```
\usepackage{ngerman,qtree,listings}
\title{Algorithmen und Datenstrukturen}
\author{Ulrich K\"othe}

\date{29. Mai 2008}

\begin{document}
\maketitle

\tableofcontents

\subsection{weiteres Beispiel zu Heaps}

\begin{enumerate}
\item Heapbedingung (MaxHeap): Wurzel hat gr\"o\ss{}ere Priorit\"at als Kinder (
\item Upheap: repariert den Heap (Heap-Bedingung) falls Element k eine zu hohe P
\item Downheap: repariert den Heap falls Element k eine zu niedrige Priorit\"at
\item Komplexit\"at O(log n) (einf\"ugen und Element h\"ochster Priorit\"at entf
\end{enumerate}

\subsubsection{Beispiel am Wort ``SORTING''}
Anmerkung: Priorit\"at entspricht der Reihenfolge im Alphabet, d.h. A hat die kl
\newline
Anfang, S eingef\"uen): \\
\Tree [.S ] \\
O einf\"ugen: \\
\Tree [.S O ] \\
R einf\"ugen: \\
\Tree [.S O R ] \\
T einf\"ugen: \\
\Tree [.S [.O T ] R ] \\
T sitzt jedoch zu tief, darum up-heap: \\
\Tree [.S [.T O ] R ] \\
und nochmal upheap: \\
\Tree [.T [.S O ] R ] \\
I, N, G einf\"ugen: \\
\Tree [.T [.S O I ] [.R N G ] ] \\

Elemente entfernen: \\
Wurzel entfernen, Element mit geringster Warscheinlichkeit nach oben setzen, Hea
$\not{T}$
```

```
\Tree [.G [.S O I ] [.R N ]] (Heapbedingung nicht ergef\"ullt) \\
\Tree [.S [.G O I ] [.R N ]] (nach down heap) \\
\Tree [.S [.O G I ] [.R N ]] (down heap) \\
$\not{S}$
\Tree [.N [.O G I ] [.R ]] \\
(gleiche Vorgehensweise wie oben) \\
\Tree [.R [.O G I ] [.N ]] \\
(down heap) \\

\subsection{andere Heapvarianten:}
\begin{itemize}
\item Min-Max-Priority Queue (``Deap'', Double-Ended-Heap)
\item Binomialer Heap, effiziente Operation ``merge Heap'' O(n log (i$N_1 + N_2$
(Beweis durch binomial Koeffizienten, Zusammenf\"uhren zweier Priorit\"atslisten
\item Fibonacci-Heap: Einf\"ugen in amortisierter Zeit O(1)\\
(Beweis durch Fibonacci Zahlen)
\end{itemize}

\section{Assotiative Arrays}
\begin{itemize}
\item \"ahnlich Arrays X = a[i], a[i] = x, wobei der Typ von i beliebig sein kan
(beim Array: i $\in$ 0\ldots N-1
\item z.B. W\"orterbuch x = toEnglish[word], typ[word] = =  string (word ist in
(h\"aufigster Fall String als indices)
\item offensichtliche Realisierung: zur\"uckf\"uhren auf Suchen \\
Datenstruktur muss so erweitert werden, dass mehr Informationen gespeichert werd
verwende sequentielle Suche oder Baumsuche um die beiden Zugriffsoperatoren zu i
Sequentielle Suche a[i] $\in$ O (n) \\
Baumsuche a[i] $\in$ O (log n) )
\item Bsp. f\"ur Realisierung mit Baumsuche, std-map in C++

\end{itemize}

Es stellt sich die Frage, ob das auch schneller, d.h. O(1) geht.
 - Ja, mit einer Hashtabelle bzw. Hashing

\begin{itemize}
\item gegeben: Schl\"ussel aus Universum U, U = Menge allem was als Schl\"ussel
\item Definition einer Kollision, wenn $f(u_1 \in U) = f(u_2 \in U)$ d.h. zwei S
Eigenschaft: Falls $|U| > M$ sind Kollisionen unvermeidlich
\item Aufgabe: Entwerfen einer Hash-Funktion mit m\"oglichst wenigen Kollisionen
```

```
$\rightarrow$ Hashfunktion \"ahnlich eines Zufallszahlengenerators \\
jede $k \in 0\cdots M-1$ soll mit gleicher Wahrscheinlichkeit herauskommen (= un
\item Aber: f\"ur jede Hashfunktion gibt es ung\"unstige Konstellationoen $(U_f
$\Rightarrow$ Wahl einer guten Hashfunktion ist eine ‘‘Kunst’’\\
$\Rightarrow$ Analyse der Daten f\"ur ein gutes $f$
\item falls man die erlaubten Schl\"ussel schon kennt: \\
$U_f \subset U$ kommt nur $U_f vor \rightarrow $ ‘‘perfekte’’ Hashfunktion

\end{itemize}

\subsection{Beispiel: Monatsnamen}
U ist Strings der L\"ange 9 $\rightarrow 30^{9}$ m\"ogliche Strings, 12 werden B
$U_f = $\{’’Januar’’, ’’Februar’’, \ldots, ’’Dezember’’\} \\

\begin{itemize}
\item Hashfunktion Anfangsbuchstabe des Monatsnamen: \\
5bit M = 32 $(2^{5} \rightarrow$ J,F,M,A,M,J,J,A,S,N,D $\rightarrow$ offensichtl
\item Hashfunktion erste 3 Buchstaben $\rightarrow$ 15bit $\rightarrow$ M = $2^{
$\Rightarrow$ Jan, Feb, M\"ar, Apr, Mai, Jun, Jul, Aug, Sep, Nov, Dez \\
$\Rightarrow$ Keine Kollision, aber M ist gro\ss{}
\end{itemize}
Aufgabe:
$\Rightarrow$ minimale, perfekte Hashfunktion $|U_f| = M$ finden
$\Rightarrow$ siehe bungsaufgabe in 3 Wochen (Algortihmus beruht auf Graphentheo

\begin{itemize}
\item universelles Hashing: w\"ahle die Hashfunktion per Zufallszahl $\Rightarro
\item Cryptographische Hash-Funktionen wie z.B. md5, SHA1 u.a. f\"ur Verschl\"us
\item Anforderungen: \begin{itemize}
\item Kollision sehr unwahrscheinlich
\item Hash darf nicht invertierbar sein \\
$\Rightarrow$ M sehr gro\ss{} (z.B. M = $2^{128}$)
\end{itemize}
\end{itemize}

\subsection{Beliebte Hashfunktionen}
\begin{itemize}
\item Assotiatives Array M ist festgelegt (Gr\"o\ss{}e des Arrays) \\
$f(U) = f’(U) \% M \Rightarrow \in 0\ldots M-1$ \\
$f’(U)$ falls U unsigned int (32bit int Datentyp) \\
U signed int $\Rightarrow |U| f’(U) = U $ oder Typkonvertierung (in C: (unsigned
```

```
\item andere Schl\"usseltypen interpretiert man als Array of Byte $f'(U)$ konver
\item Beliebte Beispiele: Bernsteinfunktion
\begin{lstlisting}
def bHash(u): #U: Array of Byte
h = 0
for k in U:
k = 33 * h + k       (*)
return h
\end{lstlisting}
\item modifiziertes Bernsteinhash: wie oben
(*) h = 33 * h \^ k \# \^ = bitweises xor
\item Shift-Add-Xor-Funktion:
\begin{lstlisting}
def saxHash(U):
h=0
for k in U:
h ^= (h << 5) + (h >> 2) + k
return h
\end{lstlisting}

\item Fowler/Novel/Vo-Funktion: \\
\begin{lstlisting}
def FNVHash(U):
h = 2166136261
for k in U:
h = (h * 16777619) ^ k
return h
\end{lstlisting}
\item \ldots
\end{itemize}
$\rightarrow$ Hashtabelle: Kollision unvermeidlich wenn Schl\"ussel nicht bekann
\begin{itemize}
\item besteht aus einer Hashfunktion + Array der gr\"o\ss{}e M
\item Insert (naiv): \\
\begin{lstlisting}
def insert(u):
index = hash(u) % M
array[index] = u
\end{lstlisting}
$\rightarrow$ funktioniert so nicht, weil Kollisionen zum \"uberschreiben f\"uhr
```

```
\end{itemize}
\subsection{geschickte Behandlung von Kollision: Zwei Ans\"atze:}
\begin{itemize}
\item lineare Verkettung
\item offene Adressierung
\end{itemize}

\subsubsection{lineare Verkettung:}
\begin{itemize}
\item jedes Arrayfeld kann dadurch beliebig viele Datenelemente speichern \\
\begin{lstlisting}
def insert(u):
index = hash(u) % M
array[index].append(u)
 #insert() O(1)

def get(u):
index = hash(u) % M # O (1)
k = sequentialsearch(array[index],u) # O (a)
# ges. O(1+a)
if k == -1:
return none
else:
return Array[index][k]
\end{lstlisting}

F\"ullstand $\alpha = \frac{N}{M}$ wobei N die Gre der Hashtabelle und M die GGr
Wenn alles klappt \\
bei uniformen Hashing ist die L\"ange der Listen O($\alpha$) bei gleichm\"a\ss{}
Bsp: C++ std.hashmap \\

$\Rightarrow$ n\"achste Woche offene Addressierung
\end{itemize}


\end{document}
```