1

a

```
func merge(l1, l2) {
    list res
    while (|l1| > 0 and |l2| > 0) {
        // Größtes Element in res[n+m]
        if (l1[0] > l2[0]) {
            /*
                append geht in O(1), da die Größe von
                res bekannt und zu beginnt allokiert werden kann.
                In tatsächlichem Code muss der aktuelle Index
                von res natürlich gespeichert werden.
            /*
            append(res, l1[0])
            /*
                annahme eines Datentypen,
                bei dem remove an stelle 0 in O(1) möglich ist
                falls dies nicht möglich ist
                müssen die Listen von hinten populiert werden
                und l1[n] entfernt werden.
            */
            remove(l1, 0)
        }
        else {
            append(res, l2[0])
            remove(l2, 0)
        }
    }
    // falls l1 leer ist, wird l2 komplett an res angehängt
    if (|l1| = 0) {
        append(res, l2)
    }
    else {
        append(res, l1)
    }
    return res
}
```

b

```
/*
    hier als kurzschreibweise für eine funktion mit 2 bis k parametern.
    Dabei wird angenommen, dass k eine zweierpotenz ist
*/
func mergeK(l1, ..., lk) {
    if (k = 2) {
        // also nur 2 listen übergeben
        return merge(l1, l2) // merge aus Aufgabenteil a
    }
    else {
        /*
            Es werden jeweils k/2 listen zusammengesetzt.
            Nach der Rekursion geschieht ausgabe als eine Liste.
        */
        return merge(
            mergeK(l1, ..., l(k/2)),
            mergeK(l(k/2+1), ..., lk)
        )
    }
}
```

c

Die delete-Routine entfernt das kleinste Element im Heap. Dabei wird das Minimum der untersten Ebene des Heaps (als Baumdarstellung) ermittelt und dieses entfernt. Im Speicher sind dies die letzten $\frac{n}{2}$ Speicherzellen.

d

Für einen ternären Heap gilt v Vaterknoten mit Söhnen s_0, s_1, s_2.
$s_0 = 3 \cdot v + 1, s_1 = 3 \cdot v + 2, s_3 = 3 \cdot v + 3$

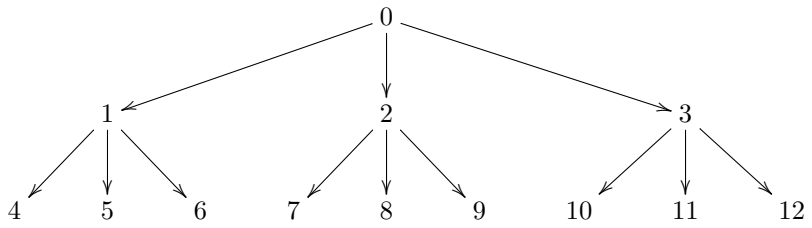

Figure 1: Darstellung ternärer Heap als Baum mit Index