9. Spojové struktury. Strom.

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Lineární spojové struktury
  - Jednosměrný spojový seznam
  - Kruhový spojový seznam
  - Obousměrný spojový seznam
- Část 2 – Stromy
  - Stromy

Část I

Lineární spojové struktury

Kolekce prvků

- V programech je velmi běžný požadavek na uchování seznamu (množiny) prvků

  Prvky mohou být hodnoty nebo struktury

- Základní kolekce je pole
  - Jedná se o kolekci položek (proměnných) stejného typu
  + Umožňuje jednoduchý přístup k položkám indexací prvku

    Položky jsou stejného typu (velikosti).

  – Velikost pole je určena při vytvoření pole
    - Velikost (maximální velikost) musí být známa v době vytváření
    - Změna velikost v podstatě není přímo možná

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

    - Využití pouze malé části pole je mrháním paměti
  - V případě řazení pole přesouváme položky
    - Vložení prvku a vyjmutí prvku vyžaduje kopírování

Seznam

- Seznam (proměnných nebo objektů) patří mezi základní datové struktury

Základní ADT – Abstract Data Type

- Seznam zpravidla nabízí sadu základních operací:
  - `insert()` – vložení prvku
  - `remove()` – odebrání prvku
  - `index_of()` – vyhledání prvku
  - `size()` – aktuální počet prvku v seznamu
- Implementace seznamu může být různá:
  - Pole
    - Indexování je velmi rychlé
    - Vložení prvku na konkrétní pozici může být pomalé

Nová alokace a kopírování.

  - Spojové seznamy

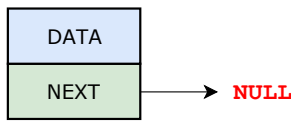
Spojový seznam

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

NULL v případě posledního prvku seznamu.

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

Realizujeme jej jako ukazatel odkazující na první prvek seznamu.



Základní operace se spojovým seznamem

- Vložení prvku
  - Předchozí prvek odkazuje na nový prvek
  - Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje

  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

I. Lineární spojové struktury

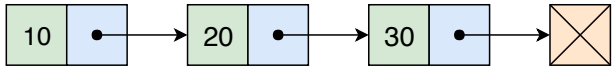
Jednosměrný spojový seznam

Kruhový spojový seznam

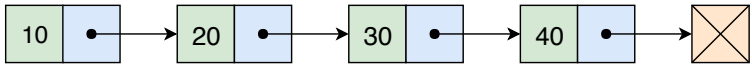
Obousměrný spojový seznam

Jednosměrný spojový seznam

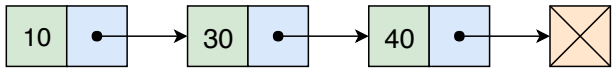
- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 40 na konec seznamu



- Odebrání prvku 20 ze seznamu



1. Nejdříve sekvenčně najdeme prvek s hodnotou 30

   Prvek, na který odkazuje NEXT odebíraného prvku.

2. Následně vyjmeme a napojíme prvek 10 na prvek 30

   Hodnotu NEXT prvku 10 nastavíme na adresu prvku 30.

Jednosměrný spojový seznam

- Seznam tvoří struktura prvku
  - Vlastní data prvku
  - Odkaz (ukazatel) na další prvek
- Vlastní seznam
  - Ukazatel na první prvek `head`,
  - nebo vlastní struktura pro seznam
- Příklad struktur pro uložení spojového seznamu celých čísel

Obecně mohou obsahovat libovolná data.

```c
/* polozka */
typedef struct entry
{
  int value;
  struct entry *next;
} entry_t;
entry_t *head = NULL;
```

```c
/* spojova struktura */
typedef struct
{
  entry_t *head;
  entry_t *tail;
  int counter; // pocet prvku
} linked_list_t;
```

Vytvoření a propojení prvků seznamu

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

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

2. Další prvek (20) přidáme propojením s aktuálně 1. prvkem

```
entry_t *new = (entry_t*)malloc(sizeof(entry_t));
new->value = 13;
new->next = head
```

3. a aktualizací proměnné `head`

```
head = new;
```

Přístup k prvkům seznamu

- 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

Přidání prvku na začátek seznamu

- Předáváme adresu, kde je uložen odkaz na start seznamu
- Head bude aktualizován na adresu nově vloženého prvku

```
1  void push(int value, entry_t **head) {
2    entry_t *new = (entry_t*)malloc(sizeof(entry_t));
3    new->value = value; // set data
4    if (*head == NULL)  // first entry in the list
5      new->next = NULL; // reset the next
6    else
7      new->next = *head;
8    *head = new;        // update the head
9  }
```

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

Odebrání prvního prvku ze seznamu

```
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;
6    *head = prev_head->next; // will be set to NULL if
7    // the last item is popped
8    free(prev_head); // relase memory of the popped entry
9    return ret;
10 }
```

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

Zjištění počtu prvků v seznamu

- Funkce vyžaduje projít seznam až k zarážce `NULL`, tj. položka `next` je `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 }
```

- Pro zjištění počtu prvků v seznamu musíme projít kompletní seznam, tj. `n` položek

Vrácení hodnoty posledního prvku ze seznamu

- Pro vrácení hodnoty posledního prvku v seznamu musíme projít všechny položky seznamu

```
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  }
```

Procházení seznamu

```c
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.

Vložení prvku

- 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

```
1  entry_t *new = (entry_t*)malloc(sizeof(entry_t));
2  // nastaveni hodnoty
3  new->value = value;
4  //propojeni s nasledujicim
5  new->next = entry->next;
6  //propojeni entry
7  entry->next = new;
```

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

Zajímavou variantou je vkládání podle velikosti – insert sort.

I. Lineární spojové struktury

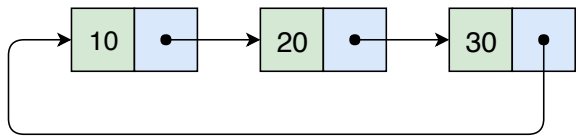
Jednosměrný spojový seznam

Kruhový spojový seznam

Obousměrný spojový seznam

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



I. Lineární spojové struktury

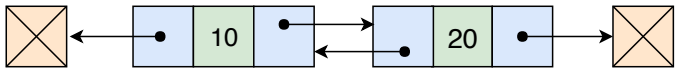
Jednosměrný spojový seznam

Kruhový spojový seznam

Obousměrný spojový seznam

Obousměrný spojový seznam

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



9.1 Implementace zásobníku spojovým seznam 1/4

- Definujeme strukturu `stack_entry_t` pro položku seznamu

```
1 typedef struct entry {
2   void *value; //ukazatel na hodnotu vlozeného prvku
3   struct entry *next;
4 } stack_entry_t;
```

- Struktura zásobníku `stack_t` obsahuje pouze ukazatel na `head`

```
1 typedef struct {
2   stack_entry_t *head;
3 } stack_t;
```

- Inicializace tak pouze alokuje strukturu `stack_t`

```
1 void stack_init (stack_t **stack) {
2   *stack = (stack_t *)malloc(sizeof(stack_t));
3   (*stack)->head = NULL;
4 }
```

- Při vkládání prvku alokujeme položku spojového seznamu

```
1  int stack_push (void *value, stack_t *stack) {
2    int ret = STACK_OK;
3    stack_entry_t *new_entry = (stack_entry_t *) malloc(sizeof(
       stack_entry_t));
4    if (new_entry) {
5      new_entry->value = value;
6      new_entry->next = stack->head;
7      stack->head = new_entry;
8    } else {
9      ret = STACK_MEMFAIL;
10   }
11   return ret;
12 }
```

- Při vyjmutí prvku paměť uvolňujeme

```
1  void* stack_pop (stack_t *stack) {
2    void *ret = NULL;
3    if (stack->head) {
4      ret = stack->head->value; //retrive the value
5      stack_entry_t *tmp = stack->head;
6      stack->head = stack->head->next;
7      free (tmp);  // release stack_entry_t
8    }
9    return ret;
10 }
```

- Zjištění prázdnosti

```
1  int stack_is_empty (const stack_t *stack) {
2    return stack->head == 0;
3  }
```

- Prvek na vrcholu zásobníku

```
1  void* stack_peek (const stack_t *stack) {
2    return stack_is_empty (stack) ? NULL : stack->head->value;
3  }
```

- Použití je identické jako v předchozím případě
  - Výhoda spojového seznamu proti implementaci v poli je v (téměř) neomezené kapacitě zásobníku

    Jsem omezen pouze dostupnou pamětí.

  - Nevýhodou spojového seznamu je větší paměťová režie

9.2 Implementace fronta spojovým seznamem 1/4

- Spojový seznam s udržováním začátku `head` a konce `end` seznamu
- Strategie vkládání a odebírání prvků
  - Vložením prvku do fronty `queue_push()` dáme prvek na konec seznamu `end`
  - Odebrání prvku z fronty `queue_pop()` vezmeme prvek z počátku seznamu `head`
  - Nemusíme tak lineárně procházet seznam a aktualizovat `end` při odebrání prvku z fronty

```
typedef struct entry {
  void *value;
  struct entry *next;
} queue_entry_t;
typedef struct {
  queue_entry_t *head;
  queue_entry_t *end;
} queue_t;
```

```
void queue_init(queue_t **
    queue) {
  *queue = (queue_t*)malloc (
    sizeof(queue_t));
  (*queue)->head = NULL;
  (*queue)->end = NULL;
}
```

```c
int queue_push (void *value, queue_t *queue) {
  int ret = QUEUE_OK;
  queue_entry_t *new_entry = malloc(sizeof(queue_entry_t));
  if (new_entry) { // fill the new_entry
    new_entry->value = value; new_entry->next = NULL;
    if (queue->end) { // if queue has end
      queue->end->next = new_entry; // link new_entry
    } else { // queue is empty
      queue->head = new_entry; // update head as well
    }
    queue->end = new_entry; // set new_entry as end
  } else
    ret = QUEUE_MEMFAIL;
  return ret;
}
```

```
1  void* queue_pop (queue_t *queue) {
2    void *ret = NULL;
3    if (queue->head) { // having at least one entry
4      ret = queue->head->value; //retrive the value
5      queue_entry_t *tmp = queue->head;
6      queue->head = queue->head->next;
7      free (tmp);  // release queue_entry_t
8      if (queue->head == NULL) { // update end if last
9        queue->end = NULL;       // entry has been
10     }                           // popped
11   }
12   return ret;
13 }
```

```
1  int queue_is_empty (const queue_t *queue) {
2    return queue->head == 0;
3  }
4  void* queue_peek (const queue_t *queue) {
5    return queue_is_empty (queue) ? NULL : queue->head->value;
6  }
```

Část II

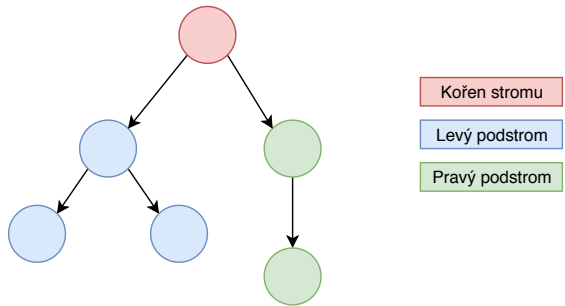
Stromy

II. Stromy

Stromy

Lineární a nelineární spojové struktury

- Spojové seznamy predstavují lineární spojovou strukturu
  - Každý prvek má nejvýše jednoho následníka
- Nelineární spojové struktury (např. stromy)
  - Každý prvek může mít více následníků
- **Binární strom** – každý prvek (uzel) má nejvýše dva následníky



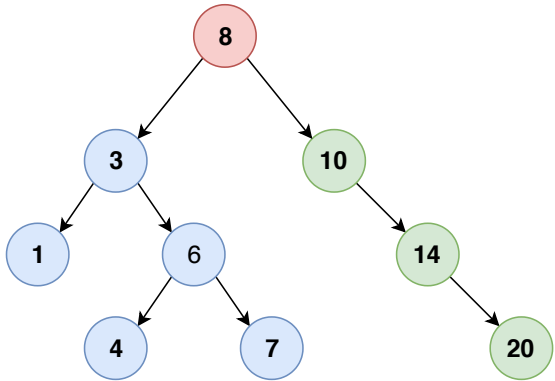
Binární strom

- uvažujme datové položky uzlu stromu jako hodnoty typu int
- uzel stromu reprezentujeme strukturou node_t
- strom je pak reprezentován kořenem stromu, ze kterého máme přístup k jednotlivým uzlům
  - potomci left a right a jejich potomci

```
1  typedef struct node {
2    int value;
3    struct node *left;
4    struct node *right;
5  } node_t;
6  node_t *tree;
```

9.3 Binární vyhledávací strom

- Binární vyhledávací strom – Binary Search Tree (BST)
- Pro každý prvek (uzel) platí
  - hodnota (value) potomka vlevo je menší (nebo NULL)
  - hodnota potomka vpravo je větší (nebo NULL)



- pro vložení prvku nejprve dynamicky alokujeme uzel

```
1  static node_t* newNode(int value)
2  {
3    node_t *node= (node_t*)malloc(sizeof(node_t));
4    node->value = value;
5    node->left = node->right = NULL;
6    return node;
7  }
```

- vložení alokovaného prvku – využijeme rekurze a vkládáme na první volné vhodné místo, splňující podmínku BST

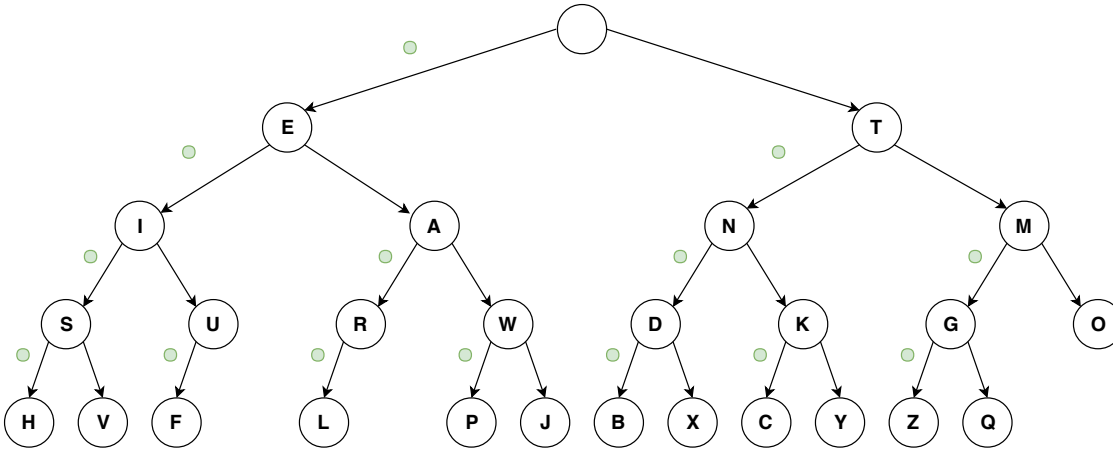
```
1  node_t* tree_insert(int value, node_t *node)
2  {
3    if (node == NULL) {
4      return newNode(value); // vracíme nový uzel
5    } else {
6      if (value <= node->value) { //vložení do levého podstromu
7        node->left = tree_insert(value, node->left);
8      } else { // vložení do pravého podstromu
9        node->right = tree_insert(value, node->right);
10     }
11     return node; // vracíme vstupní uzel!!!
12   }
13 }
```

9.3 Průchod binárním vyhledávacím stromem

- Při hledání prvku konkrétní hodnoty se postupne zanořujeme hlouběji do stromu.
- Může nastat jedna z následujících situací:
  - Aktuální prvek má hledanou hodnotu klíče
    - hledání je ukončeno
  - Hodnota klíče je menší než hodnota aktuálního prvku
    - pokračujeme v hledání v další úrovni levého potomka
  - Hodnota klíče je větší než hodnota aktuálního prvku
    - pokračujeme v hledání v další úrovni pravého potomka
  - Aktuální prvek má hodnotu null
    - hledání je ukončeno, prvek ve stromu není
- Při průchodu stromem postupujeme rekurzivně
  - nejdříve navštěvujeme levé potomky
  - následne pak pravé potomky

9.4 Dekódování morseovky

- Pro dekódování textu zapsaného v Morseově abecedě lze použít následující binární strom



Příklad – dekódování morseovky

```
1  typedef struct MUzel {
2    char znak;
3    struct MUzel *tecka, *carka;
4  } MUzel;
6  MUzel *novyMUzel(char z, MUzel *t, MUzel *c) {
7    MUzel *p = (MUzel*)malloc(sizeof(MUzel));
8    p->znak = z; p->tecka = t; p->carka = c;
9    return p;
10 }
12 MUzel *novyMUzel1(char z) {
13   MUzel *p = (MUzel*)malloc(sizeof(MUzel));
14   p->znak = z;
15   p->tecka = NULL; p->carka = NULL;
16   return p;
17 }
```

9.4 Vytvoření stromu

```
MUzel *strom(void) {
  return novyMUzel(' ',
    novyMUzel('E', /* . */
      novyMUzel('I', /* .. */
        novyMUzel('S', /* ... */
          novyMUzel1('H'), /* .... */
          novyMUzel1('V') /* ...- */
        ),
        novyMUzel('U', /* ..- */
          novyMUzel1('F'), /* ..-. */
          NULL /* ..-- */
      )),
    novyMUzel('A', /* .- */ ...)),
  novyMUzel('T', /* - */ ...));
}
```

```
1  void dekoduj(char odkud[], char kam[]) {
2
3    MUzel *aktualni = koren;
4    int i = 0, j = 0, delka = strlen(odkud);
5
6    while (i < delka)
7    {
8      if (aktualni!=NULL) {
9        if (odkud[i] == '.')
10         aktualni = aktualni->tecka;
11       else if (odkud[i] == '-')
12         aktualni = aktualni->carka;
13       else
14       {
15         // cont.
```

```
          kam[j++] = aktualni->znak;
          aktualni = koren;
        }
        i++;
      }
      else
      {
        kam[j++] = '?';
        while (odkud[i] == '.' || odkud[i] =='-')
          i++;
        aktualni = koren;
      }
    }
    kam[j] = 0;
}
```

9.4 Hlavní program

```
1  MUzel *koren;
2  #define MAXDELKA 100
4  int main()
5  {
6    koren = strom();
7    char morse[MAXDELKA], text[MAXDELKA];
9    fgets(morse, MAXDELKA, stdin);
11   dekoduj(morse, text); // + uklid strom
13   fprintf(stdout, "%s\n", text);
15   return 0;
16 }
```