

# Spojové struktury

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 09

B0B36PRP – Procedurální programování

## Část I

### Část 1 – Spojové struktury

## Přehled témat

### ■ Část 1 – Spojové struktury

Spojové struktury

Spojový seznam

Spojový seznam s odkazem na konec seznamu

Vložení/odebrání prvku

Kruhový spojový seznam

Obousměrný seznam

### ■ Část 2 – Zadání 8. domácího úkolu (HW08)

## Kolekce prvků (položek)

### ■ V programech je velmi běžný požadavek na uchování seznamu (množiny) prvků (proměnných/struktur)

### ■ Základní kolekce je pole

*Definované jménem typu a [], například `double []`*

#### ■ Jedná se o kolekci položek (proměnných) stejného typu

+ Umožňuje jednoduchý přístup k položkám indexací prvku

*Položky jsou stejného typu (velikosti)*

– Velikost pole je určena při vytvoření pole

#### ■ Velikost (maximální velikost) musí být známa v době vytváření

#### ■ Změna velikost v podstatě není přímo možná

*Nutné nové vytvoření (alokace paměti), resp. `realloc`*

#### ■ Využití pouze malé části pole je mrháním paměti

### ■ V případě řazení pole přesouváme položky

#### ■ Vložení prvku a vyjmutí prvku vyžaduje kopírování

*Kopírování objemných prvků lze případně řešit ukazatelem.*

## Seznam – list

- Seznam (proměnných nebo objektů) patří mezi základní datové struktury
  - Základní **ADT** – Abstract Data Type
- Seznam zpravidla nabízí sadu základních operací:
  - Vložení prvku (**insert**)
  - Odebrání prvku (**remove**)
  - Vyhledání prvku (**indexOf**)
  - Aktuální počet prvku v seznamu (**size**)
- Implementace seznamu může být různá:
  - Pole
    - Indexování je velmi rychlé
    - Vložení prvku na konkrétní pozici může být pomalé
  - Spojivé seznamy**
    - Nová alokace a kopírování

## Základní operace se spojovým seznamem

- Vložení prvku
  - Předchozí prvek odkazuje na nový prvek
  - Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje
    - Tzv. *obousměrný spojový seznam*
- Odebrání prvku
  - Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
  - Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek
- Základní implementací spojového seznamu je tzv. **jednosměrný spojový seznam**

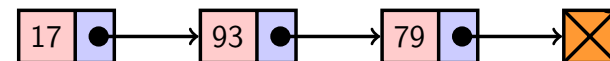
## Spojivé seznamy

- Datová struktura realizující seznam dynamické délky
- Každý prvek seznamu obsahuje
  - Datovou část (hodnota proměnné / objekt / ukazatel na data)
  - Odkaz (ukazatel) na další prvek v seznamu
    - NULL v případě posledního prvku seznamu.
- První prvek seznamu se zpravidla označuje jako **head** nebo **start**
  - Realizujeme jej jako ukazatel odkazující na první prvek seznamu

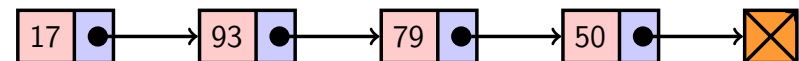


## Jednosměrný spojový seznam

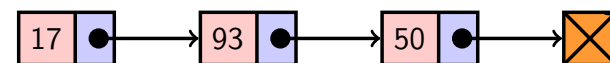
- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 50 na konec seznamu



- Odebrání prvku 79



- Nejdříve sekvenčně najdeme prvek s hodnotou 79
- Následně vyjme a napojíme prvek 93 na prvek 50

Hodnotu next prvku 93 nastavíme na hodnotu next odebíraného prvku, tj. na prvek 50

## Spojový seznam

- Seznam tvoří struktura prvku
  - Vlastní data prvku
  - Odkaz (ukazatel) na další prvek
- Vlastní seznam
  - Ukazatel na první prvek `head`
  - nebo vlastní struktura pro seznam

*Vhodné pro uložení dalších informací, počet prvků, poslední prvek.*

- Příklad tříd pro uložení spojového seznamu celých čísel

|  |  |
|--|--|
| <pre>typedef struct entry {     int value;     struct entry *next; } entry_t;  entry_t *head = NULL;</pre> | <p>Vlastní struktura, například</p> <pre>typedef struct {     entry_t *head;     entry_t *tail;     int counter; // pocet                     prvku } linked_list_t;</pre> |
|--|--|

- Pro jednoduchost prvky seznamu obsahují celé číslo.

*Obecně mohou obsahovat libovolná data (ukazatel na strukturu).*

## Spojový seznam – push()

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

```
void push(int value, entry_t **head)
{ // add new entry at front
    entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));

    new_entry->value = value; // set data
    if (*head == NULL) { // first entry in the list
        new_entry->next = NULL; // reset the next
    } else {
        new_entry->next = *head;
    }
    *head = new_entry; //update the head
}
```

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

- Přidání prvku není závislé na počtu prvků v seznamu
  - Konstantní složitost operace `push()` –  $O(1)$*

## Přidání prvku – příklad

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

```
head = (entry_t*)malloc(sizeof(entry_t));
head->value = 10;
head->next = NULL;
```
  - Další prvek (13) přidáme propojením s aktuálně 1. prvkem
 

```
entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
new_entry->value = 13;
new_entry->next = head;
```
  - a aktualizací proměnné `head`

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

## Spojový seznam – pop()

- Odebrání prvního prvku ze seznamu

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

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

- Odebrání prvku není závislé na počtu prvků v seznamu
  - Konstantní složitost operace `pop()` –  $O(1)$*

## Spojový seznam – size()

- Zjištění počtu prvků v seznamu vyžaduje projít seznam až k zarážce `NULL`, tj. položka `next` je `NULL`

- Proměnnou `cur` používáme jako „kurzor“ pro procházení seznamu

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

*Použijeme konstantní ukazatel na konstantní proměnnou, neboť seznam pouze procházíme a nemodifikujeme. Z hlavičky funkce je tak zřejmé, že vstupní strukturu ve funkci nemodifikujeme.*

- Pro zjištění počtu prvků v seznamu musíme projít kompletní seznam, tj.  $n$  položek  
*Lineární složitost operace `size()` –  $O(n)$*

## Spojový seznam – procházení seznamu

- Procházení seznamu demonstrujeme na funkci `print()`

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

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

*Z hlavičky funkce je zřejmé, že vstupní strukturu nemodifikujeme.*

- Prvky seznamu tiskneme za sebou oddělené mezerou a poslední prvek je zakončen znakem nového řádku

## Spojový seznam – back()

- Vrácení hodnoty posledního prvku ze seznamu – `back()`

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

- Pro vrácení hodnoty posledního prvku v seznamu musíme projít všechny položky seznamu  
*Lineární složitost operace `back()` –  $O(n)$*

## Příklad – jednoduchý spojový seznam

```
entry_t *head;
head = NULL; // initialization is important

push(17, &head);
push(7, &head);
printf("List: ");
print(head);
push(5, &head);
printf("\nList size: %i\n", size(head));
printf("Last entry: %i\n\n", back(head));
printf("List: ");
print(head);
push(13, &head);
push(11, &head);
pop(&head);
printf("List:r");
print(head);
printf("\nPop until head is not empty\n");
while (head != NULL) {
  const int value = pop(&head);
  printf("Popped value %i\n", value);
}
printf("List size: %i\n", size(head));
printf("Last entry value %i\n", back(head));
```

```
clang -g demo-
simple_linked_list.c
simple_linked_list.c
./a.out
List: 7 17
List size: 3
Last entry: 17
List: 5 7 17
List: 13 5 7 17
Cleanup using pop until
head is not empty
Popped value 13
Popped value 5
Popped value 7
Popped value 17
List size: 0
lec09/simple_linked_list.h
lec09/simple_linked_list.c
lec09/demo-simple_linked_list.c
```

## Spojový seznam – zrychlení operací `size()` and `back()`

- Operace `size()` a `back()` procházejí kompletní seznam
- Operaci `size()` můžeme urychlit pokud budeme udržovat aktuální počet položek v seznamu
  - Zavedeme datovou položku `int counter`
  - Počet prvků inkrementujeme při každém přidání prvku a dekrementujeme při každém odebrání prvku
- Operaci `back()` můžeme urychlit proměnou odkazující na poslední prvek
- Zavedeme strukturu pro vlastní spojový seznam s položkami `head`, `counter`, and `tail`

```
typedef struct {
    entry_t *head;
    entry_t *tail;
    int counter;
} linked_list_t;
```

- V případě přidání prvku na začátek, aktualizujeme pouze pokud byl seznam doposud prázdný
- Aktualizujeme v případě přidání prvku na konec
- Nebo při vyjmutí posledního prvku

## Spojový seznam – `push()` s odkazem na konec seznamu

```
void push(int value, linked_list_t *list)
{ // add new entry at front
    assert(list);
    entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
    new_entry->value = value; // set data
    if (list->head) { // an entry already in the list
        new_entry->next = list->head;
    } else { //list is empty
        new_entry->next = NULL; // reset the next
        list->tail = new_entry; //1st entry is the tail
    }
    list->head = new_entry; //update the head
    list->counter += 1; // keep counter up to date
}
```

*Hodnotu ukazatele `tail` nastavujeme pouze pokud byl seznam prázdný, protože prvky přidáváme na začátek.*

## Spojový seznam – urychlený `size()`

- Samostatná struktura pro seznam
- Položky `head` a `counter`
- `head` je ukazatel na `entry_t`
- Ve funkci `size()` předpokládáme validní odkaz na seznam
- Proto voláme `assert(list)`
- Přímá inicializace `linked_list_t linked_list = { NULL, 0 };`
- Do funkcí `push()` a `pop()` stačí předávat pouze ukazatel, proto pro zjednodušení použijeme proměnnou `list`

```
typedef struct {
    entry_t *head;
    int counter;
} linked_list_t;

int size(const linked_list_t *list)
{
    assert(list);
    return list->counter;
}
```

- Pro urychlení funkce `size()` stačí inkrementovat a dekrementovat proměnnou `counter` ve funkcích `push()` a `pop()`

```
void push(int data, linked_list_t *list)
{
    ...
    list->counter += 1;
}

int pop(linked_list_t *list)
{
    ...
    list->counter -= 1;
    return ret;
}
```

## Spojový seznam – `pop()` s odkazem na konec seznamu

```
int pop(linked_list_t *list)
{
    assert(list && list->head); // non-empty list
    entry_t *prev_head = list->head; // save head
    list->head = prev_head->next;
    list->counter -= 1; // keep counter up to date
    int ret = prev_head->value;
    free(prev_head); // relase the memory
    if (list->head == NULL) { // end has been popped
        list->tail = NULL;
    }
    return ret;
}
```

*Hodnotu proměnné `tail` nastavujeme pouze pokud byl odebrán poslední prvek, protože prvky odebíráme ze začátku.*

## Spojový seznam – back() s odkazem na konec seznamu

- Proměnná `tail` je buď `NULL` nebo odkazuje na poslední prvek seznamu

```
int back(const linked_list_t *const list)
{
    // const we do not allow to call back on empty list
    assert(list && list->tail);
    return list->tail->value;
}
```

- Udržováním hodnoty proměnné `tail` (ve funkcích `push()` a `pop()`) jsme snížili časovou náročnost operace `back()` z lineární složitosti na počtu prvků ( $n$ ) v seznamu  $O(n)$  na konstantní složitost  $O(1)$ .

## Spojový seznamu – popEnd()

- Odebrání prvku z konce seznamu

```
int popEnd(linked_list_t *list)
{
    assert(list && list->head);
    entry_t *end = list->tail; // save the end
    if (list->head == list->tail) { // the last entry is
        list->head = list->tail = NULL; // removed
    } else { // there is also penultimate entry
        entry_t *cur = list->head; // that needs to be
        while (cur->next != end) { // updated (its next
            cur = cur->next; // pointer to the next entry
        }
        list->tail = cur;
        list->tail->next = NULL; //the tail does not have
        next
    }
    int ret = tail->value;
    free(end);
    list->counter -= 1;
    return ret;
}
```

*Složitost je  $O(n)$ , protože musíme aktualizovat předposlední prvek. Alternativně lze řešit obousměrným spojovým seznamem.*

## Spojový seznamu – pushEnd()

- Přidání prvku na konec seznamu

```
void pushEnd(int value, linked_list_t *list)
{
    assert(list);
    entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
    new_entry->value = value; // set data
    new_entry->next = NULL; // set the next
    if (list->tail == NULL) { //adding the 1st entry
        list->head = list->tail = new_entry;
    } else {
        list->tail->next = new_entry; //update the current tail
        list->tail = new_entry;
    }
    list->counter += 1;
}
```

- Na asymptotické složitost metody přidání dalšího prvku (na konec seznamu) se nic nemění, je nezávislé na aktuálním počtu prvků v seznamu

## Příklad použití

- Příklad použití na seznam hodnot typu `int`

```
#include "linked_list.h"

linked_list_t list = { NULL, NULL, 0 };
linked_list_t *lst = &list;
push(10, lst); push(5, lst); pushEnd(17, lst);
push(7, lst); pushEnd(21, lst);
print(lst);

printf("Pop 1st entry: %i\n", pop(lst));
printf("Lst: "); print(lst);

printf("Back of the list: %i\n", back(lst));
printf("Pop from the end: %i\n", popEnd(lst));
printf("Lst: "); print(lst);

free_list(lst); // cleanup!!!
```

- Výstup programu

```
clang linked_list.c demo-linked_list.c && ./a.out
7 5 10 17 21
Pop 1st entry: 7
Lst: 5 10 17 21                                lec09/linked_list.h
Back of the list: 21
Pop from the end: 21                            lec09/linked_list.c
Lst: 5 10 17                                    lec09/demo-linked_list.c
```

## Spojový seznam – Vložení prvku do seznamu

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

```
entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
new_entry->value = value; // nastavení hodnoty
new_entry->next = entry->next; //propojení s nasledujícím
entry->next = new_entry; //propojení entry
```

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

*Případně můžeme také požadovat vložení podle hodnoty prvku, tj. vložit před prvek s příslušnou hodnotou. Např. vložení prvku vždy před první prvek, který je větší vytvoříme uspořádaný seznam – realizujeme tak řazení vkládáním (insert sort).*

## Spojový seznam – getEntry()

- Nalezení prvku na pozici **index**
- Pokud je **index** větší než počet prvků v poli, návrat posledního prvku

```
static entry_t* getEntry(int index, const linked_list_t *list)
{ // here, we assume index >= 0
  entry_t *cur = list->head;
  int i = 0;
  while (i < index && cur != NULL && cur->next != NULL) {
    cur = cur->next;
    i += 1;
  }
  return cur; //return entry at the index or the last entry
}
Pokud je seznam prázdný vrátí NULL, tj. list->head == NULL.
```

- Funkci `getEntry()` chceme používat privátně pouze v rámci jednoho modulu (`linked_list.c`)
- Proto ji definujeme s modifikátorem `static`

Viz `lec09/linked_list.c`

## Spojový seznam – insertAt()

- Vložení nového prvku na pozici **index** v seznamu

```
void insertAt(int value, int index, linked_list_t *list)
{
  if (index < 0) { return; } // only positive position
  if (index == 0) { // handle the 1st position
    push(value, list);
    return;
  }
  entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
  assert(list && new_entry); // list and new_entry != NULL
  new_entry->value = value; // set data
  entry_t *entry = getEntry(index - 1, list);
  if (entry != NULL) { // entry can be NULL for the 1st
    new_entry->next = entry->next; // entry (empty list)
    entry->next = new_entry;
  }
  if (entry == list->tail) {
    list->tail = new_entry; // update the tail
  }
  list->counter += 1;
}
Pro napojení spojového seznamu potřebuje položku next, proto hledáme prvek na pozici (index - 1)—getEntry()
```

## Příklad vložení prvků do seznamu – insertAt()

- Příklad vložení do seznam čísel

```
linked_list_t list = { NULL, NULL, 0 };
linked_list_t *lst = &list;

push(10, lst); push(5, lst); push(17, lst);
push(7, lst); push(21, lst);
print(lst);

insertAt(55, 2, lst);
print(lst);

insertAt(0, 0, lst);
print(lst);

insertAt(100, 10, lst);
print(lst);

free_list(lst); // cleanup!!!
```

- Výstup programu

```
clang linked_list.c demo-insertat.c && ./a.out
21 7 17 5 10
21 7 55 17 5 10
0 7 55 17 5 10
0 7 55 17 5 10 100
```

`lec09/demo-insertat.c`

## Spojový seznam – getAt(int index)

- Nalezení prvků v seznamu podle pozice v seznamu
- V případě „adresace“ mimo rozsah seznamu vrátí `NULL`

```
entry_t* getAt(int index, const linked_list_t *const list)
{
    if (index < 0 || list == NULL || list->head == NULL) {
        return NULL; // check the arguments first
    }
    entry_t* cur = list->head;
    int i = 0;
    while (i < index && cur != NULL && cur->next != NULL) {
        cur = cur->next;
        i++;
    }
    return (cur != NULL && i == index) ? cur : NULL;
}
Složitost operace je v nejnepříznivějším případě O(n) (v případě pole je to O(1))
```

## Spojový seznam – removeAt(int index)

- Odebrání prvku na pozici `int index` a navázání seznamu
- Pokud `index > size - 1`, smaže poslední prvek (viz `getEntry()`)
- Pro navázání seznamu potřebujeme prvek na pozici `index - 1`

```
void removeAt(int index, linked_list_t *list)
{ // check the arguments first
    if (index < 0 || list == NULL || list->head == NULL) { return; }
    if (index == 0) {
        pop(list);
    } else {
        entry_t *entry_prev = getEntry(index - 1, list);
        entry_t *entry = entry_prev->next;
        if (entry != NULL) { //handle connection
            entry_prev->next = entry_prev->next->next;
        }
        if (entry == list->tail) {
            list->tail = entry_prev;
        }
        free(entry);
        list->count -= 1;
    }
}
Složitost v nejnepříznivější případě O(n)—nejdříve musíme najít prvek.
```

## Příklad použití getAt(int index)

- Příklad vypsání obsahu seznamu funkcí `getAt()` v cyklu

```
linked_list_t list = { NULL, NULL, 0 };
linked_list_t *lst = &list;

push(10, lst); push(5, lst); push(17, lst); push(7, lst); push(21, lst);
print(lst);
for (int i = 0; i < 7; ++i) {
    const entry_t* entry = getAt(i, lst);
    printf("Lst[%i]: ", i);
    (entry) ? printf("%2u\n", entry->value) : printf("NULL\n");
}

free_list(lst); // cleanup!!!

```

■ Výstup programu

```
clang linked_list.c demo-getat.c && ./a.out
21 7 17 5 10
Lst[0]: 21
Lst[1]: 7
Lst[2]: 17
Lst[3]: 5
Lst[4]: 10
Lst[5]: NULL
Lst[6]: NULL
```

*V tomto případě v každém běhu cyklu je složitost funkce `getAt()`  $O(n)$  a výpis obsahu seznamu má složitost  $O(n^2)$ !*

## Příklad použití removeAt(int index)

```
void removeAndPrint(int index, linked_list_t *lst)
{
    entry_t* e = getAt(index, lst);
    printf("Remove entry at %i (%i)\n", index, e ? e->value : -1);
    removeAt(index, lst);
    print(lst);
}

linked_list_t list = { NULL, NULL, 0 };
linked_list_t *lst = &list;
push(10, lst); push(5, lst); push(17, lst); push(7, lst); push(21, lst);
print(lst);
removeAndPrint(3, lst);
removeAndPrint(3, lst);
removeAndPrint(0, lst);
free_list(lst); // cleanup!!!

```

■ Výstup programu

```
clang linked_list.c demo-removeat.c && ./a.out
21 7 17 5 10
Remove entry at 3 (5)
21 7 17 10
Remove entry at 3 (10)
21 7 17
Remove entry at 0 (21)
7 17
```



## Vyhledání prvku v seznamu podle obsahu – `indexOf()`

- Vrátí číslo pozice prvního výskytu prvku v seznamu
- Pokud není prvek v seznamu nalezen vrátí funkce hodnotu `-1`

```
int indexOf(int value, const linked_list_t *const list)
{
    int counter = 0;
    const entry_t *cur = list->head;
    bool found = false;
    while (cur && !found) {
        found = cur->value == value;
        cur = cur->next;
        counter += 1;
    }
    return found ? counter - 1 : -1;
}
```

## Odebrání prvku ze seznamu podle jeho obsahu – `remove()`

- Podobně jako vyhledání prvku podle obsahu můžeme prvky odebrat
- Můžeme implementovat přímo nebo s využitím již existujících metod `indexOf()` a `removeAt()`
- Příklad implementace

```
void remove(int value, linked_list_t *list) {
    while ((idx = indexOf(value, list)) >= 0) {
        removeAt(idx, list);
    }
}
```

*Odebíráme všechny výskyty hodnoty `value` v seznamu.*

## Příklad použití `indexOf()`

```
linked_list_t list = { NULL, NULL, 0 };
linked_list_t *lst = &list;

push(10, lst); push(5, lst); push(17, lst);
push(7, lst); push(21, lst);
print(lst);
```

```
int values[] = { 5, 17, 3 };
for (int i = 0; i < 3; ++i) {
    printf("Index of (%2i) is %2i\n",
        values[i],
        indexOf(values[i], lst)
    );
}
```

```
free_list(lst); // cleanup !!!
```

- Výstup programu

```
clang linked_list.c demo-indexof.c && ./a.out
21 7 17 5 10
Index of ( 5) is 3
Index of (17) is 2
Index of ( 3) is -1
```

lec09/demo-indexof.c

## Příklad `indexOf()` pro spojový seznamu textových řetězců

- Porovnání hodnot textových řetězců—`strcmp()` – knihovna `<string.h>`
- Je nutné zvolit přístup pro alokaci hodnot textových řetězců  
V `lec09/linked_list-str.c` je zvolena **alokace paměti a kopírování hodnot**

- Příklad použití

```
#include "linked_list-str.h"
linked_list_t list = { NULL }; // initialization is important
linked_list_t *lst = &list;
push("FEE", lst); push("CTU", lst); push("PRP", lst);
push("Lecture09", lst); print(lst);

char *values[] = { "PRP", "Fee" };
for (int i = 0; i < 2; ++i) {
    printf("Index of (%s) is %2i\n", values[i], indexOf(values[i], lst));
}
free_list(lst); // cleanup !!!
```

- Výstup programu

```
clang linked_list-str.c demo-indexof-str.c && ./a.out
Lecture09 PRP CTU FEE
Index of (PRP) is 1
Index of (Fee) is -1
```

lec09/demo-indexof-str.c

## Spojový seznam s hodnotami typu textový řetězec

- Zajištění správné alokace a uvolnění paměti je náročnější
- V případě volání `pop()` je nutné následně dealokovat paměť

*V C++ lze řešit tzv. „smart pointers“*

```
/* WARNING printf("Popped value \"%s\"", pop(lst)); */
/* Note, using this will cause memory leakage since we lost the
   address value to free the memory!!! */
```

```
char *str = pop(lst);
printf("Popped value \"%s\"", str);
free(str); /* str must be deallocated */
```

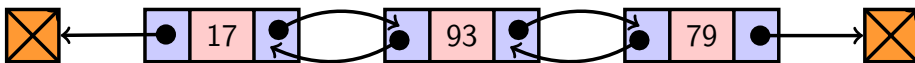
*Při práci s dynamickou pamětí a datovými strukturami je nutné zvolit vhodný model (např. kopírování dat) a zajistit správné uvolnění paměti.*

- Podobně jako textové řetězce se bude chovat ukazatel na nějakou komplexnější strukturu
- **Projděte si příložené příklady, zkuste si naimplementovat vlastní řešení a otestovat správnou alokaci a uvolnění paměti!**

`lec09/linked_list-str.h, lec09/linked_list-str.c, lec09/demo-indexof-str.c`

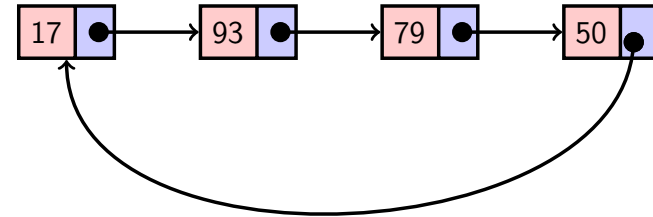
## Obousměrný spojový seznam

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



## Kruhový spojový seznam

- Položka `next` posledního prvku může odkazovat na první prvek
- Tak vznikne kruhový spojový seznam



- Při přidání prvku na začátek je nutné aktualizovat hodnotu položky `next` posledního prvku

## Příklad – Obousměrný spojový seznam

- Prvek listu má hodnotu (`value`) a dva odkazy (`prev` a `next`)
- Alokaci prvku provedeme funkcí s inicializací na základní hodnoty

```
typedef struct dll_entry {
    int value;
    struct dll_entry *prev;
    struct dll_entry *next;
} dll_entry_t;
```

```
typedef struct {
    dll_entry_t *head;
    dll_entry_t *tail;
} doubly_linked_list_t;
```

```
dll_entry_t*
allocate_dll_entry(int
value)
{
    dll_entry_t *new_entry = (
    dll_entry_t*)malloc(
    sizeof(dll_entry_t));
    assert(new_entry);

    new_entry->value = value;
    new_entry->next = NULL;
    new_entry->prev = NULL;

    return new_entry;
}
```

`lec09/doubly_linked_list.h, lec09/doubly_linked_list.c`

## Obousměrný spojový seznam – vložení prvku

- Vložení prvku před prvek `cur`:
  1. Napojení vloženého prvku do seznamu, hodnoty `prev` a `next`
  2. Aktualizace `next` předchozí prvku k prvku `cur`
  3. Aktualizace `prev` proměnné prvku `cur`

```
void insert_dll(int value, dll_entry_t *cur)
{
    assert(cur);
    dll_entry_t *new_entry = allocate_dll_entry(value);
    new_entry->next = cur;
    new_entry->prev = cur->prev;
    if (cur->prev != NULL) {
        cur->prev->next = new_entry;
    }
    cur->prev = new_entry;
}
lec09/doubly_linked_list.c
```

## Obousměrný spojový seznam – tisk seznamu

### print\_dll() a printReverse()

```
void print_dll(const doubly_linked_list_t *list)
{
    if (list && list->head) {
        dll_entry_t *cur = list->head;
        while (cur) {
            printf("%i%s", cur->value, cur->next ? " " : "\n");
            cur = cur->next;
        }
    }
}

void printReverse(const doubly_linked_list_t *list)
{
    if (list && list->tail) {
        dll_entry_t *cur = list->tail;
        while (cur) {
            printf("%i%s", cur->value, cur->prev? " " : "\n");
            cur = cur->prev;
        }
    }
}
lec09/doubly_linked_list.c
```

## Obousměrný spojový seznam– přidání prvku na začátek seznamu push()

```
void push_dll(int value, doubly_linked_list_t *list)
{
    assert(list);
    dll_entry_t *new_entry = allocate_dll_entry(value);
    if (list->head) { // an entry already in the list
        new_entry->next = list->head; // connect new -> head
        list->head->prev = new_entry; // connect new <- head
    } else { //list is empty
        list->tail = new_entry;
    }
    list->head = new_entry; //update the head
}
lec09/doubly_linked_list.c
```

## Příklad použití

```
#include "doubly_linked_list.h"

doubly_linked_list_t list = { NULL, NULL };
doubly_linked_list_t *lst = &list;

push_dll(17, lst); push_dll(93, lst);
push_dll(79, lst); push_dll(11, lst);

printf("Regular print: ");
print_dll(lst);

printf("Revert print: ");
printReverse(lst);

free_dll(lst);
```

- Výstup programu

```
clang doubly_linked_list.c demo-double_linked_list.c
```

```
./a.out
```

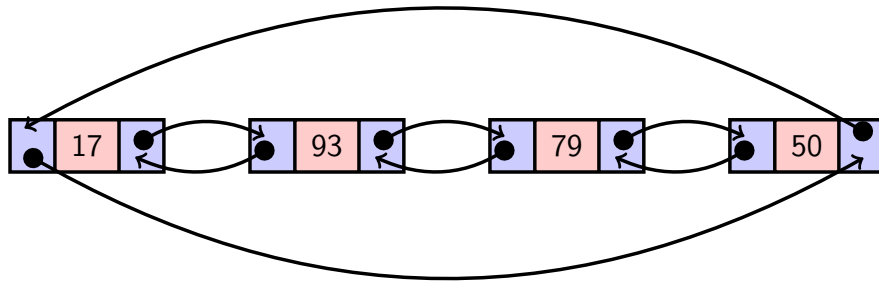
```
Regular print: 11 79 93 17
Revert print: 17 93 79 11
```

```
lec09/doubly_linked_list.c
```

```
lec09/demo-doubly_linked_list.c
```

## Kruhový obousměrný seznam

- Položka **next** posledního prvku odkazuje na první prvek
- Položka **prev** prvního prvku odkazuje na poslední prvek



## Část II

### Část 2 – Zadání 8. domácího úkolu (HW08)

## Zadání 8. domácího úkolu HW08

### Téma: Kruhová fronta v poli

Povinné zadání: **3b**; Volitelné zadání: **2b**; Bonusové zadání: **není**

- **Motivace:** Práce s pamětí a datovými strukturami
- **Cíl:** Prohloubit si znalost paměťové reprezentace a dynamické alokace paměti s uvolňováním
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw08>
  - Implementace kruhové fronty s využitím předalokovaného pole pro vkládané prvky.
  - **Volitelné zadání** rozšiřuje úlohu o dynamické zvětšování a zmenšování kapacity fronty podle aktuálních požadavků na počet vkládaných/odebíraných prvků.
- **Termín odevzdání:** **08.12.2018, 23:59:59 PST**

## Shrnutí přednášky

## Diskutovaná témata

- Spojivé struktury
  - Jednosměrný spojivý seznam
  - Obousměrný spojivý seznam
  - Kruhový obousměrný spojivý seznam
- Implementace operací `push()`, `pop()`, `size()`, `back()`, `pushEnd()`, `popEnd()`, `insertAt()`, `getEntry()`, `getAt()`, `removeAt()`, `indexOf()`
- Použití spojivého seznamu pro dynamicky alokované hodnoty prvků seznamu
  
- **Příště: Stromy.**