

Abstraktní datový typ

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 09

BAB36PRGA – Programování v C

Přehled témat

- Část 1 – Abstraktní datový typ

Datové struktury

Zásobník

Fronta

Prioritní fronta

Prioritní fronta spojovým seznamem

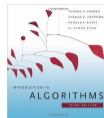
Část I

Část 1 – Abstraktní datový typ

Zdroje



Introduction to Algorithms, 3rd Edition, *Cormen, Leiserson, Rivest, and Stein*, The MIT Press, 2009, ISBN 978-0262033848.



Algorithms (4th Edition) *Robert Sedgwick and Kevin Wayne* Addison-Wesley Professional, 2010, ISBN: 978-0321573513.



<http://algs4.cs.princeton.edu/home>

- Data Structure & Algorithms Tutorial

http://www.tutorialspoint.com/data_structures_algorithms

- Algorithms and Data Structures with implementations in Java and C++

<http://www.algolist.net>

- Algoritmy jednoduše a srozumitelně
Algoritmy + Datové struktury = Programy

<http://algoritmy.eu>

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.
 - Fronta (Queue)
 - Zásobník (Stack)
 - Pole (Array)
 - Tabulka (Table)
 - Seznam (List)
 - Strom (Tree)
 - Množina (Set)

Nezávislé na konkrétní implementaci

Abstraktní datový typ

- Množina druhů dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to **nezávisle na konkrétní implementaci**.
- Můžeme definovat
 - Matematicky – signatura a axiomy
 - Rozhraním (interface) a popisem operací, kde rozhraní poskytuje
 - Konstruktor vracející odkaz (na strukturu nebo objekt).
Procedurální i objektově orientovaný přístup.
 - Operace, které akceptují odkaz na argument (data) a mají přesně definovaný účinek na data.

Abstraktní datový typ (ADT) – Vlastnosti

- 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.
Např. vložení nebo odebrání určitého prvku
- 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.

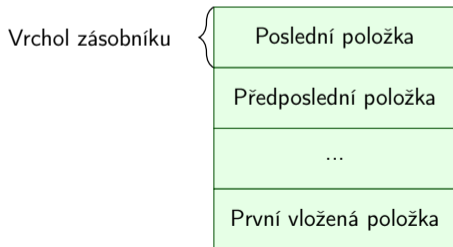
Příklad ADT – Zásobník

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:

- Vložení hodnoty na vrchol zásobníku;
- Odebrání hodnoty z vrcholu zásobníku;
- Test na prázdnotu zásobníku.



Příklad ADT – Operace nad zásobníkem

Základní operace nad zásobníkem jsou

- **push()** – vložení prvku na vrchol zásobníku;
- **pop()** – vyjmutí prvku z vrcholu zásobníku;
- **isEmpty()** – test na prázdnotu zásobníku.

Další operace nad zásobníkem mohou být

- **peek()** – čtení hodnoty z vrcholu zásobníku;
- **search()** – vrátí pozici prvku v zásobníku;
- **size()** – vrátí aktuální počet prvků (hodnot) v zásobníku.

*Alternativně také třeba **top()**.*

Pokud se nachází v zásobníku, jinak -1.

Zpravidla není potřeba.

Příklad ADT – Rozhraní zásobníku 1/2

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

```
1 int stack_push(void *value, void **stack);
2 void* stack_pop(void **stack);
3 _Bool stack_is_empty(void **stack);
4 void* stack_peek(void **stack);

5 void stack_init(void **stack); // init. dat. reprez.
6 void stack_delete(void **stack); // kompletní smazání
7 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.

Příklad ADT – Rozhraní zásobníku 2/2

- Součástí definice rozhraní ADT je také popis chování operací.

```
1  /*
2  * Function: stack_push
3  * -----
4  * This routine push the given value onto the top of the stack.
5  *
6  * value - value to be placed on the stack
7  * stack - stack to push
8  *
9  * returns: The function returns status value:
10 *
11 * OK          - success
12 * CLIB_MEMFAIL - dynamic memory allocation failure
13 *
14 * This function requires the following include files:
15 *
16 * prg_stack.h prg_errors.h
17 */
18 int stack_push(void *value, void **stack);
```

Implementace zásobníku

- **ADT není závislý naa konkrétní implementaci** – zásobník můžeme implementovat různě.
 - Polem fixní velikosti (definujeme chování při zaplnění);
 - Polem s měnitelnou velikostí (realokace);
 - Spojovým seznamem.
- Ukážeme si tři různé implementace, každá se shodným rozhraním a jménem typu `stack_t`, ale definované v samostatných modulech.
 - `lec09/stack_array.h`, `lec09/stack_array.c`
 - `lec09/stack_array_alloc.h`, `lec09/stack_array_alloc.c`
 - `lec09/stack_linked_list.h`, `lec09/stack_linked_list.c`
- Dále si ukážeme použití maker preprocesoru a jejich definici při překladu.
- Ukázkové implementace také slouží jako příklady, jak zacházet s dynamickou pamětí a jak se vyhnout tzv. únikům paměti (**memory leaks**).

Implementace zásobníku polem 1/3

- Struktura zásobníku se skládá z dynamicky alokovaného pole hodnot ukazatelů odkazujících na jednotlivé prvky uložené do zásobníku.

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

- Pro inicializaci a uvolnění paměti implementujeme pomocné funkce.

```
6 void stack_init(stack_t **stack);  
7 void stack_delete(stack_t **stack);  
8 void stack_free(stack_t *stack);
```

- Základní operace se zásobníkem mají tvar

```
10 int stack_push(void *value, stack_t *stack);  
11 void* stack_pop(stack_t *stack);  
12 _Bool stack_is_empty(const stack_t *stack);  
13 void* stack_peek(const stack_t *stack);
```

- a jsou pro všechny tři implementace totožné.

[lec09/stack_array.h](#)

Implementace zásobníku polem 2/3

- Maximální velikost zásobníku je definována hodnotou makra `MAX_STACK_SIZE`.

Lze předdefinovat při překladu, např. `clang -DMAX_STACK_SIZE=100`.

```

1 #ifndef MAX_STACK_SIZE
2 #define MAX_STACK_SIZE 5
3 #endif
4
5 void stack_init(stack_t **stack)
6 {
7     *stack = myMalloc(sizeof(stack_t));
8     (*stack)->stack = myMalloc(sizeof(
9         void*)*MAX_STACK_SIZE);
10    (*stack)->count = 0;
11 }

```

```

1 #include <stdlib.h>
2
3 void* myMalloc(size_t size)
4 {
5     void *ret = malloc(size);
6     if (!ret) {
7         fprintf(stderr, "Malloc failed
8             !\n");
9         exit(-1)
10    }
11    return ret;

```

lec09/my_malloc.c

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

- `stack_delete()` kompletně uvolní paměť alokovanou zásobníkem.

```

12 void stack_free(stack_t *stack)
13 {
14     while (!stack_is_empty(stack)) {
15         void *value = stack_pop(stack);
16         free(value);
17     }
18 }

```

```

19 void stack_delete(stack_t **stack)
20 {
21     *stack_free(*stack);
22     free((*stack)->stack);
23     free(*stack);
24     *stack = NULL;
25 }

```

lec09/stack_array.c

Implementace zásobníku polem 3/3

```
28 int stack_push(void *value, stack_t *stack)
29 {
30     int ret = STACK_OK;
31     if (stack->count < MAX_STACK_SIZE) {
32         stack->stack[stack->count++] = value;
33     } else {
34         ret = STACK_MEMFAIL;
35     }
36     return ret;
37 }

39 void* stack_pop(stack_t *stack)
40 {
41     return stack->count > 0 ? stack->stack[--(stack->count)]: NULL;
42 }

44 void* stack_peek(const stack_t *stack)
45 {
46     return stack_is_empty(stack) ? NULL : stack->stack[stack->count - 1];
47 }

49 _Bool stack_is_empty(const stack_t *stack)
50 {
51     return stack->count == 0;
52 }
```

Proč v metodě pop() používáme $--(stack \rightarrow count)$ a v peek() $count - 1$?

Zásobník – Příklad použití 1/3

- Položky (hodnoty typu `int`) alokujeme dynamicky.

```
1  int* getRandomInt()
2  {
3      int *r = myMalloc(sizeof(int)); // dynamicky alokovaný int
4      *r = rand() % 256;
5      return r;
6  }
7  stack_t *stack;
8  stack_init(&stack);

10 for (int i = 0; i < 15; ++i) {
11     int *pv = getRandomInt();
12     int r = stack_push(pv, stack);
13     printf("Add %2i entry '%3i' to the stack r = %i\n", i, *pv, r);
14     if (r != STACK_OK) {
15         fprintf(stderr, "Error: Stack is full!\n");
16         fprintf(stderr, "Info: Release pv memory and quit pushing\n");
17         free(pv); // Nutné uvolnit alokovanou paměť
18         break;
19     }
20 }
```

lec09/demo-stack_array.c

- V případě zaplnění zásobníku **nezapomenout uvolnit paměť**.

Zásobník – Příklad použití 2/3

- Po vyjmutí položky a jejím zpracování je nutné uvolnit paměť.

```
22 printf("\nPop the entries from the stack\n");
23 while (!stack_is_empty(stack)) {
24     int *pv = (int*)stack_pop(stack);
25     printf("Popped value is %3i\n", *pv);
26     free(pv);
27 }
28 stack_delete(&stack);
```

lec09/demo-stack_array.c

- Na závěr uvolníme paměť zásobníku funkcí `stack_delete()`.
- Při výchozí kompilaci má zásobník dle `MAX_STACK_SIZE` kapacitu 3.

```
$ clang stack_array.c demo-stack_array.c && ./a.out
```

```
Add 0 entry ' 77' to the stack r = 0
```

```
Add 1 entry '225' to the stack r = 0
```

```
Add 2 entry '178' to the stack r = 0
```

```
Add 3 entry ' 83' to the stack r = 1
```

```
Error: Stack is full!
```

```
Info: Release pv memory and quit pushing
```

```
Pop the entries from the stack
```

```
Popped value is 178
```

```
Popped value is 225
```

```
Popped value is 77
```

Zásobník – Příklad použití 3/3

- Při kompilaci můžeme specifikovat hodnotu makra `MAX_STACK_SIZE`.

```
$ clang -DMAX_STACK_SIZE=5 stack_array.c demo-stack_array.c && ./a.out
```

```
Add 0 entry '77' to the stack r = 0
```

```
Add 1 entry '225' to the stack r = 0
```

```
Add 2 entry '178' to the stack r = 0
```

```
Add 3 entry '83' to the stack r = 0
```

```
Add 4 entry '4' to the stack r = 0
```

```
Add 5 entry '143' to the stack r = 1
```

```
Error: Stack is full!
```

```
Info: Release pv memory and quit pushing
```

```
Pop the entries from the stack
```

```
Popped value is 4
```

```
lec09/stack_array.h
```

```
Popped value is 83
```

```
lec09/stack_array.c
```

```
Popped value is 178
```

```
lec09/demo-stack_array.c
```

```
Popped value is 225
```

```
Popped value is 77
```

- Vyzkoušejte si zakomentovat různá volání `free()` a sledovat chování programu – nástrojem `valgrind`!

Implementace zásobníku rozšiřitelným polem 1/3

- V případě naplnění pole vytvoříme nové o „něco“ větší pole, zvětšení je definované hodnotou makra `STACK_RESIZE`.
- Počáteční velikost je definována makrem `INIT_STACK_SIZE`.

```
#ifndef INIT_STACK_SIZE          #ifndef STACK_RESIZE
#define INIT_STACK_SIZE 3       #define STACK_RESIZE 3
#endif                          #endif
```

```
void stack_init(stack_t **stack)
{
    *stack = myMalloc(sizeof(stack_t));
    (*stack)->stack = myMalloc(sizeof(void*)*INIT_STACK_SIZE);
    (*stack)->count = 0;
    (*stack)->size = INIT_STACK_SIZE;
}
```

- Dále pak funkcí `push()`, kterou modifikujeme o realokaci pole `stack→stack`.

Implementace zásobníku rozšiřitelným polem 2/3

- Volání `realloc()` rozšíří alokovanou paměť nebo alokuje novou a obsah původní paměti překopíruje a následně paměť uvolní, nebo alokace selže a `realloc()` vrací `NULL`.

Viz man `realloc`

```
1 int stack_push(void *value, stack_t *stack)
2 {
3     int ret = STACK_OK;
4     if (stack->count == stack->size) { // try to realloc
5         void **tmp = (void**)realloc(
6             stack->stack,
7             sizeof(void*) * (stack->size + STACK_RESIZE)
8         );
9         if (tmp) { // realloc has been successful, stack->stack has been eventually freed
10            stack->stack = tmp; //
11            stack->size += STACK_RESIZE;
12        }
13    }
14    if (stack->count < stack->size) {
15        stack->stack[stack->count++] = value;
16    } else {
17        ret = STACK_MEMFAIL;
18    }
19    return ret;
20 }
```

lec09/stack_array-alloc.c

Implementace zásobníku rozšiřitelným polem 3/3

- Použití `stack_array-alloc` je identické jako `stack_array`.
- Soubor `demo-stack_array-alloc.c` pouze vkládá `stack_array-alloc.h` místo `stack_array.h`.

```
$ clang stack_array-alloc.c demo-stack_array-alloc.c && ./a.out
```

```
Add 0 entry ' 77' to the stack r = 0
```

```
Add 1 entry '225' to the stack r = 0
```

```
Add 2 entry '178' to the stack r = 0
```

```
Add 3 entry ' 83' to the stack r = 0
```

```
Add 4 entry '  4' to the stack r = 0
```

```
Pop the entries from the stack
```

```
Popped value is  4
```

```
Popped value is 83
```

```
Popped value is 178
```

```
Popped value is 225
```

```
Popped value is 77
```

```
lec09/stack_array-alloc.h
```

```
lec09/stack_array-alloc.c
```

```
lec09/demo-stack_array-alloc.c
```

Implementace zásobníku spojovým seznamem 1/3

- Zásobník také můžeme implementovat spojovým seznamem.
- Definujeme strukturu `stack_entry_t` pro položku seznamu.

Viz 8. přednáška.

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

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

```
6 typedef struct {
7     stack_entry_t *head;
8 } stack_t;
```

- Inicializace pouze alokuje strukturu `stack_t`.

```
1 void stack_init(stack_t **stack)
2 {
3     *stack = myMalloc(sizeof(stack_t));
4     (*stack)->head = NULL;
5 }
```

Implementace zásobníku spojovým seznamem 2/3

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

```
7 int stack_push(void *value, stack_t *stack)
8 {
9     int ret = STACK_OK;
10    stack_entry_t *new_entry = malloc(sizeof(stack_entry_t));
11    if (new_entry) {
12        new_entry->value = value;
13        new_entry->next = stack->head;
14        stack->head = new_entry;
15    } else {
16        ret = STACK_MEMFAIL;
17    }
18    return ret;
19 }
```

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

```
21 void* stack_pop(stack_t *stack)
22 {
23     void *ret = NULL;
24     if (stack->head) {
25         ret = stack->head->value; //retrive the value
26         stack_entry_t *tmp = stack->head;
27         stack->head = stack->head->next;
28         free(tmp); // release stack_entry_t
29     }
30     return ret;
31 }
```

lec09/stack_linked_list.c

Implementace zásobníku spojovým seznamem 3/3

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

```
33 _Bool stack_is_empty(const stack_t *stack)
34 {
35     return stack->head == 0;
36 }
```

```
38 void* stack_peek(const stack_t *stack)
39 {
40     return stack_is_empty(stack) ? NULL : stack->head->value;
41 }
```

`lec09/stack_linked_list.c`

- Použití je identické jako v obou předchozích případech. `lec09/demo-stack_linked_list.c`
- Výhoda spojového seznamu proti implementaci `stack_array` je v „neomezené“ kapacitě zásobníku. *Omezení pouze do výše volné paměti.*
- Výhoda spojového seznamu proti `stack_array-alloc` je v automatickém uvolnění paměti při odebrání prvků ze zásobníku.
- Nevýhodou spojového seznamu je větší paměťová režie (položka `next`).

ADT – Zásobník příklad použití různých implementací

- S využitím preprocesoru můžeme různé implementace kombinovat v jediném zdrojovém souboru.

```
1 #if STACK_ARRAY
2 # include "stack_array.h"
3 #elif STACK_ARRAY_ALLOC
4 # include "stack_array-alloc.h"
5 #elif STACK_LINKED_LIST
6 #include "stack_linked_list.h"
7 #endif
```

lec09/demo-stack.c

- Při kompilaci definujeme jedno z maker a při linkování pak volíme jednu konkrétní implementaci (.o soubor nebo .c soubor).

- Pole

```
clang -DSTACK_ARRAY stack_array.c demo-stack.c && ./a.out
```

- Pole s realokací

```
clang -DSTACK_ARRAY_ALLOC stack_array-alloc.c demo-stack.c && ./a.out
```

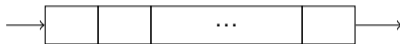
- Spojový seznam

```
clang -DSTACK_LINKED_LIST stack_linked_list.c demo-stack.c && ./a.out
```

Příklad ADT – Fronta

- **Fronta** je dynamická datová struktura, kde se odebírají prvky v tom pořadí, v jakém byly vloženy.
- Jedná se o strukturu typu **FIFO** (First In, First Out).

Vložení hodnoty
na konec fronty



Odebrání hodnoty
z čela fronty

■ Implementace

- Pole – *Pamatujeme si pozici začátku a konce fronty v poli.*

- Pozice cyklicky rotují (modulo velikost pole).

Kruhová fronta.

- Spojovým seznamem — *Pamatujeme si ukazatel na začátek a konec fronty.*

- Můžeme implementovat tak, že přidáváme na začátek (**head**) a odebíráme z konce.

`push()` a `popEnd()` z 8. přednášky

- Nebo přidáváme na konec a odebíráme ze začátku (**head**).

`pushEnd()` a `pop()` z 8. přednášky.

- Z hlediska vnějšího (ADT) chování fronty na vnitřní implementaci nezáleží.

ADT – Operace nad **frontou**

- 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.

ADT – Příklad implementace fronty

- Implementace fronty pole a spojovým seznamem.
- Využijeme shodné rozhraní a jméno typu `queue_t` definované v samostatných modulech.
 - `lec09/queue_array.h`, `lec09/queue_array.c`
 - `lec09/queue_linked_list.h`, `lec09/queue_linked_list.c`

Implementace vychází ze zásobníku, liší se zejména ve funkci `pop()` a `peek()` spolu s udržováním prvního a posledního prvku.

```
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);  
_Bool queue_is_empty(const queue_t *queue);  
void* queue_peek(const queue_t *queue);
```

Příklad implementace fronty polem 1/2

- Téměř identická implementace s implementací `stack_array`.
- 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 < MAX_QUEUE_SIZE`).

end vždy v rozsahu $0 \leq \text{end} < \text{MAX_QUEUE_SIZE}$.

- Dále implementujeme `queue_pop()` a `queue_peek()`.

lec09/queue_array.c

Příklad implementace fronty polem 2/2

- Funkce `queue_pop()` vrací hodnotu na pozici `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];
}
```

`lec09/queue_array.c`

- Příklad použití viz `lec09/demo-queue_array.c`.

Příklad implementace fronty spojovým seznamem 1/3

- Spojový seznam s udržováním začátku `head` a konce `end` seznamu.

Viz `lec08/linked_list.c`

- Strategie vkládání a odebírání prvků.

- Vložení prvku do fronty `queue_push()` dáme prvek na konec seznamu `end`.
Aktualizujeme pouze `end→next` s konstantní složitostí $O(1)$.
- Odebrání prvku z fronty `queue_pop()` vezmeme prvek z počátku seznamu `head`.
Aktualizujeme pouze `head→next` opět s konstantní složitostí $O(1)$.
- Nemusíme lineárně procházet seznam a aktualizovat `end` při odebrání prvku z fronty.

```
1 typedef struct entry {           1 void queue_init(queue_t **queue)
2     void *value;                 2 {
3     struct entry *next;          3     *queue = myMalloc(sizeof(queue_t);
4 } queue_entry_t;                4     (*queue)->head = NULL;
                                   5     (*queue)->end = NULL;
6 typedef struct {                6 }
7     queue_entry_t *head;
8     queue_entry_t *end;
9 } queue_t;                       lec09/queue_linked_list.h
                                   lec09/queue_linked_list.c
```

Implementace fronty spojovým seznamem 2/3

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

```
8 int queue_push(void *value, queue_t *queue)
9 {
10     int ret = QUEUE_OK;
11     queue_entry_t *new_entry = malloc(sizeof(queue_entry_t));
12     if (new_entry) { // fill the new_entry
13         new_entry->value = value;
14         new_entry->next = NULL;
15         if (queue->end) { // if queue has end
16             queue->end->next = new_entry; // link new_entry
17         } else { // queue is empty
18             queue->head = new_entry; // update head as well
19         }
20         queue->end = new_entry; // set new_entry as end
21     } else {
22         ret = QUEUE_MEMFAIL;
23     }
24     return ret;
25 }
```

lec09/queue_linked_list.c

Implementace fronty spojovým seznamem 3/3

- `pop()` odebírá prvky ze začátku seznamu `head`.

```
27 void* queue_pop(queue_t *queue)
28 {
29     void *ret = NULL;
30     if (queue->head) { // having at least one entry
31         ret = queue->head->value; //retrive the value
32         queue_entry_t *tmp = queue->head;
33         queue->head = queue->head->next;
34         free(tmp); // release queue_entry_t
35         if (queue->head == NULL) { // update end if last
36             queue->end = NULL; // entry has been
37         } // popped
38     }
39     return ret;
40 }
```

- Implementace `isEmpty()` a `peek()` je přímočará.

```
42 _Bool queue_is_empty(const queue_t *queue) {
43     return queue->head == 0;
44 }
45 void* queue_peek(const queue_t *queue) {
46     return queue_is_empty(queue) ? NULL : queue->head->value;
47 }
```

`lec09/queue_linked_list.c`

ADT – Fronta spojovým seznamem – příklad použití

```
1 for (int i = 0; i < 3; ++i) {
2     int *pv = getRandomInt();
3     int r = queue_push(pv, queue);
4     printf("Add %2i entry '%3i' to the queue r = %i\n", i, *pv, r);
5     if (r != QUEUE_OK) { free(pv); break; } // release allocated pv
6 }
7 printf("\nPop the entries from the queue\n");
8 while (!queue_is_empty(queue)) {
9     int *pv = (int*)queue_pop(queue);
10    printf("Popped value is %3i\n", *pv);
11    free(pv);
12 }
13 queue_delete(&queue);
```

■ Příklad výstupu

```
clang queue_linked_list.c demo-queue_linked_list.c && ./a.out
```

```
Add 0 entry ' 77' to the queue r = 0
```

```
Add 1 entry '225' to the queue r = 0
```

```
Add 2 entry '178' to the queue r = 0
```

Prioritní fronta

- Fronta
 - První vložený prvek je první odebraný prvek.

FIFO

- **Prioritní fronta**

- Některé prvky jsou při vyjmutí z fronty preferovány.

Některé vložené objekty je potřeba obsloužit naléhavěji, např. fronta pacientů u lékaře.

- Operace **pop()** odebírá z fronty prvek s nejvyšší prioritou.

Vrchol fronty je prvek s nejvyšší prioritou.

Alternativně též prvek s nejnižší hodnotou.

- Rozhraní prioritní fronty může být identické jako u běžné fronty, avšak specifikace upřesňuje chování dílčích metod.

Prioritní fronta – specifikace rozhraní

- Prioritní frontu můžeme implementovat různě složitě a také s různými výpočetními nároky, např.
 - Polem nebo spojovým seznamem s modifikací funkcí **push()** nebo **pop()** a **peek()**.
 - Například tak, že ve funkci **pop()** a **peek()** projdeme všechny dosud vložené prvky a najdeme prvek nejprioritnější.
 - S využitím pokročilé datové struktury pro efektivní vyhledání prioritního prvku (halda).
- Prioritní prvek může být ten s nejmenší hodnotou.
 - Metody **pop()** a **peek()** vrací nejmenší prvek dosud vložený do fronty.
 - Hodnoty prvků potřebujeme porovnávat, proto potřebujeme funkci pro porovnávání prvků.

Obecně můžeme realizovat například ukazatelem na funkci.

Prioritní fronta – příklad rozhraní

- V implementaci spojového seznamu upravíme funkce `peek()` a `pop()`.

Využijeme přímo kód [lec09/queue_linked_list.h](#), a [lec09/queue_linked_list.c](#).

- Prvek fronty `queue_entry_t` rozšíříme o položku určující prioritu.

Alternativně můžeme specifikovat funkce porovnání datových položek.

- Rozhraní funkcí je identické frontě až na specifikaci priority při vložení prvku do fronty.

```
1 typedef struct entry {
2     void *value;
3
4     // Nová položka
5     int priority;
6
7     struct entry *next;
8 } queue_entry_t;
9
10
11 typedef struct {
12     queue_entry_t *head;
13     queue_entry_t *end;
14 } queue_t;
```

```
16 void queue_init(queue_t **queue);
17 void queue_delete(queue_t **queue);
18 void queue_free(queue_t *queue);
19
20 int queue_push(void *value, int priority, queue_t *
    queue);
21
22 void* queue_pop(queue_t *queue);
23 _Bool queue_is_empty(const queue_t *queue);
24 void* queue_peek(const queue_t *queue);
```

lec09/priority_queue.h

Prioritní fronta spojovým seznamem 1/4

- Ve funkci `push()` přidáme pouze nastavení priority.

```
int queue_push(void *value, int priority, queue_t *queue)
{
    ...
    if (new_entry) { // fill the new_entry
        new_entry->value = value;
        new_entry->priority = priority;
    }
    ...
}
```

`lec09/priority_queue.c`

Prioritní fronta spojovým seznamem 2/4

- `peek()` lineárně prochází seznam a vybere prvek s nejnižší prioritou.

```
38 void* queue_peek(const queue_t *queue)
39 {
40     void *ret = NULL;
41     if (queue && queue->head) {
42         ret = queue->head->value;
43         int lowestPriority = queue->head->priority;
44         queue_entry_t *cur = queue->head->next;
45         while (cur != NULL) {
46             if (lowestPriority > cur->priority) {
47                 lowestPriority = cur->priority;
48                 ret = cur->value;
49             }
50             cur = cur->next;
51         }
52     }
53     return ret;
54 }
```

Prioritní fronta spojovým seznamem 3/4

- Podobně `pop()` lineárně prochází seznam a vybere prvek s nejnižší prioritou, je však nutné zajistit propojení seznamu po vyjmutí prvku.

```
59 void* queue_pop(queue_t *queue)
60 {
61     void *ret = NULL;
62     if (queue->head) { // having at least one entry
63         queue_entry_t* cur = queue->head->next;
64         queue_entry_t* prev = queue->head;
65         queue_entry_t* best = queue->head;
66         queue_entry_t* bestPrev = NULL;
67         while (cur) {
68             if (cur->priority < best->priority) {
69                 best = cur; // update the entry with
70                 bestPrev = prev; // the lowest priority
71             }
72             prev = cur;
73             cur = cur->next;
74         }
75         ...
```

`lec09/priority_queue.c`

- Proto si při procházení pamatujeme předchozí prvek `bestPrev`.

Prioritní fronta spojovým seznamem 4/4

- Po nalezení nejmenšího (největšího) prvku a jeho vyjmutí seznamem propojíme.

```
void* queue_pop(queue_t *queue)
{
    ...
    while (cur) { ... } // Finding the best entry

    if (bestPrev) { // linked the list after
        bestPrev->next = best->next; // best removal
    } else { // best is the head
        queue->head = queue->head->next;
    }
    ret = best->value; //retrive the value
    if (queue->end == best) { //update the list end
        queue->end = bestPrev;
    }
    free(best); // release queue_entry_t
    if (queue->head == NULL) { // update end if last
        queue->end = NULL; // entry has been
    } // popped
}
return ret;
}
```

lec09/priority_queue.c

Prioritní fronta spojovým seznamem – příklad použití 1/2

- Inicializaci fronty provedeme polem textových řetězců a priorit.

```
14 queue_t *queue;
15 queue_init(&queue);
16 char *values[] = { "2nd", "4th", "1st", "5th", "3rd" };
17 int priorities[] = { 2, 4, 1, 5, 3 };
18 const int n = sizeof(priorities) / sizeof(int);
19 for (int i = 0; i < n; ++i) {
20     int r = queue_push(values[i], priorities[i], queue);
21     printf("Add %2i entry '%s' with priority '%i' to the queue\n", i, values[i], priorities[i]);
22     if (r != QUEUE_OK) {
23         fprintf(stderr, "Error: Queue is full!\n");
24         break;
25     }
26 }
27 printf("\nPop the entries from the queue\n");
28 while (!queue_is_empty(queue)) {
29     char* pv = (char*)queue_pop(queue);
30     printf("%s\n", pv);
31     // Do not call free(pv); We pushed text literals into the queue.
32 }
33 queue_delete(&queue);
```

lec09/demo-priority_queue.c

Prioritní fronta spojovým seznamem – příklad použití 2/2

- Hodnoty jsou neuspořádané a očekáváme jejich uspořádaný výpis při vyjmutí funkcí `pop()`.

```
17 char *values[] = { "2nd", "4th", "1st", "5th", "3rd" };
18 int priorities[] = { 2, 4, 1, 5, 3 };
19 ...
20 while (!queue_is_empty(queue)) {
21     // Do not call free(pv);
```

- V tomto případě nevoláme `free()` neboť vložené textové řetězce jsou textovými literály!

Narozdíl od příkladu `lec09/demo-queue_linked_list.c`!

- Příklad výstupu (v tomto případě preferujeme nižší hodnoty):

```
$ make && ./demo-priority_queue
Add 0 entry '2nd' with priority '2' to the queue
Add 1 entry '4th' with priority '4' to the queue
Add 2 entry '1st' with priority '1' to the queue
Add 3 entry '5th' with priority '5' to the queue
Add 4 entry '3rd' with priority '3' to the queue
```

```
Pop the entries from the queue
```

```
1st
```

```
2nd
```

```
3rd
```

```
4th
```

```
5th
```

`lec09/priority_queue.h, lec09/priority_queue.c`

`lec09/demo-priority_queue.c`

Shrnutí přednášky

Diskutovaná témata

- Abstraktní datový typ
- ADT typu zásobník (stack)
- ADT typu fronta (queue)
- Příklady implementací zásobníku a fronty
 - polem
 - rozšiřitelným polem
 - a spojovým seznamem
- Příklady rozhraní a implementace ADT s prvky ukazatel a řešení uvolňování paměti
- Prioritní fronta – příklad implementace spojovým seznamem

- **Příště: Stromy.**