Programmieren für Mathematiker SS2018

Dozent: Prof. Dr. Wolfgang Walter

8. September 2018

Inhaltsverzeichnis

Vorwort

Schön, dass du unser Skript für die Vorlesung *Programmieren für Mathematiker 2* bei Prof. Dr. Wolfgang Walter im SS2018 gefunden hast!

Wir verwalten dieses Skript mittels Github [1], d.h. du findest den gesamten LaTeX-Quelltext auf `https://github.com/henrydatei/TUD_MATH_BA`. Unser Ziel ist, für alle Pflichtveranstaltungen von *Mathematik-Bachelor* ein gut lesbares Skript anzubieten. Für die Programme, die in den Übungen zur Vorlesung *Programmieren für Mathematiker* geschrieben werden sollen, habe ich ein eigenes Repository eingerichtet; es findet sich bei `https://github.com/henrydatei/TU_PROG`.

Es lohnt sich auf jeden Fall während des Studiums die Skriptsprache LaTeX zu lernen, denn Dokumente, die viele mathematische oder physikalische Formeln enthalten, lassen sich sehr gut mittels LaTeX darstellen, in Word oder anderen Office-Programmen sieht so etwas dann eher dürftig aus.

LaTeX zu lernen ist gar nicht so schwierig, ich habe dafür am Anfang des ersten Semesters wenige Wochen benötigt, dann kannte ich die wichtigsten Befehle und konnte mein erstes Skript schreiben (`1. Semester/LAAG`, Vorsicht: hässlich, aber der Quelltext ist relativ gut verständlich). Inzwischen habe ich das Skript überarbeitet, lasse es aber noch für Interessenten online.

Es sei an dieser Stelle darauf hingewiesen (wie in jedem anderem Skript auch ☺), dass dieses Skript nicht den Besuch der Vorlesungen ersetzen kann. Prof. Walter hat nicht wirklich eine Struktur in seiner Vorlesung, ich habe deswegen einiges umstrukturiert und ergänzt, damit es überhaupt lesbar wird. Wenn du Pech hast, ändert Prof. Walter seine Vorlesung grundlegend, aber egal wie: Wenn du noch nicht programmieren kannst, wirst du es durch die Vorlesung auch nicht lernen, sondern nur durch die Übungen; die Vorlesung ist da wenig hilfreich.

Wir möchten deswegen ein Skript bereitstellen, dass zum einen übersichtlich ist, zum anderen *alle* Inhalte aus der Vorlesung enthält, das sind insbesondere Diagramme, die sich nicht im offiziellen Skript befinden, aber das Verständnis des Inhalts deutlich erleichtern. Ich denke, dass uns dies erfolgreich gelungen ist.

Trotz intensivem Korrekturlesen können sich immer noch Fehler in diesem Skript befinden. Es wäre deswegen ganz toll von dir, wenn du auf unserer Github-Seite `https://github.com/henrydatei/TUD_MATH_BA` ein neues Issue erstellst und damit auch anderen hilfst, dass dieses Skript immer besser wird.

Und jetzt viel Spaß bei *Programmieren für Mathematiker*!

Henry, Pascal und Daniel

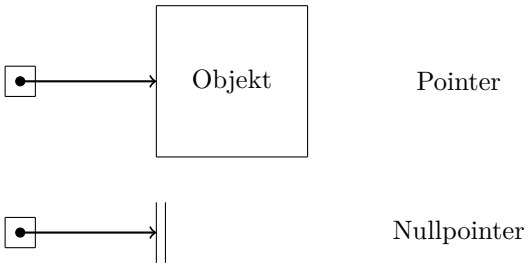
[1] Github ist eine Seite, mit der man Quelltext online verwalten kann. Dies ist dahingehend ganz nützlich, dass man die Quelltext-Dateien relativ einfach miteinander synchronisieren kann, wenn man mit mehren Leuten an einem Projekt arbeitet.

Kapitel I

Pointer

Allgemeines über Pointer

Pointer nennt man auch Zeiger , Verweise oder Datenreferenzen . Ein Pointer ist ein Verweis bzw. eine Referenz auf ein Zielobjekt/Zeigerziel/Target eines festgelegten Datentyps. In den folgenden Darstellungen ist:



Ein Pointer hat zu Beginn der Programmausführung einen undefinierten Zustand, der nicht als solcher erkannt werden kann. Die Verwendung eines solchen Pointers kann große Probleme verursachen.

Zeiger sind kein eigenständiger Typ, sondern nur mit dem Attribut `pointer` gekennzeichnet:

```
1 ! eine normale Variable
2 integer :: variable
3 ! ein Pointer
4 integer, pointer :: ptr
```

Zeiger sind streng typisiert, das heißt man kann nur auf Objekte zeigen, deren Typ identisch mit dem Zeigertyp ist. Es gibt also keine Universalpointer. Der Pointer im oberen Quelltext kann also nur auf Variablen mit dem Typ `integer` zeigen.

Jedes beliebige Objekt vom passenden Objekttyp kann als Ziel eines Zeigers dieses Typs verwendet werden, wenn die Zielvariable das Attribut `target` trägt oder das Objekt ein dynamisches im Heap erzeugtes Objekt ist.

```
1 integer, target :: ziel
2 integer, pointer :: ptr
```

Jede Pointer-Variable kann als Zeigerziel dienen. Ohne `target`-Attribut.

Implizit werden Pointer immer automatisch dereferenziert, außer in den Anweisungen `nullify()`, `allocate()`, `deallocate()`, der Pointer-Zuweisung `pointer => ziel` sowie in der `associated`-Abfragefunktion.

Pointer sind in Fortran in der Regel mehr als nur Adressen.

Werfen wir nun nochmal einen Blick auf die Pointer-Kontexte, in denen Pointer automatisch dereferenziert werden.

Anmerkung
Wird gerne in der Klausur abgefragt, steht aber auch in dem zur Klausur zugelassenen Buch des Rechenzentrums Niedersachsen über den Fortran-Standard.

- Die Funktion `nullify(p1, p2, ...)` versetzt die Pointer `p1`, `p2` und so weiter in den definierten Zustand Null = nicht assoziiert.
- `allocate(p1, p2, ...)` legt Speicherblöcke im Heap für die Zielobjekte der Pointer an und setzt die Pointer als Referenzen auf ihren jeweiligen Speicherblock. Alle Pointer sind im definierten Zustand assoziiert.
- Mit `deallocate(p1, p2, ...)` werden die Speicherblöcke, auf die die Pointer zeigen freigegeben und die Pointer auf Null gesetzt. Der Pointer muss dafür assoziiert und ein ganzen Objekt, also kein Subarray, Substring oder ähnliches, sein.
- Pointer werden mit `ptr => tgt` oder `ptr1 => ptr2` zugewiesen.
- Die Abfragefunktion `associated()` kann auf recht unterschiedliche Weisen eingesetzt werden:
  - `associated(ptr)` → `.true.`, wenn auf ein Ziel gezeigt wird; `.false.`, wenn `ptr` auf Null zeigt.
  - `associated(ptr, tgt)` → `.true.`, wenn `ptr` auf `tgt` zeigt, sonst `.false.`
  - `associated(ptr1, ptr2)` → `.true.`, wenn beide Pointer denselben Zustand (nicht Null) haben, sonst `.false.`

Wie schon oben angesprochen, ist der Umgang mit Pointern nicht ganz ungefährlich, es gibt einige Gefahren für den Hauptspeicher, insbesondere den Heap.

Anmerkung
auch wichtig in der Klausur, steht aber leider nicht im Buch, muss also auswendig gelernt werden

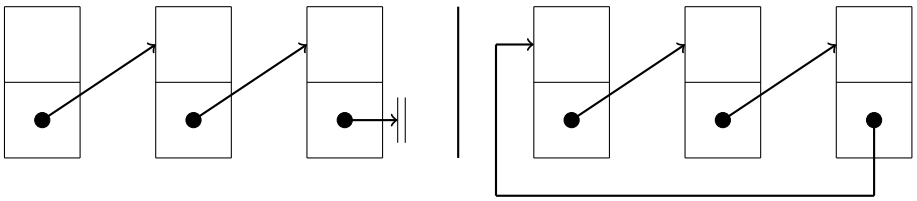
- Verwendung eines nicht definierten oder nicht gültigen Pointers in `deallocate`, `=>`, `associated`-Abfragen und normalen (nicht Pointer-) Kontext, das heißt in Expressions, in denen alle Pointer automatisch dereferenziert werden.
- Dangling Pointer entstehen, wenn das Zeigerziel verloren geht, z.B. durch `deallocate` über anderen Pointern oder eines `allocatable`-Feldes oder wenn das Zielobjekt "out of scope" geht, zum Beispiel durch Verlassen seiner Prozedur.
- Speicherleichen , Garbage, memory leaks: haben im Prinzip das ewige Leben im Heap, wenn keine Referenzen mehr auf ein Heap-Objekt existiert, über die man es freigeben könnte.

Listen

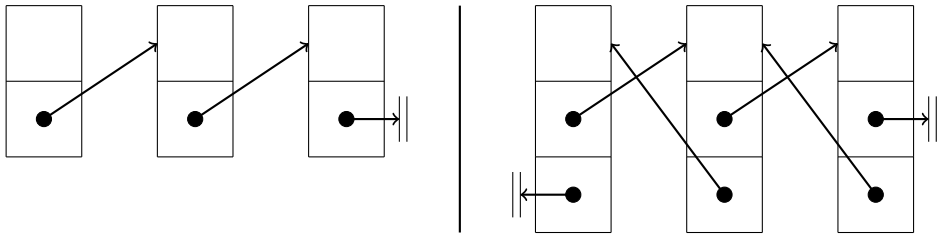
Eine Liste ist eine lineare Anordnung von Objekten des selben Typs. Eine Liste wird als verzeigerte Struktur (oder als eindimensionales Feld - wird hier aber nicht behandelt) implementiert. Eine solche Liste hat 3 Attribute:

- linear vs. zyklisch
- einfach verkettet vs. doppelt verkettet
- endogen vs. exogen

linear vs. zyklisch



einfach verkettet vs. doppelt verkettet



Eine Liste hat immer gewisse Einfüge- und Löschoperationen. Wenn diese an beiden Enden der Liste notwendig sind, spricht man von einer Deque = double-ended-queue.

Grundoperationen auf einer Liste

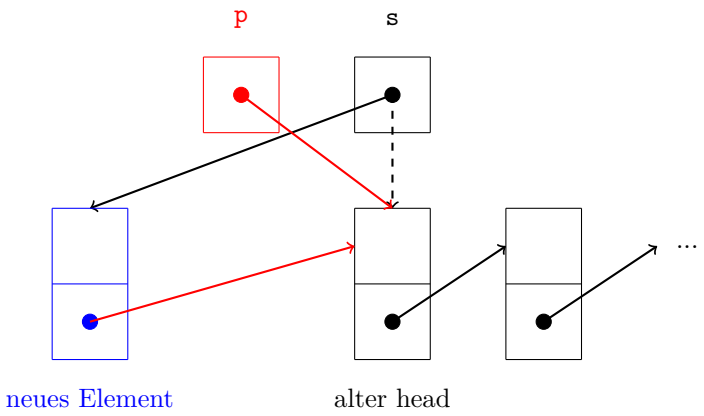
`init(L)`	Initalisierung der Liste, Anfangszustand "leer"	
`empty(L)`	als Abfragefunktion → `.true.` falls `L` leer, sonst `.false.`	
`access_head(L,e)`	als Subroutine, gibt in `e` den Wert des head-Elements	head = Beginn einer Liste
`val_head(L)`	als Funktion → Ergebnis ist Inhalt des head-Elements	
`access_tail(L,e)`	als Subroutine, gibt in `e` den Wert des tail-Elements	tail = Ende einer Liste

`val_tail(L)`	als Funktion → Ergebnis ist Inhalt des tail-Elements	
`val_elem(L,p)`	liefert Inhalt des durch `p` referenzierten Elements	
insert		
`insert_head(L,e)`	Einfügen des Elements `e` am Anfang der Liste `L`	anderer Name: `push`
`insert_tail(L,e)`	Einfügen des Elements `e` am Ende der Liste `L`	anderer Name: `inject`
`insert_after(L,p,e)`	Einfügen des Elements `e` nach dem von `p` referenzierten Element	
`insert_before(L,p,e)`	Einfügen des Elements `e` vor dem von `p` referenzierten Element	
delete		
`del_head(L,e)`	Löschen des Elements `e` am Anfang der Liste `L`	anderer Name: `pop`
`del_tail(L,e)`	Löschen des Elements `e` am Ende der Liste `L`	anderer Name: `eject`
`del_after(L,p,e)`	Löschen des Elements `e` nach dem von `p` referenzierten Element	
`del_elem(L,p,e)`	Löschen eines Elements `e`, welches von `p` referenziert wird	
Traversieren (Durchlaufen aller Elemente) der Liste `L` und Ausführen einer Task `T` auf jedem Element		
`trav_forward(L,T[,p])`	vorwärts, optional ab dem von `p` referenzierten Element	
`trav_backward(L,T[,p])`	rückwärts, optional ab dem von `p` referenzierten Element	
Suchen eines Elements mit dem Inhalt `e`		
`find_forward(L,e[,p])`	vorwärts, optional ab dem von `p` referenzierten Element	
`find_backward(L,e[,p])`	rückwärts, optional ab dem von `p` referenzierten Element	

Grundoperationen auf einer Deque

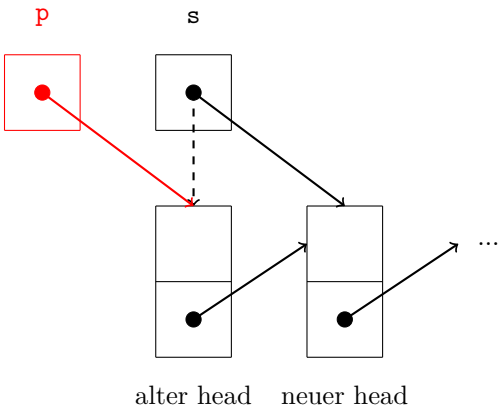
Der einfachste Fall ist der einer linearen, nicht zyklischen, einfach verketteten, endogenen Liste mit `s` als head-Pointer.

`push(s,elem)`



Zuerst haben wir die schwarze Liste mit `s` als heap-Pointer. Für spätere Verwendung setzen wir noch den Nachfolger eines p-Pointer auf den head. Jetzt wird das neue Element eingefügt und der `s`-Pointer zeigt auf den neuen head. Der Nachfolger des neuen head muss nun noch auf p zeigen, was ja auf den alten head zeigt. Schon ist das neue Element eingebunden.

`pop(s,elem)`



Wir haben wieder die schwarze Liste mit `s`-Pointer. Um auf den alten head zugreifen zu können, benutzen wir wieder den Hilfspointer p. Den `s`-Pointer setzen wir dann auf den Nachfolger des alten heads.

`inject(t,elem)`

Der Nachfolger des alten tails zeigt nun auf das neue Element. Dann muss nur noch der tail-Pointer angepasst werden und der Nachfolger des neuen tails muss mit `nullify` auf Null gesetzt werden.

`eject(t,elem)`

Hier bekommen wir ein Problem! Nicht das es nicht möglich wäre das letzte Element zu löschen, aber der Vorgänger des tail-Elements kann nur gefunden werden, indem man die ganze Liste durchläuft. Das heißt die Laufzeitkomplexität dieser Operation beträgt $T(n) = \mathcal{O}(n)$. Die Dauer dieser Operation ist also von der Listenlänge abhängig!

Queues

Eine Queue ist eine Warteschlange und sollte mit `pop` und `inject` implementiert werden. Es ist dabei in 2 verschiedene Prinzipien zu unterscheiden:

- Beim FIFO-Prinzip (first-in-first-out) wird das erste Element, was in die Warteschlange kommt, bearbeitet und verlässt die Warteschlange (so wie bei der Kassenschlange in der Mensa).
- Beim LIFO-Prinzip (last-in-first-out) wird das Element, was zuletzt in die Warteschlange kommt, bearbeitet (nach dem Prinzip bearbeite ich Mails: die neuste beantworte ich zuerst).

☺ Professor Walter bevorzugt übrigens das LIFO-Prinzip in der Mensa. Er kommt zuletzt, hat aber als Erster sein Essen. ☺

Weiterhin gibt es noch Output-resticted-queues bzw. Input-restricted-queues. Das sind deques mit `push`, `pop`, `inject`, aber ohne `eject` bzw. eine deque mit `pop` und `eject` oder `push` und `inject`.

Grundoperationen auf einer Queue

Eine Queue hat 4 wichtige Funktionen:

- `init(Q,n)`
- `empty(Q)` bzw. `full(Q)`
- `enqueue(Q,neu)`: `inject` am tail
- `dequeue(Q)`: `pop` am head

Implementiert wird dies mit einem eindimensionalen Feld mit `maxlengh`, `index_head`, `index_tail` und `elems` (Pointer auf Feld `Q`). Hier sind die notwendigen Funktionen nur angedeutet, Details kann sich jeder selber denken.

```
1  subroutine init(Q,n)
2   type(queue) :: Q
3   integer :: n
4
5   allocate(Q(0:n-1))
6   maxlengh = n
7   index_head = 0
8   index_tail = n-1
9  end subroutine init
10
11 function empty(Q)
12  type(queue) :: Q
13  logical :: empty
14
15  empty = mod(index_tail+1,n) == index_head
16
17 end function empty
```

```
18
19 function full(Q)
20 type(queue) :: Q
21 logical :: full
22
23  full = mod(index_tail+2,n) == index_head
24  ! ein Element bleibt ungenutzt
25
26 end function full
27
28 subroutine enqueue(Q,neu)
29  type(queue) :: Q
30  type(element) :: neu
31
32  ! ...
33  index_tail = mod(index_tail+1,n)
34 end subroutine enqueue
35
36 subroutine dequeue(Q)
37 type(queue) :: Q
38
39 ! ...
40 index_head = mod(index_head+1,n)
41 end subroutine dequeue
```

Rechenaufwand für Grundoperationen

	einfach verkettet		doppelt verkettet	
Grundoperationen	linear	zyklisch	linear	zyklisch
`access_head`	konstant	konstant	konstant	konstant
`push`	konstant	konstant	konstant	konstant
`pop`	konstant	konstant	konstant	konstant
`access_tail` [mit tail-Pointer]	$\mathcal{O}(n)$ [konstant]	konstant	$\mathcal{O}(n)$ [konstant]	konstant
`inject` [mit tail-Pointer]	$\mathcal{O}(n)$ [konstant]	konstant	$\mathcal{O}(n)$ [konstant]	konstant
`eject` [mit tail-Pointer]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$ [konstant]	konstant
`insert_before`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	konstant	konstant
`insert_after`	konstant	konstant	konstant	konstant
`del_elem`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	konstant	konstant
`del_after`	konstant	konstant	konstant	konstant
`trav_forward`	konstant (pro Element)	konstant (pro Element)	konstant (pro Element)	konstant (pro Element)
`trav_backward`	$\mathcal{O}(n)$ (pro Element)	$\mathcal{O}(n)$ (pro Element)	$\mathcal{O}(n)$ (pro Element)	$\mathcal{O}(n)$ (pro Element)

Kapitel II

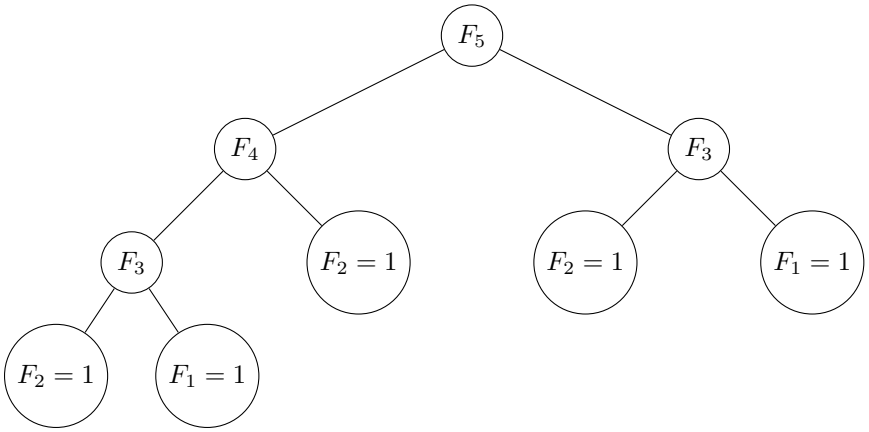
Bäume/Trees und Rekursion

Ein Baum ist entweder leer oder besteht aus einer endlichen Menge von Knoten mit einem speziell ausgezeichneten Wurzelknoten (root) und einer endlichen Anzahl von Teilbäumen.

Ein Baum ist also rekursiv definiert und besitzt eine rekursive Datenstruktur. Wir brauchen deswegen rekursive Algorithmen zur Bearbeitung.

Der Grad ist die Anzahl der Verzweigungen nach unten.
Das Level ist die Anzahl der Ebenen, beginnend bei der Wurzel mit 0.
Die Höhe eines Baums ist die Weglänge zum weitest entfernten Knoten.

Wir wollen die Wechselbeziehung zwischen einer rekursiven Datenstruktur und einem rekursiven Algorithmus untersuchen. Dazu ist es notwendig zu wissen, dass man die Ausführung rekursiver Algorithmen als Baum darstellen kann, so zum Beispiel die FIBONACCI-Zahlen. Sei dazu F_n die n-te FIBONACCI-Zahl:



Man unterscheidet dabei in die Rechtsrekursion und die Linksrekursion .

Rechtsrekursion

Ein Problem P_n lässt sich durch Ausführen eines Tasks T_n das betrachten des Problems P_{n-1} lösen.

Also lässt sich P_n durch Ausführung von $T_nT_{n-1}\dots T_1T_0$ lösen. Diese Rekursion ist leicht auflösbar. Das Problem lässt si

Bei einer allgemeinen Rekursion sieht das Problem $P_{n,j}$ so aus.

$$P_{n,j} = \begin{cases} T_0 & n = 0 \\ T_0P_{n-1,1}T_1P_{n-1,2}\dots T_{k-1}P_{n-1,k}T_k & n > 0 \end{cases} \quad 1 \le i \le k$$

Auch hier genügt die Abarbeitung einem Stack!

Ein Binärbaum ist ein Baum mit maximalem Knotengrad 2.

- maximale Anzahl an Knoten auf dem Level l: 2^l

- maximale Anzahl Knoten auf dem gesamten Baum: $N = \sum_{l=0}^{h} 2^l = 2^{h+1} - 1$
- minimale Höhe eines Baums mit N Knoten: $h_{min} = \lfloor \log_2 N \rfloor$

Binäre Suchbäume

Ein binärer Suchbaum ist ein Binärbaum, bei dem im linken Teilbaum eines Knotens nur "kleinere'Ëlemente und im rechten Teilbaum nur "größere'Ëlemente gespeichert sind. Dabei gibt es immer eine besondere Datenkomponente, die als Schlüssel (Key) dient und deren Werte eine vollständige Ordnung ermöglichen (Ordnungsrelation, typischerweise $<$).

Die elementaren Operationen auf Binärbäumen sind:

- Traversieren:
  - Preorder : $P(B) = T(B)P(B_L)P(B_R)$
  - Inorder : $P(B) = P(B_L)T(B)P(B_R)$
  - Postorder : $P(B) = P(B_L)P(B_R)T(B)$
  - Levelorder : schichtweises Durchlaufen von oben nach unten, von links nach rechts
- Einfügen und suchen: Beim Durchlaufen (Traversieren) in Inorder erhält man die in aufsteigender Schlüsselreihenfolge sortierten Elemente/Knoten(inhalte).
- Löschen eines Knotens mit gesuchtem Schlüsselwert im Suchbaum:
  - Blatt löschen ist einfach (keine Teilbäume)
  - Knoten hat genau einen Teilbaum: listenartige Reparatur
  - innerer Knoten mit 2 nichtleeren Teilbäumen: 2 Möglichkeiten
    - größeres Element im linken Teilbaum (des zu löschenden Knotens) suchen, dieses hat rechten Teilbaum ⇒ diesem Knoten durch seinen linken Teilbaum ersetzen, Inhalt dieses (ersetzten) Elements in den „zu löschenden" Knoten kopieren, sodann dieses größere Element (d.h. seinen Knoten) mittels 1 oder 2 löschen (Speicher freigeben!)
    - kleines Element im rechten Teilbaum (des zu löschenden Knotens) suchen, dieses hat linken Teilbaum ⇒ diesem Knoten durch seinen rechten Teilbaum ersetzen, Inhalt dieses (ersetzten) Elements in den „zu löschenden" Knoten kopieren, sodann dieses kleinere Element (d.h. seinen Knoten) mittels 1 oder 2 löschen (Speicher freigeben!)

Ein Sentinel (Wachposten) ist ein Knoten, der den zu suchenden Schlüssel enthält. Alle Nullpointer eines Trees zeigen auf den Sentinel. Das sorgt dafür, dass, wenn man einen Schlüssel sucht, nach links (bei kleiner) bzw. nach rechts (bei größer) geht; ist der Wert gleich muss man nur noch schauen, ob der gefundene Wert der Sentinel ist, dann ist der gesuchte Wert nicht enthalten, andernfalls schon.

Kapitel III

Suchen und Sortieren

Begriffe und Definitionen

Beim linearen Suchen sucht man in einem unsortierten Feld mit n Elementen. Der Aufwand liegt zwischen 1 und n, ist also linear abhängig von der Anzahl der Elemente. $T(n) = \mathcal{O}(n)$

Beim binären Suchen muss das Feld schon sortiert sein. Man fragt dabei den Schlüsselwert des mittleren Elements ab und kann so den zu durchsuchenden Bereich in jedem Schritt halbieren. $T(n) = \mathcal{O}(\log_2 n)$

Anmerkung
Dieses Verfahren wurde im letzten Semester schon in der Aufgabe zum Zahlenraten verwendet.

Man kann Sortierverfahren nach ihrem Speicherplatzbedarf unterteilen: in-situ Sortierverfahren vs. externes Sortierverfahren

Ein Sortierverfahren ist stabil , wenn es die relative Ordnung von Elementen mit dem selben Schlüsselwert nicht ändert.

Ein Mikroschritt bzw. eine Elementaroperation besteht in der Regel aus einem Vergleich von 2 Schlüsselwerten und einer Kopier- oder Tauschoperation. Ein Makroschritt bzw. Durchlauf besteht aus $\mathcal{O}(n)$ Mikroschritten, zum Beispiel der Durchlauf durch alle noch zu sortierenden Elemente.

Die Zeitkomplexität von Algorithmen $T(n)$ wird mit den Landau-Operatoren angegeben:

- $\mathcal{O}(g(n)) = \{f(n) : \exists c > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0\}$ (Obergrenze)
- $\Omega(g(n)) = \{f(n) : \exists c > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0\}$ (Untergrenze)
- $\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0, n_0 \in \mathbb{N}_0 \mid 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0\}$ (Sandwich)

Also gilt: $T(n) = \mathcal{O}(n^2) = \mathcal{O}(n^2 \cdot \log n) = \mathcal{O}(n^2 \cdot \sqrt{n}) = \mathcal{O}(n^3) = \cdots = \mathcal{O}(2^n) = \mathcal{O}(n^n)$.

Anmerkung
Die Schreibweise kann ziemlich verwirren; es hilft sich $\mathcal{O}(n^2)$ als Menge vorzustellen, die alle Funktionen enthält, die maximal so schnell wie n^2 wachsen. Die Schreibweise $\mathcal{O}(n^2) = \mathcal{O}(n^2 \cdot \log n)$ bedeutet dann nicht, dass diese Mengen gleich sind, sondern dass die eine Menge in der anderen enthalten ist: Es gilt also $\mathcal{O}(n^2) \in \mathcal{O}(n^2 \cdot \log n)$, denn $x^2 \leq x^2 \cdot \log x$ für alle x.

Sortieralgorithmen bekommen in der Regel 3 Komplexitätsangaben:

- worst case : $\mathcal{O}(\dots)$ oder $\Theta(\dots)$
- average case : $\mathcal{O}(\dots)$, $\Omega(\dots)$ oder $\Theta(\dots)$
- best case : $\Omega(\dots)$

Im folgenden wird die **allgemeine Annahme** gelten: Sortiert wird immer in einem eindimensionalen Feld A mit Indexmenge I mit der Relation $\leq$ bezüglich des Schlüssels in aufsteigender Reihenfolge.

Es gibt mindestens 2 Möglichkeiten, Datenelemente im Feld A zu sortieren:

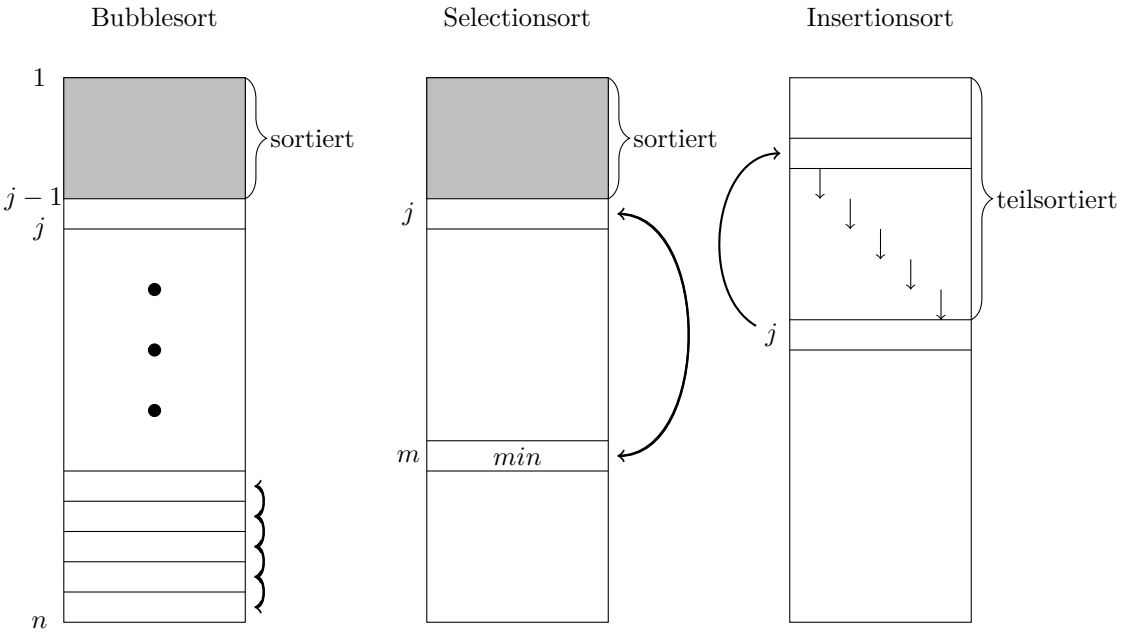
1. direktes Sortieren : Bewegen der Datenelemente inklusive key
2. indirektes Sortieren : Erzeugen einer Sortierpermutation σ der Indizes, wobei nur die Indizes in einem eigenen Feld und nicht die Datenelemente bewegt werden

Im sortierten Zustand gilt für alle $i, j \in I$:

- direktes Sortieren: $i < j \Rightarrow A(i) \leq A(j)$
- indirektes Sortieren: $i < j \Rightarrow A(\sigma(i)) \leq A(\sigma(j))$

Eine Sortierpermutation σ einer Liste A auf einer Indexmenge $I = \{1, \ldots, n\}$ ist eine Permutation von I, das heißt $\{\sigma(1), \ldots, \sigma(n)\}$ mit $\sigma(i) \neq \sigma(j)$ für $i \neq j$.

3 einfache Sortieralgorithmen



Sortierverfahren	**Anzahl Vergleiche**	**Anzahl Kopier-/Tauschoperationen**
Bubblesort	$\sum_{j=1}^{n-1}(n-j) = \sum_{j=1}^{n-1}\frac{n(n-1)}{2} = \mathcal{O}(n^2) = \Omega(n)$	$\leq \frac{1}{2}n(n-1) = \mathcal{O}(n^2)$ Tauschoperationen
Selectionsort	$\sum_{j=1}^{n-1}\frac{n(n-1)}{2} = \Theta(n^2)$	$\leq n-1 = \mathcal{O}(n)$ Tauschoperationen
Insertionsort mit binärer Suche im teilsortierten Teil	$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$ $\sum_{j=1}^{n-1}\log_2 j = \mathcal{O}(n\log_2 n)$	$\leq \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$ hier bleibt alles gleich

Allgemein kann man also sagen:

- best case: 0 Bewegungen/Kopier- und Tauschvorgänge, $\Omega(n)$ Vergleiche
- worst case: $\mathcal{O}(n^2)$ Vergleiche und Kopier-/Tauschoperationen

	Vergleiche	**Tauschoperationen**
Bubblesort, stabil	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selectionsort, nicht stabil	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Insertionsort, stabil	ohne binäre Suche: $\mathcal{O}(n^2)$, mit binärer Suche: $\mathcal{O}(n\log_2 n)$	$\mathcal{O}(n^2)$

$\Rightarrow T(n) = \mathcal{O}(n^2)$ für alle 3 einfachen Sortierverfahren

Satz 2.1
Sortierverfahren, die auf dem Schlüsselvergleich ($<$) von jeweils 2 Elementen (und einer eventuell notwendigen Tauschoperation) beruhen, benötigen im worst case mindestens $\Omega(n \log_2 n)$ Vergleiche.

Beweis. binärer Entscheidungsbaum der Höhe h zum Sortieren von n Elementen, da jeder Schlüsselwertvergleich eine binäre Entscheidung liefert. Es gibt $n!$ Permutationen der n verschiedenen Schlüsselwerte, also $n!$ verschiedene Sortierfolgen, das heißt $n!$ Entscheidungspfade. $\Rightarrow$ binärer Entscheidungsbaum benötigt mindestens $n!$ Blätter, um alle Anfangszustände in den einen Sortierten zu überführen. Ein Binärbaum der Höhe h hat $\leq 2^h$ Blätter. Also muss gelten:

$$
\begin{aligned}
& n! \leq 2^h \\
& h \geq \log_2(n!) \\
\text{Stirling:} \quad & n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right) \\
& n! > \left(\frac{n}{e}\right)^n \\
& h \geq \log_2 \left(\frac{n}{e}\right)^n = n(\log_2 n - \log_2 e) \\
& \quad = \Theta(n \log_2 n)
\end{aligned}
$$

$\Rightarrow$ mindestens $\Omega(n \log_2 n)$ Vergleiche im worst case $\square$

Quicksort

Der Quicksort ist ein rekursiver Algorithmus zum Sortieren einer Liste L im Indexbereich $a : e$. Erfunden wurde dieser von Sir Charles Antony Richard Hoare. Der Algorithmus läuft wie folgt ab:

1. Wähle beliebiges Element aus L, dieses habe den key W (sogenanntes Pivotelement). Ideal wäre, wenn W der Median aller keys wäre. Der schlechteste Fall wäre, wenn W ein Extremum wäre.
2. Bilde Partition $L_L \mid L_R$ der Liste L mit:
   - alle Elemente von L_L haben keys $\leq W$
   - alle Elemente von L_R haben keys $\geq W$
3. Sortieren der beiden Listen mittels rekursivem Aufruf von Quicksort

Als Quellcode sieht Quicksort dann so aus:

```
1  recursive subroutine QsortC(A)
2   real, intent(in out), dimension(:) :: A
3   integer :: iq
4
5   if(size(A) > 1) then
6    call Partition(A, iq)
7    call QsortC(A(:iq-1))
8    call QsortC(A(iq:))
9   endif
10 end subroutine QsortC
11
12 subroutine Partition(A, marker)
13  real, intent(inout), dimension(:) :: A
14  integer, intent(out) :: marker
15  integer :: i, j
16  real :: temp
17  real :: x      ! pivot point
18
19  x = A(1)
20  i= 0
21  j= size(A) + 1
22
23  do
24   j = j-1
25   do
26    if (A(j) <= x) exit
27    j = j-1
28   end do
29
30   i = i+1
```

```
31
32    do
33     if (A(i) >= x) exit
34     i = i+1
35    end do
36
37    if (i < j) then ! exchange A(i) and A(j)
38     temp = A(i)
39     A(i) = A(j)
40     A(j) = temp
41    elseif (i == j) then
42     marker = i+1
43     return
44    else
45     marker = i
46     return
47    endif
48   end do
49  end subroutine Partition
```

Im worst case, das heißt das Pivotelement ist ein Extremum, wird immer nur 1 Element abgespalten. Dann hat Quicksort die Komplexität $T(n) = \mathcal{O}(n^2)$. Im average case gilt: $T(n) = \mathcal{O}(n \log_2 n)$ und im best case hat Quicksort die Komplexität $T(n) = \Omega(n \log_2 n)$.

Zuletzt noch ein Blick auf die Eigenschaften:

- in situ
- nicht stabil
- hat nur sehr einfache Operationen
- für große n sehr schnell
- pro Durchlauf $\mathcal{O}(n)$
- für kleine n eher schlecht

Mergesort

2-Wege-Mergesort

Den Mergesort-Algorithmus gibt es in 2 Varianten: rekursiv und iterativ. Schauen wir uns zuerst die rekursive Variante an:

1. falls L leer oder nur 1 Element enthält $\rightarrow$ ok, return
2. divide: Teile L in 2 möglichst gleich lange Teillisten L_1 und L_2 und mache darauf rekursive Aufrufe `Mergesort(L_1)` und `Mergesort(L_2)`.
3. conquer: Merge von L_1 und L_2

```
1  ! kann nur 10 Elemente sortieren, kann man aber anpassen
2
3  subroutine _merge(lst, a, middle, b)
4   integer a
5   integer b
6   integer middle
7   integer lst(10)
8   integer tmp(10)
9   integer ai
10  integer bi
11  integer ti
12  integer x
13  ai = a
14  bi = middle
15  ti = a
16
17  do while ((ai < middle) .or. (bi < b))
18   if (ai == middle) then
19    tmp(ti+1) = lst(bi+1)
20    bi = bi + 1
21   else if (bi == b) then
22    tmp(ti+1) = lst(ai+1)
23    ai = ai + 1
24   else if (lst(ai+1) < lst(bi+1)) then
25    tmp(ti+1) = lst(ai+1)
26    ai = ai + 1
27   else
28    tmp(ti+1) = lst(bi+1)
29    bi = bi + 1
30   end if
31
32   ti = ti + 1
```

```
33   end do
34   do x = a, b - 1
35    lst(x + 1) = tmp(x + 1)
36   end do
37 end subroutine _merge
38
39 recursive subroutine mergesort(lst, a, b)
40   integer a
41   integer b
42   integer lst(10)
43   integer diff
44   diff = b - a
45
46   if (diff < 2) then
47    return
48   else
49    diff = diff / 2
50    call mergesort(lst, a, a + diff)
51    call mergesort(lst, a + diff, b)
52    call _merge(lst, a, a + diff, b)
53   endif
54 end subroutine mergesort
```

Mergesort ist nicht in situ. $T(n) = \mathcal{O}(n \log_2 n)$. All Lese- und Schreiboperationen sind streng sequenziell.

Die iterative Variante verläuft ähnlich, verwendet aber 4 Listen L_1, L_2, L_3 und L_4.

1. Init: Teile L in 2 möglichst gleich große Teillisten L_1 und L_2
2. Erzeuge 2 Listen sortierter Paare, L_3 und L_4, indem positionell sich entsprechende Elemente von L_1 und L_2 jeweils zu einem sortierten Paar gemacht werden und in die zuletzt nicht benutzte Liste L_3 bzw. L_4 (immer abwechselnd) geschrieben werden
3. Erzeuge 2 Listen sortierter Quadrupel in L_1 und L_2
4. Erzeuge 2 Listen sortierter Oktupel in L_3 und L_4
5. ...

Die Komplexität ist auch hier $T(n) = \mathcal{O}(n \log_2 n)$, tatsächlich ist die Zeit $T(n) = \Theta(n \log_2 n)$ immer die selbe, egal ob best- oder worst case.

k-Wege-Mergesort

Hier existiert nur eine rekursive Variante, in der die Liste in k Teillisten aufgeteilt wird. Man kann aber auch mit k Input-Listen und k Output-Listen arbeiten. Noch einige Bemerkungen:

- Für den Mergeschritt wird ein k-elementiger Vektor von Schlüsselwerten benötigt, um die jeweils aktuellen Kopfelemente der k zu verschmelzenden Listen sortiert zu speichern.

- Die Anzahl der Durchläufe reduziert sich gegenüber dem 2-Wege-Mergesort von $\lceil \log_2 n \rceil$ auf $\lceil \log_k n \rceil$, also um den Faktor $\frac{1}{\log_2 k} = \log_k 2$, zum Beispiel bei $k = 1024 = 2^{10}$ Teillisten auf $\frac{1}{10}$.

Markoschritt	**produziert**	k**-Tupel sortieren**	**Anzahl Elemente**	**Aufwand Insertion-Schritt**	**Anzahl Tupel**	**Aufwand**
1	k-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$0 \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k})$	$\mathcal{O}(n \log_2 k)$
2	k^2-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$k(k-1) \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k^2})$	$\mathcal{O}(nk)$
3	k^3-Tupel: $\mathcal{O}($	$(k \log_2 k +$	$(k^3 - k) \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k^3})$	$\mathcal{O}(nk)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
						$\Sigma = \mathcal{O}(kn \log_2 n)$

Heapsort

Anmerkung
Das folgende Kapitel ist ziemlich durcheinander; ich glaube Prof. Walter wusste selber nicht so richtig, was er wollte. Es empfiehlt sich ein Youtube-Tutorial anzuschauen, um zu verstehen, wie Heapsort funktioniert.

Ein binärer Heap ist ein vollständiger Binärbaum mit der sogenannten Heap-Eigenschaft : Ein vollständiger Binärbaum hat alle Schichten ab der Wurzel voll besetzt bis auf eventuell die letzte, die von links nach rechts bis zum letzten Knoten besetzt ist. Die Vollständigkeit garantiert, dass ein eindimensionales Feld `A(1:n)` mit den Elementen des Heaps in Levelorder abgespeichert keine Lücken aufweist. Außerdem gilt:

```
1  ! Index des linken Kindknotens des Knotens mit Index i
2  Left(i) = 2*i
3  ! Index des rechten Kindknotens des Knotens mit Index i
4  Right(i) = 2*i+1
5  ! i/2 ist Index des Elternknotens
6  Parent(i) = i/2
```

$\Rightarrow$ Dualität eines Heaps und eines vollständigen Binärbaums; außerdem Höhe $h = \Theta(\log_2 n)$.

Zusätzliche Heap-Eigenschaft: $A_{\texttt{Parent(i)}} \geq A_{\texttt{i}}$, das heißt Schlüsselwert des Elternelements $\geq$ Schlüsselwerte der beiden Kindknoten.

Um aus einem Feld A einen Heap zu machen, brauchen wir eine Subroutine `Heapify`. Die folgenden Quelltexte sind nur ein Konzept, soweit ich weiß, muss man sie nirgendwo reproduzieren.

```
1  n = size(A)
2
3  subroutine Heapify(A,i)
4   r = Right(i)
5   l = Left(i)
6   maxix = i
7
8   if(l <= size .and. A_l > A_i) then
9    maxix = l
10  end if
11  if(r <= size .and. A_r > A_maxix) then
12   maxix = r
13  end if
14  if(maxix /= i) then
15   tausche(A_i,A_maxix)
16   Heapify(A,maxix)
17  end if
18 end subroutine Heapify
```

Der Zeitaufwand für `Heapify(A,i)` ist proportional zur Höhe des Knotens mit Index i:
$T_{\texttt{Heapify}} = \mathcal{O}(h)$ mit $h =$`height(A,i)`$= \mathcal{O}(\log_2 n)$.

```
1 subroutine BuildHeap(A)
2  size = n
3
4  do i = n/2, 1, -1
5   Heapify(A,i)
6  end do
7 end subroutine BuildHeap
```

Versuchen wir uns an einer Komplexitätsanalyse: $T_{\texttt{BuildHeap}}(n) = \mathcal{O}\left(\frac{n}{2}\log_2 n\right) = \mathcal{O}(n \log_2 n)$. Eine bessere Analyse bekommen wir, wenn die Höhe der Knoten betrachten: In einem Heap haben höchstens $\left\lceil \frac{n}{2^{n+1}} \right\rceil$ Knoten die Höhe h.
$\Rightarrow$ Gesamtaufwand für `BuildHeap`:

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathcal{O}(h) = \mathcal{O}\left(n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} h \cdot \left(\frac{1}{2}\right)^h \right)$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$$

$$0 < x < 1: \quad \sum_{k=0}^{\infty} k \cdot x^k = \frac{1}{1-x} = (1-x)^{-1}$$

$$\text{Differenzieren} \quad \sum_{k=1}^{\infty} k \cdot x^{k-1} = \frac{1}{(1-x)^2}$$

$$x = \frac{1}{2}: \quad \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\right) = \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = 2$$

$$\Rightarrow T(n) = \mathcal{O}(n)$$

```
1 subroutine Heapsort(A)
2 BuildHeap(A) ! O(n)
3  do i=1, 2, -1 ! n-1 Iterationen
4   tausche(A_i,A_1) ! O(1)
5   size = size - 1 ! O(1)
6   Heapify(A,1) ! O(log n)
7  end do
8 end subroutine Heapsort
```

$\Rightarrow T_{\texttt{Heapsort}}(n) = \mathcal{O}(n) + (n-1)\mathcal{O}(\log_2 n) = \mathcal{O}(n \log_2 n)$

Priority Queue

Eine Priority Queue ist eine Art von Warteschlange, die nicht nach dem FIFO-Prinzip arbeitet, sondern die Elemente entsprechend ihrer Priorität (ein spezieller Key) behandelt. Implementiert wird dies durch einen Heap in einem eindimensionalen Feld.

Anmerkung
Die Grundoperationen einer Priority Queue und deren Komplexitäten werden gerne in Klausuren abgefragt.

Die Grundoperationen einer Priority Queue sind wie folgt:

- `init(A)` → initiiert eine leeres Feld `A`
- `empty(A)` → ist Feld leer?
- `HeapInsert(A,key)` → neues Element mit `key` einfügen in `A`
- `HeapExtractMax(A,key)` → key des Wurzelknotens wird in 2. Parameter (oder als Funktionswert) zurückgegeben und dieses Element aus `A` herausgenommen
- `HeapUpdate(A,key,max)` → Kombination aus `HeapExtractMax` und `HeapInsert`: liefert in `max` den Key der Wurzel und ersetzt diesen Wert durch `key`
- `Maximum(A)` → Funktion, liefert den key der Wurzel

```
1 subroutine HeapExtractMax(A, max)
2  if (size <= 0) error("empty heap")
3
4  max = A_1
5  A_1 = A_size
6  size = size - 1
7  Heapify(A,1)
8 end subroutine HeapExtractMax
```

$\Rightarrow T(n) = \mathcal{O}(\log_2 n)$

```
1 subroutine HeapUpdate(A, key, max)
2  if (size <= 0) error("empty heap")
3
4  max = A_1
5  A_1 = key
6  Heapify(A,1)
7 end subroutine HeapUpdate
```

$\Rightarrow T(n) = \mathcal{O}(\log_2 n)$

```
1 subroutine HeapInsert(A, key)
2  size = size + 1
3  i = size
```

```
4
5   do while (i > 1 .and. A_Parent(i) < key)
6    A_i = A_Parent(i)
7    i = Parent(i)
8   end do
9   A_i = key
10 end subroutine HeapInsert
```

$\Rightarrow T(n) = \mathcal{O}(\log_2 n)$

```
1  subroutine BuildHeapNew(A)
2   size = 1
3
4   do i = 2, n
5    HeapInsert(A,A_i)
6   end do
7  end subroutine BuildHeapNew
```

$\Rightarrow T(n) = \mathcal{O}(\log_2 n)$

Counting Sort

Der Counting Sort-Algorithmus ist dann besonders gut, wenn alle möglichen Schlüsselwerte aus der Menge $S = \{0, 1, 2, ..., k-1\}$ kommen. Wenn $k = \mathcal{O}(n)$, dann können wir mit Counting Sort in der Zeit $T(n) = \mathcal{O}(n)$ sortieren. Benötigt werden 3 eindimensionale Felder:

- `A(1:n)` Originaldaten/-schlüsselwerte
- `B(1:n)` für sortierte Werte
- `C(0:k-1)` Feld von Zählern

Der Algorithmus zählt, wie oft jeder Wert in der Eingabe vorkommt. Diese Anzahlen speichert er in einem zusätzlichen Array mit k Feldern ab. Mit Hilfe dieses Arrays wird anschließend für jeden Schlüsselwert die Zielposition in der Ausgabe berechnet.

Anmerkung
Der Counting Sort ist also dann besonders gut, wenn man alle Einwohner Deutschlands ($\sim$ 81 Millionen) nach ihrem Alter ($S = \{0, ..., 150\}$) sortieren möchte.
Der Counting Sort eignet sich nicht dafür die Studenten der PROG-Vorlesung ($\sim$ 30 ☺) nach ihrem Nachnamen (40 Zeichen $\Rightarrow S = \{0, ..., 26^{40}\}$) zu sortieren.

```
1  subroutine counting_sort_a(array)
2   integer, dimension(:), intent(inout) :: array
3
4   call counting_sort_mm(array, minval(array), maxval(array))
5  end subroutine counting_sort_a
6
7  subroutine counting_sort_mm(array, tmin, tmax)
8   integer, dimension(:), intent(inout) :: array
9   integer, intent(in) :: tmin, tmax
10  integer, dimension(tmin:tmax) :: cnt
11  integer :: i, z
12
13  cnt = 0 ! Initialize to zero to prevent false counts
14
15  forall (i=1:z)
16   ! Not sure that this gives any benefit over a DO loop.
17   cnt(array(i)) = cnt(array(i))+1
18  end forall
19
20  ! ok - cnt contains the frequency of every value
21  ! let's unwind them into the original array
22
23  z = 1
24  do i = tmin, tmax
```

```
25    do while ( cnt(i) > 0 )
26     array(z) = i
27     z = z + 1
28     cnt(i) = cnt(i) - 1
29    end do
30   end do
31  end subroutine counting_sort_mm
```

Radix-/Distribution Sort

Wenn wir annehmen, dass der Schlüssel z sich als Zahl mit d Ziffern zur Basis k darstellen lässt, also $z = [z_{d-1}z_{d-2}...z_1z_0]_k = \sum_{i=0}^{d-1} z_i k^i$, dann lässt sich mit Radix-/Distribution Sort in $T(n) = \Theta(d(n+k))$ sortieren. Falls $k = \mathcal{O}(n)$, dann $T(n) = \Theta(dn)$ und falls d relativ klein ist, dann $T(n) = \Theta(n)$.

```
1 subroutine RadixSort (A,B,k)
2  do i = 0, d-1
3   CountingSort(A,B,i,k)
4  end do
5 end subroutine RadixSort
```

Kapitel IV

Rekursion, Iteration, Komplexität

Beispiele

Neben der Zeitkomplexität $T(n)$ werden wir uns nun auch mit der Speicherkomplexität $S(n)$ beschäftigen.

> **Anmerkung**
> Es nicht unbedingt entscheidend, alle Komplexitäten auswendig zu lernen. Vielmehr sollte man wissen, wie die Algorithmen dahinter arbeiten und sich so die Komplexitäten herleiten.

Fakultät

```
1  recursive function fac (n) result (res)
2   integer :: n
3   integer :: res
4
5   if (n == 1) then
6    res = 1
7   else
8    res = n*fac(n-1)
9   end if
10 end function fac
```

$\Rightarrow T(n) = \Theta(n)$, $S(n) = \Theta(n)$

```
1  function fac_iterativ (n)
2   integer :: n, fac_iterativ, i
3
4   fac_iterativ = 1
5
6   do i = 1, n
7    fac_iterativ = fac_iterativ * i
8   end do
9  end function fac_iterativ
```

$\Rightarrow T(n) = \Theta(n)$, $S(n) = \Theta(1)$

Reverse String

```
1  recursive function reverse (string) result (res)
2   character (*), intent (in) :: string
```

```
3   character (len (string)) :: res
4
5   if (len (string) == 0) then
6    res = ""
7   else
8    res = string(len(string):)//reverse(string(:len(string)-1))
9   end if
10 end function reverse
```

$\Rightarrow T(n) = \Theta(n)$, $S(n) = \Theta(n)$

```
1  program reverse_iterativ
2   character(80) :: str = "This is a string"
3   character :: temp
4   integer :: i, length
5
6   write(*,*) str
7   length = len_trim(str)
8   ! Ignores trailing blanks.
9   ! Use len(str) to reverse those as well
10
11  do i = 1, length/2
12   temp = str(i:i)
13   str(i:i) = str(length+1-i:length+1-i)
14   str(length+1-i:length+1-i) = temp
15  end do
16
17  write(*,*) str
18 end program reverse_iterativ
```

$\Rightarrow T(n) = \Theta(n)$, $S(n) = \Theta(n)$

Primzahl

Um zu überprüfen, ob eine Zahl eine Primzahl ist, muss man immer bis zur Wurzel dieser Zahl auf Teiler prüfen. Egal ob man das rekursiv oder iterativ macht, $T(n) = \mathcal{O}(\sqrt{n})$. Die Speicherkomplexität ist beim rekursiven schwer vorher zu sagen, beim iterativen Algorithmus ist die $S(n) = \Theta(1)$.

Fibinacci-Zahlen

```
1  recursive function fibR(n) result(fib)
2   integer, intent(in) :: n
3   integer :: fib
4
5   select case (n)
6    case (:0); fib = 0
```

```
7    case (1); fib = 1
8    case default; fib = fibR(n-1) + fibR(n-2)
9   end select
10 end function fibR
```

$\Rightarrow T(n) = \Theta(\Phi^n)$, $S(n) = \mathcal{O}(2^n)$

```
1  function fibI(n)
2   integer, intent(in) :: n
3   integer, parameter :: fib0 = 0, fib1 = 1
4   integer :: fibI, back1, back2, i
5
6   select case (n)
7    case (:0); fibI = fib0
8    case (1); fibI = fib1
9    case default
10    fibI = fib1
11    back1 = fib0
12    do i = 2, n
13     back2 = back1
14     back1 = fibI
15     fibI   = back1 + back2
16    end do
17  end select
18 end function fibI
```

$\Rightarrow T(n) = \Theta(n)$, $S(n) = \Theta(1)$

Man kann die n-te Fibonacci-Zahl F_n auch direkt berechnen:

$$F_n = \frac{\Phi^n - (1 - \Phi^n)}{\sqrt{5}}$$

Hier wird eine ganzzahlige Potenzierung benötigt, die eine Komplexität von $T(n) = \Theta(\log_2 n)$ hat.

Ganzzahliges Potenzieren

Wenn man naiv vorgeht, ist Potenzieren nichts anderes als Multiplikation n mal mit sich selbst. Dann sind die Komplexitäten: $T(n) = \Theta(n)$, $S_{iter}(n) = \Theta(1)$ und $S_{rek}(n) = \Theta(n)$.

Man kann den Prozess aber noch deutlich verbessern. Soll man zum Beispiel $5^{10} = 5^8 \cdot 5^2$ berechnen, so berechnet man $5^2 = 25$, $5^4 = 5^2 \cdot 5^2 = 625$, $5^8 = 5^4 \cdot 5^4 = 390.625$ und abschließend $5^{10} = 5^2 \cdot 5^8 = 9.765.625$.

- rekursiv: $T(n) = \Theta(\log_2 n)$, $S(n) = \Theta(\log_2 n)$
- iterativ: $T(n) = \Theta(\log_2 n)$, $S(n) = \Theta(1)$

Man kann das Potenzieren auch direkter machen: $x^n = e^{\ln x^n} = e^{n \cdot \ln x}$. Allerdings braucht man hier die Funktionen e^x und ln, die insgesamt langsamer als die iterative Methode sind.

Größter gemeinsamer Teiler

Es gilt:

$$\begin{aligned}\text{ggT}(a,b) &= \text{ggT}(a-b,b) = \text{ggT}(a-2b,b) = ... \\ &= \text{ggT}(a \bmod b, b) = \text{ggT}(b, a \bmod b)\end{aligned}$$

Satz 1.1 (von Lamé)
$\forall k \geq 1, k \in \mathbb{N}$, wenn $a > b \geq 0$ und $b < F_{k+1}$ gilt, dann macht $\text{ggT}(a,b)$ höchstens $k-1$ rekursive Aufrufe. Zwei aufeinanderfolgende FIBONACCI-Zahlen sind der worst case für den euklidischen Algorithmus:

$$\text{ggT}(F_{k+1}, F_k) = \text{ggT}(F_k, F_{k+1} \bmod F_k) = \text{ggT}(F_k, F_{k-1})$$

Da $F_k = \frac{\Phi^n}{\sqrt{5}}$ mit $\Phi = \frac{1+\sqrt{5}}{2}$ folgt

- rekursiv: $T(n) = \mathcal{O}(\log_\Phi \min\{a,b\})$
- iterativ: $T(n) = \mathcal{O}(\log_\Phi \min\{a,b\})$

Binominialkoeffizient

Der Binominialkoeffizient lässt sich rekursiv wie folgt berechnen:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$\Rightarrow T(n) = \mathcal{O}(2^n)$, $S(n) = \mathcal{O}(n)$

Der Algorithmus klappt auch iterativ, deshalb $T(n) = \mathcal{O}(n^2)$, $S(n) = \Theta(n)$

Man kann den Binominialkoeffizienten auch ganz normal ausrechnen:

$$\binom{n}{k} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdot \dots \cdot \frac{n-k+1}{k}$$

$\Rightarrow T(n) = \Theta(k) = \mathcal{O}(n)$, $S(n) = \Theta(1)$

Collatz-Funktion

Lothar Collatz hat 1937 eine interessante Rechenvorschrift für Zahlen entwickelt, deren Problem bis heute noch nicht gelöst wurde.

```
1 collatzFun(p,n)
2  do while (n /= 1)
3   if(mod(n,2) == 1) then
4    n=p*n+1
5   else
6    n = n/2
```

```
7     end if
8   end do
9 end collatzFun
```

Für $p = 5$ ergibt sich:

n	collatzFun(5,n)
1	1
2	$2 \to 1$
3	$16 \to 8 \to 4 \to 2 \to 1$
4	$4 \to 2 \to 1$
5	$26 \to 13 \to 66 \to 33 \to 166 \to 83 \to 416 \to 208 \to 104 \to 52 \to 26 \to \ldots$

$\Rightarrow$ nicht immer berechenbar, das heißt, die Funktion kommt nie zum Ende

Für $p = 3$ (originales Collatz-Problem) ergibt sich:

n	collatzFun(3,n)
1	1
2	$2 \to 1$
3	$10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$
4	$4 \to 2 \to 1$
5	$16 \to 8 \to 4 \to 2 \to 1$

Es stellt sich die Frage, ob dieses Problem immer berechenbar ist. Man hat gezeigt, dass es für $p = 3$ bis zu $n < 2^{61}$ berechenbar ist.

Multiplikation zweier n-stelliger Zahlen x und y

Bei einer iterativen Multiplikation, so wie man sie schriftlich gelernt hat, beträgt die Komplexität $T(n) = \Theta(n^2)$ oder eben auch allgemein $T(x \cdot y) = \Theta(\log_b x \cdot \log_b y) = \Theta(m \cdot n)$ mit $m = \text{digits}(x)$ und $n = \text{digits}(y)$.

Falls man bis jetzt die Vermutung hatte, dass rekursive Algorithmen immer schlechter als iterative Algorithmen sind, hier ist ein Gegenbeispiel: Wenn man die Zifferngruppen rekursiv halbiert ergibt sich eine Komplexität von $T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1{,}525})$.

Der Schönhage-Strassen-Algorithmus, eine diskrete Fourier-Transformation, war von 1971 bis 2007 (in der Vorlesung war es auch nicht aktueller) der schnellste Multiplikationsalgorithmus mit einer Komplexität von $T(n) = \Theta(n \cdot \log n \cdot \log \log n)$.

Matrizenmultiplikation

Die iterative Methode hat eine Komplexität von $T(n) = \Theta(n^3)$. Das ist die Methode, die auch in der Schule/im Studium gelehrt wird.

Auch hier ist die rekursive Variante wieder etwas schneller, STRASSEN fand einen Algorithmus mit einer Komplexität von $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,8075})$.

Der beste bis heute bekannte Algorithmus setzt auf eine Vermischung von iterativen und rekursiven Funktionen und hat eine Komplexität von $T(n) = \mathcal{O}(n^{2,376})$.

Zyklische Zahl

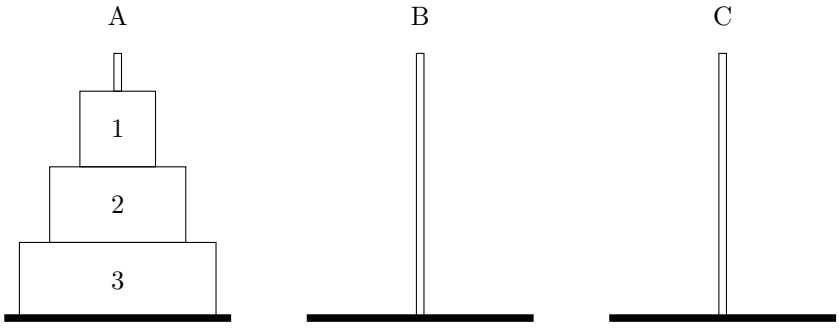
Finde Ziffern $z_1, ..., z_n$ mit:

$$5 \cdot [z_1 z_2 ... z_n]_{10} = [5 z_1 z_2 ... z_n]_{10}$$

Ein einfaches Durchprobieren aller Zahlen bis zu einer Zahl n erzeugt einen Aufwand von $T(n) = \mathcal{O}(n \log n)$. Mit systematischer Berechnung der Ziffern von hinten nach vorne beträgt der Aufwand nur noch $T(n) = \mathcal{O}(\log_2 n)$, wenn n die gesuchte Zahl ist.

Die gesuchte Zahl ist übrigens $n = 10.204.081.632.653.061.224.489.795.918.367.346.938.775$.
Als Wort: "10 Sextilliarden 204 Sextillionen 81 Quintilliarden 632 Quintillionen 653 Quadrilliarden 61 Quadrillionen 224 Trilliarden 489 Trillionen 795 Billiarden 918 Billionen 367 Milliarden 346 Millionen 938 Tausend 775" ☺.

Türme von Hanoi



Dieses Problem lässt sich nur rekursiv rechnen. Der Aufwand dafür beträgt $T(n) = 2^n - 1$.

	Schritt i	**Scheibe** d	**Quelle** s	**Ziel** t
$n = 1$	1	1	A	C
$n = 2$	1	1	A	B
	2	2	A	C
	3	1	B	C
$n = 3$	1	1	A	C

	2	2	A	B
	3	1	C	B
	4	3	A	C
	5	1	B	A
	6	2	B	C
	7	1	A	C

Im i-ten Schritt finden dann folgende Schritt statt:

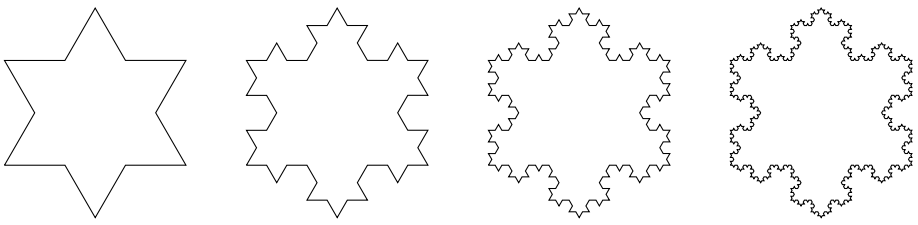
- Scheibe k wird bewegt, mit `mod(i,2**(k-1))` $= 0$, aber `mod(i,2**k) != 0`
- Scheibe k wird zum ersten Mal im Schritt 2^{k-1} und dann alle 2^k Schritte bewegt
- Die größte Scheibe n wird immer genau einmal im Schritt 2^{n-1} von A nach C bewegt, das heißt rückwärts durch die Positionsindizes, die nächstkleinere Scheibe $n-1$ zweimal vorwärts A $\to$ B $\to$ C, die nächstkleinere $n-2$ viermal rückwärts A $\to$ C $\to$ B $\to$ A $\to$ C, ...

Anmerkung
Der Legende nach soll die Welt zu dem Zeitpunkt untergehen, wenn dieses Problem mit $n = 64$ Scheiben von einem Menschen gelöst wurde.

Dafür benutzt man stattdessen einen Computer und das Programm von Unicomp, Inc (https://www.fortran.com/F/ex_hanoi.html, auch in meinen Github-Repository: https://github.com/henrydatei/TU_PROG/blob/master/Tuerme_von_Hanoi.f95).

Dieses Programm wird bei $n = 64$ einen Output von einem Zetabyte produzieren, was in etwa dem Internettraffic der Menschheit eines Jahres entspricht. Weiterhin würde mein Rechner rund 3,3 Millionen Jahre brauchen. Es dauert also noch ein bisschen bis zum Weltuntergang ☺.

Kochkurve



Die Komplexitäten lauten

- rekursiv: $T(n) = \mathcal{O}(4^n)$
- iterativ: Algorithmus ist sehr kompliziert

n-Damen-Problem

Beim n-Damen-Problem geht es darum, n Damen auf einen $n \times n$-Schachbrett zu verteilen, sodass keine Dame eine andere schlagen kann. Eine Dame kann alle Damen schlagen, die horizontal, vertikal oder diagonal zu ihr stehen.

Für $n = 1$ gibt es genau 1 triviale Lösung, $n = 2$ und $n = 3$ haben keine Lösungen, für $n = 4$ gibt es bis auf Symmetrien auch nur eine Lösung:

	D		
			D
D			
		D	

Will man alle Lösungen finden kann man die n Damen beliebig auf die n^2 Felder verteilen. Dann muss man $\binom{n^2}{n}$ Möglichkeiten durchprobieren. Verteilt man nur auf jeder Spalte eine Dame, so hat man noch n^n Möglichkeiten. Bei zusätzlich nur noch einer Dame pro Zeile sind es nur noch $n!$ Möglichkeiten. Wenn auch Diagonalen berücksichtigt werden sollen, gibt es viel weniger Möglichkeiten. Da eine systematische Planung vorausschauend alle bedrohten Felder für weitere Platzierungen kennen müsste, wird zur Lösung Backtracking verwendet.

Legepuzzle

Eine Legepuzzle ist ein zweidimensionales Puzzle mit n Teilen.

Löst man dieses Puzzle rekursiv naiv, also vergleicht eine Seite eines Teils mit jeweils allen offenen Stellen, so hat man $T(n) = \mathcal{O}(n^2)$. Löst man dies iterativ mit k Teilengen, zum Beispiel Farbgruppen mit $\approx \frac{n}{k}$ Teilen, wobei $k \leq \frac{n}{k}$, also zum Beispiel $k = \sqrt{n} = \frac{n}{k}$: $T_k(n) = \mathcal{O}(n + \frac{n^2}{k} + k^2) = \mathcal{O}(n^{1,5})$.

Ackermann-Funktion

Die Ackermann-Funktion hat folgende Bildungsvorschrift:

$$A(i,j) = \begin{cases} 2^j & i = 1 \\ A(i-1,2) & j = 1 \\ A(i-1, A(i,j-1)) & i \geq 2, j \geq 2 \end{cases}$$

$i \setminus j$	1	2	3	4	5
1	2	4	8	16	32
2	4	$2^{2^2} = 16$	$2^{2^{2^2}} = 65536$	2^{65536}	
3	$2^{2^2} = 16$	$A(2,16)$			
4	$A(2,16)$	$A(3,A(4,1))$			

Die ACKERMANN-Funktion ist nicht primitiv rekursiv!

Besonders bei $A(2,3)$ tauchen unschön zu schreibende "Potenztürme" auf. Um diese zu vermeiden, führen wir die Knuth-Up-Arrow -Notation ein. Dabei gilt:

$$
\begin{aligned}
a \uparrow b &:= a^b \\
a \uparrow\uparrow b = a \uparrow^2 b &:= \underbrace{a \uparrow \ldots \uparrow a}_{b} \\
a \uparrow\uparrow\uparrow b = a \uparrow^3 b &:= \underbrace{a \uparrow^2 a \uparrow^2 \ldots \uparrow^2 a}_{b} \\
\underbrace{a \uparrow \ldots \uparrow}_{n} b = a \uparrow^n b &:= \underbrace{a \underbrace{\uparrow \ldots \uparrow}_{n-1} a \underbrace{\uparrow \ldots \uparrow}_{n-1} \ldots \underbrace{\uparrow \ldots \uparrow}_{} a}_{b}
\end{aligned}
$$

So gilt also:

- $3 \uparrow 2 = 3^2 = 9$
- $3 \uparrow\uparrow 2 = 3^3 = 27$
- $3 \uparrow\uparrow 3 = 3^{3^3} > 7 \cdot 10^{12}$

Kapitel V

Grundrechenarten in Rechnern

Anmerkung
Der Inhalt dieses ganzen Kapitels ist eigentlich nicht prüfungsrelevant.

Addition n-stelliger ganzer Zahlen

Wir werden im Folgenden folgende Notationen verwenden:

- OR: $x + y$
- AND: $x \cdot y$
- NOT: $\overline{x}$
- XOR: $x \oplus y$

Beim Halbaddierer ist die Summe $s = a \oplus b$ und der Übertrag (carry) $c = a \cdot b$.

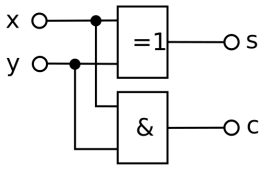


Abbildung V.1: Halbaddierer

Der Volladdierer besteht aus 2 Halbaddierern. Die Summe s ist hier: $s = a\overline{b}\overline{c_{in}} + \overline{a}b\overline{c_{in}} + \overline{a}\overline{b}c_{in}abc_{in} + a \oplus b \oplus c_{in}$. Der Übertrag ist $c_{out} = ab + ac_{in} + bc_{in}$.

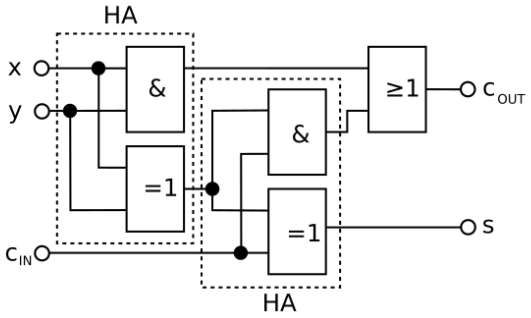


Abbildung V.2: Volladdierer

Mit dem Carry-Ripple-Adder kann man dann endlich zwei n-stellige Zahlen addieren:

- $a = [a_{n-1}a_{n-2}...a_1a_0]_2$
- $b = [b_{n-1}b_{n-2}...b_1b_0]_2$

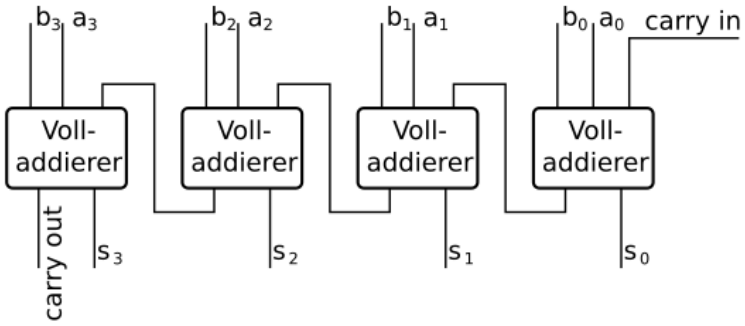


Abbildung V.3: Carry-Ripple-Adder

$\Rightarrow T(n) = \mathcal{O}(n \cdot T_{FA})$

Machen wir zum Schluss noch eine Übertragsanalyse: Eine Bitposition $i \in \{0, 1, ..., n-1\}$ kann 3 verschiedene Übertragsfälle annehmen:

1. kein Übertrag möglich, wenn $a_i = b_i = 0$
2. Übertrag wird weitergeleitet (carry propergate) $p = a_i \oplus b_i$
3. Übertrag wird auf jeden Fall generiert (generate bit) $g = a_i \cdot b_i$

Rekursionsbeziehung:

$$c_{i+1} = g_i + p_i \cdot c_i = a_i b_i + (a_i \oplus b_i)c_i = a_i b_i + (a_i + b_i)c_i$$
$$c_{i+1} = \underbrace{g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + ... + p_i p_{i-1}...p_1 g_0}_{G_{0,i}} + \underbrace{p_i p_{i-1}...p_1 p_0}_{P_{0,i}} \cdot c_0$$

Subtraktion n-stelliger ganzer Zahlen

Wir wollen möglichst die gleiche Hardware für die Subtraktion wie für die Addition verwenden. Dabei können wir uns zu Nutze machen, dass gilt:

$$a - b = a + \underbrace{(-b)}_{\text{2er Komplement}} = a + \underbrace{(\bar{b})}_{\text{1er Komplement}} + 1$$

Damit wird auch in der hinteren Postion ein Volladdierer benötigt und Addition und Subtraktion laufen über die gleiche Hardware.

Zugleich optimieren wir noch die Addierer. Vom Carry-Ripple-Adder zum Carry-Lookahead-Adder und dann zum Carry-Skip-Adder . Während der Carry-Ripple-Adder eine Komplexität von $T(n) = \mathcal{O}(n)$ hat, hat der Carry-Lookahead-Adder eine Komplexität von $T(n) = \mathcal{O}(\log_2 n)$

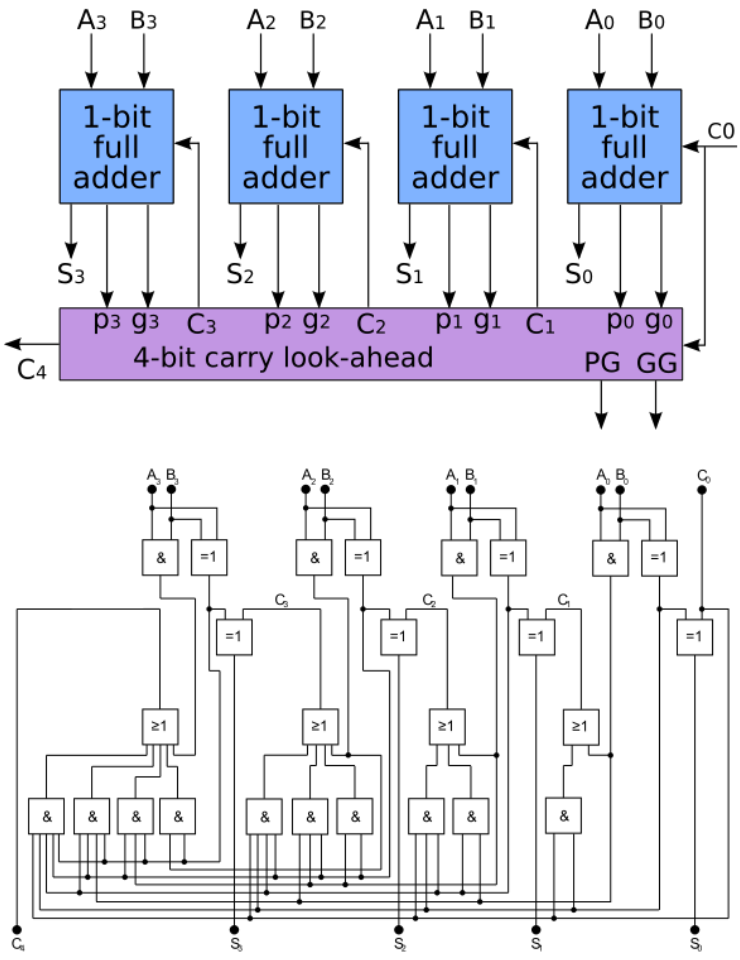


Abbildung V.4: Carry-Lookahead-Adder

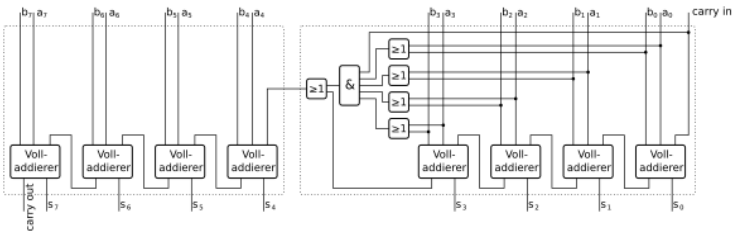


Abbildung V.5: Carry-Skip-Adder

Multiplikation n-stelliger ganzer Zahlen

Zuerst müssen wir uns mit dem sogenannten Shifting beschäftigen. Shifting ist ein Vorgang, der das Bitmuster einer Zahl verschiebt. Man unterscheidet 4 Arten des Shifting:

- logischer Links-Shift $\texttt{LSHIFT}_1(a)$: verschiebt das Bitmuster von a um 1 Zeichen nach links, a_k fliegt raus, es werden Nullen eingefügt.
- logischer Rechts-Shift $\texttt{RSHIFT}_1(a)$: verschiebt das Bitmuster von a um 1 Zeichen nach rechts, a_0 fliegt raus, es werden Nullen eingefügt.
- arithmetischer Links-Shift $\texttt{ALSHIFT}_1(a)$: verschiebt das Bitmuster von a um 1 Zeichen nach links, a_k fliegt raus, es wird der Wert des rechtesten Bits eingefügt.
- arithmetischer Rechts-Shift $\texttt{ARSHIFT}_1(a)$: verschiebt das Bitmuster von a um 1 Zeichen nach rechts, a_0 fliegt raus, es wird der Wert des linkesten Bits eingefügt.

Man kann die Multiplikation $a \cdot b$ auch als wiederholte Summation auffassen: $\sum_{i=0}^{n-1} a_i \cdot 2^i \cdot b$. In heutigen Rechnern sind allerdings Addition und Multiplikation gleich schnell. Man kann diese wiederholte Addition aber auch deutlich verbessern:

- Gruppe von k Nullen in a: sofortiger $\texttt{ARShift}_k(a)$
- Gruppe von k Einsen in a: $a = [\ldots 0 \underbrace{\overbrace{1}^{i} \ldots \overbrace{1}^{j}}_{k} 0 \ldots]$. Dann

$$\sum_{i=j}^{l} 2^i = 2^{l+1} - 2^j = 2^{j+k} - 2^j$$

Der Booth-Algorithmus ist eine andere Art der Optimierung. Die Idee ist, dass $a \cdot b$ mit $b = c - d$ $a \cdot b = a \cdot c - a \cdot d$ ergibt. Dazu muss der erste Faktor kodiert werden: Dazu sei $Y = [y_{n-1} \ldots y_0]_2$ der zu kodierende Operand. An diesen fügt man eine weitere Stelle $y_{-1} = 0$ ein. Der kodierte Operand $Y' = [y'_{n-1} \ldots y'_0 y_{-1}]_2$ und wir mit $y'_i = y_{i-1} - y_i$ berechnet.

Um den Booth-Algorithmus zu zeigen, wollen wir $[44]_{10} = [00101100]_2$ mit $[17]_{10} = [00010001]_2$ multiplizieren:

									0	0	0	1	0	0	0	1	2. Faktor
·									0	1	-1	1	0	-1	0	0	Kodierung 1. Faktor
+		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	keine Addition
+		0	0	0	0	0	0	0	0	0	0	0	0	0	0		keine Addition
+		1	1	1	1	1	1	1	1	1	1	1	1	1			2er Komplement (2. Faktor)
+		0	0	0	0	0	0	0	0	0	0	0	0				keine Addition
+		0	0	0	0	0	0	1	0	0	0	1					2. Faktor
+		1	1	1	1	1	1	1	1	1	1						2er Komplement (2. Faktor)
+		0	0	0	0	1	0	0	0	1							2. Faktor
+		0	0	0	0	0	0	0	0								keine Addition
1	0	0	0	0	0	0	1	0	1	1	1	0	1	1	0	0	Ergebnis ohne Überlauf
=							1	0	1	1	1	0	1	1	0	0	Ergebnis mit Überlauf

Dazu noch ein paar Bemerkungen:

- Statt also mit $[0100000]_2$, $[0001000]_2$ und $[0000100]_2$ zu multiplizieren und die Ergebnisse zu addieren, wird nun also mit $[1000000]_2$, $[0100000]_2$, $[0010000]_2$ und $[0000100]_2$ multipliziert und die Ergebnisse addiert bzw. subtrahiert.
- Wie man am Beispiel sieht, kann sich die Anzahl der Additionen auch erhöhen (im Beispiel von 3 auf 4), was aber eigentlich nicht gewünscht ist. Im statistischen Durchschnitt werden im Booth-Verfahren genauso viele Additionen gebraucht wie ohne Booth-Verfahren. Der Vorteil liegt aber darin, dass in der Informatik keine Gleichverteilung von Zahlen vorliegt. Vielmehr gibt es häufig Zahlen mit vielen Nullen und durch das Zweierkomplement bei negativen Zahlen häufig viele Einsen am Anfang. Nur durch diese Tatsache hat das Booth-Verfahren Vorteile gegenüber einer normalen Multiplikation.
- Das Verfahren produziert das richtige Ergebnis: $44 \cdot 17 = [748]_{10} = [1011101100]_2$.

Die 3. Möglichkeit die Multiplikation zu verbessern, ist der Wallace-Tree . Die Idee dahinter ist folgende:

$$\underbrace{\left(\sum_{k=0}^{n} a_k 2^k\right)}_{\text{Binärdarstellung } a} \cdot \underbrace{\left(\sum_{k=0}^{n} b_k 2^k\right)}_{\text{Binärdarstellung } b} = \sum_{k=0}^{2n} \sum_{i+j=k} a_i b_j 2^k$$

Der Wallace-Tree-Multiplizierer geht in 3 Schritten vor:

1. Berechne für jedes Paar (i, j) mit $1 \leq i \leq n$ und $1 \leq j \leq k$ das Partialprodukt $a_i b_j 2^{i+j}$.
2. Addiere die Resultate dieser Berechnung innerhalb der für den Wallace-Tree-Multiplizierer spezifischen Baumstruktur stufenweise mithilfe von Voll- und Halbaddierern, bis nur noch 2 Zahlen übrig sind, die addiert werden müssen.

3. Addiere diese beiden Zahlen mit einem normalen Addierwerk (Carry-Lookahead-Adder).

Der WALLACE-Tree benutzt den Carry-Save-Adder (CSA), der 3 Zahlen addieren kann und 2 Zahlen ausgibt: Die Sequenz der Partialsummen und die Sequenz der Übertragsbits → 2 gleich lange Zahlen

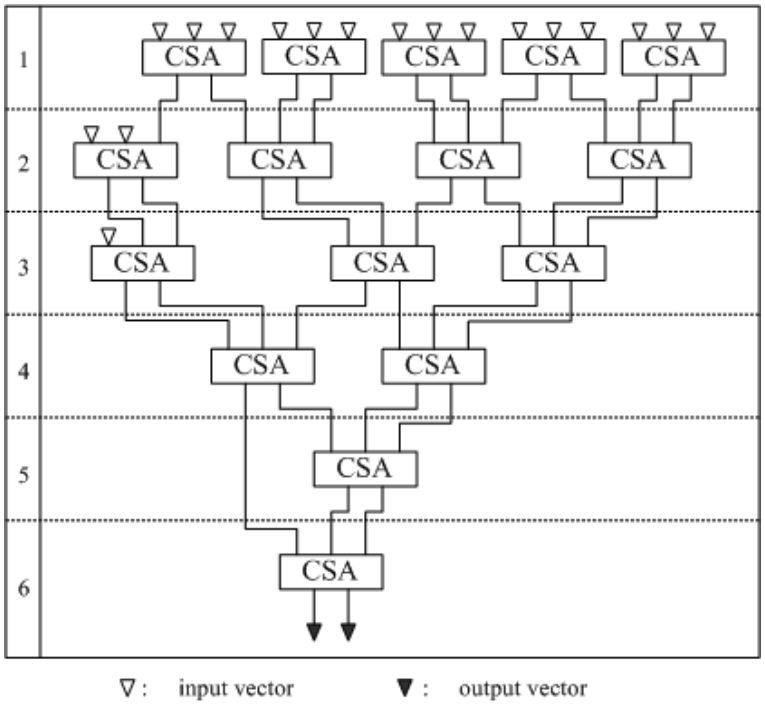


Abbildung V.6: WALLACE-Tree

Die Komplexität des WALLACE-Tree ist $T(n) = \mathcal{O}(\log n)$, also in etwa genau so lange wie die Addition!

Anhang