3. Sortieren

3.1 Elementare Sortierverfahren

3.2 Quicksort

3.3 Heap Sort

3.4 Merge Sort

3.5 Radix Sort

3.6 Externes Sortieren

Das Problem

- Es soll eine Menge von Datensätzen sortiert werden („file of records", Datei).
- Jeder Datensatz enthält einen Schlüssel („key“), nach dem sortiert wird.
- Internes Sortieren: Alle Datensätze haben im Speicher Platz.
  Externes Sortieren: Es müssen externe Speichermedien einbezogen werden (Magnetplatte, Magnetband).

Beurteilungskriterien:

- Laufzeit $O(N^2)$, $O(N\log N)$,...
- Speicherbedarf
  - Sortieren "in place" (kein zusätzlicher Bedarf)
  - verkettete Listen (Ändern der Verkettung)
  - zweite Kopie der Datensätze muß angelegt werden.
- Stabilität
  - die relative Ordnung von gleichen Schlüsseln bleibt erhalten. Wichtig, wenn nacheinander nach verschiedenen Schlüsseln sortiert werden soll.

Vorbemerkung 1 zu allen nachfolgenden Programmen

Wenn wir im folgenden Programmstücke zur Erläuterung präsentieren, so sind dies Unterprogramme zur Sortierung von N Zahlen oder Zeichen. Diese muß man sich in ein Hauptprogramm eingebettet vorstellen, etwa wie folgt (für N = 3):

```
#include <stdio.h>
sort3(int a[], int N)
  {
   int t;
   if (a[1] > a[2])
     { t = a[1]; a[1] = a[2]; a[2] = t; }
   if (a[1] > a[3])
     { t = a[1]; a[1] = a[3]; a[3] = t; }
   if (a[2] > a[3])
     { t = a[2];a[2] = a[3]; a[3] = t; }
  }

#define maxN 100
main()
  {
   int N, i, a[maxN+1]
   N = 0; while (schanf("%d", &a[N+1] !=EOF) N++;
   a[0] = 0; sort3(a, N);
   for (i = 1; i <=N; i++) printf("%d", a[i]);
   printf("\n");
  }
```

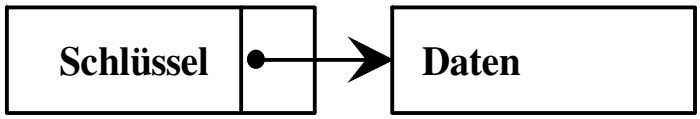
Vorbemerkung 2 zu allen nachfolgenden Programmen

Alle Beispielprogramme arbeiten auf Feldern von ganzen Zahlen oder Zeichen. Die Anpassung an andere Arten von Schlüsseln ist trivial, solange eine Kollationsfolge definiert ist.

Sind die Datensätze selbst sehr groß im Verhältnis zu den Schlüsselattributen, so ist zu überlegen, ob die Schlüsselattribute abgetgrennt, um einen Zeiger auf den Datensatz erweitert und dann getrennt sortiert werden sollten:

Aus

wird



Vorbemerkung 3 zu allen nachfolgenden Programmen:

Zur Messung der Laufzeit der Verfahren verwenden wir zwei naheliegende Größen:

- Anzahl Schlüsselvergleiche C (comparisons)
- Anzahl Bewegungen von Datensätzen M (movements)

Für beide Parameter interesssieren uns

- die Anzahl im günstigsten Fall (min)
- die Anzahl im schlechtesten Fall (max)
- die Anzahl im Mittel (avg)

3.1 Elementare Sortierverfahren

3.1.1 Sortieren durch Auswahl (Selection Sort)

Algorithmus:

Suche das kleinste Element und vertausche es mit dem ersten Element.
Suche das zweitkleinste Element und vertausche es mit dem zweiten Element usw.
bis die gesamte Menge sortiert ist.

Programm

```
selection (int a[], int N)
  {
   int i, j, min, t;
   for (i = 1; i < N; i++)
      {
      min = i;
      for (j = i+1; j <= N; j++)
          if (a[j] < a[min]) min = j;
      t = a[min]; a[min] = a[i]; a[i] = t;
      }
   }
```

Sortieren durch Auswahl, Beispiel

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
	S	O	R	T	I	N	G	E	X	A	M	P	L	E
		O	R	T	I	N	G	E	X	S	M	P	L	E
			R	T	I	N	G	O	X	S	M	P	L	E
				T	I	N	G	O	X	S	M	P	L	R
					I	N	T	O	X	S	M	P	L	R
						N	T	O	X	S	M	P	L	R
							T	O	X	S	M	P	N	R
								O	X	S	T	P	N	R
									X	S	T	P	O	R
										S	T	P	X	R
											T	S	X	R
												S	X	T
													X	T
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

Sortieren durch Auswahl, Komplexität

Berechnung der Anzahl Vergleiche:

Erstes Element: N -1 Vergleiche

Zweites Element: N - 2 Vergleiche

-
-
-

Insgesamt: $N+(N-1)+(N-2)+.....+1$

$$= \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

$$= 0(N^2)$$

Dies gilt unabhängig von einer Vorsortierung!

$$C_{min}(N) = C_{max}(N) = C_{avg}(N)$$

Berechnung der Anzahl Kopier-Operationen:

Bewegung von Datensätzen: 3 pro Austausch,

also insgesamt 3 (N - 1)

= O (N)

wobei $M_{min}(N) = M_{max}(N) = M_{avg}(N)$

Gesamtkomplexität:

Somit ist Sortieren durch Auswahl $O(N^2) + O(N) = O(N^2)$

Sortieren durch Auswahl, Kommentare

Dieser Algorithmus führt eine geringe Anzahl von Datensatz-Bewegungen durch (O (N)),
aber eine große Anzahl von Schlüsselvergleichen (O (N^2)).

Sind Datenbewegungen sehr teuer und Schlüsselvergleiche sehr billig, z.B. bei sehr kleinen Schlüsseln und sehr großen Datensätzen, so kann Sortieren durch Auswahl sehr gut sein.

3.1.2 Sortieren durch Einfügen (Insertion Sort)

Algorithmus

Die N zu sortierenden Elemente werden nacheinander betrachtet und in die jeweils bereits sortierte, anfangs leere Teilfolge an der richtigen Stelle eingefügt.

Anmerkung: So sortiert man Spielkarten auf der Hand.

Programm:

```
insertion (int a[], int N)
  {
    int i, j, v;
    for (i = 2; i <= N; i++)
      {
        v = a[i]; j = i;
        while (a[j-1] >v)
          { a[j] = a[j-1]; j--; }
        a[j] = v;
      }
  }
```

Sortieren durch Einfügen, Beispiel

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
	S	O												
	O	S	R											
		R	S	T										
				T	I									
	I	O	R	S	T	N								
		N	O	R	S	T	G							
	G	I	N	O	R	S	T	E						
	E	G	I	N	O	R	S	T	X					
									X	A				
	A	E	G	I	N	O	R	S	T	X	M			
					M	N	O	R	S	T	X	P		
								P	R	S	T	X	L	
					L	M	N	O	P	R	S	T	X	E
			E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

Sortieren durch Einfügen, Komplexität

Berechnung der Anzahl Vergleiche:

Zum Einfügen des i-ten Elements ($1 \leq i \leq N-1$) benötigt man zwischen 1 und i Vergleiche.

$C_{min}(N) = N - 1$

$C_{max}(N) = \sum_{i=2}^{N} i$

Im Mittel ist die Hälfte der dem i-ten Element vorangehenden Elemente größer als das einzufügende Element, d.h. es werden i/2 Vergleiche benötigt.

$$C_{avg}(N) = \sum_{i=2}^{N} \frac{i}{2} = O(N^2)$$

Berechnung der Anzahl Kopieroperationen:

Für i-tes Element werden zwischen 2 und i+1 Bewegungen von Datensätzen durchgeführt. Deshalb folgt analog:

$M_{min}(N) = 2*(N-1)$

$M_{max}(N) = \sum_{i=2}^{N} (i+1)$

$$M_{arg}(N) = \sum_{i=2}^{N} \frac{(i+1)}{2} = O(N^2)$$

Gesamtkomplexität:

Somit ist Sortieren durch Einfügen im Mittel

$O(N^2) + O(N^2) = O(N^2)$

Sortieren durch Einfügen, Kommentare

Für bereits sortierte Dateien ist Sortieren durch Einfügen linear (O (N)) !

Für stark vorsortierte Dateien ist Sortieren durch Einfiügen immer noch fast linear: es "erkennt" die Vorsortierung. In entsprechenden Anwendungsfällen ist es eine gute Wahl, zum Beispiel beim "Einsortieren" von wenigen neuen Datensätzen in eine große, sortierte Datei.

Genauer kann man sagen: Der Aufwand hängt direkt von der Zahl der *Inversionen* (Fehlstellungen) ab: das ist die Anzahl der Elemente links von einem Element i, die größer sind als Element i.

3.1.3 Bubble Sort

Algorithmus

Durchlaufe die Datenmenge und vertausche alle Paare (benachbarte Elemente) von "falsch angeordneten" Elementen. Wiederhole dies, bis kein falsch angeordnetes Paar mehr gefunden wird. Die kleineren Schlüssel steigen dabei wie Blasen in einem Wasserglas allmählich nach oben.

Programm:

```
bubble (int  a[], int  N)
  (
   int i, j, t;
   for (i = N; i >= 1; i--)
     for  (j = 2;  j<= i;  j++)
       if (a[j-1] > a[j])
         { t = a[j-1]; a[j-1] = a[j]; a[j] = t; }
  }
```

Bubble Sort, Beispiel

Betrachten wir das Feld *a* mit den sieben Schlüsseln 15, 2, 43, 17, 4, 8, 47. Beim ersten Durchlauf werden folgende Vertauschungen benachbarter Elemente durchgeführt:

15, 2, 43, 17, 4, 8, 47
2, 15, 43, 17
17, 43, 4
4, 43, 8
8, 43, 47

Nach dem ersten Durchlauf ist die Reihenfolge der Schlüssel des Feldes *a* also

2, 15, 17, 4, 8, 43, 47.

Der zweite Durchlauf liefert die Reihenfolge

2, 15, 4, 8, 17, 43, 47.

Der dritte Durchlauf liefert schließlich

2, 4, 8, 15, 17, 43, 47,

also die aufsteigend sortierte Folge von Sätzen. Beim Durchlaufen dieser Folge müssen keine benachbarten Elemente mehr vertauscht werden. Das zeigt den erfolgreichen Abschluß des Sortierverfahrens.

Bubble Sort, Komplexität

Im **günstigsten Fall** wird beim ersten Durchlauf der Datenmenge kein "falsch angeordnetes" Paar gefunden, so daß der Algorithmus terminiert.

$C_{min}(N) = N-1$ Anzahl Vergleiche

$M_{min}(N) = 0$ Anzahl Bewegungen

Der **ungünstigste Fall** für Bubble Sort liegt bei einer zu Beginn absteigend sortierten Datenmenge vor, da dann in jedem Durchlauf genau 1 Element "nach oben" geschoben wird.

Für den i-ten Durchlauf ($1 \leq i \leq N-1$) werden also:

$3*(N-i)$ Kopiervorgänge und

$(N-i)$ Vergleiche durchgeführt

$C_{max}(N) = N+(N-1)$ Anzahl Vergleiche

$M_{max}(N) = \sum_{i=1}^{N-1} 3*(N-i)$ Anzahl Bewegungen

Die Analyse für den durchschnittlichen Fall ist komplizierter. Es ergibt sich:

$C_{avg}(N) = M_{avg}(N) = O(N^2)$

Bubble Sort, Kommentare

Für bereits sortierte Dateien ist Bubble Sort linear (O(N)) !

Da die Leistung im durchschnittlichen Fall nicht wesentlich besser ist als im ungünstigsten Fall, ist Bubble Sort i.A. keine gute Wahl für einen Sortieralgorithmus. Nur für den Fall, daß ein bereits nahezu vollständig sortiertes Feld (mit geringer Anzahl Inversionen) vorliegt, werden wenige Vergleichsoperationen und Bewegungen von Datensätzen ausgeführt.

Dabei ist Bubble Sort jedoch stark *asymmetrisch* bezüglich der Durchlaufrichtung, d.h. Inversionen am Anfang der Datenfolge werden schneller behoben als Inversionen am Ende der Datenfolge.

Variante: Shakersort
Das Datenfeld wird abwechselnd von oben und von unten durchlaufen. Bringt jedoch nur geringfügige Verbesserungen.

3.1.4 Shellsort

Algorithmus

Sortieren durch Einfügen ist langsam, weil Elemente nur um 1 Position nach rechts verschoben werden. Shellsort versucht deshalb, Elemente durch größere Sprünge schneller an ihre endgültige Position zu bringen.
Die Datenfolge wird dabei so umgeordnet, daß sie *h-sortiert* ist, d.h. daß man eine sortierte Datei erhält, wenn man jedes h-te Element (bei beliebigem Anfangselement) entnimmt. Eine h-sortierte Datenfolge besteht aus h unabhängig sortierten Datenfolgen, die einander überlagern. Die Datenfolge wird also in Teile zerlegt, die unabhängig sortiert werden. Man beginnt mit vielen Teilen und einem großen Elementabstand h und reduziert die Anzahl der Teile und den Elementabstand h schrittweise auf 1. Die Folge {h} heißt dabei *Inkrementfolge*.

Programm:

```
shellsort (int a[], int N)
  {
    int i, j, h, v;
    for  (h = 1; h <=N/9; h = 3*h+1) ;
    for  ( ; h > 0; h /= 3)
      for (i = h+1; i <= N; i++)
        {
         v = a[i]; j = i;
         while (j >h && a[j-h]>v)
           { a[j] = a[j-h]; j -= h; }
         a [j] = v;
        }
  }
```

Shellsort, Beispiel

Gewählte Werte für h: 13, 4, 1

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A													L	
	E													S
A	E	O	R	T	I	N	G	E	X	A	M	P	L	S
A				E				P				T		
	E				I				L				X	
		A				N				O				S
			G				M				R			
A	E	A	G	E	I	N	M	P	L	O	R	T	X	S
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

Shellsort, Komplexität

Die Berechnung der Komplexität von Shellsort ist außerordentlich kompliziert und abhängig von der Distanzfolge {h}.
Die Laufzeit ist nicht sonderlich abhängig von der ursprünglichen Ordnung der Daten.
Es gilt:
Shellsort führt nie mehr als $N^{3/2}$ Vergleiche aus (für die Distanzen 1, 4, 13, 40, 121, ...)

$C_{max}(N) \leq N^{3/2}$

Es ist beim Shellsort eine gewisse Sorgfalt bei der Auswahl der Distanzfolge {h} erforderlich. Beispielsweise führt ..., 64, 32, 16, 8, 4, 2, 1 wahrscheinlich zu einem schlechtem Ergebnis, da Elemente an ungeraden Positionen erst am Schluß mit Elementen an geraden Positionen verglichen werden.

Wenn als Inkrement alle Zahlen der Form $2^p 3^q$ gewählt werden, die kleiner als N sind, ist die Komplexität des Algorithmus $O(N \log^2 N)$.

3.1.5 Distribution Counting

Voraussetzung:

Sortieralgorithmus für Fälle, in denen die Schlüssel ganze Zahlen zwischen 0 und M-1 sind.

Algorithmus:

Es wird die Anzahl Schlüssel mit jedem Zahlenwert ermittelt und dieses Ergebnis zum Sortieren verwendet. Dazu wird in einem separaten Feld dieser Zahlenwert als Index eingesetzt, um die Elemente dorthin zu kopieren.

Programm:

```
for (j = 0; j <= M; j++)  count [j] = 0;
for (i = 1; i <= N; i++) count [a[i]]++;
for (j = 1; j <= M; j++)
    count [j] = count[j-1]+count[j];
for (i = N; i>= 1; i--)
    b [count[a[i]--] = a[i];
for (i = 1; i <= N; i++)  a[i] = b[[i];
```

Distribution Counting, Beispiel

M = 4 N = 14

a:	1	2	2	1	3	1	4	1	2	2	1	4	4	1
b:						1								
														4
													4	
					1									
										2				
									2					
				1										
												4		
			1											
											3			
		1												
								2						
							2							
	1													
a:	1	1	1	1	1	1	2	2	2	2	3	4	4	4

Shellsort, Kommentare

Komplexität:

- Vergleiche: $C_{min}(N) = C_{max}(N) = C_{avg}(N) = 2*M+3*N$
- Bewegungen: $M_{min}(N) = M_{max}(N) = M_{avg}(N) = 2*N$

Kommentare:

Dieses Verfahren ist für den betrachteten Typ von Dateien sehr gut geeignet. Die Laufzeit ist linear allerdings auf Kosten eines erhöhten Speicherbedarfs durch Anlegen einer Kopie.

Shellsort heißt auch sortieren mit abnehmenden Inkrementen.
Die Folgen von Elementen im Abstand h werden jeweils mit Sortieren durch Einfügen sortiert.

Das Herstellen einer h-sortierten Folge aus einer bereits k-sortierten Folge (k>h) zerstört **nicht** die k-Sortiertheit der Folge.

Shellsort ist eine bevorzugt Methode für viele Sortieranwendungen, da es selbst für große Folgen eine akzeptable Laufzeit aufweist und nur ein sehr kurzes Programm erfordert (also geringen Programmieraufwand).

3.2 Quicksort (C.A.R. Hoare 1960)

Idee:

Anwendung von **Divide and Conquer** zum Sortieren.

Algorithmusprinzip:

Zerlege (partitioniere) die Datenfolge in zwei Teile und sortiere diese Teile unabhängig voneinander nach dem selben Prinzip.

Programm:

```
quicksort (int a[],  int  l,  int  r)
  {
    int i;
    if (r > 1)
      {
        i = partition (l, r);
        quicksort (a, l, i-1);
        quicksort (a, i+1, r);
      }
  }
```

Quicksort, Erläuterungen zum Programm

Das entscheidende Element der Methode ist die Prozedur `partition`, die das Feld so umordnen muß, daß die folgenden drei Bedingungen erfüllt sind:

1. Für ein beliebiges i befindet sich das Element a[i] an seinem endgültigen Platz im Feld.
2. Alle Elemente in a[l], ...,a[i-1] sind kleiner oder gleich a[i].
3. Alle Elemente in a[i+1], ...,a[r] sind größer oder gleich a[i].

Vertauschungsstrategie:

- Wähle als Teilungselement das Element rechtsaußen (a[r]).
- Suche ab dem linken Ende ein Element, das größer (oder gleich) dem Teilungselement ist und suche ab dem rechten Ende ein Element, das kleiner (oder gleich) dem Teilungselement ist und vertausche beide.
- Fahre mit Suchen und Umtauschen fort, bis sich die Suchbereiche überlappen.
- Vertausche rechtsaußen-Element des linken Suchbereichs mit dem Teilungselement a[r].

Damit befindet sich a[r] an seinem endgültigen Platz, da alle Elemente links von a[r] kleiner als a[r] und alle Elemente rechts von a[r] größer als a[r] sind. a[r] wird als Teilungspunkt gewählt.

Quicksort, Beispiel für Zerlegung

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S									A	M	P	L	E
A	A	O						E	X	S	M	P	L	E
A	A	E	R	T	I	N	G	O	X	S	M	P	L	E
A	A	E	E	T	I	N	G	O	X	S	M	P	L	R

Programm:

```
quicksort (int  a[a], int  l, int  r)
  {
    int  v,  i,  j,  t;
    if  (r > 1)
      {
        v = a[r]; i = 1-l; j = r;
        for (;;)
          {
            while (a[++i] < v) ;
            while (a[--j] > v) ;
            if (i >= j) break;
            t = a[i]; a[i] = a[j]; a[j] = t;
          }
        t = a[i]; a[i] = a[r]; a[r] = t;
        quicksort (a, l, i-1);
        quicksort (a, i+1, r);
      }
  }
```

Quicksort, Beispiel

Durchführung des Sortierens:

A	A	E	E	T	I	N	G	O	X	S	M	P	L	R
A	A	E												
A	A													
				L	I	N	G	O	P	M	R	X	T	S
				L	I	G	M	O	P	N				
				G	I	L								
					I	L								
								N	P	O				
									O	P				
												S	T	X
													T	X

Quicksort, Komplexität

Worst Case:

Der schlechteste Fall tritt ein, wenn jedes Folgenelement einmal als Teiler gewählt wird. Dies ist z.B. bei einer bereits sortierten Datenfolge der Fall.

$C_{max}(N) \geq N+(N-1) + (N-2) +...1 = O(N^2)$
$M_{max}(N) = O(N^2)$
Gesamtlaufzeit: $O(N^2)$

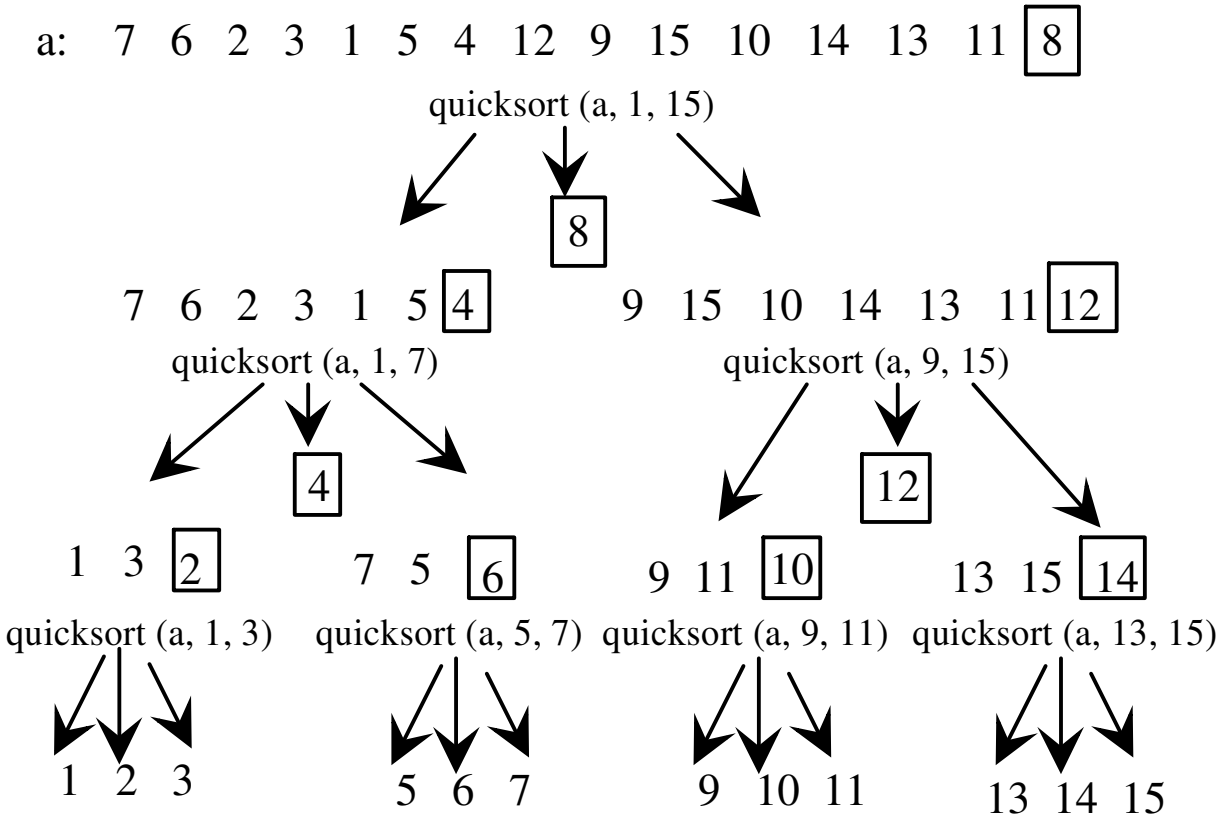
Best Case:

Im günstigsten Fall haben die durch Aufteilung entstehenden Teilfolgen stets etwa die gleiche Länge. Damit hat der Baum der initiierten rekursiven Aufrufe minimale Höhe (ungefähr log N).

Zur Aufteilung aller Folgen auf einem Niveau werden O(N) Schllüsselvergleiche benötigt, so daß gilt:

$C_{min}(N) = O(N \log N)$
Gesamtlaufzeit: O (N log N)

Quicksort, Komplexitätsbeispiel



Quicksort, Komplexität

Average Case:

Annahmen:

- Schlüssel seien die Zahlen 1, ..., N .
- Alle N! möglichen Anordnungen sind gleich wahrscheinlich.

Folgerungen:

- Jede Zahl wird mit Wahrscheinlichkeit 1/N als Teiler gewählt (Aufwand: N+1).
- Wird k gewählt, entstehen zwei Folgen der Länge (k-1) und (N-k), die wieder zufällig sind.

Behauptung:

$C_{avg}(N) = O(N \log N)$

Beweis:

Quicksort, Kommentare

- einfache Implementierung, aber trotzdem sehr effizient
- erfordert wenig Ressourcen (nur einen kleinen Hilfsstapel)
- Rekursion benötigt jedoch zusätzliche Ressourcen
- ist störanfällig bei Implementation (Shellsort ist einfacher zu implementieren und fast genauso effizient.)
- Wenn Quicksort auf einer bereits sortierten Datenfolge ausgeführt wird, hat es sein schelchtestes Verhalten ($O(N^2)$)!
- Quicksort hat im Mittel ein sehr gutes Verhalten: $O(N \log N)$!
- Wenn gleiche Schlüssel mehrfach vorkommen, muß das Programm geändert werden, da es dann für spezielle Anordnungen nicht funktioniert.

3.2.1. Quicksort ohne Rekursion

Rekursive Prozeduren können allgemein dadurch in iterative Prozeduren verwandelt werden, daß man einen Stapel verwendet, auf dem Zwischenergebnisse gespeichert werden. Im Falle von Quicksort enthält der Stapel sortierte Teildateien.

Programm:

```
quicksort (int  a[],  int  N)
  {
    int  i,  l,  r;
    l = 1; r = N; stackinit ();
    for (;;)
      {
       while (r > l)
         {
           i = partition (a, l, r);
           if (i-l > r-i)
             { push (l); push (i-1); l = i+1; }
           else
             { push (i+1); push (r); r = i-1; }
         }
       if (stackempty ()) break;
       r = pop(); l = pop();
      }
  }
```

Iterativer Quicksort, Beispiel

A	A	E	E	T	I	N	G	O	X	S	M	P	L	R
A	A	E												
A	A													
				L	I	N	G	O	P	M	R	X	T	S
												S	T	X
													T	X
				L	I	G	M	O	P	N				
				G	I	L								
					I	L								
								N	P	O				
									O	P				

Iterativer Quicksort, Kommentare

- Nur die größere von beiden Teilfolgen wird im Stapel abgelegt (Ersatz des ersten Rekursionsaufrufs).

- Es geschieht eine *End-Rekursionsbeseitigung,* d.h. da der zweite rekursive Aufruf von `quicksort` am Ende der rekursiven `quicksort`-Prozedur steht, kann er durch eine "Parameterersetzung" und ein "goto" ersetzt werden.

- Der Stapel muß nur für O(log N) Indizes Platz haben, da jeweils nur die kleinere Teilfolge abgelegt wird, d.h. eine Folge die weniger als halb so groß ist wie die zuvor abgelegte.

- Im schlimmsten Fall benötigt die iterative Version wesentlich weniger Speicherplatz (O(log N)) als die rekursive Version, die dann O(N) rekursive Aufrufe hat, was O(N) Speicherplatz auf dem Stapel entspricht.

- Wenn beide Teilfolgen nur 1 Element besitzen, wird eine Datei mit r = l auf dem Stapel abgelegt, nur um sofort wieder entfernt zu werden. (-->Verbesserung, Übungsaufgabe).

3.2.2. Strategien zur Wahl des Teilungselements

Ansatz:

Die Wahl des Teilungselements am rechten Ende des aufzuteilenden Bereiches führt bei bereits sortierten Dateien zum Worst-Case-Aufwand.

Bessere Strategien:

- Wahl eines zufälligen Elementes (Zufalls-Strategie)
- Wahl des mittleren von 3 Elementen (*3-Median-Strategie*)

Zufallsstrategie:

Algorithmus:

Man wählt einen zufälligen Index k zwischen l und r und vertauscht das Element a[k] mit a[r], bevor mit der Aufteilung des Bereichs a[l]...a[r] begonnen wird.

Kommentare:

- Der ungünstigste Fall tritt mit einer vernachllässigbar kleinen Wahrscheinlichkeit ein.

- Der Einsatz eines vollständigen Zufallszahlengenerators bringt zusätzlichen Aufwand.

- Alle Eingabefolgen werden (annähernd) gleich behandelt.

- Der Erwartungswert für die zum Sortieren einer beliebigen, aber festen Eingabefolge erforderliche Anzahl an Vergleichen ist O(N log N).

3 - Median- Strategie

Algorithmus:

Man wählt 3 Elemente im aufsteigenden Bereich (i.d.R. vom linken Ende, vom rechten Ende und aus der Mitte).
Diese 3 Elemente sortiert man auf ihren Plätzen und nimmt das mittlere als Teilungselement.

Kommentare:

- Es besteht eine gute Chance dafür, daß das mittlere Element eine Aufteilung in annähernd gleiche Teile leistet.

- Wenn die Elemente a[r], a[l] und a[(l+r)%2] gewählt werden, muß `quicksort` nur noch die Elemente zwischen a[l+1] und a[r-1] sortieren.

- Damit verkürzt sich die Gesamtlaufzeit um etwa 5%.

- Eine Kombination des iterativen Quicksort mit der 3-Median-Strategie kann die Laufzeit ggü. dem einfachen rekursiven Quicksort um 25-30% verkürzen.

3.3 Heap Sort

Idee:

Aufbau einer Datenstruktur, die die zu sortierenden Elemente enthält und auf der das Auslesen eines bestimmten Elementes besonders schnell geht. Es wird dann durch Auswahl sortiert, wobei die Auswahl jedoch geschickt organisiert wird.

3.3.1 Die Datenstruktur der Prioritätswarteschlangen Priority Queues

Idee:

Datenstruktur, die die Eigenschaft besitzt, daß das Einfügen eines neuen Elementes und das Löschen des größten Elementes besonders einfach/ schnell geht.

Vorraussetzung ist dabei, daß die Datensätze numerische Schlüssel (Prioritäten) besitzen.

Wichtigste Operationen:

Construct:	Aufbau einer Prioritätswarteschlange
Insert:	Setzt ein neues Element ein
Remove:	Löscht das größte Element

Prioritätswarteschlangen, Implementationen

Möglichkeit 1: ungeordnete Liste

In einem Array a[1],..., a[N] wird die Priorität des i-ten Elementes in a[i] gespeichert.

Construct:		Einfache sequentielle Zuweisung
Insert	:	Am Ende anhängen
Remove	:	Größte Element suchen und durch letztes Element ersetzen

Programm:

```
static int a[maxN+1], N;
construct (int b[], int M)
  { for (N = 1; N <= M; N++) a[N] = b[N]; }
insert (int v)
  { a[++N] = v; }
int remove ()
  {
    int j, max, v;
    max = 1;
    for (j = 2; j <= N; j++}
      if (a[j] > a[max]) max = j;
    v = a[max];
    a[max] = a[N--];
    return v;
  }
```

Kommmentare:

- gibt keine besonders schnelle Implementierung für das Löschen des größten Elementes (z.B. sequentielle Suche)

  Remove: O(N)

- Einfügen funktioniert besonders schnell

  Insert: O(N)

- Erzeugung der Datenstruktur ist trivial

  Construct: O(N)

Prioritätswarteschlangen, Implementationen

Möglichkeit 2: geordnete Liste

In einem Array a[1],..., a[N] werden die Elemente nach steigender Priorität geordnet.

Construct:		Erzeugen einer sortierten Liste
Insert	:	Suchen nach der Einsetzposition,
		Verschieben des Restes, Einsetzen
Remove	:	Letztes Element a[N] entfernen

Programm:

--->Übungsaufgabe

Kommentare:

- besonders schnelle Implementierung für Löschen des größten Elementes

  Remove: O(1)

- Einfügen braucht lange

  Insert: O(N)

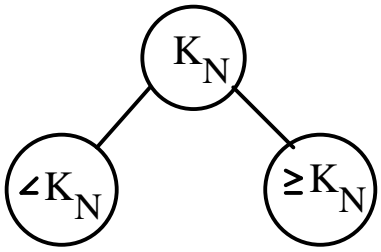
- Erzeugung muß mit Hilfe eines Sortieralgorithmus geschehen

  Construct: Z:B. O(N log N) mit Quicksort

Prioritätswarteschlangen, Implementationen

Möglichkeit 3: Binärbaum (Binärer Suchbaum)

In einem Binärbaum werden die Elemente so abgespeichtert, daß linke Kinder kleiner als der Vater und rechte Kinder größer als der Vater sind.



Construct:		Generiere Baum durch fortlaufendes Insert
Insert	:	Suche, beginnend bei der Wurzel, einen Weg bis zu der Stelle, an der das neue Element stehen muß und setzte es dort ein.
Remove	:	Knoten ganz rechts außen löschen (größte Element)

Programm:

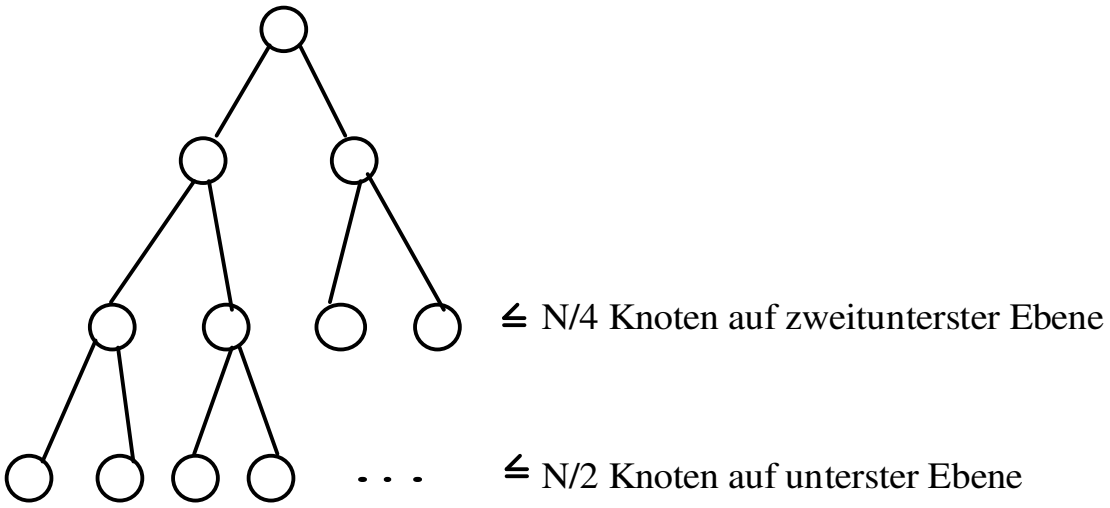
--->Übungsaufgabe

Kommentare:

- schnelle Implementierung von Remove:

  O(H) = O(log N)

  wobei H die Höhe des Baumes sei (= Anzahl Ebenen), die nicht größer als log N wird.



  Jede Ebene hat halb so viele Knoten wie die nächste
  => es gibt höchstens log N Ebenen.

- schnelle Implementierung von Insert:

  O(H) = O(log N)

- schnelle Implementierung von Construct:

  O(N*H) = O(N log N)

Prioritätswarteschlangen, Kommentare

- Mit Prioritätswarteschlangen kann ein Sortieralgorithmus implementiert werden, indem wiederholt `Remove` benutzt wird, um die Prioritätswarteschlangen abzubauen; dabei werden die Elemente in umgekehrter Reihenfolge erhalten (größtes Element zuerst).

- Implementationsmöglichkeit 1 entspricht dann Selection Sort.

- Implementationsmöglichkeit 2 entspricht dann Insertion Sort.

- Implementationsmöglichkeit 3 kann noch verbessert werden durch eine geeignetere Knotenanordnung: Heap

3.3.2 Die Datenstruktur des Heap

Idee:

Verwendung eines Binärbaums, der so aufgebaut ist, daß das Einsetzen eines Elementes und das Auslesen des größten Elementes besonders schnell geht.

=> Auslesen : größtes Element = Wurzel

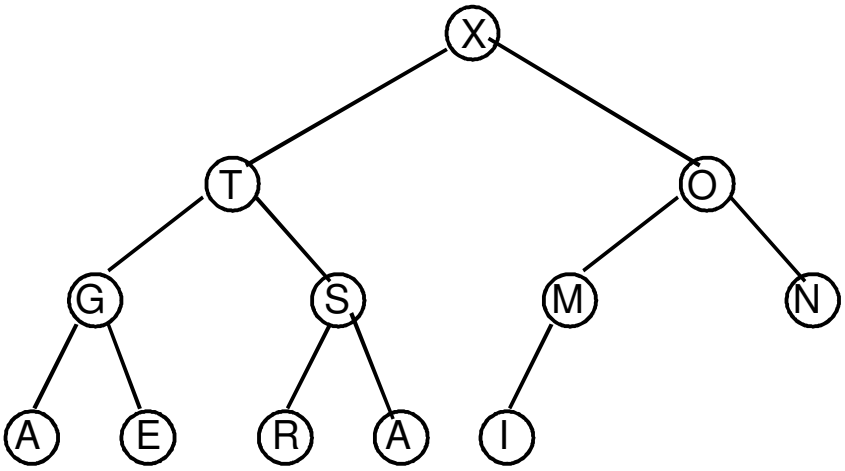
Einsetzen : Baum minimaler Höhe, d.h. vollständig

Definition:

Ein Heap ist ein **vollständiger** Binärbaum:

- bei dem der Wert der Wurzel größer oder gleich dem Wert der Kinder ist und
- bei dem die Kinder selbst Heaps sind.

Heap, Beispiel



Heap, Implementation

Die Datenstruktur des Heap (Halde) speichert Datensätze in einem Feld $F = k_1,..., k_n$ so, daß jeder Schlüssel k_i garantiert größer ist als die Schlüssel auf zwei bestimmten anderen Positionen:

$$k_i \geq k_{2i} \text{ und } k_i \geq k_{2i+1} \text{ für } 2i+1 \leq N$$

Beispiel:

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

Kommentare:

- Indexierung vorteilhaft:

  Elternknoten - Index = Kinderknoten - Index / 2

  Kinderknoten - Index = Elternknoten - Index * 2 und

  (Elternknoten - Index * 2) +1

- Indexierung entspricht einem ebenenweisen durchzählen der Knoten beginnend beim Wurzelknoten

- der größte Schlüssel befindet sich stets auf der erste Position im Feld

- Operationen entlang eines Pfades im Heap sind im logarithmischer Zeit ausführbar, da die Höhe des Heaps log N nicht übersteigt

Heap, Operationen

Alle Heap- Operationen machen zunächst eine einfache Modifikation an der Struktur des Heaps, die i.a. die Heap- Bedingung verletzt. Anschließend werden Vertauschungen vorgenommen, so daß wieder ein korrekter Heap entsteht.

Wichtigste Heap - Operationen:

- insert: Einfügen eines Elementes unter Beibehaltung der Heap - Bedingung

- remove: Löschen eines Elementes unter Beibehaltung der Heap - Bedingung

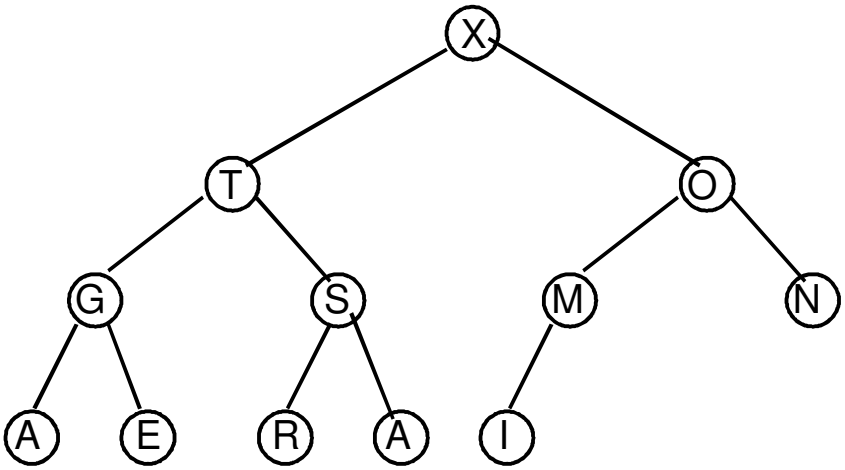
Damit ist `construct` als fortlaufendes `insert` aller Elemente der Suchtfolge zu realisieren.

Heap, Insert

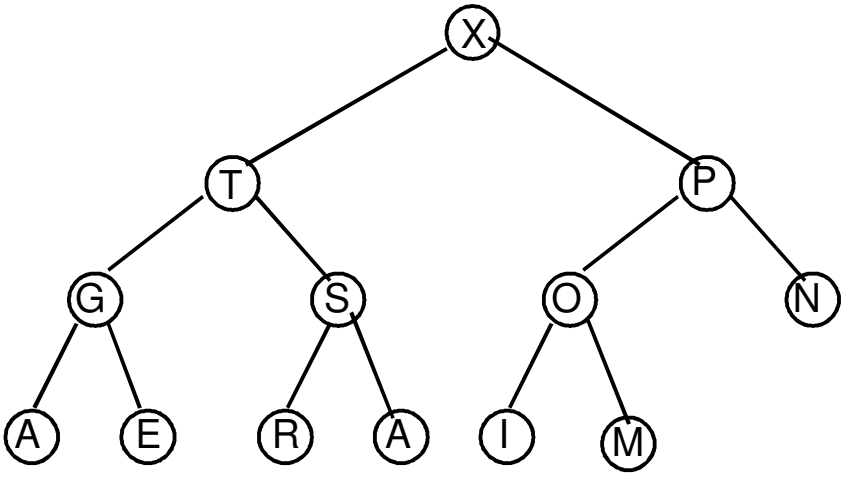
Algorithmus:

- Größe des Heaps (Array) N wird um 1 erhöht
- Neues Element wird in a[N] abgelegt.
- Solange neues Element > Elternknoten, vertausche beide.

Beispiel:



Einfügen von P:



Programm:

```
upheap (int k)
  {
    int v;
    v = a[k]; a[0] = INT_MAX;
    while (a[k/2] <= v)
      { a[k] = a[k/2]; k = k/2; }
    a[k] = v;
  }

insert (int v)
  {
    a[++N] = v;
    upheap (N);
  }
```

Aufwand:

Insert ist in O(log N) durchführbar, da maximal log N Vertauschungen gemacht werden müssen.

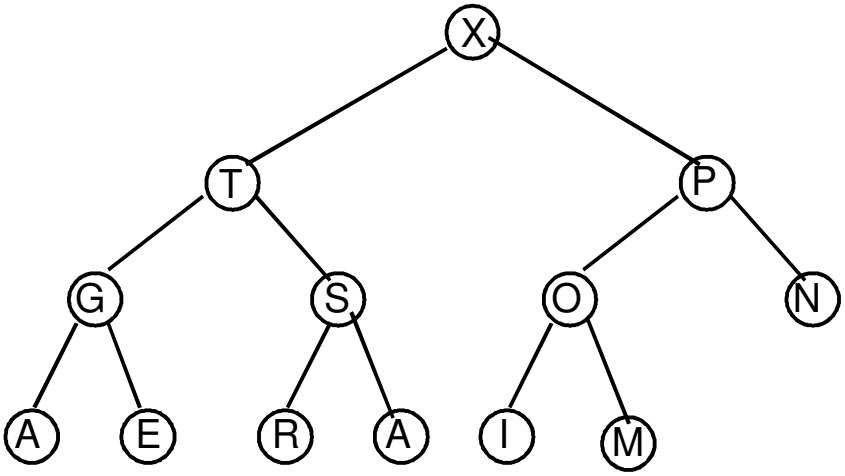
Heap, Remove

Algorithmus:

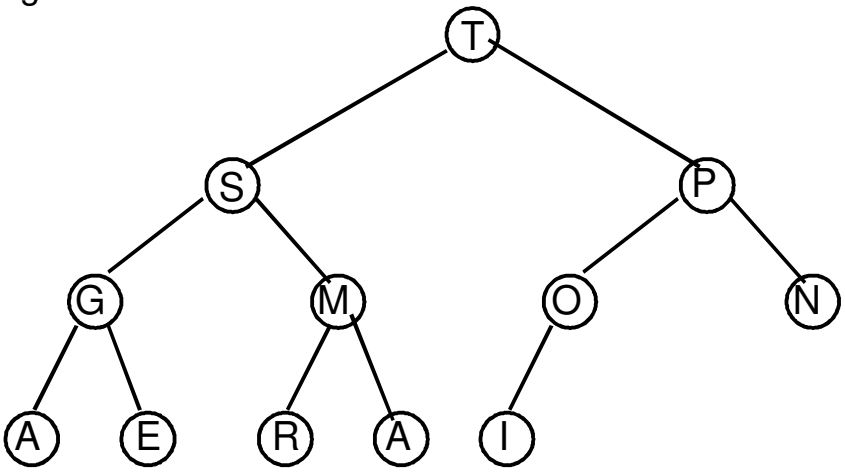
- Wurzel wird ausgegeben
- Letztes Element a[N] wird an a [1] abgespeichert
- Größe des Heaps wird um 1 Element reduziert

Solange Element kleiner ist als das größere der beiden Kinder: vertauschen!

Beispiel:



Entfernen des größten Elementes X:



Programm:

```
downheap(int k)
{
    int j, v;
    v = a[k];
    while (k<=N/2)
      {
          j = k+k;
          if (j<N && a[j]<a[j+1]) j++;
          if (v>=a[j]) break;
          a[k] = a[j]; k=j;
      }
    a[k] = v;
}
```

Aufwand:

- remove ist auch in O(log N) durchführbar
- die Bestimmung des Maximums an sich benötigt nur 1 Schritt, ist also O(1), aber der Baum muß noch reorganisiert werden.

3.3.3 Sortieren mit Hilfe eines Heap

Algorithmus von J.W.J. Williams:

- Konstruiere aus den zu sortierende Elementen einen Heap.
- Solange der Heap nicht leer ist,
  entferne das größte Element und stelle die Heap-Bedingung wieder her.

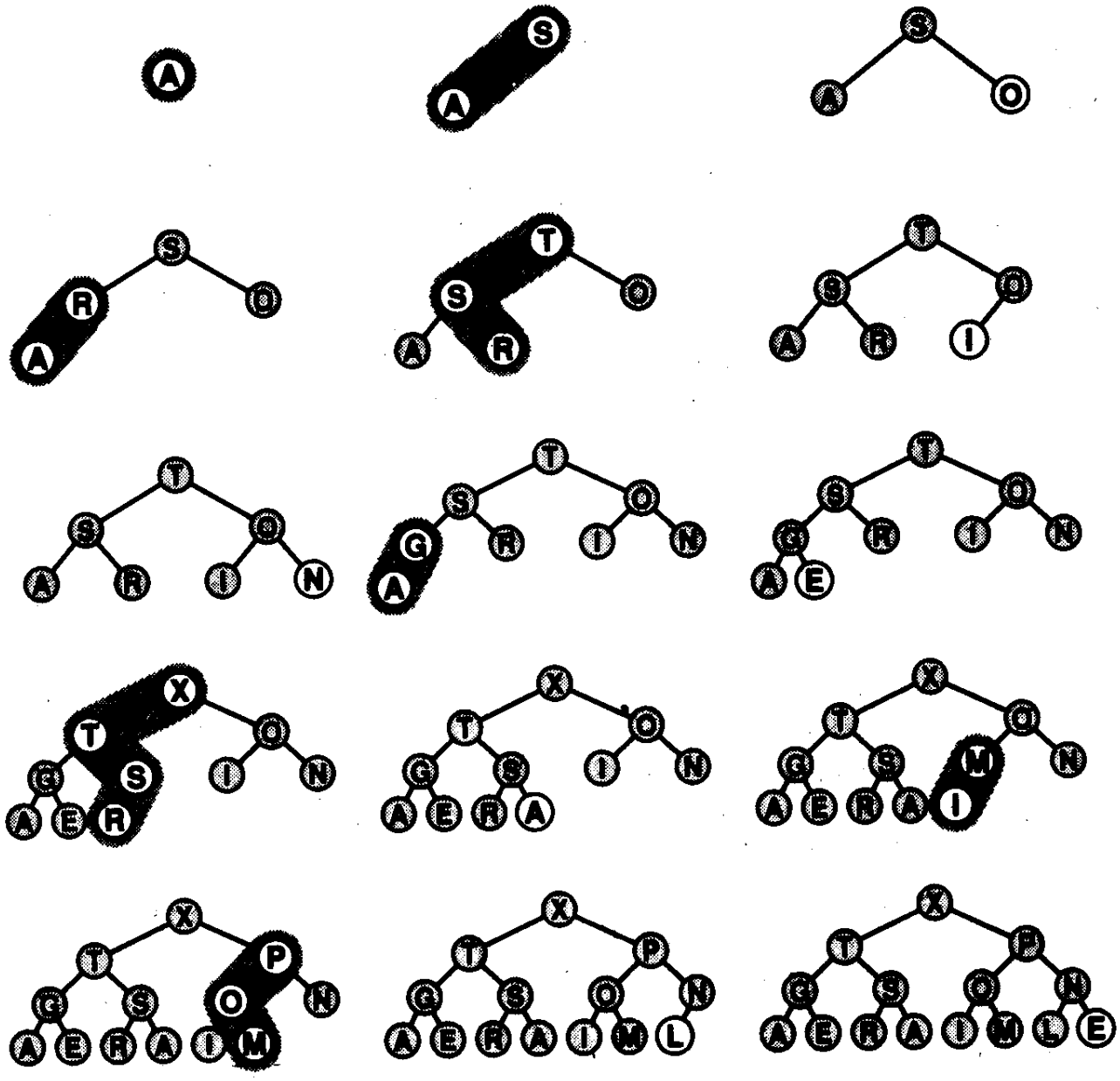
Programm

```
heapsort (int a[], int N)
  {
    int k;
    construct (a, 0);
    for (k = 1; k <= N; k++) insert (a[k]);
    for (k = N; k >= 1; k--) a[k] = remove();
  }
```

Dabei wird das entfernte Element jeweils auf den Platz a[N] kopiert, bevor N um 1 erniedrigt wird mit *remove*, so daß kein zusätzlicher Speicherplatz benötigt wird.
Am Ende ist a[1] ... a[N] sortiert.

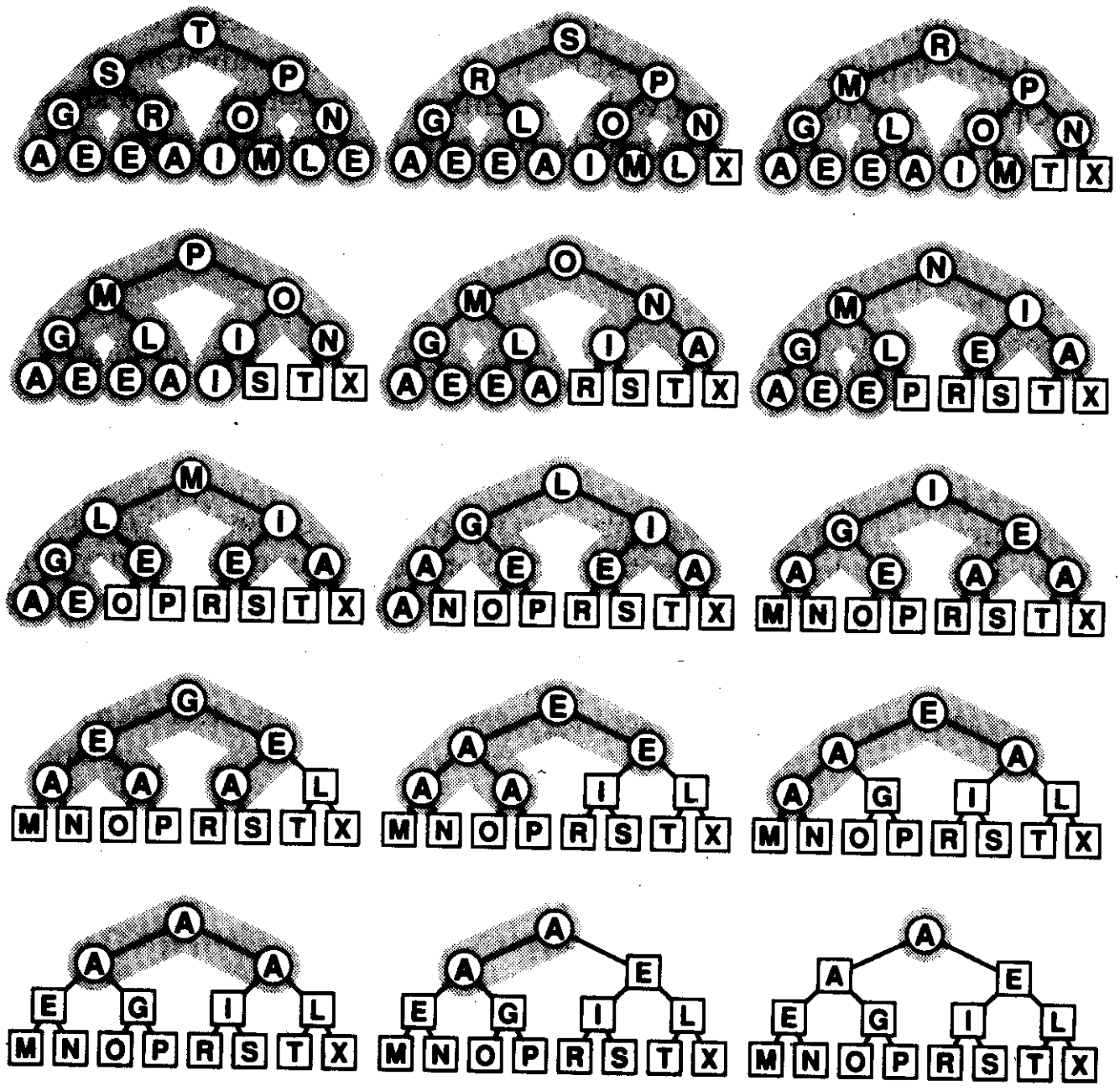
Heapsort, Beispiel

Aufbau eines Heap



Heapsort, Beispiel

Sortieren aus einem Heap



Heapsort, Kommentare

- Heapsort verwendet keinen zusätzlichen Speicherplatz.

- Heapsort garantiert, daß N Elemente in O(N log N) Schritten sortiert werden, unabhängig von den Eingabedaten:

  $C_{max}(N) = O(N \log N)$

  $M_{max}(N) = O(N \log N)$

- Der Aufbau des Heap benötigt ebenso O(N log N) Schritte.

Heapsort, Verbesserung von R.W.Floyd

Idee:

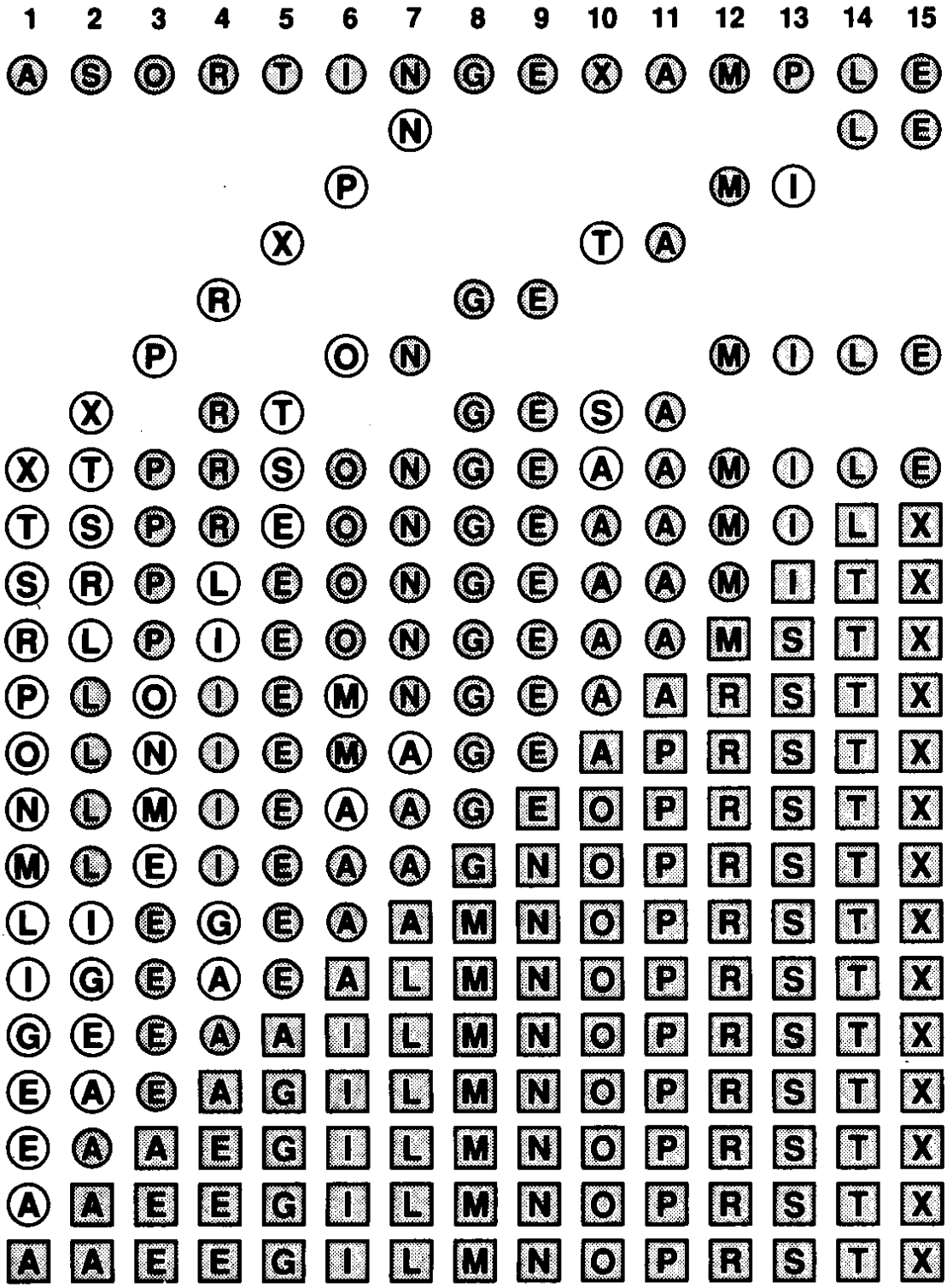
Die bisherige Methode zur Konstruktion eines Heap ist eine *Top-Down-Methode* durch wiederholtes `insert`.
Eine Verbesserung bringt der Einsatz einer *Bottom-up-Methode*:
Es wird jede Position im Feld ausgehend von der zweituntersten Ebene als Wurzel eines kleinen Heaps betrachtet und `downheap` für diesen kleinen Heap durchgeführt.

Programm

```
heapsort (int a[], int N)
  {
    int k, t;
    for (k = N/2; k >= 1; k--) downheap(a, N, k);
    while (N > 1)
      {
        t = a[1]; a[1] = a[N]; a[N] = t;
        downheap(a, --N, 1);
      }
  }
```

Heapsort, Verbesserung

Beispiel



Heapsort, Verbesserung

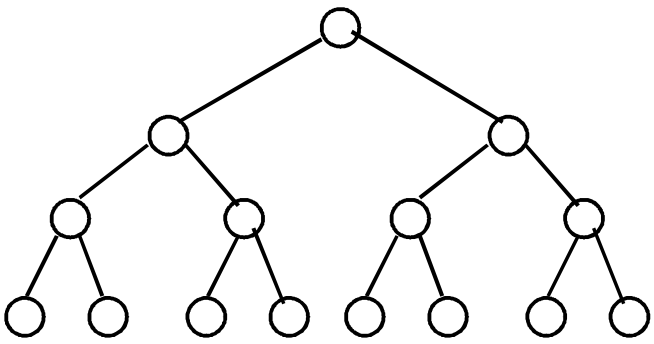
Aufwand

Behauptung:

Der Bottom-up-Aufbau eines Heaps erfolgt in O(N).

Beweis:

Sei $2^{j-1} < N \leq 2^j - 1$, also j die Anzahl der Ebenen des Heaps für N Schlüssel.



Auf Ebene k gibt es maximal 2^{k-1} Schlüssel.

Die Anzahl der Bewege- und Vergleichsoperationen in `downheap` für ein Element der Stufe k ist proportional zu j-k.

Damit gilt für die Gesamtzahl der Operationen zum Bottom-up-Aufbau eines Heaps:

$$\sum_{k=1}^{j-1} 2^{k-1}(j-k) \;=\; \sum_{k=1}^{j-1} k*2^{j-k-1} = 2^{j-1} * \sum_{k=1}^{j-1} \frac{k}{2^k} \leq N*2 \;=\; O(N)$$

qed.

Heapsort, Verbesserung

Kommentare

- Die Zeitschranke für Heapsort wird nur vom Sortieren selbst bestimmt, nicht vom Aufbau des Heap.

- Es gilt:

  $C_{max}(N) = C_{avg}(N) = O(N \log N)$

  $M_{max}(N) = M_{avg}(N) = O(N \log N)$

  damit ist Heapsort der erste uns bekannte Sortieralgorithmus, der asymptotisch optimale Laufzeit auch im schlechtesten Fall hat.
  Es gibt für Heapsort also keinen worst-case.

- Quicksort ist im Durchschnitt schneller als Heapsort, benötigt jedoch zusätzlichen Speicherplatz im Gegensatz zu Heapsort