

Prioritní fronta a příklad použití v úloze hledání nejkratsích cest Jan Faigl Katedra počítačů Fakulta elektrotechnická České vysoké učení technické v Praze Přednáška 11 B0B36PRP – Procedurální programování			Přehled témat ■ Část 1 – Prioritní fronta polem a haldou Prioritní fronta polem Halda ■ Část 2 – Příklad využití prioritní fronty v úloze hledání nejkratší cesty v grafu Popis úlohy Návrh řešení Příklad naivní implementace prioritní fronty polem Implementace pq haldou s push() a update() ■ Část 3 – Zadání bonusového úkolu (HW10B)			Prioritní fronta polem Halda Část I Část 1 – Prioritní fronta (Halda)		
Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	1 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	2 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	3 / 50
Prioritní fronta polem 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 <pre>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;</pre> ■ Další rozhraní (jména a argumenty funkcí) mohou zůstat identické jako u implementace spojovým seznamem. Viz 9. přednáška. <pre>void queue_init(queue_t **queue); int queue_push(void *value, int priority, void queue_delete(queue_t **queue); queue_t *queue); void queue_free(queue_t *queue); void* queue_pop(queue_t *queue); _Bool queue_is_empty(const queue_t *queue); void* queue_peek(const queue_t *queue);</pre>			Prioritní fronta polem Halda Prioritní fronta polem 1/3 – push() ■ Funkce push() je až na uložení priority identická s verzí bez priorit. <pre>46 int queue_push(void *value, int priority, queue_t *queue) 47 { 48 int ret = QUEUE_OK; // by default we assume push will be OK 49 if (queue->count < MAX_QUEUE_SIZE) { 50 queue->queue[queue->tail] = value; 51 queue->priorities[queue->tail] = priority; // store priority of the new value entry 52 queue->tail = (queue->tail + 1) % MAX_QUEUE_SIZE; 53 queue->count += 1; 54 } else { 55 ret = QUEUE_MEMFAIL; 56 } 57 return ret; 58 }</pre> 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 Halda Prioritní fronta polem 2/3 – getEntry() ■ Nalezení nejmenšího (největšího) prvku provedeme lineárním prohledáním aktuálních prvků uložených ve frontě (poli). <pre>61 static int getEntry(const queue_t *const queue) 62 { 63 int ret = -1; // return -1 if queue is empty. 64 if (queue->count > 0) { 65 for (int cur = queue->head, i = 0; i < queue->count; ++i) { 66 if (67 ret == -1 68 (queue->priorities[ret] > queue->priorities[cur]) 69) { 70 ret = cur; 71 } 72 cur = (cur + 1) % MAX_QUEUE_SIZE; 73 } 74 } 75 return ret; 76 }</pre> lec11/priority_queue_array/priority_queue_array.c		
Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	5 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	6 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	7 / 50
Prioritní fronta polem Halda Prioritní fronta polem 3/3 – peek() a pop() ■ Funkce peek() využívá lokální (static) funkce getEntry(). <pre>101 void* queue_peek(const queue_t *queue) Co se stane, když zavoláme getEntry() na prázdnou frontu? Proč je static? 102 { 103 return queue_is_empty(queue) ? NULL : queue->queue[getEntry(queue)]; 104 }</pre> ■ Ve funkci pop() zaplníme položku vyjmutého prvku prvkem ze startu. Tim zajistíme, že prvky tvoří souvislý blok v rámci kruhové fronty. <pre>77 void* queue_pop(queue_t *queue) 78 { 79 void *ret = NULL; 80 int bestEntry = getEntry(queue); 81 if (bestEntry >= 0) { // entry has been found 82 ret = queue->queue[bestEntry]; 83 if (bestEntry != queue->head) { //replace the bestEntry by head 84 queue->queue[bestEntry] = queue->queue[queue->head]; 85 queue->priorities[bestEntry] = queue->priorities[queue->head]; 86 } 87 queue->head = (queue->head + 1) % MAX_QUEUE_SIZE; 88 queue->count -= 1; 89 } 90 return ret; 91 }</pre>			Prioritní fronta polem Halda Prioritní fronta polem – příklad použití ■ Použití je identické s implementací spojovým seznamem. <pre>\$ 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</pre> 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. ■ Tim 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.		
Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	8 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	9 / 50	Jan Faigl, 2025	B0B36PRP – Přednáška 11: Halda a hledání nejkratsích cest	10 / 50

Prioritní fronta polemHalda

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.
- 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 $\text{push}()$, $\text{pop}()$, $\text{peek}()$ tak můžeme očekávat nikoliv $O(n)$ (jako v případě předchozí implementace prioritní fronty polem a spojovým seznamem), ale $O(\log n)$ a pro $\text{peek}()$ dokonce $O(1)$.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest12 / 50

Prioritní fronta polemHalda

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

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest13 / 50

Prioritní fronta polemHalda

Halda – přidání prvku **push()**

- Po každém provedení operace **push()** musí být splněny vlastnosti haldy.
- Prvek přidáme na konec haldy, tj. na první volnou pozici (vlevo) na nejnižší úrovni haldy.
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s nadřazeným prvkem (předkem).

V nejnepříznivějším případě prvek „probublá“ až do kořene stromu.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest14 / 50

Prioritní fronta polemHalda

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.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest15 / 50

Prioritní fronta polemHalda

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 $O(\log(n))$.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest16 / 50

Prioritní fronta polemHalda

Reprezentace binárního stromu polem

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

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest17 / 50

Prioritní fronta polemHalda

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: $i_{\text{levý}} = 2i + 1$;
 - pravý následník: $i_{\text{pravý}} = 2i + 2$.

Podobně lze odvodit vztah pro předchůdce.

- Kořen stromu reprezentuje nejprioritnější prvek.

Např. s nejmenší hodnotou nebo maximální prioritou.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest18 / 50

Prioritní fronta polemHalda

Operace vkládání a odebrání prvků

- I v případě reprezentace polem pracují operace vkládání a odebrání 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()**.

Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest19 / 50

Prioritní fronta polemHalda

Příklad implementace **pq_is_heap()**

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

```
18 typedef struct {
19     int size; // the maximal number of entries
20     int len; // the current number of entries
21     int *cost; // array with costs - lowest cost is highest priority
22     int *label; // array with labels (each label has cost/priority)
23 } pq_heap_s;
24
25 _Bool pq_is_heap(pq_heap_s *pq, int n)
26 {
27     _Bool ret = true;
28     int l = 2 * n + 1; // left successor
29     int r = l + 1; // right successor
30     if (l < pq->len) {
31         ret = (pq->cost[l] < pq->cost[n]) ? false : pq_is_heap(pq, l);
32     }
33     if (r < pq->len) {
34         ret = ret // if ret is false, further expression is not evaluated
35         && (pq->cost[r] < pq->cost[n]) ? false : pq_is_heap(pq, r) ;
36     }
37     return ret;
38 }
```

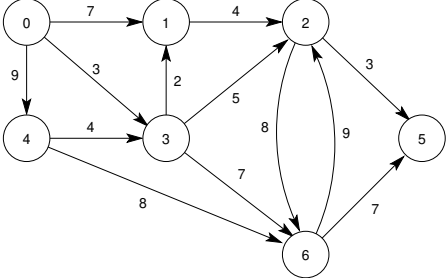
Jan Faigl, 2025BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest20 / 50

Příklad implementace push()

```
■ Prvek přidáme na konec pole a iterativně kontrolujeme, zdali je splněna vlastnost haldy. Pokud ne, prvek zaměníme s předchůdcem.
81 #define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2
82 _Bool pq_push(pq_heap_s *pq, int label, int cost)
83 {
84     _Bool ret = false;
85     if (pq->len < pq->size && label >= 0 && label < pq->size) {
86         pq->cost[pq->len] = cost; //add the cost to the next free slot
87         pq->label[pq->len] = label; //add label of new entry
88         int cur = pq->len; // index of the entry added to the heap
89         int parent = GET_PARENT(cur);
90         while (cur >= 1 && pq->cost[parent] > pq->cost[cur]) {
91             pq_swap(pq, parent, cur); // swap parent<->cur
92             cur = parent;
93             parent = GET_PARENT(cur);
94         }
95         pq->len += 1;
96         ret = true;
97     }
98     // assert(pq_is_heap(pq, 0)); // testing the implementation
99     return ret;
100 }
101 }
```

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.



Příklad řešení úlohy hledání nejkratších cest v grafu

Řešení úlohy obsahuje tři části.

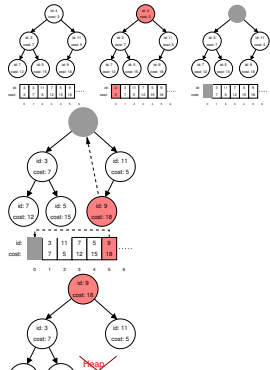
- Vstupní data** (grafu) – paměťová reprezentace a načtení hodnot.
Formát vstupního souboru.
Vstupní graf je zadán jako seznam hran.
from to cost – Viz 10. přednáška.
Dalším vstupem je výchozí uzel.
Pro jednoduchost budeme uvažovat 1. uzel (0).
- Výstupní data** (nejkratší cesty) – paměťová reprezentace 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.

Příklad volání pop()

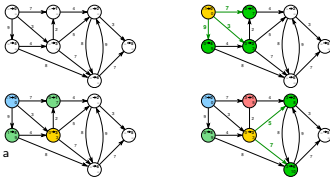
- Halda je reprezentovaná binárním polem.
- Nejmenší prvek je kořenem stromu.
- Voláním pop() odebíráme kořen stromu.
- Na jeho místo umístíme poslední prvek.
- Strom však nesplňuje podmínku haldy.
- Proto provedeme záměnu s následníky.
V tomto případě volíme pravého následníka, neboť jeho hodnota je 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.



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.



Vstupní graf, reprezentace grafu a řešení

- Graf je zadán jako seznam hran v souboru, který můžeme načíst funkcí load_graph_simple() z lec11/*load_simple.c.
- Graf je seznam hran.

```
10 typedef struct {
11     int from;
12     int to;
13     int cost;
14 } edge_t;

15 typedef struct {
16     edge_t *edges;
17     int num_edges;
18     int capacity;
19 } graph_t;

16 typedef struct {
17     int edge_start;
18     int edge_count;
19     int parent;
20     int cost;
21 } node_t;
```
- Využijeme uspořádání hran ve vstupním souboru.
 - Hrany vycházející z uzlu určíme jako index první hrany edge_start
 - a počet hran edge_count.
- Řešení nejkratších cest, reprezentujeme uložením ke každému vrcholu: cena nejkratší cesty cost a předcházející uzel na nejkratší cestě parent.

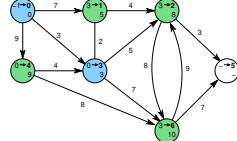
```
Příklad vstupního souboru,
seznamu hran.

1 0 5 74
2 1 6 56
3 2 8 11
4 2 9 27
5 2 4 31
6 2 3 41
7 2 1 26
8 3 5 24
9 3 9 12
10 4 9 13
11 ...
16 typedef struct {
17     int edge_start;
18     int edge_count;
19     int parent;
20     int cost;
21 } node_t;
```

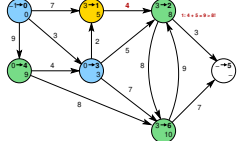
Část II

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

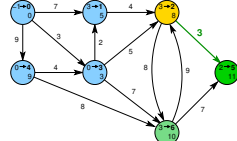
Příklad postupu řešení (pokračování)



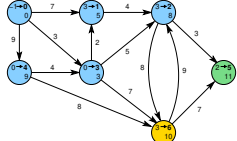
1: Po 2. expanzi má uzel 3 již nejkratší cestu.



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



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



4: Dalšími expanzemi již cesty nezlepšujeme.

Datová reprezentace

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

```
23 typedef struct {
24     graph_t *graph;
25     node_t *nodes;
26     int num_nodes;
27     int start_node;
28 } dijkstra_t;
```
- Pro alokaci použijeme myMalloc(), allocate_graph() a inicializujeme položky struktury na výchozí hodnoty.

```
6 void* myMalloc(size_t size)
7 {
8     void *ret = malloc(size);
9     if (!ret) {
10         fprintf(stderr, "Malloc failed!\n");
11         exit(-1);
12     }
13     return ret;
14 }
```
- dijskra_init(void) inicializuje strukturu dijkstra_t.

```
31 void* dijskra_init(void)
32 {
33     dijkstra_t *dij = myMalloc(sizeof(dijkstra_t));
34     dij->nodes = NULL;
35     dij->num_nodes = 0;
36     dij->start_node = -1;
37     dij->graph = allocate_graph();
38     return dij;
39 }
```

Načtení grafu a inicializace uzlů 1/2

- Hrany načteme např. `load_graph_simple()`.
- Zjistíme počet vrcholů jako nejvyšší číslo uzlu hran.

```
46 _Bool dijkstra_load_graph(const char *filename, void *dijkstra)
47 {
48     _Bool ret = false;
49     dijkstra_t *dij = (dijkstra_t*)dijkstra;
50     if (
51         dij && dij->graph &&
52         load_graph_simple(filename, dij->graph)
53     ) { // edges has not been loaded
54         // dijkstra_t and graph has been allocated and edges have been loaded here
55         // go through the edges and create array of nodes with indexing to edges
56         // 1st get the maximal number of nodes
57         int m = -1;
58         for (int i = 0; i < dij->graph->num_edges; ++i) {
59             const edge_t *const e = &(dij->graph->edges[i]); // use pointer to avoid copying
60             m = m < e->from ? e->from : m;
61             m = m < e->to ? e->to : m;
62         }
63         m += 1; // m is the index therefore we need +1 for label 0
```

Lze implementovat přímo do načítání.

lec11/graph_search/dijkstra.c

Prioritní fronta pro Dijkstrův 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.

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.`

Inicializace uzlů 2/2

- Alokuje paměť pro uzly a nastavíme (bezpečně) výchozí hodnoty.

```
64 dij->nodes = myMalloc(sizeof(node_t) * m);
65 dij->num_nodes = m;
66 for (int i = 0; i < m; ++i) { // 2nd initialization of the nodes
67     dij->nodes[i].edge_start = -1;
68     dij->nodes[i].edge_count = 0;
69     dij->nodes[i].parent = -1;
70     dij->nodes[i].cost = -1;
71 }
```

- Nastavíme indexy hran jednotlivým uzlům s využitím uspořádání vstupních dat.

```
77 for (int i = 0; i < dij->graph->num_edges; ++i) { // 3rd add edges to the nodes
78     int cur = dij->graph->edges[i].from;
79     if (dij->nodes[cur].edge_start == -1) { // first edge
80         dij->nodes[cur].edge_start = i; // mark the first edge in the array of edges
81     }
82     dij->nodes[cur].edge_count += 1; // increase number of edges
83 }
84 ret = true;
85 }
86 return ret;
```

lec11/graph_search/dijkstra.c

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 n² 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 lineárních 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.

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-lnx a tdijkstra.exe.
$ /usr/bin/time ./tdijkstra g s.ref
1.03 real    0.94 user    0.07 sys
■ Oba programy vracejí identické výsledky
1 $ md5sum s s.ref
2 MD5 (s) = 8cc5ec1c65c92ca38a8dadf83f56e08b
3 MD5 (s.ref) = 8cc5ec1c65c92ca38a8dadf83f56e08b
V základní verzi řešení HW10 nesmí být hledání nejkratší cesty více než 2x pomalejší než referenční program (tdijkstra).
```

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

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.

```
128 _Bool dijkstra_save_path(void *dijkstra, const char *filename)
129 {
130     _Bool ret = false;
131     const dijkstra_t *const dij = (dijkstra_t*)dijkstra;
132     if (dij) {
133         FILE *f = fopen(filename, "w");
134         if (f) {
135             for (int i = 0; i < dij->num_nodes; ++i) {
136                 const node_t *const node = &(dij->nodes[i]);
137                 fprintf(f, "%i %i %i\n", i, node->cost, node->parent);
138             } // end all nodes
139             ret = fclose(f) == 0; // indicate eventuell error in saving
140         }
141     }
142     return ret;
143 }
```

lec11/graph_search/dijkstra.c

Hledání nejkratších cest – dijkstra_solve()

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

```
100 dij->nodes[dij->start_node].cost = 0; // inicializace
101 void *pq = pq_alloc(dij->num_nodes); // prioritní fronta
102 int cur_label;
103 pq_push(pq, dij->start_node, 0);
104 while ( !pq_is_empty(pq) && pq_pop(pq, &cur_label)) {
105     node_t *cur = &(dij->nodes[cur_label]); // pro snažší použití
106     for (int i = 0; i < cur->edge_count; ++i) { // všechny hrany z uzlu
107         edge_t *edge = &(dij->graph->edges[cur->edge_start + i]);
108         node_t *to = &(dij->nodes[edge->to]);
109         const int cost = cur->cost + edge->cost;
110         if (to->cost == -1) { // uzel to nebyl dosud navštíven
111             to->cost = cost;
112             to->parent = cur_label;
113             pq_push(pq, edge->to, cost); // vložení vrcholu do fronty
114         } else if (cost < to->cost) { // uzel již v pq, proto
115             to->cost = cost; // testujeme cost
116             to->parent = cur_label; // a aktualizujeme odkaz (parent)
117             pq_update(pq, edge->to, cost); // a prioritní frontu pq
118         }
119     } // smýčka přes všechny hrany z uzlu cur_label
120 } // prioritní fronta je prázdná
121 pq_free(pq); // uvolníme paměť
```

lec11/dijkstra.c

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.

```
110 _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 funkci 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.

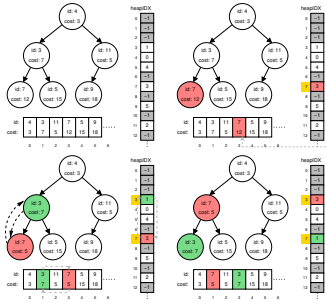
Zavoláme update(id = 7, cost = 5).

- 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.



Další možnosti urychlení programu

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

1 \$./tgraph_search s.tgs	1 \$./tdijkstra -v g s.ref	1 ./dijkstra-pv g s.pv
2 # lec11/tgraph_search	2 Dijkstra ver. 2.3.4	2 HW10 Reference solution
3 Load time1252 ms	3 Load time223 ms	3 Load time235 ms
4 Solve time ...625 ms	4 Solve time ...715 ms	4 Solve time ...610 ms
5 Save time ...431 ms	5 Save time ...106 ms	5 Save time 87 ms
6 Total time ...2308 ms	6 Total time ...1044 ms	6 Total time ...932 ms

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

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

Diskutovaná témata

Shrnutí přednášky

Příklad implementace

- V lec11/graph_search je 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.

```
100 _Bool pq_update(void *_pq, int label, int cost)
101 {
102     _Bool ret = false;
103     pq_heap_t *pq = (pq_heap_t*)_pq;
104     pq->cost[pq->heapIDX[label]] = cost; // update the cost, but heap property is not
105     satisfied
106     // assert(pq_is_heap(pq, 0));
107     pq_heap_t *pqBackup = (pq_heap_t*)pq_alloc(pq->size); //create backup of the heap
108     pqBackup->len = pq->len;
109     for (int i = 0; i < pq->len; ++i) { // backup the heap
110         pqBackup->cost[i] = pq->cost[i]; //just cost and labels
111         pqBackup->label[i] = pq->label[i];
112     }
113     pq->len = 0; //clear all vertices in the current heap
114     for (int i = 0; i < pqBackup->len; ++i) { //create new heap from the backup
115         pq_push(pq, pqBackup->label[i], pqBackup->cost[i]);
116     }
117     pq_free(pqBackup); // release the queue
118     ret = true;
119     return ret;
120 }
```

Součástí řešení HW10B je správná implementace funkce update()!

Část III

Část 3 – Zadání bonusového úkolu (HW10B)

Diskutovaná témata

Diskutovaná témata

- Prioritní fronta
 - Příklad implementace spojovým seznamem
 - Příklad implementace polem
- 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)

lec11/priority_queue-linked_list

lec11/priority_queue-array

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
- Vytvoříme graf o 1 mil. uzlů (a cca 3 mil. hran) v souboru /tmp/g.
./bin/dijkstra -c 10000000 /tmp/g

Verze s naivním update() Upravená funkce update()

1 \$./tgraph_search-time /tmp/g /tmp/s1	1 \$./tgraph_search-time /tmp/g /tmp/s2
2 Load graph from /tmp/g	2 Load graph from /tmp/g
3 Load time1179ms	3 Load time1201ms
4 Save solution to /tmp/s1	4 Save solution to /tmp/s2
5 Solve time ...965875 ms	5 Solve time ...620 ms
6 Save time ...273 ms	6 Save time ...279 ms
7 Total time ...967327ms	7 Total time ...2100ms

<https://youtu.be/LQUGP8EqeLM>

- Správnost řešení lze zkontrolovat program tdiijkstra, např.
./bin/dijkstra -t /tmp/g /tmp/s.

Zadání 10. bonusového úkolu HW10B

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

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/hw10b>
 - Funkce update() pro efektivní použití prioritní fronty implementované haldou v úloze hledání nejkratší cest v grafu.
 - Bonusové zadání spočívá v efektivnosti implementace tak, aby byl výsledný kód co možná nejrychlejší.
- Bonusová úloha: 09.01.2026, 23:59:59 PST.

Příklad použití vchozích souborů pro HW10B

Příklad ladění krokováním

Část V

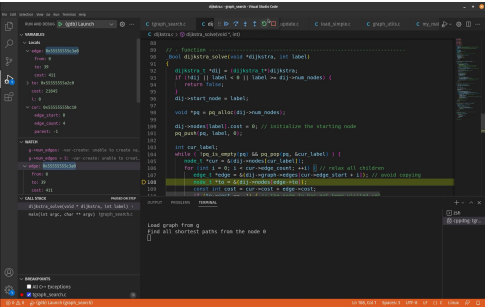
Appendix

Hledání nejkratší cesty v grafu

```
BOB36PRP-lec11-codes ls
bin                priority_queue-linked_list
graph_test         queue
graph_search       queue_test
graph_search_array solution.txt
priority_queue_array stack
BOB36PRP-lec11-codes ls bin
dijkstra-load      dijkstra-load2 dijkstra.exe
dijkstra-load2     dijkstra-no-update.exe
dijkstra-load      dijkstra-no-update.exe
BOB36PRP-lec11-codes cd graph_search
graph_search gnos
cache clang -o dijkstra.c -O2 -g -pedantic -Wall -Werror -o dijkstra.o
cache clang -o my_malloc.c -O2 -g -pedantic -Wall -Werror -o my_malloc.o
cache clang -o graph_utils.c -O2 -g -pedantic -Wall -Werror -o graph_utils.o
cache clang -o pq_heap-no-update.c -O2 -g -pedantic -Wall -Werror -o pq_heap-no-update.o
cache clang -o load_simple.c -O2 -g -pedantic -Wall -Werror -o load_simple.o
cache clang -o tgraph_search.c -O2 -g -pedantic -Wall -Werror -o tgraph_search.o
cache clang -o tgraph_search-time.c -O2 -g -pedantic -Wall -Werror -o tgraph_search-time.o
cache clang dijkstra.o my_malloc.o graph_utils.o pq_heap-no-update.o load_simple.o tgraph_search.o -lm -o tgraph_search
BOB36PRP-lec11-codes cd graph_search
```

<https://youtu.be/LQUGP8EqeLM>

Příklad ladění krokováním



https://youtu.be/rTv_ypcm9XI (~ 25 min)