12
Datenstrukturen und Algorithmen

Jörn Loviscach

Versionsstand: 25. September 2013, 18:04

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen beim Ansehen der Videos:
http://www.j3L7h.de/videos.html

Bitte hier notieren, was beim Bearbeiten unklar geblieben ist

1 Datenstrukturen

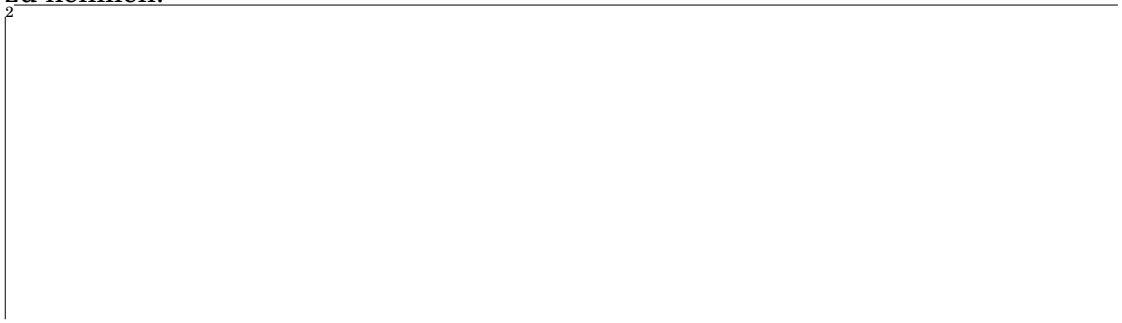
Wie organisiert man die Daten, die man verarbeiten will, am besten? Folgen von Messwerten, die Pixel von Bildern oder die Voxel von 3D-Simulationen rufen nach einer Datenstruktur, die wir schon kennen: das ein- oder mehrdimensionale Array.[1]

Hier nun einige weitere Beispiele für Datenstrukturen.

Zum Beispiel zum Bearbeiten von Druckaufträgen oder von Nachrichten benötigt man eine recht andere Datenstruktur: die Warteschlange [Queue]. Sie funktioniert

nach dem Prinzip „FIFO“ (first in – first out). Es gibt eine Funktion `enqueue`, um ein neues Element ans hintere Ende der Schlange zu stellen, und eine Funktion `enqueue`, um das vorderste vorhandene Element abzuholen und aus der Schlange zu nehmen:

[2]



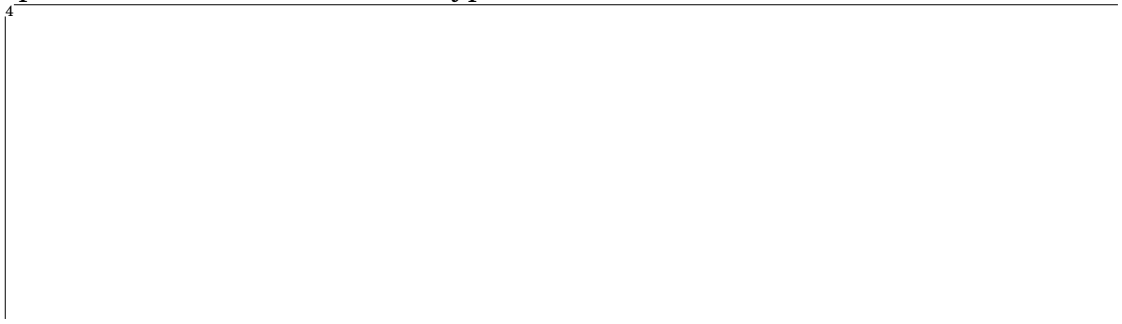
C umfasst standardmäßig keine Queue, C++ dagegen schon.

Das umgekehrte Prinzip ist „LIFO“ (last in – first out) wird vom Stapelspeicher [stack] verwirklicht. Die Funktion `push` legt ein neues Element auf den Stapel; die Funktion `pop` holt das oberste Element ab und nimmt es aus dem Stapel:

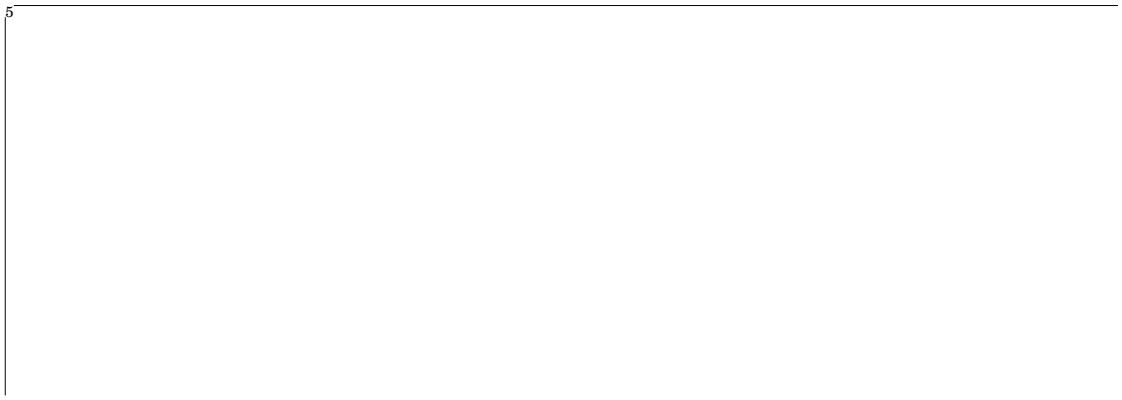
[3]

C umfasst standardmäßig keinen Stack, C++ dagegen schon. Intern benutzt der Mikroprozessor einen Stack, um Rücksprungadressen und lokale Variablen zu speichern. Dieser Stack wächst typischerweise von oben nach unten:

[4]



Bäume [trees] sind insbesondere für die schnelle Suche in Datenbanken wichtig. Ein Baum besteht aus Knoten [nodes, vertices] und Kanten [edges]. Wenn man beliebig zwei dieser Knoten wählt, muss es dazwischen eine Verbindung geben und darf es aber auch nur eine einzige Verbindung geben. Normalerweise gibt es einen besonderen Knoten: die Wurzel [root]. Diese hat Kindknoten [child nodes], diese wieder Kindknoten usw. und ganz außen hängen „Blätter“ [leaves], das sind Knoten ohne Kinder. Man sortiert eine Datenbank in einem Baum vor und kann dann Anfragen relativ schnell beantworten:

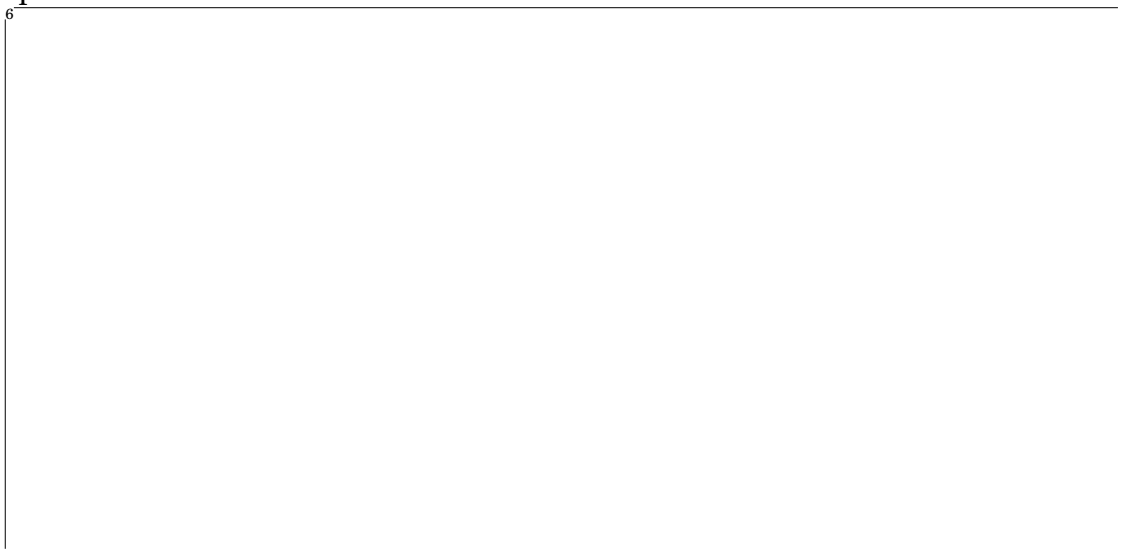
5


Bäume sind in C standardmäßig nicht enthalten, in C++ nur indirekt (`map`).

2 Algorithmen

Ein Algorithmus [algorithm] ist eine Prozedur. Der Begriff stammt vom Namen des Mathematikers Abu Dscha'far Muhammad ibn Musa al-Chwarizmi (Bagdad, um das Jahr 800). Ein Programm ist ein für einen Computer ausführbar gemachter Algorithmus – oder eine Sammlung von Algorithmen. Ein Algorithmus *muss* eine Beschreibung endlicher Länge haben und darf zu jedem Zeitpunkt nur endlich viel Speicherplatz belegen. Ob ein Algorithmus nach endlicher Zeit beendet sein muss, kann man diskutieren: In Regelungs- und Steuerungssystemen sind Endlos-Schleifen üblich. Algorithmen *dürfen* Zufallselemente enthalten.

Im allerersten Praktikum hatten wir einen einfachen Algorithmus, um in einer Folge von Werten den größten zu finden und seine Position in der Folge zu speichern:

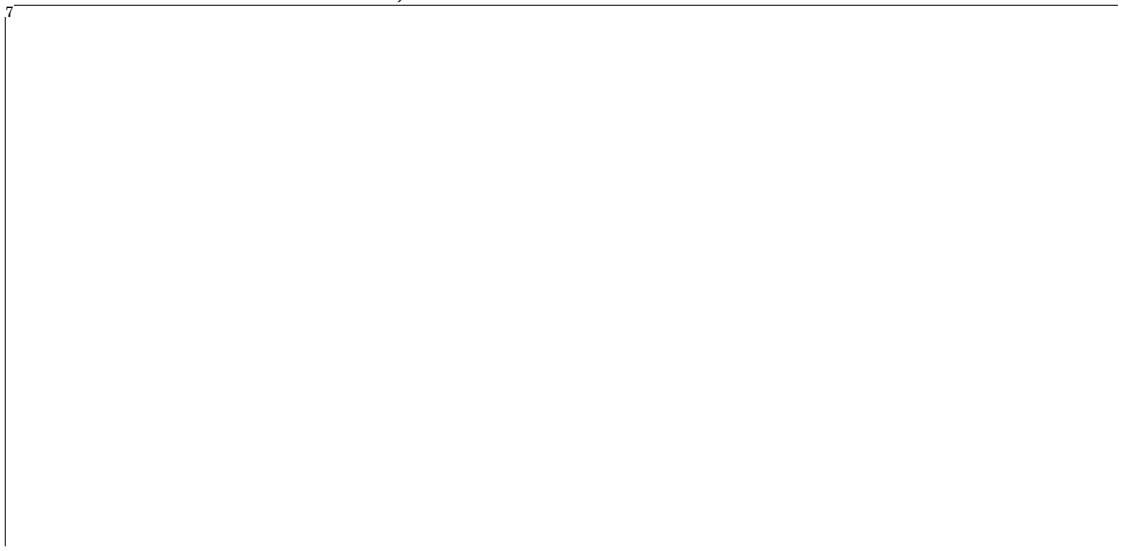
6


2.1 Suchen und Sortieren. Laufzeit

Das Suchen nach Daten und das Sortieren von Daten sind große Themen in der Informatik. Zwei Beispiele: Bubblesort und Quicksort.

Bubblesort geht eine Liste mehrmals von einem zum anderen Ende durch, vergleicht dabei jeden Eintrag mit seinem Nachbarn und vertauscht die beiden, wenn ihre Reihenfolge falsch ist. Die großen Elemente bewegen sich dabei wie Blasen im Wasser nach oben, daher der Name:

[7]



Dieser Algorithmus ist sehr einfach. Er eignet sich aber nur für kleine Mengen an Daten, weil die Liste typischerweise sehr häufig von vorne bis hinten abgearbeitet werden muss: Die Zahl an Schritten, die der Algorithmus im Mittel für eine Liste mit n Einträgen benötigt, wächst asymptotisch wie n^2. Man sagt: Der Bubblesort-Algorithmus hat eine mittlere Laufzeit von $O(n^2)$. Das soll heißen, dass die mittlere Laufzeit – für n groß genug – maximal eine feste Konstante mal n^2 ist.

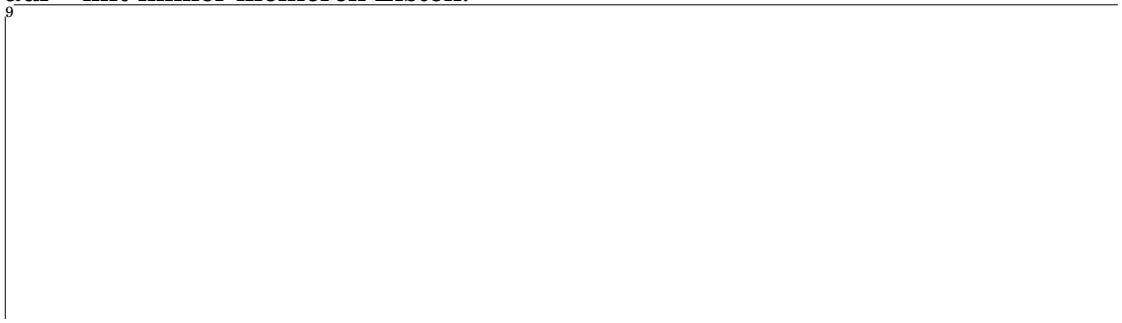
Der Quicksort-Algorithmus geht intelligenter vor, verlangt aber mehr Programmierarbeit. Allerdings gibt es ihn in C auch bereits fertig unter dem Namen `qsort` zu finden. Die Deklaration ist in `<stdlib.h>`. Der wesentliche Gedanke ist, sich ein „Pivotelement“ aus der Liste zu suchen, dann die Liste grob zu sortieren, so dass alle Elemente, die kleiner sind als das Pivotelement, links davon stehen und alle Elemente, die größer sind, rechts davon. Auf die Teillisten links und rechts vom Pivotelement wendet man diesen Schritt abermals an usw.

[8]

Dieser Algorithmus hat eine mittlere Laufzeit von $O(n \log n)$, ist also für Listen mit einer großen Zahl n an Einträgen wesentlich schneller als Bubblesort. (Welche Basis > 1 man für den Logarithmus in $O(n \log n)$ nimmt, ist egal, weil sich das Ergebnis nur um einen konstanten Faktor ändern würde.) Allerdings gilt $O(n \log n)$ für Quicksort nur im Mittel. Wenn die Daten zu Beginn sehr unglücklich liegen, kann auch der Quicksort auf $O(n^2)$ abgebremst werden („worst-case performance“).

2.2 Iteration und Rekursion

Bubblesort und Quicksort sind auch Beispiele für zwei grundsätzliche Vorgehensweisen von Algorithmen: Iteration und Rekursion. Iteration ist, was man in C mit Schleifen macht: Eine Befehlsfolge wird wiederholt. So geht der Bubblesort-Algorithmus mehrfach durch die Liste. Rekursion passiert in C, wenn sich eine Funktion selbst aufruft. So ruft sich der Quicksort-Algorithmus selbst auf – mit immer kleineren Listen.



2.3 Komplexität

Als „Zeitkomplexität“ eines Problems gibt man an, welchen Zeitaufwand ein optimal schneller Algorithmus hat – asymptotisch für eine wachsende Problemgröße (z. B. Länge der Eingabe) n. Das Suchen in einer Liste von n Elementen hat also eine Zeitkomplexität von $O(n \log n)$. Probleme mit Zeitkomplexität $O(n^3)$ werden mit wachsendem n schnell praktisch unlösbar, erst recht Probleme mit Zeitkomplexität $O(2^n)$.

Ein berühmt-berüchtigtes und immer noch ungelöstes Problem der theoretischen Informatik ist die Frage $P \stackrel{?}{=} NP$. Dabei ist P die Menge der Probleme, die in polynomialer Zeit gelöst werden können, also von der Art $O(n^{42})$ usw. sind. Und NP ist die Menge der Probleme, die (zufällig!) in polynomialer Zeit von einem nichtdeterministischen Computer gelöst werden können – ein Computer, der würfeln darf. Das ist eine recht künstliche Konstruktion. Man kann sich NP aber auch anders vorstellen: Das sind die Probleme, für die man in polynomialer Zeit prüfen kann, ob etwas Gewürfeltes eine Lösung ist. Ob eine Lösung der Frage $P \stackrel{?}{=} NP$ auch praktische Auswirkungen hätte, ist unklar.

2.4 Programmoptimierung

Auf der praktischen Seite gibt es zur Verbesserung der Laufzeit von Programmen insbesondere diese Maßnahmen:

- Optimierung im Compiler, zu finden unter Options > C/C++ compiler > Optimizations. Hier versucht der Compiler zum Beispiel, mehrfach vorkommende Ausdrücke nur einmal auszuwerten oder kurze Funktionen nicht aufzurufen, sondern in den Code zu kopieren (Inlining). Beim Einzelschritt-Debuggen sorgen diese Änderungen allerdings für Verwirrung, weil der originale Programmcode nicht mehr sagt, was wirklich passiert.
- Der Profiler, zu finden beim Debuggen unter View > Profiling. Hier kann man erfahren, welche Funktion wie häufig aufgerufen wurde und wie viel Zeit das gekostet hat. So kann man einschätzen, welche Funktionen man überarbeiten sollte, weil sie den Löwenanteil der Rechenzeit benötigen.