Cvičení 3

Příklad 1: Navrhněte způsob, jak implementovat frontu pomocí dvou zásobníků tak, aby amortizovaná složitost operací ENQUEUE (tj. přidání prvku do fronty) i DEQUEUE (tj. odebrání prvku z fronty) byla $\mathcal{O}(1)$.

Poznámka: Předpokládejte, že nemáte k dispozici nic jiného, než dva zásobníky, se kterými můžete provádět následující operace:

- PUSH(S, x) — vložení prvku x do zásobníku S
- POP(S) — odebrání prvku ze zásobníku ze zásobníku S
  (Tento prvek je vrácen jako návratová hodnota. Pokud je zásobník prázdný, nastane chyba.)
- ISEMPTY(S) — test, zda je zásobník S prázdný

Můžete předpokládat, že složitost všech těchto operací na zásobnících je $\mathcal{O}(1)$.

Kromě těchto dvou zásobníků nepřidávejte žádné další datové struktury.

Je možné využít vámi navrženou konstrukci pro implementaci fronty jako funkcionální datové struktury, tj. struktury, kde je možno přidávat nové objekty, ale jednou vytvořené objekty už není možné následně měnit? (Objekty, na které nebude nic ukazovat, budou automaticky odstraněny garbage collectorem.)

Příklad 2: Řekněme, že máme nějakou datovou strukturu, na které vykonáváme posloupnost n operací označených $op_1, \ldots, op_n$. Cena operace op_i (kde $1 \leq i \leq n$) je i právě v těch případech, kde je i mocninou čísla 2 (tj. 2^0, 2^1, 2^2, ...). Ve všech ostatních případech je cena operace op_i rovna 1.

Pomocí amortizované analýzy určete celkovou cenu celé posloupnosti n operací.

Příklad 3: Řekněme, že pracujeme se zásobníkem, jehož velikost nikdy není větší než k. Vykonáváme na něm posloupnost operací PUSH a POP, přičemž po vykonání každých k operací provedeme zálohování obsahu zásobníku, kdy celý jeho obsah zkopírujeme do nějaké jiné datové struktury. Čas nutný pro provedení tohoto kopírování je $\mathcal{O}(s)$, kde s je aktuální velikost zásobníku.

Pomocí amortizované analýzy ukažte, že i se započítáním doby kopírování obsahu zásobníku je časová složitost provedení n operací $\mathcal{O}(n)$.

Příklad 4: ***Halda*** (***heap***) je datová struktura, která je využita například v třídícím algoritmu heapsort nebo v jedné z možných implementací prioritní fronty.

Jedná se o vyvážený binární strom, který je uložen v poli. Pokud zvolíme pole A velikosti n indexované od 1 (tj. pole s prvky $A[1], \ldots, A[n]$), bude tento strom uložen tak, že prvky pole odpovídají vrcholům stromu, přičemž prvek $A[1]$ představuje kořen stromu a pro každé i, kde $1 \leq i \leq n$, jsou levý a pravý potomek vrcholu uloženého v $A[i]$ uloženi v prvcích s indexy $2i$ a $2i+1$ (samozřejmě jen tehdy, pokud jsou indexy těchto potomků menší nebo rovny n; pokud by tyto indexy vycházely větší než n, vrchol s indexem i příslušné potomky mít nebude).

V dané struktuře tedy není třeba ukládat ukazatele na potomky ani na rodiče jednotlivých vrcholů, protože jejich indexy můžeme snadno spočítat z indexu daného prvku pole, kde je uložen příslušný vrchol. (Index rodiče vrcholu s indexem i je $\lfloor i/2 \rfloor$.)

Aby se jednalo o haldu, musí navíc daný strom splňovat to, že pro každý vrchol platí, že v něm uložená hodnota je vždy menší nebo rovna hodnotám uloženým v jeho potomcích.

Haldu je možné vytvořit z nesetříděného pole pomocí algoritmu Create-Heap (viz Algoritmus 1), který očekává jako argumenty pole A indexované od 1 a číslo n udávající velikost tohoto pole.

Algoritmus 1: Vytvoření haldy z nesetříděného pole

```
Create-Heap(A, n):
    i := ⌊n/2⌋
    while i ≥ 1 do
        j := i
        x := A[j]
        while 2 * j ≤ n do
            k := 2 * j
            if k + 1 ≤ n and A[k + 1] < A[k] then
                k := k + 1
            if x ≤ A[k] then break
            A[j] := A[k]
            j := k
        A[j] := x
        i := i − 1
```

Určete co nejpřesněji časovou složitost tohoto algoritmu Create-Heap.

Nápověda: Je očividné, že časová složitost tohoto algoritmu je v $\Omega(n)$. Snadno se také zdůvodní, že tato složitost je v $\mathcal{O}(n \log n)$. Tento horní odhad však není přesný. Použitím amortizované analýzy je možné zdůvodnit, že tato složitost je ve skutečnosti v $\mathcal{O}(n)$.

Příklad 5: Uvažujme problém hledání ***konvexního obalu*** množiny bodů v rovině. Vstupem je zde libovolná konečná množina bodů v rovině, kde každý bod je dán svou x-ovou a y-ovou souřadnicí.

Konvexní obal dané množiny bodů Q je nejmenší konvexní geometrický útvar, který obsahuje všechny body Q. Není těžké si rozmyslet, že v případě konečné množiny bodů tvořené n body bude takovým útvarem m-úhelník, jehož vrcholy jsou podmnožinou bodů z množiny Q.

Úkolem je tedy najít podmnožinu bodů, které tvoří vrcholy daného m-úhelníka, který tvoří konvexní obal dané množiny bodů Q.

Jedním z algoritmů pro řešení tohoto problému je tzv. Graham scan, který je zde popsán pseudokódem jako Algoritmus 2.

a) Navrhněte, jak efektivně implementovat jednotlivé kroky tohoto algoritmu (z nichž některé jsou popsány v daném pseudokódu jen slovně).

b) Analyzuje činnost algoritmu Graham-Scan a určete co nejpřesněji jeho časovou složitost.

Algoritmus 2: Algoritmus pro spočítání konvexního obalu množiny bodů

```
Graham-Scan(Q):
    nechť p_0 je bod s minimální y-ovou souřadnicí
    (a s minimální x-ovou souřadnicí, jestliže existuje víc takových bodů)
    nechť ⟨p_1, p_2, ..., p_m⟩ jsou zbývající body z Q seřazené ve směru
    hodinových ručiček podle úhlů, které svírají vzhledem k bodu p_0
    (pokud více bodů svírá stejný úhel, odstraníme všechny kromě toho, který je od
     p_0 nejvzdálenější)
    Push(S, p_0)
    Push(S, p_1)
    Push(S, p_2)
    for i := 3 to m do
        while úhel tvořený body Next-To-Top(S), Top(S) a p_i nezatáčí doprava do
            Pop(S)
        Push(S, p_i)
    return S
```

Příklad 6: Řekněme, že bychom chtěli pracovat s prvky uloženými stejným způsobem jako v poli, to znamená tak, aby přístup pro čtení či zápis daného prvku určený indexem daného prvku bylo možné provést v konstantním čase. Zároveň bychom chtěli mít možnost přidávat nové prvky na konec příslušného pole tak, aby se velikost tohoto pole dynamicky měnila, přičemž dopředu nebudeme vědět, kolik těchto prvků budeme postupně přidávat.

Operaci přidání nového prvku do dané datové struktury nazvěme Append, tj. Append(A, x) přidá do struktury A prvek x.

Přirozeným způsobem, jak něco takového implementovat, je použít obyčejné pole (jehož velikost není možné měnit), přičemž v tomto poli bude využita jen část prvků. V pomocných proměnných si budeme pamatovat velikost alokovaného pole a počet prvků tohoto pole, které jsou momentálně využity.

Pokud bude v daném poli počet použitých prvků menší než velikost alokovaného pole, stačí při přidávaní nového prvku pouze zvýšit použitých prvků o jedna. V případě, že je pole už zaplněno (tj. alokovaná velikost a počet použitých prvků se rovnají), je třeba alokovat nové větší pole, překopírovat do něj prvky z původního pole a paměť původního pole uvolnit.

Předpokládejte, že časová složitost takové alokace a kopírování je $\Theta(n)$, kde n je počet kopírovaných prvků.

a) Analyzujte případ, kdy bychom vždy alokovali pole, které by bylo větší jen o jeden prvek.

   Jaká by byla v takovém případě celková složitost provedení sekvence n operací Append, pokud bychom začali s prázdným polem (tj. polem velikosti 0).

b) Navrhněte, jak výše popsanou implementaci upravit tak, aby celková složitost provedení n-operací Append byla i v nejhorším případě $\mathcal{O}(n)$, tj. aby amortizovaná cena jednoho přidání prvku pomocí Append byla $\mathcal{O}(1)$.

(Snažte se navrhnout postup, který zaručí, že v každém okamžiku bude dané pole zaplněno vždy alespoň z poloviny.)

c) Řekněme, že bychom chtěli kromě operací Append podporovat i operaci Delete, která vždy odebere jeden prvek z konce pole.

Popište, jak upravit implementaci dané struktury tak, aby platilo, že celková doba běhu libovolné posloupnosti n operací Append a Delete je vždy $\mathcal{O}(n)$.

d) Uvažujme stejné zadání jako v předchozím bodě. Navíc ale budeme požadovat, aby dané pole bylo vždy zaplněno z nějaké dostatečně velké části, tj. aby existovala nějaká vhodně zvolená konstanta c, kde $0 < c < 1$, taková, aby neustále platilo, že $n \geq c \cdot m$, kde n je aktuální počet prvků a velikost alokovaného pole.

Příklad 7: Řekněme, že bychom chtěli navrhnout datovou strukturu, která umožní udržovat prvky rozdělené do několika vzájemně disjunktních množin. V každé z těchto množin bude právě jeden prvek dané množiny představovat reprezentanta dané množiny.
S touto strukturou budeme chtít provádět následující operace:

- Make-Set(x) – vytvoření nové množiny obsahující pouze prvek x (x nesmí být prvkem žádné již vytvořené množiny)
- Union(x, y) – sjednocení množin obsahujících prvky x a y
- Find-Set(x) – nalezení reprezentanta množiny obsahující prvek x

Navrhněte několik přirozených možností, jak takovou strukturu implementovat.
Pro každou z těchto možných implementací analyzujte časovou složitost jednotlivých operací.

Příklad 8: Uvažujme binární čítač, jehož jednotlivé bity jsou uloženy v poli, kde hodnota tohoto čítače je inkrementována pomocí operace Increment, která byla popsána na přednášce.
Řekneme, že bychom chtěli kromě operace Increment používat na čítači i operaci Reset, která čítač vynuluje.
Navrhněte, jak upravit implementaci čítače tak, aby celková časová složitost libovolné posloupnosti n operací Increment a Reset, byla $\mathcal{O}(n)$.

Nápověda: Doplňte čítač o proměnnou, ve které si budeme pamatovat index v poli bitů čítače, od kterého jsou hodnoty v tomto poli neplatné a chápané stejně, jako by obsahovaly samé nuly.