Prioritní fronta a příklad použití v úloze hledání nejkratších cest

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 11

B0B36PRP – Procedurální programování

Přehled témat

- Část 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

  Přístup řešení

  Základní implementace (s lineárním vyhledáváním)

  Urychlení hledání prioritní frontou (haldou)

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

Část I

Část 1 – Prioritní fronta (Halda)

Prioritní fronta

- Fronta
  - První vložený prvek je první odebraný prvek

    FIFO
- Prioritní fronta
  - Některé prvky jsou při vyjmutí z fronty preferovány

    Některé vložené objekty je potřeba obsloužit naléhavěji, 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

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.
  - 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
  - Prvky potřebujeme porovnávat, proto potřebujeme funkci pro porovnávání prvků

    Můžeme realizovat například ukazatelem na funkci.

Prioritní fronta – příklad rozhraní

- Prioritní frontu můžeme realizovat spojovým seznamem, ve kterém upravíme funkce `peek()` a `pop()`

Implementace vychází z `lec10/queue_linked_list.h`,

a `lec10/queue_linked_list.c`

- Prvek fronty `queue_entry_t` rozšíříme o položku určující prioritu

Alternativně můžeme specifikovat funkce porovnání datavých položek

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

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

- Rozhraní funkcí je identické frontě až na specifikaci priority při vložení prvku do fronty

```
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`

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

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

Prioritní fronta spojovým seznamem 3/4

- pop() lineárně prochází seznam a vybere prvek s nejnižší prioritou

```
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

Prioritní fronta spojovým seznamem 4/4

- Navíc je nutné zajistit propojení seznamu po vyjmutí prvku

```
void* queue_pop(queue_t *queue)
{
    ...
        while (cur) {
            ...
        }
        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; //retrive 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

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, "Errro: 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

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 *Narozdíl 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
4th
5th
```

lec11/priority_queue.h, lec11/priority_queue.c

lec11/demo-priority_queue.c

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 s implementací spojovým seznamem

Viz snímek 8

Prioritní fronta polem 1/2

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

```
int queue_push(void *value, int priority, queue_t *queue)
{  ...
      queue->queue[queue->end] = value;
      queue->priorities[queue->end] = priority;
   ...
```

- Funkce `peek()` a `pop()` potřebují prvek s nejnižšší prioritou, který nalezneme lineární procházení pole priorit

```
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

Prioritní fronta polem 2/2

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

Případnou mezeru zaplníme prvkem ze startu

```
void* queue_pop(queue_t *queue)
{
   void *ret = NULL;
   int bestEntry = getEntry(queue);
   if (bestEntry >= 0) { // entry has been found
      ret = queue->queue[bestEntry];
      if (bestEntry != queue->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;
}
```

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

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 nejhorším případě musíme projít celý seznam
- To může být v řadě praktických aplikací nedostatečné a raději bychom chtěli „udržovat" nejmenší prvek připravený
  - Můžeme to například udělat zavedením položky **head**, ve které bude aktuálně nejmenší 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.

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

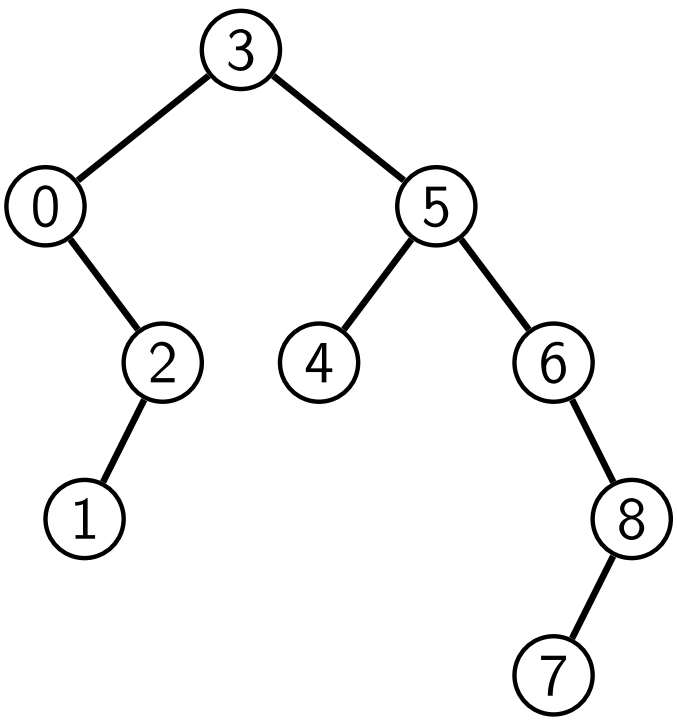
Binární plný strom

  - Prvky mohou být odebrány pouze přes kořenový uzel
- Vlastnost haldy zajišťuje, že kořen je vždy nejmenší prvek
- Nejmenší prvek je tedy první hodnota, kterou z haldy odebereme

Binární vyhledávací strom a halda

Binární vyhledávací strom

- Může obsahovat prázdná místa
- Hloubka stromu se může měnit
- Strom můžeme procházet

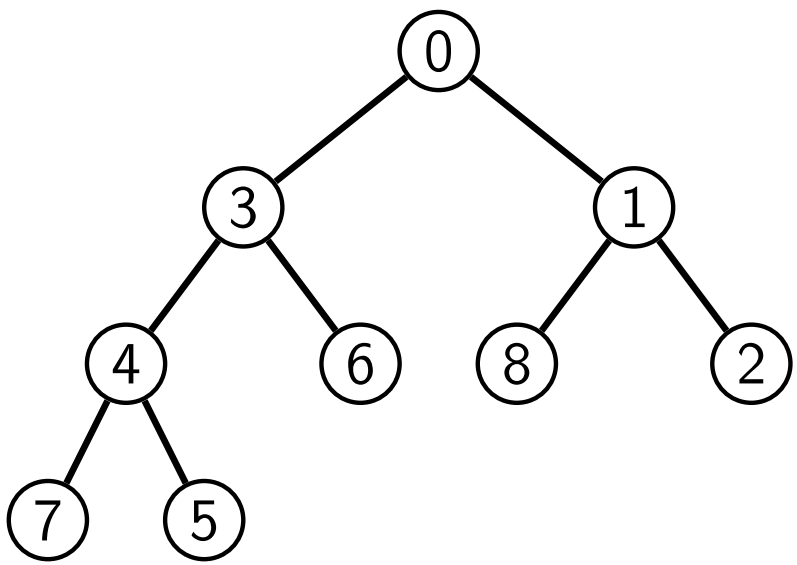


Halda

- Lze využít binární plný strom

  Hloubka stromu vždy $\lfloor \log_2(n) \rfloor$
- Kořen stromu je vždy prvek s nejmenší hodnotou
- Splňuje vlastnost haldy

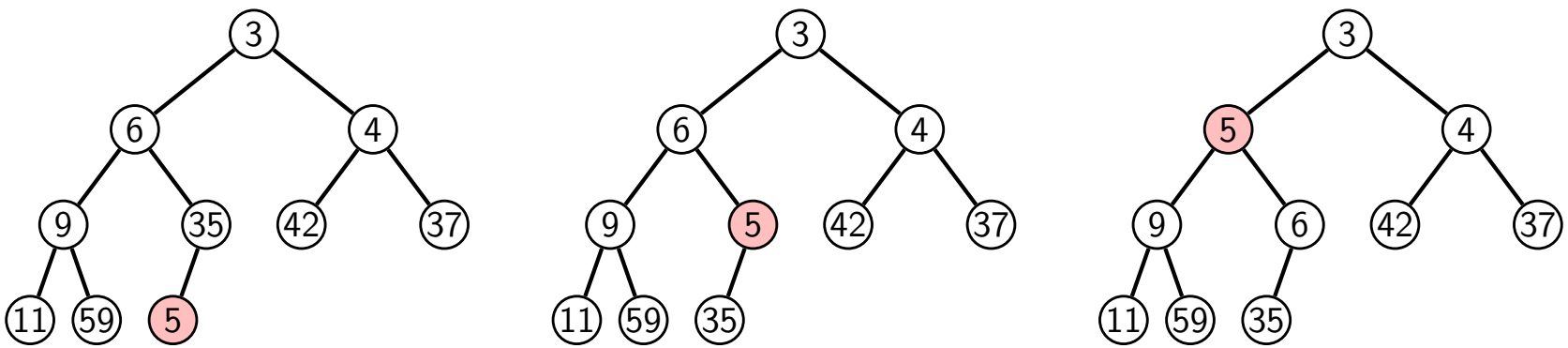
Heap property



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)

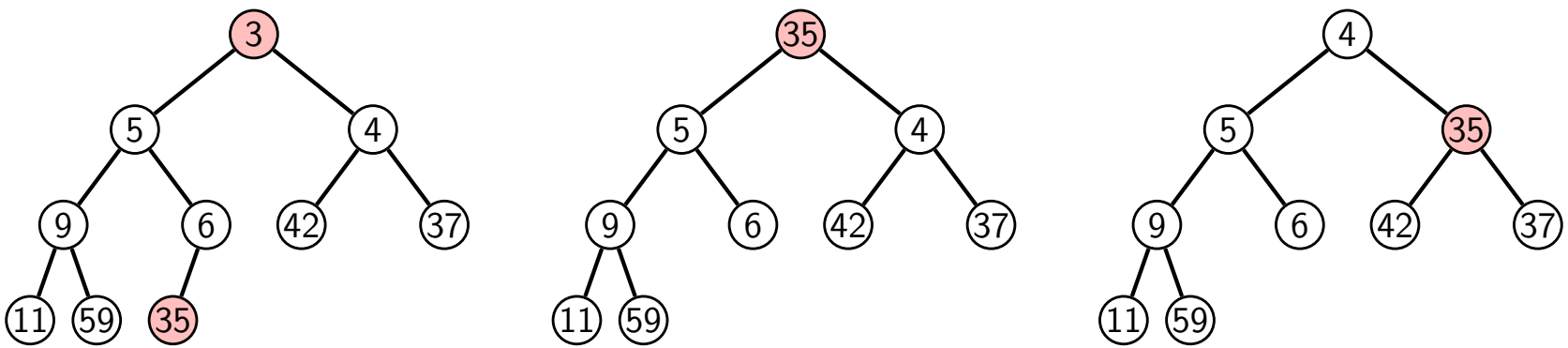
V nejhorším případě prvek „probublá" až do kořene stromu



Halda – odebrání prvku **pop()**

- Při operaci **pop()** odebereme kořen stromu
- Prázdné místo nahradíme nejpravějším listem (uzlem v poslední úrovni)
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s potomkem a postup opakujeme

V nejhorším případě prvek „probublá" až do listu stromu



Prioritní fronta haldou

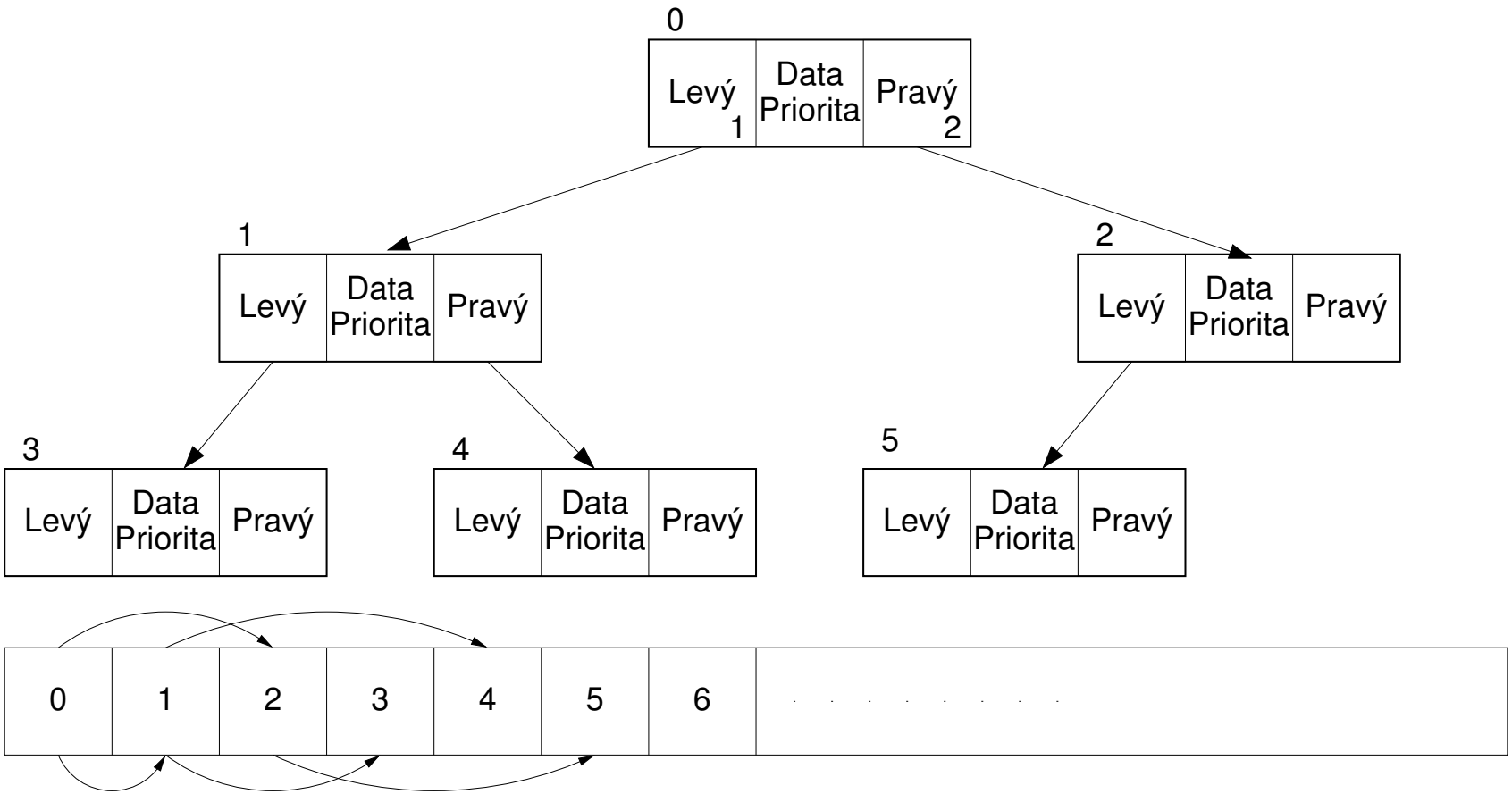
- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti **haldy**
- Operace **peek()** má konstantní složitost a nezáleží na počtu prvků ve frontě, nejmenší prvek je vždy kořen

  Asymptotická složitost v notaci velké O je $O(1)$.
- Operace **push()** a **pop()** udržují vlastnost haldy záměnami prvku až do hloubky stromu

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

Reprezentace binárního stromu polem

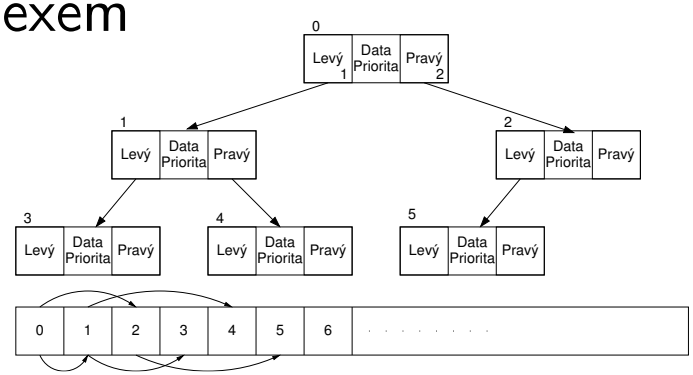
- Pokud dopředu víme jaký bude maximální počet prvku v haldě, můžeme haldu (jako binární strom) reprezentovat v poli



Halda jako binárního strom reprezentovaný pole

- Pro definovaný maximální počet prvků v haldě, můžeme haldu (jako binární plný strom) reprezentovat polem
- Binární strom musí být tzv. `plný binární strom`, tj. všechny vrcholy na úrovni rovné hloubce stromu jsou 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 hodnotu nebo maximální prioritou)

Operace vkládání a odebírání prvků

- I v případě reprezentace binárního stromu haldy polem pracují operace vkládání a odebírání identicky
  - Funkce `push()` přidá prvek jako poslední prvek zaplněného pole 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 (tj. kořen stromu) a propagován směrem dolu až je splněna vlastnost haldy
- Pouze dochází k vzájemnému zaměňování hodnot na pozicích v poli
- Hlavní výhodou reprezentace v poli je přístup do předem alokovaného bloku paměti
- Všechny prvky můžeme jednoduše projít v jedné smyčce

Relativně jednouše můžeme implementovat funkci ověřující, zdali naše implementace operací `push()` a `pop()` zachovávají podmínky haldy.

Část II

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

Část III

Část 3 – Zadání 10. domácího úkolu (HW10)

Zadání 10. domácího úkolu HW10

-
- Termín odevzdání: 07.01.2017, 23:59:59 AoE

AoE – Anywhere on Earth

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.