Datenstrukturen und Algorithmen
Heimübung 7

Eli Kogan-Wang (7251030)
David Noah Stamm (7249709)
Daniel Heins (7213874)
Tim Wolf (7269381)

28. Mai 2022

Aufgabe 1

a) Es gibt 2 Möglichkeiten dies zu implementieren:
Möglichkeit 1:
Push = Enqueue. $O(1)$

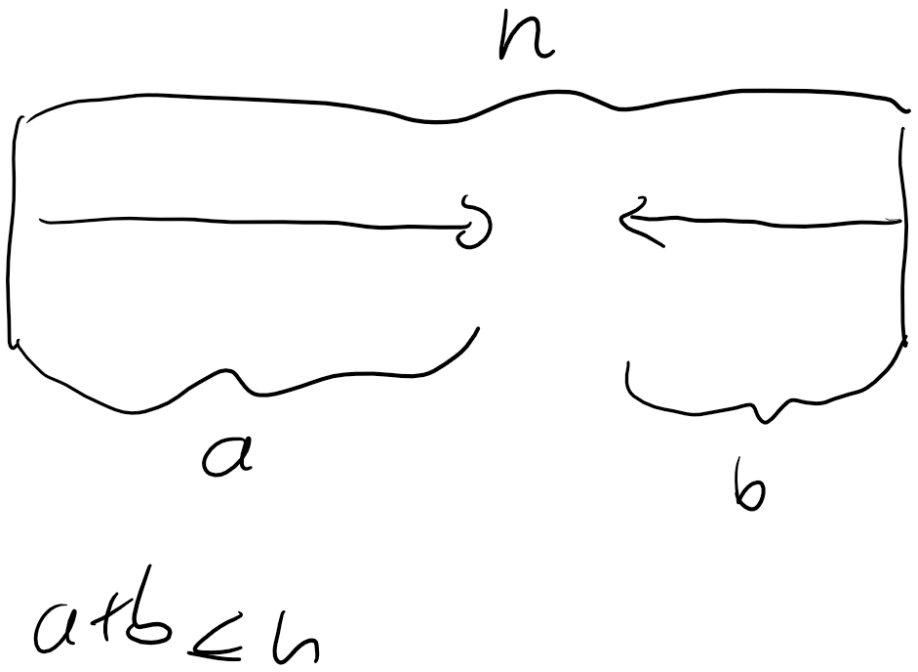
Pop: Enqueue(Q,Dequeue(Q)) $n-1$-Mal, dann return Dequeue(Q). $O(n)$

Möglichkeit 2:
Pop = Dequeue. $O(1)$

Push: Enqueue(Q,x), dann Enqueue(Dequeue(Q),x) $n-1$-Mal. $O(n)$

Ähnlich wie bei c).

b) n ist die Anzahl der Speicherplätze im Array.
a ist die Anzahl der Elemente im Stack A. b ist die Anzahl der Elemente im Stack B.



A wächst nach rechts, B wächst nach links.

c) Im allgemeinen ist es Möglich alle Objekte einer Qeue in einem der Stacks zu speichern. Der zweite Stack wird hierbei nur jeweils benötigt, um eine der Beiden Queue operationen Dequeue oder Enqueue zu realiesieren.
Folglich gibt es zwei Möglichkeiten dies beiden zu implementieren:
Sei S_1 der Stack in dem die Elemente der Queue gespeichert sind.
Möglichkeit 1: Dequeue ist in diesem Fall identisch zu Pop und hat somit eine Laufzeit von $\mathcal{O}(1)$
Für Enqueue muss der ganze Stack mit Pop und Push Operationen in den zweiten Stack übertragen werden. Hierbei wird die Reihenfolge der Elemente invertiert. Nun wird das neue Objekt mit einer Push Operation hinzugefügt. Dannach werden wieder alle Objekte in Stack 1 übertragen und die Operation ist beendet.
Da jede Push und Pop operation eine Laufzeit von $\mathcal{O}(1)$ hat liegt Enqueue in $\mathcal{O}(n)$
Möglichkeit 2: Enqueue ist in diesem Fall identisch zu Push und hat somit eine Laufzeit von $\mathcal{O}(1)$
Für Dequeue muss der ganze Stack mit Pop und Push Operationen in den zweiten Stack übertragen werden. Hierbei wird die Reihenfolge der Elemente invertiert. Nun wird das neue Objekt mit einer Pop-Operation hinzugefügt. Dannach werden wieder alle Objekte in Stack 1 übertragen und die Operation ist beendet.

Da jede Push und Pop operation eine Laufzeit von $\mathcal{O}(1)$ hat liegt Dequeue in $\mathcal{O}(n)$

Aufgabe 2

a) Die Logik beim Vorgehen bei einer Rechtsrotation ist identisch.

Algorithm 1 Linksrotation (T,x)

```
1:  y ← rc[x]
2:  rc[x] ← lc[x]
3:
4:  if lc[y] ≠ nil then
5:      p[y] ← p[x]
6:  if p[x] = nil then
7:      root[T] ← y
8:  else if x = lc[p[x]] then
9:      lc[p[x]] ← y
10: else
11:     rc[p[x]] ← y
12: lc[y] ← x
13: rc[x] ← y
14: h[x] ← 1 + max{h[lc[x]],h[rc[x]]}
15: h[y] ← 1 + max{h[lc[y]],h[rc[y]]}
16: size[x] ← 1 + size[lc[x]] + size[rc[x]]
17: size[y] ← 1 + size[lc[y]] + size[rc[y]]
```

Wobei size[nil] = 0.

Algorithm 2 SizeOf(T,x)

```
1: Return size[x]
```

b) Der Algortihmus wird mit der Wurzel des Baumes aufgerufen.

Algorithm 3 k-median(T,x,k)

```
1: if SizeOf(rc[x]) − SizeOf(x) = k ∨ lc[x] = rc[x] = Nil then
2:     return x
3: if SizeOf(lc[x]) > k − 1 then
4:     return k-median(T, rc[x], SizeOf(x) − (SizeOf(rc[x] + 1))
5: else
6:     return k-median(T, lc[x], k)
```

Budget Laufzeitanalyse:
Der Algorithmus ist rekursiv und ruft sich selber wieder über das linke oder rechte Kind des aktuellen Elements. Da dem Algorithmus zu Beginn die Wurzel vom Baum übergeben wird,kommt es im worst case zu h Rekursions aufrufen(,wobei h die höhe des Baumes ist) und die Laufzeit pro Aufruf ist kostant. Folglich hat der Algorithmus eine Laufzeit von $\mathcal{O}(h)$). Da der Betrachtete Baum ein AVL-Baum ist, ist nach Satz 12.2 gilt h = $\Theta(log(n))$. Also hat der Algorithmus eine Laufzeit von $\mathcal{O}(log(n))$

Aufgabe 3

Jeder Knoten im Baum hat eine zusätzliche Variable size, wie in 2
Auch ist jeder Knoten annotiert mit min und max, welche das Minimum und das Maximums des Teilbaums des Knoten enthält.
Erstaufruf mit $x = root[T]$

Algorithm 4 Schnittmengensuche(T,x,a,b)

```
1: if a > b then
2:     Return 0
3: if x = nil then
4:     Return 0
5: if key[x] < a then
6:     Return Schnittmengensuche(T, rc[x], a, b)
7: else if key[x] > b then
8:     Return Schnittmengensuche(T, lc[x], a, b)
9: else
10:    min ← MinimumSuche(x)
11:    max ← MaximumSuche(x)
12:    if key[min] ≤ a ∧ b ≤ key[max] then
13:        Return size[x]
14:    else
15:        result ← 1
16:        result ← result + Schnittmengensuche(T, lc[x], a, b)
17:        result ← result + Schnittmengensuche(T, rc[x], a, b)
18:        Return result
```

Die Idee ist, dass wir die Schnittmengensuche mithilfe der Annotierten Variablen $size, min, max$ frühstmöglich beenden können. Und damit nur den Baum so tief ablaufen, bis das Ergebnis Klar ist.

Unsere Laufzeit ist durch Teilbäume begrenzt, dessen Eltern-Knoten nicht im Intervall liegen.
Da der AVL Baum balanciert und ein Binärer Suchbaum ist, befinden sich diese Teilbäume maximal auf der Höhe $\leq \log(n)$.
Damit rekursieren wir uns maximal $\log(n)$ mal.
Mit einer Addition von Laufzeit $O(1)$ pro Ebene ist die Laufzeit $\mathcal{O}(log(n))$.

Der Algorithmus ist korrekt, da er ab Zeile 15 die Anzahl der Schnittmengen klassisch Berechnet, mit einer regulären Termination in Zeile 2 und in Zeile 4. Die Rekursion in Zeile 6 und in Zeile 8 ist korrekt, da die Schnittmengen nur in den Teilbäumen liegen die links oder rechts sind. Die Termination mit Size in Zeile 13 ist korrekt, da die der gesamt Teilbaum im gesuchten Intervall liegt.