C2142 Návrh algoritmů pro přírodovědce

5. Haldy, vyhledávací stromy

Tomáš Raček

Jaro 2019

Optimální algoritmy pro problém řazení

Připomenutí. Aktuálně známe několik algoritmů se složtostí $O(n^2)$ a jeden optimální se složitostí $O(n \log n)$ – merge sort.

Nevýhoda merge sortu je dodatečná paměť, kterou potřebuje pro svůj výpočet, typicky $O(n)$ na poli.

Zamyšlení. Lze navrhnout optimální algoritmus pro problém řazení s konstantní extrasekvenční prostorovou složitostí?

Idea pro nový řadicí algoritmus. Uvažme datovou strukturu, která poskytuje efektivní operaci pro odebrání minimálního prvku. Jak lze pomocí ní implementovat řazení?

Požadavky na datovou strukturu

Pozorování. Strukturu, která by poskytovala operaci odebrání minimálního prvku v nižším než $O(n)$, nepostavím na základě pole nebo spojového seznamu.

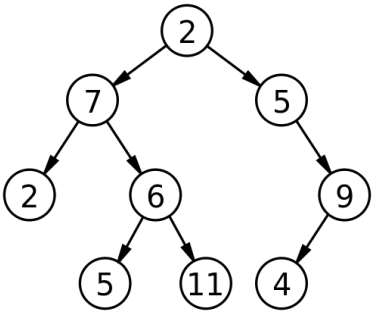
Poznámka. S lineární složitostí vyhledání minima nad polem nebo seznamem dostávám de facto selection sort.

Nápad. Uvažme strukturu, která již v sobě bude obsahovat vhodné uspořádání prvků, které nám umožní provádět operace nad prvky v nejvýše logaritmickém čase. Celkem pro n prvků tedy dostanu nejhůře $O(n \log n)$.

- Toho by mohlo jít dosáhnout, pokud všechny prvky budou „vzdáleny“ od výchozího nejvýše $O(\log n)$.
- Jak takovou strukturu navrhnout?

Binární strom I

Nápad. Uvažme rozšíření spojového seznamu, kdy každý prvek bude obsahovat dva ukazatele na další prvky. Cykly mezi prvky nepovolíme.

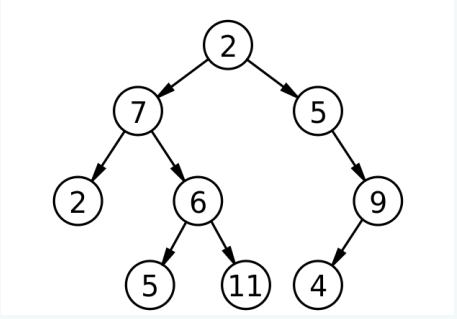


Takovou strukturu nazveme binární strom. Každý uzel (prvek) binárního stromu má nejvýše dva potomky.

Binární strom II

Názvosloví:

- kořen – výchozí prvek stomu, nevedou na něj žádné ukazatele
- list – uzel, který nemá žádné potomky
- větev – poslopnost uzlů od kořene k listu
- výška stromu – délka nejdelší větve

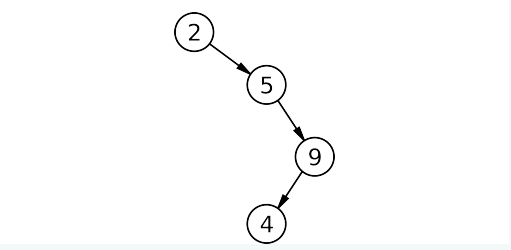


Pozorování. Složitost přístupu k jednotlivým prvkům od kořene bude nejvýše $O(h)$, kde h je výška stromu.

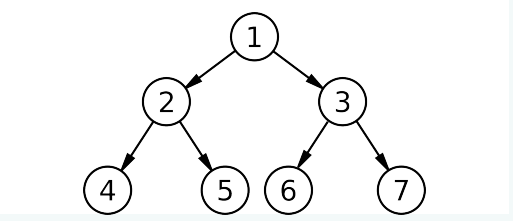
Otázka. Jaká je minimální a maximální výška binárního stromu o n uzlech?

Binární strom III

Nejhorší případ. Degenerovaný strom odpovídající de facto spojovému seznamu. Jeho výška pro n prvků je $O(n)$.



Nejlepší případ. Úplný binární stom má na h úrovních $2^h - 1$ uzlů. Obráceně, úplný binární strom o n uzlech má výšku nejvýše $O(\log_2 n)$.



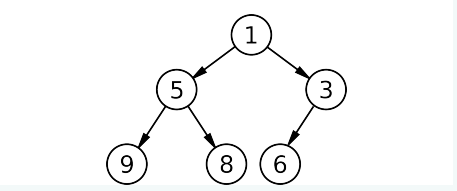
Závěr. Cílem při návrhu pro nás vhodné datové struktury je přiblížit se co nejvíce úplnému binárnímu stromu.

Halda

Halda je datová struktura poskytující efektivní operace pro přidání prvku a odebrání minima.

Binární halda má jako základ binární strom se dvěma vlastnostmi:

1. Pro každý uzel platí, že jeho potomci mají stejnou nebo větší hodnotu.
2. Na všech úrovních s výjimkou poslední je binární strom zcela zaplněn. V poslední úrovni jsou listy zaplňovany zleva doprava.



Poznámka. Druhá vlastnost zaručuje, že výška haldy je logaritmická vzhledem k počtu prvků.

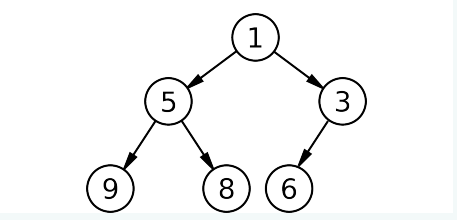
Poznámka. Uvedená definice platí pro tzv. minimovou haldu, která má v kořeni minimum. Analogicky lze definovat maximovou haldu.

Operace nad haldou

Zjištění minima. Minimum haldy je v jejím kořeni, složitost operace je zjevně $\mathcal{O}(1)$.

Přidání prvku. Nový prvek přidám na první volné místo. Musím ovšem zajistit, že bude zachováno uspořádání na větvích.

- pokud je prvek menší než jeho rodič, dojde k jejich prohození
- těchto prohození může být nejvýše $\mathcal{O}(\log n)$, kdy se nový prvek dostane do kořene haldy



Odstranění minima. Vyměním hodnotu kořene a posledního prvku, který pak mohu snadno odstranit.

- pokud je nový kořen větší než jeho potomci, je potřeba menšího z nich s kořenem prohodit a postupovat obdobně níže směrem k listům → celkově až $\mathcal{O}(\log n)$

Heap sort

Heap sort je řadicí algoritmus, který je postaven nad operacemi haldy. Má dvě fáze:

1. Přidání všech prvků do haldy.
2. Opakované odebírání minima.

Složitost Heap sortu

- první fáze má složitost $\mathcal{O}(n)$, druhá pak $\mathcal{O}(n \log n)$
- celkově tedy $\mathcal{O}(n \log n)$
- Heap sort je optimální algoritmus

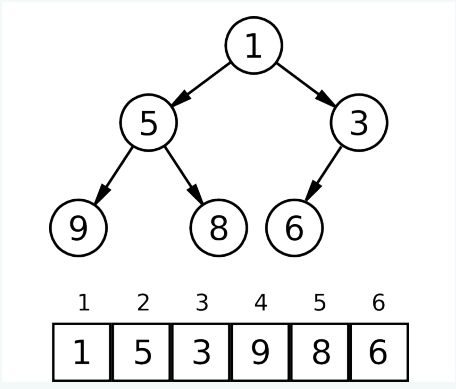
Zamyšlení. Ačkoliv asymptoticky optimální, navržená implementace vyžaduje stále lineární množství paměti pro vytvoření haldy. Jde to udělat lépe?

Implementace haldy v poli

Nápad. Každou binární haldu (obecně i každý zleva zarovnaný binární strom) lze reprezentovat v poli.

Pro každý uzel platí:

- je-li uzel uložen na indexu i, jeho levý potomek je na indexu $2i$
- analogicky pravý potomek pak na indexu $2i + 1$



Heap sort pak spočívá ve vytvoření maximové haldy přeuspořádáním prvků v poli (1. fáze) a následně odebrání všech prvků (2. fáze), které budou tvořit od konce seřazenou posloupnost.

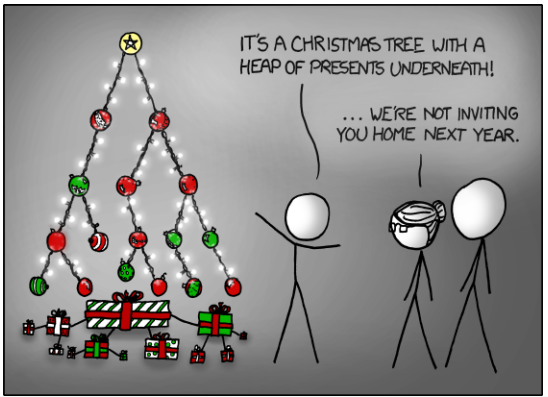
Heap sort – implementace

```
def sift_down(A, start, end):
        root = start
        while True:
                child = 2 * root + 1
                if child >= end:
                        break
                if child + 1 < end and A[child] < A[child + 1]:
                        child += 1
                if A[root] < A[child]:
                        A[root], A[child] = A[child], A[root]
                        root = child
                else:
                        break

def heap_sort(A):
        for i in range(len(A) // 2, -1,  -1):
                sift_down(A, i, len(A))

        for i in range(len(A)):
                A[0], A[len(A) - i - 1] =  A[len(A) - i - 1], A[0]
                sift_down(A, 0, len(A) - i - 1)
```

Haldy v životě informatika (http://xkcd.com/835/)



Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt.

Odbočka: Prioritní fronta

Opakování. Aktuálně umíme implementovat jednoduchou frontu pomocí spojového seznamu nebo pole pevné délky.

Zobecnění. Přiřaďme každému prvku prioritu, která bude určovat v jakém pořadí bude z fronty odstraněn.

Pozorování. Stávající implementace v tomto případě neposkytují lepší než lineární složitost pro alespoň jednu z operací fronty, tedy přidání nebo odebrání prvku.

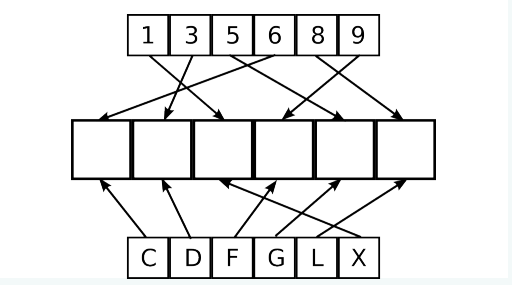
Řešení. Přímočarou implemetací prioritní fronty je binární halda.

- přidání prvku – $\mathcal{O}(\log n)$
- odebrání prvku (odpovídá odebrání minima) – $\mathcal{O}(\log n)$
- pokud je implentována v poli, k přidávání nebo odebírání prvků dochází jen na jeho konci $\rightarrow$ lze využít dynamického pole

Indexy

Shrnutí. V současnosti umíme seřadit data podle zvoleného klíče. Vyhledávat v seřazeném poli lze efektivně pomocí binárního vyhledávání v logaritmickém čase.

- není nutné řadit vlastní (často objemná) data, ale stačí seřadit klíče
- takových uspořádání, tzv. indexů, je možné udržovat více najednou



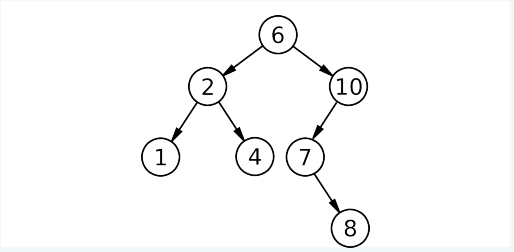
Nevýhoda současného řešení spočívá v obtížné změně velikosti indexů. Přidání nebo odebrání prvku má lineární složitost.

Nápad. Využijme místo pole jako základ indexu binární strom.

Binární vyhledávací strom

Binární vyhledávací strom (BST) je datová struktura založena na binárním stromě, kde pro každý uzel platí, že:

- všechny uzly v jeho levém podstromu mají menší ohodnocení než on sám
- analogicky všechny uzly v pravém podstromu mají ohodnocení větší

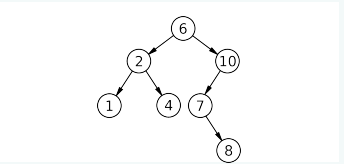


Základní operace nad BST:

- přidání prvku
- odebrání prvku
- vyhledání prvku

Operace nad BST

Vyhledání prvku. Analogie binárního vyhledávání – porovnáváme hledanou hodnotu s ohodnocením aktuálního prvku. Je-li menší, vyhledávání pokračuje v jeho levém podstromu, je-li větší, pak v pravém.



Přidání/odebrání prvku. Nejprve je potřeba nalézt místo, kde k vlastnímu přidání/odebrání dojde. Na konci pak zajistit, že změněný strom je stále BST.

Pozorování. Složitost těchto operací bude záviset na výšce stromu, v nejhorším případě bude tedy lineární (uvažme např. vytvoření BST ze seřazené posloupnosti klíčů).

Závěr. Pro dosažení efektivity je potřeba implementovat operace přidání a odebrání prvku tak, aby udržovaly logaritmickou výšku BST.