Aufgabe 1

(a)

In *f1* wird der Wert von $f2(n)$ genau n^2 mal aufaddiert; das Ergebnis von *f1* ist daher $n^2 * f2(n)$. Es bleibt die Analyse von *f2*. Dort kann die folgende Rekursionsgleichung aufgestellt werden:

$$f2(1) = 1$$
$$f2(n) = n * f2(n / 2)$$

Die Auflösung dieser Gleichung ergibt:

$$f2(2^a) = \prod_{i=1}^{a} 2^i = 2^{\sum_{i=1}^{a} i} = 2^{\frac{a(a+1)}{2}}$$

Beweis:

Anfang: $a = 0$, korrekt

Schritt: $a \rightarrow a + 1$

$$f2(2^{a+1}) = 2^{\frac{(a+1)(a+2)}{2}}$$

$$= 2^{\frac{a^2+3a+2}{2}} = 2^{\frac{a(a+1)}{2} + \frac{2a+2}{2}}$$

$$= 2^{\frac{a(a+1)}{2}} \cdot 2^{a+1}$$

$$= 2^{a+1} f2(2^a)$$ □

Diese Gleichung läßt sich umformen zu:

$$f2(n) = 2^{\frac{\log(n)(\log(n)+1)}{2}}$$

Somit ist das Gesamtergebnis:

$$f1(n) = n^2 2^{\frac{\log(n)(\log(n)+1)}{2}}$$

(b)

Für die Laufzeitanalyse kann man beobachten, daß für jeden Knoten im Aufrufbaum von *f2* nur konstante Kosten anfallen. Die Kosten für die Addition und die **for**-Schleife lassen sich in die entsprechenden Söhne im Aufrufbaum verschieben. Da jeder innere Knoten des Baumes mindestens zwei Nachfolger hat, enthält dieser mehr Blätter als innere Knoten, so daß die Laufzeit

nur von der Anzahl der Blätter des Aufrufbaums abhängt. Diese Anzahl entspricht jedoch dem Rückgabewert dieser Funktion. So daß die Laufzeit durch $O(n^2 2^{(\log n)^2})$ abgeschätzt werden kann.

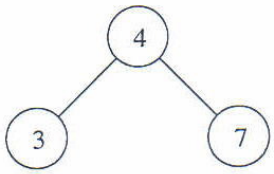
Aufgabe 2

(a)

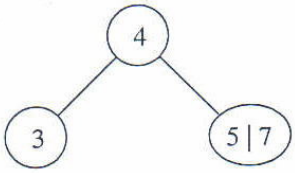
Ein 2-3 Baum ist ein B-Baum der Ordnung 1. Die inneren Knoten können also 2 oder 3 Söhne haben.

Einfügen von 3, 4, 7:

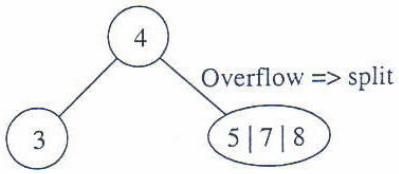
Overflow => split

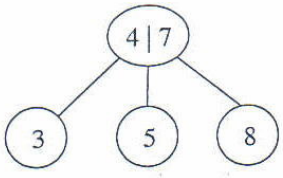


Einfügen von 5:

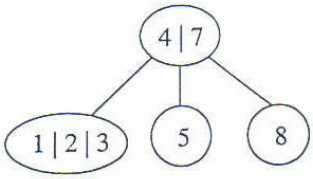


Einfügen von 8:



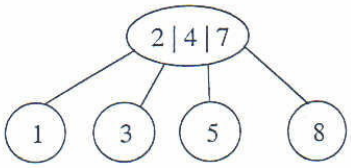


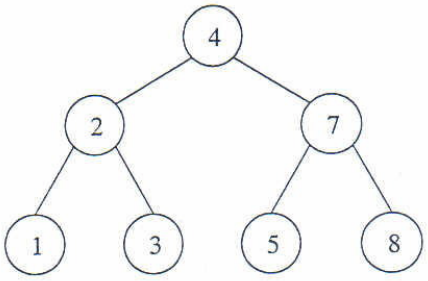
Einfügen von 1, 2:



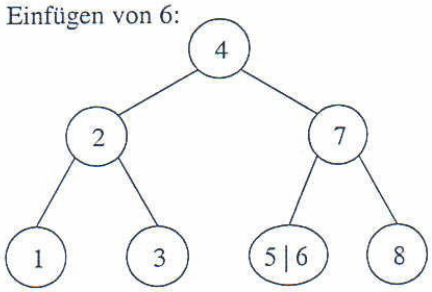
Overflow => split

Overflow => split



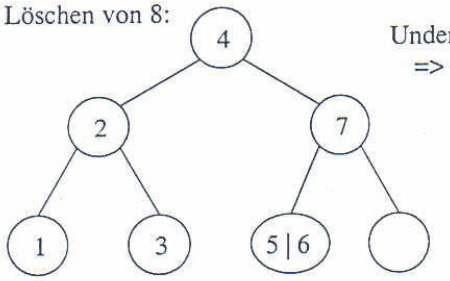


Einfügen von 6:

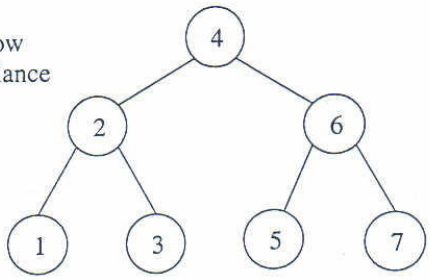


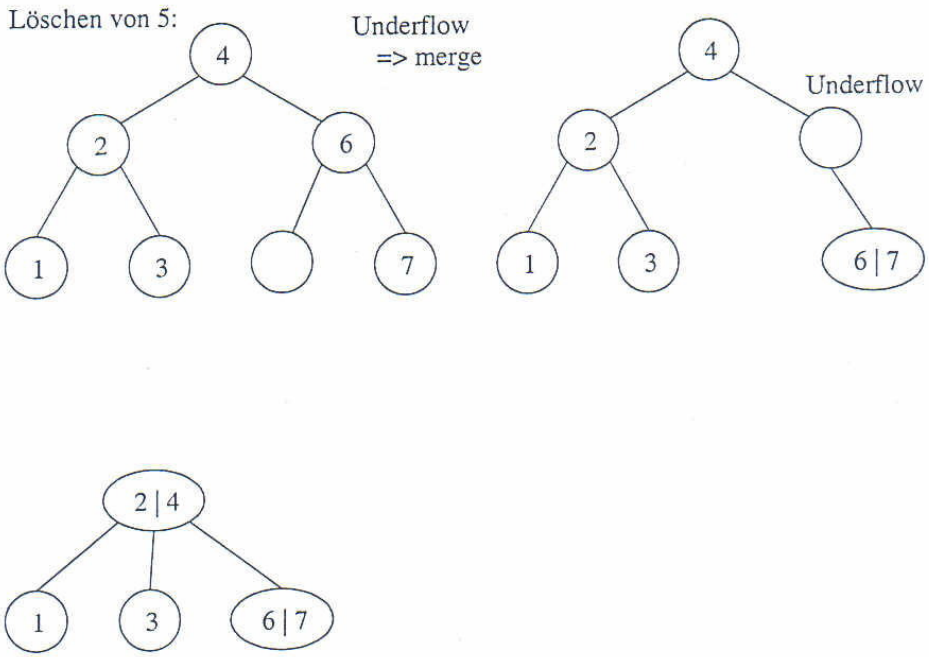
(b)

Löschen von 8:



Underflow
=> balance





Aufgabe 3

(a)

Über Hashtabellen lassen sich leicht die Operationen *insert*, *delete* und *member* des Dictonary Datentyps implementieren.

Vorteile von Hashverfahren sind die leichte Implementierbarkeit und eine *average case* Laufzeit von O(1) für die oben genannten Operationen. Nachteile sind ein *worst case* Verhalten von $O(n)$ und die unsortierte Organisation der Schlüsselwerte.

Falls die primäre Hashfunktion, die den Wertebereich eines Schlüssels in eine Behälteradresse transformiert für verschiedene Schlüssel denselben Wert berechnet und der Behälter bereits voll ist, tritt ein Konflikt auf. Dieser kann nach folgenden Methoden aufgelöst werden:

Lineares Sondieren: Man durchläuft zyklisch alle weiteren Behälter,
$h_i(x) = (h(x) + i) \bmod m$.

Quadratisches Sondieren: Die i-te Hashfunktion addiert i^2 zur zuvor ermittelten Behälternummer und nimmt den Rest nach Division mit m als neue Behälternummer; also $h_i(x) = (h(x) + i^2) \bmod m$. Dadurch wird $h_i(x)$ „besser“ über den Wertebereich 0 ... m-1 verstreut.

Doppelhashing: Wenn zwei Funktionen h und h' gegeben sind, so daß für beide Funktionen eine Kollision zweier Schlüsselwerte x und y mit der Wahrscheinlichkeit $1/m^2$

auftritt, so hat das Eintreten des Ereignisses $h(x) = h(y)$ keinen Einfluß auf das Ereignis $h'(x) = h'(y)$, daher bezeichnet man solche Funktionspaare als voneinander unabhängige Hashfunktionen. Um im i-ten Versuch einen Behälter zu ermitteln, benutzt man die Funktion $h_i(x) = (h(x) + h'(x) \cdot i^2) \bmod m$

(b)

Falls die nächste Primzahl P der Form $4 \cdot j + 3$ mit $P > n$ wesentlich größer als n ist, wird unter Umständen viel Speicherplatz verschwendet, nämlich $P - n$. Daher sollte man also überprüfen, ob nicht eine kleinere Primzahl p mit $p < n < P$ existiert, so daß für geeignetes $b \geq n/p$ ggf. weniger Speicherplatz benötigt wird. Somit ist der ungenutzte Speicherplatz über die Formel $n - b \cdot p$ berechenbar. Untersucht man nun für absteigendes p die Speicherplatzausnutzung, so läßt sich der Platzbedarf optimieren.

(c)

Es bezeichne $Alpha(x_i)$ die Position des i-ten Zeichens von x im Alphabet. Die benötigten Hashfunktionen lauten:

$h_0(x) = (Alpha(x_1) + Alpha(x_2) + Alpha(x_3)) \bmod 7$

$h_1(x) = (h_0(x) + 1) \bmod 7$

$h_2(x) = (49 + h_0(x) - 1) \bmod 7$

$h_3(x) = (h_0(x) + 4) \bmod 7$

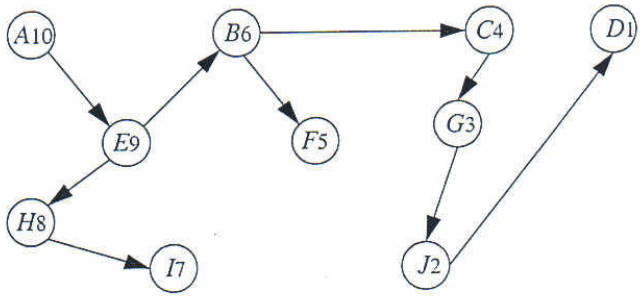
Die Anwendung dieser Funktionen ergibt folgende Aufteilung der Elemente auf die Behälter B identifiziert als Nummer zwischen 0 und 6. Die Spalte *Pos* gibt die Position innerhalb eines Behälters an.

x	$h_0(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$	B	*Pos*
ABC	6				6	1
ABB	5				5	1
BAC	6				6	2
BAB	5				5	2
CAB	6	0			0	1
BBA	5	6	4		4	1
ACB	6	0			0	2
CBA	6	0	5	3	3	1
ABA	4				4	2
AAC	5	6	4	2	2	1
CAA	5	6	4	2	2	2

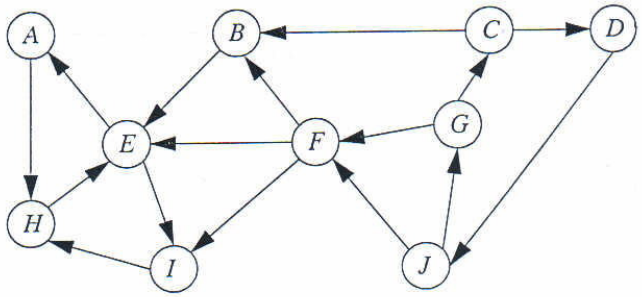
x	$h_0(x)$	$h_1(x)$	$h_2(x)$	$h_3(x)$	B	Pos
AAB	4	5	3		3	2
BAA	4	5	3	1	1	1
ABA	4	5	3	1	1	2

Aufgabe 4

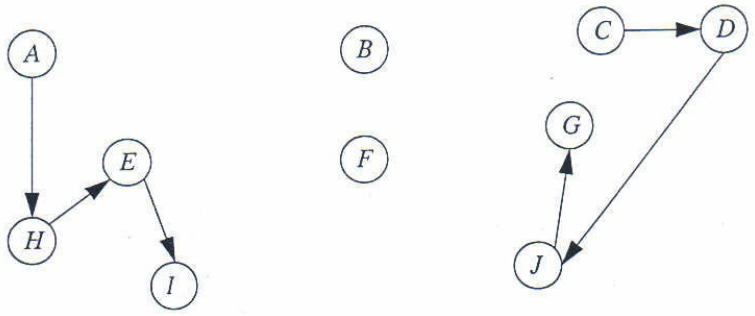
Im ersten Schritt werden die numerierten Depth-First-Spannbäume des gegebenen Graphen berechnet:



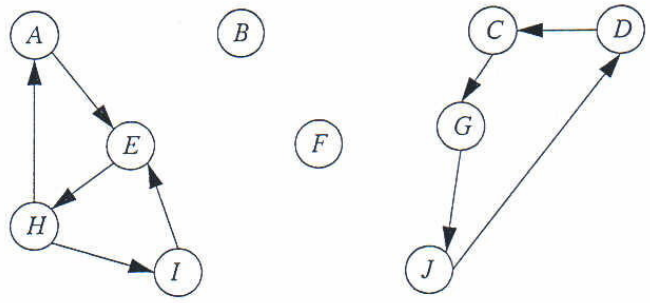
Im zweiten Schritt wird der inverse Graph gebildet:



Im dritten Schritt werden die *dfs*-Spannbäume des inversen Graphen erzeugt:

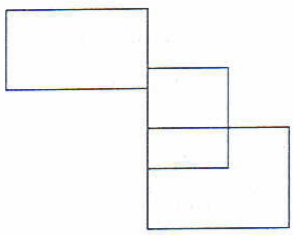


Als letztes wird das Ergebnis ausgegeben:



Aufgabe 5

Die Sweepline-Algorithmus läuft grundsätzlich wie folgt ab: In der Sweep-Event-Struktur sind die linken und rechten Enden der Rechtecke sortiert gespeichert. Bei der Ausführung des Algorithmus wird an jedem durch ein Rechteckende festgelegten Haltepunkt eine Aktion durchgeführt: Falls es sich um ein linkes Ende handelt, wird es anhand seines y-Intervalls in die Status-Struktur eingefügt und es werden ggf. Überlappungen mit den anderen dort enthaltenen Intervallen gefunden. Die Schnitte der dazugehörigen Rechtecke werden daraufhin ausgegeben. Wird ein rechtes Ende gefunden, so wird das entsprechende y-Intervall aus der Statusstruktur entfernt.



Bei der Sortierung der Rechteckenden in der Sweep-Event-Struktur ist zunächst zu überlegen, wann sich Rechtecke überhaupt schneiden können. Hier ist zu beobachten, daß für es für einen Schnitt notwendig ist, daß sich sowohl die x-Intervalle als auch die y-Intervalle der Rechtecke überlappen, d.h. der gemeinsame Bereich zweier Intervalle ist nicht leer. Die Überlappung der x-Intervalle wird durch die Sortierung nach x-Koordinaten der Rechteckenden sichergestellt; die Überlappung der y-Intervalle ihrerseits durch die Sortierung der Rechteckenden nach linken und rechten Enden der y-Intervalle. Zusätzlich muß noch beachtet werden, daß bei zwei Enden mit gleichen x-Koordinaten die linken Enden vor den rechten Enden einsortiert werden, da sonst horizontal aneinanderstoßende Rechtecke nicht gefunden werden (siehe Bild)

Beispiel: Nehmen wir an, die x-Intervalle für zwei Rechtecke t, u seien mit t_l, t_r und u_l, u_r bezeichnet. Wenn diese Intervalle in der *SES* wie folgt geordnet sind ... t_l ... t_r ... u_l ... u_r ..., dann würde der Algorithmus keine überlappenden Intervalle (und damit Rechtecke) finden. Daher muß sichergestellt sein, daß u_l vor t_r einsortiert wird.

Seien die Rechtecke in getrennter Darstellung angegeben, so daß ein Rechteckende wie folgt aussieht:

$(x, y_l, y_r, mark, z)$, mit $mark \in$ {„links“, „rechts“} und z ein Zeiger auf das Rechteck

Dann definieren wir nun eine Ordnung auf solchen Rechteckenden, nach der sortiert werden soll. Die Funktion *compare* liefert für zwei Rechteckenden re_1, re_2 den Wert 0, falls beide genau gleich sind, 1, falls re_1 kleiner ist als re_2 und 2 sonst.

function *compare* (re_1, re_2) : *integer*
  (* Vergleich bzgl. x *)
  if $re_1.x < re_2.x$ **then return** 1 **fi**;
  if $re_1.x > re_2.x$ **then return** 2 **fi**;
  (* Vergleich bzgl. *mark* *)
  if $re_1.mark$ = „links“ und $re_2.mark$ = „rechts“ **then return** 1 **fi**;
  if $re_1.mark$ = „rechts“ und $re_2.mark$ = „links“ **then return** 2 **fi**;
  (* Vergleich bzgl. y-Intervall *)
  if $re_1.y_l < re_2.y_l$ **then return** 1 **fi**;
  if $re_1.y_l > re_2.y_l$ **then return** 2 **fi**;
  if $re_1.y_r < re_2.y_r$ **then return** 1 **fi**;
  if $re_1.y_r > re_2.y_r$ **then return** 2 **fi**;
  return 0;
end compare.

In der Sweep-Status-Struktur werden Intervalle gespeichert. Die Struktur unterstützt das Einfügen und Löschen solcher Intervalle. Zusätzlich muß das Suchen mit einem Intervall nach gespeicherten, überlappenden Intervallen unterstützt werden.

Der Algorithmus kann dann folgendermaßen skizziert werden:

algorithmus *Rechteckschnitt* (Rechteckmenge R)

  füge alle Rechtecke $r \in R$ in getrennter Darstellung in die Event-Struktur *SES* ein, d.h. für
    die linken Enden (x, y_l, y_r, „links“, ^r)
    die rechten Enden (x, y_l, y_r, „rechts“, ^r)

  sortiere *SES* aufsteigend nach x, benutze dabei die durch *compare* definierte Ordung;

  (* Start des Sweeps *)
  für jedes Rechteckende $e \in SES$ führe eine der folgenden Aktionen aus:
    (a) falls e ein linkes Rechteckende ist, füge es anhand seines y-Intervalls in die Status-Struktur ein, bestimme Überlappungen mit den dort bereits enthaltenen Intervallen gib dann ggf. Paare von Schnitten aus.
    (b) falls e ein rechtes Ende ist, lösche das entsprechende y-Intervall aus der Status-Struktur
end *Rechteckschnitt*.

Aufgabe 6

Beim natürlichen Mischen werden zunächst Läufe variabler Länge berechnet. Die erzeugten Läufe sind rechts angegeben. Elemente, die erst bei der Berechnung eines neuen Laufes betrachtet werden, sind in Klammern angegeben, Werte, die bereits in den aktuellen Lauf übernommen wurden, sind unterstrichen.

1	2	3	4	1. Lauf
100	10	11	20	10
100	(7)	11	20	11
100	(7)	23	20	20
100	(7)	23	90	23
100	(7)	87	90	87
100	(7)	(86)	90	90
100	(7)	(86)	(43)	100
(99)				

1	2	3	4	2. Lauf
99	7	86	43	7
99	32	86	43	32
99	40	86	43	40
99	41	86	43	41
99	70	86	43	43
99	70	86	(31)	70
99	(30)	86	(31)	86
99	(30)	(17)	(31)	99
(18)				

1	2	3	4	3. Lauf
18	30	17	31	17
18	30		31	18
	30		31	30
			31	31

Nun werden die Läufe auf die Bänder verteilt.

f_1: 10 11 20 23 87 90 100 | 17 18 30 31

f_2: 7 32 40 41 43 70 86 99

Nun beginnt die eigentliche Sortierung; zwei Läufe werden jeweils per *merge* zusammengeführt und alternierend auf die Bänder geschrieben.

Schritt 1:

g_1: 7 10 11 20 23 32 40 41 43 70 86 87 90 99 100

g_2: 17 18 30 31

Die sortierte Folge steht nach Schritt 2 nun auf Band f_1:

f_1: 7 10 11 17 18 20 23 30 31 32 40 41 43 70 86 87 90 99 100