AD Praktikum SS 2017
Aufgabenblatt 1 - Theorieteil
Entwurf von Datentypen

Alexander Mendel
Florian Heuer
Karl-Fabian Witte

Gruppe 1
Abgabe 29.03.2017

Abstract. Ici va le résumé

1 Entwurf und Entscheidungen

Objektmenge : *List*, *Elem*, *Key*, *Pos*

Um diesen Abschnitt besser folgen zu können, wird das Klassendiagramm, welches wir mit Eclipse erstellt haben, beigelegt.

Pos ist für eine besseren Handhabung ein Integer (atomarer Datentyp). Für *Key* haben wir eine Klasse `Schluessel` entworfen, welcher einen Integer als eindeutiger *Key* beinhaltet. *Elem* ist der generische Datentyp / die Templatetyp der Liste. *Key* und *Elem* werden in der abstrakten Klasse `Knoten` miteinander verbunden. Das *Elem* ist ein konstanter Wert, da wir keine Elemente zulassen, die nichts beihalten (kein Speicher allokiert wurde: -> `NULL`).

List ist in der abstracten Klasse `Liste` realisiert. Sie besitzt zudem ein Feld, welches die Anzahl der enthaltenen Elemente beinhaltet.

Jede Listenart (A B C) von `Liste` und `Knoten` erben von den Abstrakten Klassen und werden entsprechend erweitert.

Objektmenge

ListeA ist eine Arraylist, welches die Elemente in einem Array verwaltet und diese in aufsteigender Reihenfolge ohne Lücken speichert. Wenn der Platz im Array nicht reicht, soll in ein größeres Array erzeugt und die Elemente darin umgelegt werden.

ListeB ist eine doppelt verkettete Liste, dessen Knoten jeweils ein Zeiger auf den vorherige *prev* und nächste *next* Knoten hat. Der Zugriff auf den ersten Knoten erfolgt über den Zeiger *head*. Da Java alle Objekte im Heap frei alloziiert, wurde auf das Array, welches diese Elemente halten sollte, verzichtet.

ListeC ist eine einfach verkettete Liste, dessen Knoten Referenzen *next* auf den nächsten Knoten haben. Der Einstieg in die Liste erfolgt über den *head* Zeiger. Der *tail* Zeiger zeigt auf den letzten Knoten. Dessen *next* zeigt auf den Knoten mit dem letzten Element. *head* und *tail* zeigen jeweils auf Dummy-Knoten. Bei der Suche zeigt *tail* auf ein Stoppelement.

Um die Listen einfacher zu Testen, wurden einige Hilfsfunktionen public gemacht.

1.1 OCL

Es folgt eine Auflistung der Zugriffsfunktionen der Liste, die für jede Implementation gelten. Definition:

`Sei` $n \in \mathbb{N}$ `die Anzahl der Elemente in der Liste` $l = \{a_1, a_2, ..., a_n\} \in List$.

Allgemeine Vorbedingung: Der Typ von `Elem` ist fest.

Operationen

insert : $List \times Pos \times Elem \to List \cap \{error\}; (l, p, e) \mapsto l.\text{insert}(p, e)$
pre : -
post : Wenn $p < 0$ oder $p > n + 1$ oder $e = NULL \to error$,
sonst: $l = \{a_1, a_2, ..., a_{p-1}, e, a_p, a_{p+1}, ..., a_n\}$

delete : $List \times Pos \to List \cap \{error\}; (l, p) \mapsto l.\text{delete}(p)$
pre : -
post : Wenn $p < 0$ oder $p > n \to error$,
sonst: $l = \{a_1, a_2, ..., a_{p-1}, a_{p+1}, ..., a_n\}$

delete : $List \times Key \to List \cap \{error\}; (l, k) \mapsto l.\text{delete}(k)$
pre : -
post : Wenn: $k == NULL \to error$
sonst: $\exists p \in Pos$ für ein $e \in Elem$ mit Schlüssel k:
$l = \{a_1, a_2, ..., a_{p-1}, a_{p+1}, ..., a_n\}$

find : $List \times Key \to Pos \cap \{error\}; (l, k, p) \mapsto p := l.\text{find}(k)$
pre : -
post : Wenn: $NULL == k \to error$
sonst:
$\exists pos \in Pos$ für ein $e \in Elem$ mit Schlüssel $k \to p = pos$.
$\nexists pos \in Pos$ für ein $e \in Elem$ mit Schlüssel $k \to p = -1$.

retrieve : $List \times Pos \to Elem \cap \{error\}; (l, p) \mapsto e := l.\text{retrieve}(p)$
pre : Wenn $p < 0$ oder $p > n \to error$,
post : Sei $a_p \in Elem$ in l an der Position $p : e = a_p$

concat : $List1 \times List2 \to List1; (l_1, l_2) \mapsto l_1.\text{concat}(l_2)$
pre : Die Typen von beiden Listen l_1 und l_2 sind die selben.
post : Sei $l_1 = \{a_1, a_2, ..., a_n\}$ und $l_2 = \{b_1, b_2, ..., b_m\} : l_1 = \{a_1, a_2, ..., a_n, b_1, b_2, ..., b_m\}$
(und $l_2 = \{b_1, b_2, ..., b_m\}$)

2 Aufwandsanalyse

Es wurde eine Aufwandsmessung der drei Listen und den oben beschrieben Funktionen durchgeführt. Dabei wurden die Listen mit n Elementen befüllt, wobei $n = 10^k$ mit $k = 1...5$ ist. Es wurden dabei der Aufwand, sprich die Operationen in den Funktionen, gezählt. Diese Zahlen wurden dann Linienpolt auf doppellogarithmischer Skala aufgetragen in der Form `Aufwand(Listengröße)`. (Abszisse: Listengröße n , Aufwand: Ordinate f(n)); Die Farbkodierung ist: Arrayliste: blau, doppelt verkettete Liste: rot, einfach verkettete Liste: gelb

Für diese Messung wurden zufällige Positionen zum Löschen und Einfügen verwendet. Da dies der Fall ist, wird qualitativ ausgewertet.

Zusehen ist, wie erwartet, dass die Komplexitäten der Funktionen alle nicht $O(n)$ überschreiten (zu erkennen an der linearen Steigung, bzw bei $O(1)$ an einem Konstanten Wert). Das liegt daran, dass wir keine Verschachtelung von Schleifen über die Größe der Liste haben. Auch zu sehen ist, dass die verketteten Listen sich im Prinzip gleich verhalten. Die Arrayliste schlägt sich meist schlechter wegen des ständigen Umkopierens.

Auch ist beim **concat()** schön zu sehen, dass die Arraylist (ListA) ständig in repetitiven Aktion (Schleife) umkopieren muss. Die verketteten Listen müssen unabhängig von der Größe nur einmal ihre Zeiger neu setzten.

Auffällig ist, dass bei Funktionen, die einen Schlüssel als Parameter besitzen (**find(), delete(key)**), die Messung bei allen drei Varianten relativ gleiche Ergebnisse liefern. Das liegt daran, dass bei allen Liste die Knoten von Vorne nach Hinten durchgestöbert werden, um einen Schlüsselvergleich zu machen.

Etwas widererwartend ist, dass einige Funktionen, wie die **delete(pos)** und **insert()**, bei den verketteten Listen auch $O(n)$ aufweisen. Der Grund hierfür ist, dass wir die Position als Integer definiert haben. Dadurch haben wir keinen direkten Zeiger auf das Element. Deswegen mussten die verketteten Liste die Position bis zum gewünsten Knoten in einer Schleife hochlaufen. Bei der Arrayliste ist es das umkopieren der Elemente im Array.

Dass die Arraylist bei ihrer Königsdisziplin, dem **retrieve()**, nicht einen konstanten Wert abliefert, ist uns noch nicht klar. Die Logik ist ohne Schleife und müsste deswegen $O(1)$ sein. Die verketteten Listen müssen, wie schon vorher erwähnt, zur gewünschten Position hochlaufen.

Verbesserung fürs nächste Mal: Immer den Worst-Case und Best-Case messen. Wenn man kann, auch den Durchschnittsfall messen. Damit lassen sich dann auch die verketteten Listen quantitativ vergleichen.