

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

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 1 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

Prioritní fronta

- Fronta
 - První vložený prvek je první odebraný prvek
- Prioritní fronta
 - Některé prvky jsou při vyjmutí z fronty preferovány
 - Některé vložené objekty je potřeba obslužit náležitěji, např. fronta pacientů u lékaře.*
 - Operace **pop()** odebírá z fronty prvek s nejvyšší prioritou
 - Vrchol fronty je prvek s nejvyšší prioritou.*
 - Alternativně též prvek s nejnižší hodnotou*
- Rozhraní prioritní fronty může být identické jako u běžné fronty, avšak specifikace upřesňuje chování dílčích metod

FIFO

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 5 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

Prioritní fronta spojovým seznamem 1/4

- Ve funkci **push()** přidáme pouze nastavení priority
- ```
int queue_push(void *value, int priority, queue_t *queue)
{
 ...
 if (new_entry) { // fill the new_entry
 new_entry->value = value;
 new_entry->priority = priority;
 }
 ...
}
```
- lec11/priority\_queue.c

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 9 / 57

## Přehled témat

- Část 1 – Prioritní fronta (Halda)
  - Popis
  - Prioritní fronta spojovým seznamem
  - Prioritní fronta polem
  - Halda
- Část 2 – Příklad využití prioritní fronty v úloze hledání nejkratší cesty v grafu
  - Popis úlohy
  - Návrh řešení
  - Implementace pq haldou s **push()** a **update()**
  - Příklad implementace
- Část 3 – Zadání 10. domácího úkolu (HW10)

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 2 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

## Prioritní fronta – specifikace rozhraní

- Prioritní frontu můžeme implementovat různě složitě a také s různými výpočetními nároky, např.
  - Polem nebo spojovým seznamem s modifikací funkcí **push()** nebo **pop()** a **peek()**
    - Základní implementace fronty viz předchozí přednáška.*
    - Například tak, že ve funkci **pop()** a **peek()** projdeme všechny dosud vložené prvky a najdeme prvek nejprioritnější
    - S využitím pokročilé datové struktury pro efektivní vyhledání prioritního prvku (halda)
- Prioritní prvek může být ten s nejmenší hodnotou, pak
  - Metody **pop()** a **peek()** vrací nejmenší prvek dosud vložený do fronty
  - Hodnoty prvků potřebujeme porovnávat, proto potřebujeme funkci pro porovnávání prvků
    - Obecně můžeme realizovat například ukazatelem na funkci*

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 6 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

## Prioritní fronta spojovým seznamem 2/4

- **peek()** lineárně prochází seznam a vybere prvek s nejnižší prioritou
- ```
void* queue_peek(const queue_t *queue)
{
    void *ret = NULL;
    if (queue && queue->head) {
        ret = queue->head->value;
        int lowestPriority = queue->head->priority;
        queue_entry_t *cur = queue->head->next;
        while (cur != NULL) {
            if (lowestPriority > cur->priority) {
                lowestPriority = cur->priority;
                ret = cur->value;
            }
            cur = cur->next;
        }
    }
    return ret;
}
```
- lec11/priority_queue.c

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 10 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

Část I

Část 1 – Prioritní fronta (Halda)

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 3 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

Prioritní fronta – příklad rozhraní

- V implementaci spojového seznamu upravíme funkce **peek()** a **pop()**
 - Využijeme přímo kód lec10/queue_linked_list.h a lec10/queue_linked_list.c*
- Prvek fronty **queue_entry_t** rozšíříme o položku určující priority
 - Alternativně můžeme specifikovat funkce porovnání datových položek*

```
typedef struct entry {
    void *value;
    // Nová položka
    int priority;
    struct entry *next;
} queue_entry_t;

typedef struct {
    queue_entry_t *head;
    queue_entry_t *end;
} queue_t;

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

lec11/priority_queue.h
```

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 8 / 57

Popis Prioritní fronta spojovým seznamem Prioritní fronta polem Halda

Prioritní fronta spojovým seznamem 3/4

- Podobně **pop()** lineárně prochází seznam a vybere prvek s nejnižší prioritou, je však nutné zajistit propojení seznamu po odebrání prvku
- ```
void* queue_pop(queue_t *queue)
{
 void *ret = NULL;
 if (queue->head) { // having at least one entry
 queue_entry_t* cur = queue->head->next;
 queue_entry_t* prev = queue->head;
 queue_entry_t* best = queue->head;
 queue_entry_t* bestPrev = NULL;
 while (cur) {
 if (cur->priority < best->priority) {
 best = cur; // update the entry with
 bestPrev = prev; // the lowest priority
 }
 prev = cur;
 cur = cur->next;
 }
 ...
 }
}
```
- lec11/priority\_queue.c
- Proto si při procházení pamatujeme předchozí prvek **prev**

Jan Faigl, 2016 B0B36PRP – Přednáška 11: Úvod do verzovacích systémů 11 / 57

## Prioritní fronta spojovým seznamem 4/4

- Po nalezení největšího (nejmenšího) prvku propojíme seznam

```
void* queue_pop(queue_t *queue)
{
 ...
 while (cur) { ... } // Finding the best entry
 if (bestPrev) { // linked the list after
 bestPrev->next = best->next; // best removal
 } else { // selected is the head
 queue->head = queue->head->next;
 }
 ret = best->value; //retrieve the value
 if (queue->end == best) { //update the list end
 queue->end = bestPrev;
 }
 free(best); // release queue_entry_t
 if (queue->head == NULL) { // update end if last
 queue->end = NULL; // entry has been
 } // popped
}
return ret;
lec11/priority_queue.c
```

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 12 / 57

## Prioritní fronta spojovým seznamem – příklad použití 1/2

- Inicializaci fronty provedeme polem textových řetězců a priorit

```
queue_t *queue;
queue_init(&queue);
char *values[] = { "2nd", "4th", "1st", "5th", "3rd" };
int priorities[] = { 2, 4, 1, 5, 3 };
const int n = sizeof(priorities) / sizeof(int);
for (int i = 0; i < n; ++i) {
 int r = queue_push(values[i], priorities[i], queue);
 printf("Add %2i entry '%s' with priority '%i' to the queue\n",
 i, values[i], priorities[i]);
 if (r != QUEUE_OK) {
 fprintf(stderr, "Error: Queue is full!\n");
 break;
 }
}
printf("\nPop the entries from the queue\n");
while(!queue_is_empty(queue)) {
 char* pv = (char*)queue_pop(queue);
 printf("%s\n", pv);
 // Do not call free(pv);
}
queue_delete(&queue);
lec11/demo-priority_queue.c
```

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 13 / 57

## Prioritní fronta spojovým seznamem – příklad použití 2/2

- Hodnoty jsou neuspořádané a očekáváme jejich uspořádaný výpis při odebírání funkcí `pop()`
- ```
char *values[] = { "2nd", "4th", "1st", "5th", "3rd" };
int priorities[] = { 2, 4, 1, 5, 3 };
...
while(!queue_is_empty(queue)) {
    // Do not call free(pv);
}
```
- V tomto případě nevoláme `free()` neboť vložené textové řetězce jsou textovými literály *Narozdil od příkladu v 11. přednášce!*

- Příklad výstupu (v tomto případě preferujeme nižší hodnoty):

```
make && ./demo-priority_queue
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          lec11/priority_queue.h, lec11/priority_queue.c
4th
5th          lec11/demo-priority_queue.c
```

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 14 / 57

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 lec10/queue_array.h,
a lec10/queue_array.c*

```
typedef struct {
    void **queue; // Pole ukazatelů na jednotlivé prvky
    int *priorities; // Pole hodnot priorit jednotlivých prvků
    int count;
    int start;
    int end;
} queue_t;
```

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

Viz snímek 8

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 16 / 57

Prioritní fronta polem 1/3

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

```
int queue_push(void *value, int priority, queue_t *queue)
{
    if (queue->count < MAX_QUEUE_SIZE) {
        queue->queue[queue->end] = value;

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

        queue->end = (queue->end + 1) % MAX_QUEUE_SIZE;
        queue->count += 1;
    } else {
        ret = QUEUE_MEMFAIL;
    }
    return ret;
}
lec11/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()`

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 17 / 57

Prioritní fronta polem 2/3

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

```
static int getEntry(const queue_t *queue)
{
    int ret = -1;
    if (queue->count > 0) {
        for (int cur = queue->start, 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.c
```

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 18 / 57

Prioritní fronta polem 2/3

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

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

- Ve funkci `pop()` musíme zajistit zaplnění místa, pokud je odebírán prvek z prostředka fronty (pole).

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

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 19 / 57

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.h, lec11/priority_queue_array.c
lec11/demo-priority_queue_array.c
```

Jan Faigl, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 20 / 57

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
- Při odebrání (nebo vrácení) nejmenšího prvku v nejhorsím musíme projt všechny položky

- To může být v případě mnoha prvků **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 nejmenšího prvku, však musíme frontu znovu projít a najít nový nejmenší prvek

Alternativně 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, 2016 BOB36PRP – Přednáška 11: Úvod do verzovacích systémů 21 / 57

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**
 - Hodnota každého prvku je menší než hodnota libovolného potomka
 - Každá úroveň haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava

Binární plný strom

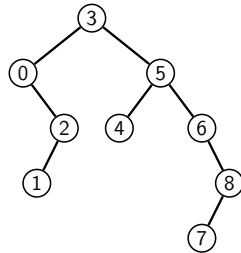
- Prvky mohou být odebrány pouze přes kořenový uzel
- Vlastnost haldy zajišťuje, že kořen je vždy prvek s nejnižším/nejvyšším ohodnocením

V případě binárního plného stromu je složitost procházení následniku úměrná hloubce stromu, která je v případě n prvků úměrná $\log_2(n)$. Složitost operací tak můžeme očekávat nikoliv $O(n)$, ale $O(\log n)$

Binární vyhledávací strom vs halda

Binární vyhledávací strom

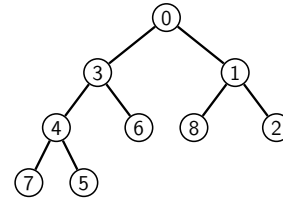
- Může obsahovat prázdná místa
 - Hloubka stromu se může měnit
- Přestože jsme raději, pokud je strom vyvážený. To je však mnohem implementačně náročnější na implementaci haldy.*



Halda

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

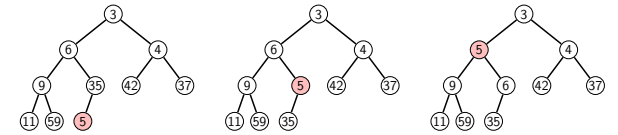
Heap property



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)

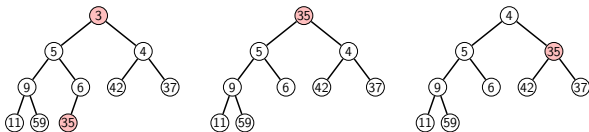
V nejhorším případě prvek „probublá“ až do kořene stromu



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 nejhorším 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 pole, pak poslední prvek v poli je nejpravější list

Prioritní fronta haldou

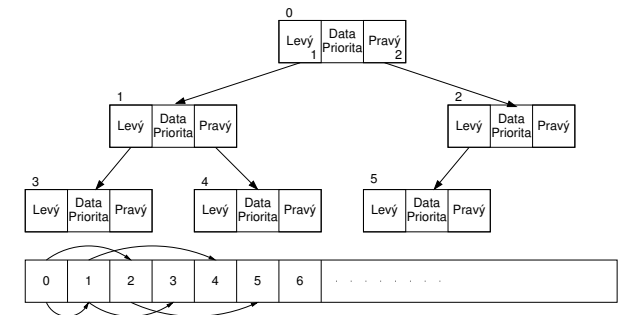
- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti haldy
- Operace peek() má konstantní složitost a nezáleží na počtu prvků ve frontě, nejnižší prvek je vždy kořen
- Operace push() a pop() udržují vlastnost haldy záměnami prvku až do hloubky stromu

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, odtud složitost operace $O(\log(n))$.

Reprezentace binárního stromu polem

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



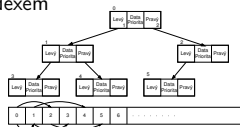
Halda jako binární plný strom reprezentovaný polem

- 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 indexem

levý následník: $i_{lev} = 2i + 1$

pravý následník: $i_{prav} = 2i + 2$

Podobně lze odvodit vztah pro předchůdce



- Kořen stromu reprezentuje nejprioritnější prvek (např. s nejmenší hodnotou nebo maximální prioritou)

Operace vkládání a odebírání prvků

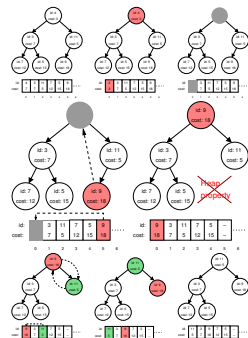
- I v případě reprezentace polem pracují operace vkládání a odebírání identicky
 - Funkce push() přidá prvek jako další prvek v poli a následně propaguje prvek směrem nahoru až je splněna vlastnost haldy
 - Prvek
 - Při odebrání prvku funkcí pop() je poslední prvek v poli umístěn na začátek pole (tj. kořen stromu) a propagován směrem dolů až je splněna vlastnost haldy
- Pouze dochází k vzájemnému zaměňování hodnot na pozicích v poli
 - Z indexu prvku v poli vždy můžeme určit jak levého a pravého následníka, tak i předcházející prvek (rodíče) ve stromové struktuře.
- Hlavní výhodou reprezentace pole je přístup do předem alokovaného bloku paměti
- Všechny prvky můžeme jednoduše projít v jedné smyčce
 - Relativně jednoduše můžeme implementovat funkci ověřující, zdali naše implementace operací push() a pop() zachovávají podmínky haldy.

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
 - Prvek
 - Při odebrání prvku funkcí pop() je poslední prvek v poli umístěn na začátek pole (tj. kořen stromu) a propagován směrem dolů až je splněna vlastnost haldy
- Pouze dochází k vzájemnému zaměňování hodnot na pozicích v poli
 - Z indexu prvku v poli vždy můžeme určit jak levého a pravého následníka, tak i předcházející prvek (rodíče) ve stromové struktuře.
- Hlavní výhodou reprezentace pole je přístup do předem alokovaného bloku paměti
- Všechny prvky můžeme jednoduše projít v jedné smyčce
 - Relativně jednoduše můžeme implementovat funkci ověřující, zdali naše implementace operací push() a pop() zachovávají podmínky haldy.

Příklad volání pop()

- Halda je reprezentovaná binární polem
- Nejmenší prvek je kořenem stromu
- Voláním pop() odebíráme kořen stromu
- a na jeho místo umístíme poslední prvek
- Strom však nesplňuje podmínku haldy
- Pro provedeme záměnu s následníky
- 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

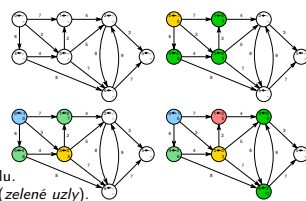


Levý potomek prvku haldy na pozici i je $2i + 1$, pravý potomek je na pozici $2i + 2$

Obdobně postupujeme při push() záměny však provádíme směrem nahoru a z indexu prvku určujeme předchůdce (dělením 2)

Dijkstrův algoritmus

- Nechť graf má pouze kladné ohodnocení hran, pak pro každý uzel
 - nastavíme aktuální cenu nejkratší cesty z výchozího uzlu
 - dále 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 ceny následníků
 - Následně vybereme takový uzel
 - Již do něj existuje nějaká cesta z výchozího uzlu
 - Má aktuálně nejnižší ohodnocení
 - Postup opakujeme dokud existuje nějaký uzel,
 - do kterého vede cesta z výchozího uzlu.
 - Tj. má již ohodnocení a předchůdce (zelené uzly).



Ohodnocení uzlů se může pouze snižovat, cena hran je nezáporná. Tzn. nemůže existovat kratší cesta.

Vstupní graf, reprezentace grafu a řešení

- Graf je zadán jako seznam hran v souboru, který můžeme načíst funkcí load_graph_simple() z lec09/load_simple.c
- Graf je seznam hran

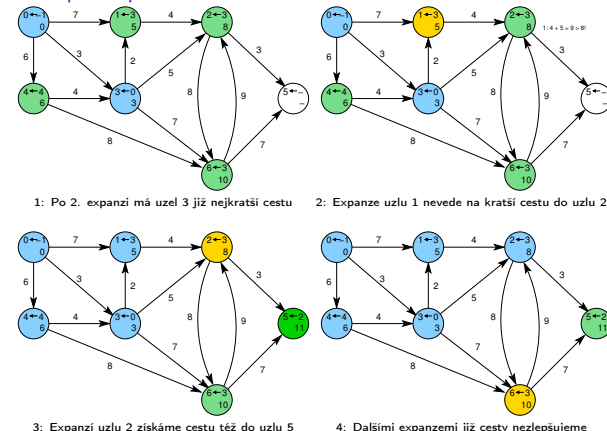

```
typedef struct {
    int from;
    int to;
    int cost;
} edge_t;
} graph_t;
```
- Navíc využijeme toho, že jsou hrany uspořádané


```
typedef struct {
    int edge_start;
    int num_edges;
    int parent;
    int cost;
} node_t;
```
- Pro vlastní řešení potřebujeme u každého uzlu uložit cenu nejkratší cesty (cost) a předcházející uzel na nejkratší cestě parent

Část II

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

Příklad postupu řešení



Datová reprezentace

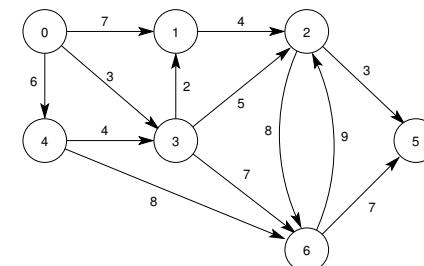
- Řešení implementujeme v modulu dijkstra
- Všechny potřebné datové struktury implementujeme jako strukturu dijkstra_t


```
typedef struct {
    graph_t *graph;
    node_t *nodes;
    int num_nodes;
    int start_node;
} dijkstra_t;
```
- Pro alokaci použijeme malloc(), allocate_graph() a inicializujeme položky struktury na výchozí hodnoty


```
dijkstra_t *dij = (dijkstra_t*)malloc(sizeof(dijkstra_t));
dij->nodes = NULL;
dij->num_nodes = 0;
dij->start_node = -1;
dij->graph = allocate_graph();
```

Hledání nejkratší cesty v grafu

- Uzly grafu mohou reprezentovat jednotlivá místa
- Hrany pak reprezentují cestu jak se mezi místy pohybovat
- Ohodnocení (cena) hrany pak může například odpovídat náročnosti pohybu mezi dvě sousedními uzly
- Cílem je nalézt nejkratší cestu z nějakého konkrétního uzlu (0) do všech ostatních uzlů



Řešení úlohy hledání nejkratších cest v grafu

Řešení úlohy se skládá z

- Vstupních dat (grafu) – paměťová reprezentace a načtení hodnot
 - Vstupní graf je zadán jako seznam hran
 - Dalším vstupem je výchozí uzel
- Výstupních dat (nejkratší cesty) – paměťová reprezentace a uložení (výpis)
 - Všechny nejkratší cesty vypíšeme jako seznam vrcholů s cenou (délkou) nejkratší cesty a bezprostředním předchůdcem (indexem) uzlu na nejkratší cestě
- Algoritmu hledání cest – Dijkstrův algoritmus
 - Algoritmus je relativně přímočarý v každém kroku expandujeme uzel s aktuálně nejkratší cestou z výchozího uzlu

Načtení grafu a inicializace uzlů 1/2

- Hrany načteme např. funkcí load_graph_simple()
 - Dále potřebujeme zjistit počet vrcholů
- Alokujeme paměť pro uzly a nastavíme (bezpečně) 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 začíná od 0 proto +1
dij->nodes = (node_t*)malloc(sizeof(node_t) * dij->num_nodes);
for (int i = 0; i < dij->num_nodes; ++i) {
 dij->nodes[i].edge_start = -1;
 dij->nodes[i].num_edges = 0;
 dij->nodes[i].parent = -1; // pokud neexistuje indikujeme -1
 // pro cenu volíme -1 ve výpise bude kratší než 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].num_edges += 1; // increase no. of edges
}
```

## Hledání nejkratších cest

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

```
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->num_edges; ++i) { // všechny hrany z uzlu
 edge_t *edge = &(dij->graph->edges[cur->edge_start + i]);
 node_t *child = &(dij->nodes[edge->to]);
 const int cost = cur->cost + edge->cost;
 if (child->parent == -1) {
 child->cost = cost;
 child->parent = cur_label;
 pq_push(pq, edge->to, cost);
 } else if (cost <= child->cost) { // uzel již v pq, proto
 child->cost = cost; // testujeme cost
 child->parent = cur_label; // a případně aktualizujeme
 pq_update(pq, edge->to, cost); // odkaz (parent) a pq
 }
 } // smyčka přes všechny hrany z uzlu cur_label
} // prioritní fronta je prázdná
pq_free(pq); // uvolníme paměť
```

## Zápis řešení

- Zápis řešení do souboru můžeme implementovat jednoduchým výpisem do souboru

```
_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;
 }
 }
 return ret;
}
```

## Příklad použití

- Základní implementace uvedeného hledání cest je dostupná v `lec11/graph_search`

- Vytvoříme graf `g` programem `tdijkstra` např. o max 1000 vrcholech,

```
./tdijkstra -c 1000 g
```

- Proram 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
```

## Prioritní fronta 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 a<sup>2</sup> n<sup>2</sup> hran*

- Proto pro prioritní fronty (haldy) implementujeme funkci `update()` a tím zaručíme, že ve frontě bude nejvýše tolik prvků, kolik je vrcholů

- Můžeme tak snadno implementovat prioritní frontu haldou reprezentovanou v poli

*Získáme tak složitost operací O(logn)*

- Pro efektivní implementaci funkce `update()` však potřebujeme získat pozici daného uzlu v haldě

- V případě hledání nejkratších cest, se délka cesty do uzlu může pouze snižovat
- Proto se aktualizovaných „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 prostředka 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.

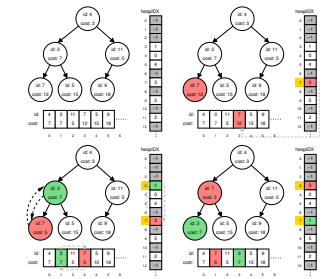
*Zavoláme update(id=7, cost=5)*

- 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.



Princip totožný, jen kromě samotné haldy ještě manipulujeme s dalším strukturou—polem s indexy `heapIDX`

## Prioritní fronta pro Dijkstraův algoritmus

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

```
void *pq_alloc(int size);
void pq_free(void *heap);
_Bool pq_is_empty(const void *heap);
_Bool pq_push(void *heap, int label, int cost);
_Bool pq_update(void *heap, int label, int cost);
_Bool pq_pop(void *heap, int *oLabel);
```

- Jedná o relativně obecný předpis, který neklade zvláštní požadavky na vnitřní strukturu

*V balíku je rozhraní implementované v modulu `pq_array-linear`, který obsahuje implementaci prioritní fronty s lineární složitostí*

- Poslední domácí úkol HW10 je zaměřen na implementaci rozhraní `pq_heap.h` haldou, která bude mít složitost odpovídající  $O(\log n)$ .

## Lineární prioritní fronta vs efektivní implementace

- Ukázková implemetace v `lec11/graph_search`, je sice funkční, pro velké grafy je však výpočet pomalý

- Například pro graf s 1 mil. vrcholů trvá načtení, nalezení všech nejkratší cest a uložení výsledku přibližně 120 sekund

```
./tdijkstra -c 1000000 g Intel Skylake@3.3GHz
/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
```

- Referenčnímu programu `tdijkstra` pouze cca 1 sekundou

```
Těž 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
```

**Základní verze řešení HW10 nesmí být více než 10× pomalejší než referenční program.**

## Další možnosti urychlení programu

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

```
./tgraph_search-time g s 2>/dev/null
Load time ...1008ms
Solve time ...118808ms
Save time ...311ms
Total time ...120127ms

./tdijkstra -v g s.ref
Dijkstra version 2.3.3
Load time ...223ms
Init time ...7ms
Find time ...707ms
Solve time ...715ms
Save time ...106ms
Total time ...1044ms

lec11/graph_search-time.c
```

- Soutěž v rychlosti programu – prvních 20 nejrychlejších programů si rozdělí v součtu 50 extra bodů

Část III  
Část 3 – Zadání 10. domácího úkolu  
(HW10)

Zadání 10. domácího úkolu HW10

- 
- Termín odevzdání: 07.01.2017, 23:59:59 PST  
*PST – Pacific Standard Time*

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)
- **Příště: Systémy pro správu verzí.**