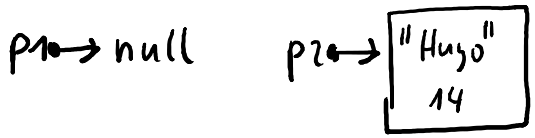
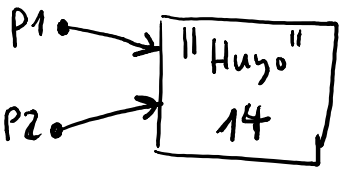
1 Bei Deklaration als class werden Zeiger zugewiesen

```
Person p1;    Person p2=new Person("Hugo", 14)
```

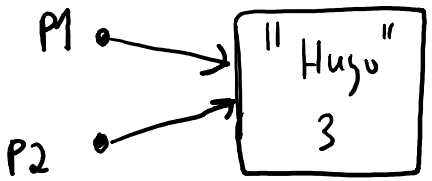
```
class Person
{
  string  name;
  int     age
}
```




```
p1=p2 // Zeiger zuweisen
```




```
p1.age = 3;
```

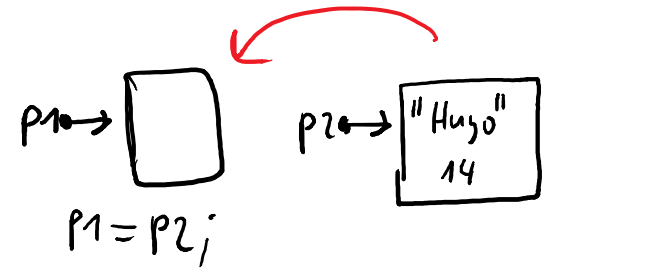


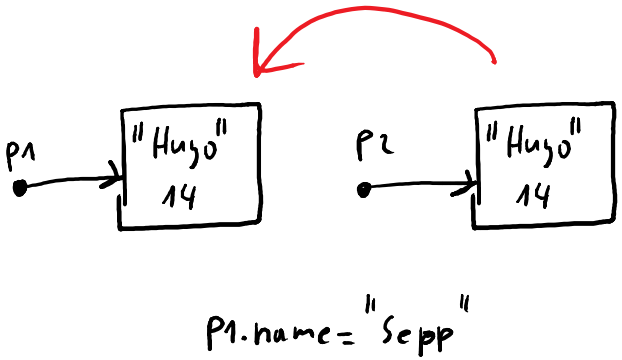
In C++ würde das so aussehen

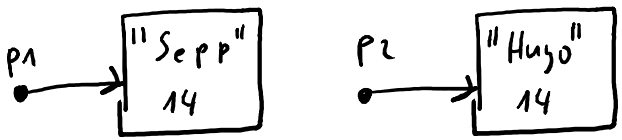
```
calss Person
{
  string  name;
  int     age
}

Person* p1;  Person* p2 = new Person("Hugo", 14);

p1 = p2;
```

2 Bei Deklaration als struct wird bei einer Zuweisung das ganze Objekt kopiert

```
Person p1;    Person p2=new Person("Hugo", 14)
```








```
struct Person
{
  string  name;
  int     age
}
```

Aus diesem Grund werden neue Datentypen
Wie z.B. Vector, Complex Matrix . Point. . . .
in C# als struct und nicht als class implementiert da man sich bei diesen Datentypen bei der Zuweisung Wertsemantik und nicht Reference-Semantik erwartet.

In C++ würde das so aussehen

```
calss Person
{
  string  name;
  int     age
}

Person p1("Sepp",13);
Person p2("Hugo",14);
p1 = p2;
```

3 Literatur zu Pointern

Gallileo Computing C von A bis Z

Kap. 12 Zeiger und Kap. 14 Dynamische Speicherverwaltung

Das Buch ist im Internet frei verfügbar

4 Datenstruktuten und Container Klassen

Datensrukturen werden zur effizienten Speicherung der Objekte eines objektorientierten Programms verwendet.
Beispiele:

- Verwaltung der Buchstaben, und Grafiksymbole in Winword.
- Verwaltung von Raumschiffen und Torpedos in einem StarWars-Spiel.
- Verwaltung der Billiardkugeln in einer Spielsimulation.

In ihrer objektorientierten Verpackung werden Datensrukturen auch als Kontainer bezeichnet.
Kontainer stellen üblicherweise die folgenden Operationen für die im Kontainer gespeicherten Objekte bereit:

- Einfügen eines Objektes (und Allokation von Speicherplatz)
- Finden eines Objektes wobei nach einer Objekteigenschaft (Attribut) gesucht wird.
  (z.B. Name, Farbe, Größe . . .)
- Löschen (entfernen) eines Objektes aus dem Kontainer.
- Sortieren der Objekte im Kontainer nach einer Eigenschaft
  (z.B sortieren nach Name oder nach Katalognummer)
- Zugriff über Index (das 3 te Objekt im Container)
- Iteration besuchen aller Objekte im Container

Kontainer können auf verschiedene Weise implementiert sein und jede Implementierung löst die oben beschriebenen Aufgaben eines Kontainers unterschiedlich gut.
Wir werden und die Datenstrukturen Array aus Zeigern und verkettete Liste (linked List) näher ansehen.

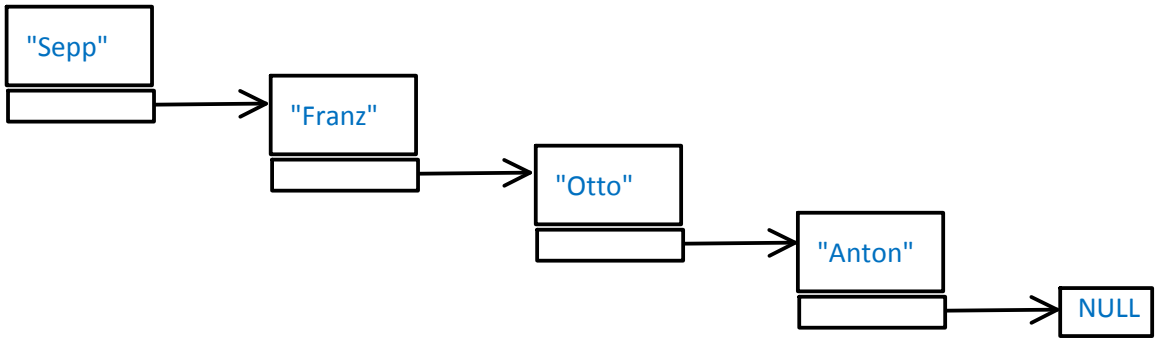
In der untenstehenden Tabelle sind die Vor und Nachteile der beiden Datensrukturen für typische Kontaineroperationen aufgelistet.

5 Datenstruktuten und Container Klassen

	Liste	**Array**	**Dynamisches Array**
Indizierter Zugriff(wie schnell ?)	**--**	**+**	**+**
Einfügen an beliebiger Stelle	**+**	**--**	**--**
Herausnehmen (Löschen) an beliebiger Stelle	**+**	**--**	**--**
Sortiert halten	**+**	**--**	**--**
Speicherverwaltung (Effizienz)	**+**	**--**	**+**

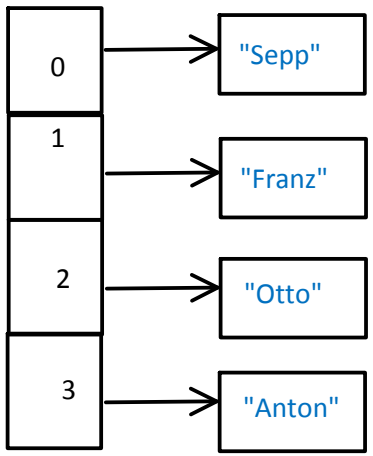
Skizze oder noch besser Film wie die obigen Funktionen bei Liste und Array funktionieren

6 Array vs. Liste



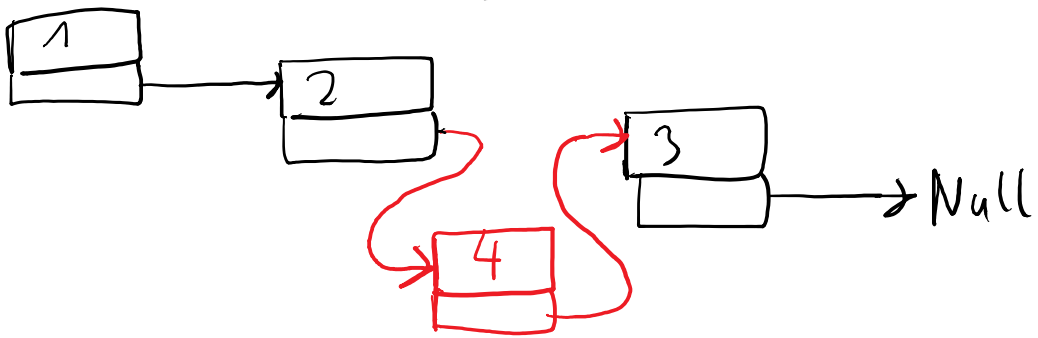
Beim Einfügen und Löschen von Objekten ist die Liste besser geeignet

Beim Indizierten Zugriff A[i] ist das Array besser geeignet

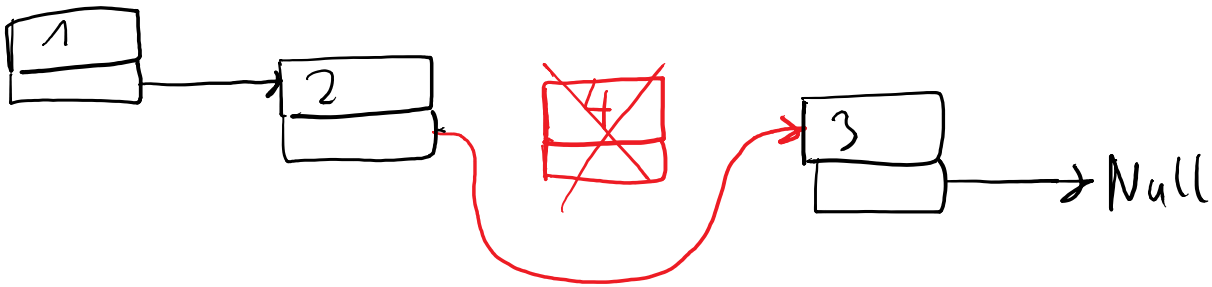


7 Objekt in eine Liste einfügen und aus einer Liste entfernen

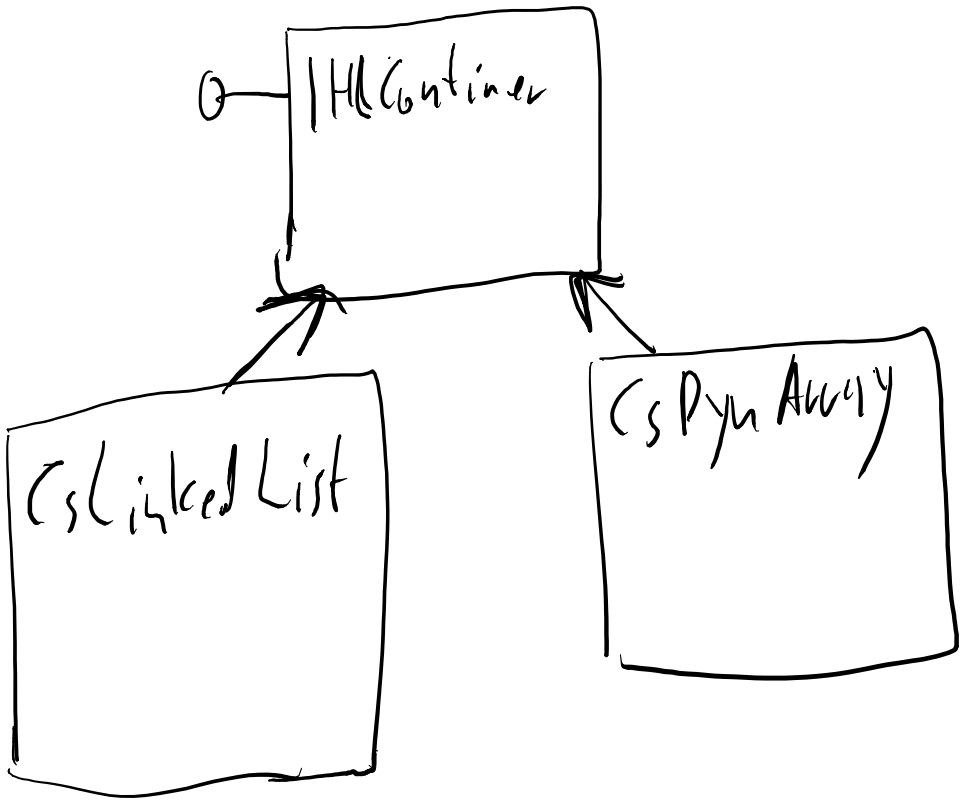
einketten



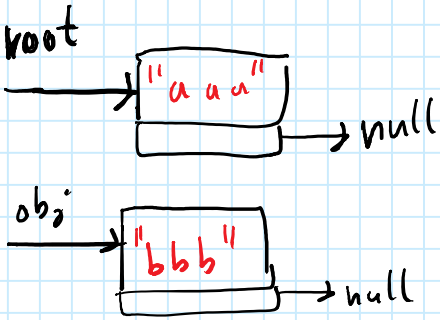
ausketten



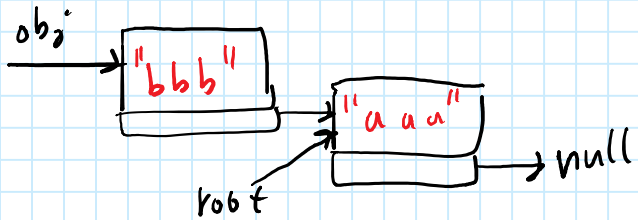
8 IHLContainer



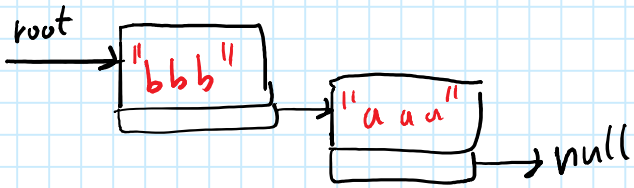
9 AddHead



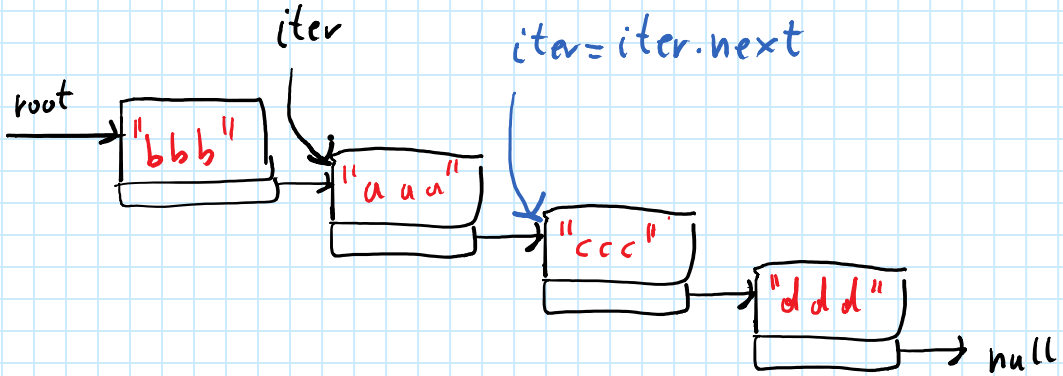
obj.next = root;



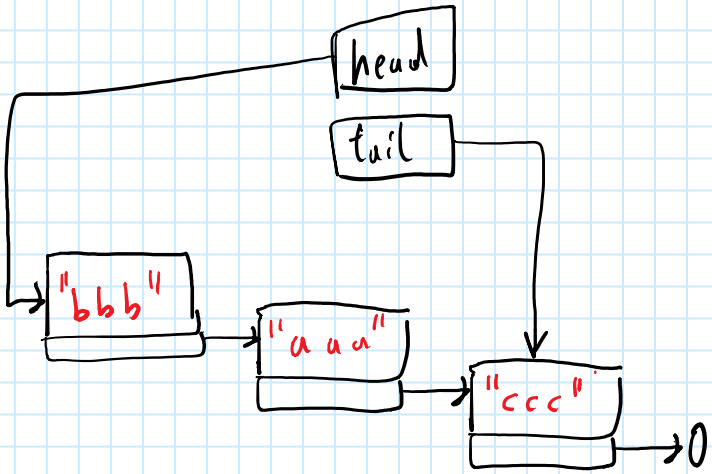
root = obj;

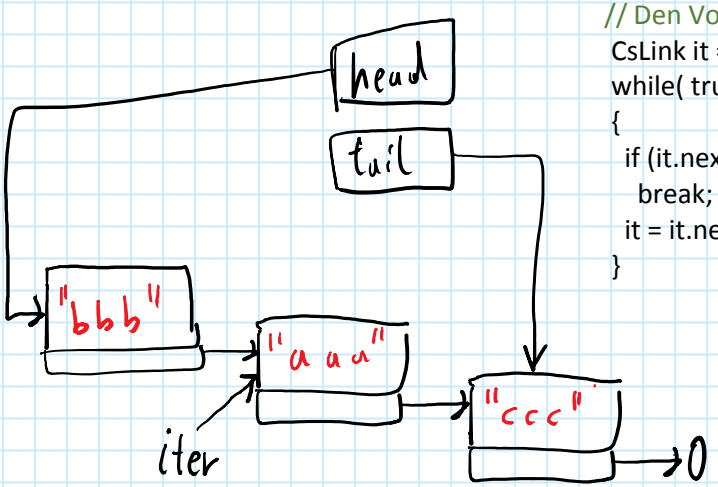


Iterate List

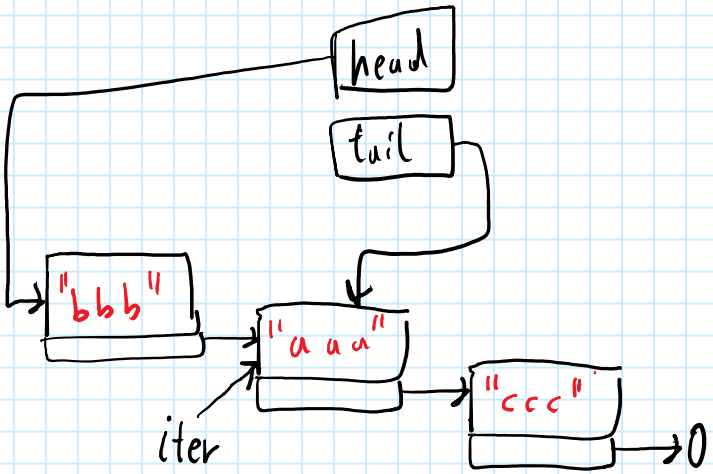


10 RemoveTail

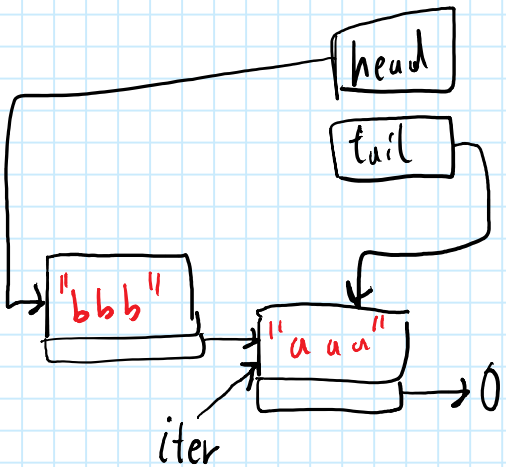





```
// Den Vorgänger von Tail suchen
CsLink it = _head;
while( true )
{
 if (it.next == _tail) // Vorgänger gefunden
   break;
 it = it.next; // iter++
}
```

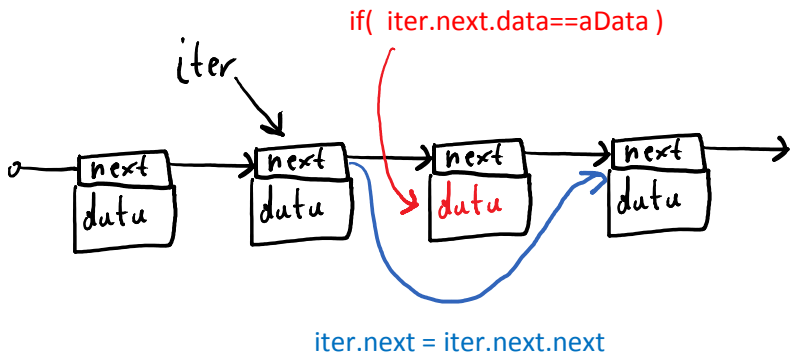


tail=iter

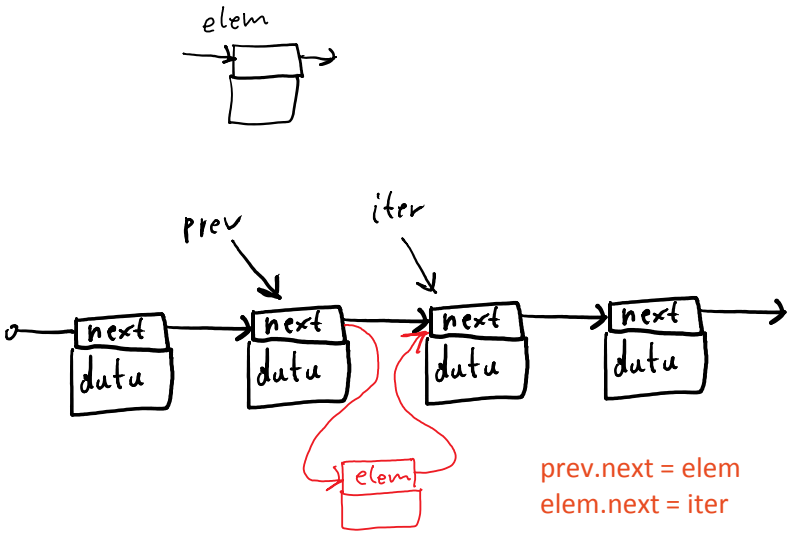


11 Remove(object aData)

```
// aObj ist ein Zeiger auf ein Objekt im Container
// d.h. Remove macht eigentlich nur zusammen mit Find Sinn
public void Remove(object aData)
{
  object dat = First();
  while (dat != null)
  {
    if (_iter.next.data == aData)
    { // ausketten und fertig
      _iter.next = _iter.next.next;
      return;
    }
    dat = Next();
  }
  return;
}
```



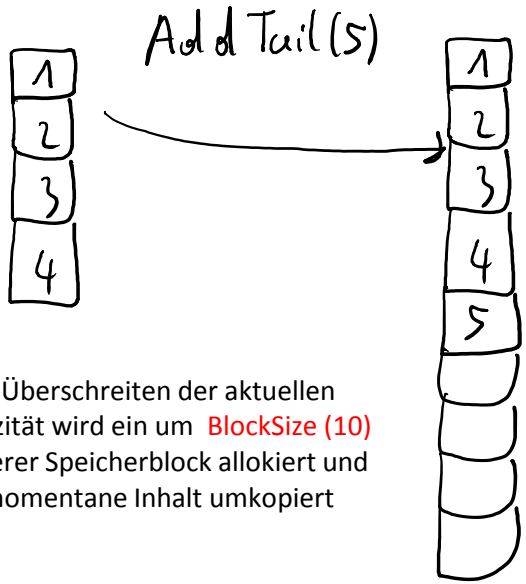
11a InsertSorted




```
public void InsertSorted(object aObj, IComparer aCmp)
  {
   CsLink iter;
   CsLink prev;
   CsLink elem = new CsLink(aObj);

   iter = prev = _head;
   while (iter != null)
   {
    if( aCmp.Compare(iter.data,aObj)>0 )
    { // vor iter einfügen
     if (iter == _head)
     {
      elem.next = _head;
      _head = elem;
     }
     else
     {
      prev.next = elem;
      elem.next = iter;
     }
     return;
    }
    prev = iter;
    iter = iter.next;
   }
   // ansonsten hinten anhängen
   _tail.next = elem;
   _tail = elem;
  }
```

12 Dynamisches Array

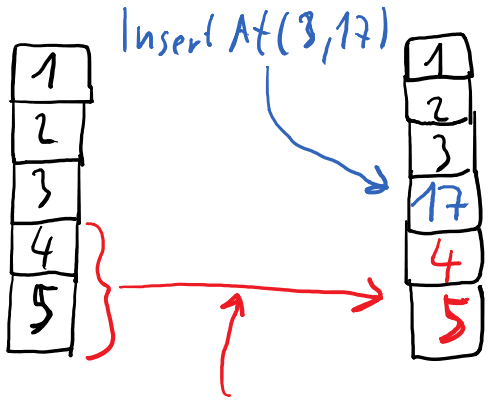
Array wächst mit den Datenanforderungen mit



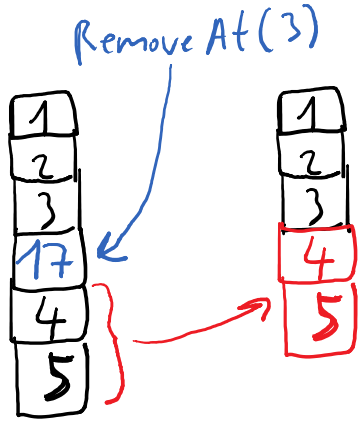
Beim Überschreiten der aktuellen Kapazität wird ein um BlockSize (10) größerer Speicherblock allokiert und der momentane Inhalt umkopiert

_count . . . . . Anzahl der gültigen Einträge
_capacity . . . Anzahl der möglichen Einträge

Bei Insert und Remove müssen beim DynArray Speicherblöcke verschoben werden



Wird um einen Slot nach hinten verschoben



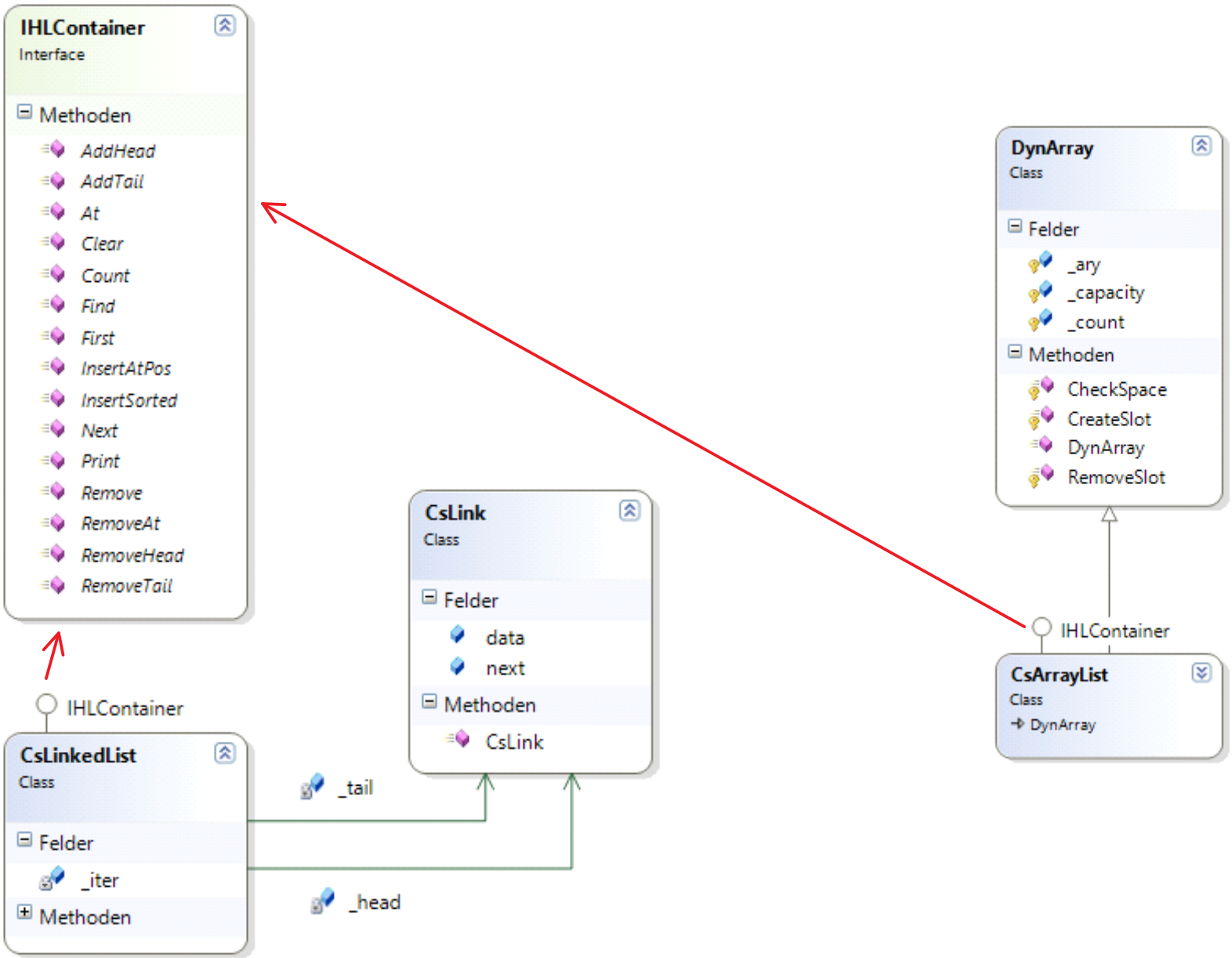
Wird um einen Slot nach vorne verschoben

```
class DynArray  {
  int _capacity;
  int _count;
  object[] _ary;

  public DynArray()
  void CheckSpace()
  void CreateSlot(int aIdx)
  void RemoveSlot(int aIdx)
}
```

Die BasisKlasse DynArray stellt die notwendigen Grundoperationen für Dynamische Arrays zur Verfügung.

CheckSpace() . . . . . // CheckSpace und wenn nötig um BlockSize vergrößern
CreateSlot() . . . . // Zum Einfügen an einem bestimmten Index
RemoveSlot() . . . . // Zum Entfernen und wieder zusammenrücken

13 Klassen Diagramme



$(1 + j \cdot 2) * (3 + j \cdot 4) =$

$= 3 + j \cdot 6 + j4 - 8 \equiv \underbrace{(3-8)}_{re} + j\underbrace{(4+6)}_{im}$

im

15 Objekt in eine Liste einfügen und aus einer Liste entfernen