

# Stromy

Jan Faigl

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 10

**B0B36PRP – Procedurální programování**



# Přehled témat

- Část 1 – Standardní knihovny, čtení/zápis ze/do souboru

Stromy

Binární strom

Příklad binárního stromu v C

Stromové struktury

- Část 2 – Příklad načítání grafu, kompilace a projekt s více soubory
- Část 3 – Zadání 9. domácího úkolu (HW09)



# Část I

## Část 1 – Stromy



# Obsah

Stromy

Binární strom

Příklad binárního stromu v C

Stromové struktury



## Lineární a nelineární spojové struktury

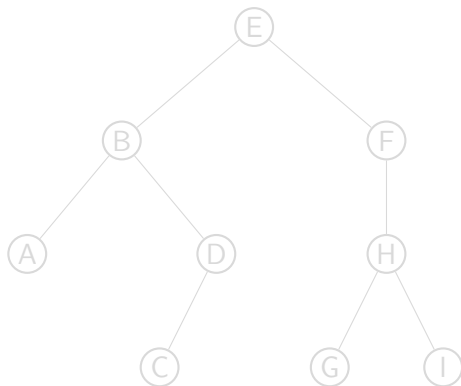
- Spojové seznamy představují lineární spojovou strukturu

*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



- kořen stromu
- list
- levý podstrom
- pravý podstrom



## Lineární a nelineární spojové struktury

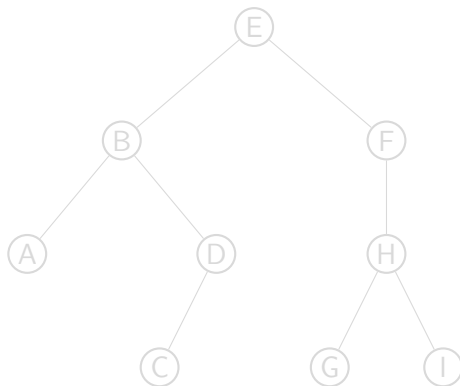
- Spojové seznamy představují lineární spojovou strukturu

*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



- kořen stromu
- list
- levý podstrom
- pravý podstrom



## Lineární a nelineární spojivé struktury

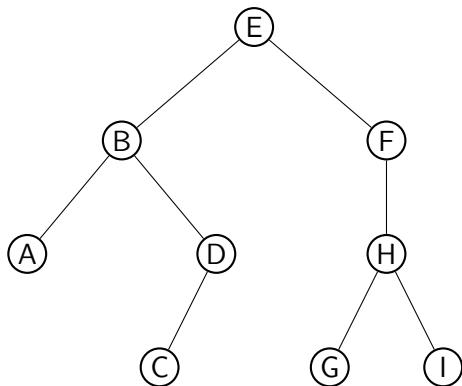
- Spojivé seznamy představují lineární spojivou strukturu

*Každý prvek má nejvýše jednoho následníka*

- Nelineární spojivé struktury (např. stromy)

*Každý prvek může mít více následníků*

- **Binární strom:** každý prvek (uzel) má nejvýše dva následníky



- kořen stromu
- list
- levý podstrom
- pravý podstrom



# Obsah

Stromy

**Binární strom**

Příklad binárního stromu v C

Stromové struktury





## Binární strom

- Pro přehlednost uvažujme datové položky uzlů stromu jako hodnoty typu `int`
- Uzel stromu reprezentujeme strukturou `node_t`

```
typedef struct node {  
    int value;  
    struct node *left;  
    struct node *right;  
} node_t;
```

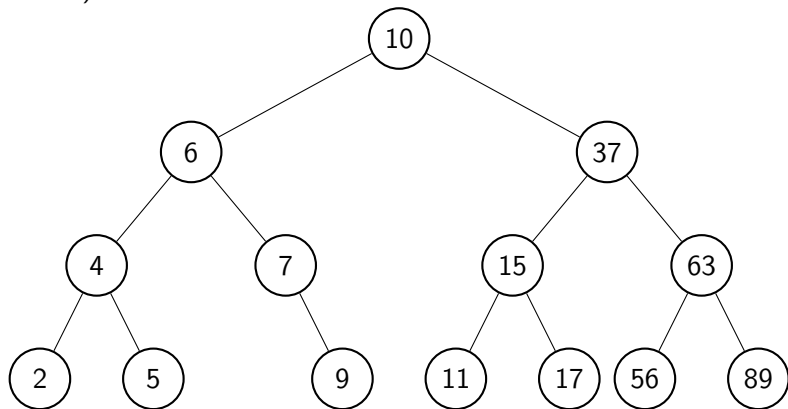
- Strom je pak reprezentován kořenem stromu, ze kterého máme přístup k jednotlivým uzlům (potomci `left` a `right` a jejich potomci)

```
node_t *tree;
```



## Příklad – Binární vyhledávací strom

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



## BST – tree\_insert() 1/2

- Při vložení prvku dynamicky alokujeme uzel pomocnou (lokální) funkcí, např. `newNode()`

```
static node_t* newNode(int value)
{
    node_t *node= (node_t*)malloc(sizeof(node_t));
    node->value = value;
    node->left = node->right = NULL;
    return node;
}
```

lec10/tree-int.c

- Uvedením klíčového slova `static` je funkce viditelná pouze v modulu `tree-int.c`



## BST – tree\_insert() 2/2

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

*Binární vyhledávací strom nemusí být nutně vyvážený!*

```
node_t* tree_insert(int value, node_t *node)
{
    if (node == NULL) {
        return newNode(value); // vracíme nový uzel
    } else {
        if (value <= node->value) { //vložení do levého podstromu
            node->left = tree_insert(value, node->left);
        } else { // vložení do pravého podstromu
            node->right = tree_insert(value, node->right);
        }
        return node; // vracíme vstupní uzel!!!
    }
}
```

lec10/tree-int.c



## Průchod binárním vyhledávacím stromem

- Při hledání prvku konkrétní hodnoty se postupně zanořujeme hlouběji do stromu. Může nastat jedna z následujících situací:

*Např. hodnota `value` představuje klíč nějaké datové položky.*

1. Aktuální prvek má hledanou hodnotu klíče, hledání je ukončeno
  2. Hodnota klíče je menší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni levého potomka
  3. Hodnota klíče je větší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni pravého potomka
  4. Aktuální prvek má hodnotu **null**, hledání je ukončeno, prvek ve stromu není
- Při průchodu stromem postupujeme rekurzivně tak, že nejdříve navštívujeme levé potomky a následně pak pravé potomky

*Pokud budeme při takovém průchodu vypisovat hodnoty v levém podstromu, pak hodnotu prvku a následně hodnoty v pravém podstromu, vypíšeme hodnoty uložené ve stromu uspořádaně (sestupně nebo vzestupně, podle toho jestli vlevo jsou prvky menší nebo větší.)*



## Průchod binárním vyhledávacím stromem

- Při hledání prvku konkrétní hodnoty se postupně zanořujeme hlouběji do stromu. Může nastat jedna z následujících situací:

*Např. hodnota `value` představuje klíč nějaké datové položky.*

1. Aktuální prvek má hledanou hodnotu klíče, hledání je ukončeno
  2. Hodnota klíče je menší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni levého potomka
  3. Hodnota klíče je větší než hodnota aktuálního prvku, pokračujeme v hledání v další úrovni pravého potomka
  4. Aktuální prvek má hodnotu **null**, hledání je ukončeno, prvek ve stromu není
- Při průchodu stromem postupujeme rekurzivně tak, že nejdříve navštěvujeme levé potomky a následně pak pravé potomky

*Pokud budeme při takovém průchodu vypisovat hodnoty v levém podstromu, pak hodnotu prvku a následně hodnoty v pravém podstromu, vypíšeme hodnoty uložené ve stromu uspořádaně (sestupně nebo vzestupně, podle toho jestli vlevo jsou prvky menší nebo větší.)*



# Obsah

Stromy

Binární strom

Příklad binárního stromu v C

Stromové struktury



## Binární strom celočíselných hodnot `int`

- Kromě vložení prvků do stromu funkcí `tree_insert()`,

*Viz předchozí příklad*

implementujete následující funkce:

- `tree_free()` – Kompletní smazání stromu, včetně uvolnění paměti všech prvků
- `tree_size()` – Vrátí počet prvků ve stromu
- `tree_print()` – Vypsání prvků uložených ve stromu (BST)

```
void tree_free(node_t **tree);  
int tree_size(const node_t *const tree);  
void tree_print(const node_t *const node);
```

`lec10/tree-int.h`





Příklad implementace `tree_free()`

```
void tree_free(node_t **tree)
{
    if (tree && *tree) {
        node_t * node = *tree;
        if ( node->left ) {
            tree_free(&(node->left));
        }
        if ( node->right ) {
            tree_free(&(node->right));
        }
        free(*tree);
        *tree = NULL; // fill the tree variable
                     // of the calling function to NULL
    }
}
```

lec10/tree-int.h

Předáváme ukazatel na ukazatel, abychom mohli po uvolnění paměti nastavit hodnotu ukazatele ve volající funkci na `NULL`.



## Příklad implementace `tree_size()` a `tree_print()`

- Určení počtu prvků implementujeme rekurzí

```
int tree_size(const node_t *const node)
{
    return node == NULL ?
        0 :
        tree_size(node->left) + 1 + tree_size(node->
right);
}
```

- Podobně výpis hodnot

```
void tree_print(const node_t *const node)
{
    if (node) {
        tree_print(node->left);
        printf("%d ", node->value);
        tree_print(node->right);
    }
}
```

lec10/tree-int.c



## Příklad použití – 1/3

- Strom naplníme `for` cyklem
- Vypíšeme počet prvků a uložené hodnoty funkcí `tree_print()`

```
...
for(int i = 0; i < n; ++i) {
    printf("Insert value %i\n", values[i]);
    if (root == NULL) {
        root = tree_insert(values[i], NULL);
    } else {
        tree_insert(values[i], root);
    }
}
printf("No. of tree nodes is %i\n", tree_size(root));

printf("Print tree: ");
tree_print(root);
printf("\n");

tree_free(&root);
printf("After tree_free() root is %p\n", root);
return 0;
}
```

lec10/demo-tree-int.c



## Příklad použití – 3/3

- Program spustíme bez a s argumentem `balanced`

<pre>clang tree-int.c demo-tree-int.c ./a.out Insert values2 that will result   in none balanced tree Insert value 5 Insert value 4 Insert value 6 Insert value 3 Insert value 7 Insert value 2 Insert value 8  No. of tree nodes is 7 Print tree: 2 3 4 5 6 7 8</pre>	<pre>clang tree-int.c demo-tree-int.c ./a.out balanced Insert values1 to make balanced   tree Insert value 5 Insert value 3 Insert value 7 Insert value 2 Insert value 4 Insert value 6 Insert value 8  No. of tree nodes is 7 Print tree: 2 3 4 5 6 7 8</pre>
--	--

- V obou případech je výpis uspořádaný

`lec10/demo-tree-int.c`

Jak otestovat, že operace na stromem (`tree_insert()`) zachová vlastnosti BST?



## Test vlastnosti binárního vyhledávacího stromu

- Ověření zdali je strom binárním vyhledávacím stromem otestujeme funkcí `tree_is_bst()`

```
_Bool tree_is_bst(const node_t *const node);
```

- Funkce rekurzivně projde strom a ověří, že pro každý uzel platí
  1. Hodnota uzlu není menší než nejvyšší hodnota v levém podstromu
  2. Hodnota uzlu není větší než nejmenší hodnota v pravém podstromu
  3. Podstrom levého následníka splňuje vlastnost BST
  4. Podstrom pravého následníka splňuje vlastnost BST
- K tomu potřebujeme pomocné funkce `getMaxValue()` a `getMinValue()`



## Příklad implementace `tree_is_bst()` - 1/3

- Za předpokladu BST můžeme maximální hodnotu nalézt iteračně

```
static int getMaxValue(const node_t *const node)
{
    const node_t *cur = node;
    while (cur->right) {
        cur = cur->right;
    }
    return cur->value;
}
```

- Podobně minimální hodnotu

```
static int getMinValue(const node_t *const node)
{
    const node_t *cur = node;
    while (cur->left) {
        cur = cur->left;
    }
    return cur->value;
}
```

lec10/tree-int.c



Příklad implementace `tree_is_bst()` - 2/3

```
_Bool tree_is_bst(const node_t *const node)
{
    _Bool ret = true;
    if (!node) {
        if (node->left
            && getMaxValue(node->left) > node->value) {
            ret = false;
        }
        if (ret && node->right
            && getMinValue(node->right) <= node->value) {
            ret = false;
        }
        if (ret
            && (
                !tree_is_bst(node->left)
                || !tree_is_bst(node->right)
            )) {
            ret = false;
        }
    }
    return ret;
}
```

lec10/tree-int.c



## Příklad implementace `tree_is_bst()` - 3/3

- Přidáme výpis a volání `tree_is_bst()`

```
...
printf("Max tree depth: %i\n", tree_max_depth(root));
printf("Tree is binary search tree (BST): %s\n",
       tree_is_bst(root) ? "yes" : "no");
```

- Program spustíme bez a s argumentem `balanced`

<pre>clang tree-int.c demo-tree-int.c ./a.out Insert values2 that will result   in none balanced tree ... Print tree: 2 3 4 5 6 7 8 Tree is binary search tree (BST):   yes Print tree by depth row</pre>	<pre>clang tree-int.c demo-tree-int.c ./a.out balanced Insert values1 to make balanced   tree ... Print tree: 2 3 4 5 6 7 8 Tree is binary search tree (BST):   yes</pre>
---	---

- V obou případech je podmínka BST splněna

`lec10/demo-tree-int.c`

Test sice indikuje, že strom je správně vytvořen, ale vizuálně nám výpis příliš nepomohl. V tomto jednoduchém případě si můžeme dále napsat funkci pro názornější výpis jednotlivých úrovní stromu. K tomu budeme potřebovat určení hloubky stromu.





## Příklad implementace `tree_max_depth()`

- Funkci implementujeme rekurzí

```
int tree_max_depth(const node_t *const node)
{
    if (node) {
        const int left_depth = tree_max_depth(node->left);
        const int right_depth = tree_max_depth(node->right);
        return left_depth > right_depth ?
            left_depth + 1 :
            right_depth + 1;
    } else {
        return 0;
    }
}
```

lec10/tree-int.c



## Výpis hodnot v konkrétní hloubce stromu `printDepth()`

- Výpis konkrétní vrstvy (hloubky) provedeme rekurzivně lokální funkcí `printDepth()`

```
static void printDepth(int depth, int cur_depth,
    const node_t *const node)
{
    if (depth == cur_depth) {
        if (node) {
            printf("%2d ", node->value);
        } else {
            printf(" - ");
        }
    } else if (node) {
        printDepth(depth, cur_depth + 1, node->left);
        printDepth(depth, cur_depth + 1, node->right);
    }
}
```

`lec10/tree-int.c`



## Příklad implementace výpisu stromu

### tree\_print\_layers()

- Výpis hodnot po jednotlivých vrstvách (hloubce) implementujeme iteračně pro dílčí hloubky stromu

```
void tree_print_layers(const node_t *const node)
{
    const int depth = tree_max_depth(node);
    for (int i = 0; i <= depth; ++i) {
        printDepth(i, 0, node);
        printf("\n");
    }
}
```

lec10/tree-int.c



## Příklad použití `tree_print_layers()`

- Přidáme výpis a volání `tree_print_layers()`

```
...
printf("Print tree by depth row\n");
tree_print_layers(root);
...
```

- Program spustíme bez a s argumentem `balanced`

<pre>clang tree-int.c demo-tree-int.c ./a.out Insert values2 that will result     in none balanced tree ... Print tree: 2 3 4 5 6 7 8 Tree is binary search tree (BST):     yes Max tree depth: 4 Print tree by depth row 5 4 6 3 - - 7 2 - - 8 - - - -</pre>	<pre>clang tree-int.c demo-tree-int.c ./a.out balanced Insert values1 to make balanced     tree ... Print tree: 2 3 4 5 6 7 8 Tree is binary search tree (BST):     yes Max tree depth: 3 Print tree by depth row 5 3 7 2 4 6 8 - - - - - - - -</pre>
---	---

`lec10/demo-tree-int.c`



# Obsah

Stromy

Binární strom

Příklad binárního stromu v C

Stromové struktury



# Stromové struktury

- Stromové struktury jsou významné datové struktury pro vyhledávání *Složitost vyhledávání je úměrná hloubce stromu.*
- Binární stromy – každý uzel má nejvýše dva následníky
  - Hloubku stromu lze snížit tzv. vyvažováním stromu
    - AVL stromy *Georgy Adelson-Velsky a Landis*
    - Red-Black stromy
  - **Plný binární strom** – každý vnitřní uzel má dva potomky a všechny uzly jsou co nejvíce vlevo
    - **Můžeme efektivně reprezentovat polem** *Pro daný maximální počet uzlů*
    - Lze použít pro efektivní implementaci prioritní fronty *Heap – halda*
    - Halda (heap) je základem řadícího algoritmu *Heap Sort*
- Vícecestné stromy – např. B–strom (Bayer tree) pro ukládání uspořádaných záznamů

*Informativní více v Algoritmizaci*



# Stromové struktury

- Stromové struktury jsou významné datové struktury pro vyhledávání  
*Složitost vyhledávání je úměrná hloubce stromu.*
- Binární stromy – každý uzel má nejvýše dva následníky
  - Hloubku stromu lze snížit tzv. vyvažováním stromu
    - AVL stromy *Georgy Adelson-Velsky a Landis*
    - Red-Black stromy
  - **Plný binární strom** – každý vnitřní uzel má dva potomky a všechny uzly jsou co nejvíce vlevo
    - **Můžeme efektivně reprezentovat polem**  
*Pro daný maximální počet uzlů*
    - Lze použít pro efektivní implementaci prioritní fronty  
*Heap – halda*
    - Halda (heap) je základem řadícího algoritmu *Heap Sort*
- Vícecestné stromy – např. B–strom (Bayer tree) pro ukládání uspořádaných záznamů

*Informativní více v Algoritmizaci*



# Část II

## Část 2 – Příklad načítání grafu, kompilace a projekt s více soubory





## Dílčí příklady použití jazykových konstrukcí v projektu

- Program složený z více souborů
- Dynamická alokace paměti
- Načítání souboru
- Parsování čísel z textového souboru
  
- Měření času běhu programu
- Řízení kompilace projektu složeného z více souborů `Makefile`



## Zadání

- Vytvořte program, který načte orientovaný graf definovaný posloupností hran
  - Graf je zapsán v textovém souboru
- Navrhněte datovou strukturu pro reprezentaci grafu
- Počet hran není dopředu znám

*Zpravidla však budou na vstupu grafy s průměrným počtem hran  $3n$  pro  $n$  vrcholů grafu.*

- Hrana je definována číslem vstupního a výstupního vrcholu a cenou (také celé číslo)
  - Ve vstupním souboru je každá hrana zapsaná samostatně na jednom řádku
  - Řádek má tvar:  
`from to cost`
  - kde `from`, `to` a `cost` jsou kladná celá čísla v rozsahu `int`
- Pro načtení hodnot hran použijte pro zjednodušení funkci `fscanf()`
- *Program dále rozšířte o sofistikovanější, méně výpočetně náročné načítání*



# Pravidla překladačů v gmake / make

- Pro řízení překladačů použijeme pravidlový předpis programu GNU `make` make nebo gmake

- Pravidla se zapisují do souboru `Makefile`  
<http://www.gnu.org/software/make/make.html>

- Pravidla jsou deklarativní ve tvaru definice cíle, závislostí cíle a akce, která se má provést

**cíl : závislosti** *dvojtečka*  
**akce** *tabulátor*

- Cíl (podobně jako závislosti) může být například symbolické jméno nebo jméno souboru

**tload.o : tload.c**  
**clang -c tload.c -o tload.o**

- Předpis může být napsán velmi jednoduše

*Například jako v uvedené ukázce.*

Flexibilita použití však spočívá především v použití zavedených proměnných, vnitřních proměnných a využití vzorů, neboť většina zdrojových souborů se překládá identicky.



## Příklad – Makefile

- Definujeme pravidlo pro vytvoření souborů `.o` z `.c`
- Definice přeložených souborů vychází z aktuálních souborů v pracovním adresáři s koncovkou `.c`

```
CC:=ccache $(CC)
CFLAGS+=-O2

OBJS=$(patsubst %.c,%.o,$(wildcard *.c))

TARGET=tload

bin: $(TARGET)

$(OBJS): %.o: %.c
    $(CC) -c $< $(CFLAGS) $(CPPFLAGS) -o $@

$(TARGET): $(OBJS)
    $(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
    $(RM) $(OBJS) $(TARGET)
ccache
```

*CC=clang make vs CC=gcc make*

- **Při linkování záleží na pořadí souborů (knihoven)!**
- Jednou z výhod dobrých pravidel je možnost paralelního překladač nezavislých cílů

`make -j 4`



## Definice datové struktury grafu – graph.h

- Zavedeme nový typ datové struktury hrana—`edge_t`,
- který použijeme ve struktuře grafu—`graph_t`

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

typedef struct {
    int from;
    int to;
    int cost;
} edge_t;

typedef struct {
    edge_t *edges;
    int length;
    int size;
} graph_t;

#endif
```

- Soubor budeme opakovaně vkládat (`include`) v ostatních zdrojových souborech, proto „zabraňujeme“ opakovaně definici konstantou preprocesoru `__GRAPH_H__`



## Pomocné funkce pro práci s grafem

- Alokaci/uvolnění grafu implementujeme v samostatných funkcích
- Při načítání grafu budeme potřebovat postupně zvyšovat paměť pro uložení načítaných hran
- Proto využijeme dynamické alokace paměti pro „nafukování“ paměti pro uložení hran grafu—`enlarge_graph()` o nějakou definovanou velikost

```
#ifndef __GRAPH_UTILS_H__
#define __GRAPH_UTILS_H__

#include "graph.h"

graph_t* allocate_graph(void);

void free_graph(graph_t **g);

graph_t* enlarge_graph(graph_t *g);

void print_graph(graph_t *g);

#endif
```



## Alokace paměti pro uložení grafu

- Testujeme úspěšnost alokace paměti—`assert()`
- Po alokaci nastavíme hodnoty proměnných na `NULL` a `0`

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "graph.h"

graph_t* allocate_graph(void)
{
    graph_t *g = (graph_t*) malloc(sizeof(graph_t));
    assert(g != NULL);
    g->edges = NULL;
    g->length = 0;
    g->size = 0;
    /* or we can call calloc */
    return g;
}
```

- Alternativně můžeme použít funkci `calloc()`



## Uvolnění paměti pro uložení grafu

- Testujeme validní hodnotu argumentu funkce—`assert()`

*Pokud se stane chyba, tak funkci v programu špatně voláme.*

*Až program odladíme můžeme kompilovat s `NDEBUG`.*

```
void free_graph(graph_t **g)
{
    assert(g != NULL && *g != NULL);
    if ((*g)->size > 0) {
        free((*g)->edges);
    }
    free(*g);
    *g = NULL;
}
```

- Po uvolnění paměti nastavíme hodnotu ukazatele na strukturu na hodnotou `NULL`





## Zvětšení paměti pro uložení hran grafu

- V případě nulové velikosti alokujeme paměť pro `NSIZE` hran
- `NSIZE` můžeme definovat při překladač, jinak výchozí hodnota 10

např. `clang -D NSIZE=100 -c graph_utils.c`

```
#ifndef NSIZE
#define NSIZE 10
#endif

graph_t* enlarge_graph(graph_t *g)
{
    assert(g != NULL);
    int n = g->size == 0 ? NSIZE : g->size * 2;
    /* double the memory */
    edge_t *e = (edge_t*)malloc(n * sizeof(edge_t));
    memcpy(e, g->edges, g->length * sizeof(edge_t));
    free(g->edges);
    g->edges = e;
    g->size = n;
    return g;
}
```

- Místo relativně komplexní alokace nového bloku paměti a kopírování původního obsahu můžeme jednoduše použít funkci `realloc()`



## Tisk hran grafu

- Pro tisk hran grafu využijeme pointerovou aritmetiku

```
void print_graph(graph_t *g)
{
    assert(g != NULL);
    fprintf(stderr, "Graph has %d edges and %d edges are
        allocated\n", g->length, g->size);
    edge_t *e = g->edges;
    for(int i = 0; i < g->length; ++i, e++) {
        printf("%d %d %d\n", e->from, e->to, e->cost);
    }
}
```

- Informace vypisujeme na standardní chybový výstup
- Graf tiskneme na standardní výstup
- Při tisku a přesměrování standardního výstupu tak v podstatě můžeme realizovat kopírování souboru s grafem

Např. `./tload -p g > g2`



## Hlavní funkce programu – main()

- V hlavní funkci zpracujeme předané argumenty programu
- V případě uvedení přepínače `-p` vytiskneme graf na `stdout`

```
int main(int argc, char *argv[])
{
    int ret = 0;
    int print = 0;
    char *fname;
    int c = 1;
    if (argc > 2 && strcmp(argv[c], "-p") == 0) {
        print = 1;
        c += 1;
    }
    fname = argc > 1 ? argv[c] : NULL;
    fprintf(stderr, "Load file '%s'\n", fname);
    graph_t *graph = allocate_graph();
    int e = load_graph_simple(fname, graph);
    fprintf(stderr, "Load %d edges\n", e);
    if (print) {
        print_graph(graph);
    }
    free_graph(&graph);
    return ret;
}
```



## Jednoduché načtení grafu – deklarace

- Prototyp funkce uvedeme v hlavičkovém souboru `load_simple.h`

```
#ifndef __LOAD_SIMPLE_H__
#define __LOAD_SIMPLE_H__

#include "graph.h"

int load_graph_simple(const char *fname, graph_t *g);

#endif
```

- Vkládáme pouze soubor `graph.h`—pro definici typu `graph_t`

*Snažíme se zbytečně nevkładat nepoužívané soubory*



## Jednoduché načtení grafu – implementace 1/2

- Používáme funkci `enlarge_graph()`, proto vkládáme `graph_utils.h`

```
#include <stdio.h>
#include "graph_utils.h"
int load_graph_simple(const char *fname, graph_t *g)
{
    int c = 0;
    int exit = 0;
    FILE *f = fopen(fname, "r");
    while(!feof(f) && !exit) {
        if (g->length == g->size) {
            enlarge_graph(g);
        }
        edge_t *e = g->edges + g->length;
        while(!feof(f) && g->length < g->size) {
            /* read and parse a single line -> NEXT SLIDE! */
        }
    }
    fclose(f);
    return c;
}
```

- `load_simple.h` vkládat nemusíme, obsahuje pouze prototyp funkce
- Obecně je to však dobrý zvykem nebo nutností (definice typů)



## Jednoduché načtení grafu – implementace 2/2

- Pro načtení řádku s definicí hrany použijeme funkci `fscanf()`

```
while(!feof(f) && g->length < g->size) {
    int r = fscanf(f, "%d %d %d\n",
                 &(e->from), &(e->to), &(e->cost));

    if (r == 3) {
        g->length += 1;
        c += 1; /* pocet nactenych hran */
        e += 1; /* posun ukazatele grafu o sizeof(edge_t)*/
    } else {
        exit = 1; /* neco je spatne ukoncujieme naciteni */
        break;
    }
}
```

- Kontrolujeme počet přečtených parametrů a až pak zvyšujeme počet hran v grafu



## Spuštění programu 1/3

- Necht' máme soubor `g` definující graf o 1 000 000 uzlech

*Velikost souboru cca 62 MB (příkaz `du`—disk usage)*

```
% du g
62M  g
```

```
% ./tload g
Load file 'g'
Load 2998898 edges
```

```
% time ./tload g
Load file 'g'
Load 2998898 edges
./tload g  1.12s user 0.03s system 99% cpu 1.151 total
```

- Příkazem `time` můžeme změřit potřebný čas běhu programu

*strojový, systémový a reálný*



## Spuštění programu 2/3

- Příznakem `-p` a přesměrováním standardního výstupu můžeme vytisknout graph do souboru

*V podstatě vstupní soubor zkopírujeme.*

```
% time ./tload -p g > g2
Load file 'g'
Load 2998898 edges
Graph has 2998898 edges and 5242880 edges are allocated
./tload -p g > g2  2.09s user 0.07s system 99% cpu 2.158
  total
```

```
% md5 g g2
MD5 (g) = d969461a457e086bc8ae08b5e9cce097
MD5 (g2) = d969461a457e086bc8ae08b5e9cce097
```

- Čas běhu programu je přibližně dvojnásobný
- Oba soubory se zdají být z otisku `md5` identické

*Na Linuxu `md5sum` případně lze použít otisk `sha1`, `sha256` nebo `sha512`*





## Spuštění programu 3/3

- Implementací sofistikovanějšího načítání

```
% /usr/bin/time ./tload g
Load file 'g'
Load 2998898 edges
0.19 real          0.16 user          0.03 sys
```

- lze získat výrazně rychlejší načítání

*160 ms vs 1100 s*

```
% /usr/bin/time ./tload g
Load file 'g'
Load 2998898 edges
1.15 real          1.05 user          0.10 sys
```



## Jak a za jakou cenu zrychlit načítání seznamu hran

- Zrychlit načítání můžeme přijmutím předpokladů o vstupu
- Při použití `fscanf()` je nejdříve načítán řetězec (řádek) pak řetěz reprezentující číslo a následně je parsováno číslo
- Převod na číslo je napsán obecně
- Můžeme použít postupné „bufferované“ načítání
- Převod na číslo můžeme realizovat přímo po přečtení tokenu
- parsováním znaků (číslic) načtené posloupnosti bytů v obráceném pořadí
  
- Můžeme získat výrazně rychlejší kód, který je však komplexnější a pravděpodobně méně obecný



# Část III

## Část 2 – Zadání 9. domácího úkolu (HW09)



## Zadání 9. domácího úkolu HW09



- Termín odevzdání: 24.12.2016, 23:59:59 AoE

*AoE – Anywhere on Earth*

*Prodloužený termín (náročná úloha)!*



# Shrnutí přednášky



## Diskutovaná témata

- Stromy – nelineární spojivé struktury
- Binární vyhledávací strom
- Vyhledání prvku a průchod stromem (rekurzí)
- Rekurzivní uvolnění paměti alokované stromem
- Test splnění vlastnosti binárního vyhledávacího stromu
- Hloubka stromu a výpis stromu po úrovních
- Příklad jednoduchého binárního vyhledávacího stromu s položkami typu `int` lec10/tree
- Plný binární strom a jeho reprezentace
- Makefile
- Příklad načtení stromu jako seznamu hran lec10/graph\_load
- Příště abstraktní datový typ (ADT)



## Diskutovaná témata

- Stromy – nelineární spojivé struktury
- Binární vyhledávací strom
- Vyhledání prvku a průchod stromem (rekurzí)
- Rekurzivní uvolnění paměti alokované stromem
- Test splnění vlastnosti binárního vyhledávacího stromu
- Hloubka stromu a výpis stromu po úrovních
- Příklad jednoduchého binárního vyhledávacího stromu s položkami typu `int` lec10/tree
- Plný binární strom a jeho reprezentace
- Makefile
- Příklad načtení stromu jako seznamu hran lec10/graph\_load
- **Příště abstraktní datový typ (ADT)**

