Sortierverfahren für Felder (Listen)

Generell geht es um die Sortierung von Daten nach einem bestimmten Sortierschlüssel. Es ist auch möglich, daß verschiedene Daten denselben Sortierschlüssel haben.

Es wird angenommen, daß alle zu sortierenden Daten vorliegen; d.h. zu sortieren ist eine Liste von n Elementen.

Beispiel für eine Sortieraufgabe: Dateienverzeichnisse

Name	Datum	Grösse	Art
`Anhang1-1.rtf`	`1.7.2002`	`21 K`	`RTF-Dokument`
`Bericht7.doc`	`14.5.2002`	`86 K`	`Textdokument`
`Detail18.jpg`	`7.7.2002`	`36 K`	`Dokument`
`Detail19.jpg`	`14.5.2002`	`36 K`	`Dokument`
`Fragen`	`8.7.2002`	`11 K`	`Simple Text`
...	...	...	...
...	...	...	...

Dateiverzeichnisse müssen oft nach verschiedenen Schlüsseln sortiert werden.

Ein Sortieralgorithmus wird stabil genannt, wenn die relative Anordnung der Listenelemente bei gleichem Schlüsselwert erhalten bleibt.
In obigem Beispiel wäre das der Fall, wenn bei der Sortierung nach Datum die Reihenfolge `Bericht7.doc` ... `Detail19.jpg` erhalten bleibt.

Einfache Sortierverfahren:

Im folgenden werden als Sortierschlüssel Buchstaben verwendet und nur diese Buchstaben in den Beispielen angegeben.

1. Bubble Sort

Die Liste wird n-mal durchlaufen. Dabei werden jeweils zwei benachbarte Buchstaben verglichen. Wenn sie falsch angeordnet sind, werden sie vertauscht.
Dabei steigen die größeren Werte auf wie Blasen (daher Bubble Sort).

C	G	C	A	B	I	E	F	B	D
C	C	A	B	G	E	F	B	D	*I*
C	A	B	C	E	F	B	D	*G*	*I*
A	B	C	C	E	B	D	*F*	*G*	*I*
A	B	C	C	B	D	*E*	*F*	*G*	*I*
A	B	C	B	C	*D*	*E*	*F*	*G*	*I*
A	B	B	C	*C*	*D*	*E*	*F*	*G*	*I*
A	B	B	*C*	*C*	*D*	*E*	*F*	*G*	*I*
A	B	*B*	*C*	*C*	*D*	*E*	*F*	*G*	*I*
A	*B*	*B*	*C*	*C*	*D*	*E*	*F*	*G*	*I*

(kursiv = sicher sortiert)

Man beachte, daß sich in den letzten 3 Durchläufen nichts mehr ändert.

2. Selection Sort (Sortieren durch Auswahl)

Suche das kleinste Element im Array und vertausche es mit dem ersten; finde das zweitkleinste und vertausche es mit dem zweiten; usw.

C	G	C	**A**	B	I	E	F	B	D
A	G	C	C	**B**	I	E	F	B	D
A	*B*	C	C	G	I	E	F	**B**	D
A	*B*	*B*	**C**	G	I	E	F	C	D
A	*B*	*B*	*C*	G	I	E	F	**C**	D
A	*B*	*B*	*C*	*C*	I	E	F	G	**D**
A	*B*	*B*	*C*	*C*	*D*	**E**	F	G	I
A	*B*	*B*	*C*	*C*	*D*	*E*	**F**	G	I
A	*B*	*B*	*C*	*C*	*D*	*E*	*F*	**G**	I
A	*B*	*B*	*C*	*C*	*D*	*E*	*F*	*G*	**I**

(kursiv = sicher sortiert; fett = Minimum)

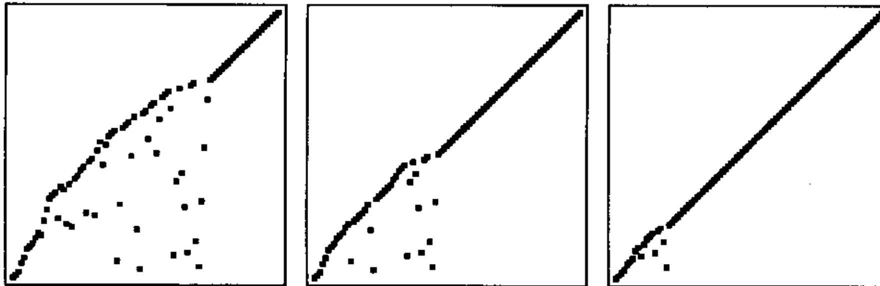
3. Insertion Sort (Sortieren durch Einfügen)

Man betrachtet eine von links nach rechts wachsende, provisorisch sortierte Teilliste. Bei jedem Durchlauf wird das nächstfolgende Element (direkt rechts von der Teilliste) an die richtige Stelle einsortiert.

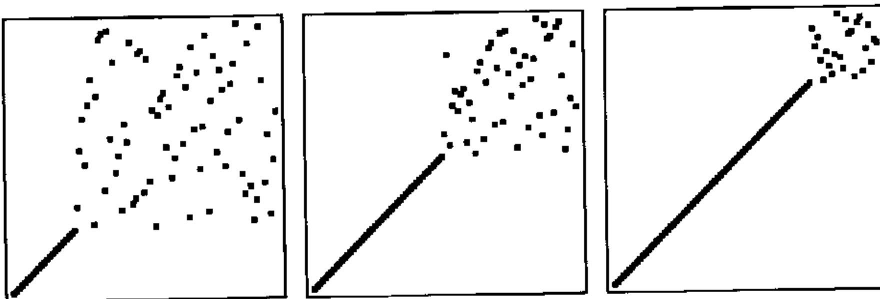
C	**G**	C	A	B	I	E	F	B	D
C	*G*	**C**	A	B	I	E	F	B	D
C	*C*	*G*	**A**	B	I	E	F	B	D
A	*C*	*C*	*G*	**B**	I	E	F	B	D
A	*B*	*C*	*C*	*G*	**I**	E	F	B	D
A	*B*	*C*	*C*	*G*	*I*	**E**	F	B	D
A	*B*	*C*	*C*	*E*	*G*	*I*	**F**	B	D
A	*B*	*C*	*C*	*E*	*F*	*G*	*I*	**B**	D
A	*B*	*B*	*C*	*C*	*E*	*F*	*G*	*I*	**D**
A	*B*	*B*	*C*	*C*	*D*	*E*	*F*	*G*	*I*

(kursiv=Teilliste; fett = nächstes Element)

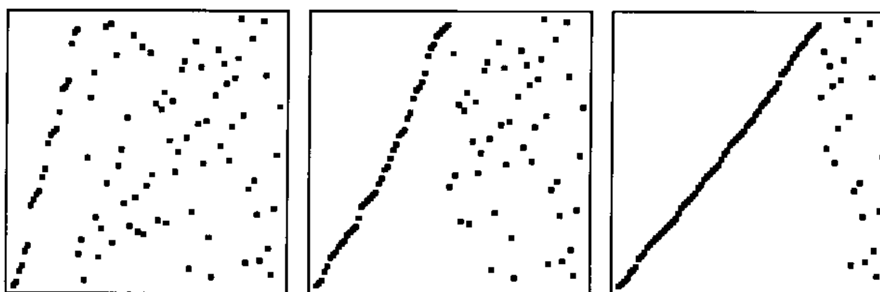
Die folgenden Abbildungen veranschaulichen das Verhalten der drei Algorithmen anhand der Sortierung der Zahlen von 1 bis 100, die zu Beginn einen zufällige Listenplatz zwischen 1 und 100 einnehmen. Die Bilder zeigen, welche Zahl (horizontale Position) welchen Listenplatz (vertikale Position) jeweils nach einem Viertel der benötigten Zeit einnimmt.



Bubble Sort



Selection Sort



Insertion Sort

Für den Fall, daß ein Programmlauf abgebrochen wird, ist es interessant zu wissen, welcher Teil der Liste schon sortiert ist.

Algorithmen und ihr Vergleich:

Laufzeit als Kriterium für die Auswahl eines Algorithmus

Andere Möglichkeiten:

Speicherplatz, Entwicklungskosten, Robustheit etc.

Messung der Laufzeit unabhängig von Faktoren wie Art der Programmierung, Prozessorgeschwindigkeit, Eigenschaften des Compilers etc.

Laufzeit hängt ab von

- Größe der Eingabe
- bei Sortieralgorithmen: anfängliche Reihenfolge

Größe der Eingabe wird auf die Betrachtung elementarer Einheiten reduziert (d.h. es spielt z.B. keine Rolle, ob jeweils nur der erste Buchstabe des Suchschlüssels oder eventuell mehrere verglichen werden müssen)

Algorithmus wird auf elementare Operationen reduziert;
bei Sortieralgorithmen also: Vergleiche, Vertauschungen

Fallunterscheidung: günstigster/durchschnittlicher/ungünstigster Fall
(z.B. richtig - gar nicht - entgegengesetzt sortiert)

Bubble Sort:

Bubble Sort benötigt (n-1) + (n-2) + ... 2 + 1 ≈ $n^2/2$ Vergleiche und im schlechtesten Fall ebenso viele Vertauschungen. Im günstigsten Fall (sortiertes Array) sind es 0 Vertauschungen.

Selection Sort:

Selection Sort benötigt ebenso wie Bubble Sort etwa $n^2/2$ Vergleiche. Im durchschnittlichen und schlechtesten Fall sind n Vertauschungen nötig, im günstigsten Fall (sortiertes Array) 0 Vertauschungen.

Insertion Sort :

Insertion Sort benötigt im durchschnittlichen Fall etwa $n^2/4$ Vergleiche und Vertauschungen; $n^2/2$ im schlechtesten Fall. (Genaugenommen handelt es sich bei den Vertauschungen allerdings um schneller zu realisierende Verschiebungen.)
Im günstigsten Fall ist Insertion Sort linear.

Vergleiche:

	Bubble Sort	Selection S.	Insertion S.
günstigster Fall	$n^2/2$	$n^2/2$	n
durchschn. Fall	$n^2/2$	$n^2/2$	$n^2/4$
schlechtester Fall	$n^2/2$	$n^2/2$	$n^2/2$

Vertauschungen:

	Bubble Sort	Selection S.	Insertion S.
günstigster Fall	0	0	0
durchschn. Fall	$n^2/4$	n	$n^2/4$
schlechtester Fall	$n^2/2$	n	$n^2/2$

O-Notation, wichtige Laufzeit-Klassen

Die bisher untersuchten Algorithmen beruhen im wesentlichen darauf, daß jedes Element mit jedem anderen verglichen wird. Ihr Laufzeitverhalten ist daher abhängig von n^2, wenn man die Faktoren ignoriert. Man sagt auch, die Funktionen, die ihr Laufzeitverhalten beschreiben, sind von derselben Ordnung.

Eine Funktion f(n) ist von der Ordnung der Funktion g(n) (= O(g(n))), falls Konstanten c und n_0 existieren, so daß $g(n) < c * f(n)$ für alle $n > n_0$.

Z.B. sind $g(n) = n^2+5*n+8$ und $h(n) = 5n^2$ beide in $O(n^2)$. Faustregel: Konstanten, Faktoren und Polynome niedrigerer Ordnung können weggelassen werden.

Durch diese O-Notation erhält man eine Einteilung der Funktionen, die das Laufzeitverhalten von Algorithmen beschreiben, in Klassen. Die wichtigsten davon sind:

1	Konstante Funktion. Egal, wie groß oder klein die Eingabe, der Algorithmus braucht immer eine festgelegte Anzahl von Elementaroperationen.
	Beispiel: Algorithmus, der bestimmt, ob die ersten 1000 Listenelemente richtig sortiert sind.
log n	logarithmische Abhängigkeit. Die Laufzeit wächst langsamer als die Eingabe.
	Beispiel: Algorithmen, die die Eingabemenge in Teile teilen; z.B. Suche nach einem Blatt in einem Binärbaum.
n	Lineare Funktion. Die Laufzeit wächst proportional zur Eingabe
	Beispiel: Algorithmus, der bestimmt, ob eine vollständige Liste richtig sortiert ist, indem er jeweils die benachbarten Elemente vergleicht.
n log n	Kombination aus logarithmischer und linearer Abhängigkeit.
	Beispiel: Aufteilung einer Datenmenge auf die Blätter

	eines Binärbaums.
n^2	Quadratisches Laufzeitverhalten.
	Beispiele: Bubble Sort, Selection Sort, Insertion Sort
2^n	Algorithmen, bei denen z.B. alle Teilmengen der Eingabe einzeln betrachtet werden; können nur bei sehr kleinen n eingesetzt werden. Leider gibt es Probleme, die wahrscheinlich nur mit nichtdeterministischen Verfahren schneller (in polynomialer Zeit) gelöst werden können („NP-vollständige Probleme“).
	Beispiel: Zerlegung einer Menge von Zahlen in 2 Teile, so daß die Summen beider Teile möglichst gleich sind.

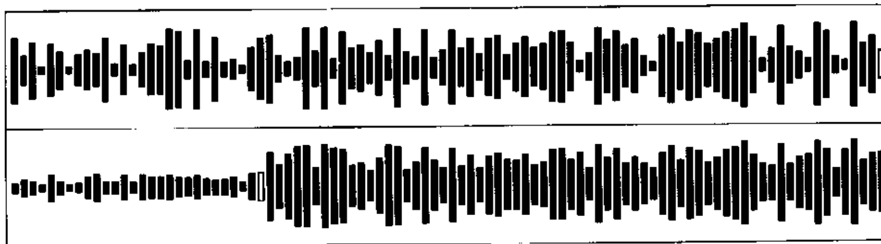
Grobe Näherungswerte:

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
3	10	30	100	1024
7	100	700	10000	10^{30}
10	1000	10.000	1.000.000	10^{300}
13	10000	130.000	100.000.000	10^{3000}
16	100000	1.600.000	10.000.000.000	10^{30000}

Quicksort

Quicksort ist ein Beispiel für einen Algorithmus nach dem "Teile und herrsche"(divide et impera)-Verfahren; die Gesamtliste wird in zwei Teile zerlegt, die dann rekursiv ihrerseits sortiert werden.

Zur Aufteilung in zwei Teillisten wählt man ein beliebiges Element aus der zu sortierenden Liste aus. Danach sorgt man dafür, daß alle Elemente, deren Wert größer ist als der des Vergleichselements, in die rechte "Hälfte" des Feldes kommen, und alle, deren Wert kleiner ist, in die linke "Hälfte" (die sogenannten Hälften können natürlich unterschiedlich groß sein). Das Element, dessen Wert den Vergleichsmaßstab gebildet hat, wird dann genau zwischen die "Hälften" gesetzt; es bildet nun die Grenze zwischen den Teilen. Beispiel:



Um dies zu erreichen, kann man wie folgt vorgehen: Als Vergleichsgröße wählt man den Wert des letzten Elements der Liste. Nun sucht man von rechts nach einem Element, dessen Wert kleiner als die Vergleichsgröße ist (es steht in der falschen Hälfte). Genauso sucht man von links ein Element, das zu groß ist. Nun werden die beiden gefundenen Elemente ausgetauscht und stehen nun richtig. Wenn die Suche aus beiden Richtungen sich trifft, wird das am weitesten links stehende Element der rechten „Hälfte“ gegen das Vergleichselement ausgetauscht.
Das Vergleichselement steht somit an seiner endgültigen Position. Es müssen nur noch die jeweiligen „Hälften“ sortiert werden.

Beispiel:

C	G	C	A	B	I	E	F	B	**D**
C	*B*	C	A	B	I	E	F	*G*	**D**
C	*B*	*C*	*A*	*B*	**D**	*E*	*F*	*G*	*I*
C	B	C	A	**B**	D	E	F	G	**I**
A	B	C	*C*	**B**	D	*E*	*F*	*G*	**I**
A	*B*	**B**	*C*	*C*	D	*E*	*F*	*G*	**I**
A	**B**	B	C	**C**	D	E	F	**G**	I

(fett = Vergleichselement; kursiv = sortierte Teilstücke)

Im idealen Fall wird die zu sortierende Liste jeweils genau in zwei gleichgroße Hälften geteilt. Dann sind $\log_2 n$ Durchgänge erforderlich, in denen jeweils maximal n Vergleiche stattfinden. Die genaue Laufzeit des Algorithmus hängt davon ab, inwieweit diese gleichmäßige Zerlegung gelingt. Im Durchschnitt sind 1,38 $n \log_2 n$ Vergleiche notwendig.

Mergesort (Mischsortieren)

Quicksort teilt eine Datei so in zwei Teile, daß der eine alle kleineren und der andere alle größeren Werte enthält. Wenn man beide Teile anschließend wiederum teilt und die sortierten Teile zum Schluß alle zusammensetzt, erhält man eine vollständig sortierte Datei.

Bei Mergesort wird die zu sortierende Datei ebenfalls in zwei Teile geteilt - allerdings beliebig. Beide Teile werden dann (durch rekursive Anwendung von Mergesort) einzeln sortiert und anschließend ineinander"gemischt" (d.h. sortiert).

Nachteil:

- man benötigt für das Ineinandermischen von Feldern ein zusätzliches Feld

Vorteile:

- selbst im schlechtesten Fall O(n log n)

- kann auch verwendet werden, um zwei Dateien zu sortieren, die zu groß sind, um zusammen in den Speicher zu passen

- ist ideal für das Sortieren verketteter Listen, die nur sequentiell durchlaufen werden können
- ist ideal für das Einsortieren einer kleinen Datenmenge in eine große, sortierte

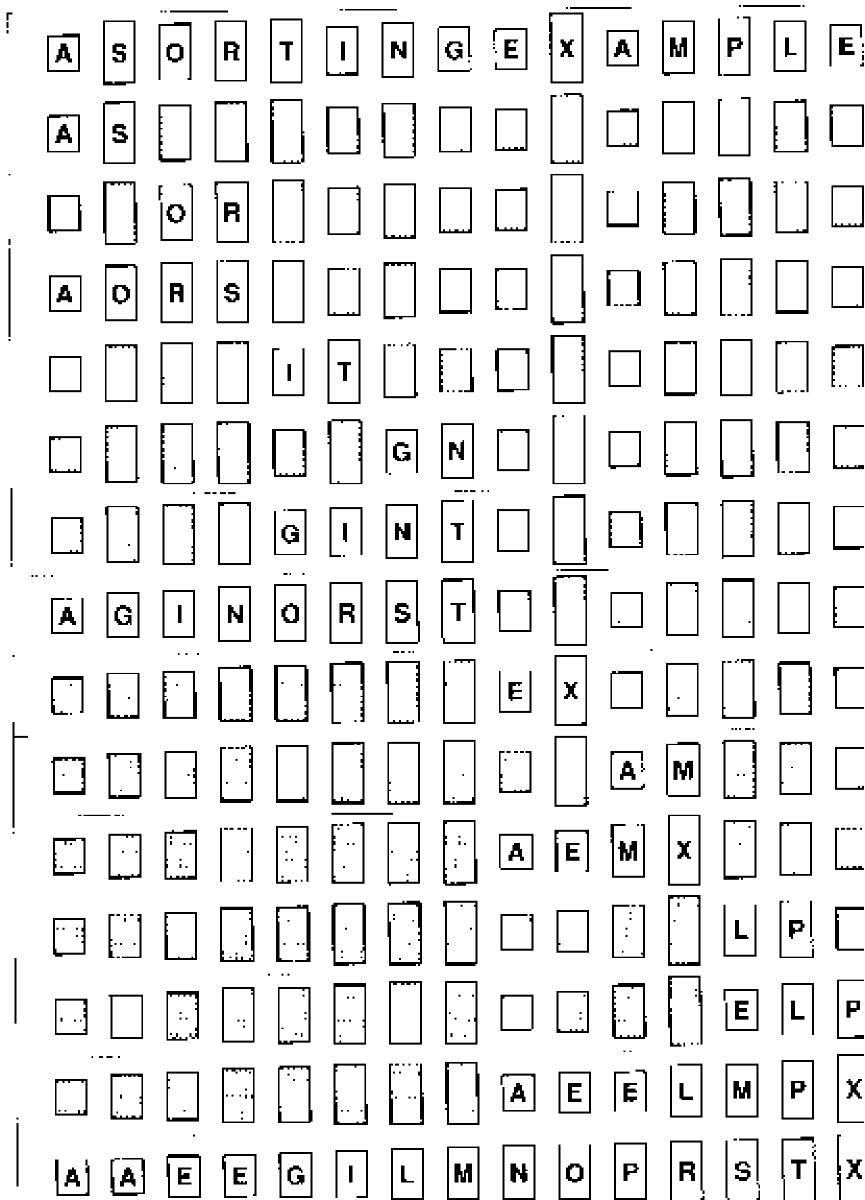


Figure 12.1 Recursive Mergesort.

Suchen und Sortieren mit Binärbäumen

Für die Suche nach einem bestimmten Element in einer Menge von Elementen gab es bisher folgende Lösungen:

- Verkettete Listen

  In diesem Fall muß man die Liste solange durchsuchen, bis man das betreffende Element gefunden hat.
  Dies gilt auch für sortierte verkettete Listen; allerdings stehen hier Elemente mit demselben Suchschlüssel hintereinander, und ein Fehlen des gesuchten Elements wird schneller bemerkt.

- Felder

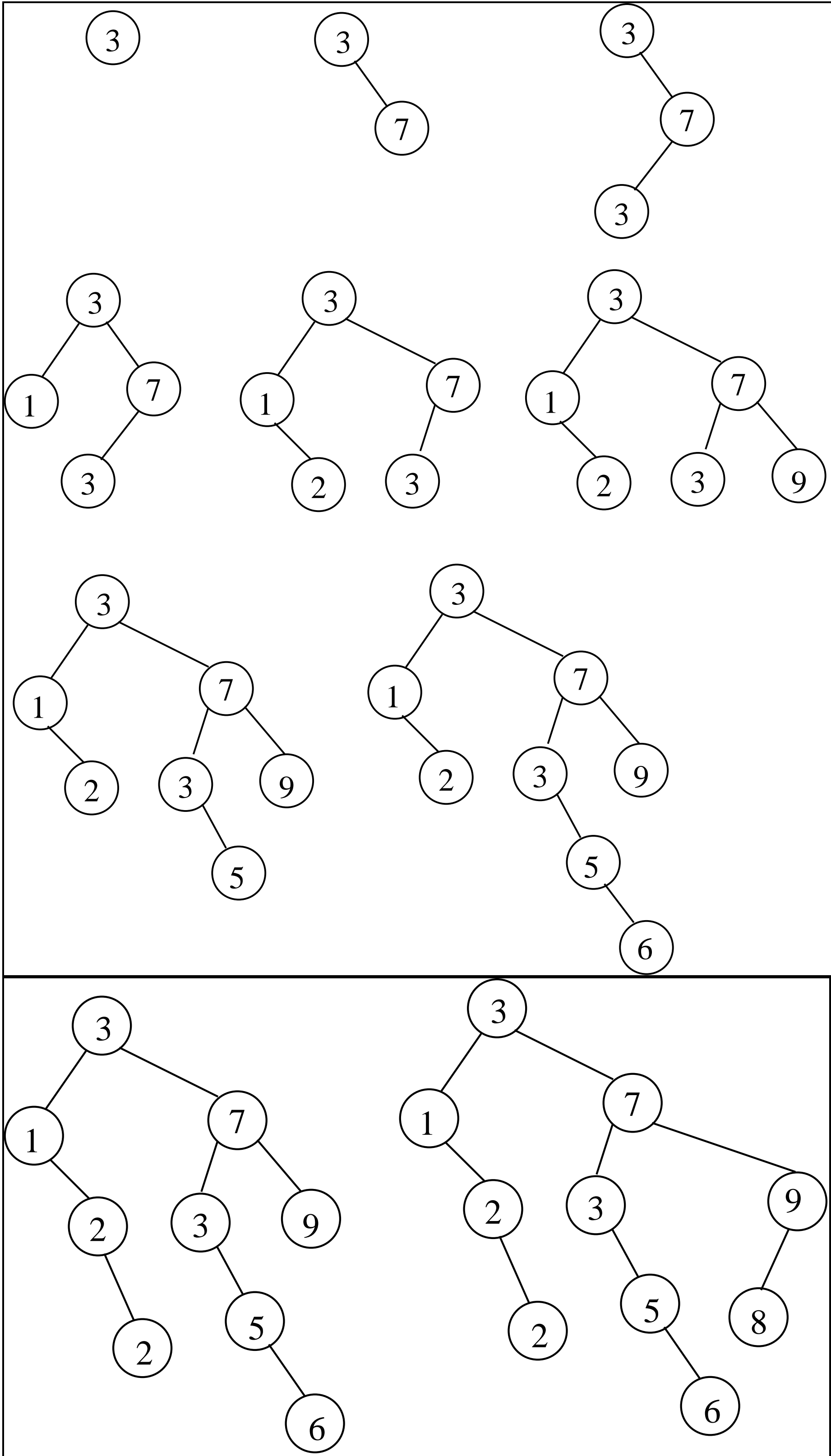
  Bei unsortierten Feldern muß man ebenfalls alle Elemente durchsuchen.
  Bei sortierten Feldern kann man aufgrund des wahlfreien Zugriffs das Verfahren "Teile und herrsche" anwenden:
  Man sortiert die Daten, vergleicht den Suchschlüssel mit dem mittleren Element und untersucht die Hälfte, in der sich der gesuchte Wert befinden muß. Diese Hälfte wird wiederum geteilt, etc.
  Allerdings sind sortierte Felder sehr aufwendig zu verwalten.

- Indexsequentielle Darstellung
  Kombination aus Feldern und verketteten Listen

Noch besser: Binärbäume

Aufbaumaxime: Jeder Knoten eines binären Suchbaums enthält eines der zu sortierenden Elemente, und zwar so, daß der linke Teilbaum, der von einem Knoten ausgeht, nur kleinere Werte enthält, der rechte nur solche, die größer oder gleich groß sind.

Einfügen der Zahlen: 3, 7, 3, 1, 2, 9, 5, 6, 2, 8:



Notwendig: Prozeduren zum Einfügen eines Elementes, zum Entfernen eines Elementes und zum Suchen eines Elementes

Suchen:

von der Wurzel ausgehend Vergleich, ob der Wert des betrachteten Elements kleiner, größer oder gleich dem gesuchten Wert ist;
in den beiden ersten Fällen Abstieg in den linken bzw. rechten Teilbaum

Einfügen:

Abstieg bis zum nächstkleineren bzw. -größeren Blatt

Löschen:

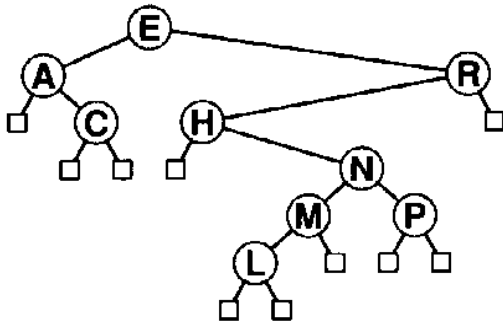
Einfache Fälle:

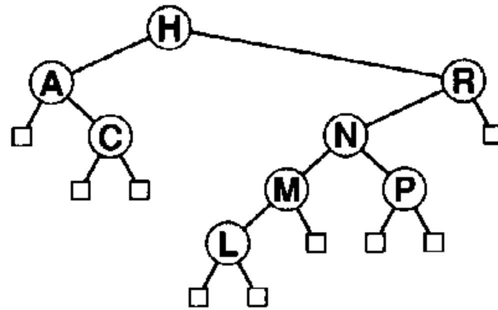
- Knoten ohne Nachfolger (6 oder 8 im obigen Beispiel)

- Knoten mit nur einem Nachfolger (5 oder 9); dieser Nachfolger ersetzt dann den zu löschenden Knoten.

- Knoten mit zwei Nachfolgern, von denen einer selbst keine Nachfolger hat; dieser Nachfolger ersetzt dann den gelöschten Knoten.

Schwierig sind jedoch die inneren Knoten, wie etwa 3 oder 7. Hier hilft die folgende Strategie: Ersetze den zu löschenden Knoten durch den Knoten mit dem nächst höheren Wert in der Sortierreihenfolge. Dieser Knoten hat garantiert keinen linken Nachfolger. Beispiel:

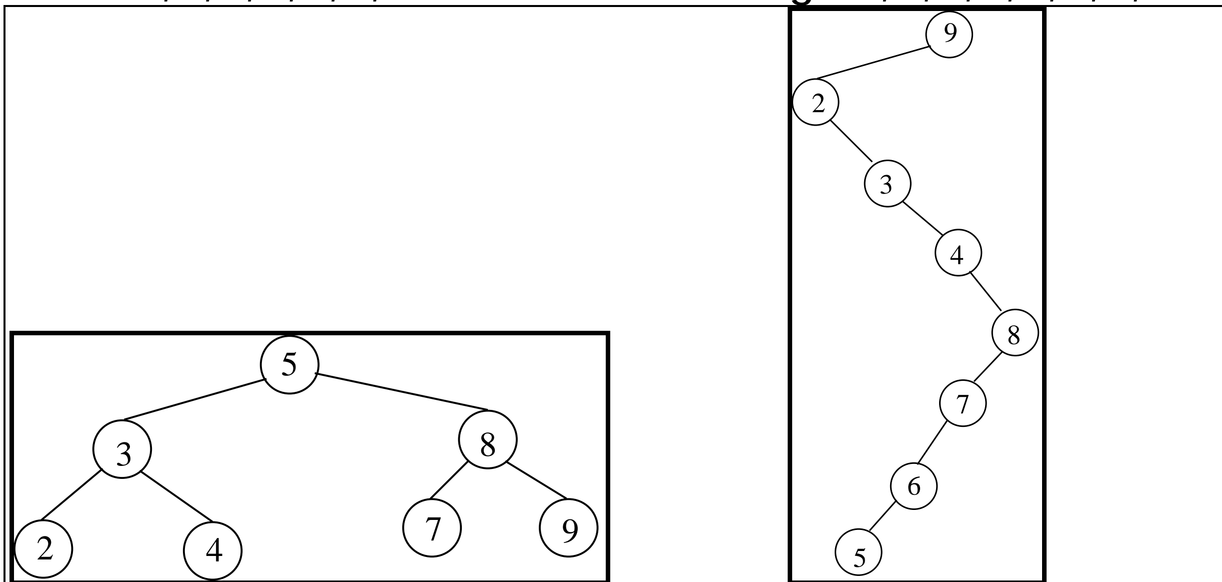




Wie effizient ein binärer Suchbaum ist, hängt davon ab, wie symmetrisch (balanciert) der Baum ist. Dies hängt von der Reihenfolge ab, in der die Elemente eingefügt werden. Im schlimmsten Fall degeneriert der Binärbaum zu einer linearen Liste. Beispiel:

Perfekt balancierter Baum der Zahlen 5, 3, 2, 4, 8, 7, 9

Degenerierter Baum bei anderer Reihenfolge: 9, 2, 3, 4, 8, 7, 6, 5



Durchschnittlicher Baum: