Zeitoptimale Sortierverfahren

Von Matthias Jauernig und Tobias Hammerschmidt

Inhaltsverzeichnis

1. Einleitung

[Zurück zum Inhaltsverzeichnis]

1.1 Grundgedanke

Auch wenn es sich beim Sortieren von Datensätzen um ein sehr "klassisches" Problemgebiet der Informatik handelt, ist es doch immernoch aktuell. Untersuchungen von Experten haben ergeben, dass schätzungsweise 1/4 der heute verbrauchten kommerziellen Rechenleistung für Sortierungen aufgewandt wird, was die Bedürfnisse nach schnellen und leistungsstarken Sortieralgorithmen verdeutlicht. Verwunderlich ist daher auch nicht, dass sich schon die Pioniere der Informatik mit dem Sortierproblem beschäftigt haben und gerade dort schnell Algorithmen entwickelt wurden, die zum Großteil bis zum heutigen Tage aktuell sind und Anwendung in nahezu allen größeren Programmen finden. Entsprechend groß ist die Fülle an Informationen über klassische Sortieralgorithmen; dass damit Bände von Büchern gefüllt werden ist keine Seltenheit. Und trotz der langjährigen Forschungen und wissenschaftlichen Abhandlungen zum Thema werden noch immer viele neue Erkenntnisse in wissenschafltichen Zeitschriften veröffentlicht und sind noch immer viele theoretische sowie praktische Probleme ungelöst.

1.2 Anliegen / Ziele

Ziel des folgenden Artikels soll *nicht* sein, die elementaren Sortierverfahren näher zu betrachten. Selection-, Insertion-, Bubble- und Shellsort habe ich hingegen in der C-Datei "elem_sort.c" erfasst, wo sie jeder im Selbststudium erforschen kann. Elementare Sortierverfahren haben im Allgemeinen eine Laufzeitkomplexität von $O(n^2)$ und sind für uns daher uninteressant (zur näheren Erläuterung der O-Notation bitte den entsprechenden Artikel heranziehen).
Mit dem vorliegenden Artikel wollen ich und Tobias vielmehr ohne Umschweife einen Einblick in die zeitoptimalen Sortierverfahren geben und diese anhand deutlicher Erklärungen, mit Code-Implementationen und Beispielen unterlegt, dem Leser vermitteln. Dass der obligatorische Einstieg weggelassen wurde, fordert allerdings einen kleinen Tribut: so ist der Artikel mehr für angehende Informatiker als für Neueinsteiger gedacht (man sollte zumindest die grundlegenden Algorithmen schon kennen), trotzdem soll er so verständlich wie möglich erscheinen - falls dennoch Fragen offen bleiben, bin ich gern bereit diese per Mail zu beantworten.
Ziel soll es also sein, einige der zeitoptimalen Sortierverfahren zu beschreiben. In den Kapiteln 2 und 3 werden Quicksort und Mergesort behandelt, im 4. Kapitel werde ich mit Tobias auf die Datenstruktur Heap und das darauf aufbauende Heapsort eingehen. Jedes dieser 3 Sortierverfahren stützt sich auf klassische Vergleiche der zu sortierenden Daten, wohingegen die Radixsort's die arithmetischen Eigenschaften der zu sortierenden Schlüssel ausnutzen - die hierzu gehörenden Verfahren werden in Kapitel 5 beschrieben, darunter auch ein Verfahren, welches linear und dazu noch sehr schnell arbeitet.

1.3 Spielregeln

Stillschweigend wird vorausgesetzt, dass bekannt ist, was "zeitoptimal" eigentlich bedeutet. Hierzu vielleicht doch noch ein paar Worte: zeitoptimale Sortierverfahren können eine Datei mit n Schlüsseln in einer Zeitkomplexität von n*log(n) sortieren - wenn es sich um Algorithmen wie Quicksort oder Mergesort handelt, die sich auf Schlüsselvergleiche stützen. Dass diese Komplexität nicht zu verbessern ist, kann leicht durch ein paar Überlegungen gezeigt werden: so kann man das Sortieren von n Schlüsseln als Binärbaum darstellen, dessen Wurzel die ursprüngliche Folge und dessen Blätter alle n! Permutationen dieser Folge sind. Denn jede dieser Permutationen kann als sortierte Folge in Frage kommen, doch nur ein Weg von der Wurzel zur Blattebene ist der gesuchte. Nun ist es aber so, dass n! Blätter eine Mindesthöhe des Baumes von n*log(n) erfordern - daher kann man nicht in weniger als O (n*log(n)) Schritten von einer zufälligen Anordnung zu einer sortierten Folge gelangen. Andere Verfahren wie z.B. das auch besprochene Radixsort greifen auf einzelne Ziffern der Schlüssel

zurück, bei ihnen kann man im Optimum von einer linearen Zeitkomplexität O(n) ausgehen, was wohl das Beste ist, worauf man bei einem Sortieralgorithmus hoffen kann.
Zu den Code-Beispielen ist es wichtig zu sagen, dass sie ausschließlich in C implementiert sind. Der Einfachheit halber und damit es noch ein wenig verständlicher wird, werden als *Schlüssel* dabei immer nur Ganzzahlen (*int*'s) verwendet. Es sollte allerdings kein Problem sein, diese Codes auf andere Schlüsseltypen zu übertragen.

Nun viel Spaß mit dem Artikel und nicht vergessen: immer schön sortiert bleiben =D

[Zurück zum Inhaltsverzeichnis]

2. Quicksort

[Zurück zum Inhaltsverzeichnis]

2.1 Historie / Hintergrund

Der grundlegende Quicksort-Algorithmus wurde 1960 von C.A.R. Hoare entwickelt und 1962 veröffentlicht (vgl. *Computer Journal*, 5:10-15, 1962) und war seit dem Thema von vielen wissenschaftlichen Untersuchungen. Quicksort ist der wohl am häufigsten angewandte Sortieralgorithmus, da seine Implementation nicht schwierig ist, es sich trotzdem sehr gut in vielen Situationen anwenden lässt und oftmals weniger Speicher verbraucht als die meisten anderen zeitoptimalen Sortierverfahren.
Die Leistungsfähigkeit ist sehr gut erforscht und mathematisch bewiesen, doch auch empirisch bestätigt worden, womit Quicksort zu einer bevorzugten Methode für ein großes Spektrum von Anwendungen wurde. Die Vorzüge des Algorithmus': in praxi wird nur durch die Rekursion programmintern ein kleiner Stapel als Hilfsspeicher benötigt, zudem ist die innere Schleife sehr kurz und im Durchschnitt werden nur n*log(n) Operationen zum Sortieren von n Schlüsseln benötigt. Dem stehen ein paar, aber mehr theoretische Aspekte gegenüber: so lässt sich nur schwer eine iterative Variante implementieren, außerdem ist der Algorithmus störanfällig gegenüber den kleinsten Programmierfehlern. Zudem werden im "worst case" $O(n^2)$ Operationen benötigt, was in der Praxis bei einer guten Implementierung allerdings kaum auftreten dürfte. Weiterhin ist Quicksort nicht stabil, d.h. die relative Sortiertheit der Datei bleibt nicht erhalten.
Im Laufe der Jahre erschienen immer wieder Verbesserungsvorschläge für Quicksort, meist war es jedoch so, dass Verbesserungen in einem Teil des Algorithmus' zu Verschlechterungen der Laufzeit in anderen Teilen führten. Am Ende dieses Artikels soll deshalb nur auf zwei Verbesserungen eingegangen werden, die sich in der Praxis aber gut bewährt haben: das Finden eines besseren Trennelements mit der Methode median-of-three sowie die Behandlung kleiner Teildateien mit insertion sort.

2.2 Der Algorithmus

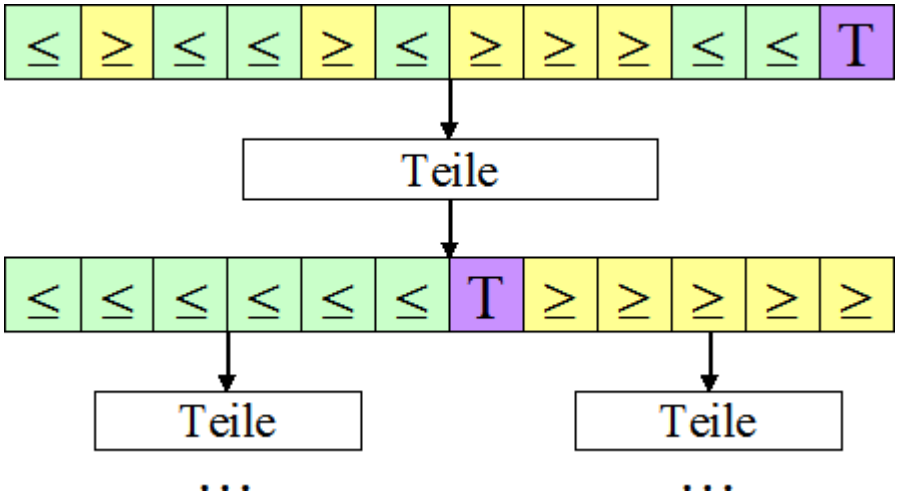
Quicksort ist ein "Teile-und-Herrsche"-Algorithmus ("divide&conquer"). Die Folge der zu sortierenden Daten wird dabei in 2 Teilfolgen getrennt ("Teile" die Datei), anschließend wird über diese beiden Teildateien unabhängig voneinander sortiert, indem diese auf dieselbe Art und Weise wie die Ausgangsfolge geteilt werden ("Herrsche" über die Teilfolgen). Diese Teilungen der Unterfolgen werden solange fortgesetzt, bis nur noch ein Element in der Teilfolge existiert - dann gilt die Gesamtfolge als sortiert.
Doch wie geschieht dieses "Teilen"? Hierin muss ja das Geheimnis stecken, wie am Ende die gesamte Folge sortiert sein kann.
Zunächst sucht man sich ein *Trennelement* (auch *Pivotelement*), welches idealerweise von der Wertigkeit her genau in der Mitte der Folge liegt (hierzu gleich noch mehr). Durch das Teilen soll nun Folgendes erreicht werden:

- Alle Elemente links vom Trennelement sollen kleiner oder gleich dem Trennelement sein.
- Alle Elemente rechts vom Trennelement sollen größer oder gleich diesem sein.
- Das Trennelement selbst soll an seine endgültige Stelle in der zu sortierenden Folge gelangen.

Die Auswahl eines geeigneten Trennelements stellt dabei schon eine Wissenschaft für sich dar. Der Einfachheit halber nimmt man beim grundlegenden Quicksort z.B. das letzte Element der Folge als Trennelement. Wie schon erwähnt wäre es laufzeitmäßig am besten, wenn das Trennelement von der Wertigkeit her in der Mitte der aktuellen (Teil-) Folge liegt. Dadurch würde die Folge in zwei gleichgroße Teilfolgen geteilt werden, wodurch sich die Rekursionstiefe des Algorithmus' gering halten lässt (im Prinzip würde man dadurch einen vollständigen Binärbaum erhalten, doch dazu in der Analyse mehr). Die Schwierigkeit ist nur, dieses mittlere Element zu finden. In einer unsortierten Folge müsste man jedes Element zumindest einmal betrachten, was eine Komplexität von O(n) zur Folge hat. Natürlich ist dies uneffektiv. Eine

Alternative ist es, 3 Elemente der Folge auszuwählen und das mittlere Element dieser 3 als Trennelement auszuwählen. Dies ist noch einfach realisierbar und verbessert die Laufzeit doch spürbar, als wenn man nur ein beliebiges Element auswählt, da die Wahrscheinlichkeit steigt, dass man ein Element der Folge findet, welches näher an der Mitte liegt. Dieses Spielchen heißt *median-of-three* und wird als Verbesserung weiter unten besprochen, man kann es aber auch weitertreiben: median-of-five, median-of-seven etc. *kann* man implementieren, wenn man will, allerdings muss man selber abschätzen, ob sich der dabei betriebene Aufwand in einer besseren Laufzeit widerspiegelt oder ob man sogar noch Zeiteinbußen hinnehmen muss. Deswegen soll *median-of-three* genügen und grundlegend das letzte Element als Trennelement benutzt werden, da es praktisch egal ist, welches Element der unsortieren Folge man auswählt - die Wahl könnte auch zufällig erfolgen...

Folgendes Bild soll die Absichten der "Teile"-Methode illustrieren:



Sehr schön zu sehen ist, dass Elemente, die kleiner oder gleich dem Trennelement sind (grün), durch das Teilen links von diesem eingeordnet werden, größer-gleich-Elemente (links) hingegen rechts - nun herrscht schon ein viel größeres Maß an Ordnung als vor dem Teilen. Anschließend erfolgt der rekursive Aufruf von "Teile" getrennt für die entstandene linke und rechte Teilfolge. Durch das Teilen wird die Folge also derart vorsortiert, dass das Trennelement seine endgültige Position in der Datei erhält und kein weiteres Mal betrachtet werden muss. Bei den beiden Teildateien kommt es nachfolgend durch ein weiteres Teilen wiederum zu einer Vorsortierung - dies geschieht rekursiv solange, bis eine Folge nur noch ein Element enthält - weiter kann nicht sortiert werden, muss auch nicht: denn durch die Vorsortierungen der größeren Folgen wurde die Folge an sich komplett sortiert.

2.3 Zur programmiertechnischen Realisierung

Das Grundprinzip ist bekannt: man nimmt das letzte Folgenelement als Trennelement und ordnet die Folge so um, dass alle Glieder, welche kleiner gleich dem Trennelement sind, nach links und größere Elemente nach rechts verschoben werden. Im Anschluss folgt ein rekursiver Aufruf jeweils getrennt für die linke sowie rechte Teilfolge. Die Lösung in einer Hochsprache sieht auch entsprechend logisch und einfach aus: man erstellt zwei Ganzzahlen *int i* und *int j*, welche Arrayindizes speichern sollen. Diese initialisiert man mit dem linken Rand der Folge für i und dem rechten Folgenrand für j. Nun wird mit diesen beiden "Zeigern" die Folge durchlaufen. i wird schrittweise inkrementiert, durchläuft die Folge also nach rechts. Dies geschieht solange ohne Aktion, bis ein Element gefunden wird, welches größer oder gleich dem Trennelement ist. An dieser Position bleibt i stehen und es wird damit begonnen, j zu dekrementieren - solange, bis ein Element gefunden wird, welches kleiner oder gleich dem Trennelement ist. Die Gleichheit wird bei beiden Tests mit eingeschlossen, da somit eine bessere Ausgeglichenheit der Zerlegung bei vielen identischen Schlüsseln erreicht werden kann, was mathematisch intensivst untersucht und gezeigt wurde (auch wenn dies ein paar Austauschoperationen mehr zu erfordern scheint). Wir haben nun ein zum Trennelement relativ "großes" Element im linken Folgenteil und ein "kleines" Element im rechten Folgenteil gefunden. Um die Vorsortierung zu gewährleisten, werden diese beiden Elemente miteinander vertauscht. Dann werden die beiden

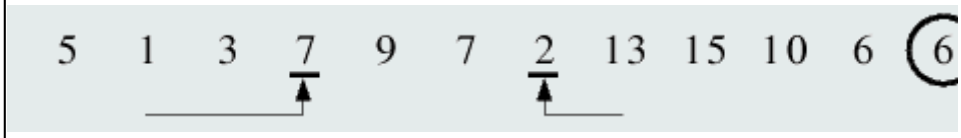
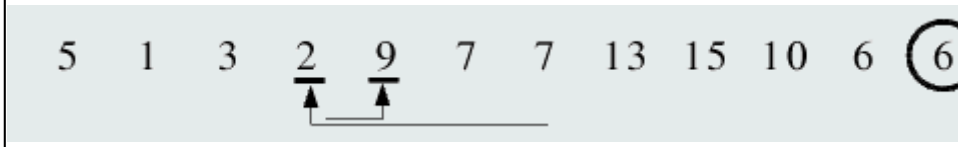
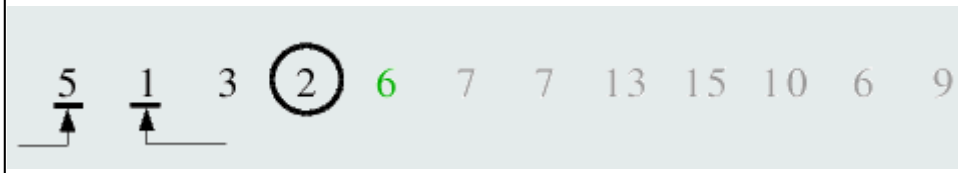
Folgenzeiger weiter in-/dekrementiert und das Tausch-Spielchen solange fortgesetzt, bis beide Zeiger aufeinander treffen. Ein Problem sollte man aber noch beachten: ist die Folge zu Beginn schon sortiert, so kann i nicht über den rechten Folgenrand hinauslaufen, da mindestens beim Trennelement gestoppt wird, vorausgesetzt man schließt die Gleichheit mit ein. Aber: j kann unterlaufen und auf das Element -1 zeigen, wenn die Folge zu Beginn in verkehrt sortierter Reihenfolge vorliegt. Da kein Element kleiner dem Trennelement gefunden werden kann, läuft j aus dem Feld hinaus, was man natürlich unterbinden muss. Zu diesem Zwecke muss man noch auf j>i prüfen, wenn man j dekrementiert. Zurück zum Algorithmus: zum Abschluss wird nun das Trennelement, welches ja immernoch an letzter Folgenposition steht, mit dem Element getauscht, welches am weitesten links in der rechten Teilfolge steht - dadurch erhält es seinen endgültigen Platz in der Folge und muss nicht noch einmal betrachtet werden. Anschließend folgen die rekursiven Aufrufe des Algorithmus' für die linke und rechte Teilfolge ohne dem Trennelement. Diese Rekursion geschieht solange, bis nur noch ein Element eine Teilfolge darstellt. Da bei jedem Rekursionsaufruf zumindest ein Element von der bisherigen Folge abgetrennt wird (auf jeden Fall das Trennelement), ist sicher gestellt, dass der Algorithmus terminiert.
Folgende quicksort()-Funktion ist eine mögliche Implementation des grundlegenden Algorithmus' und wird mit quicksort(a,0,n-1) zum Sortieren des Feldes a mit n Elementen aufgerufen:

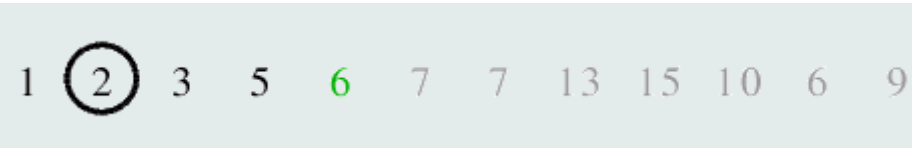
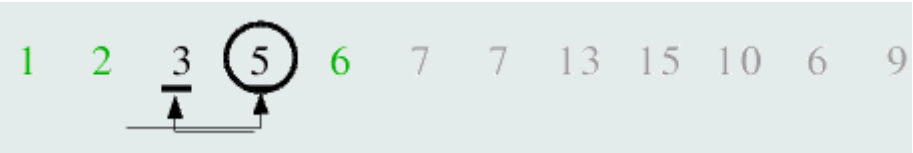
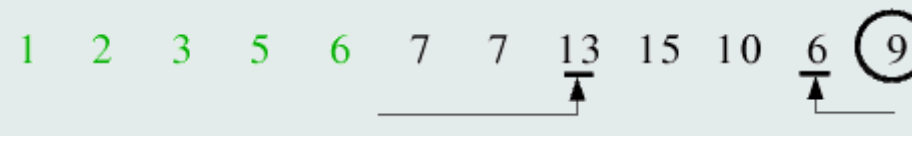
```
void quicksort(int a[], int l, int r){          //a=Array, l=linker Rand, r=recht
    if(r>l){                                    //solange mehr als 1 Folgenelemer
        int i=l-1, j=r, tmp;                    //Variableninitialisierung mit Fo
        for(;;){                                //Endlosschleife; bricht ab, wenr
            while(a[++i]<a[r]);                 //inkrem., bis größeres  Element
            while(a[--j]>a[r] && j>i);          //dekrem., bis kleineres Element
            if(i>=j) break;                     //brich ab, wenn sich die Folgenz
            tmp=a[i]; a[i]=a[j]; a[j]=tmp;      //tausche kleineres mit größerem
        }
        tmp=a[i]; a[i]=a[r]; a[r]=tmp;          //tausche Trennelement

        quicksort(a, l, i-1);                   //rekursiver Aufruf für linke Tei
        quicksort(a, i+1, r);                   //rekursiver Aufruf für rechte Te
    }
}
```

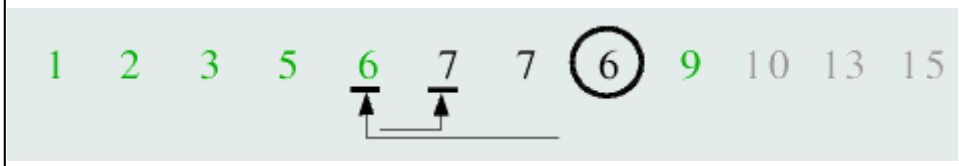
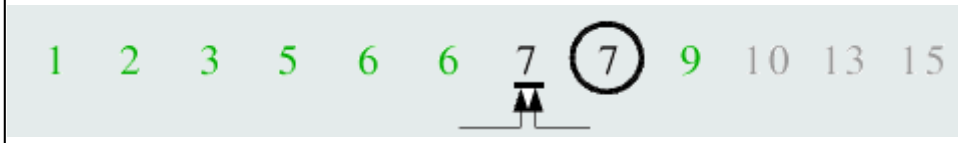
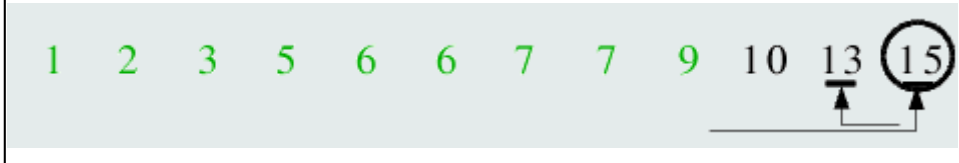
2.4 Ein Beispiel

Folgendes kommentiertes Beispiel soll den Durchlauf durch das Programm mit o.a. Code illustrieren:

1.1) Linker und rechter Folgenzeiger wurden initialisiert, 6 als Trennelement festgelegt. Von links wurde mit 6 ein Element gefunden, welches größer gleich dem Trennelement ist, von rechts wurde mit 5 ein kleineres Element gefunden. Diese werden miteinander vertauscht.	
1.2) Von links wird mit 13 sofort ein Element gefunden, welches größer als das	

Trennelement 6 ist. Von rechts läuft der Folgenzeiger bis 1, welches kleiner ist. Diese Element werden wieder getauscht.	
1.3) 7 wird als größeres, 2 als kleineres Element gefunden und miteinander vertauscht.	
1.4) Von links wird 9 als größeres, von rechts 2 als kleineres Element gefunden. Allerdings werden diese nicht miteinander vertauscht, da sich die beiden Folgenzeiger überschneiden und die Endlosschleife terminiert.	
1.5) Zum Schluss dieser ersten Vorsortierung wird noch das Trennelement mit dem äußerst linken Element der rechten Teilfolge getauscht und erhält damit seinen endgültigen Platz. Anschließend erfolgen die rekursiven Aufrufe für linke und rechte Teilfolge. Wie gut zu sehen ist, wurde die Folge in 2 Teile "geteilt": die Elemente in der linken Teilfolge sind kleiner und die in der rechten Teilfolge größer als das verbindende Trennelement.	
2.1) Der erste rekursive Aufruf erfolgt für die linke Teildatei. 2 als letztes Element ist Trennelement. Von links wird 5 als größer und von rechts 1 als kleiner gefunden und miteinander vertauscht.	
2.2) Da sich beide Zeiger überschneiden,	

wird nichts getauscht und die Endlosschleife terminiert.	
2.3) Zum Schluss wird das Trennelement wieder an seine endgültige Position an den linken Rand der rechten Teilfolge getauscht. Der rekursive Aufruf für die linke Teilfolge kehrt sofort zurück, da diese nur aus einem Element besteht, also folgt der Aufruf für die rechte Teilfolge.	
3.) 5 ist Trennelement. Die beiden Zeiger überschneiden sich sofort nach dem ersten Auffinden von größer-/kleiner-gleich-Elementen, sodass keine Vertauschungen stattfinden und die Endlosschleife zurückkehrt. Dann erfolgt das Tauschen des Trennelementes an die linke Position der rechten Teilfolge, welche allerdings schon die momentane Position des Trennelementes ist - daher kommt es zu keiner Änderung in der Folgenordnung. Die linke Teilfolge enthält 1 Element, die rechte Teilfolge sogar keines - infolge dessen kehrt die Funktion zurück.	
4.1) Die linke Teildatei der Ausgangsfolge gilt nun als sortiert, da alle Aufrufe bis zur ersten Rekursionsebene zurückkehren. Nun folgt der Aufruf für die rechte Teildatei. Zunächst ist 9 Trennelement, der linke Zeiger führt zur 13 als größeres und	

der rechte Zeiger zu 6 als kleineres Element, welche folglich miteinander vertauscht werden.	
4.2) Beim nächsten Halt der beiden Zeiger überschneiden sich diese schon, sodass die Endlosschleife abbricht.	
4.3) Nach dem Tausch des Trennelementes an seine endgültige Position erfolgt der Aufruf für die linke Teilfolge.	
5.1) 6 ist Trennelement, die Endlosschleife terminiert bereits beim ersten Anhalten der Zeiger, da der rechte Zeiger auf einem kleineren Index steht als der linke Zeiger.	
5.2) Das Trennelement wird auf seine Endposition getauscht. Die linke Teildatei ist leer, daher erfolgt der Aufruf für die rechte Teilfolge.	
6.1) 7 ist Pivot, nach dem Lauf der beiden Zeiger stehen diese auf demselben Element, sodass die Endlosschleife abbricht.	
6.2) Das Trennelement kommt auf seine endgültige Position und die Funktion kehrt zurück.	
7.1) In dieser äußerst rechten Teilfolge ist 15 das Pivot. Die beiden Zeiger überschneiden sich wieder sofort nach dem ersten Durchlauf, sodass nichts getauscht wird, nur das Trennelement noch an seine korrekte Position gelangt, auf welcher es sich jedoch schon befindet, sodass keine	

Reihenfolgeänderungen stattfinden.	
7.2) Zu guter letzt erfolgt der Aufruf für diese linke Teilfolge. 13 ist Trennelement, der erste Halt der Zeiger führt gleich zu einem Abbruch der Endlosschleife, sodass nichts getauscht wird und das Trennelement hat schon seine Endposition inne.	
8.) Alle Aufrufe kehren zurück bis zur ersten Rekursionsebene. Dort wurden linke und rechte Teildatei behandelt, sodass auch dieser Aufruf beendet ist - die Folge gilt als sortiert!	

2.5 Verbesserungen

2.5.1 Besseres Trennelement finden (Median-of-three)

Als Einleitung zu dieser Quicksort-Verbesserung zunächst noch einmal folgende Betrachtungen:
1.) Günstigster Fall: dieser tritt beim Quicksort-Algorithmus dann ein, wenn das betrachtete Teilfeld bei jedem Aufruf von quicksort() in zwei gleich große Stücke zerteilt wird.
2.) Ungünstigster Fall: diesen beobachtet man dann, wenn eines der beiden Stücke nur ein Element enthält. In diesem ungünstigsten Fall ruft sich quicksort() N-mal selber auf (N=Anzahl der Feldelemente), zudem benötigt der Algorithmus $N^2/2$ Vergleiche, hat also dieselbe Laufzeitkomplexität wie BubbleSort, welches man ja nicht wirklich als effizient bezeichnen kann.
Der ungünstigste Fall kann demnach z.B. dann eintreten, wenn man a[r] als zerlegendes Element nimmt und den Quicksort auf ein bereits sortiertes Feld loslässt. Eine sehr einfache Möglichkeit in diesem Fall wäre es, das Element a[l+(r-l)/2] als Trennelement zu benutzen, welches ja das (vom Index her) mittlere Feldelement darstellt. Hierbei kann es allerdings genauso leicht passieren, dass man ein Element erwischt, welches an einem der beiden Enden der Werteskala der Feldelemente angesiedelt ist, doch die Chance dafür ist schon geringer.
Eine sehr einfache und augenscheinlich sinnvolle Lösung ist hingegen, drei Elemente aus dem Feld zu entnehmen und das (wertmäßig) mittlere von ihnen als Trennelement zu betrachten. Zufallsgeneratoren sind in diesem Fall übertrieben, eine einfache Lösung des Herausnehmens der drei Elemente besteht darin, das erste Element mit a[l], das zweite mit a[l+(r-l)/2] und das dritte Element mit a[r] fest zu legen. Das Herausfinden und der Austausch zwischen den Elementen findet wie im folgenden C-Code gezeigt statt:

```
int m=l+(r-l)/2;
if(a[l]>a[m])
  { tmp=a[l]; a[l]=a[m]; a[m]=tmp; }
if(a[l]>a[r])
  { tmp=a[l]; a[l]=a[r]; a[r]=tmp; }
else if(a[r]>a[m])
  { tmp=a[r]; a[r]=a[m]; a[m]=tmp; }
```

Es ist zu erkennen, dass sich das wertmäßig mittlere Element der drei nach der Anwendung dieses Algorithmus' an der Stelle a[r] befindet. Dies hat Vorteile, da somit dieses Element vom Zähler j des Quicksort-Algorithmus nicht betrachtet werden muss und es sogleich an seine Stelle im sortierten Feld eingeordnet wird. Das spart natürlich Rechenzeit.
Die alles entscheidende Frage: Was bringt uns das? Die Beantwortung ist nicht leicht und es kann keine allgemeine Leistungskenngröße geben, was ganz einfach zu erklären ist: auch beim Herausnehmen und Vergleichen von drei Elementen kann es dazu kommen, dass man eben die drei größten bzw. kleinsten Elemente entnimmt, speziell kann ohne Verwendung eines Zufallsgenerators natürlich auch ein Feld konstruiert werden, welches sich am ungünstigsten für diesen Fall verhält. In der Praxis, und hier gehen wir vom Durchschnitt der Fälle aus, wird die Chance, eines der mittleren Elemente zu treffen, jedoch deutlich erhöht und somit erhöht sich natürlich auch die Wahrscheinlichkeit, dass ein Feld in zwei etwa gleich große Teilfelder zerlegt wird. Im Allgemeinen kann man von einer Geschwindigkeitssteigerung von 5% und (eher) mehr ausgehen.
Hier noch einmal der komplette verbesserte Quicksort-Algorithmus mit der Methode "median of three":

```
void qsort_median(int a[], int l, int r){
      if(r>l){
        int i=l-1, j=r, tmp;
        if(r-l > 3){ //Median of three
              int m=l+(r-l)/2;
              if(a[l]>a[m])
                { tmp=a[l]; a[l]=a[m]; a[m]=tmp; }
              if(a[l]>a[r])
                { tmp=a[l]; a[l]=a[r]; a[r]=tmp; }
              else if(a[r]>a[m])
                { tmp=a[r]; a[r]=a[m]; a[m]=tmp; }
        }

        for(;;){
              while(a[++i]<a[r]);
              while(a[--j]>a[r] && j>i);
              if(i>=j) break;
              tmp=a[i]; a[i]=a[j]; a[j]=tmp;
        }
        tmp=a[i]; a[i]=a[r]; a[r]=tmp;

        qsort_median(a, l, i-1);
        qsort_median(a, i+1, r);
      }
}
```

2.5.2 Behandlung kleiner Teilfelder

Neben der Überlegung, wie kleine Teilfelder zu vermeiden sind (median-of-three), kann man sich auch über die Behandlung solcher Gedanken machen. Wenn man sich den klassischen Quicksort anschaut, so stellt man fest, dass die Zerlegung (die rekursiven Funktionsaufrufe) für viele kleine Teilfelder bis hin zur Feldgröße 1 stattfindet. Folglich kann man festlegen, dass eine Möglichkeit gefunden werden muss, Quicksort für kleine Teilfelder schneller ablaufen zu lassen. Eine Möglichkeit, die oft angewandt wird, besteht darin, *insertion sort* auf ein Teilfeld loszulassen, falls die Größe des Feldes unter einen konstanten Schwellwert M sinkt. Dieser Quick-/Insertion-Sort-Mix lässt sich in C-Notation wie folgt darstellen:

```
void qsort_ins(int a[], int l, int r){
      int i, j, tmp;
      if(r-l > M){ //Quicksort
        i=l-1; j=r;
        for(;;){
              while(a[++i]<a[r]);
              while(a[--j]>a[r] && j>i);
              if(i>=j) break;
```

```
            tmp=a[i]; a[i]=a[j]; a[j]=tmp;
        }
        tmp=a[i]; a[i]=a[r]; a[r]=tmp;

        qsort_ins(a, l, i-1);
        qsort_ins(a, i+1, r);
      }
      else{ //insertion sort
        for(i=l+1; i<=r; ++i){
            tmp=a[i];
            for(j=i-1; j>=l && tmp<a[j]; --j)
              a[j+1]=a[j];
            a[j+1]=tmp;
        }
      }
}
```

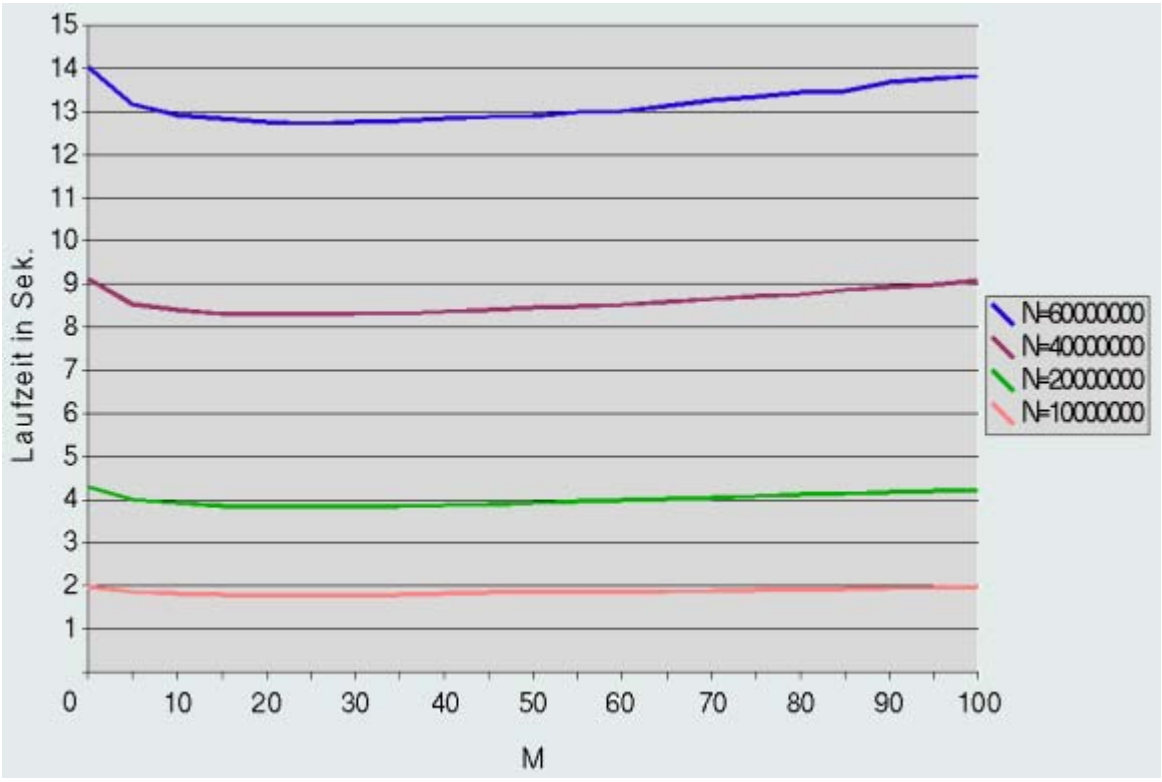
Ist die Bedingung if(r-l > M) nicht mehr erfüllt (ist das Teilfeld also "zu klein" für Quicksort), so wird für das behandelte Teilfeld insertion sort aufgerufen, ansonsten findet weiterhin das klassische Quicksort Anwendung.
Was bringt's?: Mit dieser Methode lassen sich viele rekursive Aufrufe sparen. In Abhängigkeit von M kann man Laufzeitverbesserungen von 20% und mehr beobachten, was wahrlich nicht zu unterschätzen ist.

Eine zweite Vorgehensweise in dieser Hinsicht beruht auf der Tatsache, dass insertion sort für sortierte Felder linear abläuft und für fast sortierte Felder effizient arbeitet. Also kann man o.a. Algorithmus so variieren, dass zunächst Quicksort aufgerufen wird, aber nicht mit der Abfrage if(r>l), sondern mit dem Vergleich if(r-l > M). Somit werden kleine Teilfelder von Quicksort gänzlich ignoriert. Zu seinem sortierten Feld kommt man schlussendlich, indem man dieses vorsortierte Feld einmal komplett von insertion sort durchlaufen lässt, welches in diesem Fall effektiv arbeitet. Trotzdem ist diese Vorgehensweise etwas uneffektiver als die zuerst vorgestellte, weshalb man obigen Code der folgenden C-Notation doch vorziehen sollte:

```
//Hilfsfunktion für qsort_ins2()
void qsort_M(int a[], int l, int r){
      int i, j, tmp;
      if(r-l > M){ //Quicksort
        i=l-1; j=r;
        for(;;){
            while(a[++i]<a[r]);
            while(a[--j]>a[r] && j>i);
            if(i>=j) break;
            tmp=a[i]; a[i]=a[j]; a[j]=tmp;
        }
        tmp=a[i]; a[i]=a[r]; a[r]=tmp;

        qsort_M(a, l, i-1);
        qsort_M(a, i+1, r);
      }
}

//2. Variante von Quick-/insertion-sort
void qsort_ins2(int a[], int l, int r){
      int i, j, tmp;
      qsort_M(a, l, r); //qsort_M zur Vorsortierung
      for(i=l+1; i<=r; ++i){ //insertion sort
        tmp=a[i];
        for(j=i-1; j>=l && tmp<a[j]; --j)
            a[j+1]=a[j];
        a[j+1]=tmp;
      }
}
```

2.5.3 Optimale Wahl von M

Bleibt noch die Frage offen, wie groß man den Schwellwert M festlegen sollte. Folgendes Diagramm zeigt die Laufzeit bei N=10 bis N=60 Mio. Elementen in Abhängigkeit von M=0 bis M=100:



Man sieht, dass M unabhängig von N zu sein scheint und ca. bei ***M=25*** die besten Ergebnisse liefert. Diesen Wert sollte man auch für M wählen, um die Verbesserung mit Insertion Sort optimal wirken zu lassen.

2.5.4 Beides zusammen?

Natürlich könnte man nun auf die Idee kommen, die beiden Verfahren zu kombinieren und somit eine noch größere Effizienzsteigerung zu erreichen. Dieses Vorgehen ist im folgenden C-Code fest gehalten:

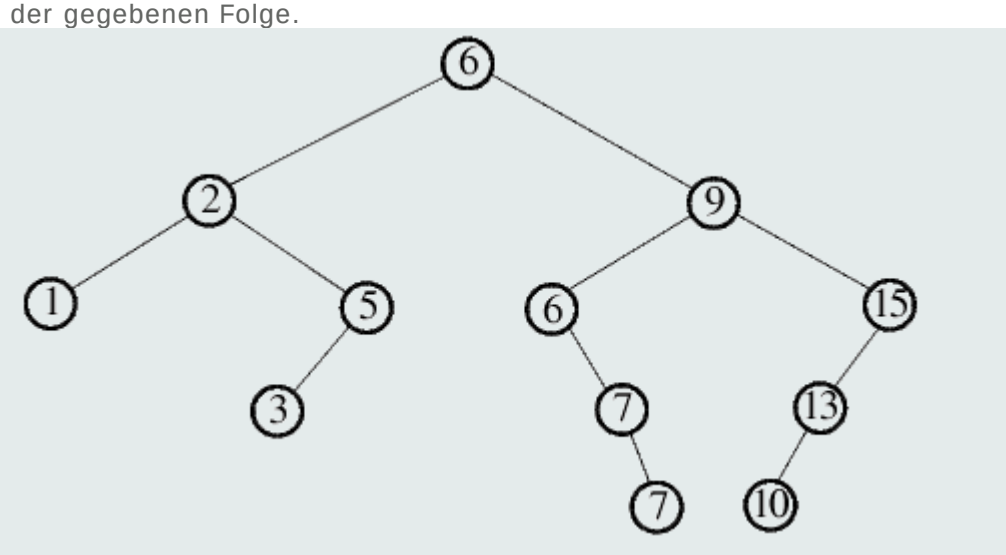
```
void qsort_comb(int a[], int l, int r){
      int i, j, tmp;
      if(r-l > M){ //Quicksort
        i=l-1; j=r;
        if(r-l > 3){ //Median of three
              int m=l+(r-l)/2;
              if(a[l]>a[m])
                { tmp=a[l]; a[l]=a[m]; a[m]=tmp; }
              if(a[l]>a[r])
                { tmp=a[l]; a[l]=a[r]; a[r]=tmp; }
              else if(a[r]>a[m])
                { tmp=a[r]; a[r]=a[m]; a[m]=tmp; }
        }

        for(;;){
              while(a[++i]<a[r]);
              while(a[--j]>a[r] && j>i);
              if(i>=j) break;
              tmp=a[i]; a[i]=a[j]; a[j]=tmp;
        }
        tmp=a[i]; a[i]=a[r]; a[r]=tmp;
```

```
          qsort_comb(a, l, i-1);
          qsort_comb(a, i+1, r);
        }
        else{ //insertion sort
          for(i=l+1; i<=r; ++i){
                tmp=a[i];
                for(j=i-1; j>=l && tmp<a[j]; --j)
                  a[j+1]=a[j];
                a[j+1]=tmp;
          }
        }
  }
```

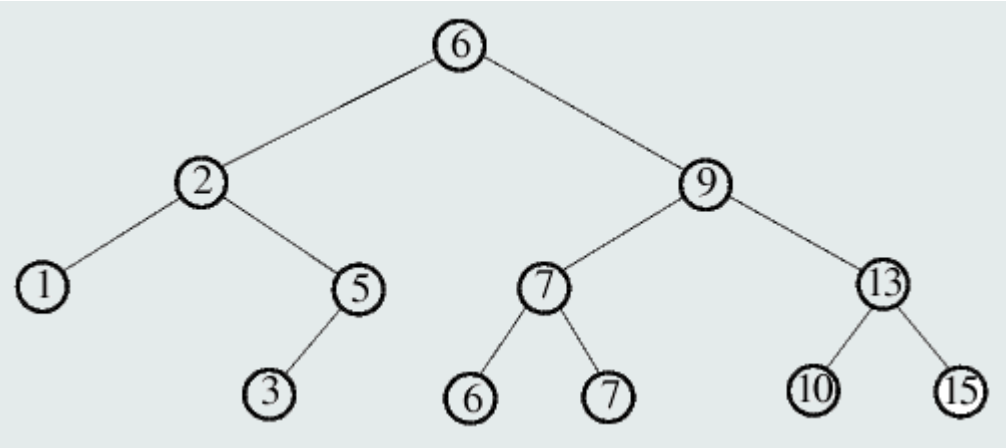
Doch Vorsicht: dieser Algorithmus *kann* uneffektiver sein als der zuvor vorgestellte insertion-/Quicksort-Mix! Grund: wählt man einen großen Schwellwert M, für welchen Quicksort in insertion sort übergeht, so ist die Chance gering, in einer Teildatei, welche ja mindestens die Größe M haben muss, als Trennelement eines heraus zu greifen, welches sich im oberen bzw. unteren Teil des Feldes befindet. Somit kann es unangebracht sein, ein median-of-three (also das wertmäßig mittlere von drei Elementen) zu ermitteln, da die Vergleiche und Austauschoperationen hierbei mehr Rechenzeit kosten können als wie sie ersparen. Es ist also eher eine Gewissensfrage, ob man median-of-three und das insertion-sort-Verfahren für kleine Teilfelder kombiniert. Im Allgemeinen wird dadurch Rechenzeit erspart, trotzdem kann selbst bei zufälligen Permutationen allein die Verbesserung mittels insertion sort schneller sein.
Zu kombinieren empfehle ich beide Verfahren bei optimal gewähltem M, wobei auch bei der Mischung mit median-of-three M=25 am effektivsten ist, doch auch dies ist subjektiv zu entscheiden und kann nicht verallgemeinert werden.

2.6 Laufzeitbetrachtungen

Ein Durchlauf von Quicksort lässt sich gut als Binärbaum darstellen. Folgende Abbildung zeigt obiges Beispiel: dabei ist ein Knoten des Baumes jeweils das aktuell gewählte Trennelement, ein linker Sohn ist das Trennelement der linken, ein rechter Sohn das Trennelement der rechten Teilfolge. Anhand dieses Binärbaumes lassen sich gute Aussagen über die Laufzeit des Algorithmus' treffen, so entspricht die Höhe des Baumes z.B. der Rekursionstiefe für eine Sortierung der gegebenen Folge.

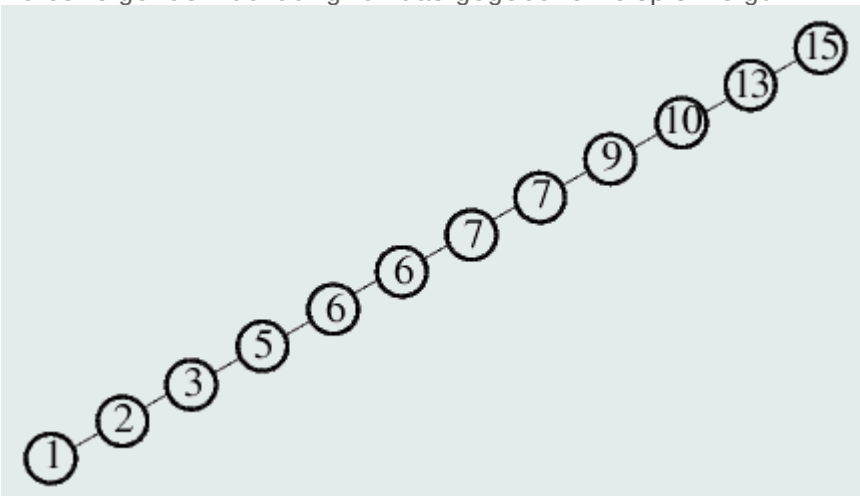


Wie schon erwähnt, tritt der günstigste Fall dann ein, wenn bei allen rekursiven Aufrufen die jeweilige Folge in 2 gleichgroße Teilfolgen zerlegt wird. Dies geschieht, wenn stets das wertmäßig mittelste Element der Teilfolge als Pivotelement gewählt wird (dass dies nicht so einfach geht, wurde ja schon ausführlich besprochen). Schauen wir uns den Binärbaum für obiges Beispiel an, wenn immer das beste Pivotelement anstelle des letzten Elements gewählt worden wäre:



Hätte die Folge noch 3 Elemente mehr gehabt, so würde es sich hierbei um einen *vollständigen Binärbaum* handeln, d.h. jeder Knoten des Baumes hätte stets 2 Söhne. Doch auch so kann man die Ausgeglichenheit erkennen - die Höhe des Baumes und damit auch die Rekursionstiefe ist für diese Anzahl von Elementen minimal. Was bedeutet das? Pro Rekursionsschicht werden bei einem vollständigen Binärbaum rund *n* Schlüsselvergleiche durchgeführt, da jedes Element bis auf die vorigen Pivotelemente einmal betrachet und mit dem jeweiligen Pivot verglichen wird. Ein vollständiger Binärbaum hat die Höhe *log(n)*, sodass man im günstigsten Fall also auf *n*log(n) Schlüsselvergleiche* kommt.

Im Gegensatz dazu tritt der ungünstigste Fall ein, wenn bei einem Rekursionsaufruf immer nur genau ein Element abgespalten wird. Der zugehörige Binärbaum würde zu einer verketteten Liste entarten, wie es folgende Abbildung für das gegebene Beispiel zeigt:



Dazu müsste immer das momentan größte (bzw. immer das momentan kleinste) Element der Folge als Trennelement ausgewählt werden. Wenn wir das grundlegende Quicksort nehmen würden, wo immer das letzte Element als Pivot gewählt wird, wäre der ungünstigste Fall dann gegeben, wenn die Folge bereits auf- bzw. absteigend sortiert vorliegt. Durch median-of-three könnte man dies schon einmal vermeiden.
Für die Abspaltung eines Elementes benötigt man pro Rekursionsschicht ca. *n/2 Vergleiche*, die Höhe des Baumes und die damit verbundene Rekursionstiefe beträgt *n*. Somit ergibt sich für den ungünstigsten Fall eine Anzahl von $n^2/2$ *Schlüsselvergleichen*.

Und was lässt sich über den mittleren Fall aussagen? Es zeigt sich, dass bei zufälligen Permutationen der grundlegende Quicksort-Algorithmus im Mittel nicht viel schlechter arbeitet als im ungünstigsten Fall, die Wahl des Pivots also im Schnitt auf ein mittleres Element fällt. So lässt sich zeigen, dass die Zahl der Schlüsselvergleiche nur um 38% höher ist als im best case - daher lässt sich sagen:
*Im Mittel benötigt Quicksort ungefähr 2*n*ln(n) Schlüsselvergleiche.*

Und dieser gute durchschnittliche Fall ist auch einer der Hauptgründe, warum sich Quicksort

über die Jahre hinweg als eines der beliebtesten Sortierverfahren etabliert hat.

[Zurück zum Inhaltsverzeichnis]

3. Mergesort

[Zurück zum Inhaltsverzeichnis]

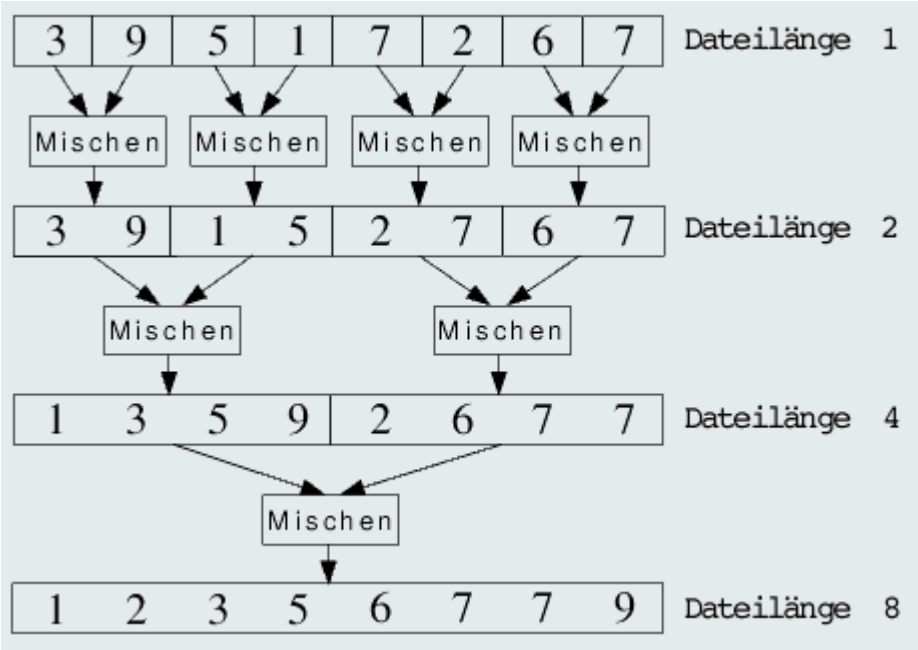
3.1 Historie / Hintergrund

Mergesort, das "Sortieren durch Verschmelzen", ist eines der ältesten und best-untersuchten Sortierverfahren, welches bereits 1945 von Computerpionier John von Neumann vorgeschlagen und auf dem EDVAC-Computer von ihm implementiert wurde. Im Vergleich zu Quicksort, wo eine Folge rekursiv in 2 Teilfolgen geteilt wurde, findet bei Mergesort ein quasi komplementärer Prozess statt: beim sogenannten *Mischen* (*Merging*) werden 2 sortierte Dateien zu einer sortierten Datei zusammen gefügt.
"Teile-und-Herrsche" im Sinne von Quicksort bedeutete: eine Datei wird so umgeordnet, dass sie als sortiert gilt, wenn ihre beiden Teile sortiert sind. Bei Mergesort, welches diesem Prinzip ebenfalls entspricht, bedeutet es hingegen, dass eine Datei in zwei Teile zerlegt wird, die zu sortieren und dann derart zu kombinieren sind, dass sich eine sortierte Gesamtdatei ergibt.
Das *Mischen* stellt nicht nur einen Konterpart zum *Teilen* dar, auch ist Mergesort selbst bei rekursiver Implementierung in gewisser Weise ein Komplement zu Quicksort: bei diesem wird zunächst der Teilprozess ausgeführt, gefolgt von den beiden rekursiven Aufrufen für die linke und rechte Teilfolge. Bei Mergesort hingegen werden erst die rekursiven Aufrufe ausgeführt, um anschließend die dadurch entstandenen Teildateien im Mischprozess wieder zu vereinen.
Mergesort bietet einige grundlegende Vorteile gegenüber anderen Sortieralgorithmen: zum einen ist es stabil, d.h. die relative Sortiertheit gleicher Schlüssel in einer Datei bleibt stets erhalten. Zum anderen hat es auch im ungünstigsten Fall eine Laufzeit proportional zu *n*log(n)*, also *O(n*log(n))*. Weiterer Vorteil: Mergesort kann so implementiert werden, dass es die Daten hauptsächlich sequentiell abarbeitet, was vor allem beim Sortieren auf Externspeicher und von verketteten Listen wichtig ist.
Hauptnachteil: für den Algorithmus wird ein zu n proportionaler zusätzlicher Speicher benötigt - daher wird Mergesort vor allem dort angewandt, wo zusätzlicher Speicherplatz keine Rolle spielt. Zudem ist es oftmals das Verfahren der Wahl, wenn es gilt eine große sortierte Datenmenge zu verwalten und immer wieder Datensätze einzuordnen. Normalerweise würden die neuen Daten angehangen und die komplette Datei neu sortiert werden. Weitaus besser wäre es hingegen, die neuen Daten getrennt zu sortieren und dann mit der bestehenden Datei zu vereinen - ein ideales Szenario für den Mischprozess von Mergesort.

3.2 Der Algorithmus

So wie man bei Quicksort das Prinzip des Teilens verstehen muss, so muss man bei Mergesort den Mischprozess verstehen. Mischen im Sinne von Mergesort bedeutet, dass man zwei sortierte Dateien zu einer einzigen sortierten Datei vereint. Zuerst erfolgt also eine Aufteilung der Gesamtdatei auf Dateien der Länge 1. Nun nimmt man benachbarte Teildateien dieser Länge, die also nur aus einem Element bestehen und trivial als sortiert gelten, und mische (sortiere und kombiniere) sie zu Dateien der Länge 2. Diese nimmt man wiederum und mische sie zu Dateien der Länge 4 usw. bis es nur noch 2 Dateien gibt. Diese werden schlussendlich ebenfalls gemischt, sodass eine einzige Datei entsteht, die der sortierten Ausgangsfolge entspricht.
Diese Art der Kombination zweier Dateien zu einer heißt *2-Wege-Mergesort*. Es gibt weiterhin x-Wege-Mergesort-Varianten, welche x Dateien zu einer vereinen - diese sollen hier allerdings nicht weiter betrachtet werden.

Folgendes kleines Beispiel illustriert das grundlegende Prinzip recht anschaulich: durch die Rekursion erfolgt zunächst eine Aufteilung der Datei bis zu einer Länge 1, dann werden die Daten durch den Mischprozess paarweise zusammen gefügt, wobei gleichzeitig die Sortierung stattfindet. Die genaue Implementation dieses Prozesses ist im folgenden Abschnitt erläutert.



3.3 Zur programmiertechnischen Realisierung

3.3.1 Rekursiv:
Zunächst soll hier die rekursive Variante des 2-Wege-Mergesort besprochen werden. Wie schon erwähnt erfolgen im Gegensatz zu Quicksort am Anfang der Funktion die rekursiven Aufrufe, wodurch die aktuelle Datei in der Mitte in 2 Hälften geteilt wird. Dies geschieht solange, bis alle Dateien nur noch die Länge 1 haben, dann erfolgen keine weiteren Rekursionen mehr. Der nächste Schritt ist das Mischen: 2 benachbarte Dateien der Länge *x* und *y* werden zu einer Datei der Länge *x+y* kombiniert, wobei die entstehende Datei sortiert sein soll. Hier macht sich Mergesort die Eigenschaft zunutze, dass Dateien der Länge 1 bereits trivial sortiert sind und somit unter der Eigenschaft kombiniert werden kann, dass die beiden Teildateien bereits sortiert vorliegen. Dies erfordert nun natürlich, dass Mergesort die Dateien beim Vereinen gleich sortiert, was allerdings einfach zu realisieren ist: für die Grundidee benötigt man nur 2 Ganzzahlen auf Arrayindizes. Den ersten Zeiger i initialisiert man mit der Anfangsposition der linken Datei, der zweite Zeiger j erhält den Anfang der rechten Datei. Da man weiß, dass die beiden Dateien schon sortiert sind, kann man wie folgt vorgehen: man durchlaufe mit den beiden Zeigern die Dateien derart, dass die beiden Schlüssel, auf welche i und j zeigen, miteinander verglichen werden. Der kleinere der beiden Schlüssel wird in einen anfangs leeren Zwischenspeicher eingefügt und der mit diesem Schlüssel verbundene Arrayzeiger um eine Position weiter gesetzt. Dieser Vorgang wird solange wiederholt, bis eine der beiden Teildateien erschöpft ist. In diesem Fall kann der Rest der anderen Teildatei an die bisherige Folge angehangen werden - die beiden Dateien sind nun vereint und ergeben eine einzige sortierte Folge.

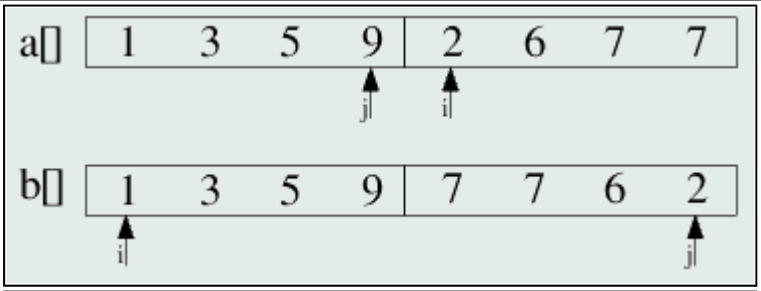
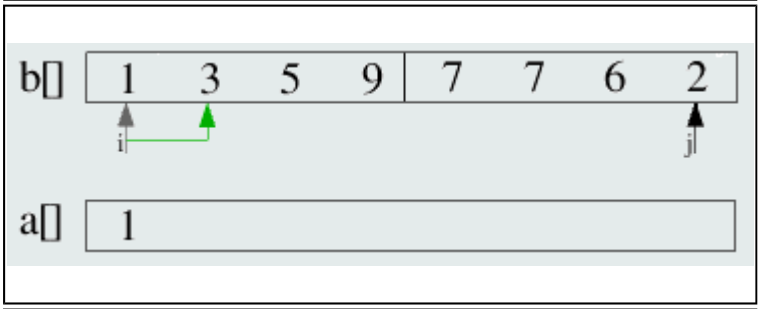
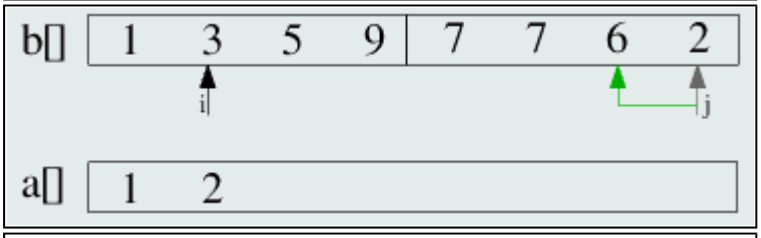
Zur Realisierung dieses Algorithmus' würde man noch mehrere Marken benötigen, um die Enden der Teildateien abfragen zu können. Es geht aber auch anders: effizienter ist es, die beiden Zeiger derart auszunutzen, dass der eine die Marke für den anderen ergibt. Somit wäre folgende rekursive Realisierung denkbar:

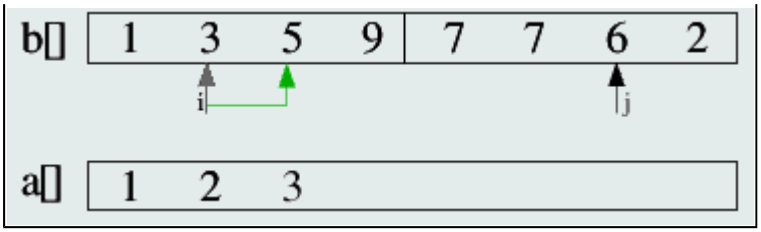
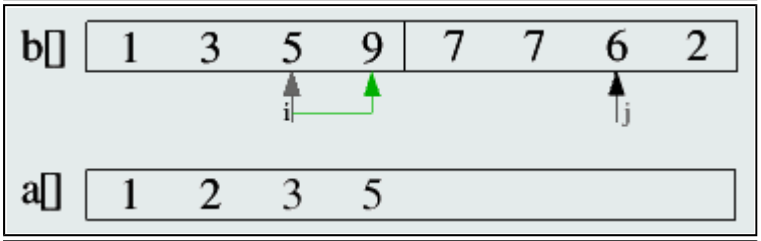
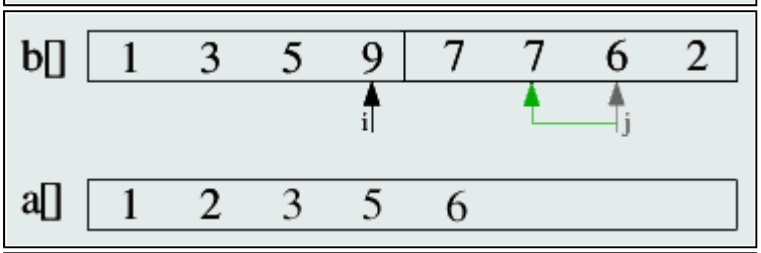
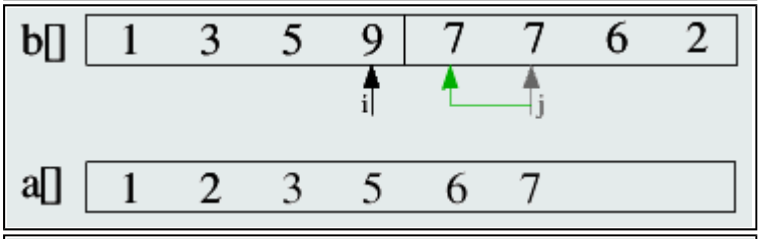
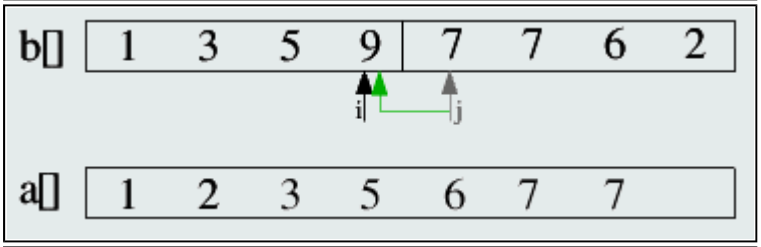
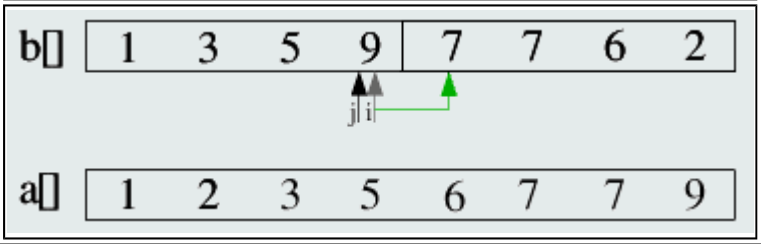
```
void mergesort(int a[], int l, int r){          //l=linker Rand, r=rechter Rand
    if(r>l){
        int i, j, k, m;                         //Variablen deklarieren
        m=(r+l)/2;                              //Mitte ermitteln
        mergesort(a, l, m);                     //linke Teildatei
        mergesort(a, m+1, r);                   //rechte Teildatei
        for(i=m+1; i>l; i--) b[i-1]=a[i-1];     //linke  Teildatei in Hilfsarray
        for(j=m; j<r; j++) b[r+m-j]=a[j+1];     //rechte Teildatei umgedreht in H
        for(k=l; k<=r; k++)
```

```
            a[k]=(b[i]<b[j])?b[i++]:b[j--];     //füge sortiert in a ein
    }
}
```

Zu beachten ist, dass man ein globales Hilfsfeld b[] zur Verfügung stellen muss, welches die gleiche Dimension wie das zu sortierende Feld a[] hat. Ein Sortieren eines Feldes a[] erfolgt dann durch den Aufruf von *mergesort(a,0,n-1);* wobei *n* die Elementeanzahl darstellt.
Zur Analyse: besteht die Datei aus mehr als 1 Element (r>l), so folgen zunächst die rekursiven Aufrufe für linke und rechte Teildatei. Dann werden diese beiden Teildateien aus Array a[] in das Hilfsarray b[] übertragen. Die erste for-Schleife ist für die linke Teildatei zuständig und speichert sie in der Reihenfolge in b[] ab, wie sie auch schon in a[] vorkommt. Dabei ist zu beachten, dass i zu Beginn auf das letzte Element der Teildatei gesetzt ist und bis zum linken Rand dekrementiert wird. Dass dies Sinn macht, wird gleich verdeutlicht. Die zweite for-Schleife kümmert sich um die rechte Teildatei. j wird vom Anfang der Teildatei ausgehend solange inkrementiert, bis es den rechten Rand erreicht hat. Die Abspeicherung dieser Teildatei in b[] erfolgt allerdings in umgekehrter Reihenfolge.
Die Zeiger i und j werden jetzt nur noch für das Auslesen aus dem Hilfsfeld verwendet: i steht auf dem linken Rand der Datei und somit auf dem kleinsten Element der linken Teildatei, j zeigt auf den rechten Rand der Datei und auch auf das kleinste Element der rechten Teildatei, da diese in umgekehrter Reihenfolge in das Hilfsfeld eingefügt wurde. Warum das Ganze? Ganz einfach: man benötigt keine weiteren Marken zur Prüfung des Endes der beiden Teildateien. Man muss nur mit der dritten for-Schleife die Elemente vergleichen, auf welche die beiden Zeiger zeigen. Das kleinere von beiden wird zurück in a[] eingefügt und der Zeiger im Falle von i eine Position weiter, im Falle von j eine Position zurück gesetzt. Ist eine der beiden Folgen erschöpft, so steht dessen Zeiger auf dem größten Element der anderen Folge und verharrt dort, bis deren Feldzeiger zum selben Element gelangt. Dieses letzte Element wird noch in a[] eingefügt und die letzte for-Schleife terminiert, da alle Elemente betrachtet wurden - die beiden Teildateien sind miteinander verschmolzen und die entstandene Datei ist sortiert.

Folgendes selbst gewählte Beispiel verdeutlicht ausführlich die Vorgehensweise des obigen Algorithmus':

1.) Zunächst werden die beiden schon sortierten Teildateien aus a[] nach b[] verschoben, wobei die rechte Teildatei in umgekehrter Reihenfolge in b[] eingeordnet wird.	
2.1) Nun erfolgt die Rückverschiebung nach a[]. i steht zu Beginn auf dem linken, j auf dem rechten Rand der Datei und auf den jeweils kleinsten Elementen der beiden Folgen. Das Element, auf das i zeigt, ist kleiner als das Element von j (1<2), also wird 1 nach a[] verschoben und i inkrementiert.	
2.2) 3 ist größer als 2, also wird 2 in a[] eingefügt und j dekrementiert.	

2.3) 3<6, also 3 in a[] einfügen und i inkrementieren.	
2.4) 5<6, also 5 in a[] einfügen und i inkrementieren.	
2.5) 9>6, also 6 in a[] einfügen und j dekrementieren.	
2.6) 9>7, also 7 in a[] einfügen und j dekrementieren.	
2.7) 9>7, also 7 in a[] einfügen und j dekrementieren.	
2.8) j zeigt auf das letzte Element der linken Teilfolge, auf das auch i zeigt. Dieses letzte Element 9 wird nach a[] verschoben - die beiden sortierten Teildateien wurden zu einer sortierten Datei gemischt und der Algorithmus ist beendet.	

3.3.2 Iterativ

Im Gegensatz zu Quicksort lässt sich eine iterative Variante von Mergesort recht gut realisieren, vor allem weil die zu sortierende Datenfolge sequentiell abgearbeitet werden kann und kein zusätzlicher Speicherplatz für einen Hilfsstapel benötigt wird, der die Grenzen von Teilfolgen abspeichern muss.

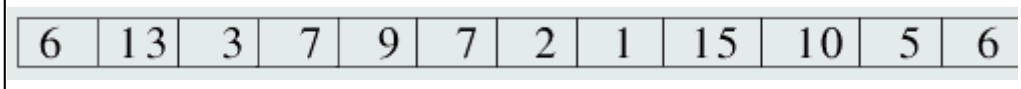
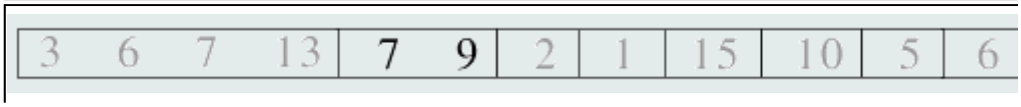
Das sogenannte *Bottom-up-Mergesort* ist die einfachste Realisierung eines iterativen Mergesort's. Dabei wird die Liste fortwährend durchlaufen, wobei bei jedem Durchlauf die zu kombinierende Folgenlänge verdoppelt wird. So beträgt diese Länge beim ersten Durchlauf 1, sodass benachbarte Folgen der Länge 2 gebildet werden. Beim zweiten Durchlauf werden dann Folgen der Länge 4, beim dritten der Länge 8 usw. erstellt, bis die Datei aus nur noch einer Folge besteht. Wichtig ist anzumerken, dass die Mischoperationen bei der iterativen Variante nicht denen bei der Rekursion gleichen. Da bei der Rekursion stets in 2 Teile zerlegt wird und bei der iterativen Variante sequentiell benachbarte Dateien gleicher Länge kombiniert werden, handelt es sich z.B. bei einer Folge mit 11 Elementen bei der letzten Sortierung mit der rekursiven Version um ein *5-mit-6*-Mischen, während bei der Iteration ein *8-mit-3*-Mischen

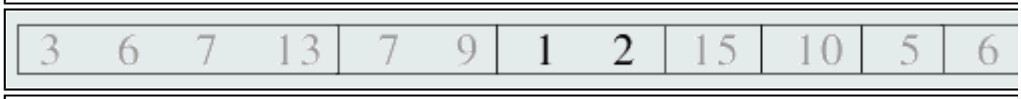
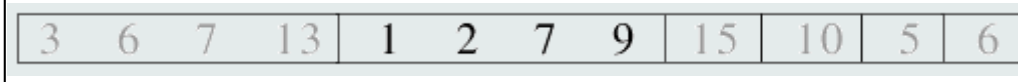
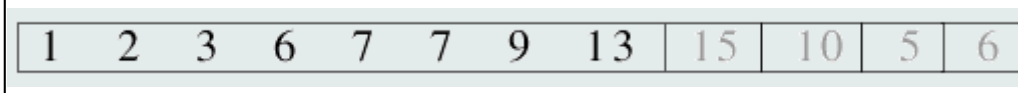
ausgeführt wird (die erste Zahl ist hier immer eine Zweierpotenz).
Folgender Code zeigt eine iterative Implementierung des Mergesort-Algorithmus', ein Aufruf der Funktion für ein zu sortierendes Feld a mit n Elementen erfolgt mit *mergesort_it(a,n-1);* :

```
void mergesort_it(int a[], int n){              //n+1=Elementezahl, größer 1
    int step, i, j, k, m, l, r;
    for(step=2; step<n*2; step*=2){             //verdopple sequentiell die Datei
        r=-1;                                   //Anfangswert für rechte Grenze
        while(r<n){                             //solange Folge nicht komplett be
            l=r+1;                              //linken Rand auf bisher rechte G
            r+=step;                            //rechten Rand um step erhöhen
            m=(l+r)/2;                          //Mitte finden
            if(r>n){                            //wenn über das Ziel hinaus gesch
                r=n;                            //dann auf letztes Element setzen
                if(m>r) m=(l+r)/2;              //wenn Mitte hinter r liegt, neu
            }
            for(i=m+1; i>l; i--) b[i-1]=a[i-1]; //linke  Teildatei in Hilfsarray
            for(j=m; j<r; j++) b[r+m-j]=a[j+1]; //rechte Teildatei umgedreht in H
            for(k=l; k<=r; k++)
                a[k]=(b[i]<b[j])?b[i++]:b[j--]; //füge sortiert in a ein
        }
    }
}
```

3.4 Ein Beispiel zu rekursivem Mergesort

Das folgende Beispiel zeigt, wie die Teildateien bei rekursivem Mergesort miteinander verbunden und dabei sortiert werden:

1.) Die Ausgangssituation: die Datei wurde durch Rekursion komplett in Teildateien der Länge 1 zerlegt, nun erfolgt paarweises Mischen.	
2.1) Die ersten beiden Teildateien werden zu einer Datei der Länge 2 gemischt.	
2.2) Die zweiten beiden Teildateien werden ebenso zu einer Datei der Länge 2 kombiniert.	
2.3) Die beiden entstandenen Dateien der Länge 2 werden zu einer Datei der Länge 4 gemischt.	
2.4) 7 und 9 werden	

kombiniert...	
2.5) ... ebenso 1 und 2.	
2.6) Kombination der beiden Teildateien zu einer Datei der Länge 4.	
2.7) Die beiden bereits sortierten Länge-4-Dateien werden miteinander gemischt.	
2.8) 10 und 15 werden zusammengefügt.	
2.9.) 5 und 6 kommen zu einer Datei der Länge 2 zusammen.	
2.10) Beide Dateien werden zu einer Länge-4-Datei vereint.	
3.) Die beiden übrig bleibenden Dateien werden zu einer einzigen sortierten Datei kombiniert - der Algorithmus ist beendet.	

3.5 Analyse der Leistungsfähigkeit

Das umstrittene Genie von Neumann hat mit Mergesort schon in der "Computersteinzeit" ein recht einfaches optimales Sortierverfahren entwickelt, welches leicht stabil implementiert werden kann.
Dabei entspricht der günstigste ungefähr dem ungünstigsten Fall, die Anzahl der zu erledigenden Vergleiche variiert kaum. Die Laufzeitkomplexität kann leicht berechnet werden: pro Rekursionsebene werden n Vergleiche ausgeführt, ein Vergleich für jeden betrachteten Schlüssel. Insgesamt gibt es log(n) Rekursionsebenen, da sich pro Schicht die aktuelle Dateilänge halbiert (mit einem Binärbaum vergleichbar). Daraus ergibt sich eine Komplexität von *O(n*log(n))*, also das Optimum für ein Sortierverfahren.
Der Grund, warum Mergesort Quicksort nicht allgemein bevorzugt wird, ist der, dass es in praxi bei zufälligen Eingabedaten mehr Zeit zum Sortieren beansprucht als Quicksort mit derselben Datei - dies gilt für das grundlegende Quicksort ebenso wie für ein verbessertes Quicksort mit median-of-three und/oder insertion sort zur Behandlung kleiner Teidateien. Der Wertebereich des Geschwindigkeitsvorteils bewegt sich hier von 10-50%. Da bei Mergesort zusätzlicher Speicher benötigt wird, fällt daher eher die Wahl auf Quicksort. Zwischen iterativem und rekursivem Mergesort gibt es ebenfalls noch einmal Geschwindigkeitsunterschiede. Bei o.a. Implementationen ist die rekursive Variante ca. 15% schneller als die iterative Version, auch dies gilt es zu berücksichtigen.
Da Mergesort gegenüber der ursprünglichen Reihenfolge der Eingabedaten unempfindlich ist, ist es bei zufälligen Permutationen genauso effizient wie etwa bei bereits sortierten Folgen.

Demgegenüber steht, dass es zu n proportionalen zusätzlichen Speicher benötigt, was bei immer höheren Speicherkapazitäten allerdings kein Problem mehr darstellen dürfte. Zudem ist es als sequentielles und stabiles Verfahren vor allem für Sortierungen von verketteten Listen und auf Externspeicher unentbehrlich, was seine Praxisrelevanz über die Jahre hin gesichert hat.

[Zurück zum Inhaltsverzeichnis]

4. Heapsort

[Zurück zum Inhaltsverzeichnis]

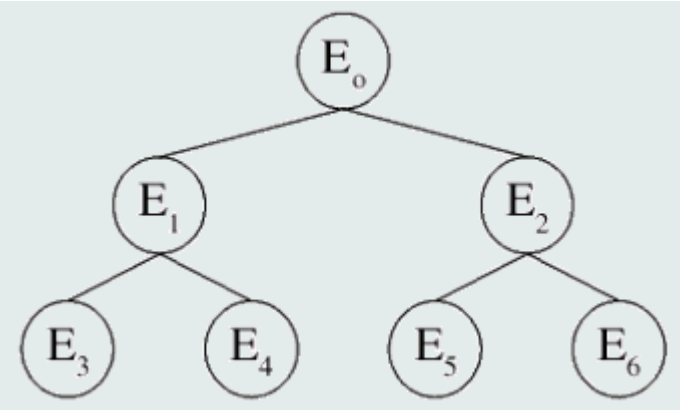
4.1 Hintergrund

Heapsort ist ein algorithmisch interessantes, wenn auch kein schnelles zeitoptimales Sortierverfahren, welches entgegen Quicksort auch im worst case eine Laufzeitkomplexität von O(n*log(n)) besitzt. Ebenso handelt es sich nicht um einen Algorithmus nach dem "Teile-und-Herrsche"-Prinzip, jedoch wird auch hier durch eine Auswahl sortiert, wobei diese allerdings geschickt organisiert wird. Hierfür verwendet das Verfahren die Datenstruktur des *Heaps* (engl.: Halde), in welcher die Bestimmung des Maximums bzw. Minimums einer Menge von n Schlüsseln in nur einem Schritt möglich ist und welche sich dadurch für ein Sortierverfahren bestens eignet. Allerdings ist die Heap-Operation des *Versickerns*, welche dies ermöglicht, nicht trivial, sodass man über eine Komplexität von O(n*log(n)) nicht hinaus kommt.
Mit der Beschreibung des Heaps und der von uns benötigten elementaren Operation soll hier zunächst eine grundlegende Wissensbasis geschaffen werden, bevor auf das damit verbundene Heapsort eingegangen wird, welches dann nahezu trivial zu verstehen ist.

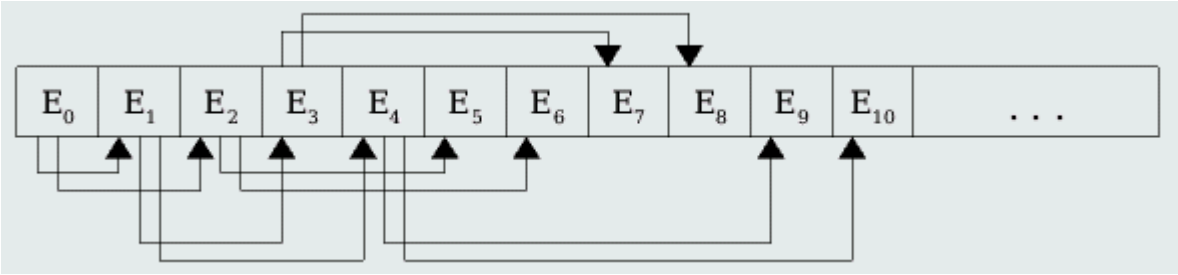
4.2 Die Datenstruktur Heap

4.2.1 Grundlagen:

Ein Heap ist im Grunde nur eine elementare Struktur zur Speicherung von Daten, wie es auch ein Stapel (Stack) oder eine Schlange (Queue) ist, nur dass die Daten hier in einer gewissen Art und Weise bereits *vorsortiert* sind. Deswegen spricht man bei Heaps auch von *Prioritätswarteschlangen*, was nicht über den Fakt hinwegtäuschen soll, dass es sich hierbei grundlegend nur um eine Folge von n vergleichbaren Schlüsseln handelt.
Die Ordnung, welche in einem gültigen Heap vorliegen muss, soll im kommenden Unterpunkt 4.2.2 verdeutlicht werden. Allgemein gefasst ist ein Heap ein Binärbaum, d.h. ein Knoten hat jeweils zwei Söhne (linker und rechter Sohn), was sich bei n zu speichernden Elementen E_0, E_1, E_2, ..., E_{n-1} wie folgt darstellt:



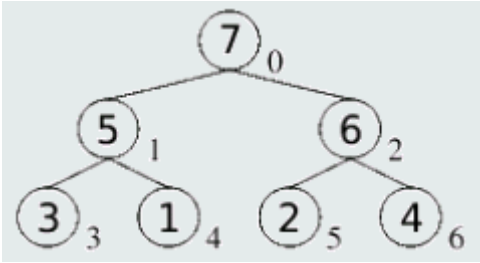
Objektorientierte Realisierungen sind zwar denkbar, die interne Speicherung erfolgt allerdings meist über ein Array, in welchem das Element E_0 die Wurzel ist und allgemein die Söhne eines Elementes E_k E_{2k+1} und E_{2k+2} sind, da sich auf einer Ebene i des Binärbaumes maximal 2^i Elemente befinden und somit effizient und eindeutig im Array gespeichert werden kann.
Folgende Abbildung illustriert die interne Speicherstruktur eines Binärbaumes in Arrayform:



4.2.2 Heapordnungen:
Ein Heap kann in zwei möglichen Varianten implementiert werden. Bei der einen, dem sogenannten *MinHeap* liegt eine Min-Ordnung vor, beim *MaxHeap* auf der anderen Seite eine Max-Ordnung. Beide Varianten sollen nachfolgend kurz vorgestellt werden.

4.2.2.1 MaxHeap:
Ein Heap heißt **MaxHeap** bzw. es liegt eine Max-Ordnung vor, wenn für alle inneren Knoten deren Schlüssel größer gleich denen der beiden Söhne sind, d.h. es gilt die (Max-) Heapbedingung: *In einem MaxHeap gilt für alle inneren Knoten i:* $E_i >= E_{2i+1}$ *und* $E_i >= E_{2i+2}$.
Anders ausgedrückt: ein Binärbaum stellt genau dann einen MaxHeap dar, wenn der Schlüssel jedes Knotens mindestens genauso groß ist wie der seiner beiden Söhne.
Beispiel: Folgender Binärbaum ist ein MaxHeap:

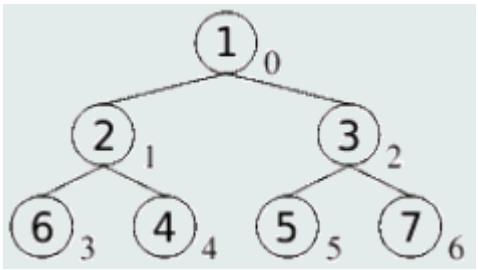
Index	0	1	2	3	4	5	6
Element	7	5	6	3	1	2	4



Deutlich zu erkennen ist, dass in dem Array an erster Stelle das *größte* Element der Folge steht.
Das Finden des Maximums stellt somit nur einen Zugriff auf das Element E_0 dar.

4.2.2.2 MinHeap:
Ein Heap heißt **MinHeap** bzw. es liegt eine Min-Ordnung vor, wenn für alle inneren Knoten deren Schlüssel kleiner gleich denen der beiden Söhne sind, d.h. es gilt die MinHeap-Bedingung: *In einem MinHeap gilt für alle inneren Knoten i:* $E_i <= E_{2i+1}$ *und* $E_i <= E_{2i+2}$.
Anders ausgedrückt: ein Binärbaum stellt genau dann einen MinHeap dar, wenn der Schlüssel jedes Knotens höchstens so groß ist wie der seiner beiden Söhne.
Beispiel: Folgender Binärbaum ist ein MinHeap:

Index	0	1	2	3	4	5	6
Element	1	2	3	6	4	5	7



Analog zum MaxHeap steht in dem Array an erster Stelle das *kleinste* Element der Folge.
Das Finden des Minimums stellt somit nur einen Zugriff auf das Element E_0 dar.

Dies sind die beiden Möglichkeiten, wie ein regulärer Heap organisiert werden kann. In der Literatur wird der MaxHeap meist nur als Heap bezeichnet, aus gutem Grund. Denn obwohl man mit beiden Heapordnungen (Min/Max) eine Sortierung durchführen kann, bringt die Verwendung von MaxHeaps Vorteile mit sich, die sie in der Praxis gegenüber MinHeaps

hervorheben, wie wir nachfolgend noch sehen werden.

4.2.3 Die Operation *Versickern*:

Diese grundlegende Heap-Methode wird für den effizienten Aufbau eines Heaps ebenso benötigt wie für die Sortierung an sich. Stellen wir uns vor, wir entfernen das erste Element eines Heaps (also die Wurzel des Binärbaums) und setzen an dessen Stelle ein anderes Element. In diesem Fall hat man im linken sowie rechten Teilbaum der Wurzel je einen Heap, der gesamte binäre Baum stellt im Allgemeinen aber keinen Heap mehr dar, da nicht klar ist, ob die Heapbedingung für die Wurzel und ihre beiden Söhne noch erfüllt ist.
Daher muss das Element in der Wurzel an seine korrekte Position im Baum gebracht werden, sodass dieser wieder einen Heap darstellt, die Heapbedingung also wieder erfüllt ist - hierzu lässt man das Element im Baum *"versickern"*.
Bei der Methode des Versickerns nutzt man die Eigenschaft aus, dass man schon zwei Teilheaps hat und durch Schlüsselvergleiche mit linkem und rechtem Sohn ein einfaches Werkzeug in der Hand hat, mit dem man das Element in den korrekten Teilbaum versickern lassen kann. Dabei geht man wie folgt vor:

1. Zunächst wird die Gültigkeit des Index des aktuellen Elementes i überprüft. Ist die Forderung 0 <= i <= n-1 nicht erfüllt, endet die Ausführung der Funktion hier.
2. Es wird getestet, ob das aktuelle Element einen Sohn besitzt, d.h. ob es einen Index gibt, der die Bedingung 2i+1 <= n-1 erfüllt. Hat das Element einen Sohn, so wird dieser zum Vergleichselement und es wird mit Schritt 3 fortgefahren, ansonsten bricht die Funktion ab.
3. Es wird nun getestet, ob das Element auch noch einen zweiten Sohn besitzt, d.h die Bedingung 2i+2 <= n-1 erfüllt ist. Wenn ja, so wird von beiden Söhnen der größere (MaxHeap) ermittelt und zum neuen Vergleichselement. Ansonsten wird sofort Schritt 4 ausgeführt.
4. Es wird getestet, ob die Heapordnung verletzt ist. Wenn ja, so wird das aktuelle Element mit dem Vergleichselement getauscht, ansonsten kommt Schritt 5 zur Ausführung.
5. Das Vergleichselement wird zum aktuellen Element und es wird wieder zu Schritt 1 verwiesen.

Bei der Umsetzung dieses Algorithmus' kann man ein wenig einsparen, was zu folgender Funktion versickern() führt, welche das Element mit dem Index i im Baum versickern lässt:

```
void versickern(int a[], int n, int i){          //i=Index des zu versickernden El
  int j, h=a[i];
  if(i<0) return;                                //wenn i ungültig, dann abbrecher
  while(i<n/2){                                  //solange noch Söhne vorhanden si
    j=i+i+1;                                     //setze Vergleichselement j auf l
    if(j+1<n && a[j]<a[j+1]) j++;                //wenn rechter Sohn größer linkem
    if(h>=a[j]) break;                           //wenn Heap-Ordnung hergestellt,
    a[i]=a[j]; i=j;                              //Element im Heap nach unten vers
  }
  a[i]=h;                                        //Element an Endposition einordne
}
```

Beim Versickern eines Elementes in einem Binärbaum werden maximal log(n) Vertauschungen durchgeführt, was der Höhe des Baumes entspricht. Daher ergibt sich auch die Zeitkomplexität von *O(log(n))* für diese Methode.
Diese Operation reicht bereits aus, um den Sortieralgorithmus und die Konstruktion eines Heaps realisieren zu können, wodurch andere Heap-Methoden im Rahmen dieses Textes nicht weiter betrachtet werden müssen.

4.2.4 Aufbauen eines Heaps:

Bevor man überhaupt zur Sortierung übergehen kann, muss zunächst einmal aus einer gegebenen Folge ein Heap konstruiert werden. Der Erfinder von Heapsort, J.W.J. Williams, hat eine Methode mit einer Komplexität von *O(n*log(n))* angegeben. Wir wollen hier hingegen ein Verfahren betrachten, welches von R.W. Floyd entwickelt wurde und einen Heap mit *O(n)* in linearer Zeit konstruieren kann. Dieses Verfahren stützt sich auf den sukzessiven Aufbau von

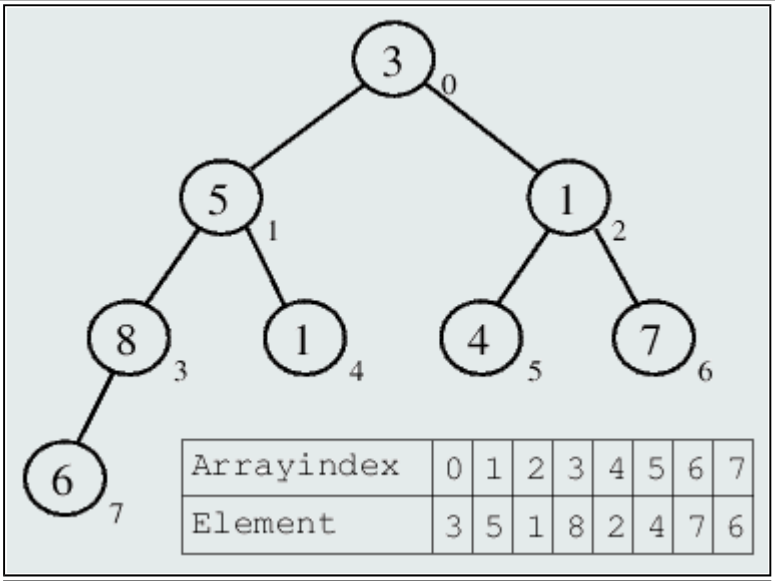
Teilheaps, von denen je zwei mit deren Vater zu einem Heap verschmolzen werden, indem für den Vater die Operation des Versickerns ausgeführt wird. Die Bedingung für das Versickern, dass die beiden Söhne bereits Heaps sind, ist für Teilheaps mit einem Element trivial erfüllt. Daher können je zwei dieser Elemente mit deren Vater zu einem Heap der Höhe 2 verschmolzen werden, diese zu Heaps der Höhe 3 usw., bis durch das Verschmelzen mit dem Versickern der Wurzel E_0 ein Gesamt-Heap erzeugt wird und die Konstruktion damit abgeschlossen ist.
Bei der programmiertechnischen Realisierung wird dieser Algorithmus wie folgt interpretiert:
Aus einer unsortierten Folge von n Schlüsseln erhält man eine Heapordnung, indem man für die Elemente n/2-1, n/2-2, ..., 0 die Funktion 'versickern()' ausführt.

Neu daran ist der Fakt, dass beim Element n/2-1 begonnen wird. Man muss ein wenig überlegen, um sich vom Funktionieren dieses Vorgehens überzeugen zu können. Dabei kann man leicht erkennen, dass die Blattebene eines vollständigen binären Baumes maximal (n+1)/2 Elemente besitzt. Z.B. hat ein vollständiger Baum mit 31 Elementen 16 Blätter. Will man in diesem Baum nun Heaps der Höhe 2 erstellen, so nimmt man die direkt über den Blättern liegende Knotenebene, deren letztes Element auf der Position n/2-1 liegt (klar: (n+1)/2 Elemente maximal in der Blattebene, also maximal n/2 innere Knoten). Diese Knoten lässt man versickern, geht dann in die höheren Ebenen, die durch einfache Dekrementierung des Arrayindex erreicht werden können und führt dort weiter das Versickern aus, bis die Wurzel erreicht ist. Dass die Wahl von n/2-1 nicht nur für maximal vollständige Binärbäume (wie gerade gezeigt) funktioniert, lässt sich schnell überlegen: entnimmt man zwei Elemente vom Ende des Arrays, so vermindert sich n/2 um nur ein Element - nämlich genau das, dessen Söhne man entnommen hat. Daher ist es korrekt, wenn man dieses Element aus dem Versickern-Prozess ausschließt.

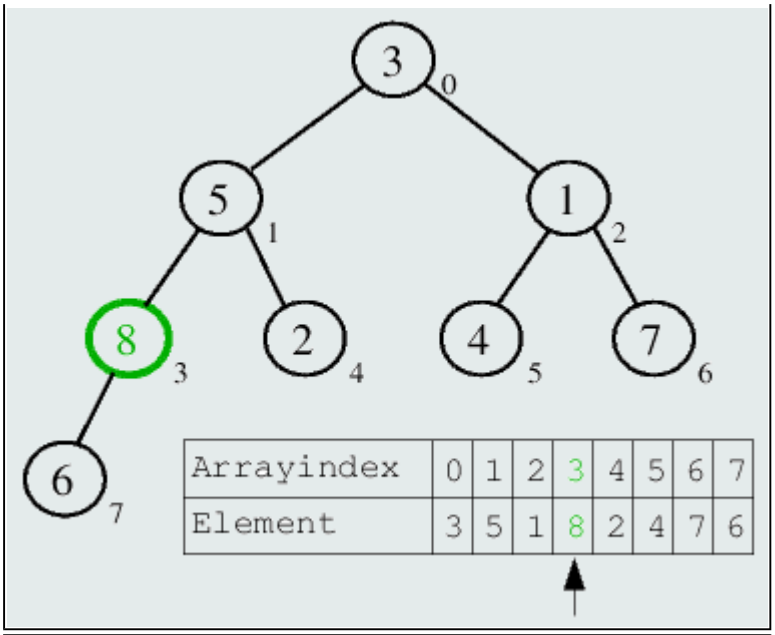
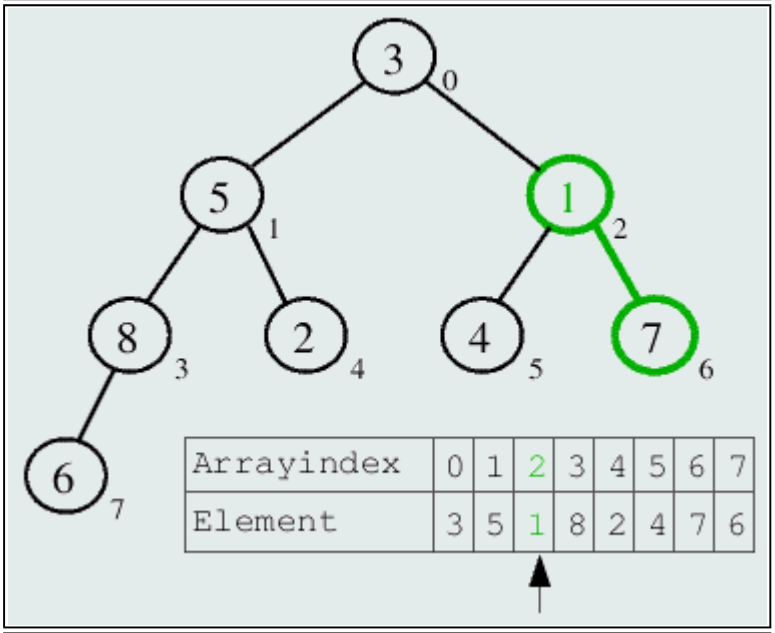
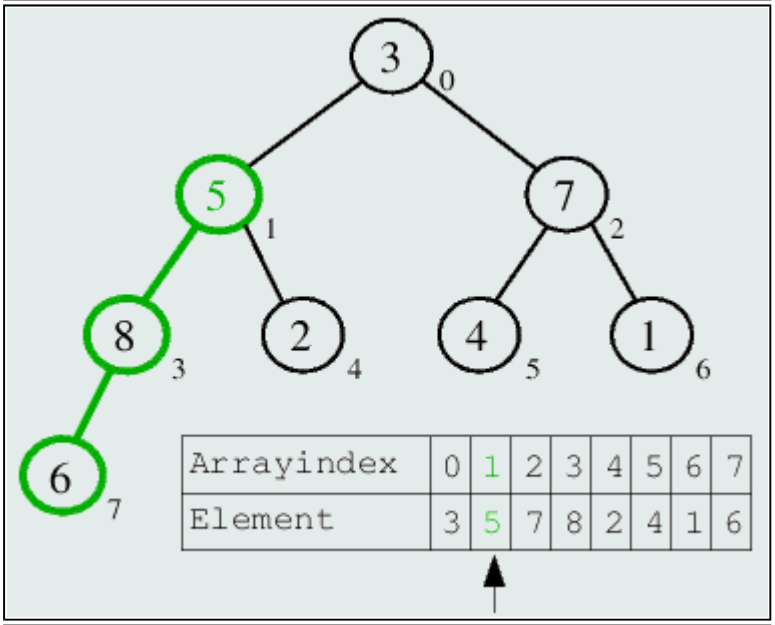
Auf Basis dieser Überlegungen kann man die Methode der Konstruktion eines Heaps nun wie folgt angeben:

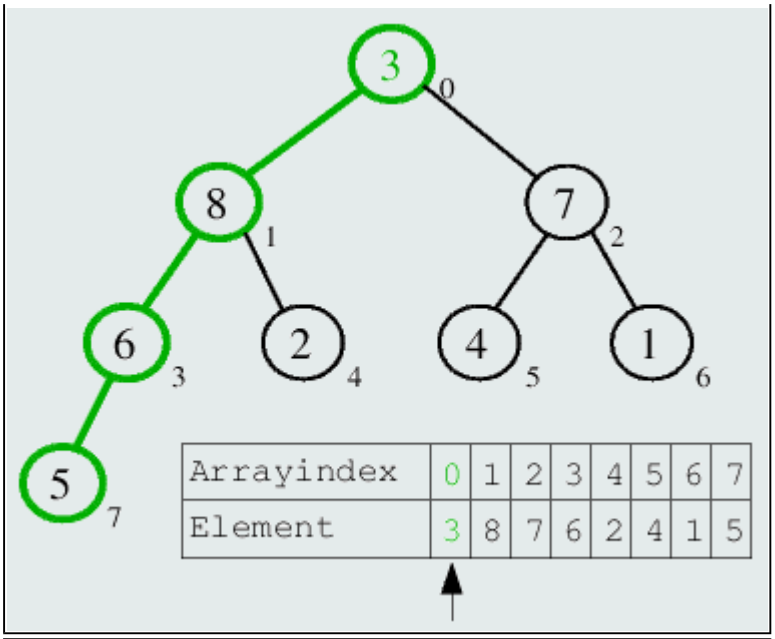
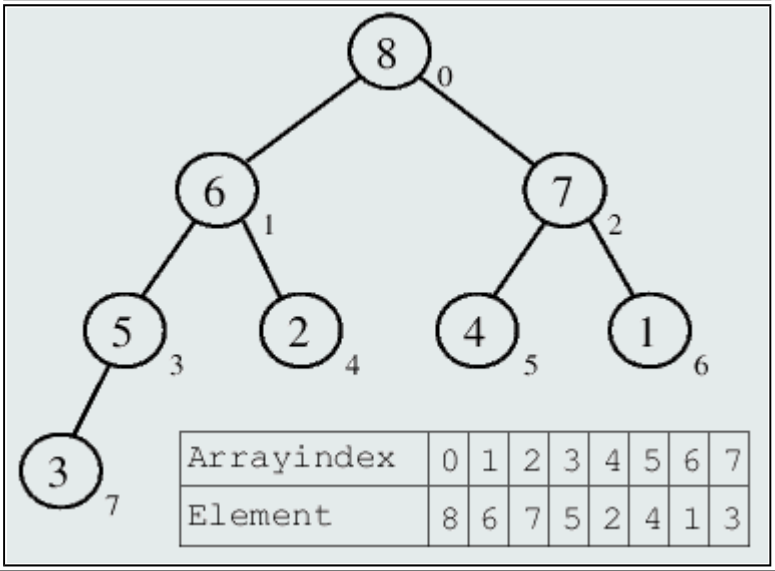
```
void konstruktion(int a[], int n){
  int i;
  for(i=n/2-1; i>=0; i--)
    versickern(a,n,i);
}
```

Folgendes Beispiel zeigt, wie diese Konstruktion schrittweise erfolgt:

<table>
<tr>
<td>1.) Ausgangssituation: die Schlüssel liegen durcheinander und fernab von jeder Heapordnung im Binärbaum (Array) vor.</td>
<td> 

Arrayindex	0	1	2	3	4	5	6	7
Element	3	5	1	8	2	4	7	6
</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</table>

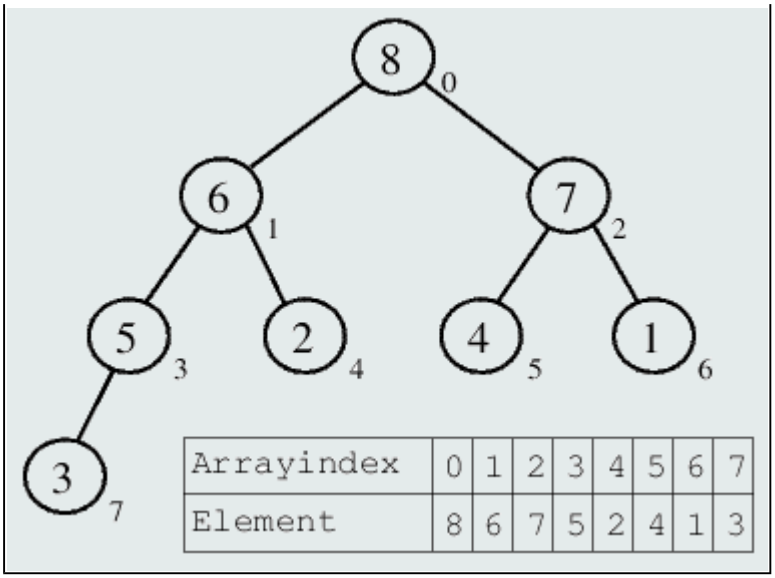
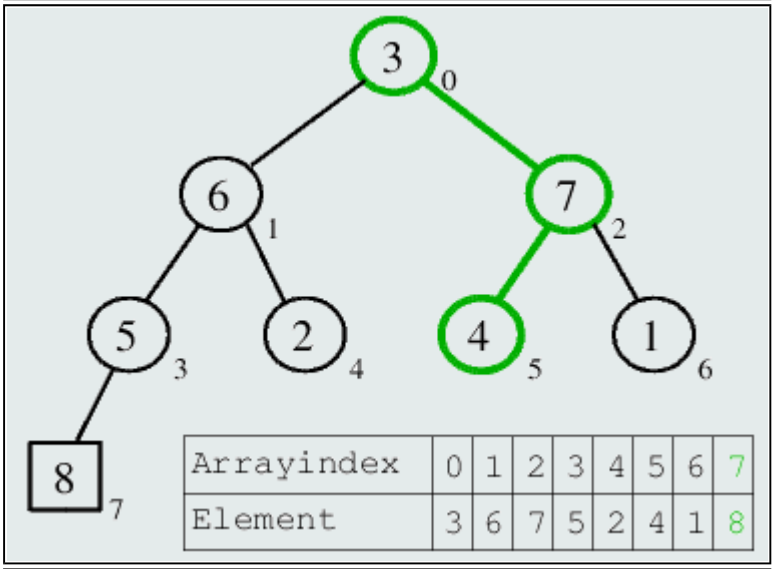
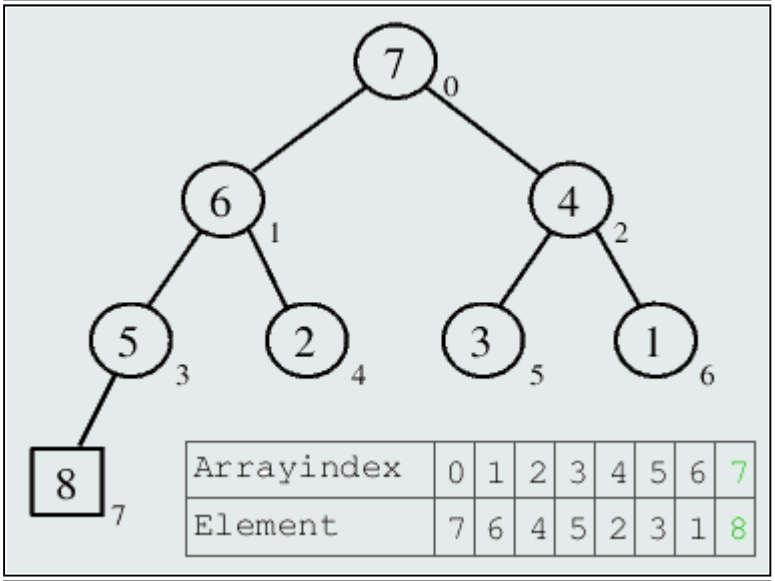
2.) Im Array sind n=8 Elemente vorhanden. n/2=4, also ist n/2-1=3 das Element, von welchem aus nach unten hin das Versickern ausgeführt wird. a[3]=8, mit diesem Element als Wurzel ist die Heap-Bedingung des darunter liegenden Teilbaumes erfüllt (8>=6), sodass nicht versickert werden muss.	
3.) Der Arrayzeiger wird dekrementiert und zeigt nun auf a [2]=1. Hier ist die Heap-Bedingung für den darunter liegenden Teilbaum ganz offensichtlich nicht erfüllt. a[2] hat die zwei Söhne a[5]=4 und a[6] =7. Da a[6]>a[5] ist, wird das aktuelle Element a[2] mit diesem getauscht, "versickert" im Baum. Da anschließend keine Söhne mehr vorhanden sind, bricht der Algorithmus ab und die 1 gilt als versickert.	
4.) Nach der Dekrementierung des Zeigers wird das Versickern für a[1] =5 ausgeführt, für welches die Heap-Bedingung ebenfalls verletzt ist. Sukzessive wird bei den Söhnen nach dem größeren Element gesucht und mit 5 getauscht. Da 8>2 ist, kommt die 5 zunächst auf deren Platz, um dann noch eine Position weiter über die 6 ans Ende des Arrays zu versickern.	

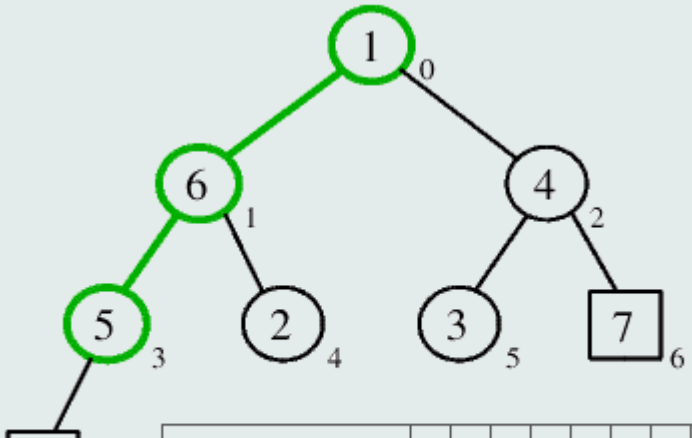
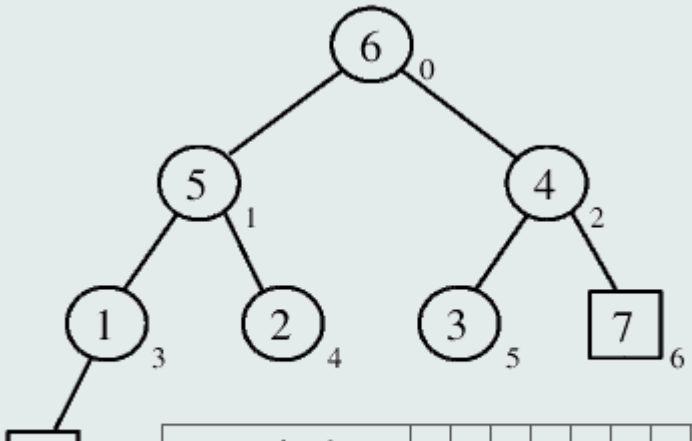
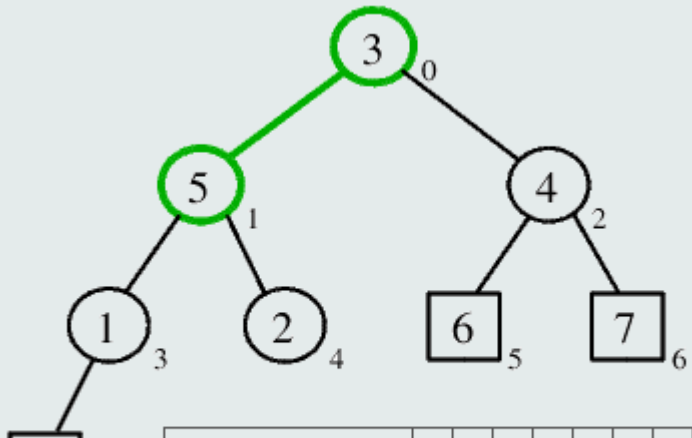
5.) Als letztes Element wird a[0]=3 betrachtet, für welches die Heap-Bedingung natürlich auch verletzt ist. Nach dem Finden der größten Söhne gelangt die 3 über a[1]=8, a[3]=6 und a[7]=5 ans Ende des Arrays.	
6.) Nachdem das erste Arrayelement versickert ist, gilt der Heap als konstruiert. Schön zu sehen, dass man nun auf das größte Arrayelement über a[0] zugreifen kann und dies nur einen einzigen Schritt erfordert. Die Heap-Bedingung ist für alle Unterbäume erfüllt, sodass es sich bei dem dargestellten Binärbaum auch wirklich um einen Heap handelt.	

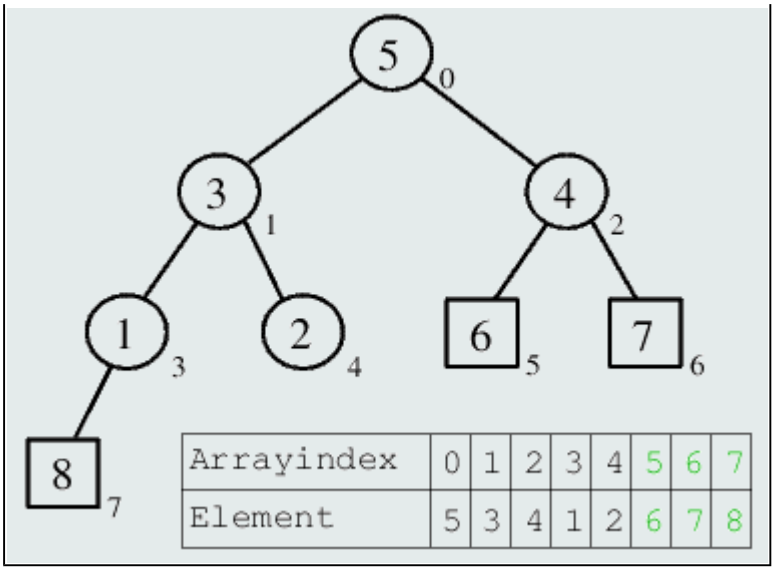
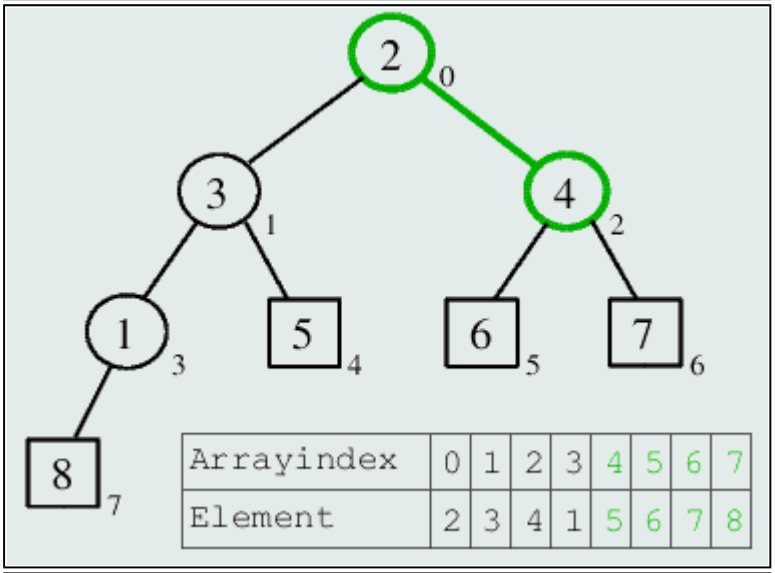
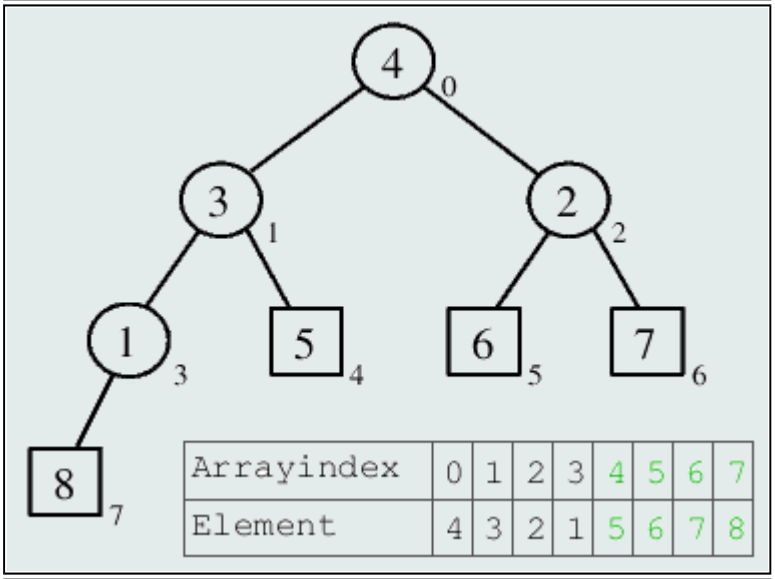
4.3 Der Heapsort-Algorithmus

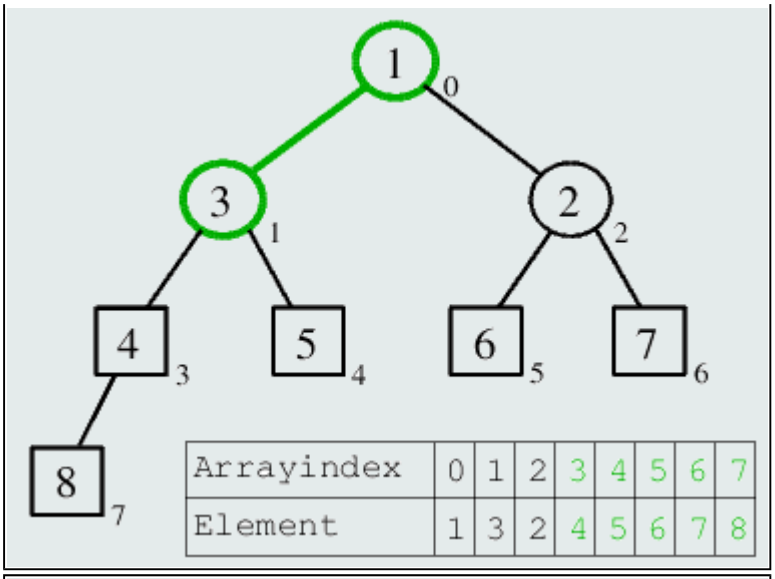
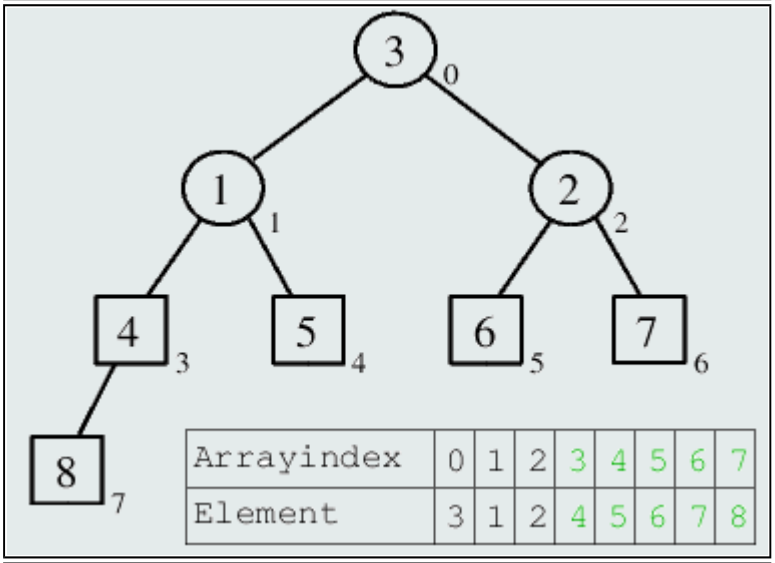
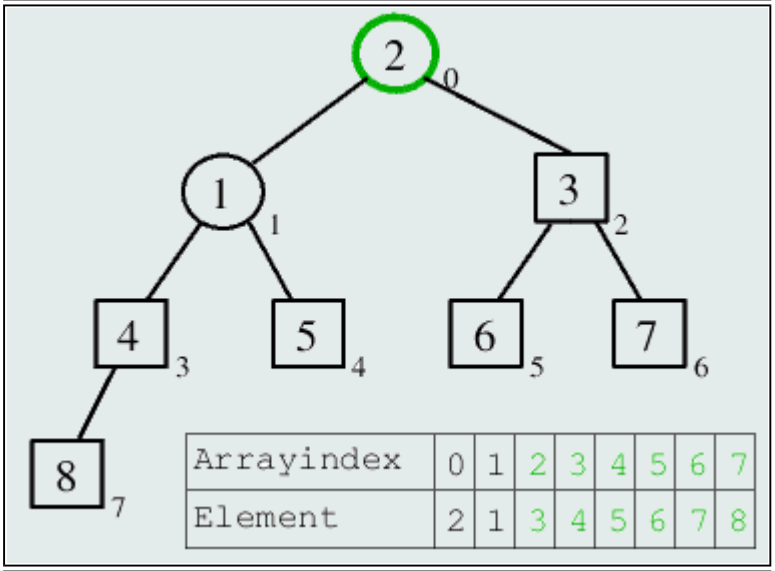
Mit dem (Max-) Heap als Datenstruktur und der damit verbundenen Operation des Versickerns lässt sich leicht ein effektives Sortierverfahren konstruieren. Die Vorgehensweise dabei ist so logisch wie einfach: entferne zunächst das größte Element auf Position 0, setze dafür das letzte Arrayelement und lass dieses versickern - wiederhole diesen Vorgang bis kein Element mehr im Array vorhanden ist.
Direkt damit verbunden ist auch der Vorteil der Verwendung von MaxHeaps: anstatt für die Speicherung der sortierten Folge ein zusätzliches Array mit n Elementen zu verwenden, können das Element auf der Position 0 und das letzte noch zu betrachtende Heap-Element vertauscht werden, wodurch das momentan größte Element auf seinen endgültigen Platz im sortierten Array gelangt. Die sortierte Folge wird also von hinten nach vorn durch sukzessive Entnahme des ersten (größten) Elementes aufgebaut, wodurch das Gesamt-Array zwar seine Heap-Eigenschaft verliert, dies uns bei der Zielstellung der Sortierung einer Folge aber nicht stören soll.

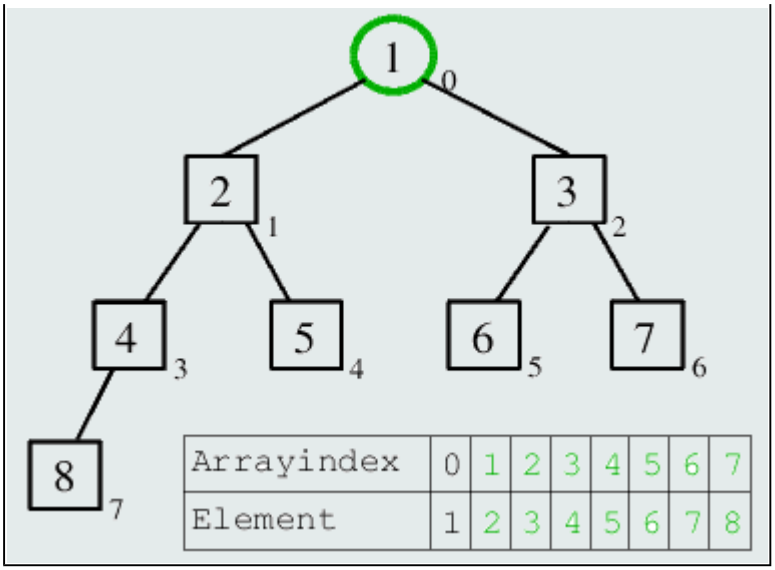
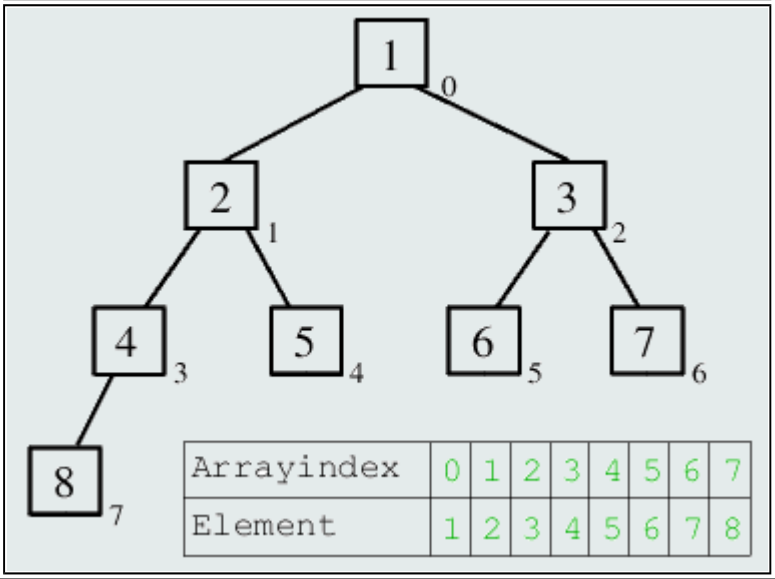
Folgendes Beispiel soll die Arbeitsweise von Heapsort illustrieren. Verwendet wurde dabei der in 4.2.4 erzeugte MaxHeap, sodass wir uns den Schritt der Konstruktion eines Heaps sparen können.

1.) Ausgangssituation: der oben konstruierte MaxHeap.	
2.1) 8 wird an das Ende des Arrays eingefügt, das dortige Element 3 wird dafür als neue Wurzel gesetzt, was ein Versickern entlang des grünen Pfades zur Folge hat.	
2.2) Nach dem Versickern der 3 steht die 7 als größtes Element auf der Wurzel-Position.	

3.1) 7 wird vor die bereits fest gelegte Position der 8 eingefügt, mit der dortigen 1 getauscht, welche anschließend versickert.	 Arrayindex 0 1 2 3 4 5 6 7 Element 1 6 4 5 2 3 7 8
3.2) Nach dem Versickern der 1 steht die 6 auf der Wurzel.	 Arrayindex 0 1 2 3 4 5 6 7 Element 6 5 4 1 2 3 7 8
4.1) 6 wird ans Ende des Rest-Heaps verschoben, die 3 von dort kommt in die Wurzel und versickert eine Position nach links, bis der Rest-Binärbaum wieder einen Heap darstellt.	 Arrayindex 0 1 2 3 4 5 6 7 Element 3 5 4 1 2 6 7 8

4.2) Nach dem Versickern der 3 steht das nächstgrößte Element, die 5, in der Wurzel.	 Arrayindex 0 1 2 3 4 5 6 7 Element 5 3 4 1 2 6 7 8
5.1) 5 wird mit der 2 ans Heap-Ende getauscht, erhält dadurch seinen festen Platz in der sortierten Folge. Die 2 muss nun versickern und bewegt sich dabei eine Position nach rechts, wo das Ende des Rest-Heaps erreicht ist.	 Arrayindex 0 1 2 3 4 5 6 7 Element 2 3 4 1 5 6 7 8
5.2) Nachdem die 2 versickert ist, steht mit 4 das nächst kleinere Element in der Wurzel.	 Arrayindex 0 1 2 3 4 5 6 7 Element 4 3 2 1 5 6 7 8

6.1) 4 kommt ans Heap-Ende, wodurch die 1 in die Wurzel gelangt und eine Position nach links versickert, bevor die Ordnung des Rest-Heaps wieder hergestellt ist.	 Arrayindex 0 1 2 3 4 5 6 7 Element 1 3 2 4 5 6 7 8
6.2) Nach dem Versickern der 1 steht mit der 3 das nächst größere Element zum Einordnen bereit.	 Arrayindex 0 1 2 3 4 5 6 7 Element 3 1 2 4 5 6 7 8
7.) Die 3 wird eingeordnet, die 2, welche dadurch in die Wurzel kommt, verletzt die Heap-Ordnung nicht, wodurch nicht versickert werden muss.	 Arrayindex 0 1 2 3 4 5 6 7 Element 2 1 3 4 5 6 7 8
8.) Die 2 kommt ans Ende des aus nur noch 2 Elementen bestehenden	

Rest-Heaps. Die 1 kommt in die Wurzel. Da der Heap aus nur noch einem Element (der Wurzel) besteht, erfolgt kein Versickern dieser.	
9.) Die finale Darstellung: alle Elemente wurden sortiert von hinten nach vorn im Array eingefügt, was nun in aufsteigender Reihenfolge vorliegt. Klar, dass dies keinen Heap mehr darstellt - das soll uns aber nicht weiter stören.	

4.4 Zur programmiertechnischen Realisierung

Der C-Code für Heapsort an sich ist nicht schwer: zunächst wird aus der beliebigen Folge ein Heap aufgebaut, was bei der sofortigen Wahl des Heaps als Datenstruktur allerdings entfällt. Dann folgt die Soriterung: man tauscht n-mal das erste Element mit dem gerade letzten und stellt dann die Heap-Ordnung wieder her, indem die Funktion versickern(a,i,0) ausgeführt wird, wie es folgender Code zeigt:

```
void heapsort(int a[], int n){
  int i, tmp;
  konstruktion(a,n);
  for(i=n-1; i>=0; i--){
    tmp=a[0]; a[0]=a[i]; a[i]=tmp;
    versickern(a,i,0);
  }
}
```

4.5 Laufzeitbetrachtungen

Die Zeitkomplexität für den Aufbau des Heaps beträgt wie schon beschrieben O(n). versickern () wird in O(log(n)) ausgeführt - ganze n-mal. Dies führt zu einer Gesamtlaufzeit von O(n)

+n*O(log(n)), was nach den allgemeinen Rechenregeln der O-Notation zu max{O(n),O(n*log(n))} und somit zu *O(n*log(n))* vereinfacht werden kann.
Heapsort ist nicht sonderlich schnell, in praxi benötigt o.a. Implementation ein Vielfaches der Zeit von Quicksort und ist in etwa vergleichbar mit dem grundlegenden Straight-Radix-Sort-Algorithmus. Seine Relevanz behält es trotzdem, zumal es wie Mergesort eine worst-case-Zeitkomplexität von O(n*log(n)) besitzt und wie etwa Quicksort keinen zusätzlichen Speicher benötigt.

[Zurück zum Inhaltsverzeichnis]

5. Radixsort

[Zurück zum Inhaltsverzeichnis]

5.1 Hintergrund

Die Sortierverfahren, welche wir in den bisherigen Kapiteln kennen gelernt haben, stützen sich allesamt auf Schlüsselvergleiche zum Sortieren der Daten, d.h. die Schlüssel als Ganzes werden miteinander verglichen und bei Bedarf getauscht. Die Sortierverfahren, die jetzt behandelt werden sollen, nutzen hingegen die arithmetischen Eigenschaften der Schlüssel aus. Dabei wird angenommen, dass es sich bei den Schlüsseln um Wörter aus einem *m*-adischen Alphabet handelt (z.B. m=10 für Dezimalzahlen, m=2 für Dualzahlen, m=26 für Wörter über dem lateinischen Alphabet), man die Schlüssel also als *m*-adische Zahlen auffassen kann. Daher nennt man *m* auch die *Wurzel* (engl./lat.: *radix*) der Schlüsseldarstellung und die darauf aufbauenden Sortieralgorithmen *Radixsort*-Verfahren.
Diese betrachten die einzelnen Stellen der m-adischen Schlüssel. Bei Dezimalzahlen würde also jede Ziffer, bei Dualzahlen jedes Bit und bei Zeichenketten jedes Zeichen zum Sortieren heran gezogen werden und nicht wie bei anderen Sortierverfahren die Schlüssel an sich. Daher bezeichnet man die Radixsort's auch als *digitale Sortieralgorithmen*.
Anwendung finden die Radixsort-Verfahren z.B. dort, wo nur nach bestimmten Schlüsselstellen sortiert werden soll - hier wären andere Sortiermethoden machtlos. Anders bei Radixsort: eine mögliche Variante wäre es, eine Bitmaske zu erstellen, die an denjenigen Schlüsselstellen, nach denen sortiert werden soll, eine 1 hat und dann nur diese Stellen in die Sortierung einzubeziehen.
Die hier vorgestellten Verfahren sollen der Einfachheit halber wie in der Computertechnik üblich auf das duale Zahlensystem zurückgreifen, also die einzelnen Bits als Schlüsselstellen betrachten und darauf ihre Sortierungen aufbauen. Dies ist auch sinnvoll, da eigentlich alles, was in einem Digitalrechner dargestellt wird, in Binärzahlen umgewandelt werden kann und moderne Programmiersprachen einfache Operatoren und Möglichkeiten bereitstellen, um auf die einzelnen Binärziffern einer Variablen zugreifen zu können.
Zunächst soll in diesem Kapitel auf *Radix Exchange Sort* eingegangen werden, welches Quicksort sehr ähnelt, die Schlüssel von links nach rechts bitweise betrachtet und sukzessive nach 0/1 in 2 Gruppen unterteilt, wodurch eine endgültige Sortierung erreicht wird.
Als Zwischenschritt soll dann eine Erläuterung von *Bucketsort* erfolgen, welches verstanden werden muss für das darauf aufbauende *Straight Radix Sort* sowie ein lineares Verfahren als Verallgemeinerung dieser digitalen Sortiermethode.

5.2 Bit-Operationen

Da sich die hier vorgestellten Sortierverfahren auf Vergleiche von Bitstellen stützen, ist es unersetzlich, einzelne bzw. mehrere Bits aus einem Schlüssel extrahieren zu können.
Die Programmiersprache C stellt dafür einige Operatoren bereit: mit x<<y und x>>y werden die Bits der Variablen x um y Stellen nach links bzw. rechts verschoben, x&y verknüpft x bitweise UND mit y - dies genügt uns schon, um die beiden benötigten Funktionen zu definieren.

Die erste Funktion getbit() wird von *Radix Exchange Sort* und *Straight Radix Sort* verwendet, um den Wert *eines* Bits an einer bestimmten Stelle zurückzuliefern:

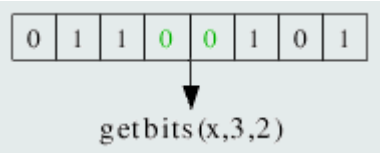
```
inline unsigned getbit(unsigned x, int k){ return (x>>k)&1; }
```

k gibt dabei die von rechts gesehene Stelle an, für welche der Bitwert geliefert werden soll, `getbit(x,0)` würde z.B. den Wert des am weitesten rechts gelegenen Bits zurückgeben.

Die Verallgemeinerung von *Straight Radix Sort* mit *Bucketsort* greift auf die folgende Funktion getbits() zurück, welche den Wert *mehrerer* Bits zurück liefert:

```
inline unsigned getbits(unsigned x, int k, int j){ return (x>>k)&~(~0<<j); }
```

Diese Funktion liefert die j Bits zurück, die k Bits von rechts in x angeordnet sind. Kurz zur Erläuterung: ~(~0<<j) stellt eine Maske dar, die aus Einsen auf den am weitesten rechts stehenden j Bitstellen und Nullen auf den übrigen Positionen besteht. Eine logische UND-Verknüpfung dieser Maske mit dem um k Bits nach rechts verschobenen Schlüssel führt dann dazu, dass duch die Nullen in der Maske diese Bits auf jeden Fall 0 sind und die j rechten Bits durch die Einsen in der Maske den Wert des verschobenen Schlüssels erhalten und zurück geliefert werden.
Folgendes Bild illustriert einen Aufruf von getbits() für den gegebenen Schlüssel. k=3 und j=2, sodass die 2 Bits zurück gegeben werden, welche 3 Bits von rechts im Schlüssel angeordnet sind.



5.3 Radix Exchange Sort

Bei diesem, auch als *digitales Austausch-Sortierverfahren* bekannten Algorithmus, werden die Bits in den Schlüsseln sukzessive von links nach rechts betrachtet. Es ist leicht ersichtlich, dass alle Schlüssel mit führender 0 kleiner sind als Schlüssel mit führender 1. Die Schlüssel werden demnach so umgeordnet, dass zunächst die gesamte Datei betrachtet und nach der am weitesten links stehenden Bitstelle sortiert wird. Das heißt, die Datei wird in 2 Gruppen unterteilt: die linke Gruppe enthält alle Schlüssel, deren gerade betrachtete Bitstelle 0 ist, die rechte enthält die Schlüssel mit den 1-Bits. Rekursiv werden diese beiden Gruppen nun nach dem gleichen Algorithmus geteilt, wobei pro Rekursionsschicht immer die nächste Bitstelle betrachtet wird. Die Rekursion erfolgt solange, bis entweder alle Bitstellen betrachtet wurden oder aber alle Gruppen nur noch aus einem Element bestehen - die Folge gilt dann als eindeutig sortiert.

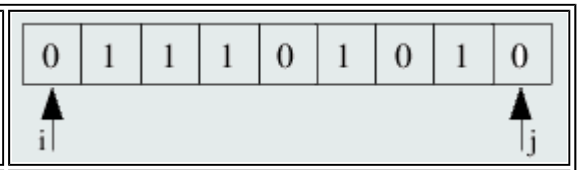
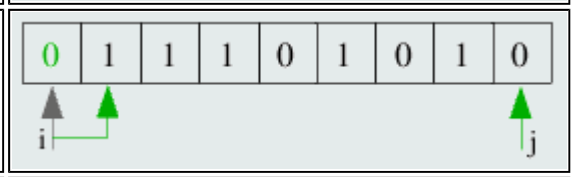
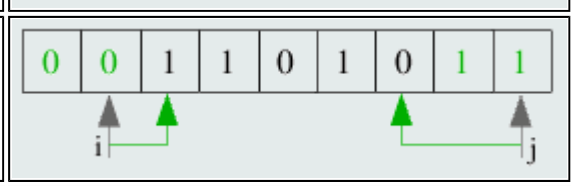
Radix Exchange Sort ist deutlich erkennbar als "Teile-und-Herrsche"-Algorithmus und kommt dem ebenfalls auf diesem Prinzip beruhenden Quicksort sehr nahe. Wie bei Quicksort wird die Datei erst in 2 Gruppen geteilt, was aus der Verwendung von binären Schlüsselstellen resultiert (2 Gruppen für 0- und 1-Bits). Daraufhin erfolgen getrennt für die linke und rechte Teildatei die beiden rekursiven Aufrufe.
Und so ist die programmiertechnische Realisierung auch erfreulich ähnlich zu der von Quicksort. Wie bei diesem wird das Feld mit zwei Zeigern durchlaufen, die Ganzzahlen auf Arrayindizes darstellen. Der linke Zeiger startet beim linken Rand der Folge nach rechts und hält an, wenn ein Schlüssel gefunden wird, dessen gerade betrachtetes Bit 1 ist. Der rechte Zeiger steht zu Beginn auf dem Ende der Folge und durchläuft diese nach links, bis ein Schlüssel mit einem 0-Bit an der entsprechenden Bitstelle gefunden wird. In diesem Fall werden die Elemente von linkem und rechtem Zeiger miteinander vertauscht. Diese Prozedur läuft solange, bis beide Zeiger aufeinander treffen und hat zur Folge, dass jetzt in der linken Teildatei nur Elemente mit 0-Bits und in der rechten Datei nur Elemente mit 1-Bits an der relevanten Bitstelle enthalten sind. Im Gegensatz zu Quicksort, bei dem das Trennelement willkürlich gewählt werden konnte, handelt es sich bei diesem digitalen Sortierverfahren stets um die Zahl 2^b, wenn b die soeben betrachtete Bitstelle ist. Da nicht gesagt ist, dass diese Zahl in der Schlüsselfolge enthalten ist, kann es auch keine Garantie dafür geben, dass wie bei Quicksort durch einen rekursiven Aufruf ein Element seinen endgültigen Platz in der Datei erhält und nicht weiter betrachtet werden muss.
Durch die Ähnlichkeiten mit Quicksort ist es auch nicht verwunderlich, dass der zugehörige Code frapierende Übereinstimmungen aufweist. Hier eine mögliche Variante, deren Aufruf mit radixexchange(a,0,n-1,8*sizeof(int)); für n Elemente und Ganzzahlen (*int*'s) als Schlüssel erfolgt:

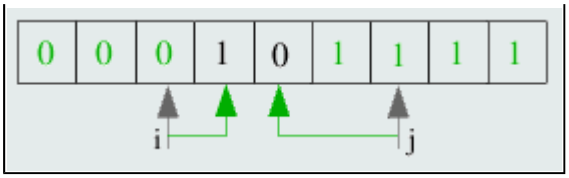
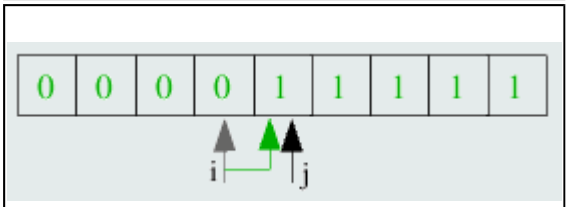
```
inline unsigned getbit(unsigned x, int k){ return (x>>k)&;1; }

void radixexchange(int a[], int l, int r, int b){       //b=von rechts zu betracht
    b--;
```

```
    int t, i, j;
    if(r>l && b>=0){                                    //Dateilänge>1 && noch Bi
        i=l; j=r;                                       //i=linker Rand, j=rechte
        while(j!=i){                                    //solange linker nicht re
            while(getbit(a[i],b)==0 && i<j) i++;        //halte an, wenn 0 von li
            while(getbit(a[j],b)!=0 && j>i) j--;        //halte an, wenn 1 von re
            t=a[i]; a[i]=a[j]; a[j]=t;                  //tausche beide Elemente
        }
        if(getbit(a[r],b)==0) j++;                      //wenn Datei komplett aus
        radixexchange(a,l,j-1,b);                       //Aufruf für linke Teilda
        radixexchange(a,j,r,b);                         //Aufruf für rechte Teilc
    }
}
```

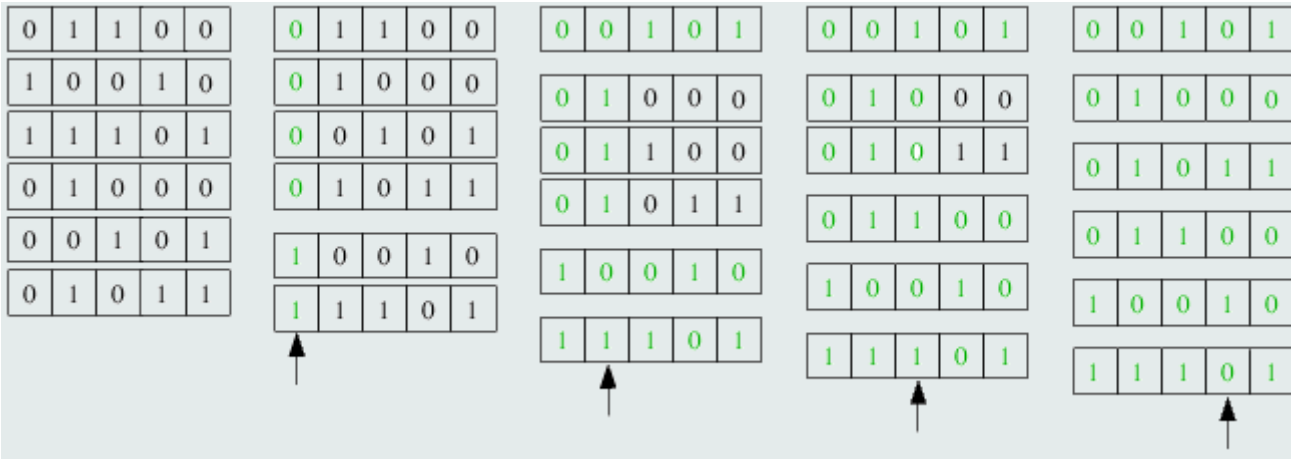
Wieder ein paar Worte zur Erklärung des Codes: in der Parameterliste gibt b die von rechts aus gesehene zu betrachende Bitstelle der momentan zwischen l und r befindlichen Datei an. Der Code wird nur ausgeführt, wenn die Länge der aktuellen Datei größer 1 ist und wenn das eben betrachtete Bit nicht aus der Folge rechts herausläuft. Dann werden die beiden Zeiger auf den linken und rechten Rand der Folge gesetzt und die Schleife betreten, die ein Aufteilen in die beiden Teildateien bewirkt. Hier wird zunächst der linke Zeiger solange inkrementiert, bis er auf eine 1 stößt oder auf den rechten Zeiger trifft. Im Gegensatz zu Quicksort ist der zusätzliche Zeigertest hier nötig, da nur ein Bit untersucht wird und man sich nicht auf Marken verlassen kann, um das Durchsuchen mit den Zeigern abzubrechen. Das Gleiche gilt für die zweite innere Schleife, die dann stoppt, wenn ein 0-Bit von rechts gefunden wurde. Sollte dies geschehen sein, so werden die beiden Elemente miteinander vertauscht. Dies ist auch der Fall, wenn die inneren Schleifen aufgrund eines Aufeinandertreffens der beiden Zeiger verlassen worden sind - tut aber nichts zur Sache, in diesem Fall würde nur das Element, bei welchem die Zeiger aufeinander getroffen sind, mit sich selber vertauscht werden, was keine Auswirkungen hat. Sollten die beiden Zeiger auf dem selben Element stehen, so ist eine Teilung der Datei fast abgeschlossen. Links stehen jetzt alle Elemente mit 0 im b-ten Bit, rechts alle Elemente mit einer 1 an der selbigen Stelle. Das Element a[i] ist jetzt das am weitesten links stehende aus der rechten Teildatei, hat also eine 1 an der b-ten Bitstelle. Dies gilt nicht, wenn die Datei an dieser Bitstelle komplett aus Nullen bestanden hat. Der Test direkt nach der äußeren Schleife berücksichtigt diesen Fall und setzt j eine Position weiter, um den ersten rekursiven Aufruf die gesamte Datei berücksichtigen zu lassen.
Da die Datei jetzt geteilt ist, muss die Spaltung nur noch abgeschlossen werden, indem wie bei Quicksort die Rekursionen erfolgen. Kehren diese zurück, so gilt die aktuelle Teildatei als sortiert.

Folgendes Beispiel verdeutlicht einen Durchlauf der Funktion ohne die rekursiven Aufrufe. Dargestellt ist die jeweils b-te Bitstelle der 9 vorhandenen Schlüssel.

1.) Die Ausgangssituation: zu sehen ist das jeweils b-te Bit eines Schlüssels, die beiden Zeiger i und j sind schon initialisiert und zeigen auf den linken sowie rechten Rand der Folge.	
2.1) i findet eine 1 auf der nächsten Position, j steht beteits auf einer 0. Die beiden Schleifen brechen deswegen ab und die beiden Schlüssel werden miteinander vertauscht.	
2.2) Eine Position weiter findet i wieder eine 1 vor, j muss zwei Positionen nach links gerückt werden, bis es eine 0 vorfindet. Die beiden Elemente werden vertauscht und die Zeiger rücken weiter.	
2.3) Wieder muss i nur eine Position weiter und j zwei Positionen zurück gesetzt werden. Nach dem	

anschließenden Tausch ist die Sortierung der Datei nach dieser Bitstelle fast abgeschlossen.	
3.) i wird noch eine Position weiter gesetzt und verharrt dort, da das darauf stehende Bit eine 1 ist. j hat dieselbe Position wie i, wodurch die entsprechende Schleife sofort abbricht. Nach dem harmlosen Tausch des Elementes mit sich selbst wird die äußere Schleife beendet und die Datei ist nach dieser Bitstelle sortiert.	

Nachfolgend noch ein kurzes Beispiel, wie ein Testdatenfeld durch den Algorithmus durchlaufen und dabei sortiert wird.



Grüne Bitstellen bedeuten, dass diese vom Algorithmus nicht mehr betrachtet werden. Größere Abstände zwischen den Testdatensätzen sollen die rekursive Abspaltung in 2 Gruppen verdeutlichen. Der Pfeil unter einem Bild soll das derzeit betrachtete Bit b darstellen.
Das erste Bild zeigt die Ausgangssituation: die 6 Datensätze, deren Schlüssel aus je 5 Bits bestehen, liegen unsortiert vor.
Im zweiten Bild wird das erste Bit betrachtet und die Daten so umverteilt, dass 2 Gruppen mit 0- und 1-Bits an dieser ersten Dualziffer entstehen.
Im dritten Bild werden die beiden entstandenen Gruppen getrennt voneinander nach dem 2. Bit von links sortiert. In der ersten Gruppe gibt es gleich eine Abspaltung von einem Element, da dieses das einzige mit einer 0 an dieser Bitstelle ist. Dieses wird aus der weiteren Betrachtung ausgeschlossen, da es eindeutig an seiner endgültigen Position eingeordnet wurde. Auch die zweite Teildatei aus dem vorherigen Bild kann in diesem Schritt eindeutig sortiert werden, da sie aus nur zwei Elementen bestand und diese an der 2. Bitstelle unterschiedliche Werte haben.
In Bild vier werden die 3 übrig gebliebenen Elemente sortiert. Dabei kommt es wieder zu einer eindeutigen Abspaltung eines Elementes und somit bleiben nur noch 2 zu betrachtende Schlüssel übrig.
Das fünfte Bild zeigt die eindeutige Einordnung der letzten zwei Elemente, da die sich in der 4. Bitstelle unterscheiden. Da alle Elemente einen eindeutigen Platz in der Folge bekommen konnten, kann diese nun als sortiert angesehen werden, wozu noch nicht einmal alle Bitstellen betrachtet werden mussten.

Noch ein paar Gedanken zur Effizienz von Radix Exchange Sort: idealerweise würde natürlich wie bei Quicksort eine Datei genau in der Mitte in 2 Teildateien getrennt werden. Das Problem ist, dass durch den Algorithmus auch führende Nullen mit betrachtet werden und diese machen oftmals den halben Schlüssel aus. Erst in den niederwertigeren Bits entscheidet sich dann die endgültige Sortierung der Datei - so kommt es auch häufig vor, dass bei den hohen Bits nur kleine Teildateien abgesprengt werden und somit der Algorithmus nicht effizient arbeiten kann. Laufzeithemmend kommt hinzu, dass bei gleichen Schlüsseln alle Bits dieser Schlüssel betrachtet werden, was zusätzlichen Rechenaufwand erfordert - daher funktioniert Radix

Exchange Sort für Dateien mit vielen gleichen Schlüsseln nicht gut. Bei zufälligen Bits in den Schlüsseln arbeitet der Algorithmus allerdings ähnlich gut wie Quicksort, welches hingegen besser an weniger zufällige Situationen angepasst werden kann und im Allgemeinen trotzdem schneller ist.
Zu guter Letzt bleibt noch zu erwähnen, dass sich Radix Exchange Sort genauso wie Quicksort verbessern lässt, indem z.B. die Rekursion beseitigt wird oder kleine Teildateien mittels Insertion Sort behandelt werden.

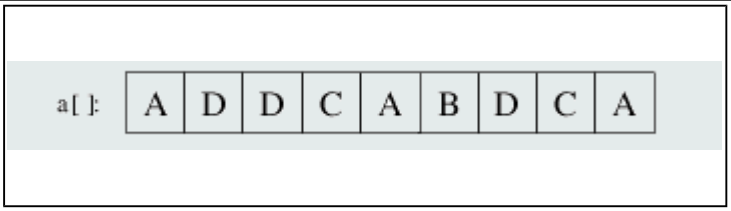
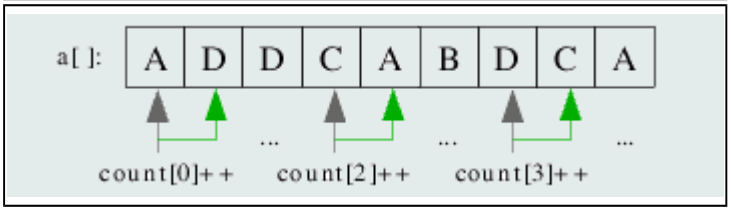
5.4 Bucketsort

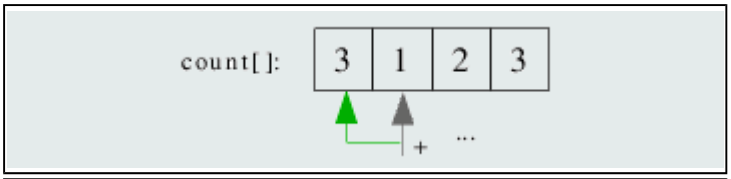
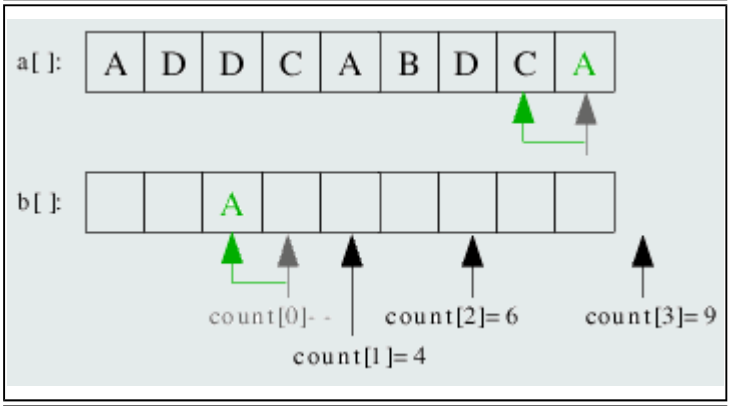
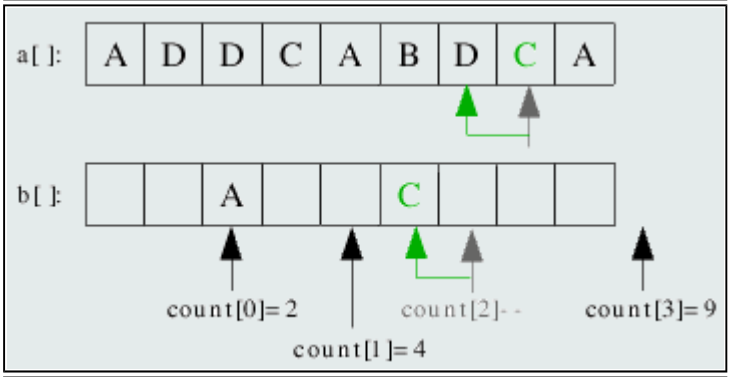
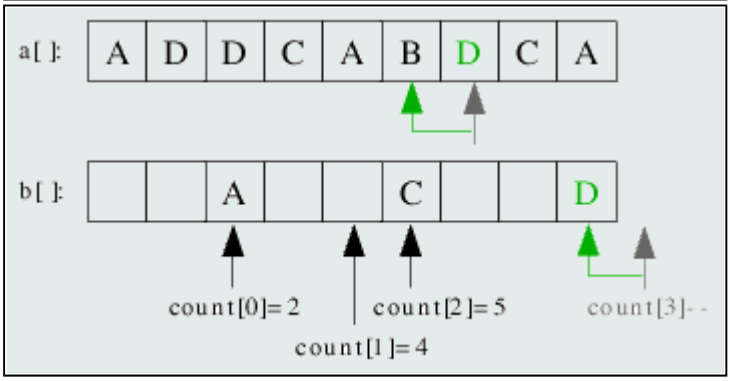
Als Zwischenschritt will ich hier eine stabile Sortiermethode betrachten, die wir für die nachfolgend dargestellten Sortierverfahren benötigen, welche aber nicht auf der binären Darstellung von Schlüsseln beruht. Beim *Bucketsort* (auch *distribution counting*/*Verteilungszählen* oder *Sortieren durch Fachverteilung* genannt) handelt es sich sogar um einen quasi linearen Sortieralgorithmus, wobei man aufpassen muss, da linear nicht unbedingt heißt, dass es auch schneller arbeitet als z.B. Quicksort.
Effizient funktioniert dieses Verfahren vor allem dann, wenn eine Datei vorliegt, in der die Anzahl unterschiedlicher Schlüssel begrenzt ist, d.h. viele identische Schlüssel vorliegen. Wenn bei n Datensätzen M unterschiedliche Schlüssel existieren und M hinreichend klein ist, kann man Bucketsort anwenden, welches in diesem Fall effektiver ist als z.B. Quicksort. Die Idee besteht darin, dass man zunächst die Datei durchläuft, um die Anzahl der Schlüssel zu jedem der M Werte zu ermitteln. Das Ergebnis dieser Zählung speichert man in einem Array count[], welches natürlich M Elemente aufnehmen muss und dann bei einem zweiten Durchlauf durch die Datei verwendet wird, um die endgültigen Positionen der Schlüssel in der sortierten Datei zu bestimmen und diese in einem Hilfsarray b[] (welches von gleicher Dimension wie das zu sortierende Array a[] sein muss) abzuspeichern. Anschließend muss aus b zurück nach a kopiert werden, will man das Ergebnis hier stehen haben.
Folgender Codeausschnitt realisiert genau das:

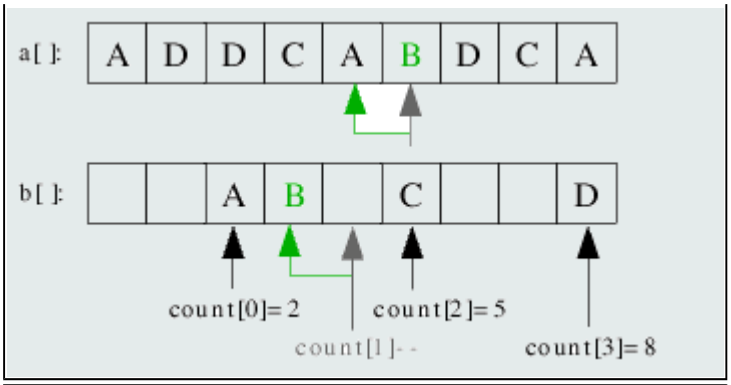
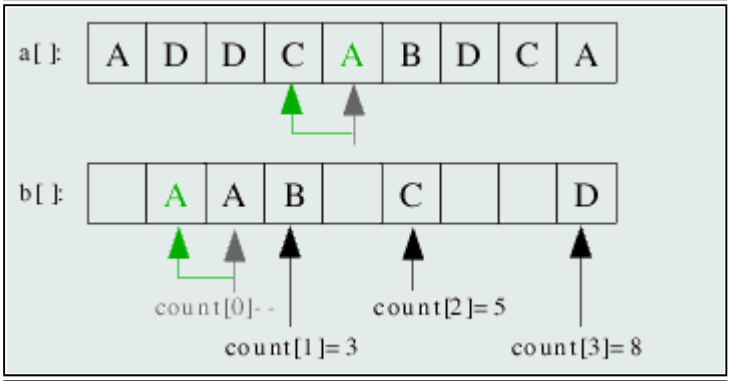
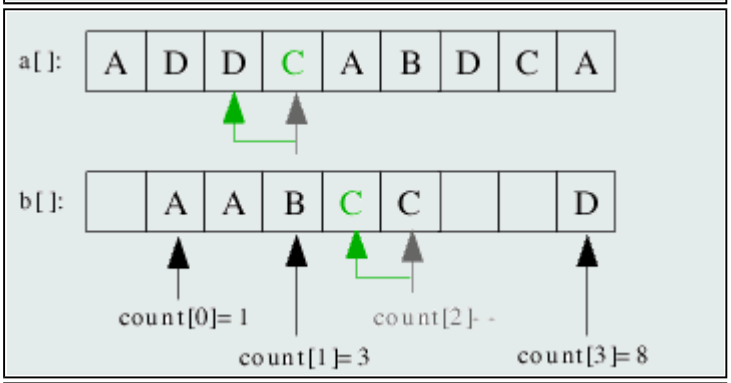
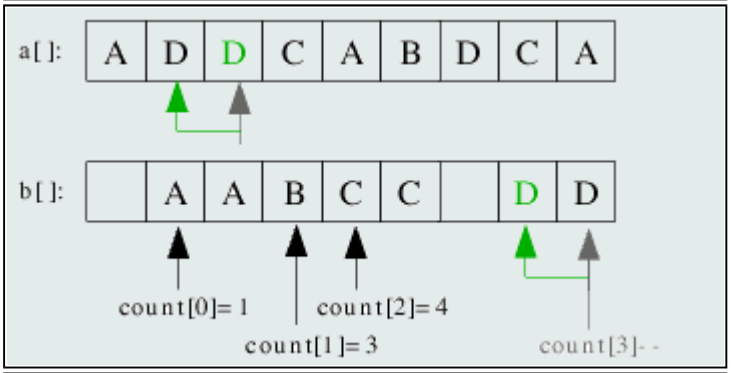
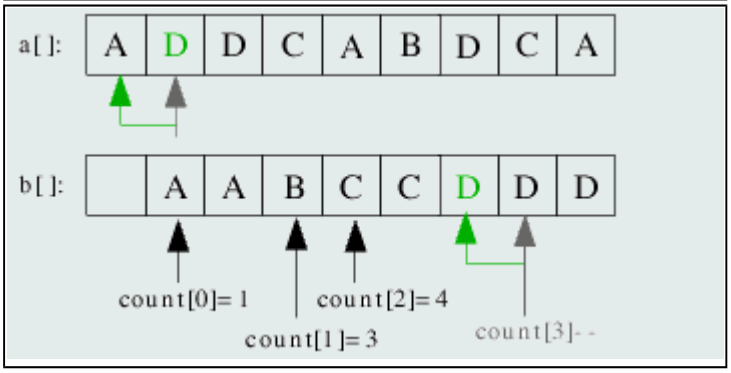
```
for(j=0; j<M; j++) count[j]=0;          //count[] mit 0 initialisieren
for(i=0; i<N; i++) count[a[i]]++;       //Vorkommen der Schlüssel speichern
for(j=1; j<M; j++)
    count[j]=count[j-1]+count[j];       //Positionen im Array bestimmen
for(i=N-1; i>=0; i--)
    b[--count[a[i]]]=a[i];              //in b einordnen, jeweiligen Zähler verri
for(i=0; i<N; i++) a[i]=b[i];           //auf a zurück speichern
```

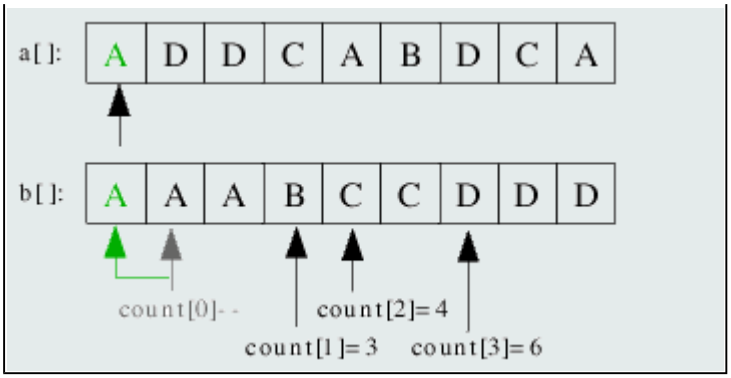
Die erste for-Schleife initialisiert count[] mit 0, in der zweiten for-Schleife erfolgt der erste Durchlauf durch die Datei, wobei die Anzahl der M verschiedenen Schlüssel gezählt und in count[] gespeichert wird. Diese werden in der dritten for-Schleife aufaddiert, um somit die Startpositionen gleicher Schlüssel im sortierten Array festzulegen. In der vierten Schleife erfolgt die eigentliche sortierte Einordnung im Hilfsspeicher b und zu guter Letzt die Rückspeicherung nach a.

Um die Verfahrensweise dieses Einordnens deutlicher zu machen, soll folgendes Beispiel verwendet werden:

1.) Ausgangssituation: wir haben N=9 Schlüssel, wobei es nur M=4 unterschiedliche Werte gibt. Eigentlich müssten diese 0-4 heißen, zur besseren Verdeutlichung wurden sich jedoch mit A, B, C und D benannt.	
2.1) Die zweite for-Schleife obigen Codes zählt die Vorkommen der einzelnen Schlüssel und speichert sie in count[]. count[] ist von der Dimension M=4 und kann somit alle Vorkommen unterschiedlicher Werte speichern.	

2.2) count[] enthält nun die Anzahl der einzelnen Werte. A ist 3-mal, B 1-mal, C 2-mal und D 3-mal vorhanden. Mit der 3. for-Schleife werden benachbarte count-Werte addiert.	
2.3) Durch die Addition ergibt sich folgendes count[]-Feld. Die einzelnen Werte zeigen jetzt im sortierten Array auf eine Position nach dem letzten Vorkommen des damit verbundenen Schlüssels. So ist count[0]=3, der dazu gehörende Schlüssel A kommt 3-mal vor, muss also auf a[0], a[1] und a[2] abgespeichert werden. a[count[0]]=a[3] zeigt somit auf das Element hinter dem letzten Vorkommen von A im sortierten Feld.	
3.1) Nun wird die Datei zum zweiten Mal durchlaufen. Unter b[] ist noch einmal illustriert, auf welchen Elementen die count[]-Zeiger stehen. Dieser Durchlauf erfolgt von hinten nach vorn, wobei jedes Element i genommen und an seine Position count[a[i]]-1 im sortierten Array eingeordnet wird. Hier wird z.B. das letzte Element A genommen, count['A']=count[0]=3, sodass A an seine Position count[0]-1=2 eingeordnet wird.	
3.2) Der nächste Schlüssel C wird auf seine Position count['C']-1=count[3]-1=5 eingeordnet.	
3.3) D wird auf der letzten Position eingeordnet und count[3] dekrementiert.	
3.4) Das einzige B in der Folge wird an	

seine Stelle count[2]-1 eingeordnet.	
3.5) Das nächste A kommt auf seine Position count[0]-1=1.	
3.6) Das zweite C wird links vom ersten C auf Position count[2]-1=4 kopiert.	
3.7) count[3] wird dekrementiert und D an die entsprechende Stelle im Hilfsarray b[] einsortiert.	
3.8) Das nächste D kommt ebenso auf seine Position.	

3.9) Der letzte fehlende Schlüssel A wird auf die letzte leere Stelle kopiert und die Datei gilt als sortiert.	

Das war doch nicht schwer, oder? Schön lässt sich auch erkennen, dass die relative Sortiertheit der einzelnen gleichen Schlüssel erhalten bleibt, da der zweite Durchlauf durch die Datei "von hinten nach vorn" stattfindet. D.h. hätte man die einzelnen A's usw. indiziert mit A_1, A_2... dann würde diese Reihenfolge auch in der sortierten Datei erhalten bleiben. Dies ist wichtig, da das gleich besprochene Straight Radix Sort davon ausgeht, dass diese Stabilität gegeben ist.

Man könnte sich fragen, warum nicht gezählt wird und dann die einzelnen Schlüssel gleich ohne weitere Betrachtung in das Array einsortiert werden. Man müsste nur den Wert von Schlüssel i kennen und könnte ihn gleich count[i]-mal in das Array schreiben und schon hätte man nach Abarbeitung für alle Schlüssel ein sortiertes Feld. Das kann man bei einstelligen Schlüsseln ruhig machen, Bucketsort findet aber vor allem dort Anwendung, wo die Schlüssel aus mehreren Ziffern bestehen. Z.B. könnte man Dezimalzahlen haben und mit Bucketsort von rechts nach links nach allen Dezimalziffern sortieren lassen, um am Ende ein sortiertes Feld zu haben. Da nur immer in 10 Fächer (für jede der 10 Ziffern 0-9, also M=10) eingeordnet werden muss, ist der zusätzliche Speicheraufwand für count[] auch überschaubar und Bucketsort arbeitet ausreichend schnell. Es muss also davon ausgegangen werden, dass es sich bei dem Wert nur um einen Teil des eigentlichen Schlüssels handelt und dann der ganze Schlüssel an seine Stelle im Array kopiert werden muss. Übrigens: ist M=2, so gleicht die eben beschriebene Vorgehensweise dem *Straight Radix Sort*, auf welches gleich noch ausführlicher eingegangen werden soll.

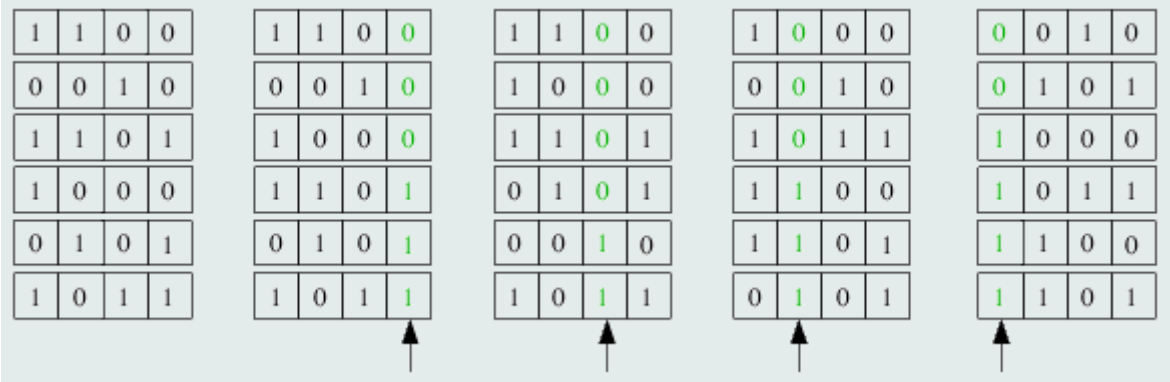
5.5 Straight Radix Sort

Dieses Verfahren, auch *geradliniges digitales Sortieren* genannt, betrachtet die einzelnen Schlüsselbits im Gegensatz zu Radix Exchange Sort von rechts nach links. Dabei werden die Schlüssel bei jedem neuen Bit wieder nach 0 und 1 sortiert, bis alle Bits betrachtet wurden - die Datei gilt als sortiert.
Man muss ein wenig überlegen, um sich von dem Funktionieren dieser Methode zu überzeugen - tatsächlich führt sie nur dann zum Erfolg, wenn der Prozess des Sortierens nach einem Bit stabil ist. Was das heißt, lässt sich schnell erklären: vor der Betrachtung des beispielsweise 3. Bits von rechts ist die Datei bei Straight Radix Sort schon nach den ersten beiden hinteren Bits sortiert. Wird nun nach dem 3. Bit sortiert, so muss die relative Sortiertheit der beiden hinteren Bits erhalten bleiben. Wenn es z.B. zwei Schlüssel gibt, welche im 3. Bit gleich sind, dann muss nach der Sortierung nach diesem Bit der Schlüssel, welcher von den hinteren beiden Bits her kleiner war, immernoch vor dem anderen Schlüssel stehen. Diese Stabilität zu erreichen ist aber nicht schwer, man kann allerdings z.B. nicht die Teil-Methoden verwenden, wie sie bei Quicksort oder Radix Exchange Sort zur Anwendung gekommen sind.
Vielmehr können wir die stabile Methode des Bucketsorts verwenden, um diesen Algorithmus zu implementieren. Die Anzahl an unterschiedlichen Schlüsseln ist dabei M=2 (0 und 1), sodass jeweils in nur 2 Fächer eingeordnet werden muss und sich somit der zusätzliche Speicherbedarf für count[] auf zwei int-Variablen beschränkt, der Speicherbedarf für das Hilfsfeld b aber natürlich noch bei n Elementen liegt. Die generelle Vorgehensweise sollte nicht schwer zu verstehen sein: betrachte die Schlüssel nach ihren Bits für alle Bitstellen von rechts nach links und sortiere nach den einzelnen Bitwerten mit Bucketsort. Führe dies solange aus, bis alle Bitstellen der Schlüssel betrachtet wurden.
Nach dieser Methode lässt sich *Straight Radix Sort* wie folgt implementieren:

```
inline unsigned getbit(unsigned x, int k){ return (x>>k)&;1; }

void straightradix(int a[], int N, int bits){
    int i, pass, count[2], b[N];
    for(pass=0; pass<bits; pass++){                      //die Bits der Reihe nach
        count[0]=0; count[1]=0;                          //count[] mit 0 initialis
        for(i=0; i<N; i++) count[getbit(a[i],pass)]++;   //Vorkommen der Bits spei
        count[1]+=count[0];                              //Position im Array besti
        for(i=N-1; i>=0; i--)
            b[--count[getbit(a[i],pass)]]=a[i];          //in b einordnen, jeweili
        for(i=0; i<N; i++) a[i]=b[i];                    //auf a zurück speichern
    }
}
```

Aufgrund des Bekanntseins von M=2 konnte Bucketsort so abgewandelt werden, dass zwei Schleifen weggefallen sind, was noch ein paar Laufzeitvorteile mit sich bringt. Trotzdem muss man aufpassen: in der derzeitigen Version ist Straight Radix Sort eigentlich nur eines: *lahm*. Und zwar genau deswegen, weil es absolut *jede* Bitstelle betrachtet und nicht wie etwa Radix Exchange Sort nur soweit, bis ein Element als definitiv eingeordnet gilt. Zudem wird pro Bitstelle die Datei von Bucketsort zweimal komplett durchlaufen und noch ein drittes Mal zur Rückspeicherung von b[] nach a[]. Dieses Verfahren liegt in seiner Laufzeit auch noch weit hinter Mergesort zurück und ist somit der langsamste zeitoptimale Sortieralgorithmus. Allerdings wird im kommenden Abschnitt eine Verallgemeinerung der Funktion angegeben werden, welche in praxi sogar schneller als Quicksort läuft.

Folgendes kleines Beispiel soll kurz die Arbeitsweise von Straight Radix Sort verdeutlichen:



Schön zu sehen ist, dass die Schlüssel bitweise von rechts nach links betrachtet werden und vor allem, dass die Schlüssel bei jeder Stufe nach den schon betrachteten Bits sortiert sind, was aufgrund des stabilen Bucketsorts möglich ist. Trotzdem ist dies eine sehr zeitaufwändige Variante, weil die Datei wie schon gesagt ganze dreimal von Bucketsort für jedes Bit betrachtet werden muss. Eine Abwandlung dieses Algorithmus' wird nachfolgend gegeben.

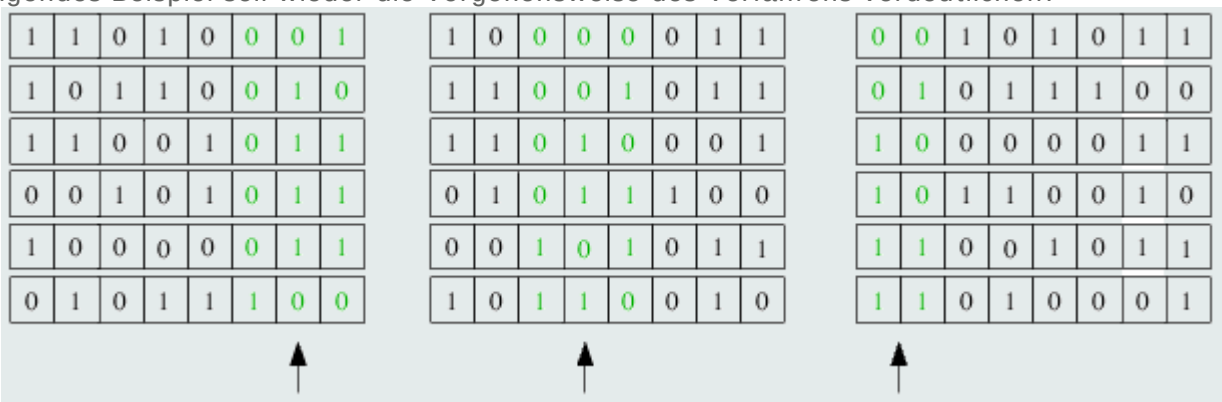
5.6 Lineares Sortieren

Das beste Ergebnis, auf das man bei einem Sortierverfahren *hoffen* kann, ist das der Linearität. Was heißt das?: es bedeutet, dass der Zeitaufwand, der für das Sortieren von n Schlüsseln benötigt wird, linear anwächst, d.h. dass man sich darauf verlassen kann, dass z.B. bei 2*n Daten sich die Laufzeit verdoppelt, bei 4*n verfierfacht usw. und nicht wie z.B. bei den elementaren Sortierverfahren mit $O(n^2)$ quadriert.
Ein solches lineares Sortierverfahren kann man leicht effektiv durch eine Abwandlung von oben gegebenem Straight Radix Sort erhalten. Vielmehr verallgemeinert man obige Funktion, wobei die hintergründige Idee ist, dass man statt nur eines Bits in jedem Durchlauf von Bucketsort gleich *m* Bits betrachtet, wodurch sich die Anzahl der Fächer, auf die verteilt wird, von M=2 auf $M=2^m$ erhöht. Dies erfordert natürlich wieder das ursprüngliche Bucketsort, da nicht wie bei M=2 zwei for-Schleifen entfallen können. Außerdem muss zur Extrahierung mehrerer Bits nun die Funktion getbits() verwendet werden, wie sie oben im Abschnitt 5.2 bereits erläutert

wurde. Der Vorteil ist, dass man das Feld nicht dreimal für jedes einzelne Bit durchlaufen muss, sondern nur für m Bits. Wie man m optimal wählen kann, wird gleich besprochen, hier erst einmal die verallgemeinerte Funktion von Straight Radix Sort:

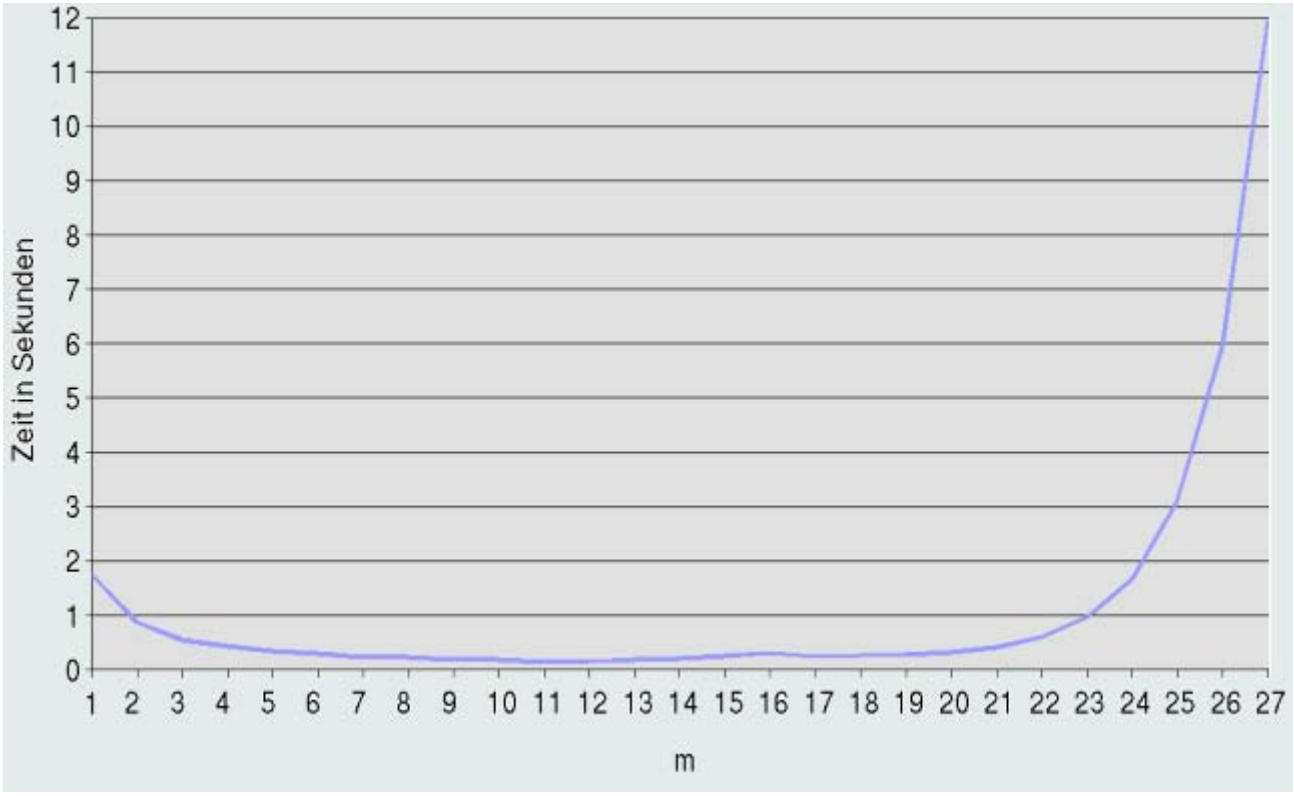
```
inline unsigned getbits(unsigned x, int k, int j){ return (x>>k)&~(~0<<j); }

void linstraightradix(int a[], int N, int bits, int m){
    int M=1, i, j, pass, *count, b[N];
    if(m<=0 || m>bits/2) m=bits/4;                          //Fehlerfälle abf
    for(i=0; i<m; i++,M*=2);                                //2^m berechnen
    count=(int*)malloc(M*sizeof(int));
    for(pass=0; pass<bits/m; pass++){                       //die Bits der Re
        for(j=0; j<M; j++) count[j]=0;                      //count[] mit 0 i
        for(i=0; i<N; i++) count[getbits(a[i],pass*m,m)]++; //Vorkommen der S
        for(j=1; j<M; j++)
            count[j]=count[j-1]+count[j];                   //Positionen im A
        for(i=N-1; i>=0; i--)
            b[--count[getbits(a[i],pass*m,m)]]=a[i];        //in b einordnen,
        for(i=0; i<N; i++) a[i]=b[i];                       //auf a zurück sp
    }
    free(count);
}
```

Folgendes Beispiel soll wieder die Vorgehensweise des Verfahrens verdeutlichen:



Wie beim ursprünglichen Straight Radix Sort werden die Bits von hinten nach vorn betrachtet, nur dass hier eben nicht nur ein Bit pro Durchlauf, sondern m Bits gleichzeitig betrachtet werden, sodass nur bits/m Durchläufe durch die äußere Schleife notwendig sind, um das gegebene Feld zu sortieren. Dadurch wird allerdings auch zusätzlicher Speicher von $M=2^m$ für count[] benötigt. Dieser Speicherbedarf ist nicht zu unterschätzen, heißt es doch, dass bei einer Verdopplung von m eine Quadrierung von M und somit des zusätzlichen Speicherplatzes erfolgt. Für m=4 sind z.B. nur 16 zusätzliche int's, für m=16 hingegen schon 65536 Plätze in count[] erforderlich. Noch gravierender fällt natürlich der zusätzliche Platz für das Feld b[] aus, welches genau soviel Elemente wie a[] aufnehmen muss. Das können bei n=10.000.000 Elementen und 32-Bit-int's schonmal über 38MB sein. Der zusätzlich notwendige Speicher für count[] und vor allem b[] ist auch der Hauptnachteil dieses linearen Sortierverfahrens. Steht ausreichend Arbeitsspeicher zur Verfügung, stellt das kaum ein Problem dar, doch zu sortierende Datenbestände sind nicht immer auf Hochleistungsrechnern zu Hause, sodass in diesem Falle arg abgewogen muss, ob der zusätzliche Speicher bereit gestellt werden kann oder ob man nicht doch lieber Quicksort verwenden sollte.

Die Frage aller Fragen ist natürlich: wie groß sollte man m wählen, damit das Sortierverfahren optimal arbeitet? Empirisch habe ich mit einem entsprechenden Programm folgende Zeiten beim Sortieren von 1.000.000 Elementen gemessen:



Natürlich muss m zwischen einschließlich 1 und der Anzahl der Bits im Schlüssel liegen. Diese waren hier bei einer normalen Ganzzahl 32 Bit, allerdings wird schon bei m=28 1GB zusätzlicher Speicher für count[] benötigt, sodass das Programm mit Speicherzugriffsverletzung abbrach. Offensichtlich ist bei m=11 ein Zeit-Minimum erreicht, auch der zusätzliche Speicher hält sich mit 2048 int's für count[] (entspricht 8KB) in Grenzen. Allgemein kann man feststellen, dass ungefähr *m=bits/4* zeitoptimal und der Speicheraufwand dabei nicht übermäßig groß ist. Mit dieser Wahl von einem Viertel der Wortlänge wird die Datei viermal von Bucketsort durchlaufen. Die Schlüssel werden als M-adische Zahlen betrachtet, also zur Basis M und es gibt insgesamt 4 (M-adische) Ziffern pro Schlüssel.
Da Bucketsort linear arbeitet und dieses verallgemeinerte Straight Radix Sort mit m=bits/4 genau 4 Durchläufe bei 32-Bit-Schlüsseln erfordert, ist das gesamte Verfahren linear. Das ist es auch mit jedem anderen m, so auch das ursprüngliche Straight Radix Sort, mit m=bits/4 kommt der Geschwindigkeitsvorteil allerdings erst richtig zur Geltung.
Folgende Funktion gibt noch einmal das optimierte Straight Radix Sort mit m=bits/4 an:

```
inline unsigned getbits(unsigned x, int k, int j){ return (x>>k)&~(~0<<j); }

void linoptstraightradix(int a[], int N, int bits){
    int M=1, m, i, j, pass, *count, b[N];
    m=bits/4;
    for(i=0; i<m; i++,M*=2);                                    //2^m berechnen
    count=(int*)malloc(M*sizeof(int));
    for(pass=0; pass<bits/m; pass++){                           //die Bits der Re
        for(j=0; j<M; j++) count[j]=0;                          //count[] mit 0 i
        for(i=0; i<N; i++) count[getbits(a[i],pass*m,m)]++;     //Vorkommen der S
        for(j=1; j<M; j++)
            count[j]=count[j-1]+count[j];                       //Positionen im A
        for(i=N-1; i>=0; i--)
            b[--count[getbits(a[i],pass*m,m)]]=a[i];            //in b einordnen,
        for(i=0; i<N; i++) a[i]=b[i];                           //auf a zurück sp
    }
    free(count);
}
```

Es hat sich bei mehreren Tests dieses Verfahrens empirisch gezeigt, dass der Geschwindigkeitsvorteil gegenüber einem verbesserten Quicksort bei zufälligen Permutationen im Bereich von 30-40% liegt.
Trotzdem sollte man mit Bedacht zwischen den einzelnen Sortierverfahren abwägen: gerade wenn man sich bewusst ist, dass man nicht den meisten Arbeitsspeicher zur Verfügung hat und Millionen von Datensätzen verwalten muss, kann der zusätzliche Speicherbedarf einer kompletten Kopie des zu sortierenden Feldes zum echten Hindernis werden. Deshalb sollte man vor allem auf Produktivsystemen mit Bedacht wählen, bevor man sich entscheidet: meist ist ein auch bei weniger Speicher stabil arbeitendes Quicksort besser als ein schnelleres lineares Sortierverfahren, welches die doppelte Arbeitsspeichermenge beansprucht. Ein weiterer Aspekt ist die Anpassbarkeit von Quicksort auf wohl definierte Probleme: sind die Eingabedaten nicht zufällig, so ist bei dem angepassten Straight Radix Sort mit erheblichen Effektivitätsverlusten zu rechnen.

5.7 Laufzeitbetrachtungen

Die Laufzeit der ersten beiden Radix-Sortierverfahren für das Sortieren von n Schlüsseln mit jeweils b Bits beträgt im Prinzip n*b. Im Wesentlichen bedeutet dies in Abhängigkeit nur von der Variablen n eine Laufzeitkomplexität von n*log(n), wenn davon ausgegangen wird, dass alle Schlüssel verschieden sind (da b in diesem Fall wenigstens log(n) sein muss), jedoch werden gewöhnlich viel weniger als n*b Operationen ausgeführt: bei Radix Exchange Sort, weil es abbricht, wenn ein Schlüssel an einer eindeutigen Position steht, beim verbesserten Straight Radix Sort, weil hier viele Bits gleichzeitig verarbeitet werden können.
Beide Sortierverfahren benötigen also für das Sortieren von n Schlüsseln mit b Bits *weniger als* n*b Bitvergleiche - d.h. in dem Sinne, dass die benötigte Zeit proportional zur Anzahl der Bits in den Schlüsseln ist, sind die digitalen Sortierverfahren linear - dies folgt auch daraus, dass sich leicht erkennen lässt, dass jedes Bit höchstens einmal betrachtet wird.
Radix Exchange Sort verwendet im Durchschnitt n*lg(n) Bitvergleiche und ist bei zufälligen Folgen mit Quicksort vergleichbar, das grundlegende Straight Radix Sort muss wie schon erwähnt ungefähr n*b Bitvergleiche ausführen - da hier jedoch absolut jedes Bit betrachtet werden muss und das Feld pro Bit dreimal komplett durchlaufen wird, arbeitet es in der Regel sehr viel langsamer als Radix Exchange Sort oder Quicksort.
Die zuletzt betrachtete lineare Version von Straight Radix Sort mit angepasstem m=bits/4 hingegen kann n Schlüssel in bits/4 Durchläufen sortieren, wobei allerdings zusätzlicher Platz für 2^m Zähler und einen Puffer für das Umordnen der Datei in der Größe dieser benötigt wird, was das größte Hindernis darstellt. Dafür hat man auf einfache Weise ein lineares Sortierverfahren auf der Hand, welches bei randomisierten Ausgangsfolgen schneller arbeitet als Quicksort und für Aufgaben, für die genügend Speicherplatz zur Verfügung steht und bei denen es primär auf Geschwindigkeit ankommt, bestens geeignet ist.

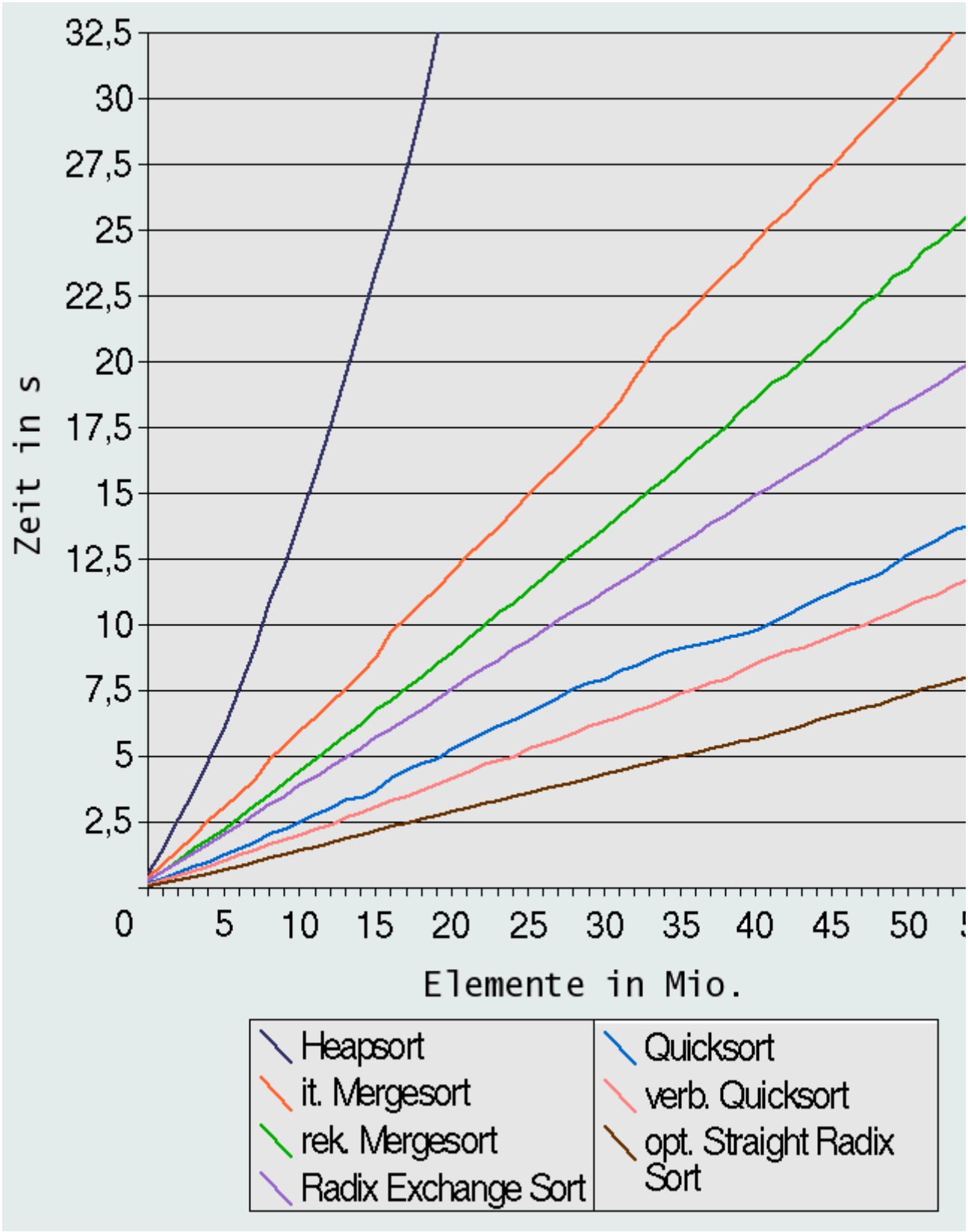
[Zurück zum Inhaltsverzeichnis]

6. Schlussbemerkungen

[Zurück zum Inhaltsverzeichnis]

6.1 Fazit

Es zeigt sich, dass die Wahl eines geeigneten Sortieralgorithmus' durchaus keine einfache Entscheidung ist. Quicksort z.B. ist das schnellste der Verfahren, welche sich auf Schlüsselvergleiche stützen, hat aber eine worst-case-Laufzeit von $O(n^2)$. Mergesort wiederhin ist nicht sehr schnell und benötigt proportional zu n zusätzlich viel Speicher, dafür hat es eine worst-case-Komplexität von O(n*log(n)), ist leicht zu implementieren, kaum störanfällig und vor allem bei Problemen zu verwenden, bei denen immer wieder Datensätze in einen bestehenden sortierten Datenbestand integriert werden sollen. Heapsort wiederhin vereint die Laufzeitkomplexität von O(n*log(n)) im schlechtesten Fall von Mergesort mit dem Quicksort-Vorteil, dass kein zusätzlicher Speicher benötigt wird - dafür ist es aber sehr langsam in der Ausführung.
Als echte Alternative kann man die digitalen Sortierverfahren ansehen. Schon Radix Exchange Sort ist bei zufälligen Permutationen Quicksort fast gleichwertig, doch erst das optimierte lineare Straight Radix Sort verspricht spürbare Geschwindigkeitvorteile von 30-50%. Zusätzlicher Speicherbedarf in Höhe der Eingabedaten ist der Tribut, den man dafür zahlen muss, daher ist es auch kein Universalmittel und sollte nur dort ohne Bedenken eingesetzt werden, wo sicher ist, dass genügend Speicher zur Verfügung steht und wo Geschwindigkeit alles entscheidend ist. Auf allen anderen Systemen ist ein verbessertes Quicksort zu empfehlen, welches auch schnell arbeitet und exzellent problemabhängig angepasst werden kann.
Die endgültige Wahl und Implementation eines der vorgestellen Algorithmen sollte am Ende allerdings vor allem bei ernsthaften Produktivsystemen einem Experten überlassen werden, um der Gefahr einer gefährlichen Fehlentscheidung vorzubeugen.
Nachfolgend soll noch ein Diagramm angegeben werden, in welchem ich die empirische Messung der Laufzeiten von den einzelnen Sortierverfahren fest gehalten habe und welches die Entscheidungsfindung nach dem optimalen Sortieralgorithmus vereinfachen soll:



6.2 Quellennachweis / Buchempfehlungen

Die hier dargestellten Informationen habe ich vor allem drei Büchern entnommen, die ich hier auch weiter empfehlen möchte. Aus dem ersten entstammen zudem die meisten Quelltexte, wobei hier und da sinnvolle bzw. notwendige Änderungen durchgeführt wurden.

- Sedewick, R.; *Algorithmen in C*

Addison-Wesley - 1992; ISBN 3-89319-376-6
- Ottmann/Widmayer; *Algorithmen und Datenstrukturen* (4. Auflage)
  Spektrum Akad. Verlag - 2002; ISBN 3-8274-1029-0
- Knuth, D.E.; *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2nd Edition)
  Addison-Wesley - 1999; ISBN 0-201-89685-0

6.3 Downloads

PDF: Zeitoptimale Sortierverfahren
ZIPPED HTML: Zeitoptimale Sortierverfahren

Testprogramm zeitoptimaler Verfahren: zeitopt_sort.c
Testprogramm elementarer Verfahren: elem_sort.c

6.4 Kontakt

Matthias Jauernig (aka RTC, author/maintainer):
Mail: rtc@linux-related.de, ICQ UIN: 81028529

Tobias Hammerschmidt (aka i84cOre, writer):
Mail: tobias.hammerschmidt@gmx.de, ICQ UIN: 148859859

[Zurück zum Inhaltsverzeichnis]