

Prioritní fronta a příklad použití v úloze hledání nejkratších cest

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

BOB36PRP – 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 10. domácího úkolu (HW10)

Prioritní fronta polem

Halda

Část I

Část 1 – Prioritní fronta (Halda)

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

1 / 50

Prioritní fronta polem

Halda

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

2 / 50

Prioritní fronta polem

Halda

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších 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

```
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.

```
void queue_init(queue_t **queue); // int queue_push(void *value, int priority, queue_t *queue);
void queue_delete(queue_t **queue); // void * queue_pop(queue_t *queue);
void queue_free(queue_t *queue); // void * queue_peek(const queue_t *queue);
_Bool queue_is_empty(const queue_t *queue);
```

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

5 / 50

Prioritní fronta polem

Halda

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

6 / 50

Prioritní fronta polem

Halda

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

7 / 50

Prioritní fronta polem

Halda

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

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

```
101 void * queue_peek(const queue_t *queue)
102 {
103     return queue_is_empty(queue) ? NULL : queue->queue[getEntry(queue)];
104 }
```

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

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

```
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 }
```

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

8 / 50

Prioritní fronta polem – příklad použití

- Použití je identické s implementací spojovým seznamem.

```
$ make && ./demo-priority_queue_array
cache clang -c priority_queue_array.c -D2 -o priority_queue_array.o
cache 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
```

Jan Faigl, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

9 / 50

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 nejnížší (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, 2024

BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest

10 / 50

Prioritní fronta polem Halda

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 $O(1)$.

Jan Faigl, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 12 / 50

Prioritní fronta polem Halda

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, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 13 / 50

Prioritní fronta polem Halda

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, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 14 / 50

Prioritní fronta polem Halda

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, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 15 / 50

Prioritní fronta polem Halda

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

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

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

Jan Faigl, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 16 / 50

Prioritní fronta polem Halda

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, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 17 / 50

Prioritní fronta polem Halda

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 hodnotou nebo maximální prioritou.

Jan Faigl, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 18 / 50

Prioritní fronta polem Halda

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, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 19 / 50

Prioritní fronta polem Halda

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;

101 _Bool pq_is_heap(pq_heap_s *pq, int n)
102 {
103     _Bool ret = true;
104     int l = 2 * n + 1; // left successor
105     int r = l + 1; // right successor
106     if (l < pq->len) {
107         ret = (pq->cost[l] < pq->cost[n]) ? false : pq_is_heap(pq, l);
108     }
109     if (r < pq->len) {
110         ret = ret // if ret is false, further expression is not evaluated
111             &&
112             ((pq->cost[r] < pq->cost[n]) ? false : pq_is_heap(pq, r));
113     }
114     return ret;
115 }
  
```

Jan Faigl, 2024 BOB36PRP – Přednáška 11: Halda a hledání nejkratších cest 20 / 50

Prioritní fronta polem

Příklad implementace push()

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

```

1 #define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2
2
3 _Bool pq_push(pq_heap_s *pq, int label, int cost)
4 {
5     _Bool ret = false;
6     if (pq->len < pq->size && label >= 0 && label < pq->size) {
7         pq->cost[pq->len] = cost; //add the cost to the next free slot
8         pq->label[pq->len] = label; //add label of new entry
9         int cur = pq->len; // index of the entry added to the heap
10        int parent = GET_PARENT(cur);
11        while (cur > 1 && pq->cost[parent] > pq->cost[cur]) {
12            pq_swap(pq, parent, cur); // swap parent<->cur
13            cur = parent;
14            parent = GET_PARENT(cur);
15        }
16        pq->len += 1;
17        ret = true;
18    }
19    // assert(pq_is_heap(pq, 0)); // testing the implementation
20    return ret;
21 }

```

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 21 / 50

Prioritní fronta polem

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 nespĺňuje podmínku haldy.
- Proto provedeme záměnu s následníky.
- 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.

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 22 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

Část II

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

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 23 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

Hledání nejkratší cesty v grafu

- Uzly grafu mohou reprezentovat jednotlivá místa a hrany cestu, jak se mezi místy pohybat.
- 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.

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 25 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

Dijkstrův algoritmus

- Nechť má graf pouze kladné ohodnocení hran, pak pro každý uzel:
 - nastavíme aktuální cenu nejkratší cesty z výchozího uzlu;
 - udržíme 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
 - 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.

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 26 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

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.

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 27 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

Příklad řešení úlohy hledání nejkratších cest v grafu

Řešení úlohy obsahuje tři části.

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

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 29 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

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.

from	to	cost
1	0	5
2	1	6
3	2	8
4	2	9
5	2	4
6	2	3
7	3	4
8	3	1
9	3	12
10	4	9
11	4	13
- 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`.

```

10 typedef struct {
11     edge_t *edges;
12     int num_edges;
13     int capacity;
14 } graph_t;
15
16 typedef struct {
17     int edge_start;
18     int edge_count;
19     int parent;
20     int cost;
21 } node_t;

```

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 30 / 50

Popis úlohy

Návrh řešení

Příklad naivní implementace prioritní fronty polem

pq_haldou s push() a update()

Datová reprezentace

- Řešení implementujeme v modulu `dijkstra`.
- Všechny potřebné datové struktury zahrneme do jediné struktury `dijkstra_t` reprezentující všechna data řešení úlohy.
- Pro alokaci použijeme `myMalloc()`, `allocate_graph()` a inicializujeme položky struktury na výchozí hodnoty.

```

23 typedef struct {
24     graph_t *graph;
25     node_t *nodes;
26     int num_nodes;
27     int start_node;
28 } dijkstra_t;

```

```

31 void* dijkstra_init(void)
32 {
33     dijkstra_t *dij = myMalloc(
34         sizeof(dijkstra_t));
35     dij->nodes = NULL;
36     dij->num_nodes = 0;
37     dij->start_node = -1;
38     dij->graph = allocate_graph();
39     return dij;
40 }

```

Jan Faigl, 2024 BOB36PRP – Prednáška 11: Halda a hledání nejkratších cest 31 / 50

Načtení grafu a inicializace uzlů 1/2

```

■ Hranu načteme např. load_graph_simple() nebo impl. HW09.
■ 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
64     }
65     return ret;
66 }
67
68     lec11/graph_search/dijkstra.c

```

Inicializace uzlů 2/2

```

■ Alokuje paměť pro uzly a nastavíme (bezpečně) výchozí hodnoty.
64 dij->nodes = malloc(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 }
72
73     lec11/graph_search/dijkstra.c

```

```

■ Nastavíme indexy hran jednotlivým uzlům s využitím uspořádání vstupních dat.
74 for (int i = 0; i < dij->graph->num_edges; ++i) { // 3rd add edges to the nodes
75     int cur = dij->graph->edges[i].from;
76     if (dij->nodes[cur].edge_start == -1) { // first edge
77         dij->nodes[cur].edge_start = i; // mark the first edge in the array of edges
78     }
79     dij->nodes[cur].edge_count += 1; // increase number of edges
80 }
81     ret = true;
82     return ret;
83 }
84
85     lec11/graph_search/dijkstra.c

```

Uložení řešení do souboru

```

■ Po nalezení všech nejkratších cest (z uzlu 0) má každý uzel nastavenou hodnotu cost s délkou cesty a v index bezprostředního předchůdce na nejkratší cestě.
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 eventual error in saving
140         }
141     }
142     return ret;
143 }
144
145     lec11/graph_search/dijkstra.c

```

Zápis řešení do souboru můžeme implementovat jednoduchým výpisem do souboru nebo implementaci HW09.

Prioritní fronta pro Dijkstraův algoritmus

```

■ Součástí balíku lec11/graph_search-array je rozhraní pq.h pro implementaci prioritní fronty s funkcí update().
1 void *pq_alloc(int size);
2 void pq_free(void *_pq);
3 _Bool pq_is_empty(const void *_pq);
4 _Bool pq_push(void *_pq, int label, int cost);
5 _Bool pq_update(void *_pq, int label, int cost);
6 _Bool pq_pop(void *_pq, int *oLabel);
7
8     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 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 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 mímě složitějšího vkládání prvků a vyšších paměťových nároků (jeden int na prvek pole).
        Operace update() bude mít výhodnou konstantní složitost.

```

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 snazší 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     } // smyčka přes všechny hrany z uzlu cur_label
120 } // prioritní fronta je prázdná
121 pq_free(pq); // uvolníme paměť
122
123     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 implementace 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 nejkratších 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-lix 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 nejkratších cest více než 2x 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 implementace 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);
111
112     lec11/graph_search/pq_heap.h

```

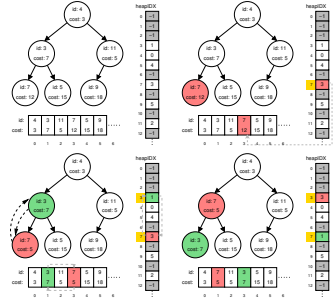
■ Pro zachování složitosti operací práce s haldou potřebujeme efektivně implementovat také funkci update(), tj. O(log n).

- 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.



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.

```

109 _Bool pq_update(void *_pq, int label, int cost)
110 {
111     _Bool ret = false;
112     pq_heap_s *pq = (pq_heap_s*)_pq;
113     pq->cost[pq->heapIDX[label]] = cost; // update the cost, but heap property is not satisfied
114     // assert(pq->is_heap(pq, 0));
115
116     pq_heap_s *pqBackup = (pq_heap_s*)pq_alloc(pq->size); //create backup of the heap
117     pqBackup->len = pq->len;
118     for (int i = 0; i < pq->len; ++i) { // backup the heap
119         pqBackup->cost[i] = pq->cost[i]; //just cost and labels
120         pqBackup->label[i] = pq->label[i];
121     }
122     pq->len = 0; //clear all vertices in the current heap
123     for (int i = 0; i < pqBackup->len; ++i) { //create new heap from the backup
124         pq_push(pq, pqBackup->label[i], pqBackup->cost[i]);
125     }
126     pq_free(pqBackup); // release the queue
127     ret = true;
128     return ret;
129 }
    
```

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`
- Vytvoříme graf o 1 mil. uzlů (a cca 3 mil. hran) v souboru `/tmp/g`.
 - `./bin/dijkstra -c 1000000 /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 time ...1179ms	3 Load time ...120ms
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/LQUG8EeqLM>

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

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 time ...1252 ms	3 Load time ...223 ms	3 Load time ...235 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 načítání a ukládání z HW09.
 - Další urychlení lze dosáhnout lepší organizací paměti a datovými strukturami.
- Jediný zásadní požadavek je implementace rozhraní dle `lec11/dijkstra.h`.*

Shrnutí přednášky

Část III

Část 3 – Zadání 10. domácího úkolu (HW10)

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)

Část V Appendix

Hledání nejkratší cesty v grafu

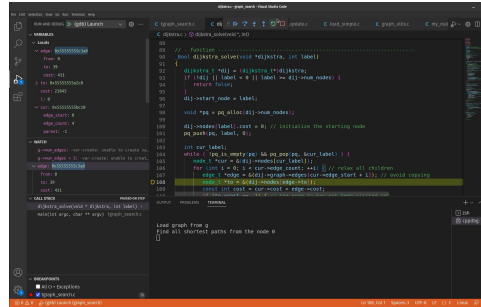
```

BOB36PRP-lev11-codes ls
bin                priority_queue-linked_list
graph_ls           queue
graph_search       solver.txt
graph_search_array solution.txt
priority_queue_array stack
BOB36PRP-lev11-codes ls bin
dijkstra-flood     dijkstra-lev2     dijkstra.exe
dijkstra-flood2    dijkstra-lev2     dijkstra.exe
dijkstra-heap     dijkstra-heap     dijkstra.exe
dijkstra-heap2    dijkstra-heap     dijkstra.exe
dijkstra-lev      dijkstra-lev      dijkstra.exe
BOB36PRP-lev11-codes cd graph_search
graph_search gmake
clang -o dijkstra.c -O2 -g -pedantic -Wall -Werror -o dijkstra.o
clang -o my_malloc.c -O2 -g -pedantic -Wall -Werror -o my_malloc.o
clang -o graph_utils.c -O2 -g -pedantic -Wall -Werror -o graph_utils.o
clang -o pq_heap-no_update.c -O2 -g -pedantic -Wall -Werror -o pq_heap-no_update.o
clang -o load_simple.c -O2 -g -pedantic -Wall -Werror -o load_simple.o
clang -o tgraph_search.c -O2 -g -pedantic -Wall -Werror -o tgraph_search.o
clang -o tgraph_search_time.c -O2 -g -pedantic -Wall -Werror -o tgraph_search_time.o
clang dijkstra.o my_malloc.o graph_utils.o pq_heap-no_update.o load_simple.o tgraph_search.o tgraph_search_time.o -la -o tgraph_search
graph_search

```

<https://youtu.be/LQUGP8EeqLM>

Příklad ladění krokováním



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