

Spojové struktury

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 08

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), kompilátor tak může vytvořit kód, ve kterém se adresa prvku spočítá z indexu a velikosti prvku.
 - Velikost pole je určena při vytvoření pole.
 - Velikost (maximální velikost) musí být známa v době vytváření.
 - Změna velikost není přímo možná.
Nutné nové vytvoření (alokace paměti), voláním `realloc()` může dojít k rozšíření, které závisí na aktuálním stavu paměti.
 - Využití pouze malé části pole (s objemnými prvky) může být plýtváním paměti.
 - V případě řazení pole přesouváme jednotlivé položky pole.
 - Vložení prvku a vyjmutí prvku vyžaduje kopírování (zachování souvislosti dat).
Kopírování objemných prvků lze případně řešit ukládáním ukazatelů.

Seznam – list

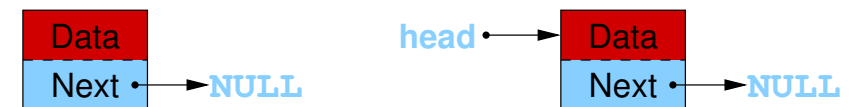
- Seznam (proměnných nebo objektů) patří mezi základní datové struktury.
Základní ADT – Abstract Data Type.
- Seznam zpravidla nabízí sadu základních operací:
 - Vložení prvku (**insert**);
 - Odebrání prvku (**remove**);
 - Vyhledání prvku (**indexOf**);
 - Aktuální počet prvku v seznamu (**size**).
- Implementace seznamu může být založena na poli nebo spojové struktuře.
 - Pole
 - Indexování je velmi rychlé.
 - Vložení prvku na konkrétní pozici může být pomalé.
Nová alokace a kopírování.
 - Spojové seznamy**
 - Položky seznamu jsou sekvencně propojeny, přímý náhodný přístup není jednoduše možný.
 - Vložení nebo odebrání prvku může být velmi rychlé.

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 nově odkazuje na následující prvek, na který odkazoval odebíraný prvek.
- Základní implementací spojového seznamu je
jednosměrný spojový seznam.

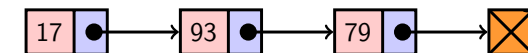
Spojové seznamy

- 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.
- První prvek seznamu se zpravidla označuje jako **head** nebo **start**.
Realizujeme jej jako ukazatel odkazující na první prvek seznamu.

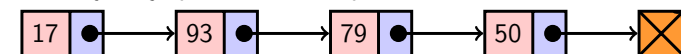


Jednosměrný spojový seznam

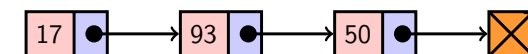
- Příklad spojového seznamu pro uložení číselných hodnot.



- Přidání nové hodnoty 50 je přidání nového prvku na konec seznamu.



- Odebrání prvku s hodnotou 79.



- Nejdříve sekvencně najdeme prvek s hodnotou 79.
- Následně vyjmeme prvek s hodnotou a propojíme prvek 93 s prvkem 50.

Položku next prvku 93 nastavíme na hodnotu next odebíraného prvku, tj. na následující prvek s hodnotou 50.

Spojový seznam

- Seznam tvoří struktura prvku s dvěma základními položkami:

- Data prvku (může být ukazatel);
- Odkaz (ukazatel) na další prvek.

- Seznam je pak:

- Ukazatel na první prvek `head`;
- nebo vlastní struktura pro seznam.

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

- Příklad struktur pro uložení spojoiného seznamu celých čísel.

```
1 typedef struct entry {
2     int value;
3     struct entry *next;
4 } entry_t;
5 entry_t *head = NULL;
```

```
1 typedef struct {
2     entry_t *head;
3     entry_t *tail;
4     int counter; // pocet prvku
5 } linked_list_t;
```

- 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.

```
1 void push(int value, entry_t **head)
2 { // add new entry at front of the list
3     entry_t *new_entry = myMalloc(sizeof(entry_t));
4     new_entry->value = value; // set data
5     if (*head == NULL) { // first entry in the list
6         new_entry->next = NULL; // reset the next
7     } else {
8         new_entry->next = *head;
9     }
10    *head = new_entry; //update the head
11 }
12 // Alternativně můžeme push() implementovat také 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`. **Kontrola dynamické alokace**

```
head = myMalloc(sizeof(entry_t));
head->value = 10;
head->next = NULL;
```

- Další prvek (13) přidáme propojením s aktuálně 1. prvkem.

```
entry_t *new_entry = myMalloc(sizeof(entry_t));
new_entry->value = 13;
new_entry->next = head;
```

- a aktualizací proměnné `head`.

```
head = new_entry;
```

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

lec08/my_malloc.h, lec08/my_malloc.c

- 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.

Spojový seznam – pop()

- Odebrání prvního prvku ze seznamu

Kdy použijeme assert() a kdy myAssert()?

```
1 int pop(entry_t **head)
2 { // linked list must be non-empty
3     assert(head != NULL && *head != NULL);
4     entry_t *prev_head = *head; // save the current head
5     int ret = prev_head->value; // retrieve data from the current head
6     *head = prev_head->next; // set to NULL if the last item is popped
7     free(prev_head); // release memory of the popped entry
8     return ret;
9 }
```

*Alternativně například také jako int pop(entry_t *head), ale nenastaví 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`.
Poslední položka je taková, pro kterou platí `next == NULL`, nebo je seznam prázdný a `head == NULL`.

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

```
1 int size(const entry_t *const head)
2 { // const - we do not attempt to modify the list
3     int counter = 0;
4     const entry_t *cur = head;
5     while (cur) { // or cur != NULL
6         cur = cur->next;
7         counter += 1;
8     }
9     return counter;
10 }
```

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

- Procházíme kompletní seznam (n prvků), abychom spočítali počet prvků seznamu.
Lineární složitost operace `size()` – $O(n)$.

Spojový seznam – procházení seznamu

- Procházení seznamu demonstrujeme na funkci `print()`.

```
1 void print(const entry_t *const head)
2 {
3     const entry_t *cur = head; // set the cursor to head
4     while (cur != NULL) {
5         printf("%i%s", cur->value, cur->next ? " " : "\n");
6         cur = cur->next; // move in the linked list
7     }
8 }
```

- 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()`.

```
1 int back(const entry_t *const head)
2 {
3     const entry_t *end = head;
4     while (end && end->next) { // 1st test list is not empty
5         end = end->next;
6     }
7     assert(end); //do not allow calling back on empty list
8     return end->value;
9 }
```

Kontrolou `assert()` vynucujeme, že při implementaci programu ladíme, že volání `back()` nebudeme provádět pro prázdný seznam. To musíme zajistit programově.

- Musíme projít všechny prvky seznamu.
Lineární složitost operace `back()` – $O(n)$.

Příklad – Spojový seznam celých čísel

```
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", 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-linked_list-
    int.c 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/linked_list-int.h`
`lec08/linked_list-int.c`

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

- Operace `size()` a `back()` procházejí kompletní seznam.
- Operaci `size()` můžeme urychlit udržováním aktuálního počtu prvku 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`.

```
1 typedef struct {
2     entry_t *head;
3     entry_t *tail;
4     int counter;
5 } linked_list_t;
```

■ V případě přidání prvku na začátek, aktualizujeme `tail` pouze pokud byl seznam doposud prázdný.

■ Proměnnou `tail` aktualizujeme při přidání prvku na konec nebo vyjmutí posledního prvku.

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

```
1 void push(int value, linked_list_t *list)
2 { // add new entry at front
3     assert(list);
4     entry_t *new_entry = myMalloc(sizeof(entry_t));
5     new_entry->value = value; // set data; exit is called if myMalloc fails
6     if (list->head) { // an entry already in the list
7         new_entry->next = list->head;
8     } else { //list is empty
9         new_entry->next = NULL; // reset the next
10        list->tail = new_entry; //1st entry is the tail
11    }
12    list->head = new_entry; //update the head
13    list->counter += 1; // keep counter up to date
14 }
```

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 použijeme proměnnou `list`
`linked_list_t *list = &linked_list;`
- Inkrementujeme a dekrementujeme 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

- Při volání musí být odkaz na spojový seznam platný (nikoliv `NULL`).
- `assert()` testujeme správnost volání, že jsme ve struktuře programu neudělali chybu. Po odladění můžeme test vypustit, např. `NDEBUG`.
- `myAssert()` testuje, že data jsou za běhu programu správně. Pokud ne, ukončujeme program a reportujeme.

V našem konkrétním případě můžeme také zajistit programově použití `assert()` a podmínit volání `pop()`, např. `if (!is_empty(list))`.

```
1 int pop(linked_list_t *list)
2 {
3     assert(list);
4     myAssert(list->head, __LINE__, __FILE__); // non-empty list
5     entry_t *prev_head = list->head; // save head
6     list->head = prev_head->next;
7     list->counter -= 1; // keep counter up to date
8     int ret = prev_head->value;
9     free(prev_head); // relase the memory
10    if (list->head == NULL) { // end has been popped
11        list->tail = NULL;
12    }
13    return ret;
14 }
```

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

`myAssert()`

```
1 #ifndef __MY_ASSERT_H__
2
3 #include <stdio.h> //fprintf()
4 #include <stdlib.h> //exit() and
5 ..
6
7 #define myAssert(x, line, file)
8 \
9 if (!(x)) {\
10     fprintf(stderr, "my_assert
11     fail, line: %d, file %s\n",
12     line, file);\
13     exit(-1);\
14 }
15 #endif
```

Výpis chyby s číslem řádku a jménem zdrojového souboru pro rychlejší nalezení kontextu a případnou opravu.

Spojový seznam – back() s odkazem na konec seznamu

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

```
1 int back(const linked_list_t *const list)
2 { // const - we do not modify the linked list
3   // we do not allow to call back on empty list that has to be assured
   programmatically
4   assert(list && list->tail);
5   return list->tail->value;
6 }
```

- 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.

```
1 int popEnd(linked_list_t *list)
2 {
3   assert(list && list->head); // structure test not runtime
4   entry_t *end = list->tail; // save the end
5   if (list->head == list->tail) { // the last entry is
6     list->head = list->tail = NULL; // removed
7   } else { // there is also penultimate entry
8     entry_t *cur = list->head; // that needs to be
9     while (cur->next != end) { // updated (its next
10       cur = cur->next; // pointer to the next entry
11     }
12     list->tail = cur;
13     list->tail->next = NULL; //the tail does not have next
14   }
15   int ret = tail->value;
16   free(end);
17   list->counter -= 1;
18   return ret;
19 }
```

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

Spojový seznamu – pushEnd()

- Přidání prvku na konec seznamu.

```
1 void pushEnd(int value, linked_list_t *list)
2 {
3   assert(list);
4   entry_t *new_entry = myMalloc(sizeof(entry_t));
5   new_entry->value = value; // set data
6   new_entry->next = NULL; // set the next
7   if (list->tail == NULL) { //adding the 1st entry
8     list->head = list->tail = new_entry;
9   } else {
10     list->tail->next = new_entry; //update the current tail
11     list->tail = new_entry;
12   }
13   list->counter += 1;
14 }
```

- 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**.

```
1 #include "linked_list.h"
2
3 linked_list_t list = { NULL, NULL, 0 };
4 linked_list_t *lst = &list;
5 push(10, lst); push(5, lst); pushEnd(17, lst);
6 push(7, lst); pushEnd(21, lst);
7 print(lst);
8
9 printf("Pop 1st entry: %i\n", pop(lst));
10 printf("Lst: "); print(lst);
11
12 printf("Back of the list: %i\n", back(lst));
13 printf("Pop from the end: %i\n", popEnd(lst));
14 printf("Lst: "); print(lst);
15
16 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
Back of the list: 21
Pop from the end: 21
Lst: 5 10 17
```

lec08/linked_list.h
lec08/linked_list.c
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 **tail** (funkce **pushEnd()**);
 - obecně – potřebujeme prvek (**entry**), za který chceme nový prvek (**new_entry**) vložit.

```
entry_t *new_entry = myMalloc(sizeof(entry_t));
new_entry->value = value; // nastavení hodnoty
new_entry->next = entry->next; //propojení s nasledujicim
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 hodnotu. 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 lec08/linked_list.c

Spojový seznam – insertAt()

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

```
void insertAt(int value, int index, linked_list_t *list)
{
  assert(list); // list != NULL
  if (index < 0) { return; } // only positive position
  if (index == 0) { // handle the 1st position
    push(value, list);
    return;
  }
  entry_t *new_entry = myMalloc(sizeof(entry_t));
  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
```

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 *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 nejnejpříznivějším 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 nejnejpříznivějším případě $O(n)$, protože nejdříve musíme prvek najít.

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!!!
```

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

- 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
```

lec08/demo-getat.c

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!!!
```

lec08/demo-removeat.c

```
1 $ clang linked_list.c demo-
  removeat.c && ./a.out
2 21 7 17 5 10
3 Remove entry at 3 (5)
4 21 7 17 10
5 Remove entry at 3 (10)
6 21 7 17
7 Remove entry at 0 (21)
8 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 podle obsahu odebrat.
- Můžeme implementovat přímo nebo s využitím již existujících funkcí indexOf() a removeAt().
- Příklad implementace. *Odebíráme všechny výskyty hodnoty value v seznamu.*

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

To co programátor/ka zpravidla řešení (má řešit), ve smyslu intelektuální náročnosti, není ani tak vlastní implementace, jako spíše návrh, jak se má funkce chovat, např. smazání prvního výskytu prvku vs. všech prvků s danou hodnotou. Implementace je do velké části dovednost („řemeslo“), kreativní činnost je spíše návrh struktury programu, definování chování funkcí a definování jmen proměnných a funkcí (identifikátorů).

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

## Příklad indexOf() pro spojový seznamu textových řetězců

- Porovnání hodnot textových řetězců—strcmp() – knihovna <string.h>.
- Je nutné zvolit přístup pro alokaci hodnot textových řetězců. *Literály vs. proměnné.*
- Příklad použití. *V lec08/linked\_list-str.c je zvolena alokace paměti a kopírování hodnot.*

```
#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
```

lec08/demo-indexof-str.c

Spojový seznam s hodnotami typu textový řetězec

- Zajištění správné alokace a uvolnění paměti je v našem příkladě náročnější.
- V případě volání `pop()` je nutné následně paměť uvolnit.

V C++ lze řešit například prostřednictvím „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 */
```

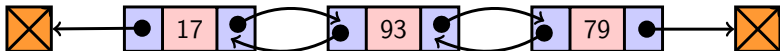
Při práci s dynamickou pamětí a datovými strukturami je nutné zvolit vhodný model (např. kopírování dat) a zajistit správné uvolnění paměti.

- 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!**

`lec08/linked_list-str.h, lec08/linked_list-str.c, lec08/demo-indexof-str.c`

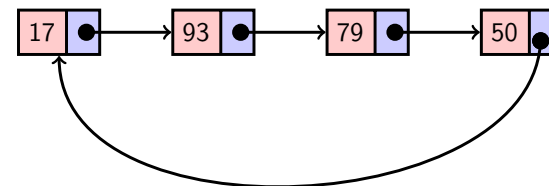
Obousměrný spojový seznam

- 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 `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 = myMalloc(sizeof(
        dll_entry_t));
    new_entry->value = value;
    new_entry->next = NULL;
    new_entry->prev = NULL;
    return new_entry;
}
```

`lec08/doubly_linked_list.h, lec08/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;
}
```

lec08/doubly_linked_list.c

Obousměrný spojový seznam – tisk seznamu

```
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;
        }
    }
}
```

lec08/doubly_linked_list.c

Obousměrný spojový seznam– přidání prvku na začátek `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
}
```

lec08/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);
```

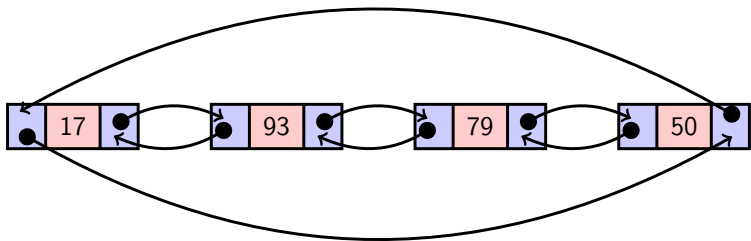
- Výstup programu

```
$ clang doubly_linked_list.c
demo-double_linked_list.c
$ ./a.out
Regular print: 11 79 93 17
Revert print: 17 93 79 11
```

lec08/doubly_linked_list.c
lec08/demo-doubly_linked_list.c

Kruhový obousměrný seznam

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



Zadání 8. domácího úkolu HW08

Téma: Kruhov^á fronta v poli

Povinné zadání: **3b**; Volitelné zadání: **není**; 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:** 09.01.2026, 23:59:59 PST.

PST – Pacific Standard Time

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.
- Přístě abstraktní datový typ (ADT).

Část IV
Appendix

Kódovací příklad – Dynamická knihovna

- Knihovna s implementací (spojového) seznamu pro ukládání dynamicky alokovaných položek.
- Pro jednoduchost při chybě dynamické alokace program ukončíme s výstupem na `stderr`.
- Implementujeme funkci `push()`, která nepřidá hodnoty `NULL`.
Návrhová volba!
- Prázdný seznam je indikován návratovou hodnotou `NULL` funkce `pop()`.
- Implementujeme nastavení funkce porovnání položek `setLess()`, kterou využijeme při vkládání položek do seznamu. Uspořádání položek dojde při volání `push()`.
Implementujeme `insert sort`.
- Vytvoříme dvě verze knihovny s/bez uspořádání položek, které budeme linkovat dynamicky.

```
1 #ifndef __LIST_H__
2 #define __LIST_H__
3
4 void* create(void); // void* - konkrétní typ považujeme za vnitřní záležitost knihovny.
5 void release(void** list); // argument list musí odpovídat typu z volání create()!
6 void setLess(void *list, bool (*isLess)(const void *a, const void *b));
7 void push(void *list, void *value);
8 void* pop(void *list);
9
10 #endif
11
12 list.h
```

Kódovací příklad – list.c 1/2

- Při chybě alokace program končí voláním `exit()` s návratovou hodnotou `101`.
- Definici složených typů implementujeme pouze v souboru `list.c`, může se případně v budoucnu měnit, proto není `struct item` součástí rozhraní v `list.h`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include "list.h"
5
6 struct item {
7     void *value;
8     struct item *next;
9 };
10
11 struct list {
12     struct item *root;
13     bool (*isLess)(const void *a, const void *b);
14 };
15
16 enum { ERROR_MEM = 101 };
17
18 list.c
```

```
19 void* create(void)
20 {
21     struct list *ret = malloc(sizeof(struct list));
22     if (!ret) {
23         fprintf(stderr, "ERROR: cannot allocate memory for list!\n");
24         exit(ERROR_MEM);
25     }
26     ret->root = NULL;
27     ret->isLess = NULL;
28     return ret;
29 }
30
31 void release(void** list)
32 {
33     if (list && *list) {
34         struct list *lst = (struct list*)*list;
35         struct item *cur = lst->root;
36         while (cur) {
37             struct item *t = cur;
38             cur = cur->next;
39             free(t->value); //Případě specifická funkce
40             free(t); // pro složený typ s ukazateli
41         }
42         free(*list);
43         *list = NULL;
44     }
45 }
```

Kódovací příklad – list.c 2/3

```
47 void setLess(void *list, bool (*isLess)(const void *a, const void *b))
48 {
49     if (!list) {
50         struct list *lst = (struct list*)list;
51         lst->isLess = isLess;
52     }
53 }
54
55 static void* allocate_item(void* value)
56 {
57     struct item *ret = malloc(sizeof(struct item));
58     if (!ret) {
59         fprintf(stderr, "ERROR: cannot allocate memory for list item!\n");
60         exit(ERROR_MEM);
61     }
62     ret->value = value;
63     ret->next = NULL;
64     return ret;
65 }
```

list.c

```
104 void* pop(void *list)
105 {
106     void *ret = NULL;
107     struct list *lst = (struct list*)list;
108     if (!lst || !lst->root) {
109         ret = lst->root->value;
110         struct item *p = lst->root;
111         lst->root = p->next;
112         free(p);
113     }
114     return ret;
115 }
```

list.c

- Funkce `setLess()` nastavuje ukazatel na funkci.
- Funkce `allocate_item()` nastavuje `next` na `NULL`.
- Položka `value` je adresa dynamicky alokované paměti.

- Funkce `pop()` vrací `NULL` v případě prázdného seznamu.
- Při práci se seznamem explicitně přetypujeme vstupní argument `list` na typ ukazatel na `struct list*`.
- Při vyjmutí delegujeme správu paměti alokované na adrese `value` volající funkci (adresa je návratová hodnota funkce).

Kódovací příklad – Volání rozhraní seznamu 1/3

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <string.h>
5
6 #include "list.h"
7
8 bool isLess(const void *a, const void *b);
9
10 int main(void)
11 {
12     int ret = EXIT_SUCCESS;
13     char *line = NULL; // musí být NULL, alokace v getline()
14     size_t linecap = 0; // délka alokované paměti v getline()
15     ssize_t ln = 0; // délka načteného řetězce
16     void *list = create(); // vytvoření seznamu z list.h
17     setLess(list, isLess); // nastavení funkce isLess()
18 }
```

demo.c

```
20 while ( (ln = getline(&line, &linecap, stdin)) > 0) {
21     if (line[ln-1] == '\n') { // ln vždy alespoň 1!
22         line[ln-1] = '\0'; // ignorujeme konec řádku
23     }
24     fprintf(stderr, "DEBUG: read \"%s\"\n", line);
25     push(list, line); // přidáme do seznamu
26     linecap = 0;
27     line = NULL; // vynucujeme novou alokaci
28 }
29 while ( (line = pop(list)) ) {
30     printf("Popped value is \"%s\"\n", line);
31     free(line); // uvolňujeme řádek z paměti
32 }
33 release(&list);
34 return ret;
35 }
36
37 bool isLess(const void *a, const void *b)
38 {
39     return strcmp(a, b) < 0; // lexikografické porovnání
40 }
```

demo.c

- Hodnoty textových řetězců jsou dynamicky alokovány.
- Načítání hodnot realizujeme funkcí `getline()`, která alokuje potřebnou paměť dynamicky.
- Seznam vytvoříme voláním funkce `create()`.
- Nastavíme funkce `isLess()`.

- Přidáním řádku do seznamu delegujeme zoodpovědnost za správu paměti (smazání) seznamu, nebo následnému volání `pop()` a smazání řádku.
- Obsah seznamu vytiskneme voláním `pop()`.

Kódovací příklad – list.c 3/3

```
66 static void pushLess(struct list *list, struct item *item)
67 {
68     if (!list->root || list->isLess(item->value, list->root->value)) {
69         item->next = list->root; // ok i pro root == NULL
70         list->root = item;
71         return;
72     }
73     struct item *prev = list->root;
74     struct item *cur = prev->next; // list->root není NULL
75     while (cur && !list->isLess(item->value, cur->value)) {
76         prev = cur;
77         cur = cur->next;
78     }
79     item->next = cur; // item bude poslední if cur == NULL
80     prev->next = item;
81 }
82 }
```

list.c

```
84 void push(void *list, void *value)
85 {
86     struct list *lst = (struct list*)list;
87     if (!lst || !value) { // ukládání hodnot NULL
88         return; // není podporováno (ignorujeme)
89     }
90     struct item *n = allocate_item(value);
91     #ifdef WITH_LESS // isLess pouze pokud WITH_LESS
92     if (lst->isLess) {
93         pushLess(lst, n);
94         return;
95     }
96     #endif
97     if (lst->root) {
98         n->next = lst->root;
99     }
100     lst->root = n;
101 }
102 }
```

list.c

- Privátní funkce `pushLess()` v rámci `list.c`.
- Funkce využívá nastavenou funkci `list->isLess`.
- Funkce vkládá `item` před první položku seznamu, která je větší (dle `isLess()`).

- Výchozí implementace `push()` vkládá hodnoty na začátek seznamu.
- Pokud je `list.c` kompilován s `WITH_LESS`, dochází k využití `isLess()`.
Např. `clang -DWITH_LESS list.c`

Kódovací příklad – Volání rozhraní seznamu 2/3

```
$ clang -fPIC -c list.c
$ clang -shared -o liblist.so.1 list.o
$ clang -g -DWITH_LESS -fPIC -c list.c
$ clang -shared -o liblist.so.2 list.o
$ ln -s liblist.so.1 liblist.so
$ clang -g -L. -Wl,-rpath=. -llist -o demo demo.c
$ ldd demo
demo:
liblist.so => ./liblist.so (0x80024d000)
libc.so.7 => /lib/libc.so.7 (0x800251000)
```

- Vytvoříme vstupní soubor `in.txt` a přesměrujeme `stdin`.

```
1 cat in.txt
2 4
3 2
4 16
5 13
6 6
7 1
8 3
9 5
10 9
11 15
12 ./demo <in.txt 2>&1 | head -n 12
13 DEBUG: read " 4"
14 DEBUG: read " 2"
15 DEBUG: read " 16"
16 DEBUG: read " 13"
17 DEBUG: read " 6"
18 DEBUG: read " 1"
19 DEBUG: read " 3"
20 DEBUG: read " 5"
21 DEBUG: read " 9"
22 DEBUG: read " 15"
23 Popped value is " 15"
24 Popped value is " 9"
```
- Vytvoříme dvě verze (bez/s `isLess()`) dynamicky linkované knihovny `liblist.so.1` a `liblist.so.2`.
- Konkrétní (aktuální verzi) odkážeme symbolickým odkazem (nebo jen nakopírujeme) jako soubor `liblist.so`.
- Program `demo.c` zkompilujeme s knihovnou `list`.
- Dynamicky linkované knihovny programu můžeme zobrazit, např. nástrojem `ldd`.
`ldd - list dynamic object dependencies.`

Kódovací příklad – Volání rozhraní seznamu 3/3

- Verze bez `isLess()`, knihovna `liblist.so.1`.
- Verze s `isLess()`, knihovna `liblist.so.2`.

```
$ rm -rf liblist.so
$ ln -s liblist.so.1 liblist.so
$ ls -l liblist.so
lrwxr-xr-x 1 liblist.so -> liblist.so.1
$ ./demo < in.txt 2>/dev/null
Popped value is " 15"
Popped value is " 9"
Popped value is " 5"
Popped value is " 3"
Popped value is " 1"
Popped value is " 6"
Popped value is " 13"
Popped value is " 16"
Popped value is " 2"
Popped value is " 4"
```

```
$ rm -rf liblist.so
$ ln -s liblist.so.2 liblist.so
$ ls -l liblist.so
lrwxr-xr-x 1 liblist.so -> liblist.so.2
$ ./demo < in.txt 2>/dev/null
Popped value is " 1"
Popped value is " 13"
Popped value is " 15"
Popped value is " 16"
Popped value is " 2"
Popped value is " 3"
Popped value is " 4"
Popped value is " 5"
Popped value is " 6"
Popped value is " 9"
```