

# 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
- Spojový seznam - zásobník vs. fronta
- Prioritní fronta
- Halda

# 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](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 vracující odkaz (na strukturu nebo objekt).
- Operace, které akceptují odkaz na argument (data) a mají přesně definovaný účinek na data.

*Procedurální i objektově orientovaný přístup.*

## Abstraktní datový typ (ADT) – Vlastnosti

- Počet datových položek může být
  - Neměnný – **statický datový typ** – počet položek je konstantní.
  - Proměnný – **dynamický datový typ** – počet položek se mění v závislosti na provedené operaci.
- 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.

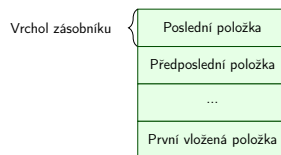
*Např. vložení nebo odebrání určitého prvku*

## 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. repréz.
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`
- Ukážkove 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`.
 

```

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 }

```
- `stack_free()` uvolní paměť vložených položek v zásobníku.
- `stack_delete()` kompletně uvolní paměť alokovanou zásobníkem.
- `stack_free(stack_t *stack)`

```

20 void stack_delete(stack_t **stack)
21 {
22     stack_free(*stack);
23     free(*stack->stack);
24     free(*stack);
25     *stack = NULL;
26 }

```

### 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 }
38
39 void* stack_pop(stack_t *stack)
40 {
41     return stack->count > 0 ? stack->stack[--(stack->count)]: NULL;
42 }
43
44 void* stack_peek(const stack_t *stack)
45 {
46     return stack_is_empty(stack) ? NULL : stack->stack[stack->count - 1];
47 }
48
49 _Bool stack_is_empty(const stack_t *stack)
50 {
51     return stack->count == 0;
52 }

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

```
- V případě zaplnění zásobníku **nezapomenout uvolnit paměť**. `lec09/demo-stack_array.c`

### 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);

```
- 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
Popped value is 83
Popped value is 178
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 funkci 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 přkopíruje a následně paměť uvolní, nebo alokace selže a `realloc()` vrátí `NULL`.
- ```
int stack_push(void *value, stack_t *stack)
{
    int ret = STACK_OK;
    if (stack->count == stack->size) { // try to realloc
        void **tmp = (void**)realloc(
            stack->stack,
            sizeof(void*) * (stack->size + STACK_RESIZE)
        );
        if (tmp) { // realloc has been successful, stack->stack has been eventually freed
            stack->stack = tmp; //
            stack->size += STACK_RESIZE;
        }
        if (stack->count < stack->size) {
            stack->stack[stack->count++] = value;
        } else {
            ret = STACK_MEMFAIL;
        }
        return ret;
    }
}
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. *Viz 8. přednáška.*
  - Definujeme strukturu `stack_entry_t` pro položku seznamu.
- ```
typedef struct entry {
    void *value; //ukazatel na hodnotu vloženého 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 pouze alokuje strukturu stack_t.
```
- ```
void stack_init(stack_t **stack)
{
    *stack = myMalloc(sizeof(stack_t));
    (*stack)->head = NULL;
}
```

### Implementace zásobníku spojovým seznamem 2/3

- 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 = 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 uvolníme.
```
- ```
void* stack_pop(stack_t *stack)
{
    void *ret = NULL;
    if (stack->head) {
        ret = stack->head->value; //retrieve the value
        stack_entry_t *tmp = stack->head;
        stack->head = stack->head->next;
        free(tmp); // release stack_entry_t
    }
    return ret;
}
lec09/stack_linked_list.c
```

### Implementace zásobníku spojovým seznamem 3/3

- Implementace `stack_is_empty()` a `stack_peek()` je triviální.
- ```
_Bool 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;
}
lec09/stack_linked_list.c
lec09/demo-stack_linked_list.c
```
- Použití je identické jako v obou předchozích případech.
  - 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`).

### 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);
```

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### 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`).

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

lec09/queue\_array.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 31 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### 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`.

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 32 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### Příklad implementace fronty spojovým seznamem 1/3

- 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`. Aktualizujeme pouze `end` s konstantní složitostí  $O(1)$ .
  - Odebrání prvku z fronty `queue_pop()` vezmeme prvek z počátku seznamu `head`. Aktualizujeme pouze `head` s konstantní složitostí  $O(1)$ .
  - Nemusíme lineárně procházet seznam a aktualizovat `end` při odebrání prvku z fronty.

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

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

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

lec09/queue\_linked\_list.h  
lec09/queue\_linked\_list.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 33 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### Implementace fronty spojovým seznamem 2/3

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

```
int queue_push(void *value, queue_t *queue)
{
    int ret = QUEUE_OK;
    queue_entry_t *new_entry = 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;
}
```

lec09/queue\_linked\_list.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 34 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### Implementace fronty spojovým seznamem 3/3

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

```
void* queue_pop(queue_t *queue)
{
    void *ret = NULL;
    if (queue->head) { // having at least one entry
        ret = queue->head->value; // retrieve 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;
}
```

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

```
_Bool 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;
}
```

lec09/queue\_linked\_list.c

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 35 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### ADT – Fronta spojovým seznamem – příklad použití

```
for (int i = 0; i < 3; ++i) {
    int *pv = getRandomInt();
    int r = queue_push(pv, queue);
    printf("Add %2i entry '%3i' to the queue r = %i\n", i, *pv, r);
    if (r != QUEUE_OK) { free(pv); break; } // release allocated pv
}

printf("\nPop the entries from the queue\n");
while (!queue_is_empty(queue)) {
    int *pv = (int*)queue_pop(queue);
    printf("Popped value is %3i\n", *pv);
    free(pv);
}

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
```

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 36 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### ADT – zásobník a fronta

- Obě datové struktury mají stejné rozhraní, např. `push()`, `pop()`, `isEmpty()`.
- Zásobník vs. fronta se liší chováním, tj. jaký prvek vrací při vyjmutí.
- Obě struktury můžeme implementovat polem nebo spojovým seznamem.
- Implementace polem (definované kapacity)
  - Zásobník inkrementujeme/dekrementujeme pouze index na volný prvek v poli.
  - Frontu implementujeme kruhovou frontou v poli, indexy na první a poslední prvek pouze inkrementujeme modulu kapacita pole.
- Postací jednosměrný spojový seznam, implementace se liší kam přidáváme nové prvky.
  - Přidávání je snadné před první prvek (`head`) nebo za poslední prvek.
  - Odebrání je snadné pro první prvek (`head`).

Zásobník

Fronta

■ `push()` i `pop()` realizujeme přes `head`. ■ `push()` realizujeme přes `tail` a `pop()` přes `head`.

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 38 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### Prioritní fronta

- Fronta
  - První vložený prvek je první odebraný prvek.
- 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.

FIFO

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 40 / 56

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda

### 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 pokročilých 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.

Jan Faigl, 2024 BAB36PRGA – Přednáška 09: Abstraktní datový typ 41 / 56

### Prioritní fronta spojovým seznamem nebo polem a výpočetní náročnost

- V naivní implementaci prioritní fronty můžeme zohlednění priority „odložit“ až do doby, kdy potřebujeme odebrat prvek z fronty. *Použijeme „lazy“ (odložený) výpočet.*
- Při odebrání (nebo vrácení) nejmenšího prvku v nejnepříznivějším případě musíme projít všechny položky.
- To může být **výpočetně náročné** a raději bychom chtěli „udržovat“ prvek připravený.
  - Můžeme to například udělat zavedením položky **head**, ve které bude aktuálně nejnižší (nejvyšší) vložený prvek do fronty.
  - Prvek **head** aktualizujeme v metodě **push()** porovnáním hodnoty aktuálně vkládaného prvku.
  - Tím zefektivníme operaci **peek()**.
- V případě odebrání prvku, však musíme frontu znovu projít a najít nový prvek. *Nebo můžeme použít sofistikovanější datovou strukturu, která nám umožní efektivně udržovat hodnotu nejmenšího prvku a to jak při operaci vložení **push()** tak při operaci vyjmutí **pop()** prvku z prioritní fronty.*

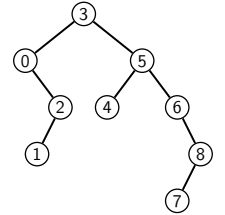
### Halda

- Halda je dynamická datová struktura, která má „tvar“ binárního stromu a uspořádání prioritní fronty.
- Každý prvek haldy obsahuje hodnotu a dva potomky, podobně jako binární strom.
- Vlastnosti haldy – „Heap property“.**
  - Hodnota každého prvku je menší než hodnota libovolného potomka.
  - Každá úroveň binárního stromu haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava. **Binární plný strom**
  - Prvky mohou být odebrány pouze přes kořenový uzel.
- Vlastnost haldy zajišťuje, že **kořen je vždy prvek s nejnižším/nejvyšším ohodnocením.**
  - V případě binárního plného stromu je složitost procházení úměrná hloubce stromu, která je pro  $n$  prvků úměrná  $\log_2(n)$ . Složitost operací **push()**, **pop()**, **peek()** tak můžeme očekávat nikoliv  $O(n)$  (jako v případě předchozí implementace prioritní fronty polem a spojovým seznamem), ale  $O(\log n)$  a pro **peek()** dokonce  $O(1)$ .

### Binární vyhledávací strom vs halda

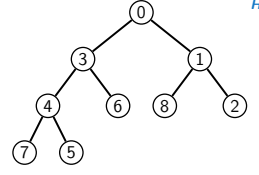
#### Binární vyhledávací strom

- Může obsahovat prázdná místa.
- Hloubka stromu se může měnit. *Zajistit vyvážený strom je implementačně náročnější než implementace haldy.*



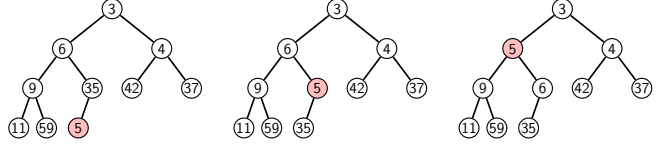
#### Halda

- Binární plný strom *Hloubka stromu vždy  $\lfloor \log_2(n) \rfloor$ .*
- Kořen stromu je vždy prvek s nejnižší (nejvyšší) hodnotou.
- Každý podstrom splňuje vlastnost haldy. *Heap property*



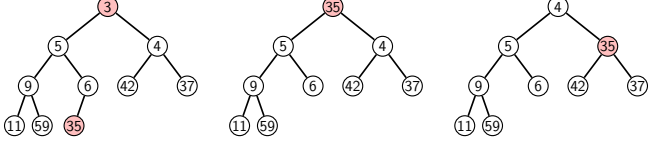
### Halda – přidání prvku push()

- Po každém provedení operace **push()** musí být splněny vlastnosti haldy.
- Prvek přidáme na konec haldy, tj. na první volnou pozici (vlevo) na nejnižší úrovni haldy.
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s nadřazeným prvkem (předkem). *V nejnepříznivějším případě prvek „probublá“ až do kořene stromu.*



### Halda – odebrání prvku pop()

- Při operaci **pop()** odebereme kořen stromu.
- Prázdné místo nahradíme nejpravějším listem.
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s potomkem a postup opakujeme. *V nejnepříznivějším případě prvek „probublá“ až do listu stromu.*



- Jak zjistit nejpravější list?
  - V případě implementace spojovou strukturou (nelineární) můžeme explicitně udržovat odkaz.
  - Binární plný strom můžeme efektivně reprezentovat polem** – pak nejpravější list je poslední prvek v poli.

### Prioritní fronta haldou

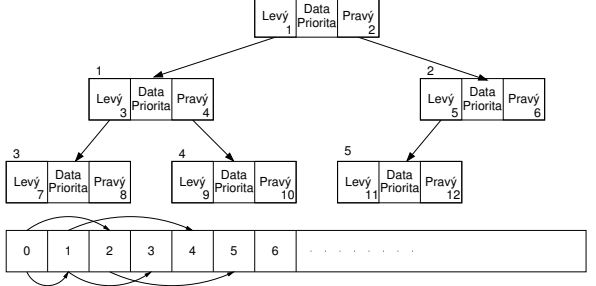
- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti **haldy**.
- Operace **peek()** má konstantní složitost a nezáleží na počtu prvků ve frontě, nejnižší prvek je vždy kořen.

*Asymptotická složitost v notaci velké O je  $O(1)$ .*

- Operace **push()** a **pop()** udržují vlastnost haldy záměnami prvku až do hloubky stromu. *Pro binární plný strom je hloubka stromu  $\log_2(n)$ , kde  $n$  je aktuální počet prvků ve stromu, odtud složitost operace  $O(\log(n))$ .*

### Reprezentace binárního stromu polem

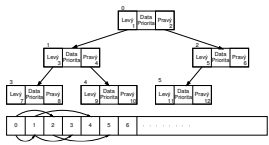
- Binární plný strom můžeme reprezentovat lineární strukturou.
- V případě známého maximálního počtu prvků v haldě, pak jednoduše předalokovaným polem.



### Halda jako binární plný strom reprezentovaný polem

- Pro definovaný maximální počet prvků v haldě si předalokujeme pole o daném počtu prvků.
- Binární **plný strom** má všechny vrcholy na úrovni rovné hloubce stromu co nejvíce vlevo.
- Kořen stromu je první prvek s indexem 0, následníky prvku na pozici  $i$  lze v poli určit jako prvky s indexy:
  - levý následník:  $i_{levý} = 2i + 1$ ;
  - pravý následník:  $i_{pravý} = 2i + 2$ .

*Podobně lze odvodit vztah pro předchůdce.*



- Kořen stromu reprezentuje **nejprioritnější prvek.**

*Např. s nejmenší hodnotou nebo maximální prioritou.*

### Operace vkládání a odebrání prvků

- I v případě reprezentace polem pracují operace vkládání a odebrání identicky.
  - Funkce **push()** přidá prvek jako další prvek v poli a následně propaguje prvek směrem nahoru až je **splněna vlastnost haldy**.
  - Při odebrání prvku funkcí **pop()** je poslední prvek v poli umístěn na začátek pole (kořen stromu) a propagován směrem dolů až je **splněna vlastnost haldy**.
- Dochází pouze k vzájemnému zaměňování hodnot na pozicích v poli (haldě). *Z indexu prvku v poli vždy můžeme určit jak levého a pravého následníka, tak i předcházející prvek (rodič) v pohledu na haldu jako binární strom.*
- Hlavní výhodou reprezentace polem je přístup do předem alokovaného bloku paměti. *Všechny prvky můžeme jednoduše projít v jedné smyčce, například při výpisu.*
- Ověření zdali implementace operací **push()** a **pop()** zachovává **podmínku haldy** můžeme realizovat ověřující funkcí **is\_heap()**.

```

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda
Příklad implementace pq_is_heap()
■ Pro každý prvek haldy musí platit, že jeho hodnota je menší než levý i pravý následník.
18 typedef struct {
19     int size; // the maximal number of entries
20     int len; // the current number of entries
21     int *cost; // array with costs - lowest cost is highest priority
22     int *label; // array with labels (each label has cost/priority)
23 } pq_heap_s;
161 _Bool pq_is_heap(pq_heap_s *pq, int n)
162 {
163     _Bool ret = true;
164     int l = 2 * n + 1; // left successor
165     int r = l + 1; // right successor
166     if (l < pq->len) {
167         ret = (pq->cost[l] < pq->cost[n]) ? false : pq_is_heap(pq, l);
168     }
169     if (r < pq->len) {
170         ret = ret // if ret is false, further expression is not evaluated
171         && (pq->cost[r] < pq->cost[n]) ? false : pq_is_heap(pq, r);
172     }
173     return ret;
174 }
175 }

```

```

Datové struktury Zásobník Fronta Spojový seznam - zásobník vs. fronta Prioritní fronta Halda
Příklad implementace push()
■ Prvek přidáme na konec pole a iterativně kontrolujeme, zdali je splněna vlastnost haldy. Pokud ne, prvek zaměníme s předchůdcem.
41 #define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2
42
43 _Bool pq_push(pq_heap_s *pq, int label, int cost)
44 {
45     _Bool ret = false;
46     if (pq && pq->len < pq->size && label >= 0 && label < pq->size) {
47         pq->cost[pq->len] = cost; //add the cost to the next free slot
48         pq->label[pq->len] = label; //add label of new entry
49         int cur = pq->len; // index of the entry added to the heap
50         int parent = GET_PARENT(cur);
51         while (cur > 1 && pq->cost[parent] > pq->cost[cur]) {
52             pq_swap(pq, parent, cur); // swap parent<->cur
53             cur = parent;
54             parent = GET_PARENT(cur);
55         }
56         pq->len += 1;
57         ret = true;
58     }
59     // assert(pq_is_heap(pq, 0)); // testing the implementation
60     return ret;
61 }
62 }

```

**Příklad volání pop()**

- Halda je reprezentovaná binárním polem.
- Nejmenší prvek je kořenem stromu.
- Voláním pop() odebíráme kořen stromu.
- Na jeho místo umístíme poslední prvek.
- Strom však nesplňuje podmínku haldy.
- Proto provedeme záměnu s následníky.
  - V tomto případě volíme pravého následníka, neboť jeho hodnota je nižší než hodnota levého následníka.
- A strom opět splňuje vlastnost haldy.
- Záměny provádíme v poli a využíváme vlastnosti plného binárního stromu.
  - Levý potomek prvku haldy na pozici i je 2i+1, pravý potomek je na pozici 2i+2.

Diskutovaná témata

## Shrnutí přednášky

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

Kódovací příklad – Prioritní fronta polem

## Část III

### Appendix

Kódovací příklad – Prioritní fronta spojovým seznamem

```

Kódovací příklad – Prioritní fronta polem
Prioritní fronta polem – rozhraní
■ V případě implementace prioritní fronty polem můžeme využít jedno pole pro hodnoty a druhé pole pro uložení priority daného prvku.
Implementace vychází z lec09/queue/queue_array.h a lec09/queue/queue_array.c
typedef struct {
    void **queue; // Pole ukazatelů na jednotlivé prvky
    int *priorities; // Pole hodnot priorit jednotlivých prvků
    int count; // Uvažujeme pouze MAX_INT prvků, zpravidla 2147483647
    int head;
    int tail;
} queue_t;
■ Další rozhraní (jména a argumenty funkcí) mohou zůstat identické jako u implementace spojovým seznamem.
void queue_init(queue_t **queue); // Pole ukazatelů na jednotlivé prvky
void queue_delete(queue_t **queue); // Pole hodnot priorit jednotlivých prvků
void queue_free(queue_t *queue);
_Bool queue_is_empty(const queue_t *queue);
int queue_push(void *value, int priority, queue_t *queue);
void *queue_pop(queue_t *queue);
void *queue_peek(const queue_t *queue);

```

```

Kódovací příklad – Prioritní fronta polem
Prioritní fronta polem 1/3 – push()
■ Funkce push() je až na uložení priority identická s verzí bez priorit.
46 int queue_push(void *value, int priority, queue_t *queue)
47 {
48     int ret = QUEUE_OK; // by default we assume push will be OK
49     if (queue->count < MAX_QUEUE_SIZE) {
50         queue->queue[queue->tail] = value;
51         queue->priorities[queue->tail] = priority; // store priority of the new value entry
52         queue->tail = (queue->tail + 1) % MAX_QUEUE_SIZE;
53         queue->count += 1;
54     } else {
55         ret = QUEUE_MEMFAIL;
56     }
57     return ret;
58 }
lec09/priority_queue_array/priority_queue_array.c
■ Funkce peek() a pop() potřebují prvek s nejnižší (nejvyšší) prioritou.
■ Nalezení prvku z „čela“ fronty realizujeme funkcí getEntry(), kterou následně využijeme jak v peek(), tak v pop().

```

```

Kódovací příklad – Prioritní fronta polem
Prioritní fronta polem 2/3 – getEntry()
■ Nalezení nejmenšího (největšího) prvku provedeme lineárním prohledáním aktuálních prvků uložených ve frontě (poli).
61 static int getEntry(const queue_t *const queue)
62 {
63     int ret = -1; // return -1 if queue is empty.
64     if (queue->count > 0) {
65         for (int cur = queue->head, i = 0; i < queue->count; ++i) {
66             if (
67                 ret == -1 ||
68                 (queue->priorities[ret] > queue->priorities[cur])
69             ) {
70                 ret = cur;
71             }
72             cur = (cur + 1) % MAX_QUEUE_SIZE;
73         }
74     }
75     return ret;
76 }
lec09/priority_queue_array/priority_queue_array.c

```



## Prioritní fronta polem 3/3 – peek() a pop()

- Funkce `peek()` využívá lokální (static) funkci `getEntry()`.

```
101 void* queue_peek(const queue_t *queue)
102 {
103     return queue_is_empty(queue) ? NULL : queue->queue[getEntry(queue)];
104 }
```

- Ve funkci `pop()` zaplníme položku vyjmutého prvku prvkem ze startu.

```
77 void* queue_pop(queue_t *queue) Tim zajistíme, že prvky tvoří souvislý blok v rámci kruhové fronty.
78 {
```

```
79     void *ret = NULL;
80     int bestEntry = getEntry(queue);
81     if (bestEntry >= 0) { // entry has been found
82         ret = queue->queue[bestEntry];
83         if (bestEntry != queue->head) { //replace the bestEntry by head
84             queue->queue[bestEntry] = queue->queue[queue->head];
85             queue->priorities[bestEntry] = queue->priorities[queue->head];
86         }
87         queue->head = (queue->head + 1) % MAX_QUEUE_SIZE;
88         queue->count -= 1;
89     }
90     return ret;
91 }
```

## 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-linked_list/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 }
```

```
lec09/priority_queue-linked_list/priority_queue.c
```

## 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         ...
76     }
```

```
lec09/priority_queue-linked_list/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; //retrieve 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-linked_list/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/priority_queue-linked_list/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!

*Narodil 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-linked_list/priority_queue.h
lec09/priority_queue-linked_list/priority_queue.c
lec09/priority_queue-linked_list/demo-priority_queue.c
BAB36PRGA – Přednáška 09: Abstraktní datový typ
```