

Programmieren für Mathematiker SS2018

Dozent: Prof. Dr. Wolfgang Walter

4. September 2018

Inhaltsverzeichnis

I	Pointer	2
1	Allgemeines über Pointer	2
2	Listen	4
2.1	Grundoperationen auf einer Liste	4
2.2	Grundoperationen auf einer Deque	6
3	Queues	8
3.1	Grundoperationen auf einer Queue	8
4	Rechenaufwand für Grundoperationen	10
II	Bäume/Trees und Rekursion	11
1	Binäre Suchbäume	12
III	Suchen und Sortieren	13
1	Begriffe und Definitionen	13
2	3 einfache Sortieralgorithmen	15
3	Quicksort	17
4	Mergesort	19
4.1	2-Wege-Mergesort	19
4.2	k -Wege-Mergesort	20
5	Heapsort	22
6	Priority Queue	24
7	Counting Sort	26
8	Radix-/Distribution Sort	28
IV	Rekursion, Iteration, Komplexität	29
1	Beispiele	29
1.1	Fakultät	29
1.2	Reverse String	29
1.3	Primzahl	30
1.4	FIBINACCI-Zahlen	30
1.5	Ganzzahliges Potenzieren	31
1.6	Größter gemeinsamer Teiler	32
V	Implementierung der Grundrechenarten in Rechnern	33
	Anhang	35
	Index	35

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 Github¹, d.h. du findest den gesamten L^AT_EX-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 L^AT_EX zu lernen, denn Dokumente, die viele mathematische oder physikalische Formeln enthalten, lassen sich sehr gut mittels L^AT_EX darstellen, in Word oder anderen Office-Programmen sieht so etwas dann eher dürftig aus.

L^AT_EX 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, das 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

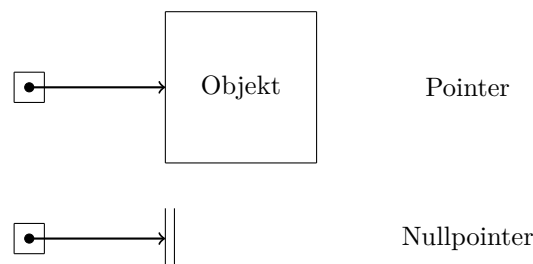
¹Github ist eine Seite, mit der man Quelltext online verwalten kann. Dies ist dahingehend ganz nützlich, dass man die Quelltext-Dateien relativ einfach miteinander synchronisieren kann, wenn man mit mehreren Leuten an einem Projekt arbeitet.

Kapitel I

Pointer

1. 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-Variablen 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

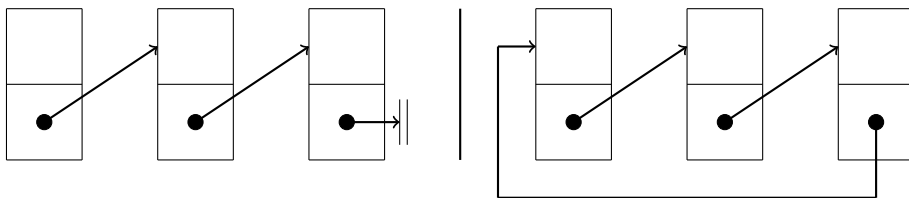
- Verwendung eines nicht definierten oder nicht gültigen Pointers in `deallocate`, `=>`, `associated`-Abfragen und normalen (nicht Pointer-) Kontext, das heißt in Expressions, in denen alle Pointer automatisch dereferenziert werden.
- Dangling Pointer entstehen, wenn das Zeigerziel verloren geht, z.B. 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.

2. Listen

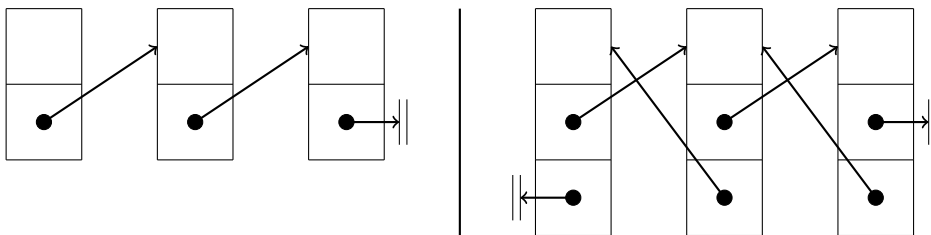
Eine Liste ist eine lineare Anordnung von Objekten des selben Typs. Eine Liste wird als verzweigte Struktur (oder als eindimensionales Feld - wird hier aber nicht behandelt) implementiert. Eine solche Liste hat 3 Attribute:

- 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 Löschoptionen. Wenn diese an beiden Enden der Liste notwendig sind, spricht man von einer Deque = double-ended-queue.

2.1. Grundoperationen auf einer Liste

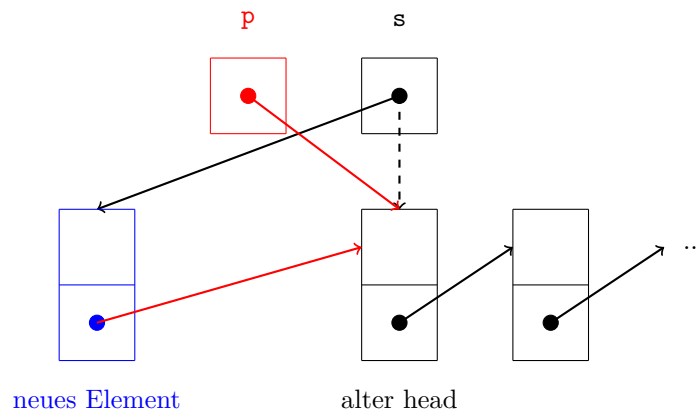
<code>init(L)</code>	Initialisierung der Liste, Anfangszustand "leer"	
<code>empty(L)</code>	als Abfragefunktion \rightarrow <code>.true.</code> falls L leer, sonst <code>.false.</code>	
<code>access_head(L,e)</code>	als Subroutine, gibt in e den Wert des head-Elements	head = Beginn einer Liste
<code>val.head(L)</code>	als Funktion \rightarrow Ergebnis ist Inhalt des head-Elements	
<code>access_tail(L,e)</code>	als Subroutine, gibt in e den Wert des tail-Elements	tail = Ende einer Liste

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

2.2. Grundoperationen auf einer Deque

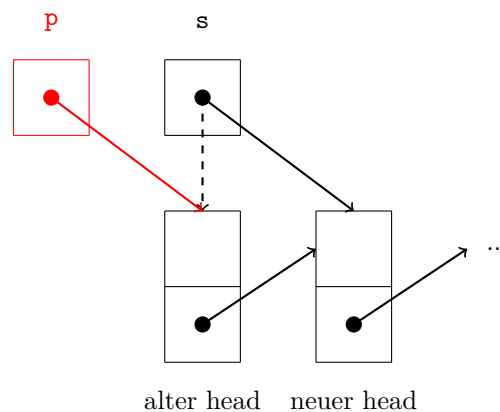
Der einfachste Fall ist der einer linearen, nicht zyklischen, einfach verketteten, endogenen Liste mit **s** als head-Pointer.

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!

3. 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-restricted-queues bzw. Input-restricted-queues. Das sind dequeues mit `push`, `pop`, `inject`, aber ohne `eject` bzw. eine deque mit `pop` und `eject` oder `push` und `inject`.

3.1. 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 `maxlength`, `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    maxlength = 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
```

4. 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.

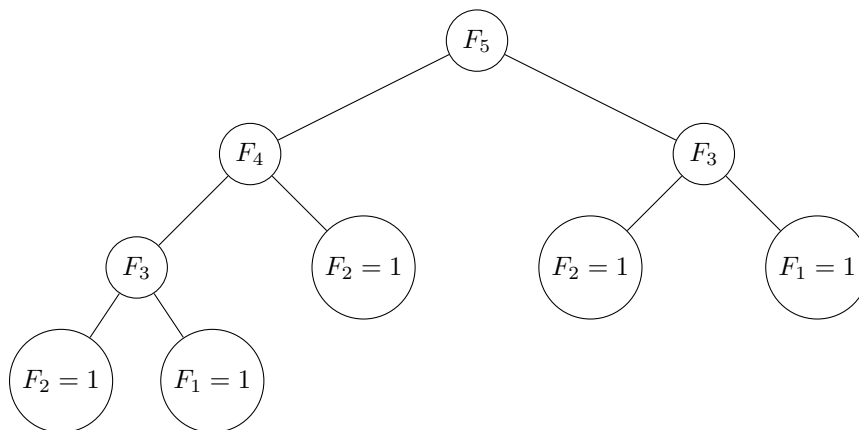
Ein Baum ist also rekursiv definiert und besitzt eine rekursive Datenstruktur. Wir brauchen deswegen rekursive Algorithmen zur Bearbeitung.

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 $T_n T_{n-1} \dots T_1 T_0$ lösen. Diese Rekursion ist leicht auflösbar. Das Problem lässt sich

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

$$P_{n,j} = \begin{cases} T_0 & n = 0 \\ T_0 P_{n-1,1} T_1 P_{n-1,2} \dots T_{k-1} P_{n-1,k} T_k & n > 0 \end{cases} \quad 1 \leq i \leq k$$

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: $h_{min} = \lfloor \log_2 N \rfloor$

1. Binäre Suchbäume

Ein binärer Suchbaum ist ein Binärbaum, bei dem im linken Teilbaum eines Knotens nur “kleinere” Elemente und im rechten Teilbaum nur “größere” Elemente 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 \Rightarrow 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 \Rightarrow 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

1. 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: $T(n) = \mathcal{O}(n^2) = \mathcal{O}(n^2 \cdot \log n) = \mathcal{O}(n^2 \cdot \sqrt{n}) = \mathcal{O}(n^3) = \dots = \mathcal{O}(2^n) = \mathcal{O}(n^n)$.

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:

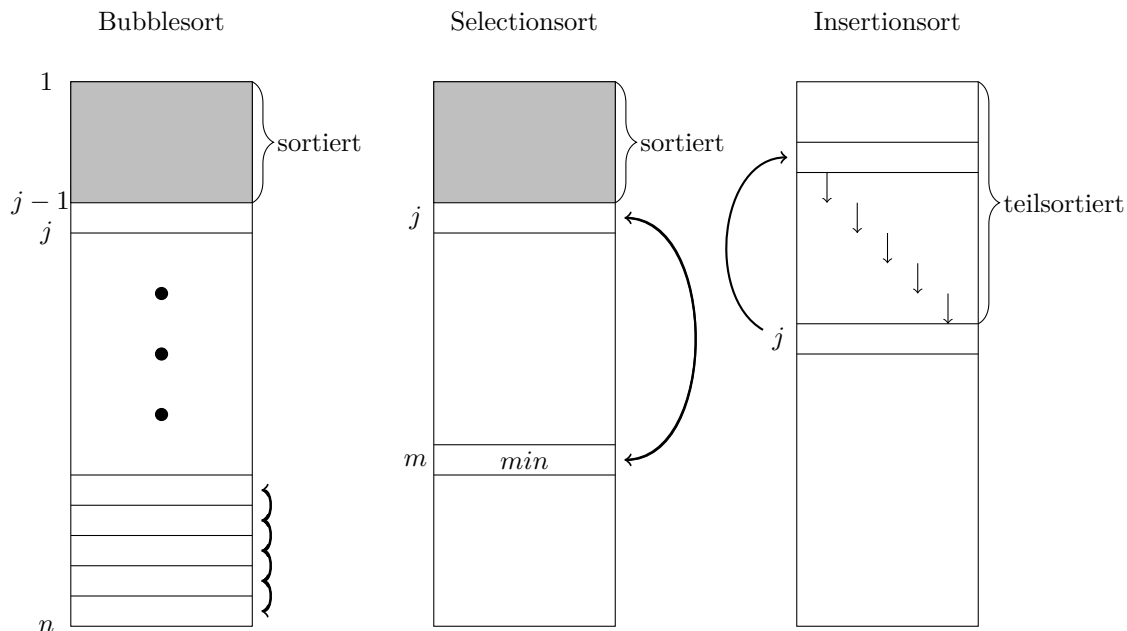
1. direktes Sortieren : Bewegen der Datenelemente inklusive key
2. indirektes Sortieren : Erzeugen einer Sortierpermutation σ der Indizes, wobei nur die Indizes in einem eigenen Feld und nicht die Datenelemente bewegt werden

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 $I = \{1, \dots, n\}$ ist eine Permutation von I , das heißt $\{\sigma(1), \dots, \sigma(n)\}$ mit $\sigma(i) \neq \sigma(j)$ für $i \neq j$.

2. 3 einfache Sortialgorithmen



Sortierv Verfahren	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, $\mathcal{O}(n)$ 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 Sortierv Verfahren

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:

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

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

□

3. 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

4. Mergesort

4.1. 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.

4.2. 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 Ele- mente	Aufwand Insertion- Schritt	Anzahl Tupel	Aufwand
1	k -Tupel: $\mathcal{O}(\quad)$	$(k \log_2 k +$	$0 \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k})$	$\mathcal{O}(n \log_2 k)$
2	k^2 -Tupel: $\mathcal{O}(\quad)$	$(k \log_2 k +$	$k(k-1) \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k^2})$	$\mathcal{O}(nk)$
3	k^3 -Tupel: $\mathcal{O}(\quad)$	$(k \log_2 k +$	$(k^3 - k) \cdot$	$\mathcal{O}(k))$	$\cdot \frac{n}{k^3})$	$\mathcal{O}(nk)$
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
						$\Sigma = \mathcal{O}(kn \log_2 n)$

5. 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

```

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

Zusätzliche Heap-Eigenschaft: $A_{\text{Parent}(i)} \geq A_i$, 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 :

$T_{\text{Heapify}} = \mathcal{O}(h)$ mit $h = \text{height}(A,i) = \mathcal{O}(\log_2 n)$.

```

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: $T_{\text{BuildHeap}}(n) = \mathcal{O}\left(\frac{n}{2} \log_2 n\right) = \mathcal{O}(n \log_2 n)$. Eine bessere Analyse bekommen wir, wenn die Höhe der Knoten betrachten: In einem Heap haben höchstens $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ Knoten die Höhe h .

⇒ Gesamtaufwand für `BuildHeap`:

$$\begin{aligned}
 \sum_{h=0}^{\lceil \log_2 n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathcal{O}(h) &= \mathcal{O}\left(n \cdot \sum_{h=0}^{\lceil \log_2 n \rceil} h \cdot \left(\frac{1}{2}\right)^h\right) \\
 \sum_{k=0}^n x^k &= \frac{x^{n+1} - 1}{x - 1} \\
 0 < x < 1 : \quad \sum_{k=0}^{\infty} k \cdot x^k &= \frac{1}{1-x} = (1-x)^{-1} \\
 \text{Differenzieren} \quad \sum_{k=1}^{\infty} k \cdot x^{k-1} &= \frac{1}{(1-x)^2} \\
 x = \frac{1}{2} : \quad \sum_{k=1}^{\infty} k \cdot \left(\frac{1}{2}\right)^{k-1} &= \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = 2 \\
 &\Rightarrow T(n) = \mathcal{O}(n)
 \end{aligned}$$

```

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

```

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

6. 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

```

⇒ $T(n) = \mathcal{O}(\log_2 n)$

```

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

```

⇒ $T(n) = \mathcal{O}(\log_2 n)$

```

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
```

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

7. Counting Sort

Der Counting Sort-Algorithmus ist dann besonders gut, wenn alle möglichen Schlüsselwerte aus der Menge $S = \{0, 1, 2, \dots, k-1\}$ 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 (~ 81 Millionen) nach ihrem Alter ($S = \{0, \dots, 150\}$) sortieren möchte.

Der Counting Sort eignet sich nicht dafür die Studenten der PROG-Vorlesung ($\sim 30 \odot$) nach ihrem Nachnamen (40 Zeichen $\Rightarrow S = \{0, \dots, 26^{40}\}$) 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
```

8. Radix-/Distribution Sort

Wenn wir annehmen, dass der Schlüssel z sich als Zahl mit d Ziffern zur Basis k darstellen lässt, also $z = [z_{d-1}z_{d-2}\dots z_1z_0]_k = \sum_{i=0}^{d-1} z_i k^i$, 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

1. Beispiele

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

Anmerkung

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

1.1. 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
```

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

1.2. 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

```

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

1.3. 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 $S(n) = \Theta(1)$.

1.4. Fibonacci-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.

1.5. 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.

1.6. Größter gemeinsamer Teiler

Es gilt:

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

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

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

Kapitel V

Implementierung der Grundrechenarten in Rechnern

Anhang

Index

- allgemeinen Rekursion, [11](#)
- average case, [13](#)
- Baum, [11](#)
 - Grad, [11](#)
 - Höhe, [11](#)
 - Level, [11](#)
- best case, [13](#)
- Binärbaum, [11](#)
- binären Suchen, [13](#)
- binärer Suchbaum, [12](#)
- Dangling Pointer, [3](#)
- Datenreferenzen, [2](#)
- Deque, [4](#)
- direktes Sortieren, [14](#)
- Durchlauf, [13](#)
- Elementaroperation, [13](#)
- externes Sortierverfahren, [13](#)
- FIFO-Prinzip, [8](#)
- Heap, [22](#)
- Heap-Eigenschaft, [22](#)
- in-situ Sortierverfahren, [13](#)
- indirektes Sortieren, [14](#)
- Inorder, [12](#)
- Levelorder, [12](#)
- LIFO-Prinzip, [8](#)
- linearen Suchen, [13](#)
- Linksrekursion, [11](#)
- Liste, [4](#)
- Makroschritt, [13](#)
- Mikroschritt, [13](#)
- Pivotelement, [17](#)
- Postorder, [12](#)
- Preorder, [12](#)
- Queue, [8](#)
- Rechtsrekursion, [11](#)
- root, [11](#)
- Sentinel, [12](#)
- Sortierpermutation, [14](#)
- Speicherleichen, [3](#)
- stabil, [13](#)
- Verweise, [2](#)
- worst case, [13](#)
- Zeiger, [2](#)