Datenstrukturen und Algorithmen: Hausübung 3

Felix Schrader, 3053850
Jens Duffert, 2843110
Eduard Sauter, 3053470

5. November 2015

Aufgabe 1

a) Der folgende Algorithmus legt zunächst ein neues Array `permuted_data` an, das mit dem Ergebnis der Permutation gefüllt werden soll. Dazu wird mit einer `for`-Schleife das ganze Array durchgegangen. Die Stelle `index` in `permuted_data` wird dann mit `data[permutation[index]]` gefüllt, da dies der Wert ist, der nach dem permutieren an der Stelle `index` stehen soll.

```
1     function permute(data, permutation) {
2         permuted_data = new Array(n);
3         for(index = 0, index < n, index++) {
4             permuted_data[index] = data[permutation[index]];
5         }
6         return permuted_data;
7     }
```

In dem Algorithmus gibt es zunächst eine Zuweisung in der Zeile:

```
1     permuted_data = new Array(n);
```

Dann wird eine `for`-Schleife durchlaufen:

```
1     for(index = 0, index < n, index++) {
2         permuted_data[index] = data[permutation[index]];
3     }
```

Der Schleifenkopf weist dann mit der Initialisierung einmal einen Wert zu, vergleicht dann n-mal, ob `index` kleiner als n ist und erhöht `index` dann n-mal, was je einer Addition und einer Zuweisung entspricht. Dann wird der entsprechenden Stelle in `permuted_data` ein Wert zugewiesen. Auch das passiert n-mal. Insgesamt erhält man also eine Laufzeit:

$$\begin{aligned} f(n) &= 1 + n + n + 2 \cdot n + n \\ &= 5 \cdot n + 1 \in \Theta(n) \end{aligned}$$

Allerdings benötigt man auch linear viel Speicher, da man ein neues Array der Länge n erstellt.

b)

Aufgabe 2

a)

```
1  function insertList(L1, L2, pos) {
2    L1_position = pos;
3    L2_position = L2.first();
4
5    while(L2_position != NIL) {
6      L1.insert(L1_position, L2.retrieve(L2_position));
7      L1_position = L1_position.next();
8      L2_position = L2_position.next();
9    }
10 }
```

b) Laufzeit Untersuchungen:

i) Nach Vorlesung sind die Operationen `insert`, `next`, `retrieve` und `first` implementiert durch die verkettete Liste in $\mathcal{O}(1)$. Es sei $n =$ `L2.length()`. Die Schleife beginnend bei Zeile 5 wird n mal durchlaufen. Jede der Operationen im Schleifenkopf sowie der Vergleich zu beginn der Schleife sind in $\mathcal{O}(1)$. Folglich ist die Laufzeit von `insertList` in $\mathcal{O}(n)$.

ii) Würde man `L2` vor `pos` einfügen wollen, so müsste man vorher die Position vor `pos` finden.

```
1  function previous(L, pos) {
2      last_pos = NIL; current_pos = L.first(); while(current_pos != pos &&
           current_pos != NIL) {
3          last_pos = current_pos;
4          current_pos = pos.next();
5      }
6      return last_pos
7  }
```

Es sei $m =$ `L1.Length` Die Worst-Case Laufzeit von `previous` ist in $\mathcal{O}(m)$, da in diesem Fall m Elemente überprüft werden müssen.

```
1    function insertListPrevious(L1, L2, pos) {
2        L1_position = previous(L1, pos);
3        L2_position = L2.first();
4
5        while(L2_position != NIL) {
6            L1.insert(L1_position, L2.retrieve(L2_position));
7            L1_position = L1_position.next();
8            L2_position = L2_position.next();
9        }
10   }
```

Analog zu `insertList` ergibt sich die Worst-Case Laufzeit von `InsertListPrevious` zu $\mathcal{O}(m+n)$.

Aufgabe 3

a) Die Methoden sehen wie folgt aus (Quelle: http://en.cppreference.com/w/cpp/container/vector). Es sein `v` ein `std::vector`:

`L.first`	$\leftrightarrow$	`0`	$\mathcal{O}(1)$
`L.last`	$\leftrightarrow$	`v.size-1`	$\mathcal{O}(1)$
`L.length`	$\leftrightarrow$	`v.size`	$\mathcal{O}(1)$
`L.retrieve`	$\leftrightarrow$	`v.at`	$\mathcal{O}(1)$
`L.delete`	$\leftrightarrow$	`v.erase`	$\mathcal{O}(n)$
`L.insert`	$\leftrightarrow$	`v.insert`	$\mathcal{O}(1)$

b) Die `std::deque` ist prinzipiell ein Listenähnlicher Typ, welcher es ermöglicht, schnell Elemente am beginn und am Ende einzufügen. Die Klasse std::stack zeichnet sich daraus aus, das immer nur das oberste Elemente verändert werden kann (entfernen, hinzufügen usw.). Dies ist auch in std::deque möglich. Dies Voraussetzung hierzu ist die Beschränkung, das nur die Befehle für das obere Element verwendet werden. Dadurch ist es möglich std::stack durch std::deque zu implementieren.

c) `std::deque` ist meist durch eine Doppelt-Verlinkte Liste implementiert. Einfügen von Elementen läuft deswegen immer in $\mathcal{O}(1)$. Siehe auch die entsprechende implementierung der ADT Liste. `std::vector` muss schnellen zufälligen Zugriff auf die Elemente garantieren. Hierzu arbeitet im Hintergrund meist ein Array. Einfügen von Elementen in ein Array kostet im Schlimmsten Fall $\mathcal{O}(n)$ Zeit. Falls der Vektor ausreichend speicher vorallokiert hat, so ist er in der Lage, Elemente schnell am Ende einzufügen. So kann eine amortisierte Laufzeit von $\mathcal{O}(1)$ ermöglicht werden.