Stromy

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 10

B0B36PRP – Procedurální programování

Přehled témat

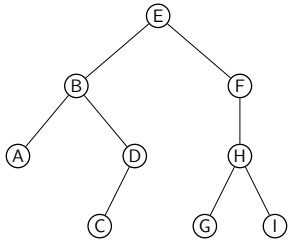
Část I

Část 1 – Stromy

Lineární a nelineární spojové struktury

- Spojové seznamy představují lineární spojovou strukturu.
- Nelineární spojové struktury (např. stromy).
- **Binární strom**: každý prvek (uzel) má nejvýše dva následníky.

Každý prvek má nejvýše jednoho následníka.
Každý prvek může mít více následníků.



- **kořen stromu**
- **list**
- **levý podstrom**
- **pravý podstrom**

Binární strom

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

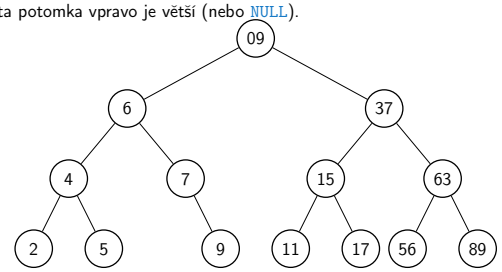
```
1 typedef struct node {
2    int value;
3    struct node *left;
4    struct node *right;
5 } node_t;
```

- 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()`.

```
8  static node_t* newNode(int value)
9  {
10    node_t *node= (node_t*)malloc(sizeof(node_t));

12    if (!node) {
13       fprintf(stderr, "ERROR: Memory allocation fail file: %s line: %d\n",
       __FILE__, __LINE__);
14       exit(-1);
15    }

17    node->value = value;
18    node->left = node->right = NULL;
19    return node;
20 }
```

lec10/tree/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!

```
23 node_t* tree_insert(int value, node_t *node)
24 {
25    if (node == NULL) {
26       return newNode(value); // vracíme nový uzel
27    } else {
28       if (value <= node->value) { //vložení do levého podstromu
29          node->left = tree_insert(value, node->left);
30       } else { // vložení do pravého podstromu
31          node->right = tree_insert(value, node->right);
32       }
33       return node; // vracíme vstupní uzel!!!
34    }
35 }
```

lec10/tree/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 můžeme postupovat 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 jsou vlevo prvky menší nebo větší).

Binární strom celočíselných hodnot int

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

Viz předchozí příklad.

implementuje 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).

```
1  void tree_free(node_t **tree); // chceme také smazat a vynulovat
       hodnotu ukazatele tree (na úrovni volající funkce), proto **tree
2  int tree_size(const node_t *const tree);
3  void tree_print(const node_t *const node);
```

lec10/tree/tree-int.h

Příklad implementace tree_free()

```
40 void tree_free(node_t **tree)
41 {
42    if (tree && *tree) {
43       node_t * node = *tree;
44       if ( node->left ) {
45          tree_free(&(node->left));
46       }
47       if ( node->right ) {
48          tree_free(&(node->right));
49       }
50       free(*tree);
51       *tree = NULL; // fill the tree variable
52                     // of the calling function to NULL
53    }
54 }
```

Předáváme ukazatel na ukazatel, abychom mohli po uvolnění paměti nastavit hodnotu ukazatele (ve volající funkci) na `NULL`. *Proměnná je předána hodnotou.*

```
...
tree_free(&tree);
// zde chceme mit
    tree == NULL
```

lec10/tree/tree-int.h

Příklad implementace tree_size() a tree_print()

- Určení počtu prvků implementujeme rekurzí.

```
56 int tree_size(const node_t *const node)
57 {
58    return node == NULL ?
59       0 :
60       tree_size(node->left) + 1 + tree_size(node->right);
61 }
```

- Podobně výpis hodnot.

```
74 void tree_print(const node_t *const node)
75 {
76    if (node) {
77       tree_print(node->left);
78       printf("%d ", node->value);
79       tree_print(node->right);
80    }
81 }
```

lec10/tree/tree-int.c

Příklad použití – 1/3

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

```
22    ...
23    for (int i = 0; i < n; ++i) {
24       printf("Insert value %i\n", values[i]);
25       if (root == NULL) {
26          root = tree_insert(values[i], NULL);
27       } else {
28          tree_insert(values[i], root);
29       }
30    }
31    printf("No. of tree nodes is %i\n", tree_size(root));

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

37    tree_free(&root);
38    printf("After tree_free() root is %p\n", root);
39    return 0;
40 }
```

lec10/tree/demo-tree-int.c

Příklad použití – 3/3

- Program spustíme bez a s argumentem pro načtení „balanced“ stromu.

```
$ clang tree-int.c demo-tree-int.c
$ ./a.out
Insert values2 that will result in an
    unbalanced 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
```

```
$ clang tree-int.c demo-tree-int.c
$ ./a.out values1

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
```

lec10/tree/demo-tree-int.c

- V obou případech je výpis uspořádaný.

Jak otestovat, že operace nad stromem (např. `tree_insert()`) zachová vlastnosti BST?

Test vlastnosti binárního vyhledávacího stromu

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

```
1  _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.

```
84  static int getMaxValue(const node_t *const node)
85  {
86     const node_t *cur = node;
87     while (cur->right) {
88        cur = cur->right;
89     }
90     return cur->value;
91  }
```

- Podobně minimální hodnotu.

```
94  static int getMinValue(const node_t *const node)
95  {
96     const node_t *cur = node;
97     while (cur->left) {
98        cur = cur->left;
99     }
100    return cur->value;
101 }
```

lec10/tree/tree-int.c

Příklad implementace tree_is_bst() - 2/3

```
105 _Bool tree_is_bst(const node_t *const node)
106 {
107    _Bool ret = true;
108    if (node != NULL) {
109       if (node->left
110             && getMaxValue(node->left) > node->value) {
111          ret = false;
112       }
113       if (ret && node->right
114             && getMinValue(node->right) <= node->value) {
115          ret = false;
116       }
117       if (ret
118             && (
119                !tree_is_bst(node->left) || !tree_is_bst(node->right)
120                )
121          ) {
122          ret = false;
123       }
124    }
125    return ret;
126 }
```

lec10/tree/tree-int.c

Příklad implementace tree_is_bst() - 3/3

- Přidáme výpis a volání `tree_is_bst()`.

```
36    ...
37    printf("Max tree depth: %i\n", tree_max_depth(root));
38    printf("Tree is binary seach tree (BST): %s\n",
39           tree_is_bst(root) ? "yes" : "no");
```

- Program spustíme bez a s argumentem pro načtení `balanced` stromu.

```
$ clang tree-int.c demo-tree-int.c
$ ./a.out
Insert values2 that will result in an unbalanced
    tree
...
Print tree: 2 3 4 5 6 7 8
Tree is binary seach tree (BST): yes
Print tree by depth row
```

```
$ clang tree-int.c demo-tree-int.c
$ ./a.out values1
Insert values1 to make balanced tree
...
Print tree: 2 3 4 5 6 7 8
Tree is binary seach tree (BST): yes
```

- V obou případech je podmínka BST splněna.

lec10/tree/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.

```
62 int tree_max_depth(const node_t *const node)
63 {
64    if (node) {
65       const int left_depth = tree_max_depth(node->left);
66       const int right_depth = tree_max_depth(node->right);
67       return left_depth > right_depth ?
68          left_depth + 1 :
69          right_depth + 1;
70    } else {
71       return 0;
72    }
73 }
```

lec10/tree/tree-int.c

Výpis hodnot v konkrétní hloubce stromu `printDepth()`

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

```
128 static void printDepth(int depth, int cur_depth, const node_t *const node)
129 {
130    if (depth == cur_depth) {
131       if (node) {
132          printf("%2d ", node->value);
133       } else {
134          printf(" - ");
135       }
136    } else if (node) {
137       printDepth(depth, cur_depth + 1, node->left);
138       printDepth(depth, cur_depth + 1, node->right);
139    }
140 }
```

lec10/tree/tree-int.c

Příklad implementace výpisu stromu `tree_print_layers()`

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

```
142 void tree_print_layers(const node_t *const node)
143 {
144    const int depth = tree_max_depth(node);
145    for (int i = 0; i <= depth; ++i) {
146       printDepth(i, 0, node);
147       printf("\n");
148    }
149 }
```

lec10/tree/tree-int.c

Příklad použití `tree_print_layers()`

- Přidáme výpis a volání `tree_print_layers()`.

```
40 ...
41 printf("Print tree by depth row\n");
42 tree_print_layers(root);
43 ...
```

- Program spustíme bez a s argumentem pro načtení *balanced* stromu.

```
1  clang tree-int.c demo-tree-int.c
2  ./a.out
3  Insert values2 that will result in an
       unbalanced tree
4  ...
5  Print tree: 2 3 4 5 6 7 8
6  Tree is binary seach tree (BST): yes
7  Max tree depth: 4
8  Print tree by depth row
9   5
10  4  6
11  3  -  -  7
12  2  -  -  8
13  -  -  -  -
```

```
1  clang tree-int.c demo-tree-int.c
2  ./a.out values1
3  Insert values1 to make balanced tree
4  ...
5  ...
6  Print tree: 2 3 4 5 6 7 8
7  Tree is binary seach tree (BST): yes
8  Max tree depth: 3
9  Print tree by depth row
10  5
11  3  7
12  2  4  6  8
13  -  -  -  -  -  -  -  -
```

lec10/tree/demo-tree-int.c

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 **A**delson-**V**elsky a Evegenii **L**andis*
    - 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, viz přednáška 11.
    - 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 rozšiřte o sofistikovanější, méně výpočetně náročné načítání.*

Pravidla překladu v `gmake` / `make`

- Pro řízení překladu 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 spočívám 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** 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 překladu nezávislých 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 num_edges;
   int capacity;
} graph_t;

#endif
```

lec10/graph_load/graph.h

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

Pomocné funkce pro práci s grafem

- Alokaci/uvolnění grafu implementujeme v samostatných funkcích.
- Při načítání grafu budeme postupně zvětšovat paměť pro uložení načítaných hran.
- Využijeme dynamické alokace paměti—enlarge_graph() o 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
```

lec10/graph_load/graph_utils.h

Alokace paměti grafu

- Testujeme úspěšnost alokace paměti.
- Po alokaci nastavíme hodnoty proměnných na NULL a 0.

Alternativně calloc().

```
#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));
   if (g == NULL) {
      fprintf(stderr, "Malloc fail: %s line %d\n", __FILE__, __LINE__);
      exit(-1);
   }
   g->edges = NULL;
   g->num_edges = 0;
   g->capacity = 0; /* or we can call calloc */
   return g;
}
```

lec10/graph_load/graph_utils.c

Uvolnění paměti pro uložení grafu

- Testujeme validní hodnotu argumentu funkce—assert().

Pokud nastane chyba, funkci v programu špatně voláme. Až odladíme můžeme kompilovat s NDEBUG.

```
void free_graph(graph_t **g)
{
   /* We request to call free_graph only with valid g.
    * The program has to be written to properly call free_graph(). */
   assert(g != NULL && *g != NULL);
   if ((*g)->capacity > 0) {
      free((*g)->edges);
   }
   free(*g);
   *g = NULL;
}
```

lec10/graph_load/graph_utils.c

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

Zvětšení paměti pro uložení hran grafu

- V případě nulové velikosti alokujme paměť pro INIT_SIZE hran.
- INIT_SIZE můžeme definovat při překladu, jinak výchozí hodnota 10.

např. clang -D INIT_SIZE=90 -c graph_utils.c

```
#ifndef INIT_SIZE
#define INIT_SIZE 10
#endif

graph_t* enlarge_graph(graph_t *g)
{
   assert(g != NULL); /* enlarge_graph() must be properly called */
   int n = g->capacity == 0 ? INIT_SIZE : g->capacity * 2;
   /* double the memory */
   edge_t *e = (edge_t*)malloc(n * sizeof(edge_t));
   if (e == NULL) {
      fprintf(stderr, "Malloc fail: %s line %d\n", __FILE__, __LINE__);
      exit(-1);
   }
   memcpy(e, g->edges, g->num_edges * sizeof(edge_t));
   free(g->edges);
   g->edges = e; /* update edges */
   g->capacity = n;
   return g;
}
```

lec10/load_graph/graph_utils.c

- Místo malloc() a memcpy() můžeme 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->num_edges, g->capacity);
   edge_t *e = g->edges;
   for (int i = 0; i < g->num_edges; ++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 main() 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;
}
```

lec10/graph_load/tgraph.c

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 nevkládat nepoužívané soubory.

Jednoduché načtení grafu – implementace 1/2

- Používáme funkci `enlarge_graph()`, proto vkládáme `graph_utils.h`.
- `load_simple.h` vkládat nemusíme, obsahuje pouze prototyp funkce.
- Obecně je to dobrým zvykem.
- Nutností v případě definice typů.

```c
#include <stdio.h>
#include "graph_utils.h"
#include "load_simple.h"

int load_graph_simple(const char *fname, graph_t *g)
{
   int c = 0;
   int exit = 0;
   FILE *f = fopen(fname, "r");
   while (f && !exit) {
      if (g->num_edges == g->capacity) {
         enlarge_graph(g);
      }
      edge_t *e = g->edges + g->num_edges;
      while (g->num_edges < g->capacity) {
       /* read and parse a single line -> NEXT SLIDE! */
      }
   }
   if (f) {
      fclose(f);
   }
```

lec10/graph_load/load_simple.c

Jednoduché načtení grafu – implementace 2/2

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

```c
   while (g->num_edges < g->capacity) {
      int r = fscanf(f, "%d %d %d\n", &(e->from), &(e->to), &(e->cost));
      if (r == 3) {
         g->num_edges += 1;
         c += 1; /* pocet nactenych hran */
         e += 1; /* posun ukazatele hran o sizeof(edge_t) */
      } else {
         exit = 1; /* neco je spatne ukoncujeme naciteni */
         break;
      }
   }
```

lec10/graph_load/load_simple.c

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

Spuštění programu 1/3

- Nechť máme soubor `g` definující graf o 1 000 000 uzlech, například vytvořený programem `lec10/graph_creator/graph_creator.c`.

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 1050 ms

```
$ /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 použití 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. Vlastní načítání bude méně obecné než `fscanf()`.

Část III

Část 2 – Zadání 9. domácího úkolu (HW09)

Zadání 9. domácího úkolu HW09

Téma: **Načítání a ukládání grafu**

Povinné zadání: 3b; Volitelné zadání: 2b; Bonusové zadání: není

- **Motivace**: Práce se soubory a binární reprezentace dat.
- **Cíl**: Osvojit si načítání a ukládání souborů a prohloubit si zkušenosti s dynamickým alokováním paměti.
- **Zadání**: `https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw09`
  - Implementace načítání a ukládání datových struktur reprezentující graf a to jak v lidsky čitelné podobě textového souboru, tak v efektivní binární formátu.
  - Volitelné zadání je zaměřeno na využití definovaného textového formátu s cílem vytvořit specifickou efektivní implementaci textového načítání/ukládání z/do textového souboru.

    Na úkor obecnosti, lze vytvořit specifický „parser/printer" a vyhnout se tak použití obecné funkce `fscanf()`/`fprintf()` a realizovat výrazně rychlejší načítání a zápis textového souboru.
- Termín odevzdání: 21.12.2024, 23:59:59 PST.

PST – Pacific Standard Time

Shrnutí přednášky

Diskutovaná témata

- Stromy – nelineární spojové 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: Prioritní fronta – polem a haldou. Příklad využití prioritní fronty (haldy) v úloze hledání nejkratší cesty v grafu.