Kapitola 4

Prioritní fronta a řazení heapsort

Co si zde procvičíme?

- Zopakujeme si pojem pojem prioritní fronty a budeme se zabývat rozdíly v implementaci fronty neseřazeným a seřazeným polem
- Odvodíme si základní charakteristiky úplného binárního stromu
- Seznámíme se s maximální a minimální haldou
- Ukážeme si algoritmus pro ověření vlastností haldy a procvičíme si všechny možné případy, které během jeho výpočtu mohou nastat
- Ukážeme si, jak využít haldu pro implementaci priritní fronty
- Naučíme se vytvořit haldu z libovolného pole v čase $\mathcal{O}(n)$
- Seznámíme se s řadícím algoritmem `heapsort`, který haldu využívá a odvodíme si jeho složitost

4.1 Prioritní fronta reprezentovaná polem

Fronta je abstraktní datový typ, kde prvek, který do fronty vstupuje jako první, je jako první zpracován. Fronta je typu FIFO - první dovnitř, první ven. Minimální implementace fronty předpokládá implementaci následujících operací:

- **len** - délka fronty
- **is_empty** - je fronta prázdná?
- **push** - vložení prvku na konec fronty
- **pop** - výběr prvku ze začátku fronty
- **top** - aktuálně první prvek fronty

Pořadí zpracování prvků ve frontě může být ovlivněno prioritou. Prvky pak nejsou zpracovávány v pořadí, v jaké do fronty vstupují, ale v pořadí podle dané priority. V tomto případě se jedná o **prioritní frontu**, kterou se zde budeme zabývat.
Pokud neuvedeme jinak, budeme předpokládat, že nejvyšší prioritu má prvek s nejmenší hodnotou.

Úloha 4.1.1. Uvažujeme prioritní frontu implementovanou neseřazeným polem. Navrhněte a popište algoritmus (program) pro jednotlivé operace fronty a určete jeho složitost. Definujte jednoduché asymptotické horní meze pro jednotlivé operace.

Řešení: Operace `pop` - z pole, které není seřazené, potřebujeme vybrat nejmenší prvek. Ten leží na indexu i a vyhledáme ho v čase $\mathcal{O}(n)$. Tento prvek z fronty odebereme, tj. všechny prvky s indexy $i+1, i+2, \ldots, n$ posuneme o jedno místo. To opět vykonáme v čase $\mathcal{O}(n)$. Celková složitost operace `pop` je tak $\mathcal{O}(n)$.
Podobné lze argumentovat, že operace `len`, `is_empty` a `push` lze realizovat v čase $\mathcal{O}(1)$ a operaci `top` v čase $\mathcal{O}(n)$.

Úloha 4.1.2. Uvažujeme prioritní frontu implementovanou seřazeným polem. Navrhněte a popište algoritmus (program) pro jednotlivé operace fronty a určete jeho složitost.

Řešení: Argumentujte, že operace `len`, `is_empty` a `top` lze realizovat v čase $\mathcal{O}(1)$ a operace `push` a `pop` v čase $\mathcal{O}(n)$.

Úloha 4.1.3. Uvažujeme prioritní frontu implementovanou uspořádaným spojovým seznamem. Určete složitost jednotlivých operací fronty při dané reprezentaci.

Řešení: Operace `is_empty`, `top` a `pop` lze realizovat v čase $\mathcal{O}(1)$ a operace `len` a `push` v čase $\mathcal{O}(n)$.

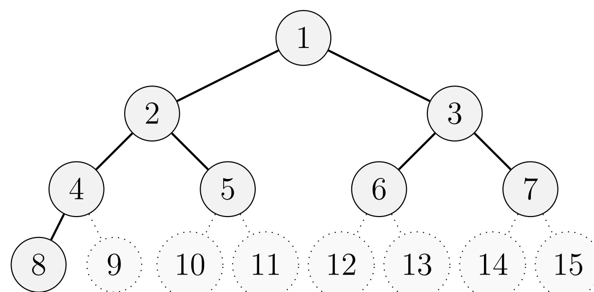
4.2 Úplný binární strom

Úplný binární strom je strom, jehož každý vrchol má 0 nebo 2 potomky. Vrcholy, které mají 0 potomků, nazýváme listy, ostatní vrcholy jsou vnitřní vrcholy stromu.

Úloha 4.2.1. Uvažujeme úplný binární strom hloubky h.

- Jaký je minimální počet vrcholů ve stromě?
- Jaký je maximální počet vrcholů ve stromě?

Řešení: Na obrázku lze vidět, že úplný binární strom hloubky 3 má minimální počet vrcholů 8 a maximální 15. Když to zobecníme, dostaneme minimum 2^h a maximum $2^{h+1} - 1$. Zdůvodněte. Určete počet vrcholů stromu na jednotlivých úrovních.



Úloha 4.2.2. Uvažujeme úplný binární strom s n vrcholy. Jaká je výška tohoto stromu? Zdůvodněte.

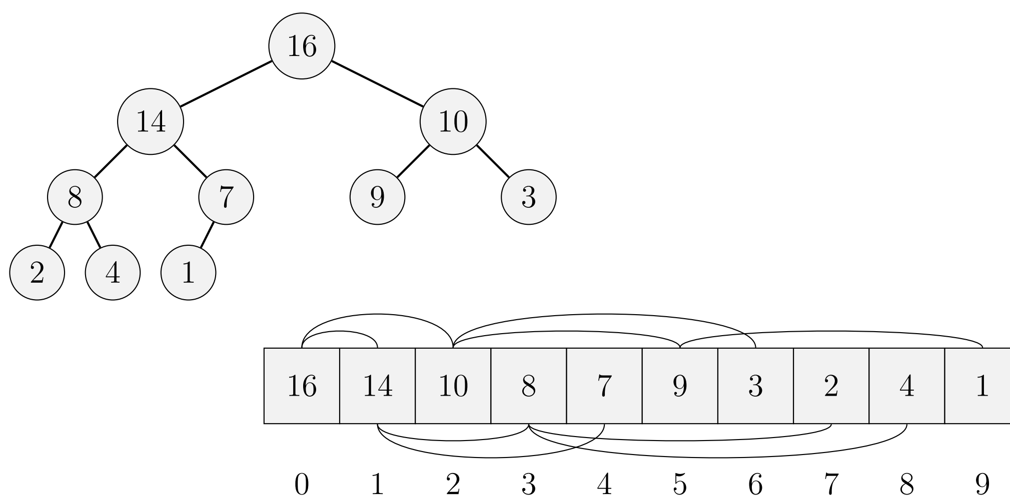
Řešení: $h = \lfloor \log_2 n \rfloor$.

Úloha 4.2.3. Kolik listů má úplný binární strom s n vrcholy?

Návod: Je počet listů úplného binárního stromu větší než počet vnitřních uzlů?

4.3 Maximální a minimální halda

Halda je stromová struktura splňující **vlastnost haldy**.
Dále se budeme zabývat haldou, která je úplným binárním stromem. Její výhodou je možnost implementace pomocí pole - viz obrázek[1]



Kořen stromu je $A[0]$. Pro index i můžeme pak snadno vypočíst indexy rodiče a levého a pravého potomka, a tedy i definovat odpovídající funkce:

$PARENT(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$	`def parent(i):` `    return (i - 1) // 2`
$LEFT(i) = 2i + 1$	`def left(i):` `    return 2 * i + 1`
$RIGHT(i) = 2i + 2$	`def right(i):` `    return 2 * i + 2`

4.3.1 Maximální halda

Maximální halda je stromová struktura, pro kterou je hodnota (klíč) rodiče větší nebo rovna hodnotě potomka (syna). Tj.

$$A[PARENT(i)] \geq A[i]$$

[1]Cormen Thomas H. et al.: *Introduction to Algorithms.* London: MIT Press, 2009, s. 151-161.

Úloha 4.3.1. Dokažte, že v každém podstromu maximální hromady platí, že kořen podstromu obsahuje největší hodnotu, která se v daném podstromu vyskytuje.

Návod: Důkaz sporem.

Úloha 4.3.2. Kde může být v maximální haldě umístěn minimální prvek za předpokladu, že jsou všechny prvky odlišné?
Kde může být v maximální haldě umístěn minimální prvek za předpokladu, že v haldě existuje více stejných minimálních prvků?

Úloha 4.3.3. Jsou následující pole maximální haldou?

- $\{15, 10, 13, 5, 4, 9, 7, 2\}$
- $\{21, 19, 18, 11, 13, 10, 9, 5, 7, 12\}$
- $\{23, 17, 14, 6, 13, 10, 1, 5, 7, 12\}$

Pokud pole maximální haldou nejsou, upravte je tak, aby maximální haldou byly. Minimalizujte počet úprav.

4.3.2 Minimální halda

Minimální halda je stromová struktura, pro kterou je hodnota (klíč) rodiče menší nebo rovna hodnotě potomka (syna). Tj.

$$A[PARENT(i)] \leq A[i]$$

Úloha 4.3.4. Rozhodněte o platnosti následujících výroků a své tvrzení zdůvodněte. Vyhledejte příklady, které tvrzení potvrzují, resp. vyvrací.

- Je seřazené pole minimální haldou?
- Je minimální halda vždy seřazeným polem?

Úloha 4.3.5. Z různých čísel 1 až 10 vytvořte minimální haldu v 10 prvkovém poli tak, aby:

- v prvku pole s indexem 4 byla maximální možná hodnota,
- v prvku pole s indexem 4 byla minimální možná hodnota, tj. nejmenší hodnota, jaká se na indexu 4 může vyskytnout.

4.3.3 Ověření vlastnosti haldy

Vlastnost haldy je definována jako vztah mezi hodnotou (klíčem) rodiče a jeho potomka. Pokud je tento vztah porušený (neplatí definovaná nerovnost), opravíme ho výměnou hodnot rodiče a potomka. Tím ale může dojít k porušení vztahu o úroveň níže, tj. na rovni potomka. Tento problém řeší algoritmus **ověření vlastností haldy** - uvádíme jeho verzi pro maximální haldu.

```
1  def max_heapify(A, length, i):
2      l = left(i)
3      r = right(i)
4      if l < length and A[l] > A[i]:
5          largest = l
6      else:
7          largest = i
8      if r < length and A[r] > A[largest]:
9          largest = r
10     if largest != i:
11         A[i], A[largest] = A[largest], A[i]
12         max_heapify(A, length, largest)
```

Úloha 4.3.6. Upravte zadané pole podle algoritmu pro ověření vlastností maximální haldy.

a) $A = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$, volání funkce `max_heapify(A, len(A), 1)`

b) $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$ a `max_heapify(A, len(A), 2)`

Řešení a): Na obrázku lze vidět postupnou úpravu pole A při opakovaném rekurzivním volání funkce `max_heapify(A, len(A),???)`.

0	1	2	3	4	5	6	7	8	9
16	4	10	14	7	9	3	2	8	1
16	14	10	4	7	9	3	2	8	1
16	14	10	8	7	9	3	2	4	1

Úloha 4.3.7. Jaký efekt má volání `max_heapify(A, len(A), i)`, když $A[i]$ je větší než jeho potomci? Uveďte seznam řádků programu, které se v tomto případě provedou. Jakou hodnotu bude mít proměnná `largest` těsně před koncem výpočtu?

Úloha 4.3.8. Jaký efekt má volání `max_heapify(A, len(A), i)`, když $i > \frac{\texttt{len(A)}}{2}$? Uveďte seznam řádků, které se v tomto případě vykonají.

Úloha 4.3.9. Odhadněte zdola počet rekurzivních volání funkce `max_heapify` při volání `max_heapify(A, len(A), 0)`.

Návod: Halda je binární strom. Vysvětlete, jak počet rekurzivních volání funkce `max_heapify` souvisí s hloubkou stromu $h = \lfloor \log_2 n \rfloor$.

Úloha 4.3.10. Upravte funkci `max_heapify` tak, aby ověřovala vlastnost **minimální** haldy.

4.3.4 Prioritní fronta implementována haldou

Vrátíme se k prioritní frontě a pokusíme se navrhnout algoritmy pro její implementaci pomocí haldy. Vzhledem k tomu, že halda je polem, lze operace `len`, `is_empty` a `top` realizovat v čase $\mathcal{O}(1)$. Definujeme operace `pop` a `push`.

Úloha 4.3.11. Je dán následující algoritmus pro výběr maximálního prvku z haldy. Jeho princip spočívá v odstranění kořene stromu, který je nahrazen posledním prvkem haldy. Funkce `max_heapify` nakonec obnoví vlastnost haldy.

```
1  def pop_heap(A):
2      if len(A) <= 0:
3          return None
4      max = A[0]
5      if len(A) == 1:
6          A.pop()
7      else:
8          A[0] = A.pop()
9          max_heapify(A, len(A), 0)
10     return max
```

a) Je dána prioritní fronta reprezentována haldou $A = [15, 10, 12, 9, 6, 8, 4, 7, 2, 1]$. Určete obsah pole po trojnásobném zavolání funkce `pop_heap`.

b) Určete časovou složitost operace `pop_heap`.

Řešení: Časová složitost je $\mathcal{O}(\log(n))$. Zdůvodněte.

Úloha 4.3.12. Je dán algoritmus pro vložení prvku do prioritní fronty reprezentované haldou. Jeho myšlenka spočívá v přidání nového prvku na konec haldy a obnově vlastností haldy směrem nahoru (viz cyklus `while`).

```
1  def push_heap(A, key):
2      A.append(key)
3      i = len(A) - 1
4      while i > 0 and A[parent(i)] < A[i]:
5          A[i], A[parent(i)] = A[parent(i)], A[i]
6          i = parent(i)
```

a) Ilustrujte výpočet operace `push_heap(A, 16)` na haldě $A = [15, 10, 12, 9, 6, 8, 4, 7, 2, 1]$. Pokračujte operacemi `push_heap(A, 13)` a `push_heap(A, 3)`.

b) Výměna prvků na řádku 5 vyžaduje 3 operace. Zamyslete se nad možností nahradit výměnu prvků jedním přiřazením. Sledujte, jakým způsobem se mění hodnoty prvků pole A ve smyčce `while`.

c) Určete časovou složitost operace `push_heap`.

4.3.5 Vytvoření haldy

Algoritmus `build_max_heap` pro tvorbu haldy z libovolného pole A s využitím funkce `max_heapify` je definován následovně:

```
1  def build_max_heap(A):
2      for i in range(len(A) // 2, -1, -1):
3          max_heapify(A, len(A), i)
```

Úloha 4.3.13. Podle algoritmu `build_max_heap` vytvořte z pole A haldu.

a) $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$

b) $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$

c) $A = [7, 0, 3, 4, 13, 10, 1, 5, 27, 12, 16, 8, 9, 17]$

Úloha 4.3.14. Na základě vlastností haldy zdůvodněte, že výstupem algoritmu `build_max_heap` je halda, která obsahuje pouze prvky pole A.
Zdůvodněte, proč je důležité, aby cyklus `for` na řádku 2 algoritmu běžel od $\lfloor \frac{\texttt{len(A)}}{2} \rfloor$ do 0. Byl by algoritmus správný i pokud by cyklus `for` běžel v opačném sledu, tj. od 0 do $\lfloor \frac{\texttt{len(A)}}{2} \rfloor$?

Návod: Definujte vhodný invariant a dokažte jeho platnost v celém průběhu algoritmu.

Úloha 4.3.15. Ukažte, že v haldě s n vrcholy není počet vrcholů s výškou h větší než $\frac{n}{2^{h+1}}$.

Návod: Celková výška stromu - $\lfloor \log_2 n \rfloor$.
Počet vrcholů na i-té úrovni stromu - 2^i.
Počet vrcholů s výškou h

$$2^{\log_2 n - 1 - h} = \frac{2^{\log_2 n}}{2^{h+1}} = \frac{n}{2^{h+1}}$$

Složitost algoritmu `build_max_heap`

Jednoduchý odhad:

- algoritmus `max_heapify` - $\mathcal{O}(n)$
- opakuje se $\frac{n}{2}$-krát
- složitost není větší než $\mathcal{O}(n \cdot \log(n))$

Přesnější odhad:

- algoritmus `max_heapify` pro vrchol s výškou h - $\mathcal{O}(h)$
- počet vrcholů s výškou h - $\frac{n}{2^{h+1}}$
- sečteme přes všechny výšky stromu

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot \mathcal{O}(h) = \mathcal{O}\left(n \cdot \sum_{h=0}^{\log n} \frac{h}{2^h}\right) \tag{1}$$

$$\sum_{h=0}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = x \left(\frac{1}{1-x}\right)'_{x=\frac{1}{2}} = \left(\frac{x}{(1-x)^2}\right)_{x=\frac{1}{2}} = \left(\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right) = 2 \tag{2}$$

- dosadíme (2) do (1) a dostaneme celkovou složitost algoritmu `build_max_heap`

$$\mathcal{O}\left(n \cdot \sum_{h=0}^{\log n} \frac{h}{2^h}\right) = \mathcal{O}(n \cdot 2) = \mathcal{O}(n)$$

4.4 Heapsort

Algoritmus `heapsort` pro řazení prvků pole A a využitím haldy je definován následovně:

```
1 def heapsort(A):
2     build_max_heap(A)
3     for i in range(len(A) - 1, 0, -1):
4         A[0], A[i] = A[i], A[0]
5         max_heapify(A, i, 0)
```

ÚLOHA 4.4.1. Podle algoritmu `heapsort` seřaďte prvky pole A.

a) $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$

b) $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$

c) $A = [7, 0, 3, 4, 13, 10, 1, 5, 27, 12, 16, 8, 9, 17]$

Jakým způsobem řadí prvky pole algoritmus `heapsort` s využitím maximální haldy - vzestupně nebo sestupně?

ÚLOHA 4.4.2. Projděte jednotlivé body algoritmu `heapsort` a určete paměťové nároky na jeho realizaci. Jaké místo v paměti je potřeba mimo definované pole A?

ÚLOHA 4.4.3. Z algoritmu `heapsort` odhadněte složitost řazení prvků pole A. Závisí složitost algoritmu `heapsort` na tom, zda je posloupnost seřazena vzestupně či sestupně nebo zda není seřazena vůbec?

Řešení: $\mathcal{O}(n \log(n))$

ÚLOHA 4.4.4. Definujeme invariant pro algoritmus `heapsort` - pro jeho část, cyklus `for` na řádcích 3-5:
Část pole $A[0..i-1]$ je maximální haldou a obsahuje i nejmenších prvků pole a část pole $A[i..n]$ je uspořádána a obsahuje $n-i+1$ největších prvků pole.
Dokažte matematickou indukcí platnost daného invariantu a na základě platnosti invariantu argumentujte správnost algoritmu `heapsort`.

ÚLOHA 4.4.5. Upravte všechny části algoritmu `heapsort` tak, aby výsledná posloupnost byla řazena sestupně.