

# Spojové struktury

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 08

**B0B36PRP – Procedurální programování**

# 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)

# Část I

## Část 1 – Spojové struktury

## 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 především mazání prvku vyžaduje kopírování
    - Kopírování objemných prvků lze řešit ukazatelem, neřeší však problem přesunu*

## 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 prvků může být pomalé (nová alokace a kopírování)
  - **Spojové seznamy**

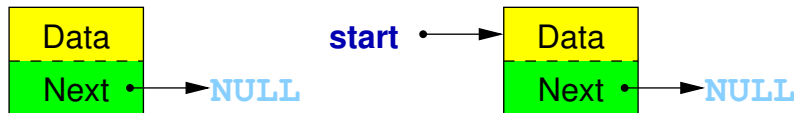
## Spojové seznamy

- Datová struktura realizující seznam dynamické délky
- Každý prvek seznamu obsahuje
  - Datovou část (hodnota proměnné / objekt)
  - Odkaz (ukazatel) na další prvek v seznamu

*NULL v případě posledního prvku seznamu.*

- První prvek seznamu se zpravidla označuje jako *head* nebo *start*

*Realizujeme jej jako ukazatel odkazující na první prvek seznamu*



## Základní operace se spojovým seznamem

### ■ Vložení prvku

- Předchozí prvek odkazuje na nový prvek
- *Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje* *Tzv. obousměrný spojový seznam*

### ■ Odebrání prvku

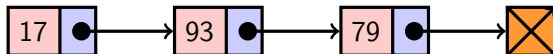
- Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
- Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek

### ■ Základní implementací spojového seznamu je tzv.

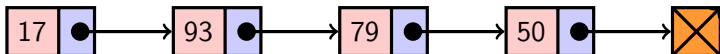
**jednosměrný spojový seznam**

## Jednosměrný spojový seznam

- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 50 na konec seznamu



- Odebrání prvku 79



1. Nejdříve sekvenčně najdeme prvek s hodnotou 79
2. Následně vyjmeme a napojíme prvek 93 na prvek 50

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



## Spojový seznam

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

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

- Příklad tříd pro uložení spojového seznamu celých čísel

```
typedef struct entry {
    int value;
    struct entry *next;
} entry_t;
```

```
entry_t *head = NULL;
```

Vlastní struktura, například

```
typedef struct {
    entry_t *head;
    entry_t *end;
    int count; // pocet
                prvku
} linked_list_t;
```

- Pro jednoduchost prvky seznamu obsahují celé číslo.

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

## 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 – 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));

  assert(new_entry); // malloc may eventually fail

  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
}
```

- Přidání prvku není závislé na počtu prvků v seznamu

*Konstantní složitost operace `push()` –  $O(1)$*

## 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;
  free(prev_head); // relase memory of the popped entry
  return ret;
}
```

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

*Použijeme 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 – back()

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

```
int back(const entry_t *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()` –  $O(n)$*

## 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

## 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

```

```

lec08/simple_linked_list.h
lec08/simple_linked_list.c
lec08/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 count**
  - Počet prvků inkrementujeme při každém přidání prvku a dekrementuje při každém odebrání prvku
- Operaci `back()` můžeme urychlit referenční proměnou odkazující na poslední prvek
- Zavedeme strukturu pro vlastní spojový seznam s položkami `head`, `count`, and `end`

```
typedef struct {  
    entry_t *head;  
    entry_t *end;  
    int count;  
} 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 – urychlený `size()`

- Samostatná struktura pro seznam

- Položky `head` a `count`

- `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`

```
linked_list_t *list = &linked_list;
```

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

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

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

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

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

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

```
void push(int value, linked_list_t *list)
{ // add new entry at front
    entry_t *new_entry = (entry_t*)malloc(sizeof(entry_t));
    assert(list && new_entry);
    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->end = new_entry; //1st entry is the end
    }
    list->head = new_entry; //update the head
    list->count += 1; // keep count up to date
}
```

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

## 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->count -= 1; // keep count up to date
    int ret = prev_head->value;
    free(prev_head); // release the memory
    if (list->head == NULL) { //end has been popped
        list->end = NULL;
    }
    return ret;
}
```

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

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

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

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

- Udržováním hodnoty proměnné `end` jsme snížili časovou náročnost operace `back()` z lineární složitosti na počtu prvků v seznamu  $O(n)$  na konstantní složitost  $O(1)$

## 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));
    assert(list && new_entry);
    new_entry->value = value; // set data
    new_entry->next = NULL; // set the next
    if (list->end == NULL) { //adding the 1st entry
        list->head = list->end = new_entry;
    } else {
        list->end->next = new_entry; //update the current end
        list->end = new_entry;
    }
    list->count += 1;
}
```

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

## Spojový seznamu – popEnd()

### ■ Odebrání prvku z konce seznamu

```
int popEnd(linked_list_t *list)
{
    assert(list && list->head);
    entry_t *end= list->end; // save the end
    if (list->head == list->end) { // the last entry is
        list->head = list->end = 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->end = cur;
        list->end->next = NULL; //the end does not have next
    }
    int ret = end->value;
    free(end);
    list->count -= 1;
    return ret;
}
```

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

## 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                                lec08/linked_list.h
Back of the list: 21
Pop from the end: 21                            lec08/linked_list.c
Lst: 5 10 17                                    lec08/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 **end** (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; // nastaveni hodnoty  
new_entry->next = entry->next; //propojeni s nasledujicim  
entry->next = new_entry; //propojeni entry
```

- Do seznamu můžeme chtít prvek vložit na příslušné pořadí, tj. podle indexu v seznamu

*Případně můžeme také požadovat vložení podle hodnoty prvku, tj. vložit před prvek s příslušnou hodnotou.*

## 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);
    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->end) {
        list->end = new_entry; // update end
    }
    list->count += 1;
}
```

*Pro napojení spojového seznamu potřebuje položku **next**, proto hledáme prvek na pozici (*index* - 1)—getEntry()*

## Spojový seznam – `getEntry()`

- Nalezení prvku na pozici `index`
- Pokud je `index` vyšší 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`.*

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

Viz `lec08/linked_list.c`

## 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
```

lec08/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 *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  $O(1)$ )*

## 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  $O(n^2)$ !*

lec08/demo-getat.c

## Spojový seznamu – `removeAt(int index)`

- Odebrání prvku na pozici `int index` a navážeme seznam
- 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->end) {
      list->end = entry_prev;
    }
    free(entry);
    list->count -= 1;
  }
}
```

*Složitost v nejnepríznivější případě  $O(n)$  (nejdříve musíme najít prvek).*

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

```

lec08/demo-removeat.c



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

## 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
```

[lec08/demo-indexof.c](#)

## Odebrání prvku ze seznamu podle jeho obsah

- 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) {  
    int idx = indexOf(value, list);  
    while(idx != -1) {  
        removeAt(idx, list);  
        idx = indexOf(value);  
    }  
}
```

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

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

- Porovnání hodnot textových řetězců—`strcmp()`
- Je nutné zvolit přístup pro alokaci hodnot textových řetězců  
 V `lec08/linked_list-str.c` je zvolena alokace paměti a kopírování hodnoty
- 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("Lecture07", 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
Lecture07 PRP CTU FEE
Index of (PRP) is 1
Index of (Fee) is -1                                lec08/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\"\n", 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\"\n", str);  
free(str); /* str must be deallocated */
```

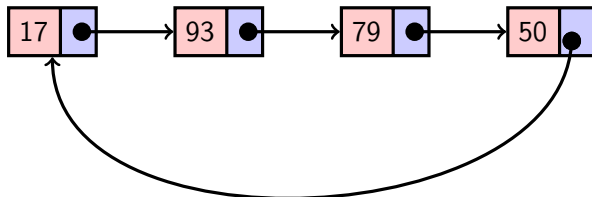
*Při práci s dynamickou pamětí a datovými strukturami je nutné si zvolit vhodný model (např. kopírování dat) a zajistit jejich (např. kopírování dat) a zajistit jejich (např. kopírování dat) a zajistit jejich (např. kopírování dat) a zajistit správné uvolnění paměti.*

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

lec08/linked\_list-str.h, lec08/linked\_list-str.c, lec08/demo-indexof-str.c

## Kruhový spojový seznam

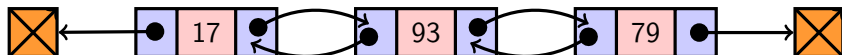
- Položka **next** posledního prvku může odkazovat na první prvek
- Tak vznikne kruhový spojový seznam



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

## 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



## 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 dcll_entry {
    int value;
    struct dcll_entry *prev;
    struct dcll_entry *next;
} dcll_entry_t;

typedef struct {
    dcll_entry_t *head;
    dcll_entry_t *end;
} dc_linked_list_t;
```

```
dcll_entry_t*
allocate_dcl_entry(int
value)
{
    dcll_entry_t *new_entry =
        (dcll_entry_t*)malloc(
            sizeof(dcll_entry_t));
    assert(new_entry);

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

    return new_entry;
}
```

lec08/double\_connected\_linked\_list.h

lec08/double\_connected\_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_dcll(int value, dcll_entry_t *cur)
{
    assert(cur);
    dcll_entry_t *new_entry = allocate_dcl_entry(value);
    new_entry->next = cur;
    new_entry->prev = cur->prev;
    if (cur->prev != NULL) {
        cur->prev->next = new_entry;
    }
    cur->prev = new_entry;
}
```

lec08/double\_connected\_linked\_list.c

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

```
void push_dcll(int value, dc_linked_list_t *list)
{
    assert(list);
    dcll_entry_t *new_entry = allocate_dcl_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->end = new_entry;
    }
    list->head = new_entry; //update the head
}
```

`lec08/double_connected_linked_list.c`

## Obousměrný spojový seznam – tisk seznamu print\_dcll() a printReverse()

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

void printReverse(const dc_linked_list_t *list)
{
    if (list && list->end) {
        dcll_entry_t *cur = list->end;
        while (cur) {
            printf("%i%s", cur->value, cur->prev? " " : "\n");
            cur = cur->prev;
        }
    }
}
```

lec08/double\_connected\_linked\_list.c

## Příklad použití

```
#include "double_connected_linked_list.h"

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

push_dc11(17, lst); push_dc11(93, lst);
push_dc11(79, lst); push_dc11(11, lst);

printf("Regular print: ");
print_dc11(lst);

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

free_dc11(lst);
```

### ■ Výstup programu

```
clang double_connected_linked_list.c demo-double_connected_linked_list.c
```

```
./a.out
```

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

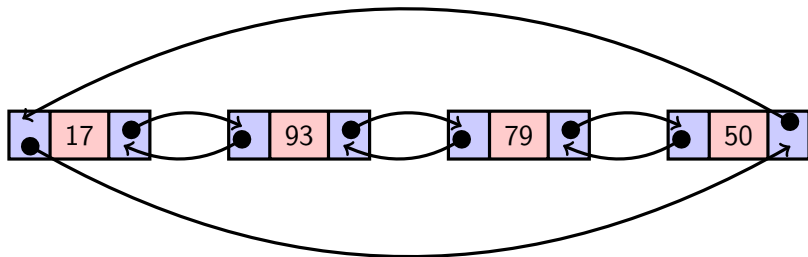
```
Revert print: 17 93 79 11
```

```
lec08/double_connected_linked_list.c
```

```
lec08/demo-double_connected_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

- 
- Termín odevzdání: 17.12.2016, 23:59:59 AoE

*AoE – Anywhere on Earth*

# Shrnutí přednášky



## Diskutovaná témata

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