Programmiertechnik 2

Unit 9: Optimierte Sortieralgorithmen

Ablauf

- Quicksort
  - Grundidee, Partitionierung, Rekursion
  - Analyse von Quicksort
  - Optimierung für kleine Mengen
- Mergesort
  - Mischen von Stapeln
  - Mischen im selben Speicher
  - Top-Down-Mergesort, Bottom-Up-Mergesort
- Heapsort
  - Priority Queues
  - Heap-Eigenschaft
- Radixsort
  - Sortieren anhand spezieller Eigenschaften der Schlüssel

Sir Charles Antony Richard Hoare

- Geboren 11. 1. 1934 in Colombo (Sri Lanka)
- Studium in Oxford (Philosophie, Latein, Griechisch) und Moskau (Übersetzung natürlicher Sprache)
- 1960 erste kommerzielle Implementierung von Algol-60
- Professor in Belfast seit 1968, in Oxford seit 1977
- 2000 geadelt "for services to computer science"
- Erfindungen:
  - Quicksort
  - Hoare logic
  - CSP (Communicating Sequential Processes)
- Zitate
  - "Premature optimization is the root of all evil"
  - "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

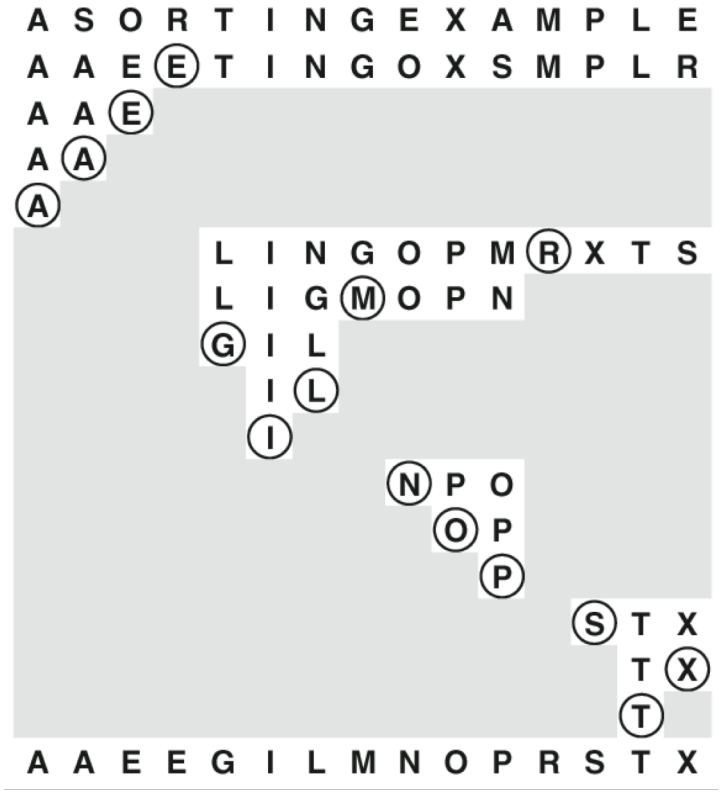
Quicksort

- C.A.R. Hoare 1960
- Meist-implementierter und meist-studierter Algorithmus
- In-place
  - Aber: Speicherbedarf für Rekursion
- Komplexität im Mittel O(N log N)
- Komplexität im schlechtesten Fall $O(N^2)$

Quicksort: Grundidee

- Teile-und-herrsche-Algorithmus
- Partitionierung:
  - Zerlege Menge in zwei Teile: Kleiner als Pivot-Element, größer als Pivot-Element
  - Auswahl des Pivotelements: Willkürlich, z.B. letztes Element
  - Pivot-Element wird an seine endgültige Position geschrieben
- Sortierung erfolgt rekursiv:
  - Partitionierung
  - Rekursives Sortieren der kleineren Elemente
  - Rekursives Sortieren der größeren Elemente

Quicksort Beispiel



Quicksort ist ein rekursiver Zerlegungsprozess: Wir teilen eine Datei, indem wir irgendein Element (das Trennelement) an seine Position bringen und dann das Array neu anordnen, sodass kleinere Elemente links vom Trennelement und größere Elemente rechts davon stehen. Dann sortieren wir den linken und den rechten Teil rekursiv. Jede Zeile in diesem Diagramm stellt das Ergebnis der Zerlegung der angezeigten Teildatei nach dem eingekreisten Element dar. Das Endergebnis ist eine vollständig sortierte Datei.

Quicksort: Rekursion

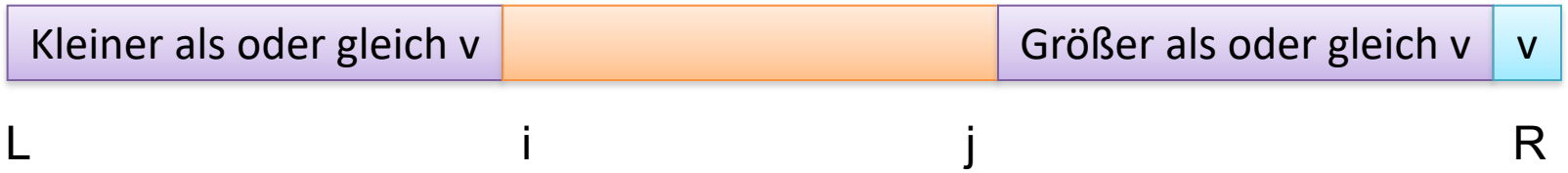
```
static void quicksort(ITEM[] a, int L, int R)
{
    if (R <= L)return;
    int i = partition(a, L, R);
    quicksort(a, L, i-1);
    quicksort(a, i+1, R);
}
```

QuickSort.java

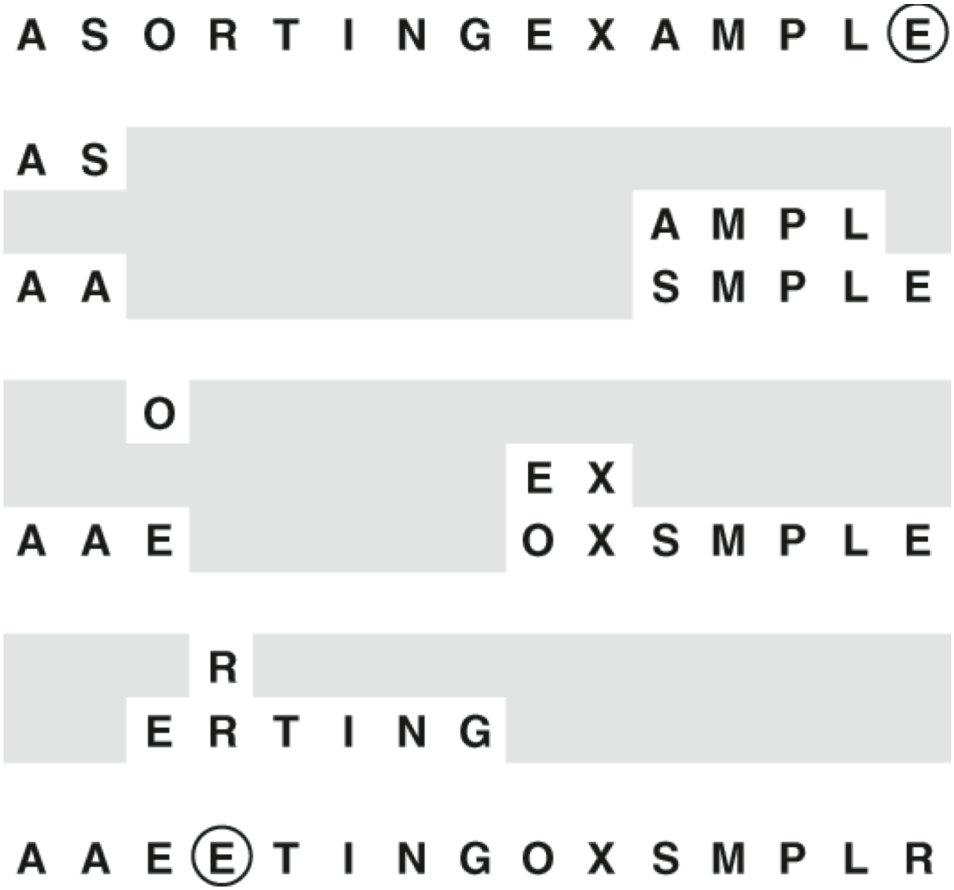
Quicksort: Partitionierung

1. Wähle a[R] als Pivot-Element v
2. Suche von links (in i) Element, das größer als v
3. Suche von rechts (in j) Element, das kleiner als v
4. Vertausche a[i] und a[j]
5. wiederhole 2-4, bis i >= j
6. Vertausche Pivot-Element in Grenze zwischen kleinen und großen Elementen (vertausche a[i] mit a[R])

→ Nach diesem Schritt steht Pivot-Element an endgültiger Position und wir können nun rekursiv linke und rechte Teilmenge sortieren.



Partitionierung



Das Zerlegen im Quicksort-Algorithmus beginnt mit der (willkürlichen) Wahl eines Trennelements. Programm 7.2 nimmt das am weitesten rechts stehende Element E. *Dann sucht es von der linken Seite aus über kleinere Elemente hinweg und von der rechten Seite aus über größere Elemente hinweg, vertauscht die Elemente, die die Suchvorgänge gestoppt haben, und fährt fort, bis sich die Suchindizes treffen. Als Erstes suchen wir von links und stoppen am* S, *dann suchen wir von rechts und stoppen am* A , *daraufhin vertauschen wir* S *und* A. *Als Nächstes fahren wir mit der Suche von links fort, stoppen am* O, *setzen die Suche von rechts fort, stoppen am* E *und vertauschen dann das* O *und das* E. *Als Nächstes kreuzen sich unsere Suchindizes: Wir fahren mit der Suche von links fort, bis wir am* R *halten, und fahren mit der Suche von rechts fort (nach dem* R*), bis wir zum* E *kommen. Um den Prozess abzuschließen, vertauschen wir das Trennelement (das* E *auf der rechten Seite) mit dem* R.

Quicksort: Partitionierung (2)

```
static int partition(ITEM a[], int L, int R)
{
    int i = L - 1, j = R;
    ITEM v = a[R];
    for(;;) {
        while (less(a[++i], v));
        while (less(v, a[--j]))
            if (j == L) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, i, R);
    return i;
}
```

- Zerlegungsschleife ist als Endlosschleife implementiert
- break sorgt für Verlassen der Schleife wenn sich Indizes kreuzen
- Test j == L gewährleistet Verlassen der Schleife wenn Trennelement das kleinste Element der Datei ist

Analyse von Quicksort

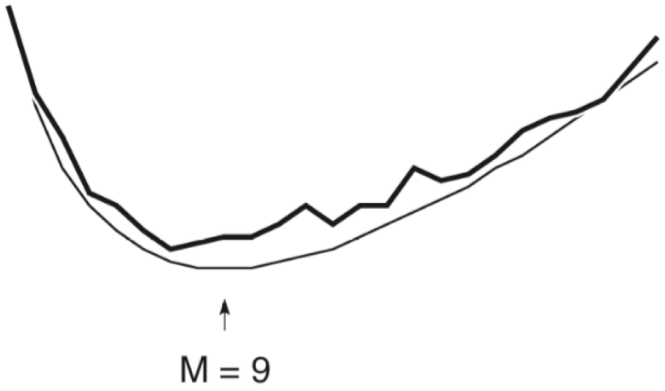
- Laufzeit hängt von Wahl des Pivot-Elements ab
  - Quicksort ist nicht stabil (jedes Element kann sich bei Partitionierung über gleiche Elemente hinwegbewegen)
  - Kein einfaches Verfahren bekannt, um Quicksort auf Arrays stabil zu machen
- Bester Fall: Pivot-Element teilt Menge in zwei gleiche Teile
  - Partitionierung braucht R-L Vergleiche
  - Rekursionstiefe lg N
  - Gesamtlaufzeit etwa N lg N Vergleiche
- Im Mittel: 2N lg N
- Schlechtester Fall: Menge ist bereits sortiert
  - Pivot-Element ist stets größtes Element
  - Rekursionstiefe ist N
  - Gesamtlaufzeit etwa $N^2/2$ Vergleiche

Optimierung: Kleine Mengen

- Rekursion und Partitionierung meist langsam für kleine Elementsammlungen
  - Im entarteten Fall einer sortierten Datei ruft sich quicksort für jedes Element einmal auf und bewegt genau ein Element
- Ausweg: insertion sort
  - if (R-L <= M) insertion(a, L, R);
  - Wahl von M empirisch, etwa 5 bis 20
- Alternativ: Abbruch, wenn R-L<=M
  - Sortieren des von Quicksort vorsortierten Ergebnisses mit Insertionsort am Ende (über gesamtes Array)

Cutoff für kleine Teildateien

- Insertionsort für Teilmengen mit weniger als M Elementen
- Laufzeitverbesserungen bis 10% gegenüber naiver Auswahl M = 1



Wählt man einen optimalen Cutoff für kleine Teildateien, lässt sich im Durchschnitt eine Verbesserung um etwa 10 Prozent erreichen. Die genaue Wahl des Cutoffs ist nicht kritisch; für die meisten Implementierungen eignen sich Werte in einem breiten Bereich (von ungefähr 5 bis ungefähr 20). Die dicke Linie (oben) wurde empirisch ermittelt; die dünne Linie (unten) wurde analytisch abgeleitet.

QuickInsSort.java

Optimierung: Median-of-Three-Partitionierung

- Problem: Quicksort ist ineffizient wenn Pivot größtes oder kleinstes Element
- Heuristik: Median-of-Three
  - Betrachte a[L], a[R], a[(L+R)/2], wähle mittleres dieser Elemente als Pivot-Element

```
exch(a, ((L+R)/2, R-1);
compExch(a, L, R-1);
compExch(a, L, R);
compExch(a, R-1, R);
int i = partition(a, L, R-1);
```

Mergesort

- Sortieren durch Mischen
- Mischen (merge): zwei sortierte Folgen werden in eine Folge integriert
- Algorithmus vergleicht kleinste Elemente beider Folgen, wählt kleineres der beiden für Ergebnisfolge
  - ausgewähltes Element wird aus Eingabe entfernt
- evtl. ist eine der beiden Folgen zuerst erschöpft
- zusätzlicher Speicherbedarf: Elemente werden beim Mischen in Zielcontainer kopiert
  - Speicherbedarf O(n)
  - Mischen ohne zusätzlichen Speicher: möglich, kompliziert und komplex

Mischen

```
static void mergeAB(ITEM[] c, int cL,
                    ITEM[] a, int aL, int aR,
                    ITEM[] b, int bL, int bR)
{
    int i = aL, j = bL;
    for (int k = cL; k <= cL+aR-aL+bR-bL+1; k++) {
        if (i > aR) { c[k] = b[j++]; continue; }
        if (j > bR) { c[k] = a[i++]; continue; }
        c[k] = less(b[j], a[i]) ? b[j++] : a[i++];
    }
},
```

Sedgewicks Version kopiert ein Element zu wenig

Formulierung bei Sedgewick ist nicht stabil

Mischen im selben Speicher

- Abstrakte Operation
  - static void merge(ITEM[] a, int L, int M, int R);
  - beide Eingabefolgen stehen hintereinander in a (von L..M, M+1..R)
  - Ergebnis steht in a (von L..R)
- Idee: Kopieren aller Elemente in temporären Speicher
- Mischen aus temporären Speicher in Originalspeicher
- stabiles Mischen: gleiche Elemente behalten ihre relative Position (auch wenn sie aus verschiedenen Teilfolgen stammen)
- Zahl der Kopieroperationen: 2 * (R-L)
- Zahl der Vergleiche
  - min (M-L, R-M) im besten Fall
  - R-L im schlechtesten Fall

MergeSort.java

Top-Down Mergesort

- Teile-und-herrsche-Algorithmus
  - Teile Eingabe in zwei gleiche Teile
  - Sortiere jeden Teil
  - Mische die Ergebnisse
- Abbruchkriterium: Zahl der Elemente in Eingabe <= 1

Top-Down Mergesort

```
static void mergesort(ITEM[] a, int L, int R)
{
    if (R <= L) return;
    int M = L+(R-L)/2;
    mergesort(a, L, M);
    mergesort(a, M+1, R);
    merge(a, L, M, R);
}
```

Analyse von Mergesort

- Benötigt ungefähr N lg N Vergleiche im schlechtesten Fall
- Benötigt zusätzlichen Speicher proportional zu N
- Stabil, falls merge-Operation stabil

Verbesserung: Mischen ohne Kopieren

- Idee: Temporärer Speicher und Eingabespeicher vertauschen pro Rekursionsschritt ihre Rollen
- Problem: bei naiver Implementierung steht das Ergebnis am Ende im falschen Speicher
  - Lösung: Eingabe wird in beide Speicher kopiert
  - Garantie des richtigen Ausgabespeichers durch Konstruktion des Algorithmus

Verbesserung: Mischen ohne Kopieren (2)

```
static void mergesortABr(ITEM[] a, ITEM[]b, int L, int R)
{   if (R <= L) return;
    int M = L+(R-L)/2;
    mergesortABr(b, a, L, M);
    mergesortABr(b, a, M+1, R);
    mergeAB(a, L, b, L, M, b, M+1, R);
}
static void mergesort(ITEM[]a, int L, int R)
{   ITEM[] aux = new ITEM[a.length];
    for (int i=L; i <= R; i++) aux[i] = a[i];
    mergesortABr(a, aux, L, R);
}
```

Bottom-up Mergesort

- nicht-rekursiv
- Folge wird erst in Zweiergruppen sortiert, diese dann in Vierergruppen gemischt usw.
- Algorithmus basiert weiter auf merge()
- Partitionierung der Eingabe ist anders als bei Top-Down-Algorithmus, falls Feldlänge keine Zweierpotenz

Bottom-up Mergesort

```
static int min(int A, int B)
{
    return (A<B) ? A : B;
}

static void mergesort(ITEM[] a, int L, int R)
{
    if (R <= L) return;
    for (int M = 1; M <= R-L; M = M+M)
          for (int i = L; i <= R-M; i += M+M)
              merge(a, i, i+M-1, min(i+M+M-1, R));
}
```

Priority Queue

- Abstrakter Datentyp
- Inhalt: Elemente mit Priorität
- Operationen:
  - Einfügen: Angabe des Elements und seiner Priorität
    - Operation liefert keinen Ergebniswert
  - Entfernen: Parameterlos
    - Operation liefert das Element mit der höchsten Priorität

- Grundlegende Datenstruktur für Implementierung von Heapsort

Priority Queue: Implementierungsvarianten

- Speicherung in Liste/Ringpuffer
- Konstante Zeit entweder für Einfügen oder Entfernen
- Einfügen in konstanter Zeit:
  - Element wird am Ende der Queue angehängt: O(1)
  - Entfernen durchsucht Liste nach Element mit höchster Priorität: O(n)
- Entfernen in konstanter Zeit:
  - Elemente sind stets nach Priorität sortiert
  - Einfügen durchsucht Liste nach richtiger Position: O(n)
  - Entfernen entfernt vorderstes Element
- Effiziente Implementierungen
  - Heap (Einfügen und Entfernen in O(lg N))
  - sortierter Baum (Komplexität hängt von Balanzierungsalgorithmus ab)

Heaps

- Ein Baum heißt “heap-geordnet”, wenn der Schlüssel in jedem Knoten größer-oder-gleich den Schlüsseln in allen Kindknoten ist
  - Eigenschaft läßt sich für beliebige Bäume definieren (nicht notwendig binär)
  - Ordnung zwischen den Kindern eines Knotens ist nicht vorgeschrieben
- In einem heap-sortierten Baum ist kein Schlüssel größer als der in der Wurzel
- Ein Heap ist eine Menge von Schlüsseln in einem heap-geordneten (fast) vollständigen Binärbaum

  ...der in einem Array dargestellt ist

  - Kindknoten von Knoten i stehen an Position 2i und 2i+1
    - Annahme: Nummerierung beginnt bei 1
  - letzte Ebene des Baums wird “von links” aufgefüllt

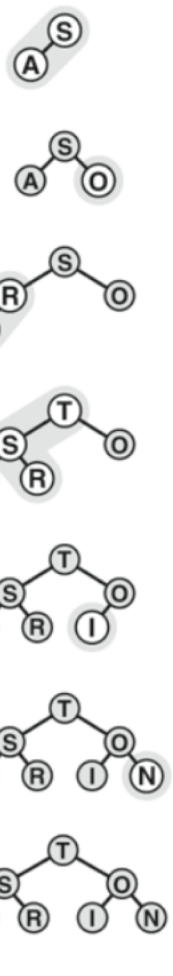
Erzeugung von Heaps

- Operation: Anhängen eines neuen Elements
  - auch: Erhöhen der Priorität eines Elements im Heap
  - Neues Element wird am Ende des Heaps eingefügt
  - Heap-Eigenschaft u.U. verletzt, wenn neuer Knoten größer als Elternknoten
- Wiederherstellen der Eigenschaft: Vertauschen des Knotens mit seinem Elternknoten (swim)
  - Elternknoten von Position N ist an Position N/2
  - Knoten ist nun größer als seine beiden Kindknoten
    - per Definition größer als sein alter Elternknoten
    - auch größer als der andere Kindknoten, weil dieser kleiner als der alte Elternknoten
  - Operation muss u.U. wiederholt werden, evtl. bis neuer Knoten zum Wurzelknoten wird

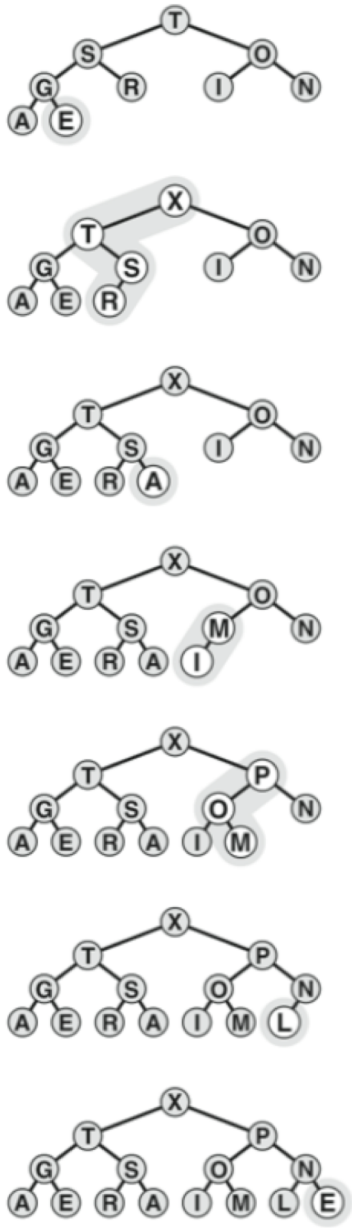
Erzeugen von Heaps (2)

- Operation: Senken der Priorität eines Elements
  - auch: Entfernen der Wurzel, Austausch z.B. mit letztem Element
- Wiederherstellen der Heap-Eigenschaft: Vertauschen des Elements mit dem größeren seiner Kinder (sink)
  - eventuell wiederholt, bis Element Blatt im Baum ist
- Basis der Algorithmen: less(index1, index2), exch(index1, index2)

Erzeugen von Heaps (3)



Diese Sequenz zeigt das Einfügen der Schlüssel A S O R T I N G *in einen anfänglich leeren Heap. Neue Elemente werden dem Heap am unteren Ende hinzugefügt und wandern auf der untersten Ebene von links nach rechts. Jedes Einfügen beeinflusst nur die Knoten auf dem Pfad zwischen dem Einfügungspunkt und der Wurzel, sodass sich die Kosten im ungünstigsten Fall proportional zum Logarithmus der Größe des Heaps verhalten.*



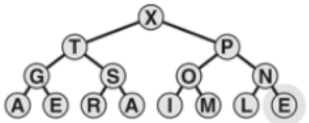
Diese Sequenz zeigt das Einfügen der Schlüssel E X A M P L E *in den Heap, dessen anfängliche Konstruktion in Abbildung 9.5 zu sehen ist. Die Gesamtkosten für die Konstruktion eines Heaps der Größe N sind kleiner als* lg *1* + lg *2* + … lg *N, was auch kleiner ist als N* lg *N.*

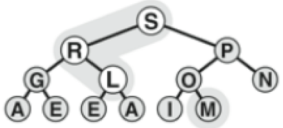
Erzeugung von Heaps: swim

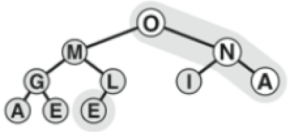
```
void swim(int k)
{
    while (k > 1 && less (k/2, k))
    {
     exch(k, k/2); k = k/2;
    }
}
```

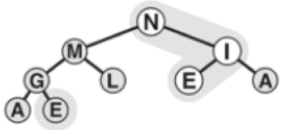
- Einfügen eines neuen Elements verletzt u.U. Heap-Eigenschaft
- Aufruf von swim() stellt Heap-Eigenschaft durch Vertauschen von Elementen im Heap wieder her

Entfernen des größten Elements

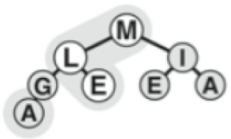


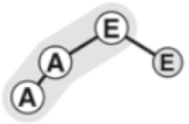






Nachdem das größte Element im Heap durch das äußerst rechte Element in der untersten Ebene ersetzt worden ist, können wir die Heap-Ordnung wiederherstellen, indem wir den Heap ausgehend von der Wurzel nach unten hin durchkämmen.





Diese Sequenz zeigt das Entfernen der restlichen Schlüssel aus dem Heap gemäß Abbildung 9.7. Selbst wenn jedes Element den gesamten Weg nach unten zurücklegen muss, betragen die Kosten für die Sortierphase weniger als ld *N* + … + ld *2* + ld *1, was wiederum weniger als N* log *N ist.*

Entfernen von Elementen aus Heap: sink

```
void sink(int k, int N)
{
    while (2*k <= N)
    {
     int j = 2*k;
     if (j < N && less(j, j+1)) j++;
     if (!less(k, j)) break;
     exch(k, j); k = j;
    }
}
```

- Entfernen des größten Elements verletzt u.U. Heap-Eigenschaft
- Aufruf von sink() stellt sie wieder her

Priority-Queue auf Heap-Basis

- Initialzustand: Feld ist leer
- Einfügen
  - Anfügen des neuen Elements ans Ende, N++
  - swim(N)
  - Maximal lg N Vergleiche
- Entfernen des größten Elements
  - Tauschen des ersten und des letzten Elements
  - N--
  - sink(1, N)
  - Ergebnis: Element N+1
  - Maximal 2*lg N Vergleiche

Heapsort

1. Heap-Aufbau von unten nach oben
   - Aufbau von Teilheaps, beginnend bei N/2 ... 1
   - Integration mehrerer Teilheaps in größere
2. Sortieren durch wiederholtes Entfernen des größten Elements
   - jeweils größtes Element wird an jeweils letzte Position des Felds vertauscht
   - danach Wiederherstellen der Heap-Eigenschaft

HeapSort.java

Heapsort (2)

```
for (int k = N/2; k >= 1; k--)
    sink(k, N);
    while (N > 1) {
        exch(1, N);
        sink(1, --N);
    }
```

Analyse von Heapsort

- Phase 1 (Aufbau): O(N)
  - N/4 Heaps der Tiefe 1: N/2 Vergleiche (2 pro sink)
  - N/8 Heaps der Tiefe 2: 2*N/4 Vergleiche
  - N/16 Heaps der Tiefe 3: 3*N/8 Vergleiche
  - N/32 Heaps der Tiefe 4: 4*N/16 Vergleiche
  - ...
  - insgesamt weniger als 2*N Vergleiche
- Phase 2 (Sortieren): O(N lg N)
  - N Elemente, pro Elemente einmal sink, jeweils weniger als lg N Vergleiche
- Gesamtlaufzeit: O(N lg N)
  - Weniger als 2 N lg N Vergleiche
- Speicherverbrauch: O(1)
- nicht stabil
- Worst-case-Laufzeit besser als Quicksort, Speicherverbrauch besser als Mergesort
  - aber: höhere Laufzeit für Zufallsdaten

Spezialisierte Sortierverfahren

- Annahmen über die Struktur der Schlüssel:
  - Schlüssel selber ist strukturierte Folge von “Ziffern”
  - Beispiel: Zahlen als Schlüssel (Folge von Bits)
  - Beispiel: Strings als Schlüssel (Folge von Zeichen)
  - Sortierung erfolgt “lexikographisch”
  - Elemente des Schlüssels stammen aus “kleiner” Menge (Bits: zwei Werte; Zeichen: z.B. 256 Werte)
    - Basis des Positionssystems sei **R**
- Sortierverfahren mit “stückweiser” Schlüsselverarbeitung: *radix sort*

Radixsort

- Abstrakte Schlüsseloperation: Liefere n-te Ziffer des Schlüssels
- MSD-Sortierung (most-significant digit):
  - Sortiere erst nach vorderster Ziffern, dann nach zweiter Ziffer, usw.
  - Quicksort-Verallgemeinerung: Partitionierung in viele Teilmengen
- LSD-Sortierung (least-significant digit):
  - Sortierung erst nach minderwertigster Ziffer, dann nach vorletzter Ziffer, usw.
  - unintuitiv: Wie hilft die Sortierung nach der letzten Stelle zur Lösung des Gesamtproblems?

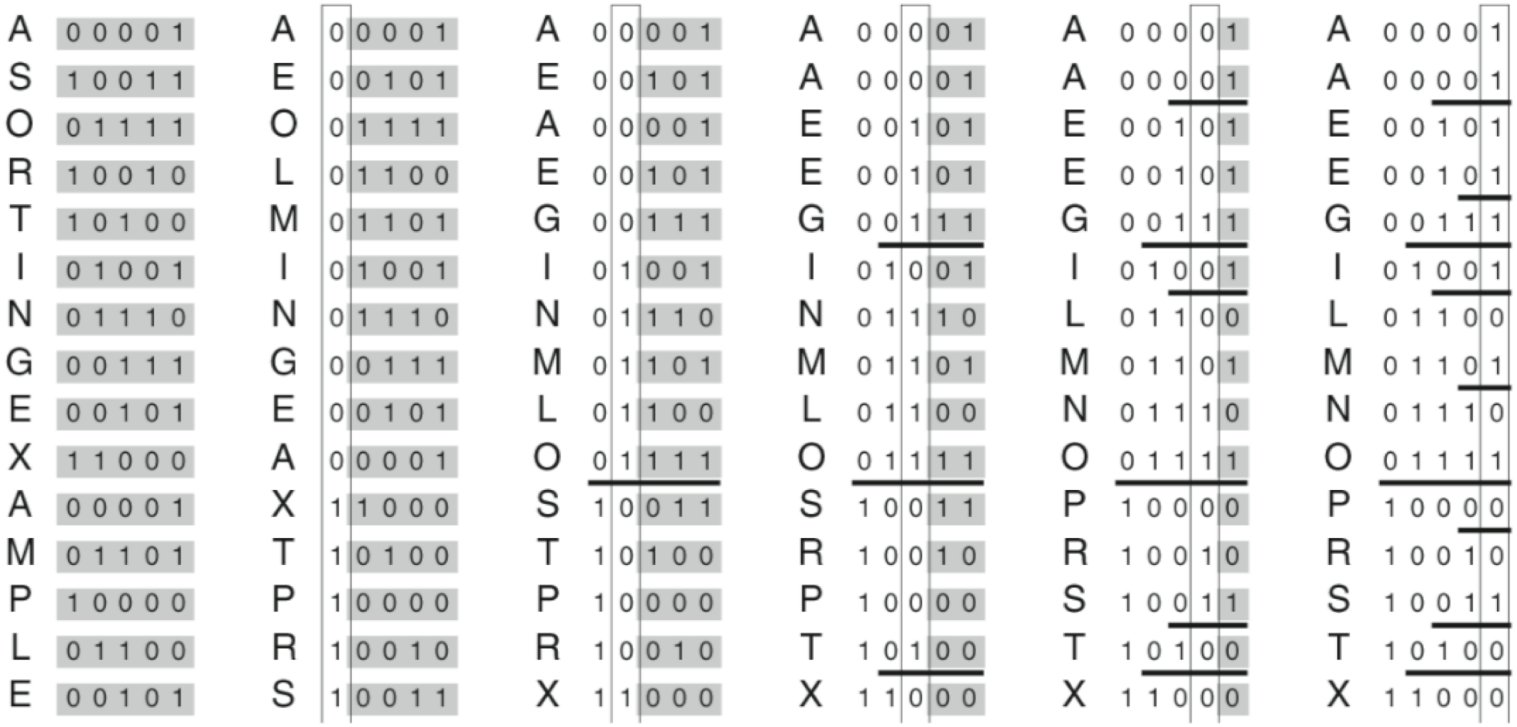
MSD-Sortierung

- Aufteilung der Gesamtmenge in Teilmengen nach der ersten Ziffer:
  - Array von **R** Körben (*bins, buckets*)
  - Elemente werden in passenden Korb eingefügt

- Rekursive Aufteilung jedes Korbs nach zweiter, dritter, usw. Stelle
- Übergang zu alternativem Algorithmus, wenn Zahl der verbleibenden Elemente klein ist (etwa: kleiner als **R**)

- Bitweise Betrachtung → binärer Quicksort

Radix-Exchange-Sort
(binärer Quicksort)



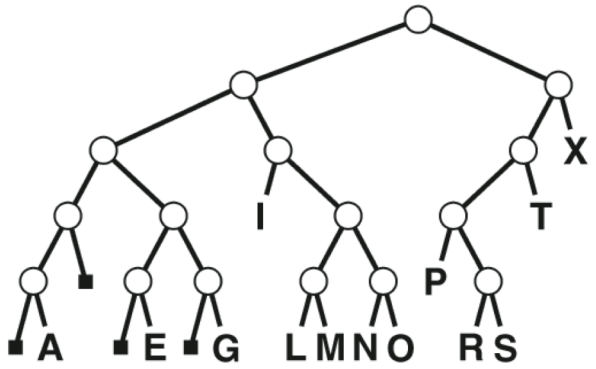
- Erste Stufe erzeugt 2 Teildateien, die mit Bit 0 und Bit 1 beginnen
- Zweite Stufe erzeugt 4 Teildateien, die mit Bits 00, 01, 10 und 11 beginnen
- Dritte Stufe: 000, 001, 010, 011, 100,101, 110, 111 (leer)...
- Verfahren bricht ab wenn Bits aufgebraucht sind oder Teildateien Größe 1 haben

BinQuickSort.java

Binärer Quicksort - Beispiel

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

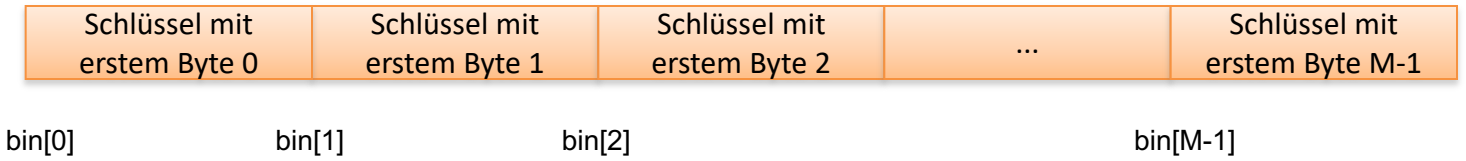
Die Zerlegung bezogen auf die führenden Bits garantiert nicht, dass ein Wert an seine Position kommt; es ist nur sichergestellt, dass alle Schlüssel mit führenden 0-Bits vor allen Schlüsseln mit führenden 1-Bits stehen.



- Jeder Knoten im Baum repräsentiert 1 relevantes Schlüsselbit
- A = 00001..., P = 1001...
- ■ stellt leeren „bin“ dar (Nullknoten)
- Doppelte Schlüssel lassen sich nicht zerlegen

MSD -Radixsort

- Betrachten nun byteweise Zerlegung der Schlüssel
- Algorithmen verwenden eine Gruppe von R Fächern (bins, buckets)



- Durchlaufen die Schlüssel, verteilen sie auf Fächer
- Sortieren dann Fachinhalte bezüglich der um 1 Byte verkürzten Schlüssel

now	ace	ace	ace
for	ago	ago	ago
tip	and	and	and
ilk	bet	bet	bet
dim	cab	cab	cab
tag	caw	caw	caw
jot	cue	cue	cue
sob	dim	dim	dim
nob	dug	dug	dug
sky	egg	egg	egg
hut	for	few	fee
ace	fee	fee	few
bet	few	for	for
men	gig	gig	gig
egg	hut	hut	hut
few	ilk	ilk	ilk
jay	jam	jay	jam
owl	jay	jam	jay
joy	jot	jot	jot
rap	joy	joy	joy
gig	men	men	men
wee	now	now	nob
was	nob	nob	now
cab	owl	owl	owl
wad	rap	rap	rap
caw	sob	sky	sky
cue	sky	sob	sob
fee	tip	tag	tag
tap	tag	tap	tap
ago	tap	tar	tar
tar	tar	tip	tip
jam	wee	wad	wad
dug	was	was	was
and	wad	wee	wee

MSD-Radixsort Beispiel

- Teilen Worte nach erstem Buchstaben in 26 Fächer ein
- Sortieren danach jedes Fach nach zweitem Buchstaben
- Etc.

Problem:

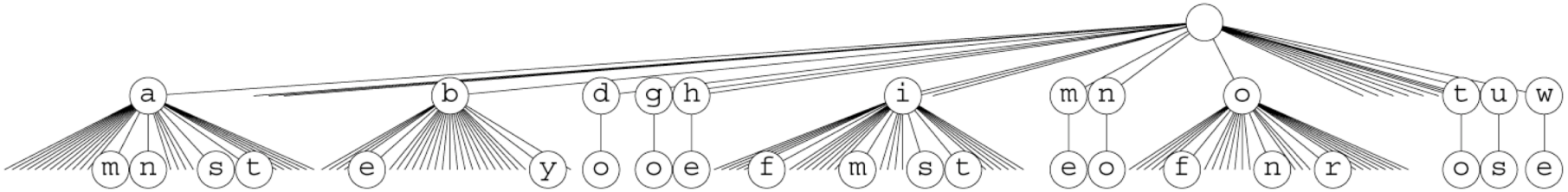
- Schon nach zweiter Stufe viele leere Fächer

Idee:

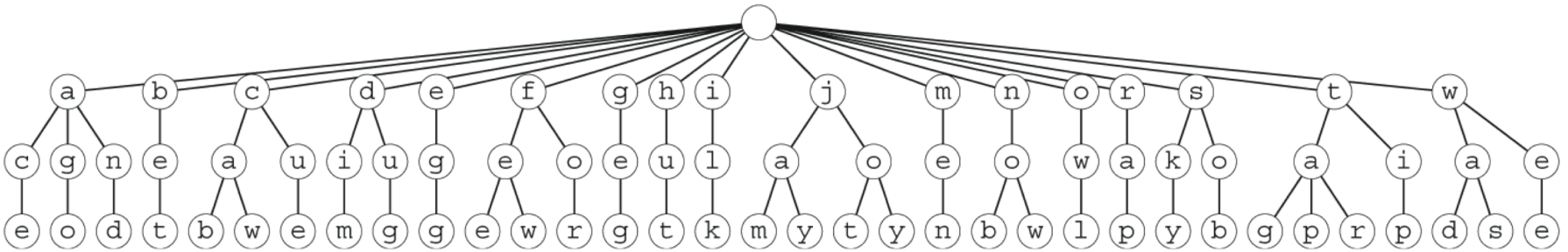
- Schlüsselbytes nicht linear von vorne nach hinten (MSD) betrachten
  - Folge von auszuwertenden Schlüsselbytes zufällig generieren
- Eliminieren von leeren Fächern (bins)

no	an	am
if	am	an
be	at	as
do	as	at
he	be	be
an	by	by
by	do	do
of	go	go
us	he	he
on	if	if
am	is	in
we	it	is
is	in	it
at	me	me
it	no	no
to	of	of
or	on	on
me	or	or
go	to	to
in	us	us
as	we	we

Eliminierung leerer Fächer



- Rekursive Struktur von MSD-Radixsort
  - Für Dateigröße 0 oder 1 keine rekursiven Aufrufe
  - Ansonsten 26 Aufrufe: für jeden möglichen Wert des aktuellen Bytes



- Kompakte Darstellung – Ignorieren leerer Teildateien
  - Knotenbeschriftung von Wurzel zu Blatt entsprich Sortierschlüssel

MSD-Sortierung: Speicherung der Körbe

- Idee: Ein Array für alle Körbe
  - Erst alle Elemente mit Ziffer 0, dann alle Elemente mit Ziffer 1 usw.
- Partitionierungsinformation: Ein Array für den Anfangsindex jeder Korbes
- Zwei Pässe:
  - Zählung der Elemente in gleicher Ziffer, um die Korbgrößen zu bestimmen (und damit die Anfangsindizes)
  - Aufteilung der Elemente in die Körbe: Array für die Speicherung der aktuellen Position in jedem Korb

RadixSort.java

LSD-Sortierung

- Sortierung zuerst nach letzter Ziffer (Position L), dann nach vorletzter, usw.
- historisches Verfahren, verwendet z.B. zur Sortierung von Lochkarten (Nummerierung der Karten in den letzten Spalten)
  - allgemein: praktikabel für Sortierung von ganzen Zahlen, nicht anwendbar auf Strings variabler Länge
- Verfahren sortiert nur, wenn jeder Sortierschritt **stabil** ist:
  - Beweis induktiv: Nach Sortierung von Position i ist die Menge bezüglich des Teilstrings i, i+, ..., L sortiert

Leistungsbewertung

- Vergleich mit anderen Sortierverfahren schwierig, weil Radixsort explizit Schlüssellänge berücksichtigt
  - bei anderen Algorithmen üblicherweise Annahme, dass Schlüsselvergleiche in konstanter Zeit stattfinden
- LSD: Sortierung von N Schlüsseln mit Länge w proportional zu Nw
  - asymptotisch im Mittel etwa N log N, weil Schlüssellänge von N Schlüsseln etwa log N ist
- MSD: schlechtester Fall: Alle Ziffern aller Schlüssel müssen berücksichtigt werden
  - z.B. falls alle Schlüssel gleich sind
- Sortierung oft *sublinear* bzgl. der Zahl der Schlüsselbits, weil nicht alle Bits aller Schlüssel untersucht werden müssen

Timsort
(modified mergesort)

- hybrider Sortieralgorithmus
  - Abgeleitet von Mergesort und Insertionsort
    - Gute Performance auf verschiedenen realen Daten
  - 2002 , Tim Peters, Implementierung in C vorgeschlagen
    - seit der Version 2.3 Standard-Sortieralgorithmus in Python
    - in Java SE 7 zum Sortieren von Arrays und auf Android-Plattform genutzt
- Performance:
  - Best-case: O(n); Worst-case / average-case: O(n log n)
- Idee:
  - Findet bereits sortierte Abschnitte in den Daten
  - Absteigend sortierte Abschnitte werden umgedreht
  - minimale Abschnittslänge für jeweilige Array-Größe überprüfen:
    - Array < 64 Elemente: Insertionsort ausführen
    - Größere Arrays: 32 <= minimale Abschnittslänge <= 64 wählen
    - Abschnitte mit Insertionsort erweitern
  - Sortierte Abschnitte mit Mergesort zusammenfügen

Aus OpenJDK:

TimSort.java

Zusammenfassung

- Quicksort
  - Grundidee, Partitionierung, Rekursion
  - Analyse von Quicksort
  - Optimierung für kleine Mengen
- Mergesort
  - Mischen von Stapeln
  - Mischen im selben Speicher
  - Top-Down-Mergesort, Bottom-Up-Mergesort
- Heapsort
  - Priority Queues
  - Heap-Eigenschaft
- Radixsort
  - Sortieren anhand spezieller Eigenschaften der Schlüssel