Datenstrukturen

1 Lineare Datenstrukturen (Sequenzen)

1.1 Definition

Eine Sequenz von Werten vom Typ T ist eine Abbildung einer Indexmenge [istart,...,iend] in eine Menge von Werten des Typs T (=array)
> statische Umgebung

Eine Sequenz von Werten vom Typ T ist entweder leer oder sie besteht aus einem Element vom Typ T und einer Sequenz von Werten
des Typs T (=linked list)
> dynamische Umgebung

1.2 Abstrakte Datentypen ADT

1.2.1 Arrays (Vektoren)

1.2.1.1 Stack

Funktionen: Initialisierung (Anlegen eines Stacks, Positionszähler auf 0) – init
Einfügen – push
Zurückgeben des zuletzt eingefügten Objektes – top
Löschen des zuletzt eingefügten Objektes – pop LIFO
Überprüfen, ob der Stack leer ist
Überprüfen, ob der Stack voll ist

1.2.1.2 Queue (Schlange)

Funktionen: Initialisierung – init
Einfügen – Enqueue
Zurückgeben des zuerst eingefügten Objektes – front
Löschen des zuerst eingefügten Objektes – back FIFO
Überprüfen, ob die Queue leer ist
Überprüfen, ob die Queue voll ist

1.2.1.3 Set (Menge)

Funktionen: Initialisierung – init
Einfügen
Suchen
Löschen
Überprüfen, ob die Queue leer ist
Überprüfen, ob die Queue voll ist

1.2.1.4 Vor- und Nachteile von Arrays

Vorteile von Arrays
Direkter, unmittelbarer Zugriff auf jedes Element (in O(1))
- Einfaches sequentielles Durchlaufen, z.B. mit Schleifen
- Die aufeinanderfolgende Anordnung im Feld ist die direkte Repräsentation der Anordnung der Elemente im Speicher
- direkte Entsprechung der Vektoren (eindimensional) und Matrizen (höherdimensional) der Mathematik
- einfache, unmittelbare Umsetzbarkeit mathematischer Operationen
- **überschreibendes** Einfügen an feste Position erfolgt in konstantem Aufwand (in O(1))

- Entfernen eines Elements anhand des Indexes erfolgt ebenfalls in O(1)
- einfache, intuitive Navigation durch Index
- Effizienteste Struktur für Datenzugriffe überhaupt

Nachteile von Arrays
- Die Feldgröße ist fest (PASCAL: statisch fest, andere: dynamisch fest)
- Bei Einfügen neuer Elemente müssen alle vorhandenen Elemente in den neuen Speicherbereich umkopiert werden O (n)
- nicht-überschreibendes Einfügen oder Löschen erfordert i.d.R. Umkopieren der nachfolgenden Werte, d.h. ebenfalls O(n)
- bei nur dünn besetzten Vektoren oder Matrizen wird viel Speicher verschwendet
- Alle Feldelemente müssen den gleichen Typ besitzen oder wenigstens konform dazu sein (d.h. ein Untertyp des angegebenen Typs sein).

1.2.2 Listen

1.2.2.1 Darstellung

Es gibt verschiedene Darstellungen für Listen die für unterschiedliche Anwendungen genutzt werden:

- ungeordnete und geordnete Listen
- einfach und doppelt verlinkte Listen
- lineare und kreisförmige Listen
- Listen mit und ohne Dummyelementen

Jeder Listenknoten besteht aus einem Eintrag (key), einem Vorwärtszeiger auf den Nachfolger und einem Rückwärtszeiger auf den Vorgänger. Bei kreisförmigen Listen zeigt der Vorwärtszeiger des letzten Listenknotens auf das erste Element der Liste.
Die Liste selbst wird Dargestellt durch eine Referenz (auf den ersten Knoten bzw.) auf einen Dummyknoten. Eine leere Liste besteht aus dem Dummyknoten, dessen Zeiger auf sich selbst zurückweisen.

Für ungeordnete Listen sind folgende Basisfunktionen zu betrachten:
Insert - Einfügen (am Anfang, am Ende, nach einer beliebigen Position)
Delete - Löschen (am Anfang, am Ende, an einer beliebigen Position)
Empty
Get Key (and Data) (erster Knoten, letzter Knoten, Knoten an beliebigen Positionen)
Search

Für geordnete Listen sind folgende Operationen von Bedeutung:
Insert
Delete
Search

1.2.2.2 Ungeordnete Listen

Einfügen eines Elementes **am Beginn der Liste** (vor den alten Anfang)
1. Erstelle einen neuen Knoten N
2. Weise dem Knoten den Schlüsselwert s zu
3. Setze den Vorwärtszeiger von N auf den Anfang der alten Liste

4. Setzte den Rückwärtszeiger von N auf den Vorgänger des Anfangsknoten der alten Liste (entspricht dem letzten Knoten)
5. Setze den Rückwärtsknoten des alten Anfangsknotens auf den neuen Knoten N
6. Setze den Vorwärtszeiger des letzten Knotens auf den neuen Knoten N
7. Ersetze den Referenzzeiger für die Liste durch den Zeiger auf den neuen Knoten N

Analog kann der Algorithmus angewandt werden, um einen Knoten am Ende der Liste einzufügen (vor den Dummyknoten). Dabei bleibt die Listenreferenz unverändert.

Einfügen eines Knotens **an eine beliebige Position** (nach einem existierenden Knoten P)
1. Erstelle einen neuen Knoten N
2. Weise dem Knoten den Schlüsselwert s zu
3. Setze den Vorwärtszeiger von N auf den Nachfolger von P
4. Setze den Rückwärtszeiger von N auf P
5. Setze den Vorwärtszeiger von P auf N
6. Setze den Rückwärtszeiger des Nachfolgers von P auf N

Empty
1. Überprüfe, ob der Referenzzeiger der Liste auf sich selbst verweist

Löschen des ersten Knoten N der Liste (wenn die Liste nicht leer ist)
1. Setze den Vorwärtszeiger des Vorgängers von N auf den Nachfolger von N
2. Ersetze den Referenzzeiger der List durch den Vorwärtszeiger von N
3. Lösche den Knoten N

Analog kann der Algorithmus angewandt werden, um jeglichen Knoten der Liste zu löschen, wenn dessen Position gegeben ist. Dabei bleibt die Listenreferenz unverändert.
Einen Schlüssel eines beliebigen Knotens der Liste mit gegebener Position zurückzugeben ist trivial. Wenn die Liste durch eine Referenz auf den ersten Knoten dargestellt wird ist, kann der erste Schlüssel sofort ausgegeben werden. Um den Schlüssel des letzten Knotens zu ermitteln muß der Vorgänger der Listenreferenz betrachtet werden.

1.2.2.3 Geordnete Listen

Einfügen eines Schlüssels s
1. Suche die Position des ersten Knotens, dessen Schlüssel größer als der einzufügende Schlüssel s ist
2. Ist die Suche erfolgreich, führe die Schritte 3 bis 10 aus
3. Erstelle einen neuen Knoten N.
4. Weise dem Knoten den Schlüsselwert s zu
5. Setze den Vorwärtszeiger von N auf den Knoten P
6. Setze den Rückwärtszeiger von N auf den Vorgänger von P
7. Setze den Vorwärtszeiger des Vorgängers von P auf N
8. Setze den Rückwärtszeiger von P auf N
9. Wenn P der Anfang der Liste ist, fahre mit Schritt 10 fort
10. Ersetze den Listenreferenzzeiger durch einen Zeiger auf den neuen Knoten N
11. Wenn die Suche erfolglos ist, füge den Schlüssel zu der geordneten Liste hinzu, indem die Schritte 3 bis 8 ausführt werden und der ersten Knoten der Liste als der Knoten P betrachtet wird

Löschen eines Schlüssels s
1. Suche die Position des Schlüssels s
2. Führe das Löschen des gefundenen Knotens durch

Suchen
Das Suchen wird als sequenzielle Suche formuliert, die jedoch abgebrochen werden kann, sobald ein Schlüsselwert gefunden wird, der größer als der gesuchte Schlüssel ist.

1.2.2.4 Stack

Stacks können als ungeordnete Listen implementiert werden, wobei das Einfügen und das Löschen am Ende der Liste durchgeführt werden muß.

1.2.2.5 Queue

Queues können als ungeordnete Listen implementiert werden, wobei das Einfügen am Ende und das Löschen am Anfang der Liste durchgeführt werden muß.

1.2.2.6 Set

Sets können als geordnete Listen ohne Duplikate betrachtet werden. Zusätzliche Algorithmen werden jedoch für die Operationen Vereinigungsmengenbildung, Durchschnittsmengenbildung und Teilmengenbildung benötigt:

Vereinigungsmenge
1. Betrachte die Mengen s1 und s2 als die erste und die zweite Menge.
2. Die Menge r entspricht der resultierenden Menge
3. Solange der Durchlauf durch die Mengen nicht beendet ist führe die Schritte 4 bis 13 durch
4. Wenn die betrachteten Schlüssel von s1 u s2 identisch sind fahre mit den Schritten 5 bis 7 fort
5. Füge den betrachteten Schlüssel von s1 zur Menge r hinzu
6. Setze die Position in der Menge s1 um ein Element weiter
7. Setze die Position in der Menge s2 um ein Element weiter
8. Wenn der betrachtete Schlüssel von s1 kleiner als der von s2 ist, führe die Schritte 9 und 10 aus
9. Füge den betrachteten Schlüssel von s1 zur Menge r hinzu
10. Setze die Position in der Menge s1 um ein Element weiter
11. Wenn der betrachtete Schlüssel von s1 größer als der von s2 ist, führe die Schritte 12 und 13 aus
12. Füge den betrachteten Schlüssel von s2 zur Menge r hinzu
13. Setze die Position in der Menge s2 um ein Element weiter
14. Solange der Durchlauf durch die Menge s1 noch nicht abgeschlossen ist führe die Schritte 15 und 16 aus
15. Füge den betrachteten Schlüssel von s1 zur Menge r hinzu
16. Setze die Position in der Menge s1 um ein Element weiter
17. Solange der Durchlauf durch die Menge s2 noch nicht abgeschlossen ist führe die Schritte 18 und 19 aus
18. Füge den betrachteten Schlüssel von s2 zur Menge r hinzu
19. Setze die Position in der Menge s2 um ein Element weiter

Die Algorithmen für die Operationen Durchschnittsmenge und Teilmenge sind ähnlich!

1.2.2.7 Vor- und Nachteile der verketteten Liste

Vorteile
- Dynamische Länge
- Es ist kein Kopieren der Elemente bei Löschen oder Einfügen erforderlich
- Keine Speicherverschwendung durch Vorreservierung des Speicherplatzes
- Einfache Einfüge- und Löschoperationen
- In den Listenelementen enthaltene Objekte können verschiedenen Typ haben

Nachteile
- Zugriff erfolgt immer sequentiell
- Positionierung liegt in O(n)
- Vorgänger sind nur aufwendig erreichbar
- Operationen Einfügen / Löschen nur hinter dem aktuellen Element möglich, sonst komplette Positionierung erforderlich

1.3 Suchen

1.3.1 Sequential Search (sequentielle Suche)

Die einfachste und allgemeinste Methode der Suche von Elementen ist die sequentielle Suche, bei der einfach alle Elemente der Reihe nach überprüft werden.
Aufgrund des überaus einfachen Vorgehens wird sie auch als brute force search (" Suche mit brutaler Gewalt") bezeichnet.
Die sequentielle Suche kann auf allen Datenstrukturen einfach durch eine Schleife (Array, Listen) oder Tiefensuche (Bäume, Graphen) implementiert werden.

Sequentielle Suche auf sortierten Daten
Sind die Daten sortiert, so kann die sequentielle Suche abgebrochen werden, sobald das aktuelle Element größer als das gesuchte Element ist.

Algorithmus (rekursiv)
1. Wenn f leer ist, ist die Suche erfolglos
2. Wenn f[istart]=s, dann ist die Suche erfolgreich
3. Ansonsten führe die sequenzielle Suche in der Indexmenge set [istart+1,iend] durch

Komplexität der sequenziellen Suche

Schlechtester Fall	O(n)
Durchschnitt (Erfolg)	1/n*sum(i:i=1,...,n)=(n+1)/2 ... O(n/2)
Durchschnitt (Mißerfolg)	O(n)

Anmerkung
Wenn f bereits sortiert vorliegt, dann ist die Durchschnittskomplexität für den Mißerfolg gleich der Durchschnittskomplexität für den Erfolgsfall

Alle folgenden Suchalgorithmen setzen auf die Tatsache, dass f sortiert vorliegt:

1.3.2 Block Search

Voraussetzung
- f ist sortiert (aufsteigend)
- die Indexmenge ist in n/t Blöcke der Größe t aufgeteilt

Algorithmus
1. Suche in den letzten Einträgen der Blöcke (entspricht jeweils dem größtem Schlüssel des Blockes) nach dem ersten Schlüssel, der größer/gleich dem gesuchten Element ist.
2. Wenn diese Suche erfolglos ist, dann ist auch die Suche nach dem Schlüssel erfolglos
3. Ansonsten führe eine sequenzielle Suche im entsprechenden Block durch

Komplexität der Block Search

schlechtester Fall	n/t+t
Durchschnitt (Erfolg)	t/n*sum(i:i=1,...,n/t)+(t+1)/2=(n/t+t)/2+1
Durchschnitt (Mißerfolg)	vergleichbar

1.3.3 Binary Search

Die binäre Suche ist ein sehr einfaches, gleichzeitig aber auch sehr effizientes Verfahren, das in der Praxis oft eingesetzt wird.
Die binäre Suche vergleicht in jedem Schritt das gesuchte Element mit dem mittleren Element des Suchraums.
Sind die beiden Elemente identisch, so bricht die Suche mit Erfolg ab. Ist das mittlere Element kleiner als das gesuchte Element, so wird die Suche in der rechten Hälfte des Suchraums fortgesetzt, andernfalls in der linken Hälfte.
Damit unterteilt das Verfahren in jedem Schritt den Suchraum in zwei Hälften und untersucht nur eine der beiden Hälften weiter, bis das Element gefunden wurde oder das gleiche Element zweimal aufgesucht wird (dann ist das Element nicht enthalten).

Algorithmus
1. Wenn f leer ist, ist die Suche erfolglos
2. Vergleiche den zu suchenden Schlüssel s mit dem mittleren Schlüssel der Indexmenge (imid=(istart+iend)/2)
3. Wenn die Schlüssel identisch ist, ist die Suche erfolgreich
4. Wenn s kleiner als imid dann wiederhole den Algorithmus in der Indexmenge [istart,imid-1]
5. Wenn s größer als imid dann wiederhole den Algorithmus in der Indexmenge [imid+1,iend]

Komplexität der Binary Search (für $n=2^{k-1}$)

schlimmster Fall	log(n+1)
Durchschnitt (Erfolg)	$1/n*sum(2^{(i-1)}*i; i=1,...,k)=(n+1)/n*log(n+1)-1$
Durchschnitt (Mißerolg)	log(n+1)

O(log(n))

Anmerkungen
1. Binary Search kann als der optimale Suchalgorithmus der auf Vergleichen beruht betrachtet werden, wenn keine zusätzlichen Annamen gemacht werden
2. Interpolation Search ist die Abänderung des Binary Search Algorithmus dahingehend, dass die Indexmenge nicht in zwei gleich große Teile aufgespaltet wird, sondern bezüglich der Interpolation an den Suchschlüssel in ein Intervall zwischen dem erstem und dem letzten Schlüssel.

1.3.4 Hashing

Aus dem Schlüsselwert wird die Position des Schlüssels in der Hash Table abgeleitet. Die Funktion, mit der die Schlüssel in die Hash Table abgebildet werden, wird Hash Funktion genannt. Wenn zwei verschiedene Schlüssel auf den selben Position abgebildet werden, nennt man das Kollision. Daher wird eine Strategie benötigt, die Kollision aufzulösen.

Spezialfall: Wenn die Schlüssel in einer bestimmten Reihenfolge gegeben sind, können diese direkt auf die Hash Table abgebildet werden > Table Lookup

Algorithmus
Einfügen
1. Erstelle eine leere Hash Table mit N>n Einträgen
2. Für alle einzutragenden Schlüssel führe die Schritte 3 bis 5 durch
3. Berechne mit Hilfe der Hashfunktion die Einfügeposition von s
4. Wenn die Position nicht leer ist, suche eine alternative Einfügeposition anhand der gegebenen Kollisions – Auflösungs – Strategie
5. Füge den Schlüssel in die Einfügeposition ein

Suchen
1. Berechne die Position des Suchschlüssels anhand der Hashfunktion
2. Starte bei der so ermittelten Position und verfolge die Auflösungsstrategie bis der Schlüssel gefunden wird

1.3.4.1 Hashfunktionen

Bedingungen
1. Sie müssen deterministisch sein
2. Sie sollten möglichst einfach sein
3. Sie sollten die Schlüssel gleichmäßig über das gesamte Intervall verteilen

Beispiele
1. Wenn die Schlüsseleinträge je ein Maschinenwort darstellen, dann wähle b Bits aus der m Mitte des Wortes und bilde den Eintrag auf den Index [0.2^(b-1)]
2. Wenn die Schlüsseleinträge Integerwerte sind, ist eine Hashfunktion hash(s)=s mod N möglich (mit N als Primzahl)
3. Wenn die Schlüsseleinträge aus mehreren Maschinenwörtern zusammengesetzt sind, bilde den Index durch die logische Verknüpfung der einzelnen Wortbits (Exclusiv – Oder, ...)

Wenn Informationen über die Verteilung der einzelnen Schlüsseleinträge bekannt sind, so macht es Sinn wenn diese Information in einer speziellen Hashfunktion verarbeitet werden.

Bedingungen einer Kollisionsauflösungsstrategie:
Jeder Eintrag in der Hash Table muß ein mögliches Ziel darstellen!

1.3.4.2 Open Hashing

Wenn man das Hashing in einer dynamischen Umgebung betrachtet wird, ist eine Auflösungsstrategie innerhalb der Hash Table ungeeignet. In diesem Fall bietet es sich an für jeden Table - Eintrag (bucked) einen Container anzulegen, der die Mehrfachbelegungen aufnimmt.
Wenn die Hash Table den abstrakten Datentyp Set repräsentiert (Suchschlüssel kommen nicht doppelt vor), so sollte auch der Container durch eine Set implementiert werden. Wenn Suchschlüssel auch mehrfach auftreten können, so ist als Containerdatentyp die ungeordnete Liste zu wählen.

1.4 Sortieren

1.4.1 Voraussetzung

Gegeben ist eine Folge f von Schlüssel in der Indexmenge [istart,...,iend].

1.4.2 Ergebnis

Als Ergebnis der Sortierung erhält man ein Permutation der Folge f, in der die Elemente geordnet sind.

Mögliche zusätzliche Anforderungen an den Algorithmus:
Stabilität - betrachtet die relative Position identischer Schlüssel vor und nach dem Sortieren
Zeitkomplexität (Vergleichsoperationen (C) stehen Austauschoperationen (E) gegenüber) für den schlechtesten Fall und für den Durchschnittsfall (worst case / average case).
Platzkomplexität für in situ Algorithmen und ex situ Algorithmen.

1.4.3 Selection Sort

Algorithmus (ex situ)
1. Initialisiere die Zielfolge als leer.
2. So lange die Quellfolge nicht leer ist, führe die folgenden Schritte aus:
3. Suche das Minimum der Quellfolge
4. Füge das Minimum an das Ende der Zielfolge an
5. Lösche das Minimum aus der Quellfolge

Dieser Algorithmus ist akzeptabel, Korrektheit und Stabilität sind gewährleistet. Jedoch gestaltet sich die Implementierung des Löschens aus der Quellfolge als umständlich. Weiter wird durch die Formulierung des Algorithmus in situ das Anlegen einer Zielfolge und das Löschen in der Quellfolgen durch das Austauschen des jeweiligen Schlüssels mit dem Folgenanfang umgangen.

Algorithmus (in situ)
1. Wenn istart < iend, führe die folgenden Schritte durch:
2. Suche den ersten Index imin mit dem kleinsten Schlüssel
3. Tausche f[istart] und f[imin]
4. Führe den Algorithmus im Indexbereich [istart+1,iend] durch

Komplexität
Die Zeitkomplexität des Algorithmus ergibt sich zu $C*n*(n+1)/2+(n-1)*E$ ... $O(n^2)$

Stabilität
Wenn der Algorithmus ex situ realisiert wird, ist die Stabilität gewährleistet, bei der in situ – Formulierung geht die Stabilitätseigenschaft verloren.

1.4.4 Bubble Sort

(Sortieren durch Austauschen)

Bubble Sort ist ein elementares Suchverfahren, das in vielen Informatikveranstaltungen gelehrt wird (so auch hier).
Die Grundidee des Verfahrens ist es, benachbarte Elemente so zu vertauschen, daß sie sortiert sind.

Algorithmus
1. Wenn istart < iend, führe die folgenden Schritte aus:
2. Beginne mit iend und ende bei istart+1
3. Wenn f[i] < f[i-1], tausche den die Einträge
4. Wiederhole den Algorithmus in der Indexmenge [istart+1,iend]

Korrektheit und **Stabilität** sind gewährleistet.

Komplexität
Worst case 0.5*C*(n*(n+1)-1+E*(n-1)
Average case 0.5*C*n*(n+1)-1+0.25*E*(n-1)*n
$O(n^2)$

Anmerkungen
Bei Bubble Sort wandert durch die Vergleiche mit dem Nachbarn in einem Schritt immer das i-t größte Element an die korrekte Position der Rest der Elemente wird nur " vorsortiert". Bubble Sort ist daher vor allem dann geeignet, wenn die Folge bereits vollständig oder fast vollständig sortiert vorliegt, da es dann linear ist. Bereits im durchschnittlichen Fall aber ist das Verhalten von Bubble Sort nicht mehr günstig.
Im allgemeinen Fall ist Bubble Sort - trotz seiner großen Beliebtheit in der Lehre und der Industrie – daher keine gute Wahl für die meisten Sortieranforderungen.

Modifikation
Normalerweise ist Bubble Sort der schlechteste Algorithmus für das Sortieren durch Vergleichen und Austauschen. Manchmal werden verschieden Abwandlungen von Bubble Sort diskutiert (zum Bsp. stoppe wenn im letzten Durchgang kein Austausch stattgefunden hat, durchlaufe den die Folge in beiden Richtungen – Shaker Sort).
Diese Modifikationen bringen aber in der Regel keine wirkliche Verbesserung.

1.4.5 Insertion Sort

Sortieren durch Einfügen, auch als Insertion Sort bezeichnet, fügt der Reihe nach Elemente in eine bereits sortierte (Teil-)Liste ein, die anfangs leer ist.
Damit ist das Vorgehen dem Sortieren von Spielkarten ähnlich: in jedem Schritt wird eine neue Spielkarte zwischen die bereits sortierten Karten eingefügt.

Algorithmus
1. Beginne mit einer sortierten Folge, welche nur das eine Element f[istart] enthält
2. Für f[i], i=istart+1,...,iend führe den folgenden Schritt durch:
3. Füge f[i] in die sortierte Folge f[istart,...,f[i-1] ein, so dass die sich ergebende Folge sortiert ist

Zu Schritt 3
3.1. Suche die richtige Einfügeposition in der Indexmenge
3.2. Verschiebe alle nachfolgenden Elemente in der Indexmenge um eine Stelle
3.3. Füge das Element in die so freigewordene Position ein

Anm: die Schritte 3.1. und 3.2. können gleichzeitig ausgeführt werden, wenn die Suche durch eine sequenzielle Suche realisiert wird.

Korrektheit und **Stabilität** sind gewährleistet.

Komplexität
Best case: O(n)
Worst case: (C+E)*(n*(n+1)/2-1)
Average case: (C+E)*(n-1+0.5(n*(n+1)/2-1))
$O(n^2)$

Anmerkung
Es liegt die Überlegung nahe, die Suche in Schritt 3.1. als eine binäre Suche zu realisieren. Dies bringt jedoch keine Vorteile, da die Anzahl der Austauschoperationen nicht reduziert wird.

Sortieren durch Einfügen stellt einen Algorithmus dar, der sich besonders für gut vorsortierte Folgen eignet. Auf dieser Eigenschaft beruhen andere Algorithmen wie Shell Sort. Weiter besteht auch die Möglichkeit einen schnellen Sortieralgorithmus zu verwenden, um eine Folge bis z einem bestimmten Grad zu sortieren und zum endgültigen Sortieren noch den Insertion Sort anzuwenden.

1.4.6 Shell Sort

Die Idee, die hinter dem Shell Sort steht, ist die, dass man den Insertion Sort in Unterfolgen mit konstanten Indexversatz wiederholt. Zuerst in kleinen Untersequenzen mit einem großen Indexversatz (die quadratische Komplexität des Insertion Sort stellt hier kein Problem dar), dann mit abnehmendem Indexversatz und schließlich mit der Indexversatz 1.

Die Erfahrung hat gezeigt, das man gute Ergebnisse erzielt, wenn man mit einer Indexversatz von $2^{(k-1)}$ beginnt mit k=floor(log(n))* und nach jedem Durchgang die Indexversatz halbiert.

*floor(f(x)) entspricht dem nach unten abgerundeten Ergebnis

Algorithmus
1. Starte mit einem Indexversatz h
2. Solange h>0, führe die folgenden Schritte durch:
3. Führe ein Insertion Sort für das Intervall $i,i_1,...,i_n$ durch, wobei $i_1=i+h$, $i_2=i+2h$,... und $i_n<=n$ sei
4. Teile h durch 2

Korrektheit und Stabilität
Da der letzte Schritt des Shell Sort ein Insertion Sort darstellt, ist die Korrektheit gewährleistet. Doch obwohl der Basisalgorithmus des Insertion Sort stabil ist, ist der Shell Sort Algorithmus nicht stabil.

Komplexität
Die Analyse des Algorithmus hinsichtlich der Komplexität ist etwas schwierig. Es kann aber gezeigt werden, dass sich die Komplexität im schlechtesten Fall zu $O(n^{1.5})$ und im Durchschnittsfall zu $O(n^{1.25})$ ergibt.

1.4.7 Quick Sort

Quicksort ist eines der populärsten Sortieralgorithmen.
Die Grundidee dieses bereits 1960 von C. A. R. Hoare entwickelten Sortieralgorithmus ist dabei sehr einfach:

Algorithmus (Grundformulierung)
1. Wenn istart < iend, führe die folgenden Schritte durch:
2. Betrachte f[istart] als Teilungsschlüssel
3. Bilde eine Teilfolge aus der Indexmenge istart,...,iend und einen Index ipart so das gilt:
   f[i]<f[istart] für alle i in der Folge istart+1,...,ipart
   f[i]>=f[istart] für alle i in ipart+1,...,iend
4. Tausche die Elemente f[istart] und f[ipart]
5. Wiederhole den Algorithmus in der Indexmenge istart,...,ipart-1
6. Wiederhole den Algorithmus in der Indexmenge ipart+1,...,iend

Das Anlegen der Teilfolge in Schritt 3 kann durch den folgenden Algorithmus realisiert werden. Dadurch ergibt sich eine Folge von f wobei ein Prädikat existiert, welches wahr ist in [istart,ipart] und welches falsch ist für [istart+1,iend].

1. Initialisiere ipart mit istart-1
2. Wenn istart > iend ist der Algorithmus abgeschlossen
3. Wenn istart = iend führe den Schritt 4 durch:
4. Wenn P für f[istart] wahr ist, so ist ipart = istart
5. Beginne mit ipart und suche das erste Element für das P falsch ist -> i1
6. Beginne mit iend und suche das erste Element für das P wahr ist -> i2
7. Wenn i1 < i2, dann tausche f[i1] und f[i2], inkrementiere i1 und dekrementiere i3 um 1
8. Wiederhole den Algorithmus im Intervall [i1,i2]

Korrektheit und Stabilität
Die Korrektheit des Quick Sort beruht auf der Bildung der Teilfolge im Schritt 3.
Die Stabilität wird nicht gewährleistet.

Komplexität

Worst case (sortierte Folge)	$T(n)=C*n+E+T(n-1)$, $n > 1$, $T(1) = 0$
	$\Rightarrow Z(n)=C*(n*(n+1)/2-1)+E*(n-1)$, $n >=1 \ldots O(n^2)$
Avarage case	$T(n)=(C+E/6)*2*\ln(2)*n*\log(n)=O(n*\log(n))$

Anmerkungen zu Quicksort
Quick Sort hat eine sehr einfache Implementierung, ist aber dennoch sehr effizient Es sind wenig Ressourcen nötig: nur ein kleiner Stack für die Rekursion
Quick Sort arbeitet im besten und durchschnittlichen Fall sehr effizient, bei sortierten Eingaben aber sehr schlecht.

Mögliche Abänderungen des Quick Sort Algorithmus
- Betrachte den Mittelwert der Folge f[(istart+iend)/2] als Teilungsschlüssel. Dies Vermindert nicht die Komplexität des schlechtesten Falles, doch die Wahrscheinlichkeit, dass dieser eintritt wird reduziert. Weiter wird dadurch erreicht, dass bereits sortierte Folgen am besten verarbeitet werden.
- Um die Stackbeanspruchung des Quick Sort möglichst gering zu halten, sollten die zuerst die kürzeren und dann die längeren Teilfolgen sortiert werden.
- Es ist auch möglich eine iterative Formulierung des Quick Sort zu finden, indem man den abstrakten Datentyp Stack verwendet.
- Eine Möglichkeit zur Optimierung des Sortierens ist es Quick Sort nur für Folgen ab einer bestimmten Größe einzusetzen (20 – 25) und den Algorithmus mit anderen Algorithmen zu kombinieren (z.Bsp. Insertion Sort)
- Quick Sort eignet sich auch zum in situ - Sortieren externer Datenstrukturen im Direktzugriff.

1.4.7.1 Algorithmus, um das k-kleinste Element in einer unsortierten Folge zu finden

K=istart-1+k
1. Wenn istart<=K<=iend, dann führ die folgenden Schritte durch:
2. Betrachte des erste Element der Folge als Teilungsschlüssel
3. Erzeuge eine Teilfolge von f mit dem Indexbereich istart+1,...,iend.
   Betrachte den Index ipart, für den gilt:
   f[i]<f[istart] für alle i in istart+1,...,ipart
   f[i]>=f[istart] für alle i in ipart+1,...,iend

4. Tausche f[istart] und f[ipart]
5. Wenn ipart=K, dann ist die Suche abgeschlossen
6. Wenn ipart<K, dann wiederhole den Algorithmus im Intervall ipart+1,...,iend
7. Wenn ipart>K, dann wiederhole den Algorithmus im Intervall istart,...,ipart-1

1.4.8 Merge Sort

Algorithmus
1. Wenn die Folge leer ist oder aus einem Element besteht, der Algorithmus ist abgeschlossen. Ansonsten führe die folgenden Schritte durch:
2. Bilde Teilfolgen von f wobei f1 und f2 möglichst gleich groß sein sollten
3. Sortiere die einzelnen Teilfolgen
4. Vermische f1 und f2 auf eine Weise, so dass die Zielfolge die sortierte Folge ist

Der Mischalgorithmus in Schritt 4 kann folgendermaßen realisiert werden:
1. Initialisiere die Zielfolge f als leere Folge
2. Solange der Durchlauf weder durch die Teilfolge f1 noch durch f2 beendet ist, führe die Schritte 3 und 4 durch:
3. Wenn das aktuelle Element in f1 kleiner oder gleich dem Element aus f2 ist, füge es an die Zielfolge f an und erhöhe die aktuelle Position in f1 um 1
4. Ansonsten füge das betrachtete Element aus f2 an die Zielfolge an und erhöhe die aktuelle Position in f2 um 1
5. Hänge den Rest von f1 an f an
6. Hänge den Rest von f2 an f an

Korrektheit und **Stabilität** sind gewährleistet.

Komplexität
Die Platzkomplexität von Merge Sort ergibt sich zu 2*n ... O(n)
Zeitkomplexität: T(n)=2*(C+E)*n+T(n/2) => T(n)=2*(C+E)*n*log(n), n>=1 (T(1)=0!)
O(n*log(n))

1.4.8.1 Iterative Formulierung des Merge Sort

Algorithmus
1. Initialisiere l mit 1
2. Teile die Folge f in zwei möglichst gleichgroße Teilfolgen f1 und f2
3. Betrachte die Folgen f1 und f2 als Quellfolgen
4. Solange l kleiner ist als die Länge der Folge f (n), führe die Schritte 5 bis 7 durch:
5. Mische jeweils Teilfolgen der Länge l aus beiden Quellfolgen in sortierte Zielfolgen der Länge 2l und füge sie zu den Zielfolgen t1 und t2 hinzu, wobei t1 und t2 den Feldern f1 und f2 entsprechen
6. Betrachte die Zielfolgen als Quellfolgen und umgekehrt.
7. Die zu letzt als Quellfolge definierte Folge ist das sortierte Ergebnis

Korrektheit und **Stabilität** sind gewährleistet.

Komplexität
Platzkomplexität 2n ... O(n)
Zeitkomplexität (C+E)*n*(1+log(n)); n>=1 ... O(n*log(n))

Anmerkungen
Die iterative Formulierung des Merge Sort beruht allein auf dem sequentiellen Zugriff auf die Folge. Daher Eignet sich dieser Algorithmus besonders zum sortieren von sequentiellen Daten (Dateien).

1.4.9 Sortieren durch Zählen

Der Algorithmus setzt voraus, dass die Sortierschlüssel über einen geeigneten Indexbereich *imin,...,imax* verteilt sind. Diese Folge wird count genannt und man zählt wie viele Schlüssel den gleichen Wert unabhängig von ihrem aktuellen Index haben.

Algorithmus
1. Initialisiere die Einträge in der Folge count als 0
2. Durchlaufe die Ausgangsfolge f und führe Schritt 3 für jeden Index aus:
3. Erhöhe count[f[i]] um 1
4. Ermittle pos[i] mit i=istart,…,iend folgendermaßen:
   pos[min]=istart,
   pos[i]=pos[i-1]+count[i],
   i=min+1,...,max
5. Durchlaufe die Ausgangsmenge f und führe die Schritte 6 und 7 aus:
6. Füge f[i] an der Position pos[f[i]] in die Folge g ein
7. Erhöhe pos[f[i]] um 1
8. die resultierende Folge g ist dann die sortierte Folge

Korrektheit und **Stabilität** sind gewährleistet.

Komplexität
Die Zeitkomplexität des Algorithmus ergibt sich zu c1*n+c2*m mit m=max-min+1 und somit O(n).
Die Raumkomplexität ist 2*n+2*m, kann aber noch verbessert werden.

1.4.10 Interpolation Sort

Die Idee, die hinter dem Algorithmus steht ist die, die Einfügeposition eines Schlüssels anhand seines Wertes zu ermitteln (vergleichbar mit dem Ansatz von Hashing). Um diesen Gedanken zu realisieren braucht man eine Funktion, die die Schlüssel auf eine Indexmenge abbildet, die streng monoton ansteigend ist.

Wir betrachten den Fall, dass die Indexmenge eine Folge von 0 bis N-1 mit N>n ist. Damit vermindert sich auch die Kollisionswahrscheinlichkeit. Die Einträge in der resultierenden Folge werden buckets genannt (analog zur Formulierung der Hash Table).

Eine häufig verwendete Funktion ist die der Bestimmung eines Zwischenwertes (Interpolation) für den jeweiligen Folgewertes mit Hilfe des Minimumwertes min und des Maximumwertes max innerhalb des Intervalls 0,...,N-1:

$F(s) = ((s-min)*(N-1))/(max-min)$

Wie in der Formulierung des Hash – Algorithmus wird auch hier eine Kollisionsauflösungsstrategie benötigt. Eine Möglichkeit ist es eine sortierte Liste für jeden Eintrag (bucked) anzulegen.

Die Interpolation kommt jedoch nur bei der Verteilung der Schlüssel auf die Indexmenge zur Anwendung. Anschließend muß ein anderer Sortieralgorithmus für die bucket – Einträge angewandt werden. In dem Fall, dass die Zahl der Einträge relativ gering ist, wird Insertion Sort verwendet, ansonsten wählt man Merge Sort (Merge Sort und Insertion Sort eignen sich besonders zum Sortieren von Listen).

Die Kombination aus dem Verteilen der Schlüssel und einem anschließenden Sortieren der buckets wird **Bucked Sort** genannt.

1.4.11 Bottom Up Radix Sort

Dieser Algorithmus beruht auf der Wiederholung des Sorting By Counting – Algorithmus, um eine Folge von Schlüsseln zu sortieren, die als bestimmte Bytemuster betrachtet werden (z.Bsp. Strings, es können jedoch auch Integerwerte betrachtet werden). Vom minderwertigsten Bit zum höchstwertigsten Bit wird das Sortieren durchgeführt, wobei das jeweils aktuelle Bit als Schüssel betrachtet wird.

Algorithmus

1. Betrachte die Schlüssel als d Teile vom Umfang m
   (m=256 -> Aufteilung in Bytes, m=2 -> Aufteilung in Bits)
2. Wende Sorting By Counting vom LSB zum MSB an, wobei das aktuelle Bit als Schlüsselwert betrachtet wird

Korrektheit und **Stabilität** werden hierbei für die ex situ Variante gewährleistet.

Komplexität

Die Zeitkomplexität ist linear, jedoch ist für den Fall, dass die Aufteilungsmuster zu komplex sind, ist der Quick Sort überlegen. Wenn diese Muster relativ klein gehalten werden (z.Bsp. bei Integerwerten, die aus jeweils 4 Bytes bestehen), ist die Sorting By Counting Formulierung dem Quick Sort überlegen.

Anmerkung

Bottom Up Radix Sort kann mit dem Insertion Sort kombiniert werden – eine Möglichkeit wäre dann, nur die höherwertigen Bits mit dem Bottom Up Radix Sort zu bearbeiten und dann einen abschließenden Sortierdurchlauf mittels Insertion Sort durchzuführen.

1.4.12 Sortieralgorithmen für (verlinkte) Listen

1. Selection Sort eignet sich aufgrund der Linearität der Listen
2. Bubble Sort kann bei doppelt verlinkten Listefan angewandt werden
3. Insertion Sort baut auf der Linearität der Liste auf. Dieser Sortieralgorithmus ist für verlinkte Listen sogar besser geeignet, da kein Verschieben der Elemente notwendig ist
4. Shell Sort bildet für das Sortieren Teilfolgen des Indexes und ist deshalb für eine Anwendung bei verlinkten Listen ungeeignet
5. Quick Sort kann bei einfach verlinkten Listen angewendet werden, arbeitet aber effizienter bei doppelt verlinkten Listen
6. Merge Sort eignet sich aufgrund der Linearität der Listen
7. Sorting By Counting und Bottom Up Radix Sort eignen sich ebenfalls aufgrund der Linearität der Listen
8. Bucked Sort ist im Zusammenhang mit Insertion Sort oder Merge Sort ebenfalls für verlinkte Listen geeignet, da wieder nur die Linearität der Liste von Bedeutung ist. Es kann auch als Gegenstück zum Open Hashing betrachtet werden

2 Hierarchische Datenstrukturen (Bäume)

Der Baum (engl. Tree) ist eine der wichtigsten Datenstrukturen der Informatik. Bäume sind zwei- oder mehrdimensionale Datenstrukturen, bei denen ein Element mit mehreren anderen Elementen verknüpft sein kann. Sehr viele Algorithmen arbeiten auf Bäumen.
Bäume werden in vielen Bereichen in und außerhalb der Informatik eingesetzt, z.B.
- Stammbäume in der Ahnenforschung,
- Ablaufstrukturen bei Turnieren wie der Fußball-Weltmeisterschaft (ab der Qualifikationsrunde),
- hierarchischer Aufbau von Unternehmensstrukturen,
- Syntaxbäume sowie Ableitungsbäume bei der Verarbeitung von Computersprachenprogrammen

Es gibt mehrere Untertypen von Bäumen. Zunächst soll aber ein überblick über die Terminologie egeben werden.

2.1 Definition

Ein Baum nennt man eine hierarchische Datenstruktur, in welcher alle Elemente (ausgenommen eines) einen eindeutigen Vorgänger (parent) und beliebig viele Nachfolger (children) besitzt. Das einzige Element ohne Vorgänger wird Wurzel des Baumes genannt. Die größte Anzahl von Nachfolgern eines Elementes entspricht dem Grad des Baumes.

Ein binärer Baum ist ein Baum mit dem Grad 2.

Die Elemente eines Baumes werden Knoten genannt. Knoten ohne Nachfolger heißen Blätter. Wenn man Knoten mit Vorgänger und Nachfolger betrachtet spricht man auch von internen Knoten.

Die Höhe eines Baumes wird durch die maximale Tiefe seiner Knoten definiert.

Die Pfadlänge eines Baumes ergibt sich wenn man die Tiefe eines jeden Knotens aufaddiert.

Ein Baum ist genau dann voll, wenn alle internen Knoten vom selben Grad sind und alle Blätter in der selben Tiefe liegen.

Ein Baum aus dem sich ein anderer Baum zusammensetzt wird Unterbaum genannt. Ein Baum, der verschiedene Unterbäume beinhaltet heißt Superbaum.

2.2 Binäre Suchbäume

Definition
Ein binärer Suchbaum ist ein Baum mit den folgenden Eigenschaften:
- Jeder Knoten enthält einen Schlüsseleintrag von einem ordinalen Datentyp (um Vergleiche durchführen zu können)
- Alle Schlüssel des linken Teilbaumes sind kleiner als die Schlüssel des rechten Teilbaumes
- Alle Schlüssel des rechten Teilbaumes sind größer oder gleich dem Wurzelschlüssel

2.2.1 Durchläufe in Binärbäumen

Da man einen Binärbaum immer als Wurzel, einem rechten Unterbaum (mit dem rechten Nachfolger als neue Wurzel) und einem linken Unterbaum (mit dem linken Nachfolger als neue Wurzel) betrachten kann, kann man drei verschiedene Formen von Durchläufen unterscheiden:

1. **Inorder** durchlaufe zuerst den linken Unterbaum, gehe dann zur Wurzel und durchlaufe zum Schluß den rechten Unterbaum
2. **Preorder** gehe zuerst zur Wurzel des Baumes, durchlaufe dann den linken Unterbaum und dann den rechten Unterbaum
3. **Postorder** durchlaufe zuerst den linken Unterbaum, dann den rechten Unterbaum und gehe zum Schluß zur Wurzel

Bei einem Inorder – Durchlauf durch einen binären Baum erhält man eine sortierte Liste der Schlüsseleinträge. Aus diesem Grund erhält der binäre Suchbaum eine Bedeutung beim Sortieren von Daten.

Anmerkung
Ein binärer Baum kann sowohl durch die Pre- als auch durch die Postorder – Bedingung aufgebaut werden.

2.2.2 Implementierung einer ADT – Menge als binären Suchbaum

Jeder Knoten des Baumes besteht aus einem Schlüsseleintrag, einem Zeiger auf den linken Nachfolger und einem Zeiger auf den rechten Nachfolger. Die Implementierung mit einem zusätzlichen Zeiger auf das Wurzelelement wird nicht betrachtet, da es nur in einigen speziellen Situationen benötigt wird.
Der Suchbaum selbst wird durch den Zeiger auf seine Wurzel repräsentiert.

Für binäre Suchbäume werden im Folgenden drei Basisoperationen betrachtet:

a) **Suche** (nach einem Knoten mir dem Schlüssel s)
1. Wenn der Baum leer ist, ist die Suche gescheitert
2. Ansonsten durchlaufe die Schritte 3 bis 5:
3. Ist der Schlüssel der Wurzel gleich dem gesuchten Schlüssel s ist die Suche positiv
4. Wenn s kleiner als der Wert der Wurzel ist, dann suche im linken Teilbaum
5. Wenn s größer als der Wert der Wurzel ist, dann suche im rechten Teilbaum

b) **Einfügen** (eines Knotens mit dem Schlüssel s)
1. Wenn der Baum leer ist, dann füge den Knoten als Wurzel ein
2. Ansonsten durchlaufe die Schritte 3 bis 5:
3. Wenn s gleich dem Wert des Schlüssels ist, so kann der Knoten nicht eingefügt werden
4. Wenn s kleiner als der Wert der Wurzel ist, so füge den Knoten in den linken Teilbaum ein
5. Wenn s größer als der Wert der Wurzel ist, so füge den Knoten in den rechten Teilbaum ein

c) **Löschen** (eines Knotens mit dem Schlüssel s)
1. Wenn der nicht leer ist, führe die folgenden Schritte durch:
2. Wenn s kleiner als der Wert der Wurzel ist, führe das Löschen im linken Teilbaum durch
3. Wenn s größer als der Wert der Wurzel ist, führe das Löschen im rechten Teilbaum durch
4. Ansonsten (s ist gleich dem Wert der Wurzel) führe die folgenden Schritte durch:
5. Wenn beide Teilbäume leer sind, lösche die Wurzel
6. Wenn der linke Teilbaum leer ist, dann ersetze die Wurzel durch die Wurzel des rechten Teilbaumes
7. Wenn der rechte Teilbaum leer ist, dann ersetze die Wurzel durch die Wurzel des linken Teilbaumes
8. Ansonsten führe die folgenden Schritte durch:
9. Ersetze die Wurzeleintrag durch den kleinsten Schlüssel aus dem rechten Teilbaum
10. Lösche im rechten Teilbaum den Knoten mit dem kleinsten Schlüssel (smin)

Das Suchen/Löschen des Knotens smin in Schritt 9/10 kann folgendermaßen geschehen:
1. Wenn der linke Teilbaum leer ist, führe Schritt 2 aus:
2. Speichere den Schlüssel des Knotens und ersetze ihn durch die Wurzel des rechten Teilbaumes
3. Ansonsten wiederhole die Operation im linken Teilbaum

Komplexität
Da die Zeitkomplexität des Einfügens bzw. des Löschens eines Knotens hauptsächlich auf dem Suchen der Einfüge- bzw. Löschposition beruht, reicht es, wenn man nur die Komplexität des Suchens betrachtet.

Den schlimmsten Fall erhält man wenn die Schlüssel in sortierter Reihenfolge in den Baum eingefügt wurden. In diesem Fall entartet der Baum zu einer dynamischen Liste, in der man eine sequentielle Suche der Komplexität O(n) durchzuführen hat.

Im besten Fall ist der Baum voll, dann ist die Höhe des Baumes gleich ceil(log(n))* und damit ergibt sich die Komplexität O(log(n)).

Im Durchschnitt ist die Suchoperation von der Komplexität 2*ln(2)*log(n) und damit O(log(n)).

*ceil(f(x)) entspricht dem nach oben abgerundeten Ergebnis

Anmerkung
Die Komplexitätsuntersuchung ist ähnlich der des Quick Sort!

2.3 Dynamisch optimierte binäre Suchbäume (AVL Bäume)

Mit binären Suchbäumen erhält man im Durchschnittsfall gute Ergebnisse. Für den schlimmsten Fall sind diese Ergebnisse jedoch nicht mehr akzeptabel. Daraus ergibt sich das Problem, wie die Entartung eines binären Suchbaumes vermieden werden kann.

Die ersten, die Operationen vorgestellt haben, die dieses Problem lösen, waren Adelson-Velskii und Landis; Bäume, die nach deren Operationen konstruiert wurden nennt man deshalb AVL-Bäume.

Definition
Für einen binären Suchbaum wird eine Balancezahl definiert, die sich aus der Differenz aus dem rechten und dem linken Teilbaum ergibt.
Von einem AVL-Baum spricht man dann, wenn sich für jeden Knoten des Baumes eine Balancezahl von –1,0 oder 1 ergibt.

Die AVL Eigenschaft eines Baumes kann dann verloren gehen, wenn entweder ein Knoten gelöscht oder hinzugefügt wird. Deshalb ist es wichtig diese Operationen genauer zu untersuchen:

Einfügen
Wenn die Balancezahl des Baumes gleich –1 (1) ist und die Höhe des rechten (linken) Teilbaumes zunimmt, dann ist der Baum nach dem Einfügen ausgeglichen und die Höhe bleibt unverändert.

Wenn der Baum ausgeglichen (Balancezahl = 0) ist und die Höhe des linken (rechten) Teilbaumes zunimmt, dann ändert sich die Balancezahl zu 1 (-1) und die Höhe des Baumes hat zugenommen.

Wenn die Balancezahl des Baumes –1 (1) ist und die Höhe des linken (rechten) Teilbaumes zunimmt, so ist die AVL Eigenschaft verloren und zusätzliche Operationen müssen gefunden werden.

Löschen
Wenn die Balancezahl des Baumes gleich –1 (1) ist und die Höhe des linken (rechten) Teilbaumes nimmt ab, dann ist der Baum am Ende ausgeglichen und die Höhe bleibt unverändert.

Wenn ein Baum ausgeglichen ist und die Höhe des linken (rechten) Teilbaumes nimmt ab, so ist die neue Balancezahl des Baumes –1 (1) und die Höhe bleibt unverändert.

Wenn die Balancezahl des Baumes gleich –1 (1) ist und die Höhe des rechten (linken) Teilbaumes nimmt ab, so ist die AVL Eigenschaft verloren und es müssen zusätzliche Operationen gefunden werden.

Operationen zur Wiederherstellung der AVL Eigenschaft

Wegen der Symmetrie des Problems, werden hier nur zwei Fälle betrachtet:
1. Die Balancezahl des Baumes ist –2, die des linken Teilbaumes ist kleiner oder gleich null.
2. Die Balancezahl des linken Teilbaumes ist –2, die des rechten ist größer als null.

zu 1.)
Im ersten Fall, wird die Operation, die den Baum in einen AVL Baum zurückformatiert (linke) Einfachrotation genannt:
r sei die Wurzel des Baumes; rl ist dann die Wurzel des linken Teilbaumes, LST und RST bezeichnen den linken sowie den rechten Teilbaum von r und LSTl und RSTl seien der linke bzw. der rechte Teilbaum von rl.
Bei einer (linken) Einfachrotation wird dann rl zur Wurzel des Baumes, r ist nach dieser Operation die Wurzel des rechten Teilbaumes, mit RSTl als linken Teilbaum und RST als rechten Teilbaum. LSTl wird zum linken Teilbaum von r.

Nach dieser Operation sind dann folgende Zustände möglich:
- Wenn der Balancefaktor des linken Teilbaumes –1 war, ist der Baum nach der Rotation ausgeglichen und die Höhe wurde um eins vermindert.
- Wenn der linke Teilbaum ausgeglichen war, verändert sich der Balancefaktor des Baumes bei der Rotation zu 1, die Höhe bleibt dabei unverändert

zu 2.)
Im zweiten Fall wird eine Operation durchgeführt, die (links rechts) Doppelrotation genannt wird:
r sei die Wurzel des Baumes; LST und RST sind entsprechend der linke und der rechte Teilbaum von r. Weiterhin sei rl die Wurzel des linken Teilbaumes mit wiederum LSTl und RSTl als Teilbäume. Darüber hinaus sei noch rlr die Wurzel des rechten Teilbaumes von rl (von RSTl) mit den Teilbäumen LSTlr und RSTlr.
Bei der (links rechts) Doppelrotation wird dann rlr zur Wurzel des Baumes, rl ist die Wurzel des linken Teilbaues mit LSTl als linken und LSTlr als rechen Teilbaum. r wird Wurzel des rechten Teilbaumes mit wiederum RSTlr und RST als Teilbäume.

Nach dieser Operation sind folgende Zustände möglich:
- Der Baum ist nach der Rotation ausgeglichen und die Höhe hat sich um eins verringert
- Wenn der Baum mit der Wurzel rlr ausgeglichen war, sind der linke und der rechte Teilbaum nach der Rotation wiederum ausgeglichen. Wenn rlr einen negativen Balancefaktor besessen hat, hat der linke Unterbaum nach der Operation einen positiven Balancefaktor und der rechte Teilbaum ist ausgeglichen. Ansonsten, wenn rlr positiv balanciert war, ist der linke Teilbaum anschließend ausgeglichen und der rechte Teilbaum hat einen negativen Balancefaktor.

Komplexität
Die Zeitkomplexität in einem AVL Baum für den schlechtesten Fall kann ermittelt werden, indem man die AVL Bäum mit der kleinsten Knotenanzahl bei einer festen Höhe h untersucht.

Wir gehen davon aus, das ein Baum der Höhe n nur dann die AVL Eigenschaft besitzt, wenn der Balancefaktor der Knoten –1, 0 oder 1 ist. Der schlechteste Fall liegt dann vor, wenn der linke Teilbaum der schlechteste Teilbaum der Höhe h-1 und der rechte Teilbaum der schlechteste Teilbaum der Höhe h-2 ist.
Folglich erhält man:

N(0) = 0,
N(1) = 1,
N(2) = n(h-1)+n(h-2)+1; n >= 2.
...

Diese Art von Bäumen wird auch Fibonacci – Baum genannt. Die sich ergebende Zahlenfolge ist die Leonardo – Folge.

Es kann somit gezeigt werden, dass gilt: h(n) <= 1,44*(log(n+1)+1)
und sich damit die Komplexität für den schlechtesten Fall zu O(log(n)) ergibt.

Weiter kann auch gezeigt werde, dass sich die Komplexität für den Durchschnittsfall zu 1,01*log(n) mit n >= 1000 ergibt. Dieses Ergebnis ist damit fast identisch mit dem Optimum für binäre Suchbäume.

2.4 Blattorientierte binäre Suchbäume

Die binären Suchbäume, die wir bis jetzt betrachtet haben, waren knotenorientierte Bäume. Das heißt, dass die Schlüsseleinträge auf alle Knoten verteilt wurden, sowohl auf interne Knoten als auch auf Blattknoten.
Bei blattorientierten Bäumen werden die Schlüssel nur in Blattknoten eingetragen. Die internen Knoten dienen nur zur Navigation. Auf den ersten Blick scheint dies nicht besonders effizient, da bei dieser Konstruktion alle Schlüssel bis auf den kleinsten zweimal eingetragen werden: einmal in die internen Knoten und dann noch in die Blattknoten. Es besteht aber die Möglichkeit die Knoten bei blattorientierten Bäumen als doppelt verlinkte sortierte Listen zu organisieren, so dass sowohl ein direkter Zugriff als auch der sequentielle Zugriff ermöglicht wird.

Eine weitere Anwendung der blattorientierten Bäumen sind Strukturen mit Mehrfachverzweigungen, wie zum Beispiel externe Daten. Hierbei wird die Höhe des Suchbaumes reduziert, da mehr Elemente in den internen Knoten Platz finden.

Für blattorientierte Bäume werden im Folgenden drei Operationen betrachtet:
Suchen, Einfügen, Löschen.

a) **Suchen**

1. Wenn der Baum leer ist, ist die Suche erfolglos
2. Ansonsten führe die folgenden Schritte aus:
3. Solange der aktuelle Knoten kein Blattknoten ist, führe die Schritte 4 und 5 aus
4. Wenn der Suchschlüssel kleiner als der Eintrag des aktuellen Knotens ist, suche im linken Teilbaum
5. Ansonsten, wenn der Suchschlüssel kleiner als der Eintrag des aktuellen Knotens ist, suche im rechten Teilbaum
6. Wenn der gefundene Blattknoten gleich dem gesuchten Schlüssel ist, dann ist die Suche erfolgreich

b) **Einfügen** (des Schlüssels s)
1. Wenn der Baum leer ist, führe Schritt 2 aus:
2. Erzeuge einen neuen Blattknoten mit dem Schlüssel s und ersetze die Baumreferenz durch eine Referenz auf den neuen Knoten
3. Ansonsten führe die folgenden Schritte durch:
4. Suche den Blattknoten lp (den Zeiger p auf den Knoten) in den der neue Schlüssel eingetragen werden soll
5. Erzeuge einen neuen Blattknoten ln mit dem Schlüssel s
6. Wenn ln kleiner ist als lp, dann fahre mit Schritt 7 fort:
7. Erzeuge einen neuen internen Knoten ns mit dem Wert s, lp sei linkes und ln rechtes Blatt
8. Ansonsten, wenn ln größer als lp, fahre mit Schritt 9 fort:
9. Erzeuge einen neuen internen Knoten mit dem Wert s, ln sei linkes und lp rechtes Blatt
10. Ersetze p durch einen Zeiger auf den neuen Knoten ns

Anmerkung
Die Schritte 6 und 7 werden nur dann ausgeführt, wenn ein Schlüssel eingefügt werden soll, der kleiner als alle Schlüssels des Baumes ist. Wenn man jetzt einen Dummy-Knoten hinzufügt, der kleiner ist als alle möglichen Einträge in den Baum, dann werden die Schritte 6 und 7 überflüssig.

c) **Löschen**
1. Falls der Baum nicht leer ist, führe die folgenden Schritte aus:
2. Wenn der Baum aus einem einzelnen Blattknoten besteht, führe Schritt 3 aus:
3. Setze den Referenzzeiger des Baumes auf NULL (Baum ist leer) und lösche den Blattknoten
4. Ansonsten fahre wie folgt fort:
5. Suche (den Zeiger lp auf) den Vorgänger p des Blattknotens l, der gelöscht werden soll.
6. Wenn die Suche erfolgreich war, dann führe die folgenden Schritte aus:
7. Wenn l der rechte Nachfolger von p ist, ersetze lp durch einen Zeiger auf den linken Nachfolger von p
8. Ansonsten, wenn l der linke Nachfolger von p ist, ersetze lp durch einen Zeiger auf den rechten Nachfolger von p
9. Lösche den Zeiger p und den Blattknoten l

2.5 B – Bäume und B* - Bäume

Wenn man mehrfachverzweigte Baumstrukturen betrachtet, können die Vorgehensweisen, die bei binären Bäumen zum Einsatz kommen, nicht verwendet werden.
Deshalb wird hier u.a. das Modell der B Bäume betrachtet, welches erstmals von den Informatikern Bayer und Mc Creight vorgestellt wurde. Diese Betrachtung spielt eine wichtige Rolle, wenn es um die Implementierung externer Baumstrukturen bzw. um Datenbanken geht.

2.5.1 Definition

Ein mehrfachverzweigter Baum ist dann ein B Baum vom Grad m wenn folgende Bedingungen erfüllt sind:

1. Alle Blätter liege in der selben Tiefe
2. Die Anzahl k der Schlüsseleinträge in einem Wurzelknoten ist 1<=k<=2m bzw. m<=k<= 2m in allen anderen Knoten
3. Die Anzahl der Nachfolger eines Knotens sind s=0 bei Blattknoten und s=k+1 bei internen Knoten

Alle Bedingungen müssen auch nach beliebigen Operationen noch erhalten bleiben.

Die Grundoperationen der B Bäume können folgendermaßen beschrieben werden:

a) **Suchen**

1. Suche den Knoten, in dem der Schlüsseleintrag vorkommen müßte
2. Suche im Knoten nach dem Schlüsseleintrag

b) **Einfügen**

1. Suche den Knoten, in dem das Einfügen stattfinden muß
2. Füge den Eintrag in dem gefundenen Knoten ein
3. Wenn der Knoten bereits voll ist, dann führe eine geeignete Überlaufoperation durch

c) **Löschen**

1. Suche den Knoten, in dem der Schlüsseleintrag vorkommen müßte
2. Lösche den Schlüssel aus dem Knoten
3. Wenn der Füllgrad des Knotens 50% unterschreitet, dann führe eine geeignete Unterlaufoperation durch

Überlaufoperationen könnten folgendermaßen realisiert werden:

1. Wenn es einen Nachbarknoten mit weniger als 2m Schlüsseleinträgen gibt, verschiebe einige Einträge aus dem aktuellen Knoten. Dann muß noch der alte Teilungsschlüssel des Vorgängers durch den jeweils neuen ersetzt werden.
2. Erzeuge einen neuen Knoten. Die kleinsten m Einträge werden im alten Knoten, in dem der Überlauf aufgetreten ist belassen. Die anderen m Einträge werden in den neuen Knoten verschoben. Der Eintrag in der Mitte wird im Vorgänger hinzugefügt. Wenn dadurch ebenfalls eine Überlaufsituation auftritt, muß auch hier eine entsprechende Operation angewandt werden.

Unterlaufoperationen könne folgendermaßen realisiert werden:

1. Wenn ein Nachbarknoten mit mehr als m Knoten existiert können hier einige Einträge entfernt und in den betrachteten Knoten eingefügt werden. Jetzt muß noch der Teilungsschlüssel des Vorgängers aktualisiert werden.

2. Wenn ein Nachbarknoten mit m Einträgen existiert, dann muß dieser mit dem betrachteten Knoten und dem Teilungsschlüssel zu einem neuen Knoten zusammengeführt werden. Der neue Schlüssel wir dann mittels der Überlaufoperationen organisiert.

In praktischen Implementierungen werden die Unterlaufoperationen nicht weiter betrachtet, da es in der Regel mehr Schlüssel eingefügt, als gelöscht werden.

Komplexität

Besonders interessant bei dieser Struktur ist die Zeitkomplexität der Suche, da sich die Komplexität des Einfügens und Löschens im wesentlichen auf diese Betrachtung zurückführen lässt.

Die Zeitkomplexität der Suchoperation ergibt sich aus der Anzahl der Zugriffe auf die Baumknoten. (praktisch gesehen sind diese Knoten Blöcke auf einem Datenträger – somit kann die Suche nach dem Schlüssel innerhalb des Knotens gegenüber den Zugriffen auf die Knoten vernachlässigt werden).

Um einen Eindruck von der Zeitkomplexität in B Bäumen zu bekommen, betrachten wir den Fall, dass 100 Schlüsseleinträge pro Knoten (und damit auch mehr als 100 Nachfolger für jeden Knoten) existieren.
Wenn diese Annahmen erfüllt sind, erhält man:
100 Schlüsseleinträge bei einer Baumhöhe von 1
10000 Schlüsseleinträge bei einer Baumhöhe von 2
1000000 Schlüsseleinträge bei einer Baumhöhe von 3
...

2.5.2 B* Bäume

Wenn es notwendig ist B Bäume zu implementieren (z.Bsp. im Zusammengang mit Datenbanksystemen) wird gewöhnlich nicht die einfache B Baum Struktur verwendet, sondern eine geringfügig abgeänderte Variante. Diese blattorientierten B Bäume werden dann B* Bäume genannt. Die Vorteile dieser Formulierung sind die gleichen wie bei blattorientierten binären Suchbäumen im Vergleich zu knotenorientierten Bäumen.
Eine Möglichkeit der Organisation sind hier wieder die doppelt verlinkten Listen, so dass ein sequentieller Zugriff ebenso ermöglicht wird ein direkter Zugriff.
Üblich ist, dass bei B* Bäumen sowohl in den internen Knoten als auch in den Blattknoten nur Schlüsseleinträge (und bei internen Knoten Zeiger auf Nachfolger) vorkommen bzw. sind in Blattknoten Daten abgelegt, die dann auf die Schlüssel schließen lassen.
Im Falle, dass Schlüssel doppelt vorkommen können sog. Overflow Listen für die Blattknoten angelegt werde, die dann auf gleiche Weise wie die Bäume behandelt werden. Das einzige Problem, das dabei entsteht ist die Schwierigkeit der Split Operation, bei der die Beziehungen zwischen den Blattknoten und Overflow Listen verloren gehen kann und eine entsprechende Reorganisation gestaltet sich oft als schwierig.

2.6 ADT Prioritätsschlange (Priority Queue)

Der ADT der Prioritätsschlange kann als Verallgemeinerung von Stack und Queue angesehen werden. Das Prinzip ist jetzt nicht mehr FIFO oder LIFO sondern es gestaltet sich so, dass immer das beste Element zuerst herausgenommen wird, wobei dann eine Definition benötigt wird, die eine Aussage trifft, welches von zwei Elementen das bessere ist. Im Falle des Stacks ist diese Priorität die des späteren Einfügens und bei der Queue die des früheren Einfügens.
Wenn wir nach einer Struktur suchen, die diese Bedingungen zulässt scheint es naheliegend eine geordnete Liste zu implementieren, wobei die Ordnung der Priorität folgt. Die Lösch Operation wird dadurch trivial, jedoch hat das Einfügen, welches dem Suchen der Einfügeposition entspricht eine lineare Komplexität, was diesen Ansatz unbrauchbar macht.
Eine andere Möglichkeit ist es, einen (dynamisch optimierten) binären Suchbaum zur Grundlage der Überlegung zu machen. Da sowohl das Löschen als auch das Einfügen wiederum der Such-Operation entsprechen, ist hier die Komplexität logarithmisch. Jedoch wird eine zusätzliche Betrachtung notwendig, für den Fall, dass Prioritätseinträge doppelt vorkommen.

Eine weitere Möglichkeit, eine Prioritätsschlange zu formulieren ist die, keine Suchbäume zu verwenden, sondern eine Struktur zu finden, bei der jeweils der größere Schlüssel (mit der höheren Priorität) im Wurzelknoten untergebracht ist. So wird das Löschen wie in den

geordneten listen trivial. Das Einfügen vereinfacht sich dann, wenn man als Grundstruktur einen Heap wählt.

Definition
Ein binärer Baum wird dann Heap genannt, wenn folgende Bedingungen erfüllt sind:
1. Für jeden Knoten gilt, dass der Schlüsseleintrag größer oder gleich den Einträgen der Nachfolger ist.
2. Der Baum ist komplett

Für die Implementierung von Heaps eignen sich Arrays, wobei die Wurzel dem Index 0 entspricht und sich die Nachfolger des Knotens an der Stelle N an den Positionen 2N+1 und 2N+2 befinden.

Die Heapbedingungen lassen sich deshalb auch folgendermaßen formulieren:
f[2N+k]<=f[N]; k=1,2

2.7 Heapify Algorithmus

Die Grundoperation der Heap – Organisation ist das Einfügen. Wir betrachten den Fall, dass f bereits einen Heap darstellt, f[istart+1],...,f[n] erfüllen die Heapeigenschaften und ein Schlüssel s soll hinzugefügt werden, so dass auch nach dem Einfügen diese Eigenschaft gewährleistet wird.

Algorithmus
1. Beginne mit der Betrachtung des Indexes k=istart wobei f[istart] der neu hinzugekommene Schlüssel s ist und istart beim Index 0 beginnt
2. Wenn 2k+1 nicht im Indexbereich liegt, dann ist die Operation abgeschlossen
3. Ansonsten führe die folgenden Schritte durch:
4. Wenn f[k]<f[2k+1], dann führe die Schritte 5 und 6 aus:
5. Vertausche f[k] und f[2k+1]
6. Betrachte 2k+1 als den neuen Index k
7. Ansonsten führe die folgenden Schritte durch:
8. Wenn 2k+2 nicht im Indexbereich liegt, dann ist die Operation abgeschlossen
9. Ansonsten fahre wie folgt fort:
10. Wenn f[k]<f[2k+2] dann führe die Schritte 11 und 12 aus
11. Vertausche f[k] und f[2k+2]
12. Betrachte 2k+2 als den neuen Index k

Komplexität
Da die Länge des Heaps der Knotenanzahl eines Baumes entspricht, ist die Komplexität O(n)=log(n)

2.8 Heap Sort

Das Heapify – Verfahren kann auch dazu verwendet werden um einen weiteren Suchalgorithmus zu formulieren. Dieser wird dann Heap Sort genannt und kann mit dem Selection Sort verglichen werden. Jedoch führt das Verwenden eines Heaps für die Maximumsuche zu einer logarithmischen Zeitkomplexität im Gegensatz zur linearen Komplexität der Minimumsuche beim Selection Sorts in einer ungeordneten Liste.

2.8.1 Stufe 1 (Aufbau des Heaps)

Für den Aufbau des Heaps gibt es zwei Möglichkeiten. Bei der ersten Variante beginnt man mit einer leeren Folge, die man als Heap betrachtet und fügt jedes Element der zu sortierenden Folge

nach dem Einfügealgorithmus am Ende des hinzu. Die zweite Variante betrachtet zuerst alle Elemente, die keinen Nachfolger besitzen und fügt am Ende eines leeren Arrays der Länge der zu sortierenden Folge ein. Danach werden alle anderen Elemente hinzugefügt.

Algorithmus
Variante 1
1. Lege eine neue Liste an und betrachte diese als Heap
2. Füge den Schlüssel s am Ende des Heaps hinzu
3. Betrachte den Index in den s eingefügt wurde als den Index k
4. Vertausche s solange mit dem Vorgängerschlüssel f[⌊(k-1)/2⌋] solange dieser existiert und kleiner als s ist

Variante 2
1. Betrachte f[⌊(n+1)/2⌋],...,f[n-1] als Heap
2. Für k von ⌊(n+1)/2⌋-1 bis 0 führe den Heapify Algorithmus mit f[k] als einzufügendes Element und f[k+1],...,f[n-1] als Heap durch

2.8.2 Stufe 2 (Sortieren der Folge)

Algorithmus
1. Führe für k=n-1 bis k=2 die folgenden Schritte durch:
2. Vertausche f[k] und f[0]
3. Führe den Heapify Algorithmus mit f[0] als einzufügendes Element und f[1],...,f[k-1] als Heap durch

Komplexität
Da die Zeitkomplexität des Heapify Algorithmus logarithmisch ist und dieser Algorithmus in der Aufbaustufe n/2 mal und in der Sortierstufe n mal angewandt wird, ergibt sich für die Komplexität des Heap Sort O(n*log(n)) (schlechtester Fall).
Somit kann Heap Sort als Alternative zum Quick Sort mit seiner quadratischen Komplexität im schlechtesten Fall und zum Merge Sort mit seiner schlechten Platzkomplexität betrachtet werden.

3 Zusammenfassung

3.1 Suchverfahren

Sequentielle Suche O(n)
Binäre Suche O(log(n)) -> *Bedingung: sortierter Vektor*
Hashing (closed/open) O(1) worst / O(n) average

3.2 Sortierferfahren

Bubble Sort $O(n^2)$ Prinzip:	*Tausch*
Quick Sort $O(n^2)$ wc / O(nlog(n)) av	
Insertion Sort $O(n^2)$	*Einfügen*
Shell Sort	
Selection Sort $O(n^2)$	*Auswahl*
Heap Sort O(nlog(n))	
Merge Sort O(nlog(n))	*Mischen*

*-> Algorithmen, die auf Vergleichen und Tauschen beruhen erreichten nie eine bessere Komplexität als O(n*log(n))*

Sortieren durch Zählen O(n)
Bottom up radix sort O(n) .. *Wertebereich sollte nicht zu groß werden*