Lehrstuhl für Informatik VI
der RWTH Aachen
Prof. Dr.-Ing. H. Ney

Aachen, den 11. Mai 1998

J. Dahmen, S. Nießen

Datenstrukturen und Algorithmen – Informatik II

Probeklausur

Zur Erinnerung: ***Master-Theorem***

Sei $f(n)$ eine Funktion, $a \geq 1$ und $b > 1$ reellwertige Konstanten und $T(n)$ durch die folgende Rekursionsgleichung auf nicht-negativen ganzen Zahlen definiert:

$$T(n) = a\,T(n/b) + f(n)\,,$$

wobei n/b entweder als $\lfloor n/b \rfloor$ oder als $\lceil n/b \rceil$ interpretiert wird. Dann läßt sich $T(n)$ wie folgt in asymptotischer Schreibweise ausdrücken:

a) Falls $f(n) = O(n^{-\epsilon+\log_b a})$ für eine Konstante $\epsilon > 0$, dann ist $T(n) = \Theta(n^{\log_b a})$.

b) Falls $f(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$. (Bemerkung: $\lg n := \log_2 n$.)

c) Falls $f(n) = \Omega(n^{\epsilon+\log_b a})$ für eine Konstante $\epsilon > 0$, und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und genügend große n, dann ist $T(n) = \Theta(f(n))$.

Aufgabe 1: Rekursionsgleichungen **2+3+2 Punkte**

1. Schätzen Sie die Lösungen folgender Rekursionsgleichungen möglichst genau. Formulieren Sie ihre Antwort mit Hilfe der Θ-Notation und begründen Sie sie.

   a) $T(n) = 16\,T\left(\frac{n}{4}\right) + n^2\,, \qquad T(1) = 1$

   b) $T(n) = T\left(\sqrt{n}\right) + 1\,, \qquad T(2) = 1$

2. Beweisen oder widerlegen Sie: $\Theta\left(f(n)\right) = O\left(f(n)\right) \cap \Omega\left(f(n)\right)$, wobei $f(n) \geq 0\ \forall n$.

Aufgabe 2: Algorithmenentwurf: Sortieren **3+1 Punkte**

Es soll ein Array der Länge n aufsteigend „*in situ*" sortiert werden. Die Array-Elemente können nur zwei verschiedene Werte annehmen. Diese Werte selbst sind jedoch nicht bekannt. Geben Sie verbal oder in einer Modula3-ähnlichen Programmiersprache einen Sortieralgorithmus für diesen Spezialfall an, der eine Laufzeit von $O(n)$ hat. Begründen Sie, daß Ihr Algorithmus die angegebene Maximallaufzeit hat. Das Sortierverfahren braucht nicht stabil zu sein.

Aufgabe 3: Sortieren einer Buchstabenfolge **2+4 Punkte**

Sortieren Sie die Buchstabenfolge

$$W\ O\ R\ L\ D\ C\ U\ P$$

und geben Sie die Zwischenzustände nach jeder Bewegung eines Buchstabens an. Verwenden Sie die folgenden Sortierverfahren:

1. SelectionSort
2. HeapSort.

Aufgabe 4: Suchbäume 4 Punkte

Zeichnen Sie die Zustände eines anfangs leeren AVL-Baums, in den die Zeichenkette

$$W\ O\ R\ L\ D\ C\ U\ P\ 9\ 8$$

eingetragen wird. Gehen Sie davon aus, daß Zahlen lexikographisch größer als Buchstaben sind.

Aufgabe 5: Algorithmenanalyse 4+4 Punkte

Gegeben sei der folgende Sortieralgorithmus:

```
StoogeSort(A,i,j)

1 IF A[i] > A[j] THEN vertausche A[i],A[j]
2 IF i+1 >= j THEN RETURN
3 k := ABS((j - i + 1) / 3)
4 StoogeSort(A,i,j-k) (* Erste zwei Drittel *)
5 StoogeSort(A,i+k,j) (* Zweite zwei Drittel *)
6 StoogeSort(A,i,j-k) (* Nochmal die ersten zwei Drittel *)
```

a) Begründen Sie, daß *StoogeSort* $(A, 1, length(A))$ den Array $A[1..N]$ korrekt sortiert, wobei $length(A) = N$ ist.

b) Geben Sie eine Rekursionsgleichung für die Worst-Case Laufzeit von StoogeSort und eine möglichst genaue Abschätzung für diese Laufzeit in Θ-Notation an.

Aufgabe 6: Unbekannter Algorithmus 1+3+2 Punkte

Ein Programm enthalte die folgenden Vereinbarungen:

```
CONST  n = 500;
TYPE   feld = ARRAY[1..n] OF INTEGER;
VAR    a: feld.
```

und die folgende Funktion gtest:

```
PROCEDURE gtest (li, re: INTEGER) : INTEGER =
VAR m: INTEGER;
BEGIN (*gtest*)
  IF (li > re) THEN
    return 0;
  ELSE
    m := (li + re) DIV 2;
    return gtest(li, m-1) + g(a[m]) + gtest(m+1, re);
  END (*IF*)
END gtest
```

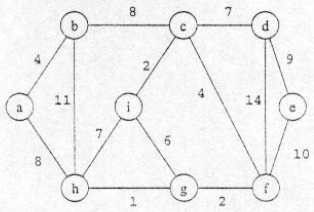
Die Prozedur g implementiert folgende Funktion:

$$g(x) = \begin{cases} 0 & \text{, falls } x \leq 5 \\ 1 & \text{, sonst} \end{cases}$$

1. Beschreiben Sie, welches Resultat die Funktion beim Aufruf `gtest(1, n)` für ein gegebenes Feld `a` liefert.
2. Ermitteln Sie größenordnungsmäßig die Anzahl der Additionen bei der Ausführung eines Aufrufs von `gtest(li, re)` im *worst case* in Abhängigkeit von `|re − li|` mit Hilfe der Rekursionsformel.
3. Geben Sie in einer Modula3-ähnlichen Programmiersprache ein *iteratives* Verfahren zur Ermittlung des Funktionswertes `gtest` an; verwenden Sie denselben Prozedurkopf.

Aufgabe 7: Prim-Algorithmus **5 Punkte**

Gegeben sei der folgende ungerichtete Graph: Konstruieren Sie mit Hilfe des in der Vorlesung vorgestellten Prim–Algorithmus den minimalen Spannbaum. Der Startknoten sei dabei der Knoten 'a'.



Aufgabe 8: Graph-Durchläufe **1+4 Punkte**

a) Erläutern Sie kurz das Prinzip der *Breadth-First-Suche* in Graphen.

b) Geben Sie in einer Modula3-ähnlichen Programmiersprache ein Verfahren für einen Breadth-First-Graphdurchlauf an. Setzen Sie dabei die zum Graphen gehörige Adjazenzliste als gegeben voraus.

Aufgabe 9: Binäre Suchbäume **5 Punkte**

In einem AVL-Baum seien als Schlüsselwerte ganze Zahlen gespeichert. Es sollen Anfragen der folgenden Form unterstützt werden: „Ermitteln Sie für ein beliebiges ganzzahliges Suchintervall die Summe der darin enthaltenen Schlüsselwerte."

Welche zusätzliche Information muß in jedem Knoten verwaltet werden, damit Anfragen dieser Art in $O(\lg\ n)$ Zeit beantwortet werden können? Wie sieht dann der Anfrage-Algorithmus aus?

Aufgabe 1: Rekursionsgleichungen

1.

a) $T(n) = 16T(\frac{n}{4}) + n^2, T(1) = 1$

Nach Mastertheorem: $a = 16, b = 4, f(n) = n^2$
Es gilt: $n^{\log_b a} = n^{\log_4 16} = n^2$
Fall 2 des MT gilt: $\Rightarrow f(n) = \Theta(n^{\log_b a})$
$\Rightarrow T(n) = \Theta(n^2 * \lg n)$

b) $T(n) = T(\sqrt{n}) + 1, T(2) = 1$

Iteration:

$$\begin{aligned} T(n) &= T(n^{1/2}) + 1 \\ &= T(n^{1/4}) + 1 + 1 \\ &= T(n^{1/8}) + 1 + 1 + 1 \\ &= \ldots \\ &= T(2) + i * 1 \end{aligned}$$

Wie berechnet sich i ?
Es ist:

$$\begin{aligned} & n^{\left(\frac{1}{2}\right)^i} = 2 \\ \Leftrightarrow \quad & \left(\frac{1}{2}\right)^i * \lg n = \lg 2 = 1 \\ \Leftrightarrow \quad & \lg n = 2^i \\ \Leftrightarrow \quad & i = \lg \lg n \end{aligned}$$

Damit:

$$\begin{aligned} T(n) &= \Theta(\lg \lg n) + T(2) \\ &= \Theta(\lg \lg n) + \Theta(1) = \Theta(\lg \lg n) \end{aligned}$$

2.

a) $g(n) \in \Theta(f(n)) \Rightarrow g(n) \in O(f(n)) \wedge g(n) \in \Omega(f(n))$

1. $g(n) \in \Theta(f(n)) \Rightarrow \exists c > 0, \exists n_0, \forall n > n_0 : 0 \leq g(n) \wedge g(n) \leq c * f(n)$
   $\Rightarrow g(n) \in O(f(n))$
2. $g(n) \in \Theta(f(n)) \Rightarrow \exists c > 0, \exists n_0, \forall n > n_0 : g(n) \geq \frac{1}{c} * f(n) \geq 0$
   $\Rightarrow g(n) \in \Omega(f(n)), da \frac{1}{c} > 0$

b) $g(n) \in O(f(n)) \cap \Omega(g(n)) \Rightarrow g(n) \in \Theta(f(n))$

$\Rightarrow$ 1. $\exists c_1 > 0, \exists n_1, \forall n \geq n_1 : 0 \leq g(n) \leq c_1 * f(n)$
2. $\exists c_2 > 0, \exists n_2, \forall n \geq n_2 : 0 \leq c_2 * f(n) \leq g(n)$

Wähle: $c = max\left(c_1, \frac{1}{c_2}\right)$
Dann folgt:

$$\forall n \geq max(n_1, n_2) : 0 \leq \frac{1}{c} * f(n) \leq g(n) \leq c * f(n)$$
$$\Rightarrow g(n) \in \Theta(f(n))$$

Aufgabe 2: Algorithmenentwurf: Sortieren

Der Algorithmus sortiert ein Array der Länge n, in dem die Array-Elemente nur zwei verschiedene Werte annehmen können, aufsteigend „in situ", wobei die Werte vorher nicht bekannt sein müssen.
Alle Werte die links von der Variablen *links* stehen, sind kleiner als das Array-Element „links" und alle Werte die rechts von der Variablen *rechts* stehen, größer als dieses Element. Mit der Variablen *zaehl* werden alle Elemente des Array **einmal** durchlaufen, mit den Elementen an den Stellen *links* und *rechts* verglichen und eventuell richtig einsortiert.
Der Algorithmus terminiert und liefert das korrekte Ergebnis, wenn *zaehl* größer als *rechts* ist.

```
MODULE sortieren EXPORTS Main;
```

```
IMPORT IO;
CONST n : INTEGER = 8;
VAR a                      : ARRAY[1..n] OF INTEGER;
    links, zaehl, rechts   : INTEGER;
    tmp                    : INTEGER;

BEGIN
  (* Initialisierung des Testfeldes *)
  a[1]:=5; a[2]:=5; a[3]:=5; a[4]:=4;
  a[5]:=5; a[6]:=5; a[7]:=4; a[8]:=4;
  (* Initialisierung der Zaehlelemente *)
  links:=FIRST(a);
  zaehl:=links + 1;
  rechts:=LAST(a);

  REPEAT
    IF a[links] < a[zaehl] THEN
      tmp:=a[zaehl]; a[zaehl]:=a[rechts]; a[rechts]:=tmp;
      rechts:=rechts - 1;
    ELSIF a[zaehl] < a[links] THEN
      tmp:=a[zaehl]; a[zaehl]:=a[links]; a[links]:=tmp;
      links:=links + 1;
      zaehl:=zaehl + 1;
    ELSIF a[zaehl]=a[links] THEN
      zaehl:=zaehl + 1;
    END;
  UNTIL zaehl > rechts;

  (* Ausgabe des Ergebnisses *)
  IO.Put("Sortierte Folge: \n");
  FOR i:=1 TO n DO
    IO.PutInt(a[i]); IO.Put(", ");
  END;
  IO.Put("\n");
END sortieren.

Laufzeitprotokoll:
Sortierte Folge:
4, 4, 4, 5, 5, 5, 5, 5,
```

Aufgabe 3: Sortieren einer Buchstabenfolge

1. Selection Sort

Prinzip:

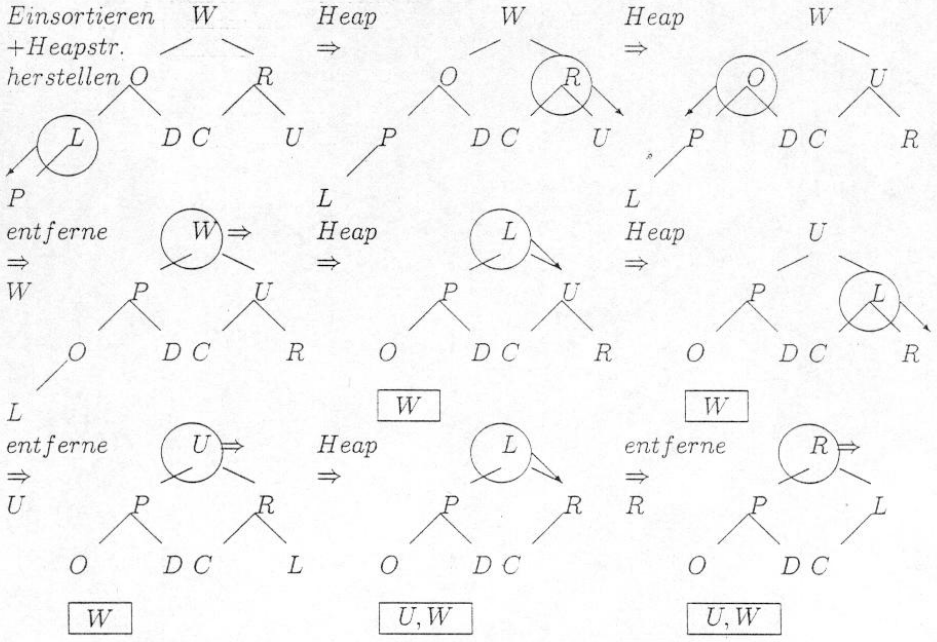
- bestimme den Datensatz mit dem kleinsten Schlüssel
- vertausche diesen Datensatz mit a[1]
- wiederhole diese Schritte für die (n-1) restlichen Datensätze

W O R L D **C** U P
C|O R L **D** W U P
C D|R **L** O W U P
C D L|R **O** W U P
C D L O|R W U **P**
C D L O P|W U **R**
C D L O P R|U W
C D L O P R U|W

2. Heap Sort

Prinzip:

- Einsortieren der Folge in den Heap
- Herstellen der Heapstruktur
- Entfernen des größten Elementes
- Einfügen des letzten Elementes an die erste Stelle
- Herstellen der Heapstruktur usw.



Zu diesem Zeitpunkt sind bereits U und W sortiert und R wird als nächstes herausgenommen. In der folgenden Darstellung sind rechts vom | die bereits sortierten Elemente dargestellt. Links davon steht der Heap, wobei die Söhne vom Element i an den Stellen $2 * i$ und $2 * i + 1$ stehen:

```
R P L O D C|U W    Entfernen vom R
C P L O D|R U W    Heapstruktur herstellen
P O L C D|R U W    Entfernen vom P
D O L C|P R U W    Heapstruktur herstellen
O D L C|P R U W    Entfernen vom O
C D L|O P R U W    Heapstruktur herstellen
L D C|O P R U W    Entfernen vom L
C D|L O P R U W    Heapstruktur herstellen
D C|L O P R U W    Entfernen vom D
C|D L O P R U W    Entfernen vom C
|C D L O P R U W   und fertig...
```

Aufgabe 4: Suchbäume

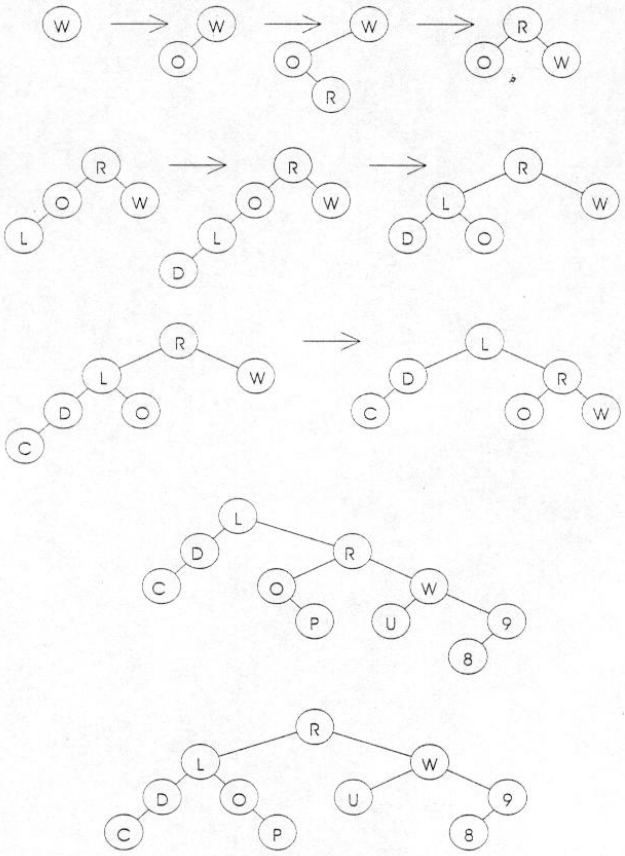


Abbildung 1: Die Bäume

Aufgabe 5: Algorithmenanalyse

Teil a

Auf dem ersten Blick sieht man, daß `StoogeSort(A,i,i+1)`, also eine beliebig angeordnetes Feld der Länge 2, korrekt sortiert. Der Rest baut induktiv darauf auf. `StoogeSort(A,1,N)` sortiert auch richtig, da mit dem ersten rekursiven Aufruf die ersten zwei Drittel der Liste in die richtige Reihenfolge gebracht werden. Somit ist kein Element des ersten Drittels größer als eines des zweiten Drittels. Sortiert man nun die letzten zwei Drittel, so wird im letzten Drittel kein Element kleiner sein können, als eins in den ersten zwei Dritteln. Warum? Sei $A_1..A_{i-1}$ das erste, $A_i..A_{j-1}$ das zweite, und $A_j..A_N$ das letzte Drittel. Sei A_k aus dem letzten Drittel kleiner als A_i, so wird beim Sortieren das gesamte mittlere Drittel dahinter gesetzt. Dieses mittlere Drittel ist aber auch immer größer als das erste. Demnach wird im letzten Drittel also keine Zahl kleiner sein können , als im ersten! Nun brauchen also nur noch die ersten zwei Drittel erneut soriert zu werden.

Teil b

Kosten für die einzelnen Zeilen von `StoogeSort`:

Zeile	Befehl	Kosten
1	`IF A[i] > A[j] THEN swap(i,j)`	$\Theta(1)$
2	`IF i+1 > j THEN RETURN`	$\Theta(1)$
3	`k := ABS((j-i+1)/3)`	$\Theta(1)$
4	`StoogeSort(A,i,j-k)`	$T(2n/3)$
5	`StoogeSort(A,i+k,j)`	$T(2n/3)$
6	`StoogeSort(A,i,j-k)`	$T(2n/3)$

Rekursionsgleichung:

$$T(n) = 3T(2n/3) + \Theta(1)$$

Diese ergibt sich aus der Addition der Kosten der einzelnen Zeilen.

Mit Hilfe des ersten Falles des Master-Theorems kann man nun mit $a = 3$, $b = 1,5$ und $f(n) = 1 = n^0 = O(n^{-\epsilon+\log_{1,5} 3})$, wobei $\epsilon = 2,7 > 0$ gewählt werden kann, eine genaue Laufzeitabschätzung angeben:

$$T(n) = \Theta(n^{2,7})$$

Aufgabe 6: Unbekannter Algorithmus

Teil 1

Die rekursive Funktion `gtest` prüft zunächst, ob die angegebenen linken und rechten Begrenzungen `li` und `re` sich nicht überschneiden. Wenn der linke Rand noch links von rechten steht, so suche den mittleren Wert beider - `m` - und addiere `g(a[m])` zu den Ergebnissen der rekursiven Aufrufe der beiden Teilfolgen `(li, m-1)` und `(m+1, re)`.

Durch `m-1` und `m+1` wird dafür gesorgt, daß jeder Wert `g(a[m])` maximal ein mal in die Summe aufgenommen wird. Andererseits sorgt die Halbierung dafür, daß jeder Wert mindestens einmal zur Summe hinzugenommen wird, woraus folgt, daß jeder Wert in den im ersten Aufruf angegebenen Grenzen `gtest(1, n)` genau einmal zur resultierenden Summe hinzugenommen wird. Die Funktion berechnet also nichts anderes als:

$$\texttt{gtest(1, n)} = \sum_{i=1}^{n} g(a[i]) = \sum_{\forall a[i]>5, 1\leq i\leq n} 1.$$

Umgangssprachlich bedeutet dies: die Anzahl der Werte des Feldes `a`, die größer als 5 sind.

Teil 2

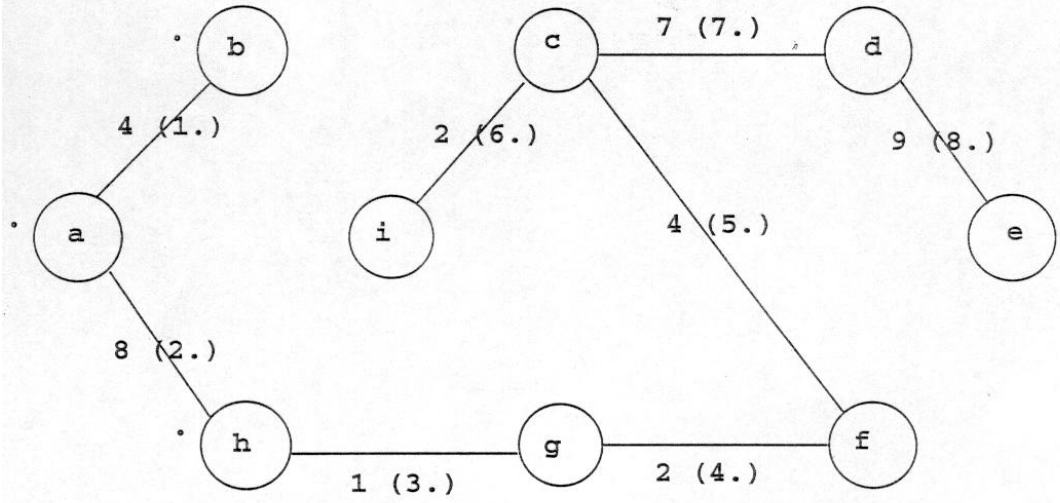
In jedem Aufruf der Funktion werden, solange die Abbruchbedingung nicht erfüllt ist, drei Additionen - hierzu sollen nur die Addition aus der Berechnung für `m`, und die beiden Additionen aus der Berechnung zwischen den rekursiven Aufrufen zählen - durchgeführt. Wie in Teil 1 angegeben, wird jedes Element aus `a[1..n]` nur genau einmal zur Berechnung hinzugenommen. Hieraus folgt, daß genau $3n$ Additionen durchgeführt werden.
In der Θ-Notation ist dies natürlich $\Theta(n)$. In Bezug auf die Größenordnung an Additionen ist der Algorithmus also nicht unbedingt als schlecht zu bewerten.

Teil 3

Der iterative Algorithmus ist dennoch einfacher, schöner und schneller, und benötigt nur genau n Additionen:

```
PROCEDURE gtest (li, re: INTEGER) : INTEGER =
VAR sum: INTEGER;
BEGIN (*gtest*)
  sum := 0;
  FOR i:=li TO re DO
    sum:=sum + g(a[i]);
  END;
  return sum;
END;
```

Aufgabe 7 : Prim-Algorithmus
Der Primalgorithmus geht bei ungerichteten, zusammenhängenden Graphen dieselben durch, indem er von einem vorgegebenen Anfangsknoten aus immer den Weg auswählt der am kürzesten zu einem bisher noch nicht besuchten Knoten führt.



17

Aufgabe 9: Binäre Suchbäume
Das gesuchte Intervall ist beliebig, kann also auch alle Elemente enthalten.
Diese aufzusummieren benötigt eine Zeit von Omega(n).
Um den Zeitaufwand logarithmisch zu halten muß man also eine Information über Teilsummen in den Knoten
gespeichert werden. Der Einfachheit halber bietet es sich an z.B. die Summe der Schlüssel zu speichern die
kleiner oder gleich dem Aktuellen sind.
Der Algorithmus sieht dann wie folgt aus:

1. Suche nach unterer Intervallgrenze [O(lg n)]
   Zwei Fälle: a) Die Suche endet auf dem Schlüsselwert oder darüber.
   Setze low := Summe - Schlüssel
   b) Suche endet unter der unteren Intervallgrenze:
   Setze low := Summe
2. Suche nach oberer Intervallgrenze [O(lg n)]
   Zwei Fälle: a) Suche endet über auf dem Element:
   Setze high := Summe - Schlüssel
   b) Suche endet auf oder unter der oberen Intervallgrenze:
   Setze high := Summe
3. Die gesuchte Summe erhält man als (high -low).

Aufgabe 8: Graph-Durchläufe
a) Sei G = (V,E) ein Graph und s aus V ein Knoten. Dann geht Breadth-First-Suche sukzessive die Kanten
von G durch und liefert so die von s aus erreichbaren Knoten. Dabei werden die näheren Kanten vom
Anfang aus immer zuerst besucht, mit anderen Worten, wird der Graph wie ein Baum mit
dem Ausgangsknoten als Wurzel jeweils niveaugleich durchlaufen.

b)

```
const maxV = 1000;
type link = ^node;
       node = record
                      v: integer;
                      next : link;
                end;
var adj : array[1..maxV] of link;

procedure bfs;
var id, k : integer;
     val : array[1..maxV] of integer;

     procedure visit (k : integer);
     var t : link;
     begin
       put(k);
       repeat
              k := get;  inc(id);  val[k] := id;
              t := adj[k];
              wile t <> t do begin
                     if val[t^.v] = 0 then begin
                            put(t^.v);
                            val[t^.v] := -1;
                     end
              until queueempty
       end;

begin
       id := 0 ;
       queueinitialize;
       for k := 1 to V do val[k] := 0;
       for k := 1 to V do
```

```
        if val[k] = 0 then visit(k)
end;
```