník
 - Fronta
- Část 3 – Zadání domácího úkolu

Část I

Spojový seznam

I. Spojový seznam

Spojový seznam

Spojový seznam

Vložení / odebrání prvku

Kruhový spojový seznam

Obousměrný spojový seznam

Spojový seznam

- 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:

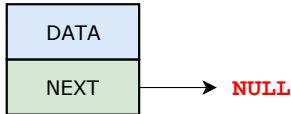
- Pole
 - Indexování je velmi rychlé
 - Vložení prvku na konkrétní pozici může být pomalé

Nová alokace a kopírování.

- Spojové seznamy

Spojové seznamy

- 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

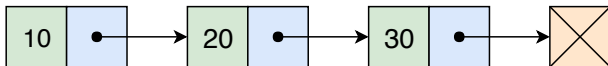
Obousměrný spojový seznam.

- 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.

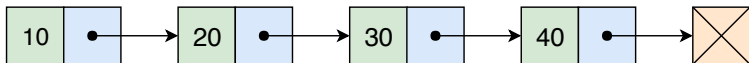
jednosměrný spojový seznam

Jednosměrný spojový seznam

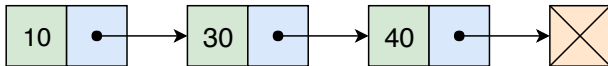
- Příklad spojového seznamu pro uložení číselných hodnot



- 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.

I. Spojový seznam

Spojový seznam

Spojový seznam

Vložení / odebrání prvku

Kruhový spojový seznam

Obousměrný spojový seznam

Spojový seznam

- 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.

```
/* polozka */
typedef struct entry
{
    int value;
    struct entry *next;
} entry_t;
entry_t *head = NULL;
```

```
/* 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`

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

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

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

3. a aktualizací proměnné `head`

```
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, proto předáváme adresu proměnné, tj. `&head` a parametr je ukazatel na ukazatel.

```
void push(int value, entry_t **head)
{ // add new entry at front
  entry_t *new = (entry_t*)malloc(sizeof(entry_t));
  new->value = value; // set data
  if (*head == NULL) // first entry in the list
    new->next = NULL; // reset the next
  else
    new->next = *head;
  *head = new; // update the head
}
```

- Alternativně můžeme `push()` implementovat také například jako `entry_t* push(int value, entry_t *head)`

Spojový seznam – pop()

- Odebrání prvního prvku ze seznamu

```
int pop(entry_t **head)
{ // linked list must be non-empty
  assert(head != NULL && *head != NULL);
  entry_t *prev_head = *head; // save the current head
  int ret = prev_head->value;
  *head = prev_head->next; // will be set to NULL if
  // the last item is popped
  free(prev_head); // release memory of the popped entry
  return ret;
}
```

- Alternativně například také jako `int pop(entry_t *head)`, ale nenastaví `head` na `NULL` v případě vyjmutí posledního prvku

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

```
int size(const entry_t *const head)
{ // const - we do not attempt to modify the list
  int counter = 0;
  const entry_t *cur = head;
  while (cur) { // or cur != NULL
    cur = cur->next;
    counter += 1;
  }
  return counter;
}
```

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

Spojový seznam – back()

- Vrácení hodnoty posledního prvku ze seznamu

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

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

Spojový seznam – print()

- Procházení seznamu

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

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

Z hlavičky funkce je zřejmé, že vstupní strukturu nemodifikujeme.

I. Spojový seznam

Spojový seznam

Spojový seznam

Vložení / odebrání prvku

Kruhový spojový seznam

Obousměrný spojový seznam

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

```
entry_t *new = (entry_t*)malloc(sizeof(entry_t));  
new->value = value; // nastaveni hodnoty  
new->next = entry->next; //propojeni s nasledujicim  
entry->next = new; //propojeni entry
```

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

Zajímavou variantou je vkládání podle velikosti – insert sort.

I. Spojový seznam

Spojový seznam

Spojový seznam

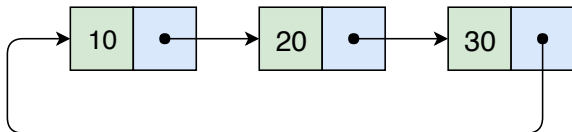
Vložení / odebrání prvku

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



I. Spojový seznam

Spojový seznam

Spojový seznam

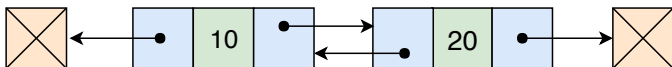
Vložení / odebrání prvku

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 nastavenou položku `prev` na hodnotu `NULL`
- Poslední prvek má `next` nastavenou na `NULL`
- Příklad obousměrného seznamu celých čísel



Část II

Abstraktní datový typ

II. Abstraktní datový typ

Datové struktury a abstraktní datový typ

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

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 (vlození / vyjmutí položky)
- Typ položek (dat):
 - Homogenní – všechny položky jsou stejného typu
 - Nehomogenní – položky mohou být různého typu
- Existence bezprostředního následníka
 - Lineární – existuje bezprostřední následník prvku, např. pole, fronta, seznam, . . .
 - Nelineární – neexistuje přímý jednoznačný následník, např. strom

II. Abstraktní datový typ

Datové struktury a abstraktní datový typ

Zásobník

Fronta

- Zásobník je dynamická datová struktura umožňující 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 prázdnotu 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 se skládá z dynamicky alokovaného pole hodnot ukazatelů odkazujících na jednotlivé prvky uložené do zásobníku

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

- Pomocné funkce pro inicializaci a uvolnění paměti

```
void stack_init(stack_t **stack);  
void stack_delete(stack_t **stack);  
void stack_free(stack_t *stack);
```

- Základní operace se zásobníkem

```
int stack_push(void *value, stack_t *stack);  
void* stack_pop(stack_t *stack);  
int stack_is_empty(const stack_t *stack);  
void* stack_peek(const stack_t *stack);
```

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

```
#ifndef STACK_SIZE
#define STACK_SIZE 5
#endif

void stack_init (stack_t **stack) {
    *stack = (stack_t*) malloc(sizeof(stack_t));
    (*stack)->stack = (void**) malloc(sizeof(void*)*
        STACK_SIZE);
    (*stack)->count = 0;
}
```

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

```
void stack_free(stack_t *stack) {
    while(!stack_is_empty (stack)) {
        void *value = stack_pop (stack);
        free (value);
    }
}
```

- `stack_delete()` kompletně uvolní paměť alokovanou zásobníkem

```
void stack_delete (stack_t **stack) {
    stack_free (*stack);
    free((*stack)->stack);
    free(*stack);
    *stack = NULL;
}
```

```
int stack_push (void *value, stack_t *stack) {
    int ret = STACK_OK;
    if (stack->count < MAX_STACK_SIZE) {
        stack->stack[stack->count++] = value;
    } else {
        ret = STACK_MEMFAIL;
    }
    return ret;
}

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

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

int stack_is_empty (const stack_t *stack) {
    return stack->count == 0;
}
```

- Definujeme strukturu `stack_entry_t` pro položku seznamu

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

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

```
typedef struct {  
    stack_entry_t *head;  
} stack_t;
```

- Inicializace tak pouze alokuje strukturu `stack_t`

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

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

```
int stack_push (void *value, stack_t *stack) {
    int ret = STACK_OK;
    stack_entry_t *new_entry = (stack_entry_t *) malloc(
        sizeof(stack_entry_t));
    if (new_entry) {
        new_entry->value = value;
        new_entry->next = stack->head;
        stack->head = new_entry;
    } else {
        ret = STACK_MEMFAIL;
    }
    return ret;
}
```

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

```
void* stack_pop (stack_t *stack) {
    void *ret = NULL;
    if (stack->head) {
        ret = stack->head->value; //retrive the value
        stack_entry_t *tmp = stack->head;
        stack->head = stack->head->next;
        free (tmp); // release stack_entry_t
    }
    return ret;
}
```


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

```
int stack_is_empty (const stack_t *stack) {  
    return stack->head == 0;  
}  
  
void* stack_peek (const stack_t *stack) {  
    return stack_is_empty (stack) ? NULL : stack->head->  
        value;  
}
```

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

Jsem omezen pouze dostupnou pamětí.

II. Abstraktní datový typ

Datové struktury a abstraktní datový typ

Zásobník

Fronta

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 prázdnotu 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

```
typedef struct {  
    ...  
} queue_t;  
  
void queue_delete (queue_t **queue);  
void queue_free (queue_t *queue);  
void queue_init (queue_t **queue);  
int queue_push (void *value, queue_t *queue);  
void* queue_pop (queue_t *queue);  
int queue_is_empty (const queue_t *queue);  
void* queue_peek (const queue_t *queue);
```

Fronta – implementace polem 1/2

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

```
int queue_push (void *value, queue_t *queue) {
    int ret = QUEUE_OK;
    if (queue->count < MAX_QUEUE_SIZE) {
        queue->queue[queue->end] = value;
        queue->end = (queue->end + 1) % MAX_QUEUE_SIZE;
        queue->count += 1;
    } else {
        ret = QUEUE_MEMFAIL;
    }
    return ret;
}
```

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

Fronta – implementace polem 2/2

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

```
void* queue_pop (queue_t *queue) {
    void* ret = NULL;
    if (queue->count > 0) {
        ret = queue->queue[queue->start];
        queue->start = (queue->start + 1) % MAX_QUEUE_SIZE;
        queue->count -= 1;
    }
    return ret;
}

void* queue_peek (const queue_t *queue) {
    return queue_is_empty(queue) ? NULL : queue->queue[
        queue->start];
}
```

- Spojový seznam s udržováním začátku `head` a konce `end` seznamu
- Strategie vkládání a odebírání prvků
 - Vložení 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

```
typedef struct entry {  
    void *value;  
    struct entry *next;  
} queue_entry_t;
```

```
typedef struct {  
    queue_entry_t *head;  
    queue_entry_t *end;  
} queue_t;
```

```
void queue_init(queue_t **  
queue) {  
    *queue = (queue_t*)malloc  
        (sizeof(queue_t));  
    (*queue)->head = NULL;  
    (*queue)->end = NULL;  
}
```


- `push()` vkládá prvky na konec seznamu `end`

```
int queue_push (void *value, queue_t *queue) {
    int ret = QUEUE_OK;
    queue_entry_t *new_entry = (queue_entry_t*) malloc (
        sizeof(queue_entry_t));
    if (new_entry) { // fill the new_entry
        new_entry->value = value;
        new_entry->next = NULL;
        if (queue->end) { // if queue has end
            queue->end->next = new_entry; // link new_entry
        } else { // queue is empty
            queue->head = new_entry; // update head as well
        }
        queue->end = new_entry; // set new_entry as end
    } else
        ret = QUEUE_MEMFAIL;
    return ret;
}
```

```
void* queue_pop (queue_t *queue) {
    void *ret = NULL;
    if (queue->head) { // having at least one entry
        ret = queue->head->value; //retrive the value
        queue_entry_t *tmp = queue->head;
        queue->head = queue->head->next;
        free (tmp); // release queue_entry_t
        if (queue->head == NULL) { // update end if last
            queue->end = NULL; // entry has been
        } // popped
    }
    return ret;
}

int queue_is_empty (const queue_t *queue) {
    return queue->head == 0;
}

void* queue_peek (const queue_t *queue) {
    return queue_is_empty (queue) ? NULL : queue->head->value;
}
```

Část III

Zadání domácího úkolu

Zadání 7. domácího úkolu (HW07)

Téma: Zpracování číselné řady

- **Motivace:** Práce s polem v dynamické paměti
- **Cíl:** Zpracování teoreticky neomezeně dlouhé řady celých čísel
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b99prpa/hw/hw07>
 - Načtení předem neznámého počtu vstupních hodnot
 - Výpočet standardních statistických ukazatelů – medián, histogram
- **Termín odevzdání:** 30.11.2019, 23:59:59