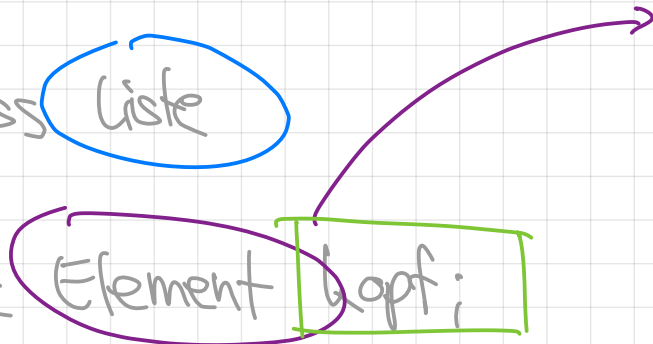


# Dynamische Datenstrukturen

## Einfach verkettete Liste

```
public class Liste
{
    public Element Kopf;
    public int length;

    public void Add(int value)
    {
        // ...
    }
}
```

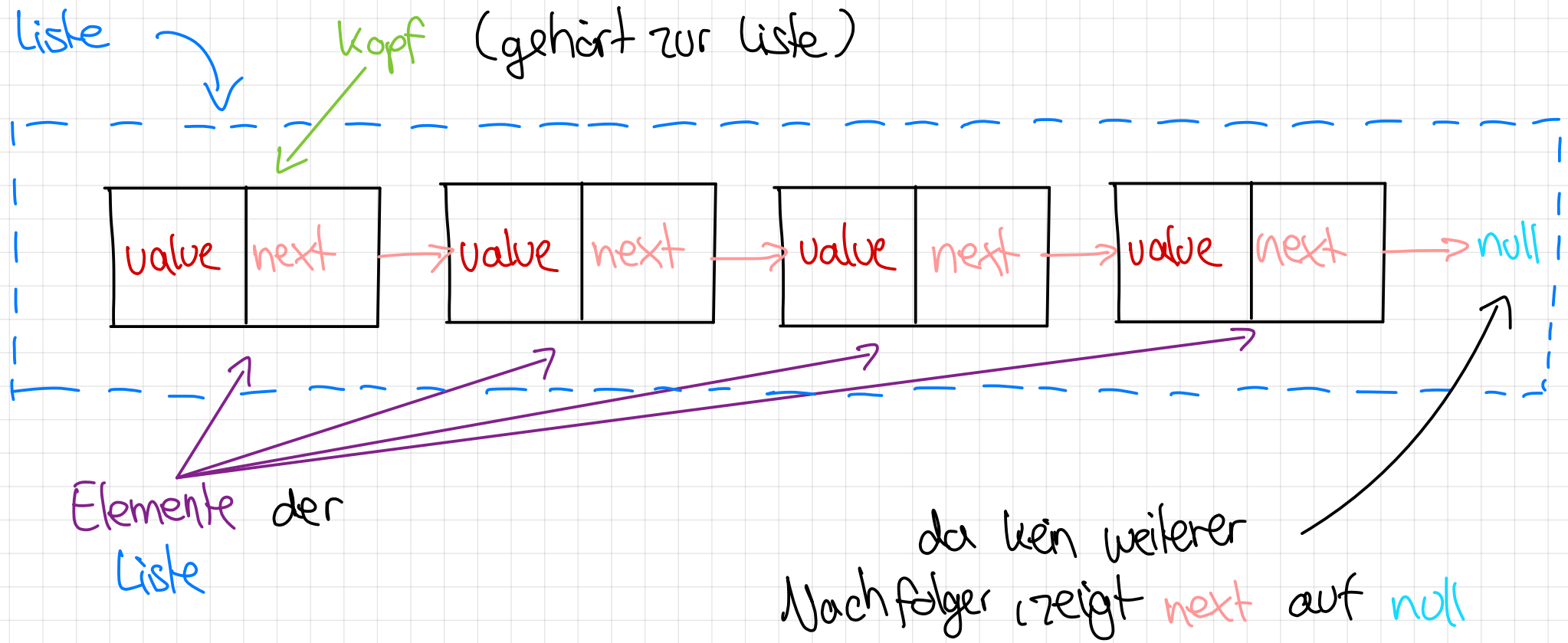


```
public class Element
{
    public int value;
    public Element next;

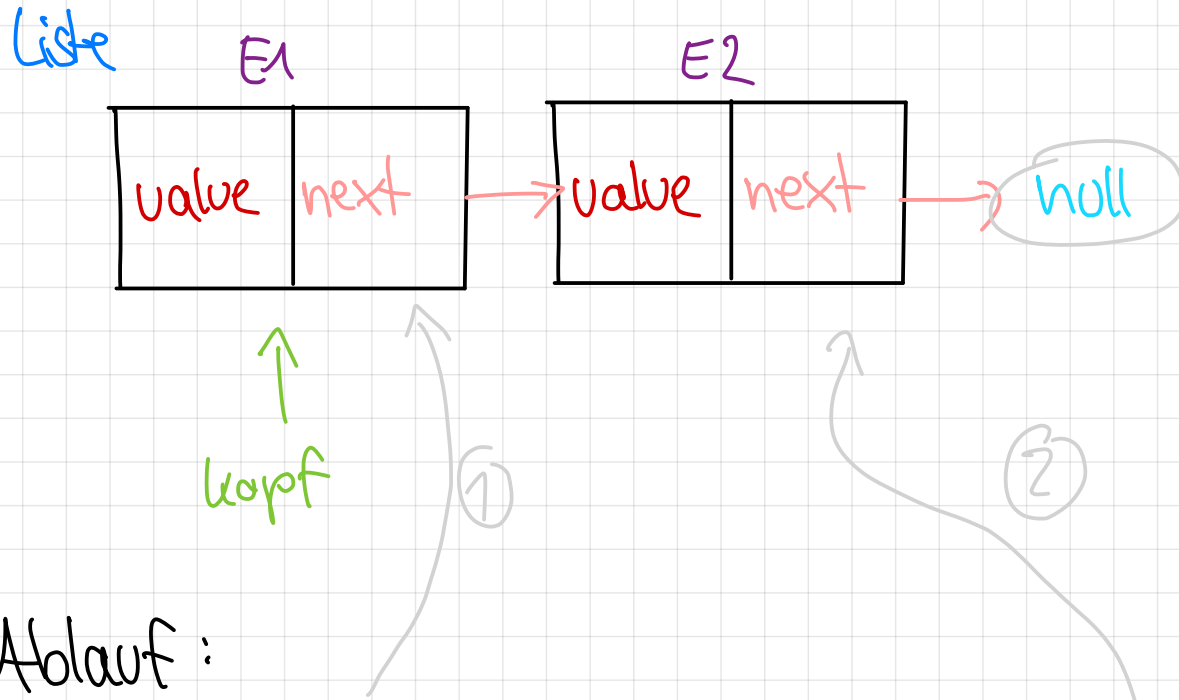
    public void Add(int value)
    {
        // ...
    }
}
```

Eine Liste besteht aus einer übergeordneten Klasse Liste, welche nur das erste Element in der Liste kennt.

Jedes Element kennt (bei einer einfach verketteten Liste) nur seinen Nachfolger (das Element, welches logisch gesehen in der Liste als nächstes kommt, und hat einen Wert).



# Einfügen ans Ende der Liste



Ablauf:

1.  $next == null$  ? (L3)  $\rightarrow$  nein (L9)
2.  $next == null$  ? (L3)  $\rightarrow$  ja (L5)

```
1 public void Addl (int v)
2 {
3     if (next == null)
4     {
5         next = new Element(v);
6     }
7     else
8     {
9         next.Addl(v);
10    }
11 }
```

rekursiver Aufruf im Nachfolger

Jedes **Element** hat einen Nachfolger. Da jeder Nachfolger auch ein **Element** ist, hat dieser die gleichen Methoden wie die Vorgänger.

→ dies macht den rekursiven Aufruf möglich

Zur Liste oben:

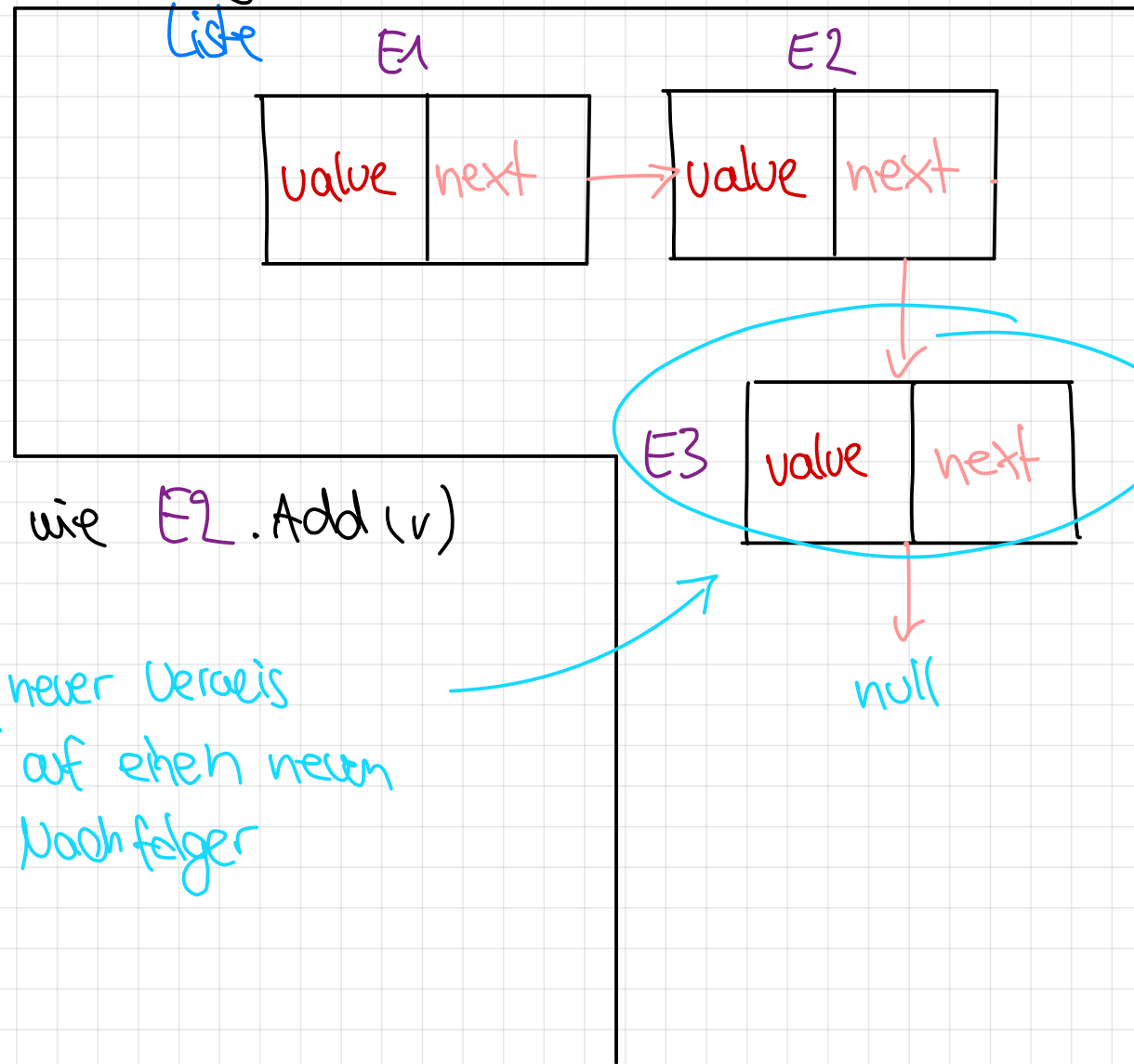
$E1.next == null ? \rightarrow \text{nein}$

$E1.next.Add(v) \rightarrow \text{das Gleiche wie } E2.Add(v)$

$E2.next == null ? \rightarrow \text{ja!}$

$E2.next = \underbrace{\text{new Element}(v)}_{E3};$

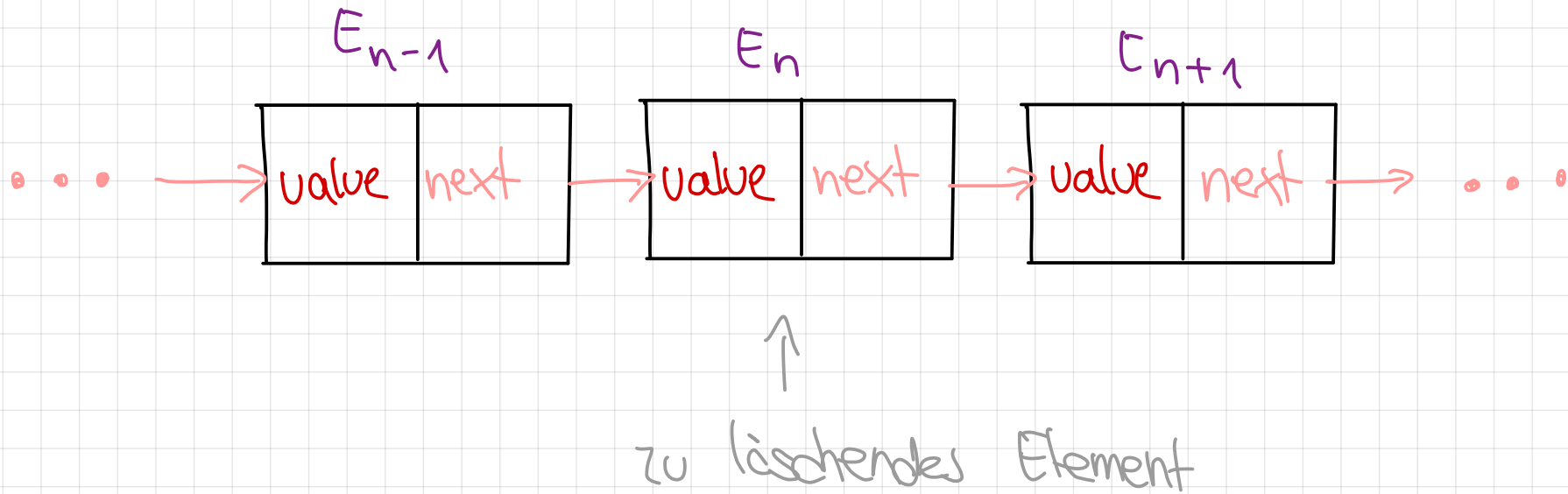
neuer Verweis  
auf einen neuen  
Nachfolger



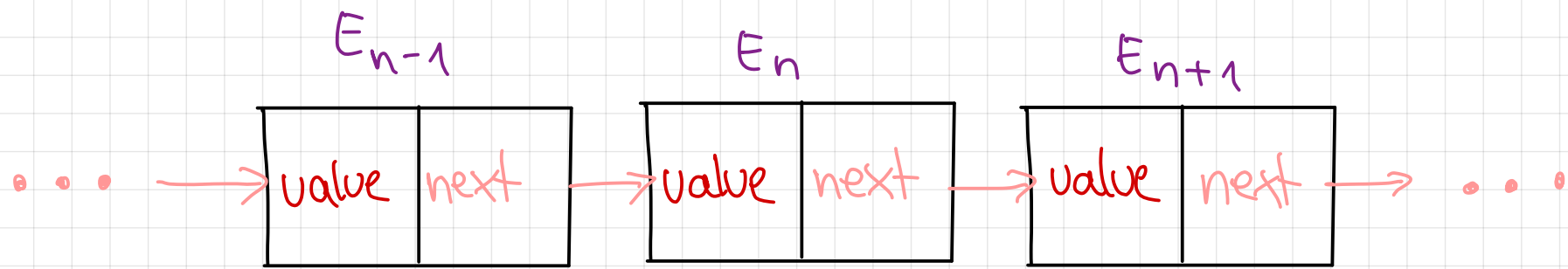
Zu erkennen: listen müssen immer **Element für Element** durchlaufen werden, der bei direktem Verweis auf das letzte **Element** der **liste** existiert.

---

löschen eines **Elements** der **liste**

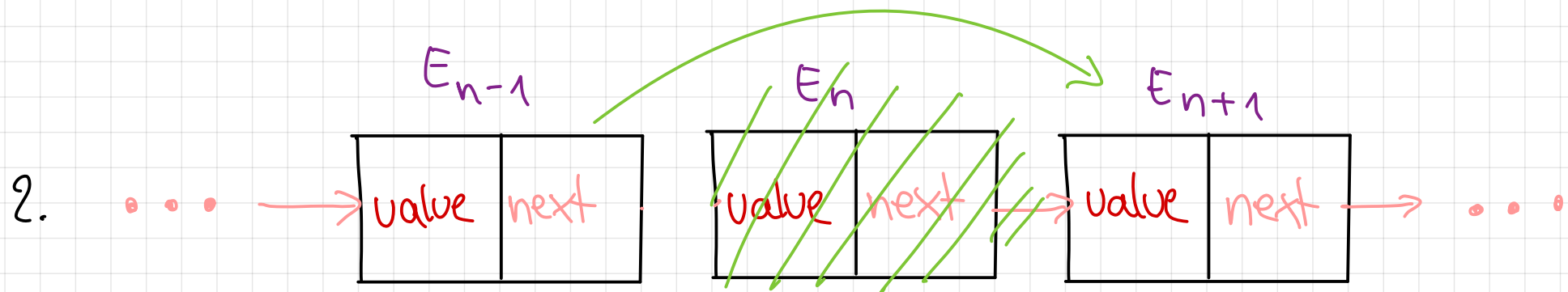
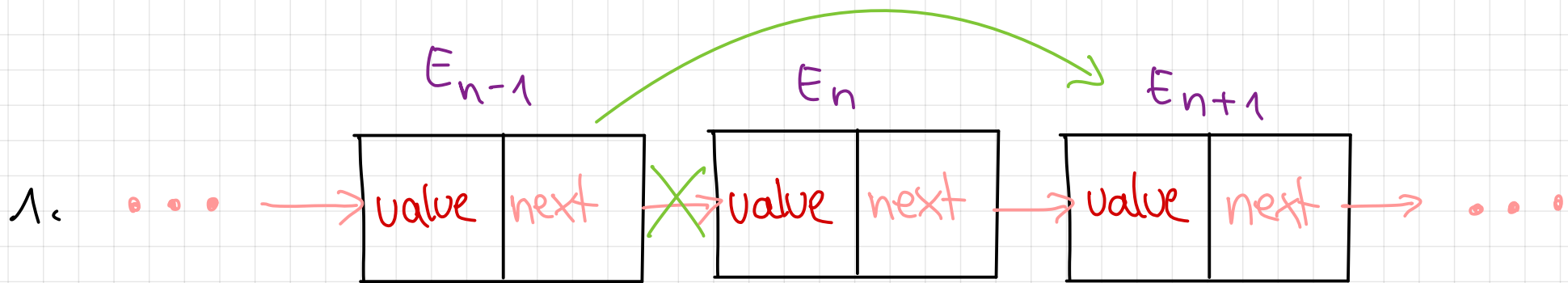


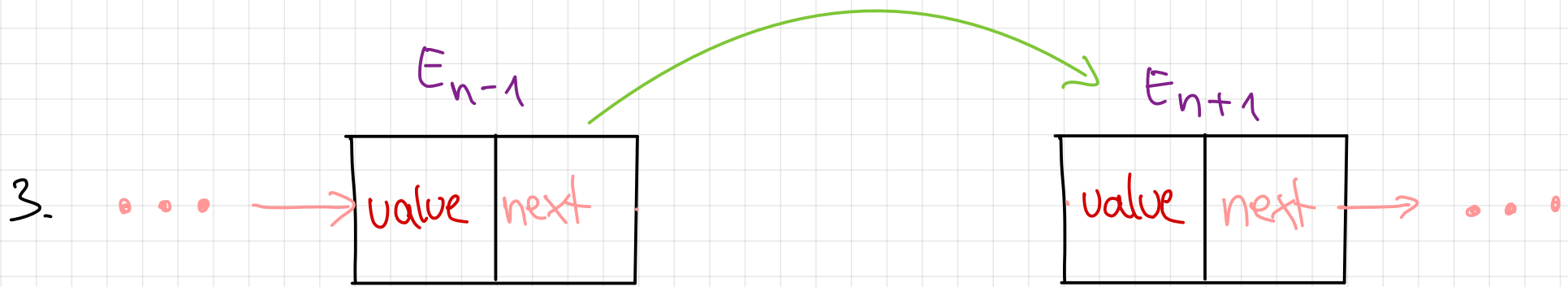
Sobald man durch die **liste** gelaufen ist, und auf das zu löschende **Element** getroffen ist, ist das Löschen an sich sehr trivial.  
(das finden dieses Elements ist sehr zeitaufwendig, das Löschen nicht).



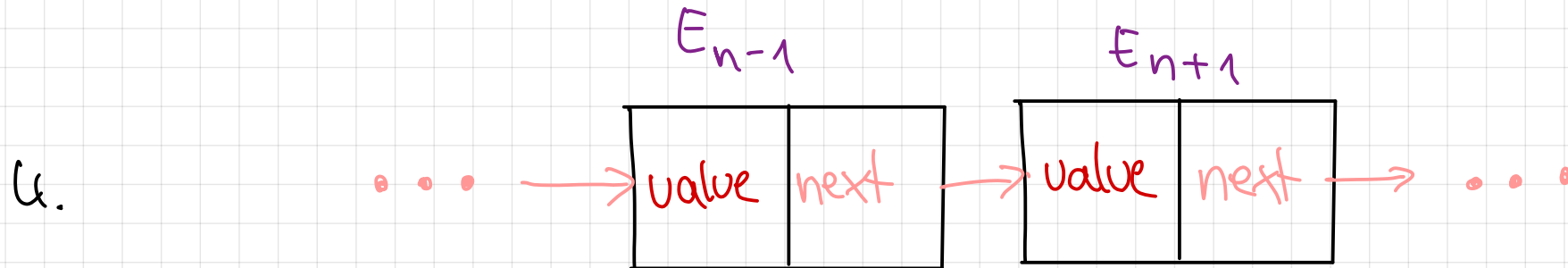
Der Nachfolger von  $E_{n-1}$  wird von  $E_n$  auf  $E_{n+1}$  gesetzt.

$\rightarrow$  nun ist das Element bereits effektiv aus der Liste gelöscht.





(Das eigentliche Löschen des Objekts aus dem Speicher übernimmt dann irgendwann der Garbage Collector)



Wann haben Listen einen Vorteil?

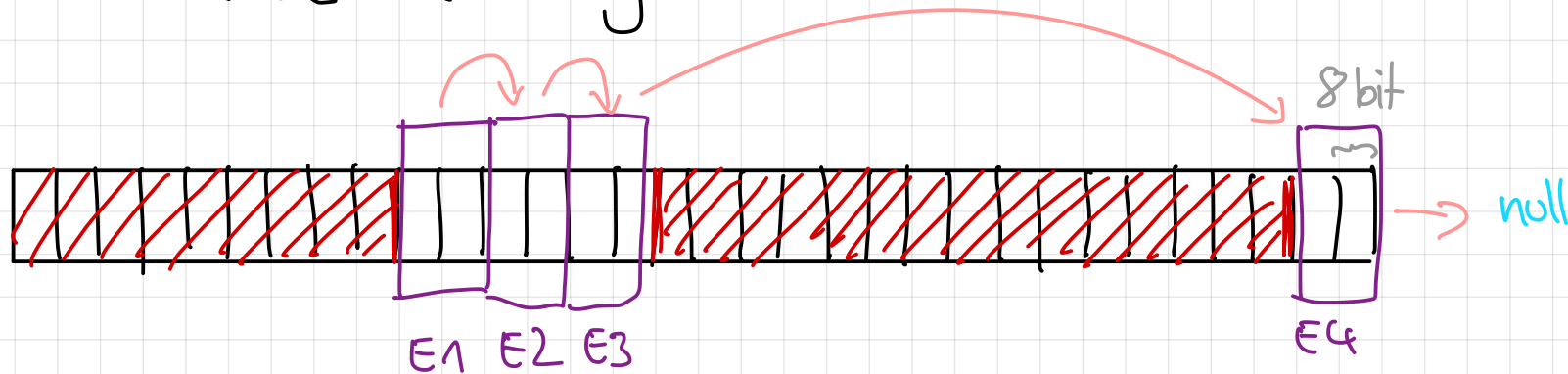
- Listen müssen (im Gegensatz zu Arrays) nicht hintereinander im Speicher stehen



bereits belegter  
Speicher

für ein Array der Länge  $k$  (je 16 bits) ist kein Platz im Speicher!

für eine Liste allerdings schon:





- Einfügen und Löschen von **Elementen** (sobald gefunden) sehr schnell.

Wann haben **Listen** einen Nachteil?

- höherer Speicherbedarf durch Referenz auf den **Nachfolger**
- kein Indexing wie bei Arrays möglich (kein konstant schneller Zugriff auf **Elemente** möglich; ganze Liste muss immer vom **Kopf** an durchlaufen werden!)

Interessant: **doppelt** verkettete Listen haben noch einen Verweis auf den Vorgänger, alles andere bleibt wie bei einfach verk. Listen

