Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Jörn Müller-Quade
Björn Kaidel, Sebastian Schlag, Sascha Witt

3. Übungsblatt zu Algorithmen I im SoSe 2017

`http://crypto.iti.kit.edu/index.php?id=799`
`{bjoern.kaidel,sascha.witt}@kit.edu`

Musterlösungen

Aufgabe 1 (*Amortisierte Analyse,* 4 *Punkte*)

Modifizieren Sie die Implementierung von unbeschränkten Arrays aus der Vorlesung so, dass bei jedem Aufruf von `reallocate` das Array aufsteigend sortiert wird. Dabei sollen die amortisierten Kosten für jede Operation weiterhin in $\mathcal{O}(1)$ liegen. Gehen Sie dazu davon aus, dass dem Array nur Zahlen aus $\mathbb{N}$ kleiner gleich einer festen und konstanten Schranke $\ell \in \mathbb{N}$ hinzugefügt werden und Ihnen ein zweites (beschränktes) Array A mit ℓ Elementen zur Verfügung steht. Sie dürfen alle Operationen des unbeschränkten Arrays modifizieren. Argumentieren Sie, dass die amortisierten Kosten auch nach Ihrer Modifikation noch in $\mathcal{O}(1)$ liegen. Es ist kein Pseudocode nötig, es genügt, wenn Sie Ihre Modifikationen eindeutig beschreiben.

Musterlösung:
Wir verwenden den i-ten Eintrag des Arrays A, um zu zählen, wie häufig welche Zahl ins unbeschränkte Array eingefügt wurde. $A[i]$ gibt also an, wie häufig die Zahl i im Array vorkommt. Da nach Voraussetzung alle einzufügenden Zahlen kleiner gleich ℓ sind, bietet A genug Platz, um diese Information zu speichern.

Wir modifizieren die Operationen `pushBack` und `popBack` so, dass der jeweilige Eintrag in A modifiziert wird: bei `pushBack`(i) wird $A[i]$ inkrementiert, bei `popBack` wird der passende Eintrag in A dekrementiert. `reallocate` modifizieren wir so: Anstatt die Elemente aus dem Array direkt in das neue und größere Array zu kopieren, durchlaufen wir das beschränkte Array A von 1 bis ℓ und fügen zuerst $A[1]$-mal die 1 ins vergrößerte Array ein, dann $A[2]$ mal die 2 etc. Da wir die Einträge von A immer konsistent verändert haben, entspricht dies genau dem Inhalt des Arrays vor Aufruf von `reallocate`, jedoch in aufsteigend sortierter Reihenfolge. Da ℓ eine Konstante ist, benötigt ein einzelner Aufruf von `reallocate` Zeit $\mathcal{O}(\ell + n) = \mathcal{O}(n)$ Schritte (wobei n die aktuelle Anzahl der Elemente im Array ist), da wir alle ℓ Einträge von A betrachten, aber nur n Elemente ins neue und vergrößerte Array einfügen müssen.

Da alle Modifikationen nur einen konstanten Aufwand zu den Algorithmen hinzufügen, ändert sich an der Analyse nichts (d.h., wir können die Algorithmen genauso analysieren, wie in der Vorlesung), womit sich wieder die gewünschte amortisierte Laufzeit von $\mathcal{O}(1)$ ergibt.

Aufgabe 2 (*Doppelt verkettete Liste,* $3 + 3 = 6$ *Punkte*)

Implementieren Sie (im Pseudocode) eine doppelt verkettete Liste mit den folgenden Funktionen. Die Elemente der Liste sollen Zahlen aus $\mathbb{N}$ enthalten. Gehen Sie dazu davon aus, dass zu Beginn keine Funktion implementiert ist.
Anmerkung: Mit „Element“ ist hier (und überlicherweise) ein Item der Liste gemeint, also die Objekte, die die Zeiger und den Wert speichern. Im Folgenden erhalten die Methoden `insert` und `remove` also dementsprechend einen Zeiger auf ein Listenelement als Übergabewert.

- `last` soll das letzte Element der Liste zurückgeben. Die Laufzeit von `last` soll dabei immer in $\mathcal{O}(1)$ sein.

- `insert`(e) soll ein neues Element ans Ende der Liste anhängen.
- `isSorted` soll 1 ausgeben, wenn die Liste aufsteigend sortiert ist, 0 falls nicht.
- `remove`(e) soll das Element e aus der Liste entfernen.

Ansonsten soll die Liste keine Funktionen implementieren. Im Folgenden bezeichne n die Anzahl von Elementen in der Liste.

a) Geben Sie eine Implementierung der Liste an, sodass `insert` und `remove` in $\mathcal{O}(1)$ und `isSorted` in $\mathcal{O}(n)$ laufen.

b) Geben Sie eine Implementierung der Liste an, sodass `insert` und `isSorted` in $\mathcal{O}(1)$ und `remove` in $\mathcal{O}(n)$ laufen.

Musterlösung:

a) Um das Gewünschte zu erreichen, funktionieren `insert` und `remove` wie gewohnt und `isSorted` durchläuft bei jedem Aufruf die Liste komplett und überprüft, ob sie sortiert ist.

```
Class List
    h : Item                                        −− head element
    Function last() : return h.prev

    Function insert(a : Handle) :
        l := last()
        a → next := h
        a → prev := l
        l → next := a
        h → prev := a

    Function remove(a : Handle) :
        prev := a → prev
        next := a → next
        prev → next := next
        next → prev := prev

    Function isSorted() : Boolean
        sorted : Boolean
        sorted := true
        e1 := h → next
        e2 := e1 → next
        while e2 ≠ h ∧ sorted = true do
            if e1.e > e2.e do
                sorted := false
            else
                e1 := e2
                e2 := e2 → next
        return sorted
```

b) Um das Gewünschte zu erreichen, führen wir eine neue Variable sorted ein und aktualisieren diese bei `insert` und `remove`. Dazu muss `remove` aber nun nach dem Entfernen des Elements die komplette Liste durchlaufen, um festzustellen, ob die Liste danach sortiert ist oder nicht. Als Optimierung kann man noch prüfen, ob die Liste vorher sortiert war; in dem Fall ist sie nach dem Entfernen auf jeden Fall immer noch sortiert. Dadurch erreicht man dann im Best-Case (sortierte Liste) $\mathcal{O}(1)$, aber im Worst-Case (nicht sortierte Liste) bleibt es bei $\mathcal{O}(n)$.

```
Class List
    h : Item                                                  −− head element
    sorted : Boolean
    sorted := true
    Function last() : return h.prev

    Function insert(a : Handle) :
        l := last()
        a → next := h
        a → prev := l
        l → next := a
        h → prev := a
        if l.e > a.e do
            sorted := false

    Function remove(a : Handle) :
        prev := a → prev
        next := a → next
        prev → next := next
        next → prev := prev

        sortedTemp : Boolean
        sortedTemp := true
        e1 := h → next
        e2 := e1 → next
        while e2 ≠ h ∧ sortedTemp = true do
            if e1.e > e2.e do
                sortedTemp := false
            else
                e1 := e2
                e2 := e2 → next
        sorted := sortedTemp

    Function isSorted() : Boolean
        return sorted
```

Aufgabe 3 (*Mengen als Listen,* $1 + 1 + 3 = 5$ *Punkte*)

Es seien M_1, M_2 zwei endliche Mengen aus $\mathbb{N}$ mit $n := |M_1| = |M_2| \in \mathbb{N}$. In dieser Aufgabe seien Mengen als einfach verkettete Liste implementiert, indem jedes Element aus der Liste ein Element aus der Menge repräsentiert. Im Folgenden sollen die Mengenoperationen $\bigcup, \bigcap$ und $\setminus$ algorithmisch implementiert werden. Ihnen stehen dazu die üblichen Operationen auf einfach verketteten Listen zur Verfügung. Die Algorithmen müssen dabei die Listen M_1 und M_2 nicht zwingend erhalten, d.h. sie dürfen M_1 und M_2 beliebig modifizieren und die Algorithmen müssen die Strukturen von M_1 und M_2 nicht aufrecht erhalten. Implementieren Sie die Operationen so, dass Duplikate entfernt werden, beispielsweise soll $\{1, 2\} \bigcup \{1\} = \{1, 2\}$ sein. Sie können davon ausgehen, dass die Mengen M_1 und M_2 jeweils nicht leer sind und keine Duplikate enthalten.

a) Es gelte $M_1 \bigcap M_2 = \emptyset$. Geben Sie für diesen Fall Algorithmen für $\bigcup, \bigcap$ und $\setminus$ an, sodass sie jeweils in $\mathcal{O}(1)$ laufen.

b) Es gelte $M_1 \bigcap M_2 \neq \emptyset$. Lassen sich $\bigcup, \bigcap$ und $\setminus$ weiterhin in $\mathcal{O}(1)$ implementieren? Begründen Sie Ihre Antwort!

c) Geben Sie Algorithmen an, die $\bigcup, \bigcap$ und $\setminus$ für beliebige endliche Mengen M_1 und M_2 korrekt implementieren und deren Laufzeit in $\mathcal{O}(n^2)$ liegt.

Musterlösung:

a) Die folgenden Algorithmen erfüllen das gewünschte, wenn $M_1 \bigcap M_2 = \emptyset$.

Function union(M_1 : List of Element M_2 : List of Element) : List of Element
  M_1.last() $\rightarrow$ next := M_2.head() $\rightarrow$ next
  M_2.last() $\rightarrow$ next := M_1.head()
  return M_1

Function intersection(M_1 : List of Element M_2 : List of Element) : List of Element
  L : List of Element −− create new and empty list
  return L

Function complement(M_1 : List of Element M_2 : List of Element) : List of Element
  return M_1

b) Nein, dass ist nicht mehr möglich. Zu $\bigcup$: Die Mengen M_1 und M_2 enthalten in diesem Fall gleiche Elemente. Nach Aufgabenstellung müssen diese Duplikate gefunden und gesondert behandelt werden. Somit können die Listen also z.B. nicht mehr simpel aneinander gehängt werden. Zu $\bigcap, \setminus$: Da der Schnitt der beiden Mengen nicht mehr leer ist, müssen jetzt die Elemente gefunden werden, die in beiden Mengen vorhanden sind. Auch dies ist nicht in konstanter Zeit möglich.

c) Die folgenden Algorithmen erfüllen das gewünschte:.

Function intersection(M_1 : List of Element M_2 : List of Element) : List of Element
  L : List of Element −− create new and empty list

  $e_1 := M_1$.head() $\rightarrow$ next
  while $e_1 \neq M_1$.head() **do**
    found := false
    $e_2 := M_2$.head() $\rightarrow$ next
    while $e_2 \neq M_2$.head() $\wedge$ found = false **do**
      if $e_1.e = e_2.e$ **do**
        found = true
        item : Item
        item.e = $e_1.e$
        L.pushBack(item)
      $e_2 := e_2$.next
    $e_1 := e_1$.next
  return L

Function complement(M_1 : List of Element M_2 : List of Element) : List of Element
  $e_1 := M_1$.head() $\rightarrow$ next
  prevTemp := M_1.head() −− Vorgänger merken, damit Löschen von e_1 in $\mathcal{O}(1)$ möglich ist
  while $e_1 \neq M_1$.head() **do**
    found := false
    nextTemp = e_1 $\rightarrow$next −− Zeiger auf das nächste Element sichern (falls e_1 gelöscht wird)
    $e_2 := M_2$.head() $\rightarrow$ next
    while $e_2 \neq M_2$.head() $\wedge$ found = false **do**
      if $e_1.e = e_2.e$ **do**

found = true
prevTemp $\rightarrow$ next := $e_1 \rightarrow$ next — — e_1 aus M_1 löschen
$e_2 := e_2 \rightarrow$ next
prevTemp := e_1
e_1 := nextTemp
return M_1

Function union(M_1 : List of Element M_2 : List of Element) : List of Element
M_3 := complement(M_2,M_1)
M_1.last() $\rightarrow$ next := M_3.head() $\rightarrow$ next
M_3.last() $\rightarrow$ next := M_1.head()
return M_1

Aufgabe 4 (*Listen invertieren,* 3 *Punkte*)

Geben Sie einen nicht rekursiven Algorithmus in Pseudocode an, der eine einfach verkettete Liste (mit head-Element) invertiert, d.h., die Reihenfolge der Elemente umdreht (das erste Element wird zum letzten, dass zweite Element zum vorletzten etc.). Die Laufzeit des Algorithmus soll in $\Theta(n)$ (n sei die Länge der Liste) sein und darf nur konstant viel Speicherplatz (zusätzlich zur Liste) benötigen. Argumentieren Sie kurz, dass ihr Algorithmus diese Bedingungen erfüllt.

Musterlösung:
Der folgende Algorithmus leistet das Gewünschte:

Function invertList(L : List of Element) :
head := L.head()
prev := head
current := head $\rightarrow$ next

while current $\neq$ head **do**
temp := current $\rightarrow$ next
current $\rightarrow$ next := prev

prev := current
current := temp

head $\rightarrow$ next := prev

Die Laufzeit ist in $\Theta(n)$, da die Liste genau ein Mal komplett durchlaufen wird. Der zusätzliche Speicherplatz ist konstant, da nur vier zusätzliche Zeiger benötigt werden (Der Zeiger „head“ könnte noch eingespart werden und wurde hier nur aus Bequemlichkeit verwendet).