

# Spojové struktury

Jan Faigl

Katedra počítačů  
Fakulta elektrotechnická  
České vysoké učení technické v Praze

Přednáška 08

BAB36PRGA – Programování v C

## 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 (HW8)

## 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), 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.

## 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. *NULL v případě posledního prvku seznamu (zarážka).*
- 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 nově odkazuje na následující prvek, na který odkazoval odebraný prvek.
- Základní implementací spojového seznamu je **jednosměrný spojový seznam**.

## 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.
- 1. Nejdříve sekvencně najdeme prvek s hodnotou 79.
- 2. Následně vyjme prvek s hodnotou a propojíme prvek 93 s prvkem 50.  
*Položku next prvku 93 nastavíme na hodnotu next odebraného 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:
  1. Ukazatel na první prvek **head**;
  2. nebo vlastní struktura pro seznam. *Vhodné pro uložení dalších informací, počet prvků, poslední prvek.*
- Příklad struktur pro uložení spojového seznamu celých čísel.

```
1 typedef struct entry { 1 typedef struct {
2 int value; 2 entry_t *head;
3 struct entry *next; 3 entry_t *tail;
4 } entry_t; 4 int counter; // počet prvků
6 entry_t *head = NULL; 5 } linked_list_t;
```
- Pro jednoduchost prvky seznamu obsahují celé číslo. *Obecně mohou obsahovat libovolná data (ukazatel na strukturu).*

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### Přidání prvku – příklad

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

```
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;
```

- 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.

**Kontrola dynamické alokace**

```
#include <stdlib.h>

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`

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 12 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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
5   new_entry->value = value; // set data
6   if (*head == NULL) { // first entry in the list
7     new_entry->next = NULL; // reset the next
8   } else {
9     new_entry->next = *head;
10  }
11  *head = new_entry; //update the head
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()` –  $O(1)$ .

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 13 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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()` –  $O(1)$ .

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 14 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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;
```

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

- Procházejíme kompletní seznam ( $n$  prvků), abychom spočítali počet prvků seznamu.

Lineární složitost operace `size()` –  $O(n)$ .

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 15 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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()` –  $O(n)$ .

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 16 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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.

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 17 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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`  
`lec08/demo-linked_list-int.c`

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 18 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

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

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

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 20 / 50

Úvod Spojový seznam Start/End Vložení/odebrání prvku Kruhový spojový seznam Obousměrný seznam

### 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;
}
```

Jan Faigl, 2024 BAB36PRGA – Prednáška 08: Spojové struktury 21 / 50

### 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 – pop() s odkazem na konec seznamu

```
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); // release 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 odebráme z začátku.*

### Spojový seznam – back() s odkazem na konec 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
4   // assured programmatically
5   assert(list && list->tail);
6   return list->tail->value;
7 }
```

*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 – pushEnd()

```
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.*

### Spojový seznamu – popEnd()

```
1 int popEnd(linked_list_t *list)
2 {
3   assert(list && list->head);
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 = end->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 spojovým seznamem.*

### Příklad použití

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

### 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 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).*

■ Nalezení první implementujeme „privátní“ (static) funkcí getEntry().

### Spojový seznam – getEntry()

```
1 static entry_t * getEntry(int index, const linked_list_t *list)
2 { // here, we assume index >= 0
3   entry_t *cur = list->head;
4   int i = 0;
5   while (i < index && cur != NULL && cur->next != NULL) {
6     cur = cur->next;
7     i += 1;
8   }
9   return cur; //return entry at the index or the last entry
10 }
```

*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.

### Spojový seznam – insertAt()

```
1 void insertAt(int value, int index, linked_list_t *list)
2 {
3   assert(list); // list != NULL
4   if (index < 0) { return; } // only positive position
5   if (index == 0) { // handle the 1st position
6     push(value, list);
7     return;
8   }
9   entry_t *new_entry = myMalloc(sizeof(entry_t));
10  new_entry->value = value; // set data
11  entry_t *entry = getEntry(index - 1, list);
12  if (entry != NULL) { // entry can be NULL for the 1st
13    new_entry->next = entry->next; // entry (empty list)
14    entry->next = new_entry;
15  }
16  if (entry == list->tail) {
17    list->tail = new_entry; // update the tail
18  }
19  list->counter += 1;
20 }
```

*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!!!

```

```

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

### 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!!!

```

```

$ clang linked_list.c demo-removeat.c
    && ./a.out
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

```

```

lec08/demo-removeat.c

```

### Příklad spojový seznamu textových řetězců

- 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 };
linked_list_t *lst = &list;

push("FEE", lst); push("CTU", lst);
push("PRG", lst); push("lec08", lst);
print(lst);
for (int i = 0; i < 2; ++i) {
  char *str = pop(lst);
  printf("Md. %s\n", i+1, str);
  free(str); // Free is needed!
}

```

```

$ ./demo-linked_list-str
lec08 PRG CTU FEE
1. popped string lec08
2. popped string PRG
Popped value "CTU"
String of the popped value is at address 0
x80024c010
Due to selected implementation the memory must
be explicitly deallocated!

```

```

lec08/linked_list-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!

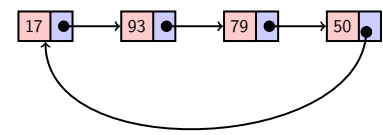
```

lec08/linked_list-str.h lec08/demo-linked_list-str.c

```

### Kruhový spojový seznam

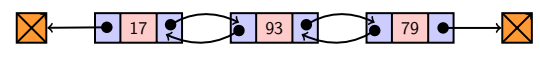
- Položka next posledního prvku může odkazovat na první prvek.
- Tak vznikne kruhový spojový seznam.



- Při přidání prvku na začátek je nutné aktualizovat hodnotu 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)
- Alokaci prvku provedeme funkci 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:
  - Napojení vloženého prvku do seznamu, hodnoty prev a next;
  - Aktualizace next předchozí prvku k prvku cur;
  - 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 – 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
```

### 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
```

### 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);

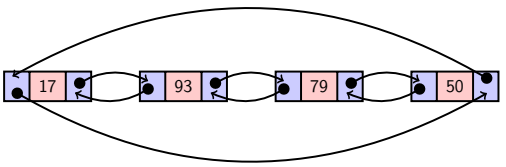
printf("Revert print: ");
printReverse(lst);

free_dll(lst);
//lec08/doubly_linked_list.c
//lec08/demo-doubly_linked_list.c
```

■ 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

### 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 (HW8)

### Zadání 8. domácího úkolu HW8

#### Téma: Fronta spojovým seznamem s řazením

Povinné zadání: 3b; Volitelné zadání: 4b; Bonusové zadání: není

- **Motivace:** Práce s pamětí a datovými strukturami.
- **Cíl:** Prohloubit si znalost paměťové reprezentace a dynamické alokace při práci se spojovým seznamem.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/bab36prga/hw/hw8>
  - Implementace spojového seznamu s vkládáním a odebráním prvků.
  - **Volitelné zadání** rozšiřuje úlohu o vkládáním prvků s řazením.
- **Termín odevzdání:** 11.05.2024, 23:59:59 PDT.

### 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 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`. Návroha 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.
- Vytvoríme 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
7 void setLess(void *list, bool (*isLess)(const void *a, const void *b));
8
9 void push(void *list, void *value);
10 void* pop(void *list);
11
12 #endif
                                     list.h

```

## Kódovací příklad – list.c 1/2

```

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

```

- 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`.

```

18 void* create(void)
19 {
20     struct list *ret = malloc(sizeof(struct list));
21     if (!ret) {
22         fprintf(stderr, "ERROR: cannot allocate memory
23         for list!\n");
24         exit(ERROR_MEM);
25     }
26     struct item *cur = NULL;
27     ret->root = NULL;
28     ret->isLess = NULL;
29     return ret;
30 }
31
32 void release(void** list)
33 {
34     if (!list || !*list) {
35         struct list *list = (struct list)*list;
36         struct item *cur = list->root;
37         while (cur) {
38             struct item *t = cur;
39             cur = cur->next;
40             free(t->value); // Připadá specifická funkce
41             free(t); // Pro složený typ s ukazateli
42         }
43         free(*list);
44         *list = NULL;
45     }
46 }

```

## 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 *list = (struct list)*list;
51         list->isLess = isLess;
52     }
53     struct item *p = NULL;
54     if (!p) {
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         }
66     }
67     struct item *ret = malloc(sizeof(struct item));
68     if (!ret) {
69         fprintf(stderr, "ERROR: cannot allocate memory for list item!\n");
70         exit(ERROR_MEM);
71     }
72     ret->value = value;
73     ret->next = NULL;
74     return ret;
75 }
                                     list.c

```

```

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

```

- 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 návratovou hodnotou funkce).

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

## 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->
69     value)) {
70         item->next = list->root; // ok 1 pro root == NULL
71         list->root = item;
72         return;
73     }
74     struct item *prev = list->root;
75     struct item *cur = prev->next; // list->root není NULL
76     while (cur && list->isLess(item->value, cur->value)) {
77         prev = cur;
78         cur = cur->next;
79     }
80     item->next = cur; // item bude poslední if cur == NULL
81     prev->next = item;
82 }
                                     list.c

```

```

83 void push(void *list, void *value)
84 {
85     struct list *list = (struct list)*list;
86     if (!list || !value) { // ukládání hodnot NULL
87         return; // není podporováno (ignorujeme)
88     }
89     struct item *n = allocate_item(value);
90     #ifdef WITH_LESS // isLess pouze pokud WITH_LESS
91     if (!list->isLess) {
92         pushLess(list, n);
93         return;
94     }
95     #endif
96     if (!list->root) {
97         n->next = list->root;
98     }
99     list->root = n;
100 }
                                     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 hodnotu 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 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     size_t ln = 0; // délka načteného řetězce
16     void *list = create(); // vytvoření seznamu z list.h
17
18     setLess(list, isLess); // nastavení funkce isLess() demo.c

```

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

```

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

```

- 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 – Volání rozhraní seznamu 2/3

```

1 $ clang -fPIC -c list.c
2 $ clang -shared -o liblist.so.1 list.o
3
4 $ clang -g -DWITH_LESS -fPIC -c list.c
5 $ clang -shared -o liblist.so.2 list.o
6
7 $ ln -s liblist.so.1 liblist.so
8
9 $ clang -g -L. -Wl,-rpath=. -llist -o demo demo.c
10
11 $ ldd demo
12     liblist.so => ./liblist.so (0x80024000)
13     libc.so.7 => /lib/libc.so.7 (0x800251000)

```

- Vytvoríme 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`.

```

13 ./demo <in.txt 2>/dev/null
14 DEBUG: read " 4"
15 DEBUG: read " 2"
16 DEBUG: read " 16"
17 DEBUG: read " 13"
18 DEBUG: read " 6"
19 DEBUG: read " 1"
20 DEBUG: read " 3"
21 DEBUG: read " 5"
22 DEBUG: read " 9"
23 DEBUG: read " 15"
24 Popped value is " 15"
25 Popped value is " 9"

```

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`.

```

1 $ rm -rf liblist.so
2 $ ln -s liblist.so.1 liblist.so
3 $ ls -l liblist.so
4 lrwxr-xr-x 1 liblist.so -> liblist.so.1
5
6 $ rm -rf liblist.so
7 $ ln -s liblist.so.2 liblist.so
8 $ ls -l liblist.so
9 lrwxr-xr-x 1 liblist.so -> liblist.so.2

```

```

10 $ ./demo <in.txt 2>/dev/null
11 Popped value is " 15"
12 Popped value is " 1"
13 Popped value is " 13"
14 Popped value is " 15"
15 Popped value is " 16"
16 Popped value is " 2"
17 Popped value is " 3"
18 Popped value is " 4"
19 Popped value is " 5"
20 Popped value is " 6"
21 Popped value is " 9"
22 Popped value is " 15"
23 Popped value is " 2"
24 Popped value is " 9"

```