Prioritní fronta a příklad použití v úloze hledání nejkratších cest

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

B0B36PRP – Procedurální programování

Přehled témat

Část I

Část 1 – Prioritní fronta (Halda)

Prioritní fronta polem – rozhraní

- V případě implementace prioritní fronty polem můžeme využít jedno pole pro hodnoty a druhé pole pro uložení priority daného prvku.

Implementace vychází z `lec11/queue_array.h` *a* `lec11/queue_array.c`

```c
typedef struct {
   void **queue;    // Pole ukazatelů na jednotlivé prvky
   int *priorities; // Pole hodnot priorit jednotlivých prvků
   int count;       // Uvažujeme pouze MAX_INT prvků, zpravidla 2147483647
   int head;
   int tail;
} queue_t;
```

- Další rozhraní (jména a argumenty funkcí) mohou zůstat identické jako u implementace spojovým seznamem.

Viz 9. přednáška.

```c
void queue_init(queue_t **queue);
void queue_delete(queue_t **queue);
void queue_free(queue_t *queue);

_Bool queue_is_empty(const queue_t *queue);
```

```c
int queue_push(void *value, int priority,
      queue_t *queue);
void* queue_pop(queue_t *queue);
void* queue_peek(const queue_t *queue);
```

Prioritní fronta polem 1/3 – push()

- Funkce push() je až na uložení priority identická s verzí bez priorit.

```
int queue_push(void *value, int priority, queue_t *queue)
{
   int ret = QUEUE_OK; // by default we assume push will be OK
   if (queue->count < MAX_QUEUE_SIZE) {
      queue->queue[queue->tail] = value;

      // store priority of the new value entry
      queue->priorities[queue->tail] = priority;

      queue->tail = (queue->tail + 1) % MAX_QUEUE_SIZE;
      queue->count += 1;
   } else {
      ret = QUEUE_MEMFAIL;
   }
   return ret;
}
```

lec11/priority_queue-array/priority_queue-array.c

- Funkce peek() a pop() potřebují prvek s nejnižší (nejvyšší) prioritou.
  - Nalezení prvku z „čela“ fronty realizujeme funkcí getEntry(), kterou následně využijeme jak v peek(), tak v pop().

Prioritní fronta polem 2/3 – getEntry()

- Nalezení nejmenšího (největšího) prvku provedeme lineárním prohledáním aktuálních prvků uložených ve frontě (poli).

```
static int getEntry(const queue_t *const queue)
{
  int ret = -1; // return -1 if queue is empty.
  if (queue->count > 0) {
    for (int cur = queue->head, i = 0; i < queue->count; ++i) {
      if (
          ret == -1 ||
          (queue->priorities[ret] > queue->priorities[cur])
        ) {
        ret = cur;
      }
      cur = (cur + 1) % MAX_QUEUE_SIZE;
    }
  }
  return ret;
}
```

lec11/priority_queue-array/priority_queue-array.c

Prioritní fronta polem 3/3 – peek() a pop()

- Funkce peek() využívá lokální (static) funkce getEntry().

Tím zajistíme, že prvky tvoří souvislý blok v rámci kruhové fronty.

```
void* queue_peek(const queue_t *queue)
{
  return queue_is_empty(queue) ? NULL : queue->queue[getEntry(queue)];
}
```

- Ve funkci pop() zaplníme položku vyjmutého prvku prvkem ze startu.

```
void* queue_pop(queue_t *queue)
{
   void *ret = NULL;
   int bestEntry = getEntry(queue);
   if (bestEntry >= 0) { // entry has been found
      ret = queue->queue[bestEntry];
      if (bestEntry != queue->head) { //replace the bestEntry by head
         queue->queue[bestEntry] = queue->queue[queue->head];
         queue->priorities[bestEntry] = queue->priorities[queue->head];
      }
      queue->head = (queue->head + 1) % MAX_QUEUE_SIZE;
      queue->count -= 1;
   }
   return ret;
}
```


Prioritní fronta polem – příklad použití

- Použití je identické s implementací spojovým seznamem.

```
make && ./demo-priority_queue-array
ccache clang -c priority_queue-array.c -O2  -o priority_queue-array.o
ccache clang priority_queue-array.o demo-priority_queue-array.o  -o demo-
    priority_queue-array
Add  0 entry '2nd' with priority '2' to the queue
Add  1 entry '4th' with priority '4' to the queue
Add  2 entry '1st' with priority '1' to the queue
Add  3 entry '5th' with priority '5' to the queue
Add  4 entry '3rd' with priority '3' to the queue

Pop the entries from the queue
1st
2nd
3rd
4th
5th
```

lec11/priority_queue-array/priority_queue-array.h

lec11/priority_queue-array/priority_queue-array.c

lec11/priority_queue-array/demo-priority_queue-array.c

Prioritní fronta spojovým seznamem nebo polem a výpočetní náročnost

- V naivní implementaci prioritní fronty jsme zohlednění priority „odložili“ až do doby, kdy potřebujeme odebrat prvek z fronty. *Použili jsme „lazy“ (odložený) výpočet.*
- Při odebrání (nebo vrácení) nejmenšího prvku v nejnepříznivějším případě musíme projít všechny položky.
- To může být výpočetně náročné a raději bychom chtěli „udržovat“ prvek připravený.
  - Můžeme to například udělat zavedením položky **head**, ve které bude aktuálně nejnižší (nejvyšší) vložený prvek do fronty.
  - Prvek **head** aktualizujeme v metodě **push()** porovnáním hodnoty aktuálně vkládaného prvku.
  - Tím zefektivníme operaci **peek()**.
  - V případě odebrání prvku, však musíme frontu znovu projít a najít nový prvek.

Nebo můžeme použít sofistikovanější datovou strukturu, která nám umožní efektivně udržovat hodnotu nejmenšího prvku a to jak při operaci vložení **push()** tak při operaci vyjmutí **pop()** prvku z prioritní fronty.

Halda

- Halda je dynamická datová struktura, která má „tvar“ binárního stromu a uspořádání prioritní fronty.
- Každý prvek haldy obsahuje hodnotu a dva potomky, podobně jako binární strom.
- **Vlastnosti haldy** – „*Heap property*“.
  - Hodnota každého prvku je menší než hodnota libovolného potomka.
  - Každá úroveň binárního stromu haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava. **Binární plný strom**
  - Prvky mohou být odebrány pouze přes kořenový uzel.
- Vlastnost haldy zajišťuje, že kořen je vždy prvek s nejnižším/nejvyšším ohodnocením.

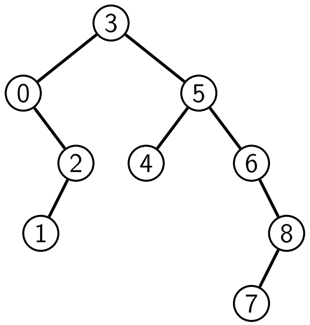
V případě binárního plného stromu je složitost procházení úměrná hloubce stromu, která je pro n prvků úměrná $\log_2(n)$. Složitost operací `push()`, `pop()`, `peek()` tak můžeme očekávat nikoliv $O(n)$ (jako v případě předchozí implementace prioritní fronty polem a spojovým seznamem), ale $O(\log n)$ a pro `peek()` dokonce $O(1)$.

Binární vyhledávací strom vs halda

Binární vyhledávací strom

- Může obsahovat prázdná místa.
- Hloubka stromu se může měnit.

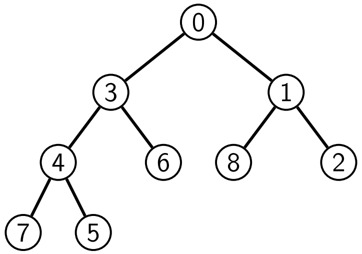
Zajistit vyvážený strom je implementačně náročnější než implementace haldy.



Halda

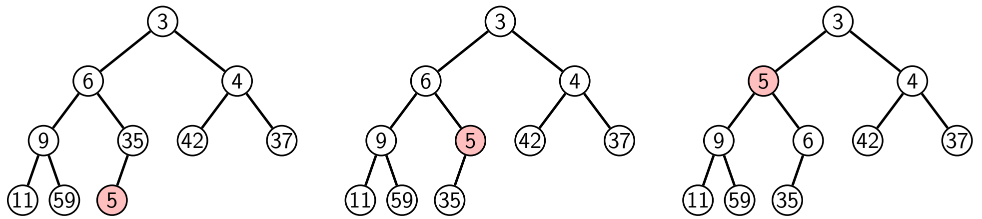
- Binární plný strom *Hloubka stromu vždy $\lfloor \log_2(n) \rfloor$.*
- Kořen stromu je vždy prvek s nejnižší (nejvyšší) hodnotou.
- Každý podstrom splňuje vlastnost haldy.

Heap property



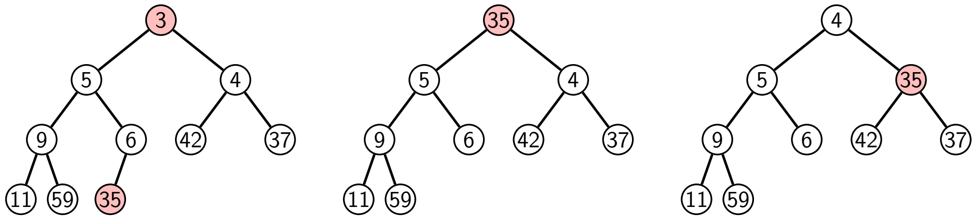
Halda – přidání prvku **push()**

1. Po každém provedení operace **push()** musí být splněny vlastnosti haldy.
2. Prvek přidáme na konec haldy, tj. na první volnou pozici (vlevo) na nejnižší úrovni haldy.
3. Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s nadřazeným prvkem (předkem). *V nejnepříznivějším případě prvek „probublá“ až do kořene stromu.*



Halda – odebrání prvku **pop()**

- Při operaci **pop()** odebereme kořen stromu.
- Prázdné místo nahradíme nejpravějším listem.
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s potomkem a postup opakujeme. *V nejnepříznivějším případě prvek „probublá" až do listu stromu.*



- Jak zjistit nejpravější list?
  - V případě implementace spojovou strukturou (nelineární) můžeme explicitně udržovat odkaz.
  - Binární plný strom můžeme efektivně reprezentovat polem – pak nejpravější list je poslední prvek v poli.

Prioritní fronta haldou

- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti **haldy**.
- Operace **peek()** má konstantní složitost a nezáleží na počtu prvků ve frontě, nejnižší prvek je vždy kořen.

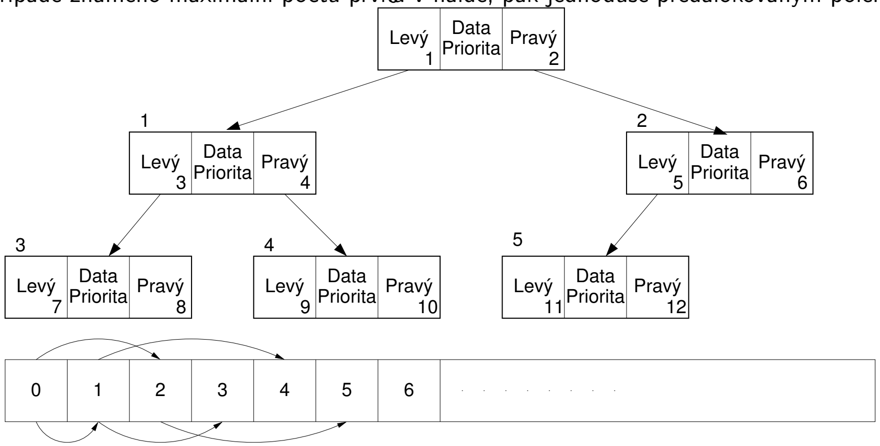
  Asymptotická složitost v notaci velké O je $O(1)$.

- Operace **push()** a **pop()** udržují vlastnost haldy záměnami prvku až do hloubky stromu.

  Pro binární plný strom je hloubka stromu $\log_2(n)$, kde n je aktuální počet prvků ve stromu, odtud složitost operace $O(\log(n))$.

Reprezentace binárního stromu polem

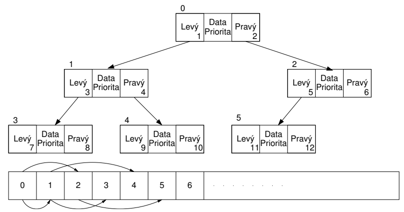
- Binární plný strom můžeme reprezentovat lineární strukturou.
- V případě známého maximální počtu prvků v haldě, pak jednoduše předalokovaným polem.



Halda jako binární plný strom reprezentovaný polem

- Pro definovaný maximální počet prvků v haldě si předalokujeme pole o daném počtu prvků.
- Binární **plný strom** má všechny vrcholy na úrovni rovné hloubce stromu co nejvíce vlevo.
- Kořen stromu je první prvek s indexem 0, následníky prvku na pozici i lze v poli určit jako prvky s indexy:
  - levý následník: $i_{levý} = 2i + 1$;
  - pravý následník: $i_{pravý} = 2i + 2$.

  Podobně lze odvodit vztah pro předchůdce.



- Kořen stromu reprezentuje nejprioritnější prvek.

  Např. s nejmenší hodnotu nebo maximální prioritou.

Operace vkládání a odebírání prvků

- I v případě reprezentace polem pracují operace vkládání a odebírání identicky.
  - Funkce `push()` přidá prvek jako další prvek v poli a následně propaguje prvek směrem nahoru až je splněna vlastnost haldy.
  - Při odebrání prvku funkcí `pop()` je poslední prvek v poli umístěn na začátek pole (kořen stromu) a propagován směrem dolů až je splněna vlastnost haldy.
- Dochází pouze k vzájemnému zaměňování hodnot na pozicích v poli (haldě).

  Z indexu prvku v poli vždy můžeme určit jak levého a pravého následníka, tak i předcházející prvek (rodič) v pohledu na haldu jako binární strom.
- Hlavní výhodou reprezentace polem je přístup do předem alokovaného bloku paměti.

  Všechny prvky můžeme jednoduše projít v jedné smyčce, například při výpisu.
- Ověření zdali implementace operací `push()` a `pop()` zachovává podmínku haldy můžeme realizovat ověřující funkcí `is_heap()`.

Příklad implementace pq_is_heap()

- Pro každý prvek haldy musí platit, že jeho hodnota je menší než levý i pravý následník.

```c
typedef struct {
   int size;    // the maximal number of entries
   int len;     // the current number of entries
   int *cost;   // array with costs - lowest cost is highest priority
   int *label;  // array with labels (each label has cost/priority)
} pq_heap_s;

_Bool pq_is_heap(pq_heap_s *pq, int n)
{
   _Bool ret = true;
   int l = 2 * n + 1; // left successor
   int r = l + 1;     // right successor
   if (l < pq->len) {
      ret = (pq->cost[l] < pq->cost[n]) ? false : pq_is_heap(pq, l);
   }
   if (r < pq->len) {
      ret = ret  // if ret is false, further expression is not evaluated
         &&
         ( (pq->cost[r] < pq->cost[n]) ? false : pq_is_heap(pq, r) );
   }
   return ret;
}
```


Příklad implementace push()

- Prvek přidáme na konec pole a iterativně kontrolujeme, zdali je splněna vlastnost haldy. Pokud ne, prvek zaměníme s předchůdcem.

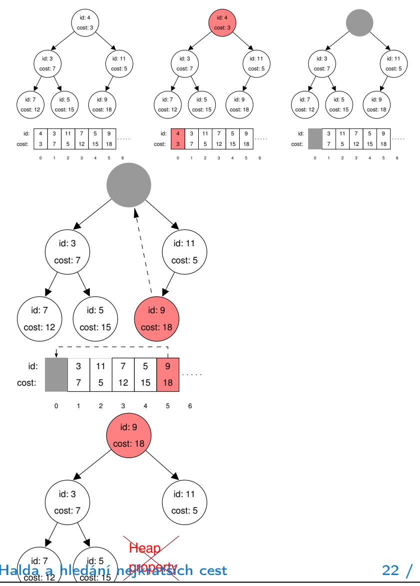
```c
#define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2

_Bool pq_push(pq_heap_s *pq, int label, int cost)
{
   _Bool ret = false;
   if (pq && pq->len < pq->size && label >= 0 && label < pq->size) {
      pq->cost[pq->len] = cost; //add the cost to the next free slot
      pq->label[pq->len] = label; //add label of new entry
      int cur = pq->len; // index of the entry added to the heap
      int parent = GET_PARENT(cur);
      while (cur >= 1 && pq->cost[parent] > pq->cost[cur]) {
         pq_swap(pq, parent, cur); // swap parent<->cur
         cur = parent;
         parent = GET_PARENT(cur);
      }
      pq->len += 1;
      ret = true;
   }
   // assert(pq_is_heap(pq, 0)); // testing the implementation
   return ret;
}
```


Příklad volání pop()

- Halda je reprezentovaná binárním polem.
- Nejmenší prvek je kořenem stromu.
- Voláním `pop()` odebíráme kořen stromu.
- Na jeho místo umístíme poslední prvek.
- Strom však nesplňuje podmínku haldy.
- Proto provedeme záměnu s následníky.

  V tomto případě volíme pravého následníka, neboť jeho hodnota je nižší než hodnota levého následníka.
- A strom opět splňuje vlastnost haldy.
- Záměny provádíme v poli a využíváme vlastnosti plného binárního stromu.

Levý potomek prvku haldy na pozici i je $2i+1$, pravý potomek je na pozici $2i+2$.

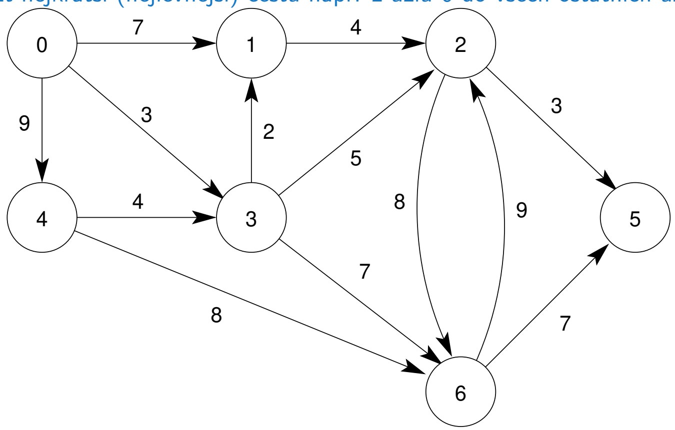


Část II

Část 2 – Příklad využití prioritní fronty v úloze hledání nejkratší cesty v grafu

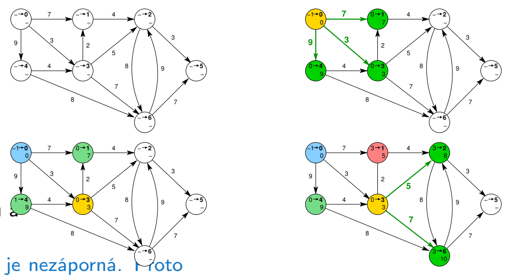
Hledání nejkratší cesty v grafu

- Uzly grafu mohou reprezentovat jednotlivá místa a hrany cestu, jak se mezi místy pohybovat.
- Ohodnocení (cena) hrany může odpovídat náročnosti pohybu mezi dvě sousedními uzly.
- Cílem je nalézt nejkratší (nejlevnější) cestu např. z uzlu 0 do všech ostatních uzlů.



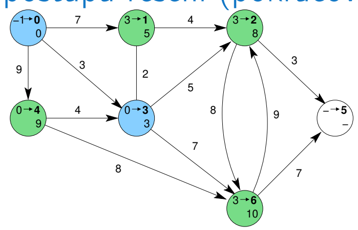
Dijkstrův algoritmus

- Nechť má graf pouze kladné ohodnocení hran, pak pro každý uzel:
  - nastavíme aktuální cenu nejkratší cesty z výchozího uzlu;
  - udržujeme odkaz na bezprostředního předchůdce na nejkratší cestě ze startovního uzlu.
- Hledání cesty je postupná aktualizace ceny nejkratší cesty do jednotlivých uzlů.
  - Začneme z výchozího uzlu (cena 0) a aktualizujeme délku cesty do následníků.
  - Následně vybereme takový uzel,
    - do kterého již existuje nějaká cesta z výchozího uzlu a zároveň má aktuálně nejnižší ohodnocení.
  - Postup opakujeme dokud existuje nějaký dosažitelný uzel.
    - Tj. uzel, do kterého vede cesta z výchozího uzlu a
    - má již ohodnocení a předchůdce (*zelené uzly*).

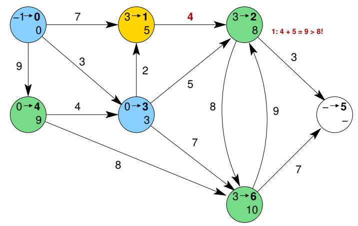


Ohodnocení uzlů se může pouze snižovat, cena hran je nezáporná. Proto pro uzel s aktuálně nejkratší cestou již nemůže existovat cesta kratší.

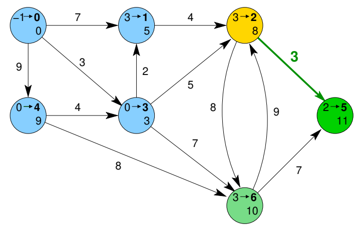
Příklad postupu řešení (pokračování)



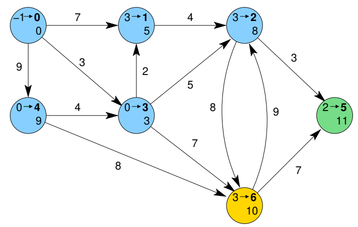
1: Po 2. expanzi má uzel 3 již nejkratší cestu.



2: Expanze uzlu 1 nevede na kratší cestu do uzlu 2.



3: Expanzí uzlu 2 získáme cestu též do uzlu 5.



4: Dalšími expanzemi již cesty nezlepšujeme.

Příklad řešení úlohy hledání nejkratších cest v grafu

Řešení úlohy obsahje tři části.

- **Vstupní data** (grafu) – paměťová reprezentace a načtení hodnot.
  Formát vstupního souboru.
  - Vstupní graf je zadán jako seznam hran.
    `from to cost` – Viz 10. přednáška.
  - Dalším vstupem je výchozí uzel.
    Pro jednoduchost budeme uvažovat 1. uzel (0).
- **Výstupní data** (nejkratší cesty) – paměťová reprezentace a uložení (zápis).
  Formát výstupního souboru.
  - Všechny nejkratší cesty vypíšeme jako seznam vrcholů s cenou (délkou) nejkratší cesty a bezprostředním předchůdcem (indexem) uzlu na nejkratší cestě z výchozího uzlu (uzel 0).
    `label cost parent`
- **Algoritmus** hledání cest – Dijkstrův algoritmus.
  - Algoritmus je relativně přímočarý – v každém kroku expandujeme uzel s aktuálně nejkratší cestou z výchozího uzlu.

V každém kroku potřebujeme uzel s aktuálně nejnižší délkou cesty – použijeme prioritní frontu.

Vstupní graf, reprezentace grafu a řešení

- Graf je zadán jako seznam hran v souboru, který můžeme načíst funkcí `load_graph_simple()` z `lec10/*/load_simple.c`.
  - Graf je seznam hran.

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

```
typedef struct {
   edge_t *edges;
   int num_edges;
   int capacity;
} graph_t;
```

lec10/graph.h

```
0 5 74
1 6 56
2 8 11
2 9 27
2 4 31
2 3 41
2 1 26
3 5 24
3 9 12
4 9 13
...
```

- Navíc využijeme toho, že jsou hrany uspořádané.
  - Hrany vycházející z uzlu určíme jako index první hrany `edge_start`
  - a počet hran `edge_count`.

```
typedef struct {
   int edge_start;
   int edge_count;
   int parent;
   int cost;
} node_t;
```

- Dále potřebujeme pro vlastní řešení u každého uzlu uložit cenu nejkratší cesty `cost` a předcházející uzel na nejkratší cestě `parent`.

Datová reprezentace

- Řešení implementujeme v modulu `dijkstra`.
- Všechny potřebné datové struktury zahrneme do jediné struktury `dijkstra_t` reprezentující všechna data řešení úlohy.

```
typedef struct {
   graph_t *graph;
   node_t *nodes;
   int num_nodes;
   int start_node;
} dijkstra_t;
```

- Pro alokaci použijeme `myMalloc()`, `allocate_graph()` a inicializujeme položky struktury na výchozí hodnoty.

```
void* dijkstra_init(void)
{
   dijkstra_t *dij = myMalloc(
    sizeof(dijkstra_t));
   dij->nodes = NULL;
   dij->num_nodes = 0;
   dij->start_node = -1;
   dij->graph = allocate_graph();
   return dij;
}
```

```
#include <stdlib.h>

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

lec11/my_malloc.c

Načtení grafu a inicializace uzlů 1/2

- Hrany načteme např. `load_graph_simple()` nebo impl. HW09.
  Pro jednoduchost a lepší přehlednost zde předpokládáme bezchybné načtení.
- Dále potřebujeme zjistit počet vrcholů. *Lze implementovat přímo do načítání.*
- Alokujeme paměť pro uzly a nastavíme (bezpečné) výchozí hodnoty.

```
load_graph_simple(filename, dij->graph);
int m = -1;
for (int i = 0; i < dij->graph->num_edges; ++i) {
   const edge_t *const e = &(dij->graph->edges[i]);
   m = m < e->from ? e->from : m;
   m = m < e->to ? e->to : m;
} // smyčka pro určení maximálního počtu vrcholů

dij->num_nodes = m + 1; //m je index a začína od 0 proto +1
dij->nodes = myMalloc(sizeof(node_t) * dij->num_nodes);
for (int i = 0; i < dij->num_nodes; ++i) {
   dij->nodes[i].edge_start = -1;
   dij->nodes[i].edge_count = 0;
   dij->nodes[i].parent = -1;  // pokud neexistuje indikujeme -1
   // pro cenu volíme -1 ve výpise bude kratší než např.  MAX_INT
   dij->nodes[i].cost = -1;
} // nastavení výchozích hodnot uzlů
```

Inicializace uzlů 2/2

- Nastavíme indexy hran jednotlivým uzlům.

```
for (int i = 0; i < dij->graph->num_edges; ++i) {
   int cur = dij->graph->edges[i].from;
   if (dij->nodes[cur].edge_start == -1) { // first edge
      // mark the first edge in the array of edges
      dij->nodes[cur].edge_start = i;
   }
   dij->nodes[cur].edge_count += 1; // increase no.  of edges
}
```

Využíváme uspořádání vstupních dat.

Uložení řešení do souboru

- Po nalezení všech nejkratších cest (z uzlu 0) má každý uzel nastavenou hodnotu `cost` s délkou cesty a v `parent` index bezprostředního předchůdce na nejkratší cestě.

Případně -1 pokud cesta do uzlu neexistuje.

```
typedef struct {
   int edge_start;
   int edge_count;
   int parent;
   int cost;
} node_t;
```

```
_Bool dijkstra_save_path(void *dijkstra, const char *filename)
{
   _Bool ret = false;
   const dijkstra_t *const dij = (dijkstra_t*)dijkstra;
   if (dij) {
      FILE *f = fopen(filename, "w");
      if (f) {
         for (int i = 0; i < dij->num_nodes; ++i) {
            const node_t *const node = &(dij->nodes[i]);
            fprintf(f, "%i %i %i\n", i, node->cost, node->parent);
         } // end all nodes
         ret = fclose(f) == 0; // indicate eventuall error in saving
      }
   }
   return ret;
}
```

lec11/dijkstra.c

Zápis řešení do soubou můžeme implementovat jednoduchým výpisem do souboru nebo implementací HW09.

Prioritní fronta pro Dijsktrův algoritmus

- Součástí balíku `lec11/graph_search-array` je rozhraní `pq.h` pro implementaci prioritní fronty s funkcí `update()`.

```
void *pq_alloc(int size);

void pq_free(void *_pq);

_Bool pq_is_empty(const void *_pq);

_Bool pq_push(void *_pq, int label, int cost);

_Bool pq_update(void *_pq, int label, int cost);

_Bool pq_pop(void *_pq, int *oLabel);
```

lec11/graph_search-array/pq.h.

- Jedná se o relativně obecný předpis, který neklade zvláštní požadavky na vnitřní strukturu.
  V balíku je rozhraní implementované v modulu `pq_array-linear.c`, který obsahuje implementaci prioritní fronty polem s lineární složitostí funkcí `push()` a `pop()`.
- `lec11/graph_search-array` základní funkční řešení hledání nejkratší cesty, prioritní fronta implementována polem.

Prioritní fronta (polem) s push() a update()

- Při expanzi uzlu, můžeme do prioritní fronty vkládat uzly s cenou pro každou hranu vycházející z uzlu.
- Obecně může být hran výrazně více než počet uzlů. *Pro plný graf o n uzlech až n^2 hran.*
- Proto pro prioritní frontu implementujeme funkci `update()` a tím zaručíme, že ve frontě bude nejvýše tolik prvků, kolik je vrcholů.
- V prioritní frontě tak můžeme předalokovat maximální počet položek.
- Při volání `update()` však potřebujeme získat pozici daného uzlu v prioritní frontě a změnit jeho hodnotu.
  - Prvek v poli najdeme lineárním průchodem prvků ve frontě.
    Budeme však mít lineární složitost!
  - *Pozici prvku v prioritní frontě uložíme do dalšího pole a získáme okamžitý přístup za cenu mírně složitějšího vkládání prvků a vyšších paměťových nároků (jeden `int` na prvek pole).*
    Operace `update()` bude mít výhodnou konstantní složitost.

Hledání nejkratších cest

- Využijeme implementaci prioritní fronty s `push()` a `update()`.

```c
dij->nodes[dij->start_node].cost = 0; // inicializace
void *pq = pq_alloc(dij->num_nodes); // prioritní fronta
int cur_label;
pq_push(pq, dij->start_node, 0);
while ( !pq_is_empty(pq) && pq_pop(pq, &cur_label)) {
   node_t *cur = &(dij->nodes[cur_label]); // pro snažší použití
   for (int i = 0; i < cur->edge_count; ++i) { // všechny hrany z uzlu
      edge_t *edge = &(dij->graph->edges[cur->edge_start + i]);
      node_t *to = &(dij->nodes[edge->to]);
      const int cost = cur->cost + edge->cost;
      if (to->cost == -1) { // uzel to nebyl dosud navštíven
         to->cost = cost;
         to->parent = cur_label;
         pq_push(pq, edge->to, cost); // vložení vrcholu do fronty
      } else if (cost < to->cost) { // uzel již v pq, proto
         to->cost = cost; // testujeme cost
         to->parent = cur_label; // a aktualizujeme odkaz (parent)
         pq_update(pq, edge->to, cost); // a prioritní frontu pq
      }
   } // smyčka přes všechny hrany z uzlu cur_label
} // prioritní fronta je prázdná
pq_free(pq); // uvolníme paměť
```

lec11/dijkstra.c

Příklad použití

- Základní implementace hledání cest s prioritní frontou implementovanou polem je dostupná v `lec11/graph_search-array`.
- Vytvoříme graf `g` programem `tdijkstra`, např. o max 1000 vrcholech.

  `./tdijkstra -c 1000 g`
- Program zkompilujeme a spustíme, např.

  `./tgraph_search g s`.
- Programem `tdijkstra` můžeme vygenerovat referenční řešení, např.

  `./tdijkstra g s.ref`.
- a naše řešení pak můžeme porovnat, např.

  `diff s s.ref`.

Lineární prioritní fronta vs efektivní implementace

- Ukázková implemetace v `lec11/graph_search-array`, je sice funkční, pro velké grafy je však výpočet pomalý. *Např. pro graf s 1 mil. vrcholů trvá načtení, nalezení všech nejkratší cest a uložení výsledku přibližně 120 sekund na Intel Skylake@3.3GHz.*

```
./tdijkstra -c 1000000 g
/usr/bin/time ./tgraph_search g s
Load graph from g
Find all shortest paths from the node 0
Save solution to s
Free allocated memory
      120.53 real       115.92 user         0.07 sys
```

  - Referenční program `tdijkstra` najde řešení za cca 1 sekundu.

    Též k dispozici jako `tdijkstra.Linux` *a* `tdijkstra.exe`.

```
/usr/bin/time ./tdijkstra g s.ref
        1.03 real         0.94 user         0.07 sys
```

  - Oba programy vracejí identické výsledky

```
md5sum s s.ref
MD5 (s) = 8cc5ec1c65c92ca38a8dadf83f56e08b
MD5 (s.ref) = 8cc5ec1c65c92ca38a8dadf83f56e08b
```

V základní verzi řešení HW10 nesmí být hledání nejkratší cesty více než $2\times$ pomalejší než referenční program (`tdijkstra`).

Prioritní fronta haldou s `push()` a `update()`

- Prioritní frontu implementujeme haldou reprezentovanou v poli.

  Maximální počet prvků dopředu známe.
- Halda zaručí složitost operací `push()` a `pop()` $O(\log n)$.

  Oproti $O(n)$ u jednoduché implemetace prioritní fronty polem.
- Je nutné udržovat vlastnost haldy. Pro kontrolu zachování „*heap property*“ implementujeme rozhraní `pq_is_heap()`. Použijeme pouze pro ladění.

```c
_Bool pq_is_heap(void *heap, int n);
```

lec11/graph_search/pq_heap.h

- Pro zachování složitosti operací práce s haldou potřebujeme efektivně implementovat také funkci `update()`, tj. $O(\log n)$.
  - Potřebujeme znát pozici daného uzlu v haldě.

    Zavedeme pomocné pole s indexem `heapIDX`.
  - Při hledání nejkratších cest se délka cesty pouze snižuje.
  - Proto se aktualizovaný „uzel“ může v haldě pohybovat pouze směrem nahoru.

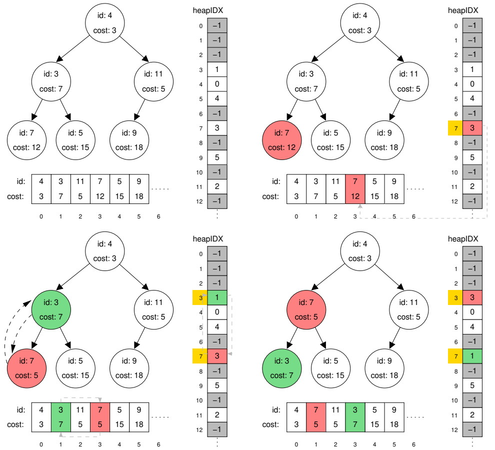
    Jedná se tak o identický postup, jako při přidání nového prvku funkcí `push()`. *V tomto případě však prvek může startovat z vnitřku stromu.*

Příklad reprezentace haldy v poli a aktualizace ceny cesty

V haldě jsou uloženy délky dosud známých nejkratších cest pro vrcholy označené: 3, 4, 5, 7, 9, a 11.

- Při expanzi dalšího uzlu jsme našli kratší cestu do uzlu 7 s délkou 5.

  Zavoláme `update(id = 7, cost = 5)`.
- Abychom mohli aktualizovat cenu v haldě, potřebujeme znát pozici uzlu v poli haldy.
- Proto vedle samotné haldy udržujeme pole, které je indexované číslem uzlu.
- Po aktualizaci ceny, není splněna vlastnost haldy. Provedeme záměnu.
- Při záměně udržujeme nejen prvky v samotné haldě, ale také pole `heapIDX` s pozicemi vrcholů v poli haldy.



Příklad implementace

- V `lec11/graph_search` je uveden příklad implementace hledání nejkratších cest s prioritní frontou realizovanou haldou.
- Implementace funkce `update()` využívá pole `heapIDX` pro získání pozice prvku v haldě, záměrně je však splnění vlastnosti haldy realizováno vytvořením nové haldy s aktualizovanou cenou uzlu.

```c
_Bool pq_update(void *_pq, int label, int cost)
{
   _Bool ret = false;
   pq_heap_s *pq = (pq_heap_s*)_pq;
   pq->cost[pq->heapIDX[label]] = cost; // update the cost, but heap property is not satified
   // assert(pq_is_heap(pq, 0));

   pq_heap_s *pqBackup = (pq_heap_s*)pq_alloc(pq->size); //create backup of the heap
   pqBackup->len = pq->len;
   for (int i = 0; i < pq->len; ++i) { // backup the help
      pqBackup->cost[i]= pq->cost[i]; //just cost and labels
      pqBackup->label[i] = pq->label[i];
   }
   pq->len = 0; //clear all vertices in the current heap
   for (int i = 0; i < pqBackup->len; ++i) { //create new heap from the backup
      pq_push(pq, pqBackup->label[i], pqBackup->cost[i]);
   }
   pq_free(pqBackup); // release the queue
   ret = true;
   return ret;
}
```

Součástí řešení 10. domácího úkolu je správná implementace funkce `update()`!

Příklad řešení a rychlost výpočtu

- Po úpravě funkce `update()` získáme prioritní frontu se složitostí operací $O(\log n)$ a vlastní výpočet bude relativně rychlý.
- Pro získání představy rychlosti výpočtu je v souboru `tgraph_search-time.c` volání dílčích funkcí modulu `dijkstra` s měřením reálného času (`make time`). `lec11/graph_search-time.c`

  Alternativně lze řešit nástrojem `time` *nebo pro Win platformu* `lec11/bin/timeexec.exe`.
- Vytvoříme graf o 1 mil. uzlů (a cca 3 mil. hran) v souboru `/tmp/g`.

  `./bin/tdijkstra -c 10000000 /tmp/g`

Verze s naivním `update()`

```
tgraph_search-time /tmp/g /tmp/s1
Load graph from /tmp/g
Load time ....1179ms
Save solution to /tmp/s1
Solve time ...965875 ms
Save time ....273 ms
Total time ...967327ms
```

Upravená funkce `update()`

```
tgraph_search-time /tmp/g /tmp/s2
Load graph from /tmp/g
Load time ....1201ms
Save solution to /tmp/s2
Solve time ...620 ms
Save time ....279 ms
Total time ...2100ms
```

- Správnost řešení lze zkontrolovat program `tdijsktra`, např.

  `./bin/tdijkstra -t /tmp/g /tmp/s.`

Další možnosti urychlení programu

- Kromě efektivní implemetace prioritní fronty haldou, která je zásadní, lze běh programu dále urychlit
  - efektivnějším načítáním grafu
  - a ukládáním řešení do souboru.

```
tgraph_search s.tgs
lec11/tgraph_search
Load time ....1252ms
Solve time ...625 ms
Save time ....431 ms
Total time ...2308ms
```

```
tdijkstra -v g s.ref
Dijkstra ver. 2.3.4
Load time ....223ms
Solve time ...715ms
Save time ....106ms
Total time ...1044ms
```

```
dijkstra-pv g s.pv
HW10 Reference solution
Load time ....235ms
Solve time ...610 ms
Save time ....87 ms
Total time ...932ms
```

- HW10 – Soutěž v rychlosti programu – extra body navíc.
  - Na odevzdání stačí opravit funkci `update()` případně využít načítání a ukládání z HW09.
  - Dalšího urychlení lze dosáhnout lepší organizací paměti a datovými strukturami.

Jediný zásadní požadavek je implementace rozhraní dle `lec11/dijkstra.h`.

Část III

Část 3 – Zadání 10. domácího úkolu (HW10)

Zadání 10. domácího úkolu HW10

Téma: **Integrace načítání grafu a prioritní fronta v úloze hledání nejkratších cest**

Povinné zadání: **3b**; Volitelné zadání: **3b**; Bonusové zadání: Soutěž o body

- **Motivace**: Větší programový celek, využití existujícího kódu a efektivní implementace programu.
- **Cíl**: Osvojit si integraci existujících kódu do funkčního celku složeného z více souborů.
- **Zadání**: `https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw10`
  - Funkce `update()` pro efektivní použití prioritní fronty implementované haldou v úloze hledání nejkratší cest v grafu.
  - Volitelné zadání rozšiřuje binární načítání/ukládání grafu o specifikovaný binární formát, tj. rozšíření HW 09.
  - Bonusové zadání spočívá v efektivnosti implementace tak, aby byl výsledný kód co možná nejrychlejší.
- Termín odevzdání: 08.01.2023, 23:59:59 PST.
- Bonusová úloha: 14.01.2023, 23:59:59 PST.

Shrnutí přednášky

Diskutovaná témata

- Prioritní fronta
  - Příklad implementace spojovým seznamem
    `lec11/priority_queue-linked_list`
  - Příklad implementace polem
    `lec11/priority_queue-array`
- Halda - definice, vlastnosti a základní operace
- Reprezentace binárního plného stromu polem
- Prioritní fronta s haldou
- Hledání nejkratší cesty v grafu – využití prioritní fronty (resp. haldy)