Grundzüge DS & Alg (WS14/15)

Lösungsvorschlag zu Aufgabenblatt 5

Aufgabe 1

Pseudocode:

```
datatype DeQueue<T>
   T[] a = new T[5]
   int L = 2, R = 2
   size() = R - L
   empty() = size() == 0
   first() = if not empty(): a[L] else throw Exception
   last() = if not empty(): a[R-1] else throw Exception
   resize() =
      a.len = size()*2
      var mid = a.len/2
      var start = mid-size()/2
      var end = start+size()
      move a[L..R] to a[start..end]
      (L, R) = (start, end)
   endpush(T x) =
      if R == a.len:
         resize()
      a[R] = x
      R = R + 1
   frontpush(T x) =
      if L == 0:
         resize()
      L = L - 1
      a[L] = x
   endpop() = if empty(): throw Exception else
      if size() < a.len/4 and a.len > 10: resize()
      R = R - 1
      return a[R]
   frontpop() = if empty(): throw Exception else
      if size() < a.len/4 and a.len > 10: resize()
      var tmp = a[L]
      L = L + 1
      return tmp
```

Hinweis: Ein Range $a \ldots b$ steht für $a \leq i < b$, Arrayindizierung beginnt hier bei 0.

resize() realloziiert das Array, sodass es doppelt so viele Elemente speichern kann, wie in der Queue sind und verschiebt die vorhandenen Elemente so, dass sie mittig im Array liegen.

Laufzeit: Mit Bankkonto-Methode: Für die Push/Pop-Operationen werden 5 Einheiten eingezahlt. Wenn ein resize() durchgeführt wird, sind die eigentlichen Kosten a.len ansonsten 1 (es werden also 4 Einheiten aufs Konto eingezahlt). Nachdem ein resize() durchgeführt wurde, können mindestens $(R-L)/2$ Push/Pop-Operationen mit eigentlichen Kosten 1 durchgeführt werden, bevor von ihnen wieder resize() aufgerufen wird. Nach $(R-L)/2$ Push/Pop-Operationen

ergibt sich ein Kontostand von $2 \cdot (R - L)$, da jede dieser Operationen 4 Einheiten aufs Konto einzahlt, außerdem ist a.len=$2 \cdot (R - L)$. Beim darauf folgenden resize() werden diese eingezahlten Einheiten aufgebraucht, um das Array zu realloziieren und die Elemente zu verschieben: Wenn L' und R' die Grenzen beim resize-Aufruf sind, gilt: $R' - L' < 2 \cdot (R - L)$. Daher reicht der Kontostand aus, um die Kosten von resize() zu decken. Somit hat jede Operation amotisierte konstante Laufzeit.

Aufgabe 2

Damit die Worst-Case Laufzeit für Push, Pop und Get immer in O(1) ist, muss man dafür sorgen, dass sobald das Array voll ist, sofort mit einem größeren Array weitergearbeitet werden kann. Ebenso muss, sobald das Array zu leer wird, d.h. das Array weniger als ein viertel gefüllt ist, sofort mit einem kleineren Array weitergearbeitet werden. Das heißt ein korrekt gefülltes Array muss sofort zur Verfügung stehen. Dies wird erreicht, indem man das Kopieren der Elemente, was im amortisierten Ansatz erst passiert, sobald ein Array nicht mehr verwendet werden soll, parallel zum Füllen des aktuellen Arrays passiert. In Pseudocode sieht dies folgendermaßen aus:

```
class nStack:
   fill: int;
   Akt: Array;
   New: Array;

   push(e):
      if (Akt.size/2) == fill: //hier wird ein neues Array erstellt, dessen erstes
          Viertel mit der ersten Haelfte des aktuellen Arrays nach und nach bei
          jedem push gefuellt wird
         New = new Array with size Akt.size*2;
         Akt[fill] = e;
         New[fill] = e;
         New[fill-(Akt.size/2)] = Akt[fill-(Akt.size/2)];
         fill++;
      elif fill>Akt.size/2: //an diesem Punkt werden Elemente kopiert, sobald das
          Array mehr als die Haelfte gefuellt ist
         Akt[fill]=e;
         New[fill]=e;
         New[fill-(Akt.size/2)]=Akt[fill-(Akt.size/2)];
         fill++;
      else: //falls das Array weniger als die Haelfte gefuellt ist, kann man
          einfach das Element einfuegen
         Akt[fill]=e;
         New[fill]=e;
         fill++;
      if fill == Akt.size: //hier ist das Array voll, allerdings kann direkt mit
          dem naechsten Array weitergearbeitet werden
         Akt = New;
```


```
    New = new Array with size Akt.size*2;

pop():
    if fill==0:
        return;
    a = Akt[fill-1];
    if ((Akt.size/2)-1 == fill) and (Akt.size>8): //eine Arraymindestgroesse von
        8 wird durch die zweite Bedingung sicher gestellt, weiterhin wird hier ein
        neues Array erstellt, das nach und nach bei jedem Loeschen gefuellt wird.
        New = new Array with size Akt.size/2;
        New[fill-(Akt.size/4)] = Akt[fill-(Akt.size/4)];
        fill--;
    elif ((Akt.size/2)-1)>fill:
        if (fill-(Akt.size/4)) >= 0:
            New[fill-Akt.size/4]=Akt[fill-Akt.size/4]; //hier werden die Elemente,
                die im ersten Viertel des Arrays stehen in das neue Array
                kopiert,bei jedem Loeschen genau eines
            fill;
        else:
            fill--;
    else: //falls das Array mehr als die Haelfte gefuellt ist, kann man das
        Element einfach loeschen (in diesem Fall die Zelle freigeben)
        fill--;
    if (fill == (Akt.size/4)) and (Akt.size>8): //hier wird mit dem
        naechstkleineren Array weitergearbeitet
        Akt = New;
    return a;

get(i):
    if (i<fill) and (i>=0):
        return Akt[i];
    else:
        return Null;
```

Der obenstehende Code hat offensichtlich für Push, Pop und Get immer die Laufzeit O(1), da jede Operation in allen Fällen konstante Laufzeit hat. Der Speicherverbrauch liegt in $O(n)$, da der Worst Case folgendermaßen aussieht: Das aktuelle Array ist zur Hälfte gefüllt und man hat ein doppelt so großes Array bereits alloziert. D.h. man hat Speicherverbrauch von n zum Speichern der Elemente, weitere n freie Zellen im aktuellen Array und $4n$ freie Zellen im nächsten Array. Dies macht in der Summe $6n \in O(n)$ Speicherverbrauch. Umgekehrt hat man ein halb so großes Array, falls man weniger als die hälfte aber mehr als ein viertel der möglichen Elemente im aktuellen Array hat.