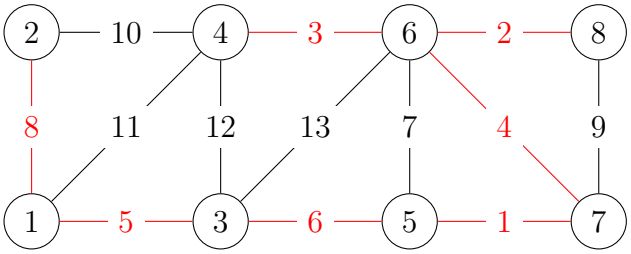
Übungsblatt 3, Algorithmen und Datenstrukturen

Aufgabe 17

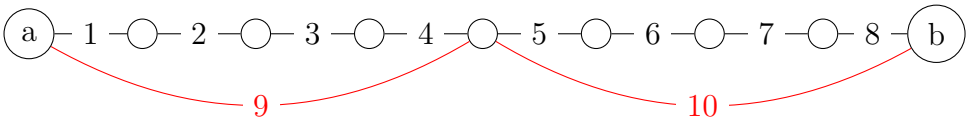


Die Kanten werden in aufsteigender Reihenfolge ihrer Gewichte betrachtet.

Aufgabe 18

1. So wie bei Kruskal.
2. Es wird immer die kleinste Kante ausgewählt die genau einen Knoten in S hat.
3. Ja, dann wäre in der letzten Iteration $(1, 2)$ statt $(2, 4)$ hinzugefügt worden.

Aufgabe 19



Der kürzeste Pfad ist rot, der Rest ist der MST.

Aufgabe 20

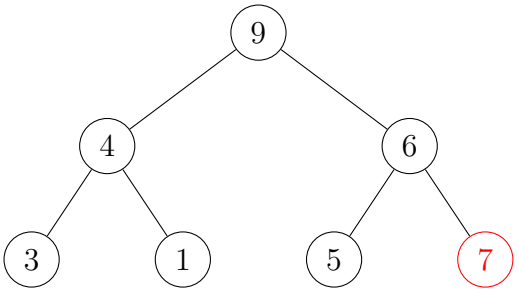
```
COLOR-FIRST(A, i, color):
  j = i
  while j < 3n:
    if Examine(j) == color:
      Swap(i, j)
      i++
    j++

RGB(A):
  COLOR-FIRST(A, 0, "red")
  COLOR-FIRST(A, n, "green")
```

Ähnlich wie Partitioning bei QuickSort sortiert *COLOR-FIRST* in linearer Zeit die Elemente einer gewissen Farbe an den Anfang. Das ganze wird zweimal ausgeführt, was wieder $O(n)$ ergibt.

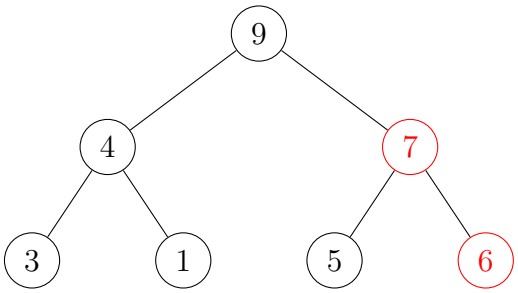
Aufgabe 21

1. Ein beliebiger Array kann in $O(n)$ heapified werden, sortieren liegt jedoch in $O(n \log n)$
2. Finden des minimalen Elements geht bei einem sortierten Array in $O(1)$, bei einem Max-Heap in $O(n)$.
3. Finden des maximalen Elements geht bei beiden in $O(1)$.
4. Löschen eines Elements geht bei einem Max-Heap in $O(\log n)$, bei einem Array liegt diese Operation in $O(n)$.

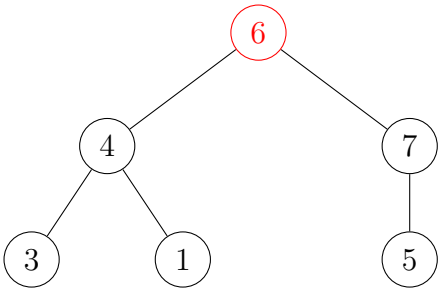
Aufgabe 22



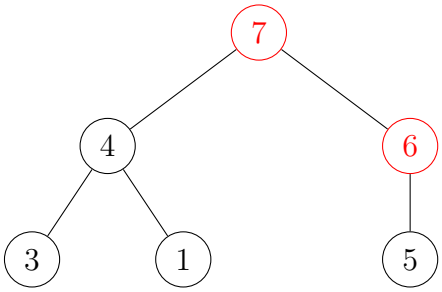
(a) 7 wird am Ende eingefügt.



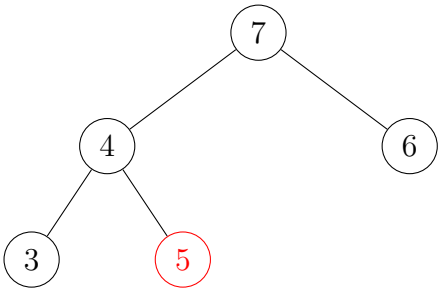
(b) Heapify-up vertauscht 7 und 6. 9 ist größer als 7 und Heapify-up endet.



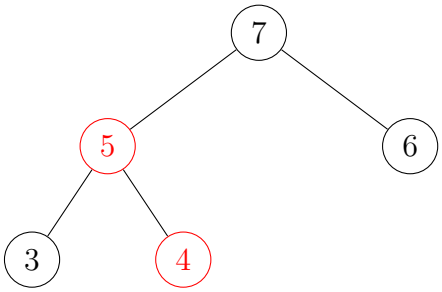
(c) Um 9 zu löschen wird es mit 6 vertauscht und dann gelöscht.



(d) Heapify-down vertauscht 6 mit 7, da 7 das größere der beiden Kinder von 6 ist und größer als 6 ist.

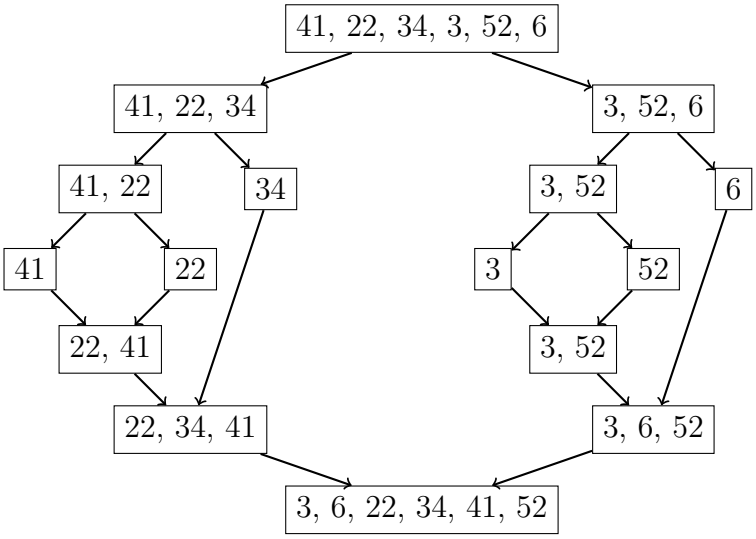


(e) Um 1 zu löschen wird es mit 5 vertauscht und dann gelöscht.



(f) Heapify-up vertauscht 5 mit 4.

Aufgabe 23



Aufgabe 24

```
FIND–UNIMOD–PIVOT(A):
  return FIND–UNIMOD–PIVOT(A, 0, n)

FIND–UNIMOD–PIVOT(A, i, j):
  if i == j - 1:
    return i
  x = (i + j - 1) / 2
  if A[x] < A[x + 1]:
    return FIND–UNIMOD–PIVOT(A, x + 1, j)
  else:
    assert A[x] > A[x + 1]
    return FIND–UNIMOD–PIVOT(A, i, x + 1)
```

Bei jedem Rekursionsschritt schauen wir uns die Steigung in der Mitte unseres Arraybereichs i bis exklusive j an. Wenn die Folge an diesem Punkt wächst, verwenden wir fortfolgend nur mehr die zweite Hälfte des Arrays, wenn sie fällt, die erste Hälfte. Irgendwann gelangen wir zum Basecase von einem Array mit einem Element, welches der Pivot sein muss.

Die Laufzeitanalyse geht analog zur Telefonbuchsuche. Bei jedem Rekursionsschritt wird die Problemgröße halbiert, jeder Rekursionsschritt hat eine asymptotisch konstante Laufzeit.

Bei leeren Arrays führt dieser Algorithmus einen ungültigen Indexzugriff durch. Das könnte man durch eine Anpassung des Basecases vermeiden, es stellt sich halt die Frage wie man solche Fälle behandelt, da ein leerer Array nicht unimodal ist (es gibt keinen Index p).