B1.2

(a) Beschreibung des Algorithmus Mergesort (2 VP):
Das Verfahren beruht auf dem Divide-and-Conquer-Prinzip (*teile und herrsche*). Das zu sortierende Feld wird zunächst in zwei Hälften aufgeteilt (*divide*), die jeweils für sich durch einen rekursiven Aufruf von Mergesort sortiert werden (*conquer*). Dann werden die sortierten Hälften zu einer insgesamt sortierten Folge verschmolzen (*combine*). Dabei werden wiederholt jeweils die vordersten Elemente der Hälften verglichen und das kleinere von beiden in das sortierte Feld übertragen.

(b) Erläuterung der Merge-Methode (3 VP)
Die Methode bekommt die Verweise auf den jeweiligen Anfang der Listen `a` und `b` übergeben. Wenn eine der beiden Listen leer ist, ist die jeweils andere Liste die Ergebnisliste und es wird der Verweis auf deren Anfang zurückgegeben.
Andernfalls werden die Daten der Listenköpfe verglichen. Der kleinere der beiden Listenköpfe wird als neuer Listenkopf des Ergebnisses genommen und die durch den rekursiven Aufruf verschmolzenen restlichen Elemente dahinter gehängt.

(c) Analyse der rekursiven Aufrufe (2 VP)
Rekursive Aufrufe erfolgen nur, solange beide Listen noch Elemente enthalten. Im besten Fall wird deswegen die kürzere Liste in jedem Schritt verkürzt; die Anzahl der Aufrufe ist daher mindestens so groß wie deren Länge.
Im schlechtesten Fall bleiben bis zum Schluss in beiden Listen Elemente übrig, deshalb benötigt man m+n rekursive Aufrufe:
$\min(n,m) \leq$ Anzahl der Aufrufe $\leq m+n$

Wahlaufgabe B2:

B2.1

(a) Implementierung der Klasse Listenknoten<T> (1 VP):

```
public class Listenknoten<T> {
  public T daten;
  public Listenknoten<T> nachfolger;
  public Listenknoten(T daten, Listenknoten<T> nachfolger) {
    this.daten = daten;
    this.nachfolger = nachfolger;
  }
}
```

(b) Implementierung von `enqueue` und `dequeue` (2 VP):

```
public void enqueue(T x) {
  dieListe.anhaengen(x);
}
public T dequeue() {
  T x = dieListe.get(0);
  dieListe.entferneBei(0);
  return x;
}
```