Grundlagen der Informatik

Me

WS 20/21

Inhaltsverzeichnis

1 Grundlagen

1.1 Turing Halteproblem

TODO:

2 Datenobjekte und -typen

2.1 Definition

Datenobjekte

Bei Datenobjekten unterschiedet man zwischen zwei verschiedenen Kategorien, variable und konstante Datenobjekte. Der Unterschied ist dass sich variable Datenobjekte währen der Laufzeit des Programms ändern können, während konstante Datenobjekte sich eben nicht ändern können.

Datentypen

Bei Datentypen unterschiedet man zwischen 3 Arten:

Elemtare Datentypen
: Nicht weitere zerlegbare (auch atomar genannt), int, char float etc

Strukturierte Datentypen
: Weisen eine Struktur auf und lassen sich 'ausseinanderbauen', string lässt sich in chars zerlegen

Referenzen
: Weißen auf andere Daten hin (pointer)

2.2 Kommazahlen

2.2.1 Festkomma

Bei festkommazahlen wird die komplette Kommazahl (Vor- und Nachkommastelle) abgespeichert. Wo sich das Komma befindet wird nicht in der Zahl angeben, sondern vom Rechner interpretiert. So kann

$$11111111_2 = 1111111.1_{2,1} = 127.5_{10} \tag{1}$$

ergeben, wenn jetzt die Position des Kommas aber anders interpretiert wird kann

$$11111111_2 = 1111.1111_{2,4} = 15.9375_{10} \tag{2}$$

Das Problem mit Festkommazahlen ist, dass die Bits nicht effizient genutzt werden können. Hat man 8 Bit zu Verfügung und und setzt das Komma in der Mitte kann eine Zahl wie 100 nicht abgespeichert werden, da nur 4 Bit für die Vorkommazahl verfügbar sind, obwohl die Nachkommabits nicht gebraucht werden.

2.2.2 Gleitkommazahlen

Bei Gleitkommazahlen gibt es 2 Standards: float(32 Bit) und double (64 Bit). Das MSB beim float ist die Angabe des Vorzeichen, darauf folgen 8 Bit für den Exponenten und die restlichen 27 Bit sind die Mantisse [1].
Ist das Vorzeichenbit gesetzt so ist die Zahl negativ. Bei Exponenten muss man noch mit einem offset/bias von 127 (1023 bei double). Dies sorgt dafür, dass auch negative Exponenten möglich sind, wodurch eben auch sehr kleine Zahlen möglich. Ist das MSB bei Exponenten eine 1, weiß man direkt dass der Exponenten ≥ 1. Die Mantisse ist im Endeffekt die Zahl, die man darstellen will. Diese wird zuerst normalisiert, dh. solange verschoben, bis die komplette Zahl rechts hinter dem Komma steht[2]. Zudem wird vor der Mantisse eine 1 gespeichert. Diese ist immer implizit angegeben und wird für eine korrekte Berechnung der Kommazahl benötigt.
Daraus folgt die folgende Formel zur Berechnung der eine Gleitkommazahl:

$$2^{Vorzeichen} \cdot 1.Mantisse \cdot 2^{Exponent-Offset} \tag{3}$$

Der IEEE 754 Standard definiert noch einige Spezialwerte:

Exp = 0:
: Vor der Mantisse steht nun eine Null anstatt einer 1, wodurch man auch eine Null darstellen kann. Der offset beträgt jetzt auch 126 um die '0' vor der Mantisse auszugleichen. (Exp = 000000000 $\equiv$ -126)

Exp = 0 und Mantisse = 0 :
: Repräsentiert $\pm\infty$, je nachdem wie das Vorzeichenbit gesetzt ist. Passiert wenn man zB. durch 0 teilt

Exp = 0:
: Repräsentiert NaN. Tritt oft wenn Rechenoperationen nicht definiert sind zB. $\infty - \infty$

2.3 Speicherung von Datentypen

2.3.1 Definitionen

sequentiell

Elemente werden im Speicher direkt hintereinander gespeichert. Der Vorteil ist eine schnelle Zugriffszeit, da man die Position der anderen Elemente berechnen kann jedoch ist das Einfügen bzw. Löschen aufwendig, da meisten ein Teil der Datenstruktur zwischengespeichert wird, das Element hinzugefügt/gelöscht wird und dann der Rest wieder angehängt werden muss[3].

[1] double: 1 Bit VZ, 11 Bit Exponent, 52 Bit Mantisse
[2] Durch den Exponenten wird die Zahl wieder in den 'Normalzustand' verschoben
[3] Beispiel: Arrays

verkettet

Elemente haben hier nicht nur den Wert den sie speichern wollen, sondern haben noch eine Referenz zum nächsten Wert. Die Zugriffszeit leidet hier, da man nicht weiß wo sich die Elemente befinden. Außerdem wird mehr Speicher verbraucht, da die ganzen Referenzen auch noch gespeichert werden müssen. Der Vorteil ist, dass man Objekte einfach einfügen/löschen kann da man hier nur die Referenzen auf ein anderes Element 'umleiten' muss[4].

2.4 Strukturierte Datentypen

Strukturierte Datentypen haben wie der Name schon sagt eine Struktur, dh. dass die aus anderen Datenstrukturen oder auch aus atomaren Datenobjekten bestehen können (Ein String ist zum Beispiel ein strukturierter Datentyp der aus chars besteht). Hier unterschiedet wiederum zwischen 2 verschiedenen Datentypen.

Arrays (oa. Felder)

Ein Array speichert seine Elemente sequenziell. Der Zugriff folgt über einen Index und die Anzahl der Elemente ist fix. Außerdem müssen die Elemente des Arrays den selben Typ haben. Zudem kann ein Array eine oa. mehrere Dimensionen haben. Der Unterschied ist eigentlich nur der Zugriff auf die Elemente. Bei zwei Dimensionen ist kann man sich ein Array mit Zeilen und Spalten vorstellen. Die Spalten werde dann einfach nur hintereinander gespeichert.

Compound

Ein Compound speichert seine Elemente auch sequenziell, jedoch können die Elemente verschiedene Datentypen haben. Die Anzahl ist auch fix und der zugriff verläuft ebenso durch einen Index. Hier kann aber die Position der anderen Elemente nicht berechnet werden, da ja verschieden Datentypen vorkommen können und diese nicht immer gleich viel Speicher brauchen.
Ein Beispiel ist die Datenstrukturen Dictionary (oa. Assoziatives Datenfeld). Der Zugriff von den Werten verläuft durch Angabe eines sogenannten Keys, wie wenn man in einem Wörtebuch eine Definition eines Wortes haben will.

2.5 Dynamische Datenstrukturen

Dynamische Datenstrukturen können ihren Inhalt, Beziehung zueinander und auch ihre Größe während der Laufzeit ändern.

Listen

Objekte von Listen speichern ihre Daten und eine Referenz zu einem andere Objekte in der Liste.

[4]Beispiel: Linked List

Linked List
Bei einer Linked ist das erste Objekt nur ein Dummyobjekt. Das Dummyobjekt speichert an sich keine Daten sondern nur die Referenz aufs nächste Objekt. Dadurch tritt nie der Sonderfall auf, dass die Liste leer. Dies ist besonders bei Double Linked List hilfreich (Objekt hat zusätzliche Referenz auf den Vorgänger). Bei einer DLL ist dann das Dummyobjekt Anfang und Ende der Liste.
Stack
Eine Stack ist eine Liste die Nach dem LIFO (Last In First Out) Prinzip arbeitet. Zur Änderung der Datenstruktur stehen nur push (Objekt hinzugefügen) und pull (Objekt wieder runternehmen) als Methoden zur Verfügung welche sich immer auf zuletzt geänderte Element des Stacks beziehen.
Queue
Ähnlich wie der Stack nur wird hier nach dem FIFO (First In First Out) Prinzip gearbeitet, d.h das Objekt welches als erstes hinzugefügt wurde ist das erste Objekt, welches verarbeite wird. Die Methoden sind dementsprechen enqueue und dequeue

Bäume

Ein Baum besteht aus Knoten welche auf weitere Knoten zeigen können, einer Wurzel (Knoten ohne Vorgänger) und Blätter (Knoten ohne Nachfolger).

Binärbäume

Ein Binärbaum hat die besondere Eigenschaft, dass jeder Knoten maximal 2 weiter Knoten haben darf. Hier versucht man auch idR den Baum zu Ordnen, als dass das gilt:

$$\text{linke kind} < \text{Wurzel} \leq \text{rechtes Kind}.$$

Bei den Binärbäumen gibt es weitere Spezialsierungen:

Vollständiger Baum Jeder Knoten hat 2 oder keine Nachfolger

Perfekter Binärbaum Alle Blätter sind auf der selben Ebene

TODO Zugriffszeit etc Ein degenerierter ist ein (Teil-)Baum welcher nur einen Nachfolger hat und somit im Prinzip zur einer Liste wird. Dadurch gehen einige Vorteile der Baumstruktur verloren. Außerdem gibt es noch den ausgewogener Baum (auch AVL-Baum genannt). Hier darf der Höhenunterschied der Teilbäume eines Knotens maximal 1 sein. Möchte man auf die Elemente eines Baumes zugreifen wurden verschiedene Ansätze ausgearbeitet:

pre order: Wurzel → linkes Kind → rechtes Kind

in order: linkes Kind → Wurzel → rechtes Kind

post order: linkes Kind → rechtes Kind → Wurzel

level order:

3 Laufzeitkomplexität

3.1 Definition

Bei der Laufzeitkomplexität untersucht man wie schnell ein Algorithmus ein bestimmtes Problem lösen kann. Hier schaut man nicht wie viele Sekunden ein Algorithmus braucht sondern eher wie sich ein ALgorithmus verhält, wenn man die Problemgröße erhöht (zB.: ein Problem ist das Sortieren und die Problemgröße wären die Anzahl der Zahlen die man sortieren möchte). Die Problemgröße wird dabei als 'n' bezeichnet.
Das n bildet dann die Zeitfunktion

$$t = f(n) \tag{4}$$

Hier muss aber noch eine Fallunterscheidung gemacht werden. Manche Algorithmen verhalten sich unter bestimmten Umständen anders (und somit auch ihr Laufzeitverhalten). Man unterschiedet zwischen Best Case (minimale Anzahl an benötigten Schritten), Worst Case (minimale Anzahl an benötigten Schritten) und den Average Case (durchschnittliche Anzahl an benötigten Schritten). Manche Sortieralgorithmen können brechen dass sortieren ab, wenn die Daten schon sortiert sind, womit sie ein besseres Best Case Szenario haben als im Average und Worst Case.

3.2 Klassenbildung

Um jetzt die Laufzeitkomplexität eines Algorithmus zu spezifizieren hat man sogenannte Kassen gebildet in die man einen Algorithmus klassifiziert:

$\mathcal{O}$-Notation: auch obere Schranke genannt. Wenn $c \cdot g(n) \geq f(n)$ gilt für ein beliebiges c dann ist $f(n) = \mathcal{O}(g(n))$. Das n kann auch beliebig gewählt werden solange sich f und g ab diesem n nicht mehr schneiden[5].

Ω-Notation: auch untere Schranke genannt. Wie bei $\mathcal{O}$ nur gilt jetzt $c \cdot g(n) \leq f(n)$.

Θ-Notation: auch enge/harte Schranke: sozusagen $\mathcal{O}$ und Ω zusammen, also $c \cdot g(n) \leq f(n) \leq d \cdot g(n)$.

3.3 Rechenregeln

1. $f = \mathcal{O}(f)$
2. $f, g = \mathcal{O}(F) \Rightarrow f + g = \mathcal{O}(F)$
3. $f = \mathcal{O}(F) \Rightarrow c \cdot f = \mathcal{O}(F)$
4. $f = \mathcal{O}(F), g = \mathcal{O}(f) \Rightarrow g = \mathcal{O}(F)$
5. $f = \mathcal{O}(F), g = \mathcal{O}(G) \Rightarrow f \cdot g = \mathcal{O}(f \cdot g)$
6. $f = \mathcal{O}(F \cdot G) \Rightarrow f = |F| \cdot \mathcal{O}(G)$
7. $f = \mathcal{O}(F)$ und $|F| \leq |G| \Rightarrow f\mathcal{O}(G)$

[5]Davor kann es aber zu Schnittpunkten kommen

3.4 Rekursion

Bei Rekursiven Algorithmen benutzt man eher die Schreibweise $T(n)$ anstatt $\mathcal{O}$. Zur Ermittlung der Laufzeit hat sich dann folgende Formel geformt:

$$T(n) = A \cdot T(\frac{n}{b} + f(n)) \tag{5}$$

- a ist dabei die Anzahl der Teilprobleme die der Algorithmus zerlegt (oder auch Anzahl der rekursiven Aufrufe)
- b beschreibt das Teilproblem verglichen zum vorherigem Problem (FAktor in die das Problem pro Aufruf zerlegt wird)
- f(n) sind die nicht rekursiven Aufrufe des Algorithmus

Substitutionsmethode

Bei der Substitutionsmethode setzt man T(n) in $\frac{n}{b}$ ein. Dies macht man so oft bis man eine Regelmäßigkeit findet. Dadurch kann man eine Vermutung anstellen wie sich der Algorithmus wahrscheinlich verhalten wird. Diese Vermutung muss dann noch per Vollständiger Induktion bewiesen werden[6].

Rekursionsbaum

Beim Rekursionsbaum müssen vier Schritte angewendet werde.:

1. Als erstes muss der Rekursionsbaumaufgestellt werden. Man schaut sich das $f(n)$ an und arbeitet mit diesem weiter. Zuerst wird das $f(n)$ zur Wurzel des Baumes. In der Nächsten ebene gibt kommen dann 'a'-fach neue Knoten zu der Wurzel hinzu. Die Werte der neuen Knoten ist dann jeweils $\frac{n}{b}$. Dies wird wiederholt bis man auf $T(1)$ kommt.
2. Nun untersucht man den Baum und schätzt wieder die Höhe, Anzahl der Knoten der Ebene udn die 'Kosten' der Knoten. Ein Knoten ist dabei ein Teilproblem des gesamten Problems. Durchs Aufsummieren aller Knoten einer Ebene erhält man den Aufwand pro Ebene. Wenn man nun jede Ebene Aufsummiert erhält man den Gesamtbedarf.
3. Für die Letzte Ebene wird dann eine Allgemeine Formel aufgestellt
4. DIe Allgemeine Formel muss mit Vollständiger Induktion bewiesen werden

7

[6] Bsp.: Kapitel 3 Folie 122(95)
[7] Bsp.: Kapitel 3 Folie 131(101)

Mastertheorem

Das Mastertheorem ist eine relativ schnelle Methode um die Laufzeit eines rekursiven Algorithmus zu ermitteln, da man hier nur mit Formeln arbeitet. Dabei sind drei Fälle zu beachten:

1. $f(n) \in \prime(n^{E-\epsilon}), \epsilon > 0 \Rightarrow T(n) \in \prime(n^E)$, obere Abschätzung
2. $f(n) \in \Theta(n^E) \Rightarrow T(n) \in (n^E \cdot \log n)$, exakte Abschätzung
3. $f(n) \in \Omega(n^{E+\epsilon}), \epsilon > 0, a \cdot f(\frac{n}{b}) \leq c \cdot f(n), 0 < c < 1 \Rightarrow T(n) \in (n^E)$, untere Abschätzung

Dabei kann es auch vorkommen dass keine der 3 Fälle anwendbar sind, weshalb man dann Die Substitutionsmethode versuchen könnte und dann die

4 Sortieralgorithmen

Algorithmus	Best Case	Average Case	Worst Case
Quicksort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$
Mergesort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
Heapsort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
Selectionsort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$

4.1 Definitionen

stabil Die Reihenfolge von 2 gleichen Elementen ist nach dem Sortieren gleich (wenn das i-te und j-te Element also gleich sind gilt nach dem Sortieren immer noch i < j)

instabil Gegensatz zu stabil, Algorithmus achtet nicht auf Reihenfolge von gleichen Elementen

in place Es wir direkt im Datensatz sortiert

out of place Es wird extern sortiert

4.2 Selectionsort

Das kleinste Element wird gesucht und mit dem Element der ersten Stelle getauscht. Das erste Element gilt als sortiert und man fährt mit diesem Prinzip für die restlichen ELemente fort. In place und stabil. Laufzeit ist $\mathcal{O}(n^2)$.

4.3 Insertion Sort

Das erste Element wird vorerst als sortiert betrachtet. Das Nächste Element wird je nach seinem Wert vor oder hinter dem ersten eingefügt. Analog werden die nächsten Felder bearbeitet. Dabei ist noch zu beachten, dass eine Felde immer nur um eine Stelle verschoben wird Stabil und In-Place. Worst/Average case $\mathcal{O}(n^2)$, best case $\mathcal{O}(n)$

4.4 Bubblesort

Das Feld durchlaufen und benachbarte Felder werden verglichen und ggf. getauscht. Dies wird solange wiederholt, bis das Feld sortiert ist. Da es im Normalfall keine Fallunterscheidung im Bubblesort existiert, wird das Feld n mal durchlaufen und bei jedem Durchlauf werden n ELemente bearbeitet $\rightarrow$ Laufzeit ist $\mathcal{O}(n^2)$. Die Laufzeit kann verbessert werden, da Bubblesort bei jedem Durchlauf immer da Größte Element ans Ende bringt, kann mna den Endbereich auch also sortiert betrachtet und muss nicht mehr überprüfen ob der Bereich sortiert ist. Worst und Average case sind weiterhin $\mathcal{O}(n^2)$ ber der Best case ist jetzt $\mathcal{O}(n)$. Außerdem ist der verbesserte Bubblesort idR. schneller als der normale Bubblesort obwohl beide den selben Average Case haben.

4.5 Quicksort

Quicksort arbeite nach dem divide and conquer Prinzip. Das Feld wird aufgeteilt und die Teilfelder werden einzeln sortiert. Dabei wird ein beliebiges Element hergenommen, das sogenannte Pivotelement. Alle Elemente mit niedrigerer Wertigkeit als das Pivotelement werden links, die mit höherer rechts vom Pivotelement gespeichert. Der Vorgang wird solange wiederholt, bis nicht mehr geteilt werden kann. Das ständige Teilen deutet meistens auf ein logarithmisches Laufzeitverhalten hin, was beim best und average case auch der Fall ist $\mathcal{O}(n \log n)$ Der Worst case kann aber $\mathcal{O}(n^2)$ sein, wenn das Pivotelement schlecht gewählt wurde (ganz links oder ganz rechts), das Problem wird auch als Degeneration bezeichnet.

4.5.1 Pivotelement

Ein gutes Pivotelement zu suchen ist relativ schwer, da das Pivotelement möglichst nah an der Mitte des Medians des Feldes sein sollte. Hierführ könnte man theoretisch erst das ganze Feld durchsuchen, was aber recht aufwändig sein kann. Verschiedene Ansätze wurden hier herausgearbeitet:

Wahl des mitteleren Elements $\frac{LetztesElement - ErstesElement}{2}$, relativ schlecht da Pivot immernoch an erster bzw. letzter Stelle stehen kann

Iteratives Löschen der Minima Die n kleinsten Werte werden an das Ende des Arrays getauscht. Dieser Schritt hat schon eine Laufzeit von $\mathcal{O}(n^2)$, ist also viel zu aufwändig

Sortieren des Feldes Macht Quicksort überflüssig

Partielle Sortierung mit Quicksort (Tony Hoare) Pivot wird zufällig gewählt, nun wird vorerst nur eine Hälfte sortiert. Verringert die Wahrscheinlichkeit des Worst Case ist aber weiterhin möglich.

Median of k Es werden k Elemente zufällig ausgewählt und aus denen wird dann der das Mittlere genommen um somit das Pivotelement zu bestimmen. k muss dabei ungerade sein. Verringert wieder nur die Wahrscheinlichkeit des worst case.

SELECT Algorithmus Das Feld wird in fünfer Gruppen aufgeteilt. Aus jeder Fünfergruppe wird dann das mittlere Element herausgezogen. Aus allen mittleren Element wird wieder das mittlere Element herausgenommen was dann zum Pivotelement wird. Worst case ist immer $\mathcal{O}(n \log n)$ aber aufgrund der Komplexität wird das Verfahren nur bei großen Mengen verwendet (oder bei zeitkritischen Systemen).

4.6 Mergesort

Wie beim Quicksort wird nach dem divide and conquer Prinzip gearbeitet, nur sortiert Mergesort erst wenn aufgeteilt wurde. Das Feld wird rekursiv in der Mitte geteilt. Dann werden je 2 Felder rekursiv gemerged (=zusammensortiert). Dadurch entstehen je nach Iteration Teilfelder die sortiert sind. Teilfelder können auch in die Mitte (bzw. nicht nur am Anfang/Ende) eingefügt werden. Hat ein Laufzeitverhalten von $\mathcal{O}(n \log n)$. Dabei sollte man aber darauf achten dass man nicht zu oft teil (am Anfang), da sich hier die Laufzeit auf $\mathcal{O}(n^2)$ erhöht.

4.7 Heapsort

Selectionsort sucht das kleinste/größte Element und hängt es dementsprecheend an eine Liste. Die Suche ist aber relativ zeitaufwändig. Um die Suche zu beschleunigen baut man sich als Datenstruktur einen Heap auf. Ein Heap ist ein binärer Baum und hat die Eigenschaften, dass die Kinder eines Knotens immer kleiner/größer /je nachdem obs ein Min/Max-Heap ist). Außerdem muss gelten dass die Blätter des Baumes linksbündig sind. Es werden für den Heapsort drei Methoden gebraucht:

- BUILD-MAX-HEAP: Baue einen Max-Heap aus den Daten
- HEAP-EXTRACT-MAX: Nimm das größte Element aus dem Heap
- MAX-HEAPIFY: Nach der entnahme muss der Heap wieder 'geflickt' werden

[8] Build-Heap nimmt einen Datensatz und erstellt damit erstmal einen binären Baum. Wenn ein Array vorliegt kann man direkt einen Baum erstellen da ein Baum ja nur ein Array mit speziellen Zugriffsverfahren ist. Nun wird aus dem Baum das größte Element extrahiert. Dazu geht man in die unterste Ebene des Baumes und vergleicht ob die Knoten größer sind als der Elternknoten. Ist das Element nicht größer wird zur nächsten wird der nächste Teilbaum in der Ebene analysiert. Wenn irgendwo der Kindknoten größer als

[8] Analog gilt dasselbe für Min-Heap

der Elternknoten ist, wird getauscht. So wird Ebene für Ebene analysiert bis die Wurzel erreicht ist. Hier muss aber noch beachtet werden, dass wenn ein Knoten getauscht wurde auch der Ast nochmals betrachtet werden muss. Wenn ein Kindknoten zum Elternknoten wird, kann es ja sein dass der getauschte Knoten größer als sein Elternknoten ist. Der größte Knoten ist jetzt an der Wurzelposition. Dieser wird dann durch Extrakt-Max ans Ende des Baumes (oa. Array) vertauscht. Das größte Element ist nun am Ende des Baumes (Array), wodurch man auch weiß dass das letzte Element sortiert ist. Da die Wurzel aber einen neuen Wert hat muss max-heapify aufgerufen werden, um die Bedingungen zu erfüllen. Nun wird aber das letzte Element ignoriert, da es ja schon sortiert ist. Es wird wieder extract-max aufgerufen, und dann wieder max heapify bis der komplette Baum sortiert ist.

4.8 Shellsort

Wie Insertionsort, allerdings wird über eine größere Distanz verglichen. Die Distanz wird mit jedem Durchgang verringert, bis man am Ende einen Durchgang Insertionsort hat. Erst vorteilhaft bei einer großen Zahl an Elementen, bei 9 Elementen ist normaler Insertionsort besser. Die Folge, wie groß die Distanz reduziert werden soll, hat großen Einfluss auf die Laufzeit

4.9 Combsort

Combsort wendet die Vergleiche über eine größere Distanz aus Shellsort auf Bubblesort an. Die Distanz wird immer weiter verringert bis im letzten Schritt mit Bubblesort sortiert wird.

4.10 Weirdsort

Nur bei gleicher Verteilung, kein echtes Sortierproblem.

4.11 Bucketsort

Die Elemente werden entsprechend ihrer Wertigkeit in Buckets aufgeteilt. Die Buckets werden mit einem anderen Sortieralgorithmus sortiert und wieder aneinander gefügt.

4.12 Radixsort

Für jede Stelle in der Zahl wird ein Bucketsort durchgeführt (hinten anfangend).

4.13 Countingsort

Es wird die Anzahl der Vorkommen jedes Wertes gezählt und dann in einem nächsten Schritt das Array wieder aufgebaut. Sinnvoll wenn viele gleiche Werte vorkommen.

5 Sprachen und Automaten

5.1 Alphabet

Das Alphabet $\sum$ ist eine endliche, nicht leere Menge, die alle Zeichen enthält, die im Alphabet erlaubt sind. Die Kleenesche Hülle $\sum *$ umfasst alle Wörter, die man mit dem Alphabet erzeugen kann, inklusive dem leeren Wort (ϵ). Die Transitive Hülle $\sum +$ enthält alle Wörter außer dem leeren Wort.

$|\omega|$ = Länge des Wortes (=Anzahl der Zeichen)

$\sum n$ = Menge der Wörter der Länge n

$x = 101, y = 010, xy = 101010$ Konkatenation

Eine Teilmenge aus $\sum *$ wird als Sprache bezeichnet. Eine Sprache muss nicht alle Zeichen verwenden. Die Verkettung von 2 Sprachen enthält alle möglichen Verkettungen der Worte der Sprachen. Die Kleensche Hülle einer Sprache enthält alle Zeichenkombinationen, die durch Kombination von Zeichenfolgen der Sprache erzeugt werden können.

5.2 Formale Sprache

Eine formale Sprache ist eine Teilmenge über einem endlichen Alphabet. Eine Sprache ist definiert durch Grammatik. Eine Grammatik beschreibt Ersatzinstruktionen für jedes Zeichen, bis am Ende nur noch Terminalsymbole übrig sind (Vorrausgesetzt es ist endlich). Die Ersetzung beginnt beim Startsymbol S.

$$P = S \to aA, A \to aS|a$$

Das Beispiel erzeugt jedes Wort, das aus einer geraden Anzahl aus a erzeugt werden kann.

Allgemeine Sprache Alle Worte können rekursiv aufgezählt werden, ein Automat kann alle Worte bestätigen. Es gibt keine Grammatik die alle Wörter aufzählt, die nicht zur Sprache gehören. Links vom Pfeil mindestens ein nichtterminal, rechts vom Pfeil beliebig.

Kontextsensitive Sprache Variable darf nur im Kontext einer Zeichenkette ersetzt werden, der Kontext selbst bleibt unverändert. Links vom Pfeil mindestens ein nichtterminal und nie mehr Zeichen als rechts vom Pfeil, Rechts beliebig.

Kontextfreie Sprache Links vom Pfeil immer nur ein einzelnes nichtterminal, rechts beliebig. Alle Programmiersprachen können so beschrieben werden.

Reguläre Grammatik Ein nichtterminal wird gegen ein Terminal und optional ein nichtterminal ersetzt.

Für jede dieser Sprachen kann man einen Automaten konstruieren, der entscheiden kann, ob ein Wort zu einer Sprache gehört.

5.3 Automat

$$A = (Q, \sum, \delta, q_0, F)$$

Q: Menge von Zuständen

$\sum$: Menge von Eingabesymbolen

δ: Übergangsfunktion

q_0: Startzustand

F: Menge der akzeptierter/finaler Zustände

Die Übergangsfunktion δ ordnet jedem paar (q_i, σ_j) aus Zustand q_i und Eingabezeichen σ_j einen neuen Folgezustand q_k zu. $\delta(q_i, \sigma_j) \rightarrow q_k$. Kann auch als Tabelle oder Diagram dargestellt werden. Die erweiterte Übergangsfunktion $\tilde{\delta}$ beschreibt, was mit dem Zustand bei der Eingabe eines Wortes passiert.

Ein Automat gilt als endlich, wenn die Zustandsmenge Q und das Alphabet $\sum$ endlich ist.

Ein deterministisch endlicher Automat akzeptiert eine Eingabe, wenn der letzte Zustand in F enthalten ist.

Bei nicht deterministischen Automaten bildet die Übergangsfunktion auf mehrere mögliche Folgezustände ab.

Die Sprache eines Automaten enthält alle Zeichenketten, die der Automat akzeptiert.

6 Glossar

Bytespeicherung in Big-Endian(höchstes bit an erster (also ganz links) Stelle) bzw Little-Endian NIL Not in List