Abstraktní datový typ

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

B0B36PRP – Procedurální programování

Přehled témat

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 Sedgewick 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 které 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.

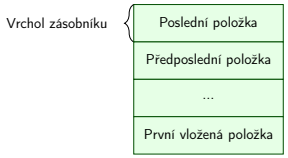
Příklad ADT – Zásobník

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 prázdnost 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 prázdnost zásobníku.

Další operace nad zásobníkem mohou být

- **peek()** – čtení hodnoty z vrcholu zásobníku;
  *Alternativně také třeba **top()**.*
- *search()* – vrátí pozici prvku v zásobníku;
  Pokud se nachází v zásobníku, jinak -1.
- *size()* – vrátí aktuální počet prvků (hodnot) v zásobníku.
  Zpravidla není potřeba.

Příklad ADT – Rozhraní zásobníku 1/2

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

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

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

```
/*
 * Function: stack_push
 * --------------------
 * This routine push the given value onto the top of the stack.
 *
 *  value - value to be placed on the stack
 *  stack - stack to push
 *
 *  returns: The function returns status value:
 *
 *  OK           - success
 *  CLIB_MEMFAIL - dynamic memory allocation failure
 *
 *  This function requires the following include files:
 *
 *  prp_stack.h prp_errors.h
 */
int stack_push(void *value, void **stack);
```


Implementace zásobníku

- Součástí ADT není volba konkrétní implementace – zásobník můžeme implementovat např.
  - 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.
  - `lec11/stack_array.h`, `lec11/stack_array.c`
  - `lec11/stack_array_alloc.h`, `lec11/stack_array_alloc.c`
  - `lec11/stack_linked_list.h`, `lec11/stack_linked_list.c`
- Dále si ukážeme použití maker preprocesoru a jejich definici při překladu.
- Ukázkové implementace také slouží jako demonstrátory 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 odkazující na jednotlivé prvky uložené do zásobníku.

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

- Pro inicializaci a uvolnění paměti implementujeme pomocné funkce.

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

- Základní operace se zásobníkem mají tvar

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

- a jsou pro všechny tři implementace totožné.

lec11/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`.

```
#ifndef MAX_STACK_SIZE
#define MAX_STACK_SIZE 5
#endif

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

```
#include <stdlib.h>

void* myMalloc(size_t size)
{
   void *ret = malloc(size);
   if (!ret) {
      fprintf(stderr, "Malloc failed!\n");
      exit(-1)
   }
   return ret;
}
```

lec11/my_malloc.c

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

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

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

lec11/stack_array.c

Implementace zásobníku polem 3/3

```
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];
}

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

Proč v metodě `pop()` *používáme (--(stack→count)) a v* `peek()` *count −1?*

Zásobník – Příklad použití 1/3

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

```
int* getRandomInt()
{
   int *r = myMalloc(sizeof(int)); // dynamicky alokovaný int
   *r = rand() % 256;
   return r;
}
stack_t *stack;
stack_init(&stack);

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

lec11/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.

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

lec11/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                                    lec11/stack_array.h
Popped value is  83
Popped value is 178                                    lec11/stack_array.c
Popped value is 225
Popped value is  77                                    lec11/demo-stack_array.c
```

- 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.

Viz `man realloc`

```
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
         stack->stack = tmp; // has been freed
         stack->size += STACK_RESIZE;
      }
   }
   if (stack->count < stack->size) {
      stack->stack[stack->count++] = value;
   } else {
      ret = STACK_MEMFAIL;
   }
   return ret;
}
```

lec11/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
```

lec11/stack_array-alloc.h
lec11/stack_array-alloc.c
lec11/demo-stack_array-alloc.c

Implementace zásobníku spojovým seznamem 1/3

- Zásobník také můžeme implementovat spojovým seznamem, viz 9. 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 tak 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 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;
}
```

lec11/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;
}
```

lec11/stack_linked_list.c

- Použití je identické jako v obou předchozích případech.

lec11/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 odebírání 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

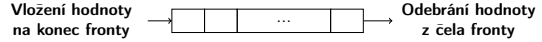
```
#if STACK_ARRAY
# include "stack_array.h"
#elif STACK_ARRAY_ALLOC
# include "stack_array-alloc.h"
#elif STACK_LINKED_LIST
#include "stack_linked_list.h"
#endif
```

lec11/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).



- Implementace
  - Pole – *Pamatujeme si pozici začátku a konce fronty v poli.*
    - Pozice cyklicky rotují (modulo velikost pole).
  - 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 9. přednášky
    - Nebo přidáváme na konec a odebíráme ze začátku (`head`).
      `pushEnd()` a `pop()` z 9. 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 prázdnost fronty .
- Další operace mohou být
  - **peek()** – čtení hodnoty z čela fronty;
  - *size()* – vrátí aktuální počet prvků ve frontě.
- Hlavní rozdíl je v operacích `pop()` a `peek()`, které vracejí nejdříve vložený prvek do fronty.

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

ADT – Příklad implementace fronty

- Implementace fronty pole a spojovým seznamem.
- Využijeme shodné rozhraní a jméno typu `queue_t` definované v samostatných modulech.
  - `lec11/queue_array.h`, `lec11/queue_array.c`
  - `lec11/queue_linked_list.h`, `lec11/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 $0 \leq$ `end` $<$ `MAX_QUEUE_SIZE`.

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

`lec11/queue_array.c`

Příklad implementace fronty 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];
}
```

`lec11/queue_array.c`

- Příklad použití viz `lec11/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.
  - Vložením 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 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 = myMalloc(sizeof(queue_t));
  (*queue)->head = NULL;
  (*queue)->end = NULL;
}
```

`lec11/queue_linked_list.h`
`lec11/queue_linked_list.c`

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;
}
```

`lec11/queue_linked_list.c`

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; //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;
}
```

- `isEmpty()` a `peek()` je triviální.

```
_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;
}
```

`lec11/queue_linked_list.c`

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

Pop the entries from the queue
Popped value is  77
Popped value is 225
Popped value is 178
```

`lec11/queue_linked_list.h`
`lec11/queue_linked_list.c`
`lec11/demo-queue_linked_list.c`

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()**.

Základní implementace fronty viz předchozí přednáška.

    - 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, pak
  - 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 `lec11/queue_linked_list.h`*,a* `lec11/queue_linked_list.c`.

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

  Alternativně můžeme specifikovat funkce porovnání datavých položek.

```
typedef struct entry {
   void *value;

   // Nová položka
   int priority;

   struct entry *next;
} queue_entry_t;

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

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

```
void queue_init(queue_t **queue);
void queue_delete(queue_t **queue);
void queue_free(queue_t *queue);

int queue_push(void *value, int priority, queue_t *queue);

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

lec11/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;
   ...
```

lec11/priority_queue.c

Prioritní fronta spojovým seznamem 2/4

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

```
void* queue_peek(const queue_t *queue)
{
   void *ret = NULL;
   if (queue && queue->head) {
      ret = queue->head->value;
      int lowestPriority = queue->head->priority;
      queue_entry_t *cur = queue->head->next;
      while (cur != NULL) {
         if (lowestPriority > cur->priority) {
            lowestPriority = cur->priority;
            ret = cur->value;
         }
         cur = cur->next;
      }
   }
   return ret;
}
```

lec11/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.

```
void* queue_pop(queue_t *queue)
{
   void *ret = NULL;
   if (queue->head) { // having at least one entry
      queue_entry_t* cur = queue->head->next;
      queue_entry_t* prev = queue->head;
      queue_entry_t* best = queue->head;
      queue_entry_t* bestPrev = NULL;
      while (cur) {
         if (cur->priority < best->priority) {
            best = cur; // update the entry with
            bestPrev = prev;  // the lowest priority
         }
         prev = cur;
         cur = cur->next;
      }
      ...
```

lec11/priority_queue.c

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

Prioritní fronta spojovým seznamem 4/4

- Po nalezení největšího (nejmenšího) prvku propojíme seznam.

```
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;
}
```

lec11/priority_queue.c

Prioritní fronta spojovým seznamem – příklad použití 1/2

- Inicializaci fronty provedeme polem textových řetězců a priorit.

```
queue_t *queue;
queue_init(&queue);
char *values[] = { "2nd", "4th", "1st", "5th", "3rd" };
int priorities[] = { 2, 4, 1, 5, 3 };
const int n = sizeof(priorities) / sizeof(int);
for (int i = 0; i < n; ++i) {
   int r = queue_push(values[i], priorities[i], queue);
   printf("Add %2i entry '%s' with priority '%i' to the queue\n", i, values[i],
    priorities[i]);
   if (r != QUEUE_OK) {
      fprintf(stderr, "Error: Queue is full!\n");
      break;
   }
}
printf("\nPop the entries from the queue\n");
while (!queue_is_empty(queue)) {
   char* pv = (char*)queue_pop(queue);
   printf("%s\n", pv);
   // Do not call free(pv);
}
queue_delete(&queue);
```

lec11/demo-priority_queue.c

Prioritní fronta spojovým seznamem – příklad použití 2/2

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

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

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

  Narozdíl od příkladu `lec11/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
```

lec11/priority_queue.h, lec11/priority_queue.c

lec11/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: Prioritní fronta – polem a haldou. Příklad využití prioritní fronty (haldy) v úloze hledání nejkratší cesty v grafu.