Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Jakob Vogel

WiSe 2012/13
Übungsblatt 4
7. November 2012

Algorithmen und Datenstrukturen

Aufgabe 1 (P) Doppelt verkettete Liste

In der Vorlesung haben Sie die (einfach) verkettete Liste (Linked List) kennengelernt, und dazu auch entsprechenden Code gesehen. Ebenso haben Sie bereits das Konzept der doppelt verketteten Liste (Doubly Linked List) besprochen. Dabei kamen insbesondere die Grundstruktur, das Einfügen und das Löschen von Elementen zur Sprache. In dieser Aufgabe beschäftigen wir uns mit einer möglichen Implementierung.

Um den Aufwand für Sie in Grenzen zu halten, finden Sie eine teilweise fertiggestellte Implementation auf unserer Website, die Sie als Grundlage nutzen können.

a) Für jeden zu speichernden Wert benötigen wir einen Container, der neben den eigentlichen Daten auch zwei Zeiger zum vorherigen und zum nächsten Container enthält. Schreiben Sie also eine Klasse namens `DoublyLinkedListNode`, die diese drei Komponenten hat, wobei wir uns auf `int`-Typen beschränken. Der Konstruktor muss dabei die beiden Zeiger auf `NULL` setzen und kann den zu speichernden Wert als Parameter übergeben bekommen. *Bei Nutzung der Vorlage ist diese Aufgabe bereits gelöst.*

b) In der Vorlesung hat die Implementation in C die verschiedenen Funktionen explizit aufgelistet, während in C++ der Standard-Typ `std::list<...>` die Funktionalität abstrahiert. Auch wir wollen dem Nutzer ein einfaches Interface bereitstellen, und „verpacken“ dementsprechend die Hilfsfunktionen. Schreiben Sie also eine Klasse `DoublyLinkedList`, die die zwei Container-Zeiger `start` und `stop` enthält. Initialisieren Sie die Pointer im Konstruktor auf `NULL`. *Bei Nutzung der Vorlage ist diese Aufgabe bereits gelöst.*

c) Um dann unsere Implementation testen zu können, benötigen wir als nächstes eine Ausgabe-Routine. Im Unterschied zum Beispiel aus der Vorlesung wollen wir nicht nur die Daten sehen, sondern auch die Zeigeradressen, um unseren Code debuggen zu können. Schreiben Sie also in `DoublyLinkedList` eine Funktion `print()`, die die Liste vorwärts durchläuft und dabei zu jedem Container Wert, Adresse des eigenen Containers (`this`) und der beiden benachbarten Container ausgibt. Letzteres könnte man auch sehr elegant als Funktion von `DoublyLinkedListNode` umsetzen. *Bei Nutzung der Vorlage ist diese Aufgabe bereits gelöst.*

d) Jetzt sind wir soweit, die Liste mit Daten füllen zu können. Dazu schreiben wir zunächst in `DoublyLinkedList` eine Hilfsfunktion `insertAfterNode(DoublyLinkedListNode* node, int value)`, die einen Container für den gegebenen Wert `value` *nach* einem gegebenen Zeiger `node` einfügt. Erzeugen Sie den Container mittels `new` auf dem Heap. Implementieren Sie die Funktion so, dass bei `node = NULL` am Anfang der Liste eingefügt wird, und vergessen Sie die beiden Zeiger `start` und `stop` nicht!

   Wir wollen den Nutzer später aber nicht mit internen Details belästigen, sondern einfache Einfüge-Operationen bereitstellen. Schreiben Sie also zwei einfacher zu bedienende Funktionen `insertFront(int value)` und `insertBack(int value)`, die am Anfang bzw. Ende der Liste einfügen und dazu die zuvor geschriebene Hilfsfunktion nutzen.

e) In der vorherigen Aufgabe haben wir die Container auf dem Heap erzeugt, diese bislang aber noch nicht gelöscht. Wir müssen nun also sicherstellen, dass spätestens am Ende der Lebenszeit von `DoublyLinkedList` aufgeräumt wird. Erzeugen Sie also eine Funktion `clear()`, die die Liste durchgeht und alle noch bestehenden Elemente ordentlich freigibt, und rufen Sie sie im Destruktor auf. Vergessen Sie nicht die beiden Zeiger `start` und `stop`!

f) Wir wollen aber auch schon zuvor einzelne Knoten aus einer bestehenden Liste löschen können. Auch dazu schreiben wir uns in `DoublyLinkedList` zunächst eine Hilfsfunktion namens `eraseNode(DoublyLinkedListNode* node)`, die den gegebenen Container aus der Verknüpfung „herauslöst“ und dann freigibt. Denken Sie auch hier an die beiden Zeiger `start` und `stop`!

   Abschließend wollen wir wieder den Nutzer entlasten, und stellen ihm daher zwei Hilfsfunktionen zur Verfügung: Schreiben Sie also zwei einfacher zu bedienende Funktionen `eraseFirst(int value)` und `eraseLast(int value)`, die die Liste von vorne bzw. hinten durchgehen und unter Verwendung der zuvor geschriebenen Hilfsfunktion den ersten Container des gegebenen Werts `value` löschen. Sollte der Wert nicht gefunden werden, darf die Liste natürlich nicht verändert werden.

g) Fügen Sie letztendlich eine Funktion `isEmpty()` hinzu.

Aufgabe 2 (P) Queues und Unit Tests

In vorhergehenden Übungen und in der Vorlesung haben wir bereits den Stack besprochen. Diese Datenstruktur war ja definiert über eine Reihe von *Operationen* sowie über den LI-FO-*Constraint*, und ist insofern auf mehrere Arten gültig implementierbar. Der Stack ist also ein Beispiel für einen *Abstrakten Datentyp*.

In dieser Aufgabe beschäftigen wir uns mit einer eng verwandten Struktur, der *Queue*, die ein dem Stack sehr ähnliches Interface hat, aber nach FI-FO-Prinzip arbeitet. Der erste hinterlegte Wert wird dementsprechend auch als erstes wieder ausgelesen, also *First In, First Out*!

a) Auch bei der Queue gibt es verschiedene Möglichkeiten der Implementierung. Wenn Sie die erste Aufgabe gelöst haben, können Sie die dort gebaute `DoublyLinkedList` verwenden, oder anderenfalls auf die `std::list<...>` aus der Standardbibliothek zurückgreifen, deren Verwendung in der Vorlesung gezeigt wurde.

   Auf der Website zur Vorlesung finden Sie eine Vorlage, in der bereits eine Klasse `Queue` definiert ist, bei der die Funktionen aber noch ausgefüllt werden müssen. Fügen Sie die fehlenden Befehle und Variablen hinzu, um eine zur Definition aus der Vorlesung konforme Implementierung zu erhalten!

b) Die Klasse verfügt bereits über eine `main(...)` Funktion, in der eine Variable des Typs `Queue` erzeugt wird und diese an eine Funktion `unitTest(...)` weitergibt. Starten Sie das Programm und verfolgen Sie diesen Aufruf!

   Programme müssen fehlerfrei sein, also syntaktisch korrekt („kompilierbar“) sein und das richtige Verhalten zeigen. Letzteres bedeutet sowohl, dass das Programm nicht abstürzt, als auch, dass das Ergebnis den Erwartungen entspricht. Beispielsweise muss ein Array, nachdem es eine Sortierfunktion durchlaufen hat, offensichtlich auch sortiert sein.

   Theoretisch gibt es Methoden, die Korrektheit eines Programms tatsächlich, also mathematisch, zu beweisen, doch sind derartige Methoden bei hinreichender Komplexität nicht mehr gangbar. Eine Alternative dazu sind *Unit Tests*, die den Code als *Black Box* behandeln, aber an der Schnittstelle die *Constraints* abtesten. Im Falle der Queue wird also etwa getestet, dass bei einer bekannterweise leeren Queue die Methode `isEmpty()` tatsächlich `true` ergibt.

   Welche Probleme sehen Sie bei Unit Tests?