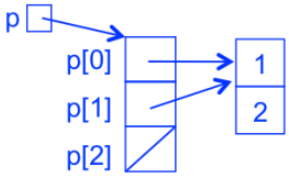
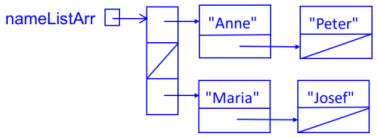
```java
class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public static void main(String[] a) {
        Point[] p = new Point[3];
        p[0] = new Point(1,2);
        p[1] = p[0];
    }
}
```



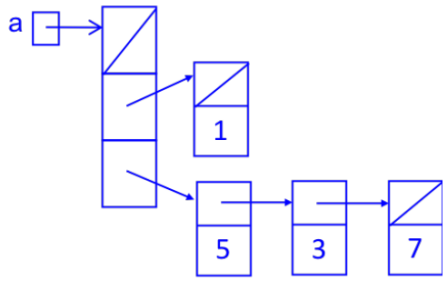

```java
class NameList {
    public String name;
    public NameList next;

    public NameList(String s, NameList l) {
        this.name = s;
        this.next = l;
    }
}
```

a) Beschreiben Sie mit einem Speicherbelegungsbild, was durch folgende Anweisungen geleistet wird:

```java
NameList[] nameListArr = new NameList[3];

nameListArr[0] = new NameList("Peter", null);
nameListArr[0] = new NameList("Anne", nameListArr[0]);
nameListArr[2] = new NameList("Maria", null);
nameListArr[2].next = new NameList("Josef", null);
```

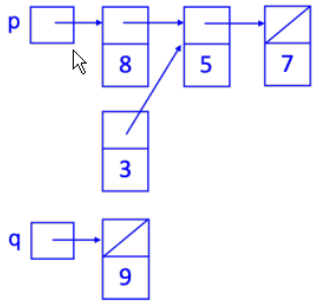



```java
public static void main(String[] sf) {
    Node[] a = new Node[3];
    a[2] = new Node(7, null);
    a[2] = new Node(5,a[2]);
    a[2].next = new Node(3, a[2].next);
    Node q = new Node(1, null);
    a[1] = q;

    for (Node r : a) {
        System.out.print("[");
        for (Node s = r; s != null; s = s.next)
            System.out.print(s.data + ", ");
        System.out.println("]");

    }
}
```

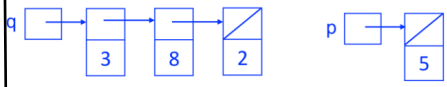



```
[ ]
[1, ]
[5, 3, 7, ]
```

```java
static class Node {
    Node next;
    int data;

    Node(int x, Node p) {
        next = p;
        data = x;
}

public static void main(String[] a) {
    Node p = new Node(5, null);
    p.next = new Node(7, p.next);
    p = new Node(8, p);
    Node q = p.next;
    q.next = new Node(3, q.next);
    q = new Node(9, null);
}
```




```java
class Node {
    Node next;
    int data;

    Node(int x, Node p) {
        next = p;
        data = x;
}

public static void main(String[] a) {
    Node p = new Node(2, null);
    p = new Node(8, p);
    Node q = p;
    q = new Node(3, q);
    p = new Node(5, null);
}
```




```java
class LV {
    public String name;
    public int sem;          // Semester, in dem die LV stattfindet
    public int anzStud;      // Anzahl Studierende in der LV
    LV(String n, int s , int n) {name = n; sem = s; anzStud = n;}
}
LinkedList<LV> lvList = new LinkedList<>();
lvList.add(new LV("Prog2",2,30));
lvList.add(new LV("Prog1",1,50));
// ...
```

a) Definieren Sie ein Prädikat istGross, das prüft, ob eine Lehrveranstaltung mehr als 40 Studierende hat.

```java
Predicate<LV> istGross = lv -> lv.anzStud >= 40;
```

b) Definieren Sie ein 2-stelliges Prädikat findetStatt, das prüft, ob die Lehrveranstaltung lv im Semester sem stattfindet.

```java
BiPredicate<LV, Integer> findetStatt = (lv, sem) -> lv.sem == sem;
```

c) Schreiben Sie einen removeIf-Aufruf, der aus der Liste lvList alle Lehrveranstaltungen entfernt, die nicht im 2. Semester stattfinden. Verwenden Sie dabei das Prädikat findetStatt aus b).

```java
lvList.removeIf(lv -> !findetStatt.test(lv,2));
```

d) Erzeugen Sie aus der Liste lvList einen Strom und berechnen Sie die Anzahl der großen Lehrveranstaltungen. Verwenden Sie dabei das Prädikat istGross aus a).

```java
long size = lvList.stream()
                  .filter(istGross)
                  .count();
```

e) Erzeugen Sie aus der Liste lvList einen Strom und berechnen Sie die Anzahl der Studierende, die im zweiten Semester studieren. Verwenden Sie dabei das Prädikat findetStatt aus b).

```java
long anz2Sem = lvList.stream()
                     .filter(lv -> findetStatt.test(lv,2))
                     .mapToInt(lv -> lv.anzStud)
                     .sum();
```

a) Definieren Sie ein Prädikat Pruefung istBestanden = n -> n.note <= 4.0;

b) Definieren Sie einen Comparator (mit Typ) für Prüfungen als Lambda-Ausdruck, indem die Noten numerisch verglichen werden. (3 Punkte)

```java
Comparator<Pruefung> comparePruefung = (p1, p2) -> {
    if (p1.note < p2.note) return -1;
    else if (p1.note > p2.note) return +1;
    else return 0;
};
```

c) Erzeugen Sie aus pruefListe einen Strom und berechnen Sie mit Hilfe von Strom-Operationen die Durchschnittnote für alle bestandenen Prüfungen im Fach „Prog2“ aus. Benutzen Sie das Prädikat bestanden aus a). (5 Punkte)

```java
OptionalDouble average = pruefListe.stream()
    .filter(istBestanden)
    .filter(p -> p.fach.equals("Prog2"))
    .mapToDouble(p -> p.note)
    .average();
```

d) Erzeugen Sie aus pruefListe einen Strom und geben Sie die von „Maier“ absolvierten Prüfungen aus. Die Ausgabe ist alphabetisch nach dem Fach sortiert und besteht nur aus Fach und Note. (4 Punkte)

```java
pruefListe.stream()
    .filter(p -> p.name.equals("Maier"))
    .sorted(Comparator.comparing(p -> p.fach))
    .forEach(p -> System.out.println(p.fach + ": " + p.note));
```

e) Erzeugen Sie aus pruefListe einen Strom und sammeln Sie die Namen aller Studenten in einer sortierten Liste. Hinweis: collect-Aufruf verwenden. (4 Punkte)

```java
List<String> nl = pruefListe.stream()
    .map(p -> p.name)
    .sorted()
    .collect(Collectors.toList());
```

a) Definieren Sie ein 2-stelliges Prädikat geborenNach, das prüft, ob ein Kunde k im Jahr j oder später geboren wurde.

```java
BiPredicate<Kunde,Integer> geborenNach = (k,j) -> k.geb >= j;
```

b) Schreiben Sie eine foreach-Schleife, die mit Hilfe des Prädikats geborenNach aus a) alle Kunden aus der Liste kundenListe ausgibt, die im Jahr 2000 oder später geboren wurden.

```java
for (Kunde k : kundenListe)
    if (geborenNach.test(k,2000))
        System.out.println(k);
```

c) Erzeugen Sie aus der Liste kundenListe einen Strom und geben Sie mit Hilfe von Strom-Operationen die 10 jüngsten Kunden mit der Postleitzahl 78462 aus:

```java
kundenListe.stream()
    .filter(k -> k.plz.equals("78462"))
    .sorted(Comparator.comparing((Kunde k) -> k.geb).reversed())
    .limit(10)
    .forEach(System.out::println);
```

d) Erzeugen Sie aus der Liste kundenListe einen Strom und geben Sie mit Hilfe von Strom-Operationen alle Namen von Kunden (nur die Namen!) in alphabetischer Reihenfolge aus, die im Jahr 2000 oder später geboren wurden.

```java
kundenListe.stream()
    .filter(k -> geborenNach.test(k,2000))
    .map(k->k.name)
    .sorted()
    .forEach(System.out::println);
```

Laufzeiten von Quicksort

	Best-Case	Average-Case	Worst-Case
	O(n log n)	O(n log n)	$O(n^2)$

```java
class Verwandtschaft {
    private Map<String, String> mutterVonKind = new TreeMap<>();

    public void addKindMutter(String kind, String mutter) {
        mutterVonKind.put(kind, mutter);
    }

    Set<String> getAlleKinder(String n) {
        Set<String> kinder = new TreeSet<>();
        for (var mbez : mutterVonKind.entrySet()) {
            if (mbez.getValue().equals(n))
                kinder.add(mbez.getKey());
        }
        return kinder;
    }

    Set<String> getAlleGeschwister(String n) {
        if (!mutterVonKind.containsKey(n))
            return new TreeSet<>();
        Set<String> kinder = new TreeSet(getAlleKinder(mutterVonKind.get(n)));
        kinder.remove(n);
        return kinder;
    }

    boolean istTanteVon(String x, String y) {
        if (!mutterVonKind.containsKey(y))
            return false;
        else
            return getAlleGeschwister(mutterVonKind.get(y)).contains(x);
    }
}
```