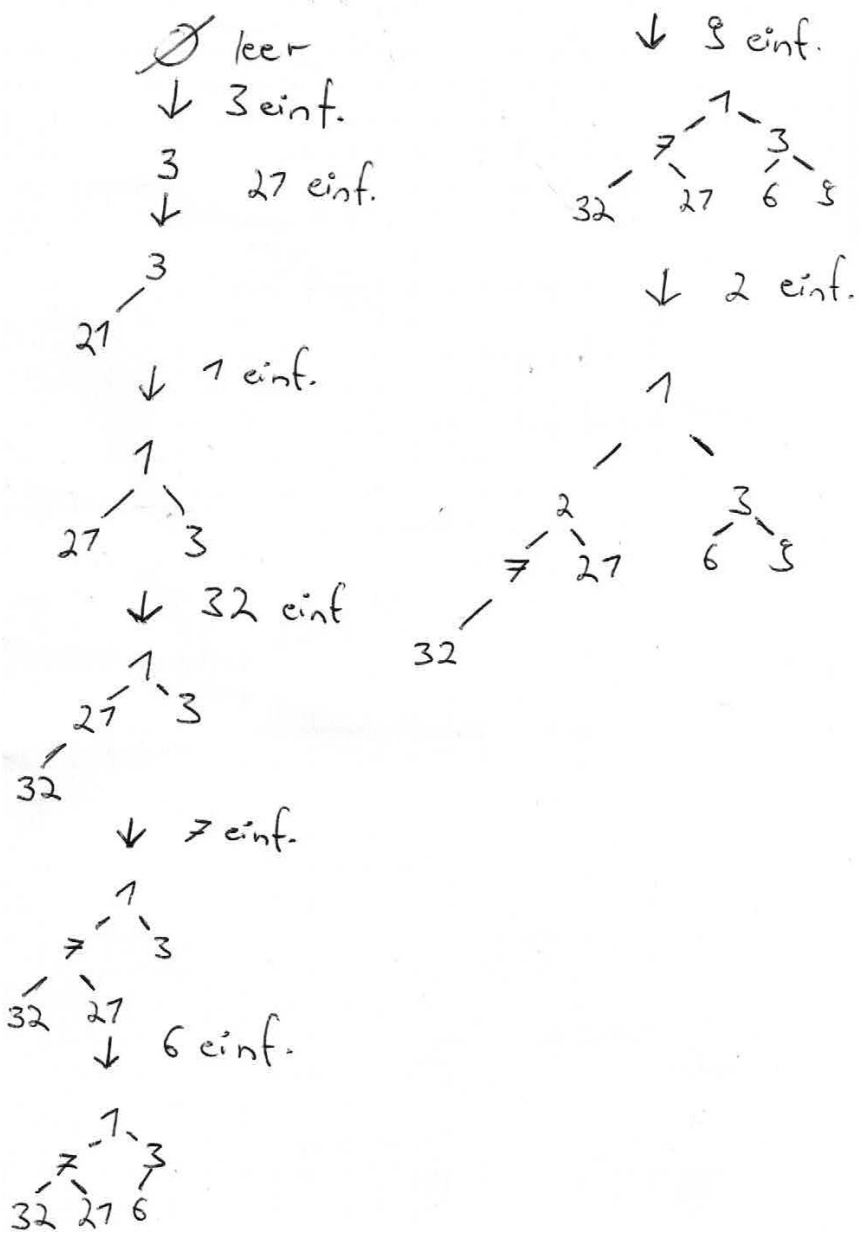
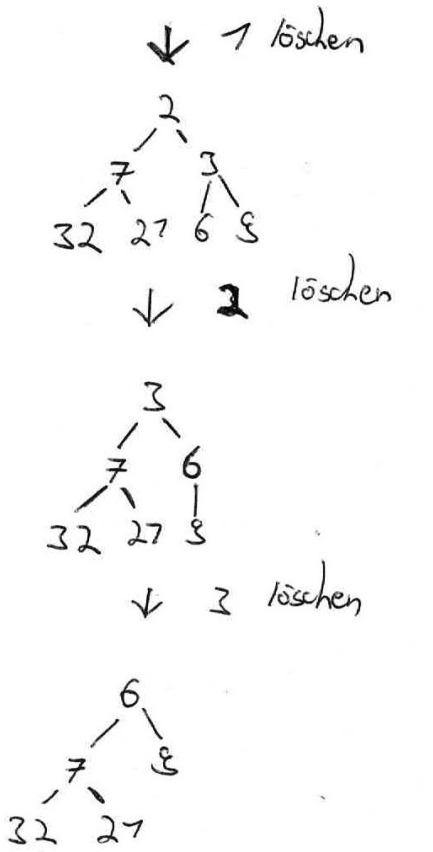
Nicolai Schönberg, 9314806, Datenstrukturen I, EA 3

Aufgabe 1

a)



b)



Aufgabe 2

Um die first-come, first-serve Funktionalität zu erhalten, würde ich eine priorty queue als heap implementieren und dabei jeden Knoten als Queue darstellen. Dadurch ist es dann möglich Elemente mit der selben Priorität hinten in der Queue einzureihen.

Mögliche Implementierung für Heap-Elemente mit Queue:

```
public interface NodeElement<T> {
	int getPriority();
	Queue<T> getElementQueue();
}
```

Es gibt also im Heap jede Priorität genau einmal mit beliebig vielen Elementen in der Queue.

```
algorithm insert(h, e, p)
{ Fügt ein Element e mit Priorität p in Heap h ein }
prio_knoten = Finde Knoten in h mit Priorität p; // Dies ist das Minimum
if prio_knoten != null {
      prio_knoten.getElementQueue().enqueue(e);
}
else {
      Erzeuge einen neuen Knoten q mit Priorität p;
      Füge q auf der ersten freien Position der untersten Ebene ein (falls
      die unterste Ebene voll besetzt ist, beginne eine neue Ebene);

      sei p der Vater von q;
      while p existiert and μ (q) < μ (p) {
            vertausche die Einträge in p und q; setze q auf p und p auf den Vater
            von p.
      }

      q.getElementQueue().enqueue(e);
}
```

Aufgrund der Ordnung des Hash liegt das Minimum in der Wurzel des Baumes. Das Element was als nächstes verarbeitet werden muss, liegt in der Queue des Wurzelknotens ganz vorne. Um es zu löschen genügt es ein dequeue auf die Queue des Wurzelknotens anzuwenden. Sollte die Queue danach leer sein, muss der Wurzelknoten gelöscht werden und der Baum umsortiert werden, damit das neue Minimum dann in der neue Wurzelknoten wird.

```
algorithm deletemin(h)
{lösche das minimale Element aus dem Heap h und gibt es als minimum aus}
wurzelKnoten = Entnimm h die Wurzel;
element = wurzelKnoten.getElementQueue.dequeue();
Ausgabe von element als minimum;

if wurzelKnoten.getElementQueue().empty {
      nimm den Eintrag der letzten besetzten Position im Baum (lösche
      diesen Knoten) und setze ihn in die Wurzel;
      sei p die Wurzel und seien q, r ihre Söhne;
      while q oder r existieren and ( μ (p) > μ (q) or μ (p) > μ (r)) {
            vertausche den Eintrag in p mit dem kleineren Eintrag der beiden
            Söhne;
            setze p auf den Knoten, mit dem vertauscht wurde, und q, r auf dessen
            Söhne;
      }
}
```

Aufgabe 4

a) A, D, B, E, H, G, I, F, C

b) Um die Aufgabe zu lösen, wird ein Algorithmus benötigt der aus den übergebenen Knoten-Reihenfolgen einen Baum konstruiert. Dieser kann dann an den Algorithmus *Postorder* übergeben werden, um die Postorder-Reihenfolge zu erhalten.

Allgemeine Erläuterung zum Algorithmus:

- Um den Baum zu konstruieren, wird zunächst das erste Element des Preorder-Durchlaufs betrachtet. Dies ist die Wurzel des zu konstruierenden Baumes
- Anschließend wird das erste Element des Preorder-Durchlaufs im Inorder-Durchlauf gesucht. Der Index i des gefunden Elementes bestimmt nun wie die Teilbäume unter der Wurzel aufgebaut sind. Alle Elemente mit Index < i liegen im linken Teilbaum der Wurzel, Alle Elemente mit Index > i liegen im rechten Teilbaum
- Diese Schritte werden nacheinander rekursiv für alle Elemente der Reihenfolge wiederholt. Dabei wird also zum Beispiel für den Preorder-Durchlauf C,D,A,F,G,H,B,E,I zuerst C, dann D, dann A usw. bis I im Inorder-Durchlauf gesucht und die darunter liegenden Teilbäume bestimmt

```
class Node
{
        char key;
        Node left, right;
}

algorithm printPostorder(char[] inorder, char[] postorder) {
        tree = buildTree(inorder, postorder, 0, 0);
        postorderList = Postorder(tree);
        Ausgabe von postorderList;
}

algorithm buildTree(char[] inorder, char[] preorder, int start, int end) : Node {
        {Rekursiver Algorithmus der aus den Arrays inorder und preorder einen Baum zusammenbaut.
         Beim ersten Aufruf von buildTree muss start = end = 0 sein.
         Gibt die Wurzel des baumes zurück.}

        if (start > end) {
                // Ungültig
                return null;
        }

        element = Nimm Element von Preorder anhand Preorder Index Variable;
        Erhöhe Preorder Index Variable um 1 für nächsten rekursiven Durchlauf;
        node = Erzeuge neuen Knoten, setze key = element;

        if (start == end) {
                // Knoten hat keine Kinder, fertig
                return node;
        }

        inIndex = Finde index von element in inorder;
```

```
        // Für die nachfolgenden rekursiven aufrufe müssen start und end korrekt berrechnet werden
        node.left = Rufe buildTree rekursiv auf für alle Elemente vor inIndex
        node.right = Rufe buildTree rekursiv auf für alle Elemente hinter inIndex und

        return node;
}
```

Aufgabe 5

Kodierung:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

h(x) = Mittel-Quadraht-Methode
$h_i(x) = (h(x) + i^2) \bmod 10$

Wort	Codierung	Ergebnis (x)	x^2	Initial Wert=h(x)
Cplusplus	3+16+12	31	961	6
Java	10+1+22	33	1089	8
Prolog	16+18+15	49	2401	0
Python	16+25+20	61	3721	2
Cobol	3+15+2	20	400	0
Fortran	6+15+18	39	1521	2
Modula	16+15+4	32	1024	2
QBasic	17+2+1	20	400	0

Nr	Inhalt	Kollision → Korrektur
0	Prolog	Cobol → $h_1(x) = 1$; Modula → $h_4(x) = 8$; QBasic → $h_1(x) = 1$
1	Cobol	QBasic → $h_2(x) = 4$
2	Python	Fortran → $h_1(x) = 3$; Modula → $h_1(x) = 3$;
3	Fortran	Modula → $h_2(x) = 6$;
4	QBasic	
5		
6	Cpluspl	Modula → $h_3(x) = 1$;
7	Modula	
8	Java	Modula → $h_5(x) = 7$;
9		