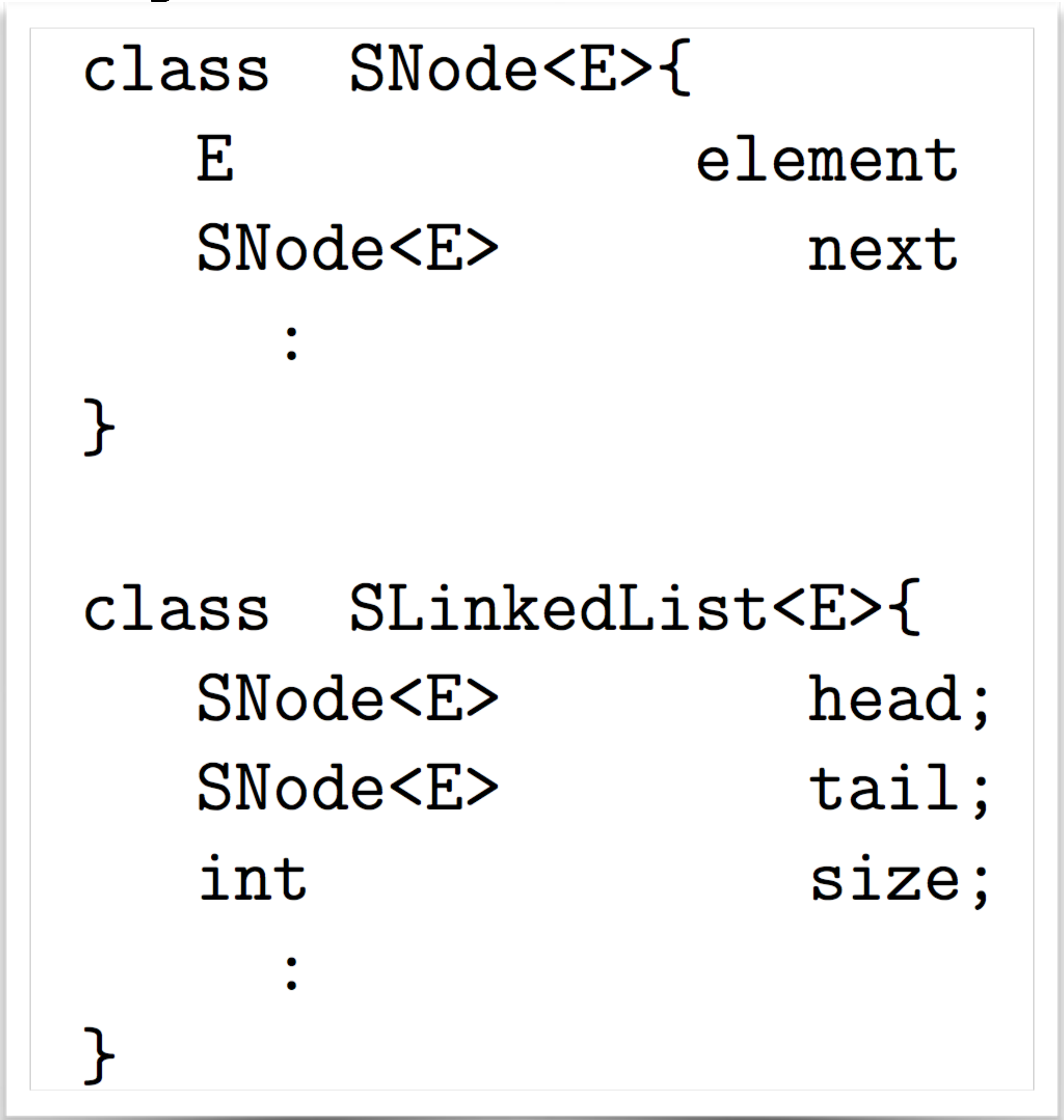
Winter 2016 COMP-250: Introduction to Computer Science

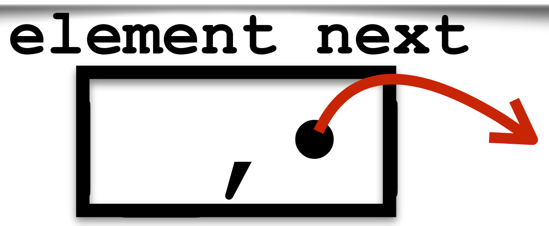
Lecture 6, January 28, 2016

Java Generics

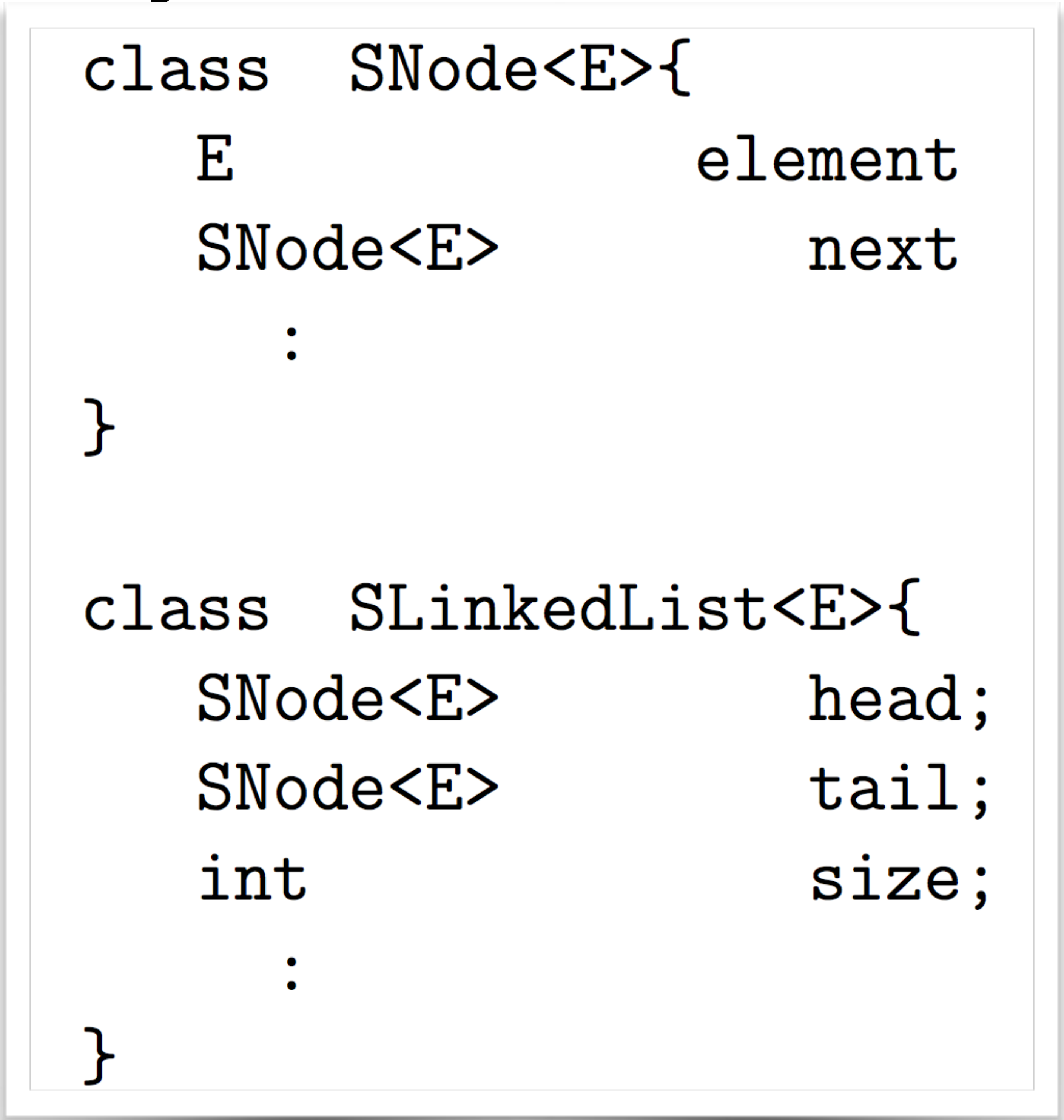


```
class  SNode<E>{
   E             element
   SNode<E>         next
     :
}

class  SLinkedList<E>{
   SNode<E>         head;
   SNode<E>         tail;
   int              size;
     :
}
```



Java Generics



```
class  SNode<E>{
   E               element
   SNode<E>           next
     :
}

class  SLinkedList<E>{
   SNode<E>           head;
   SNode<E>           tail;
   int                size;
     :
}
```

```
SLinkedList<Shape>    shapelist = new SLinkedList<Shape>();
SLinkedList<Student>  studentlist = new SLinkedList<Student>();
```

Java Generics

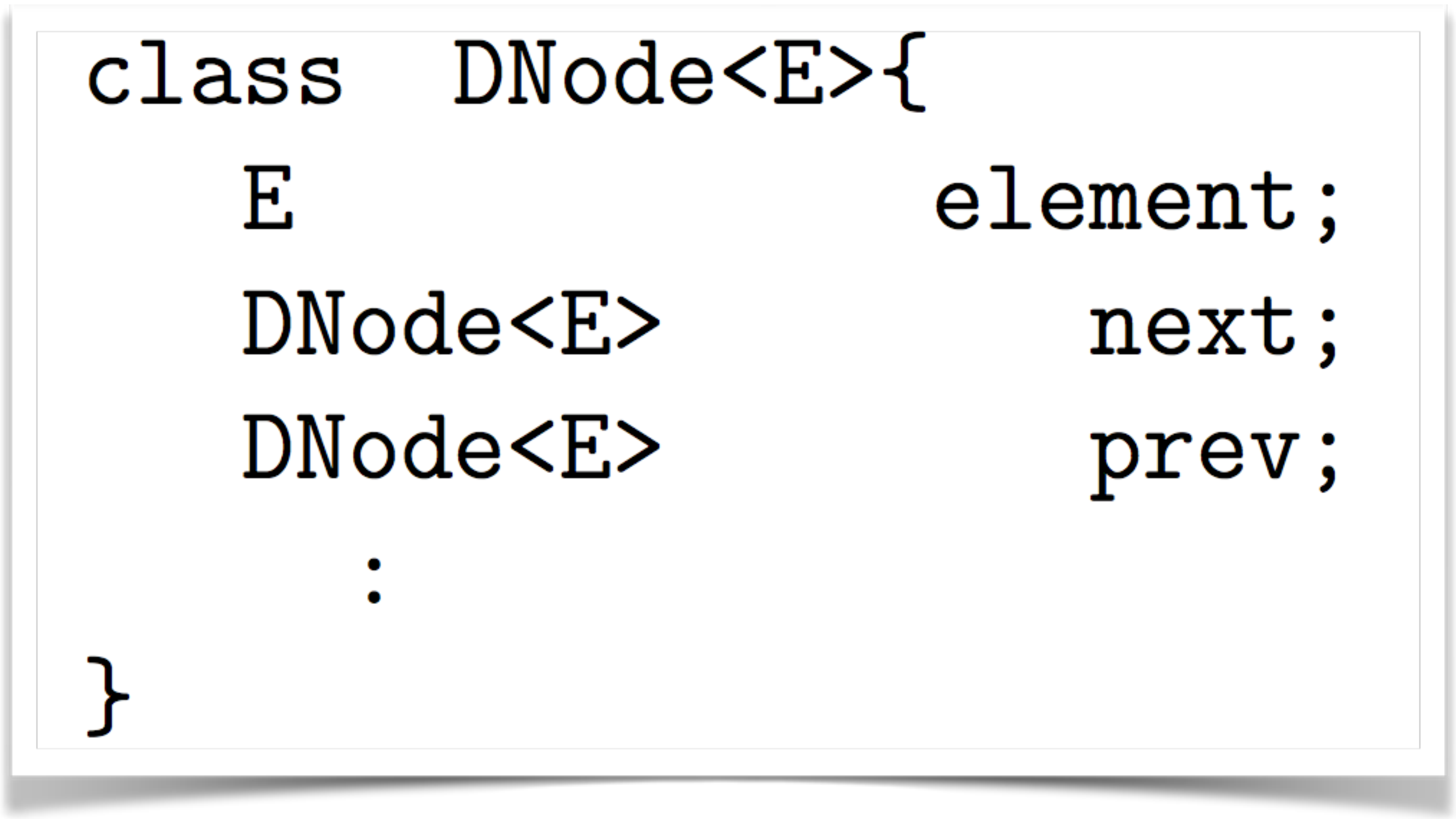
```
class  DNode<E>{
   E                element;
   DNode<E>            next;
   DNode<E>            prev;
     :
}

class  DLinkedList<E>{
   DNode<E>            head;
   DNode<E>            tail;
   int                 size;
     :
}
```

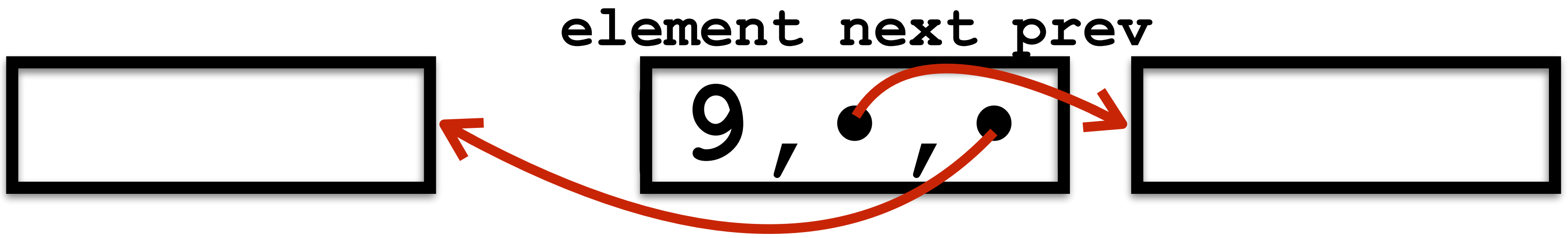
```
DLinkedList<Shape>    shapelist = new DLinkedList<Shape>();
DLinkedList<Student>  studentlist = new DLinkedList<Student>();
```

(Doubly) Linked List

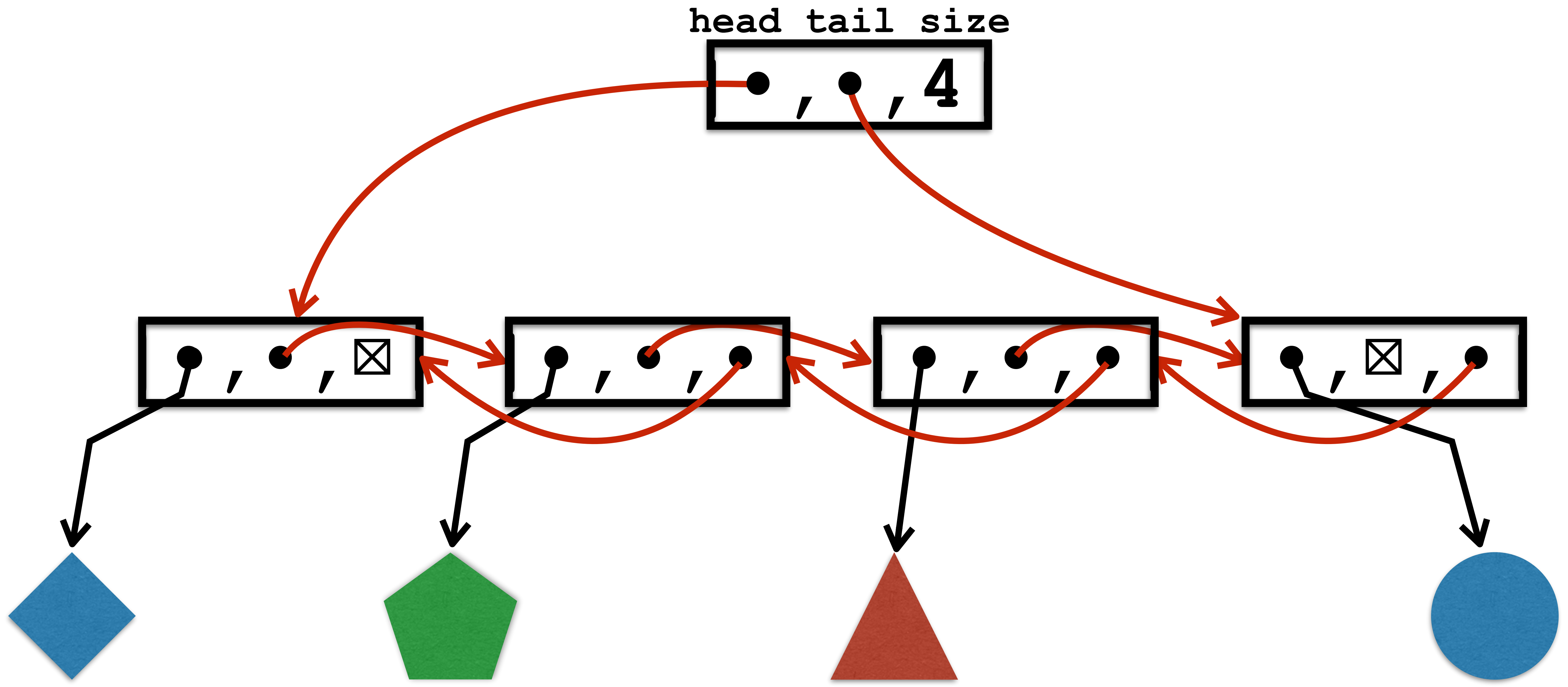
(Doubly) Linked List Node



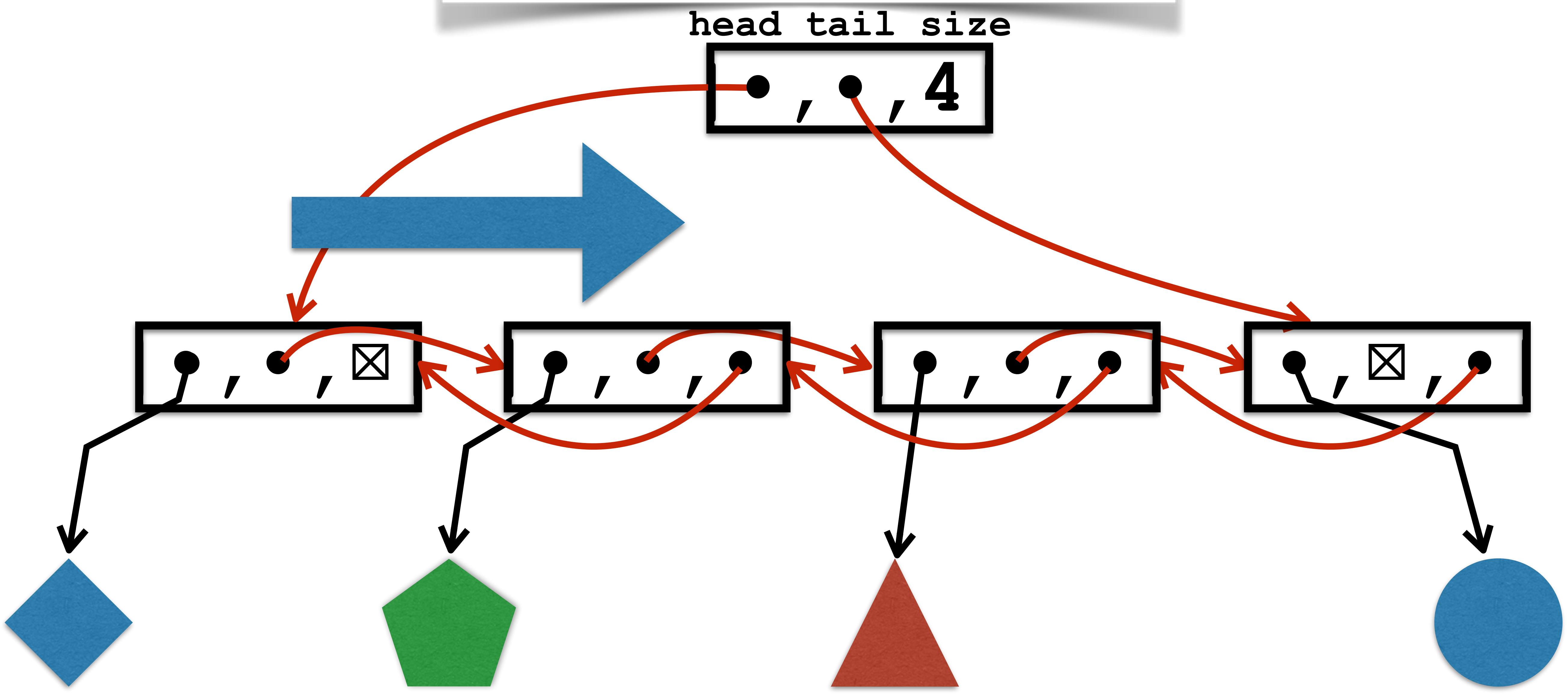
```
class  DNode<E>{
    E              element;
    DNode<E>          next;
    DNode<E>          prev;
       :
}
```



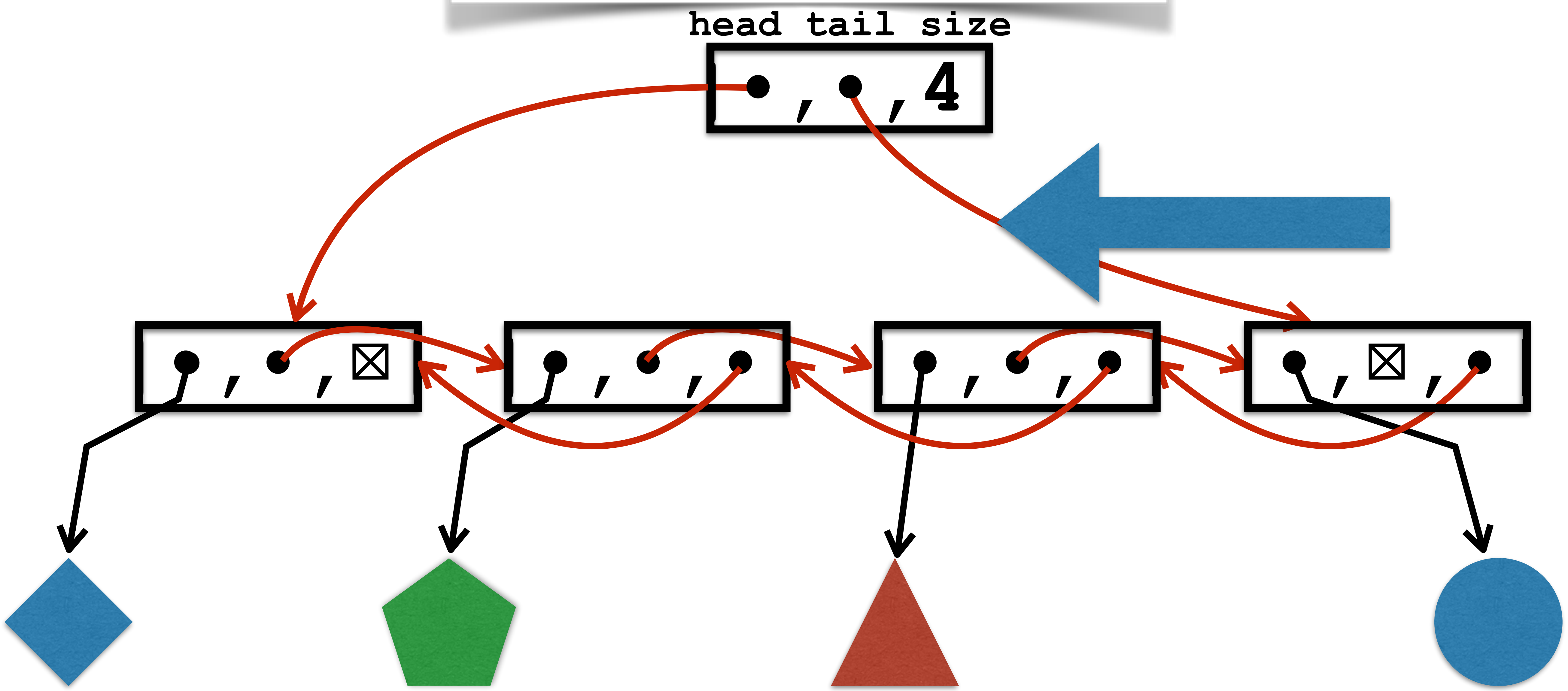
```
class  DLinkedList<E>{
    DNode<E>            head;
    DNode<E>            tail;
    int                 size;
       :
}
```



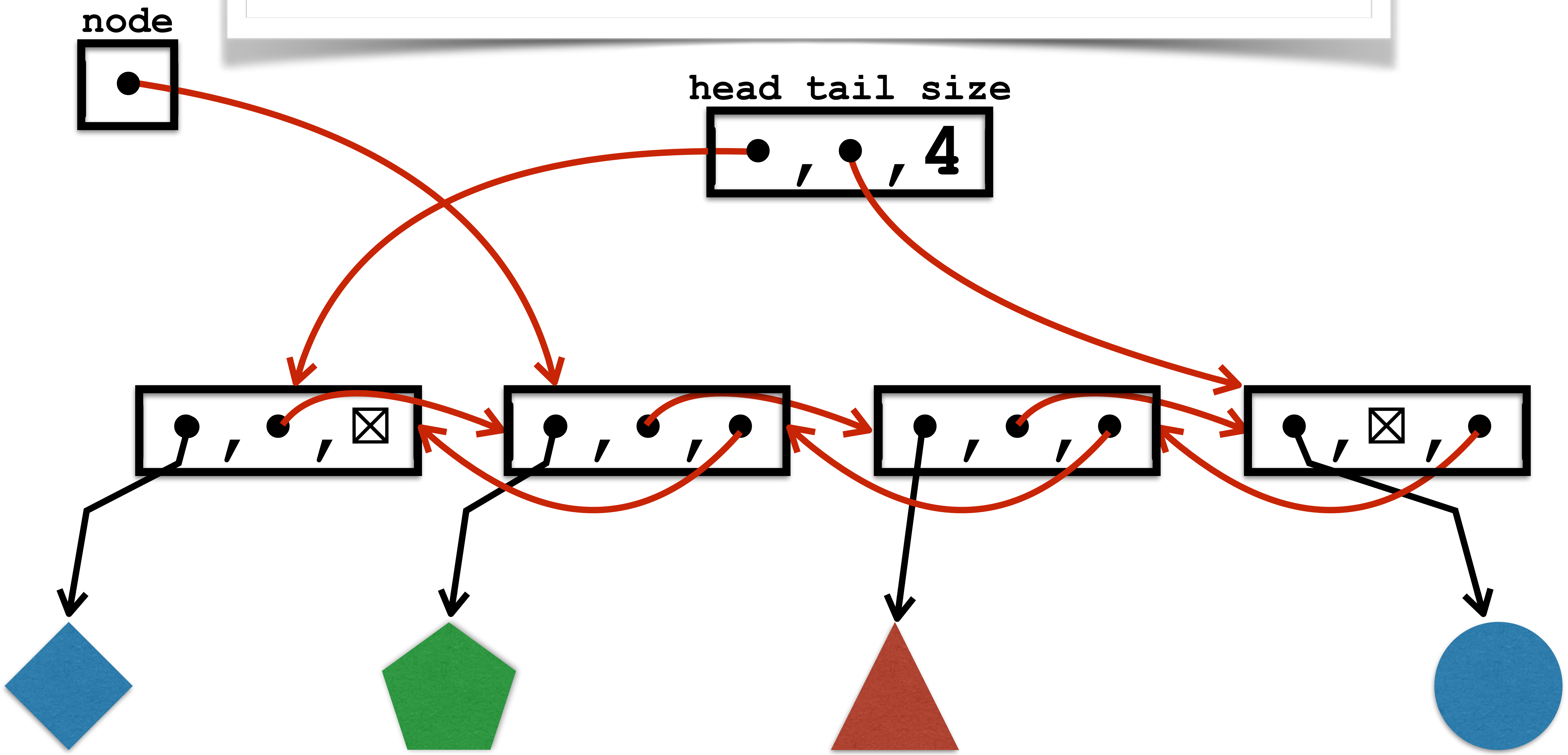
```
getNode(i){
    if (i < size/2){
        tmp = head
        index = 0
        while (index < i){
            tmp = tmp.next
            index++
        }
```



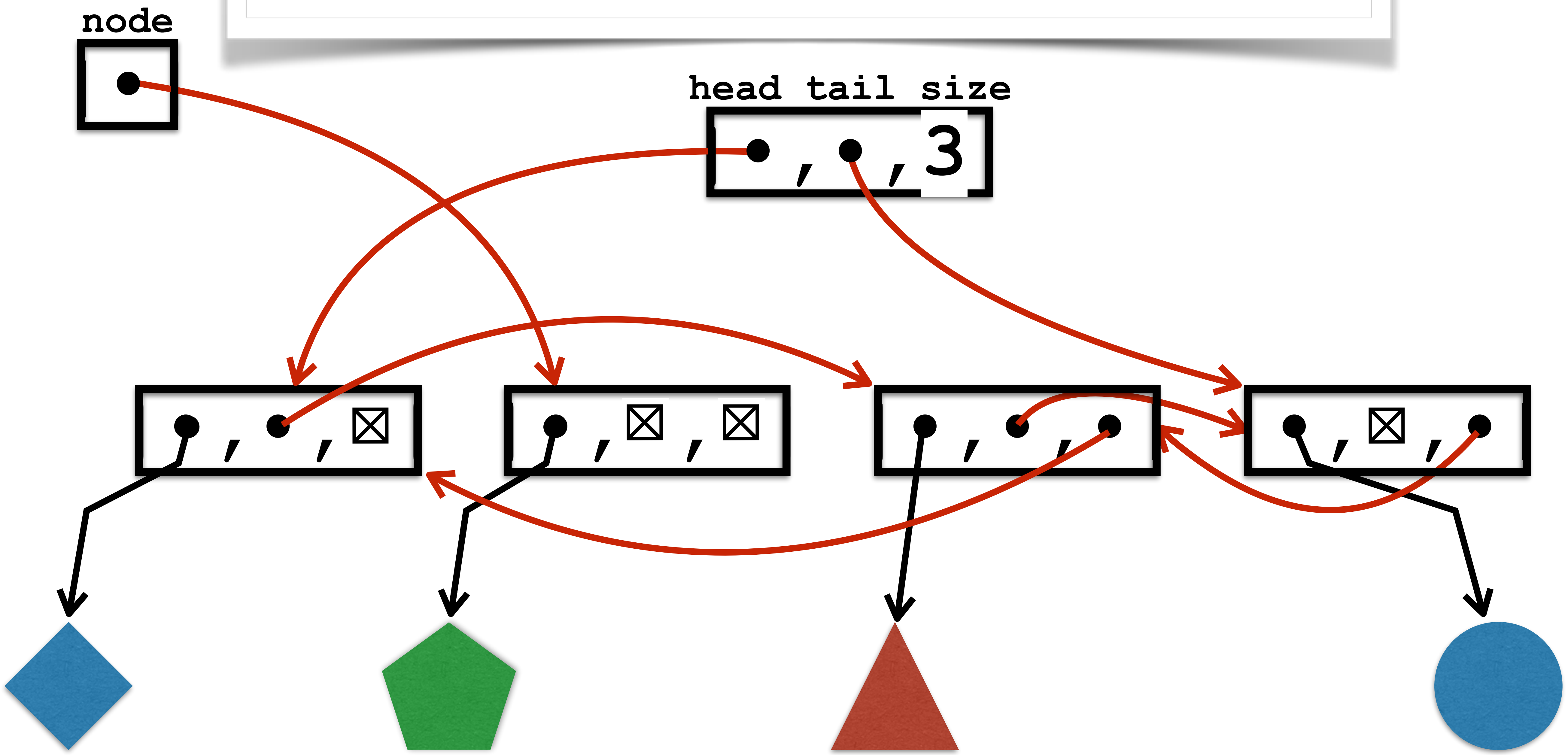
```
    else{
        tmp = tail
        index = size - 1
        while (index > i){
            tmp = tmp.prev
            index--
        }
    return tmp
```



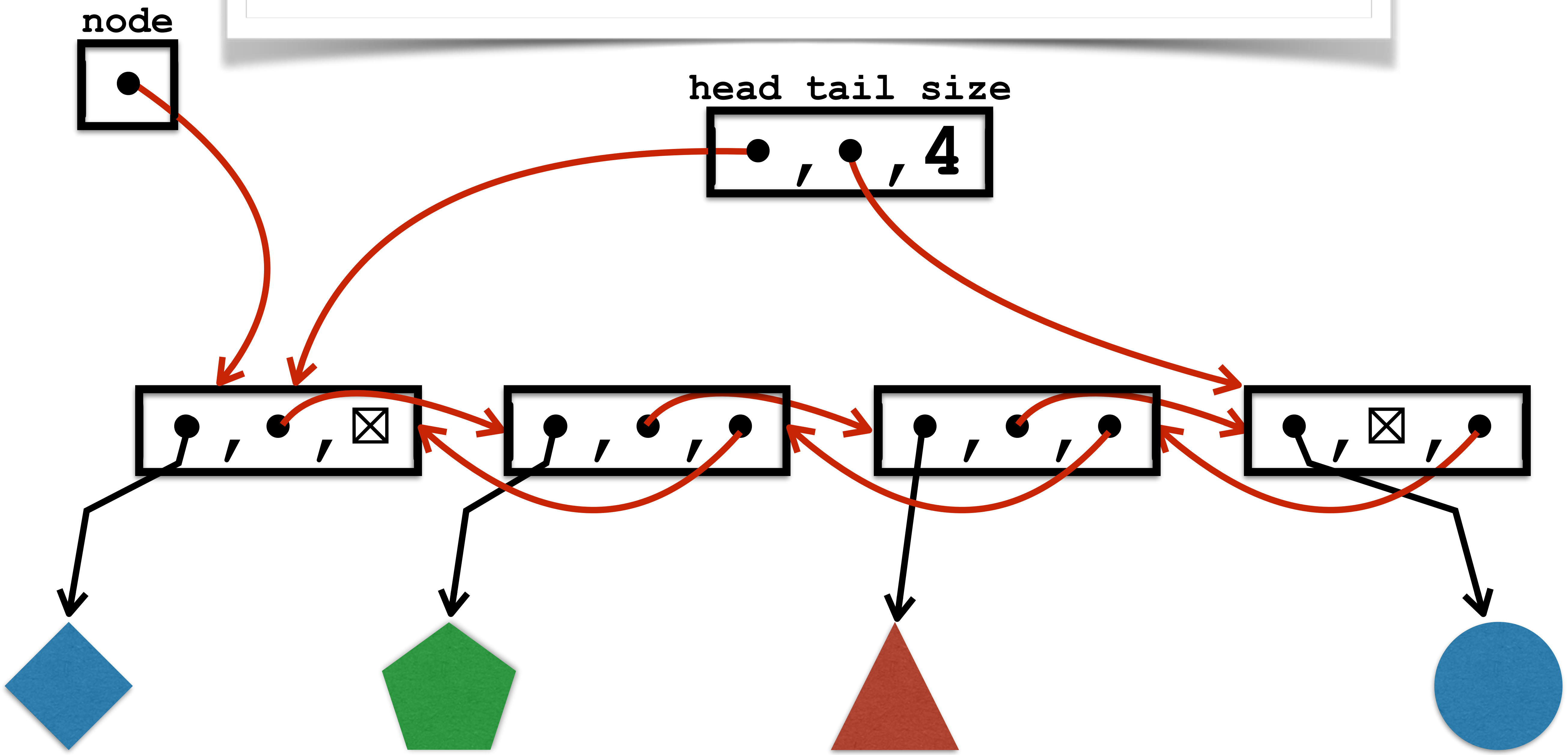
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



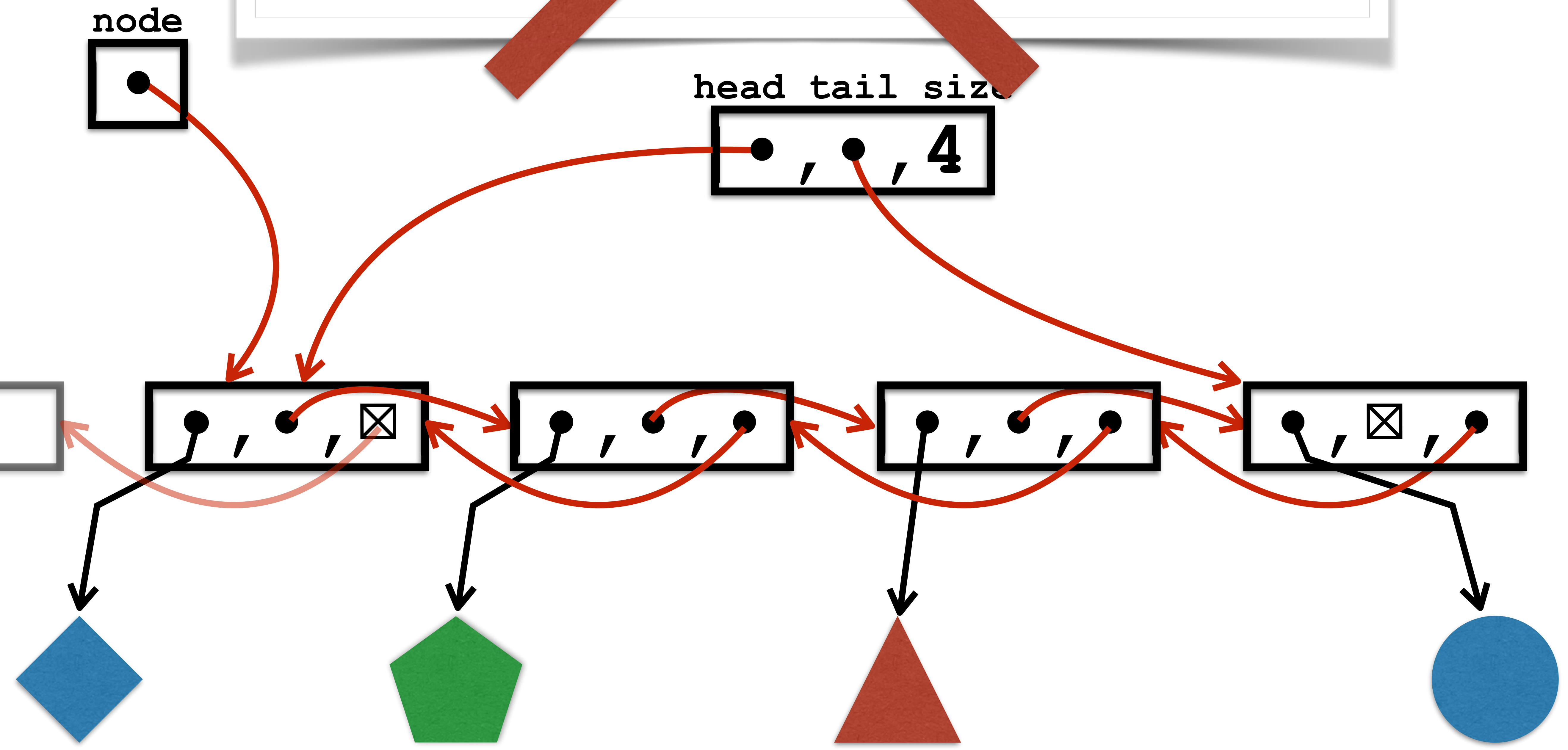
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



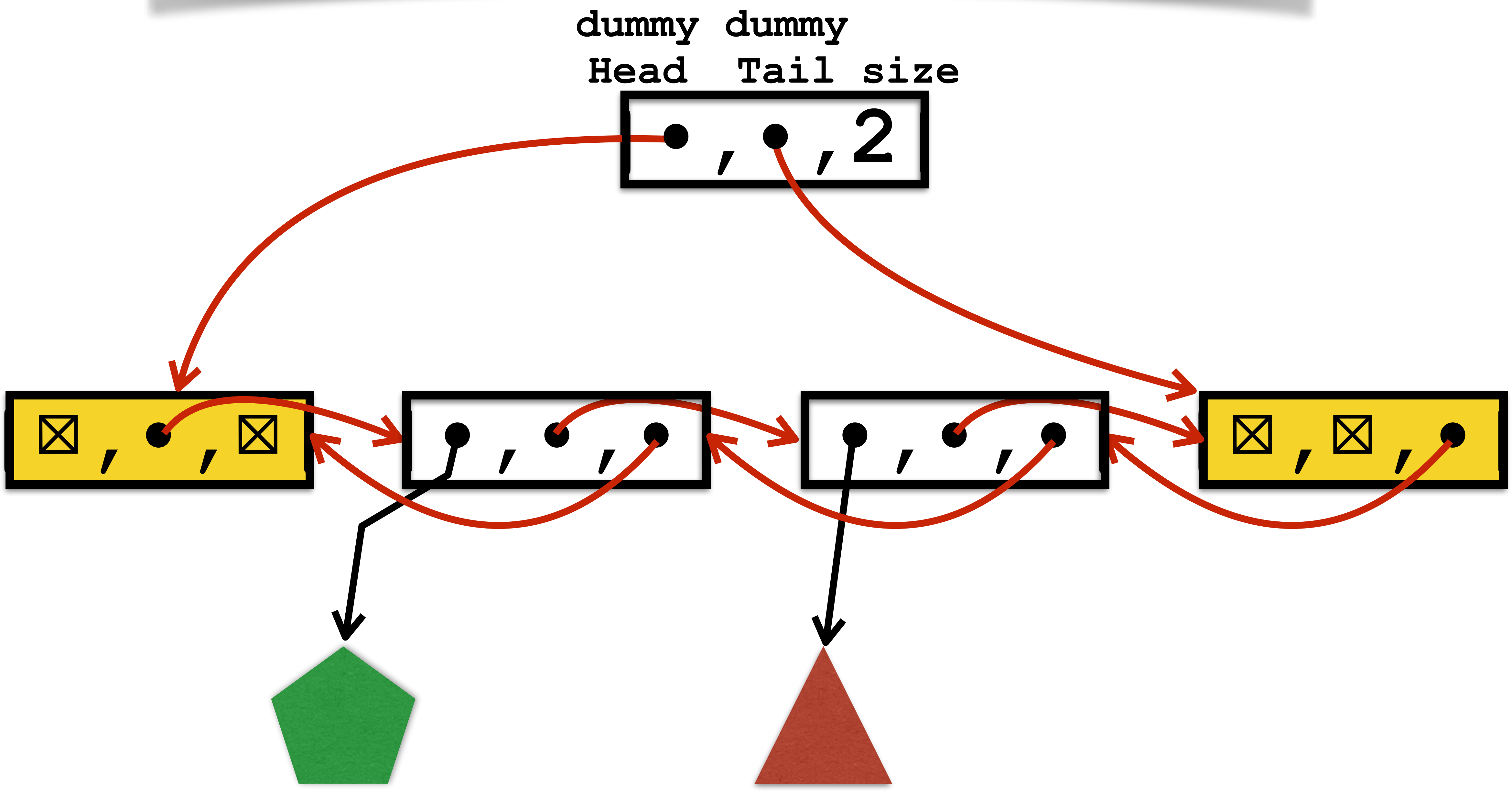
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



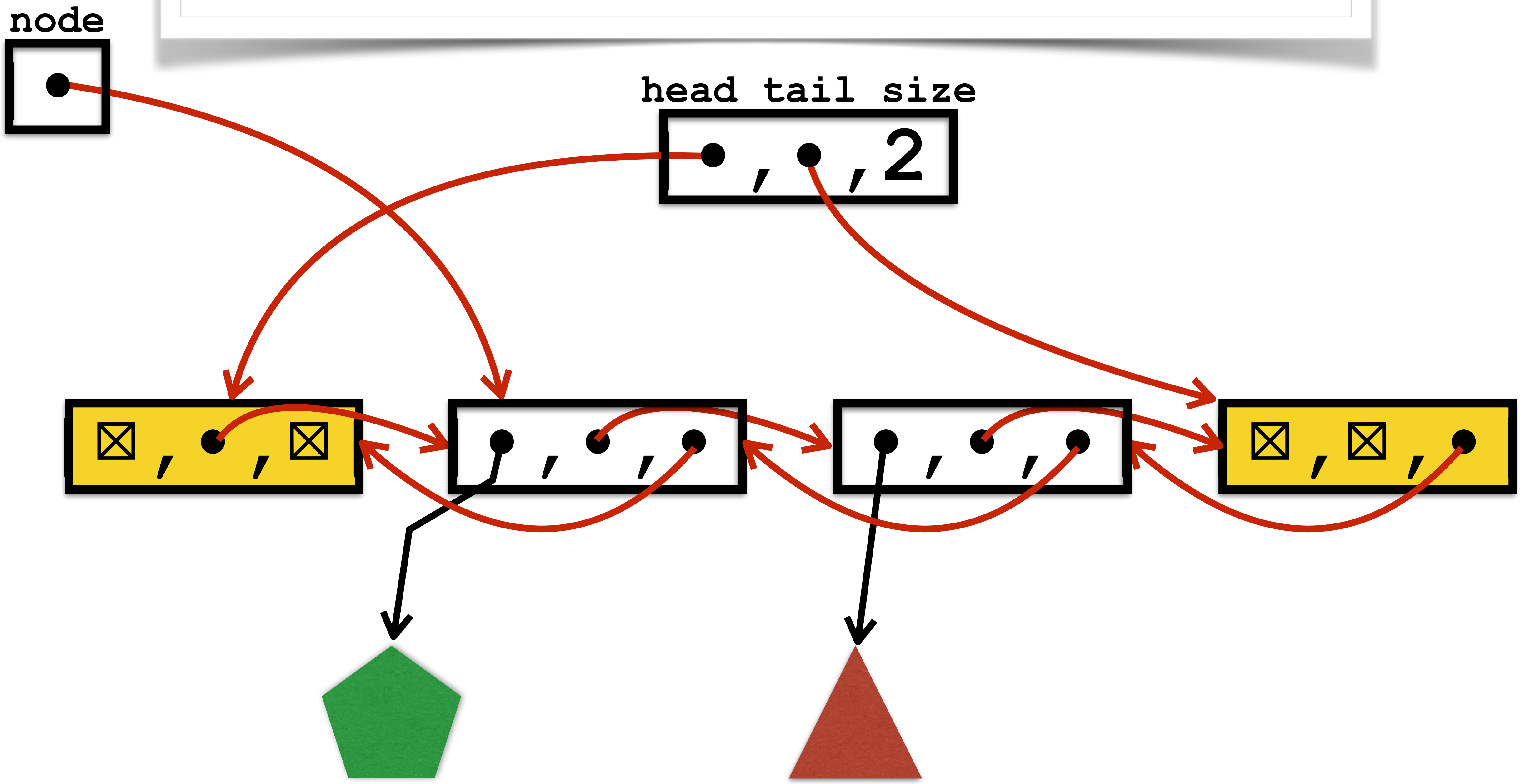
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



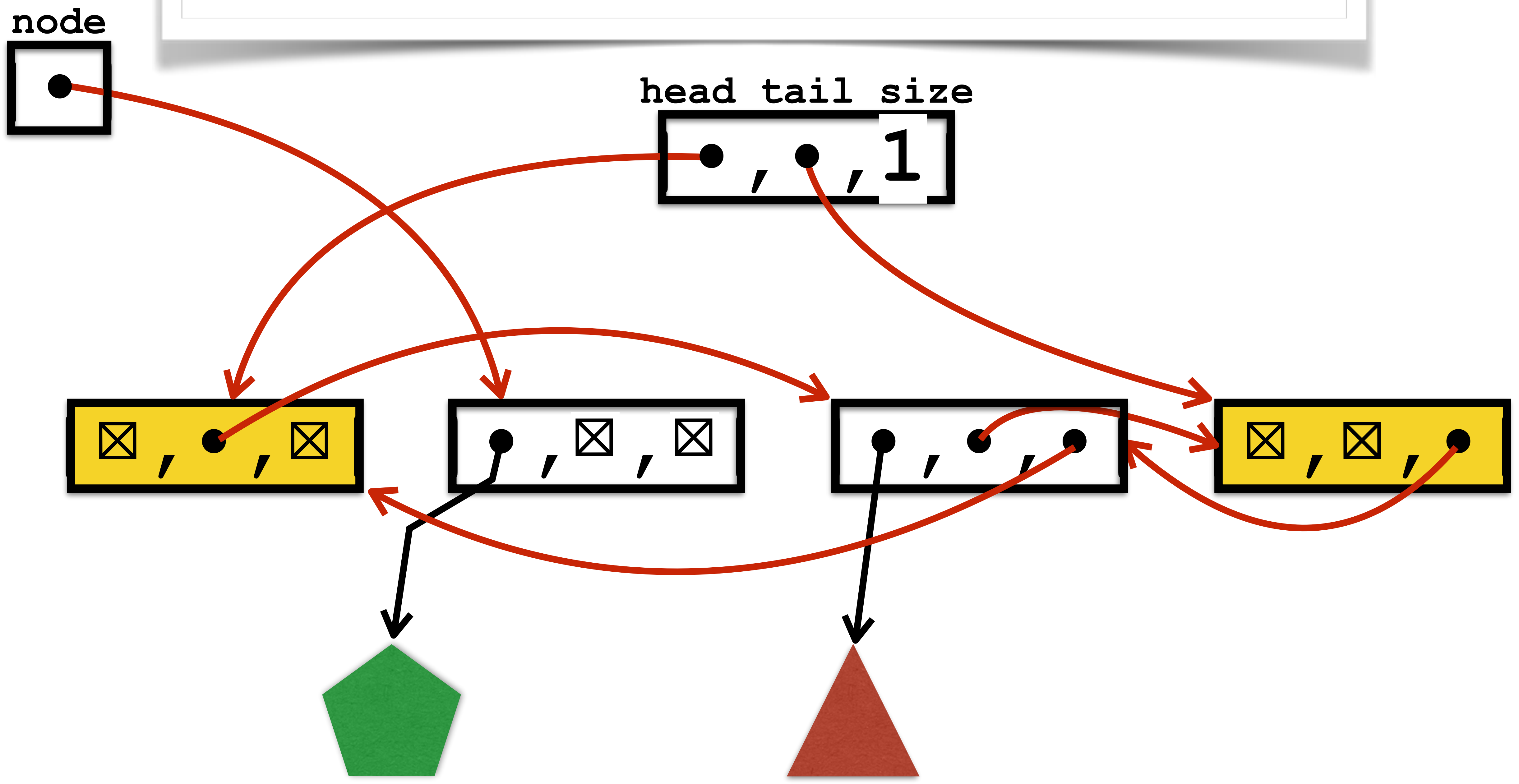
```
class  DLinkedList<E>{
   DNode<E>            dummyHead;
   DNode<E>            dummyTail;
   int                 size;
   }
```



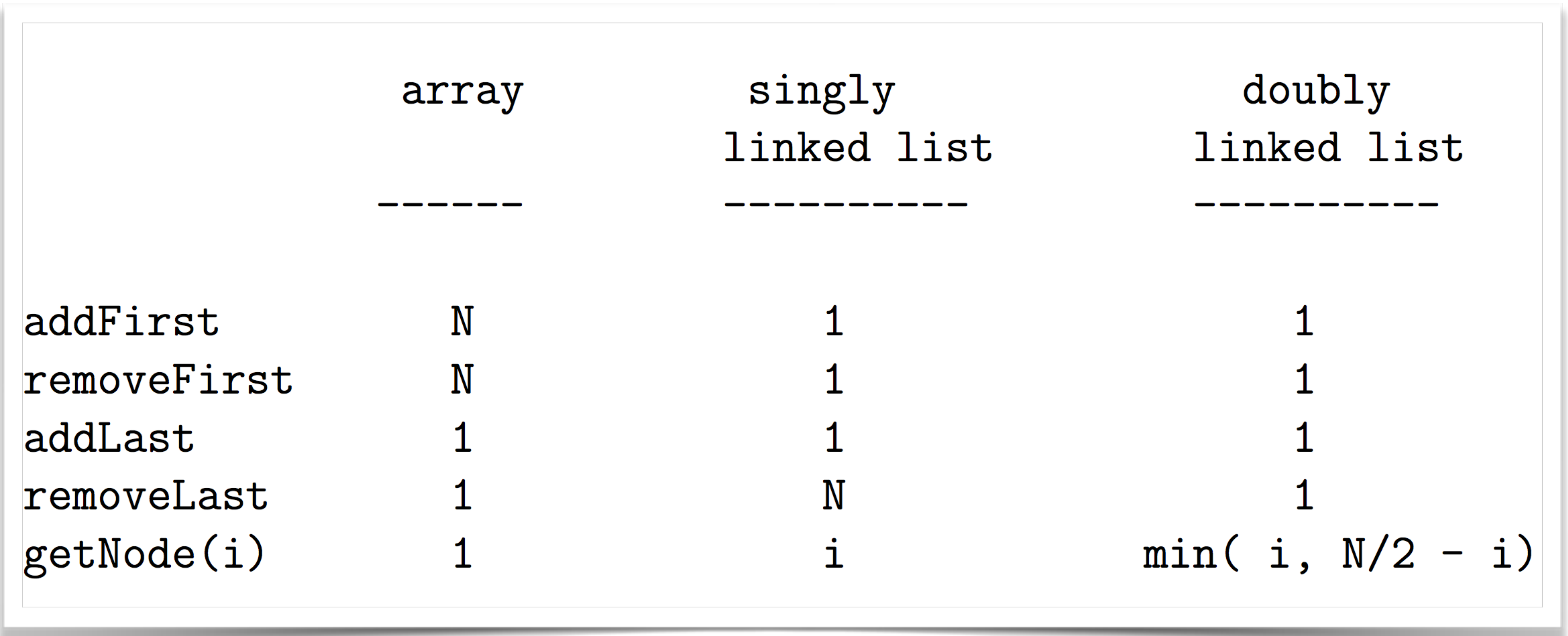
```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



```
remove( node ){
    node.prev.next = node.next;
    node.next.prev = node.prev;
    size           = size-1;
}
```



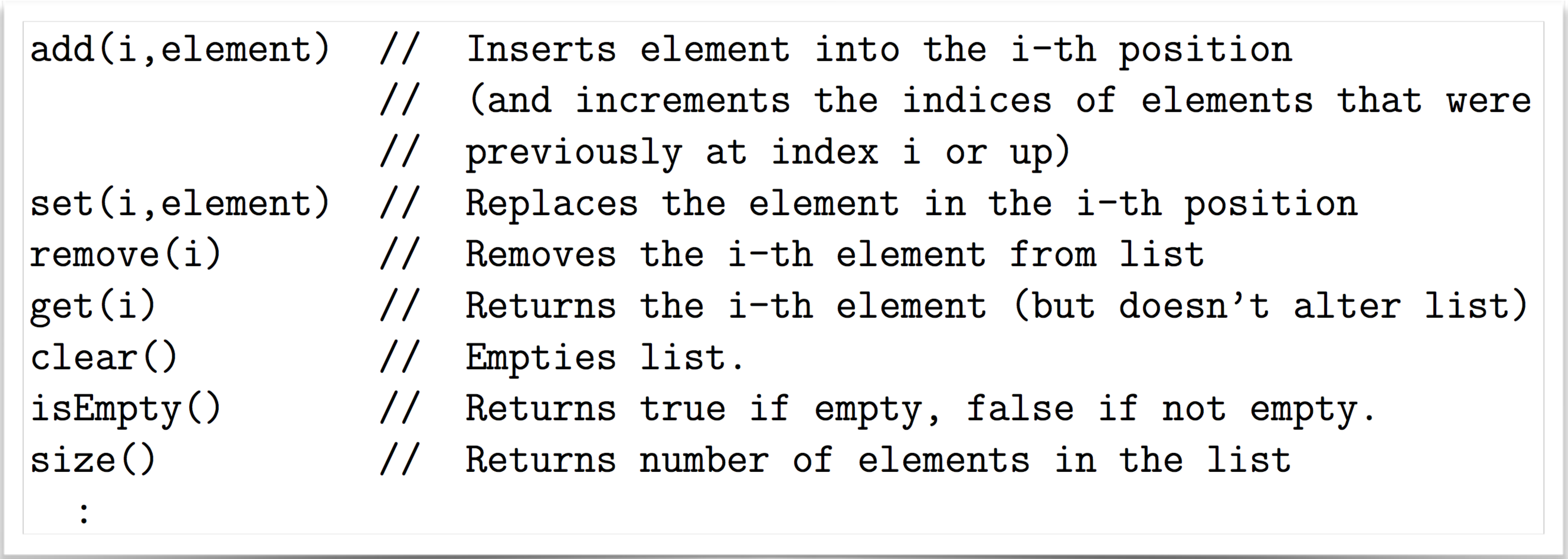
Array vs Linked List



	array	singly linked list	doubly linked list
addFirst	N	1	1
removeFirst	N	1	1
addLast	1	1	1
removeLast	1	N	1
getNode(i)	1	i	min(i, N/2 - i)

Linked Lists as ADT (Abstract Data Type)

Linked List operations

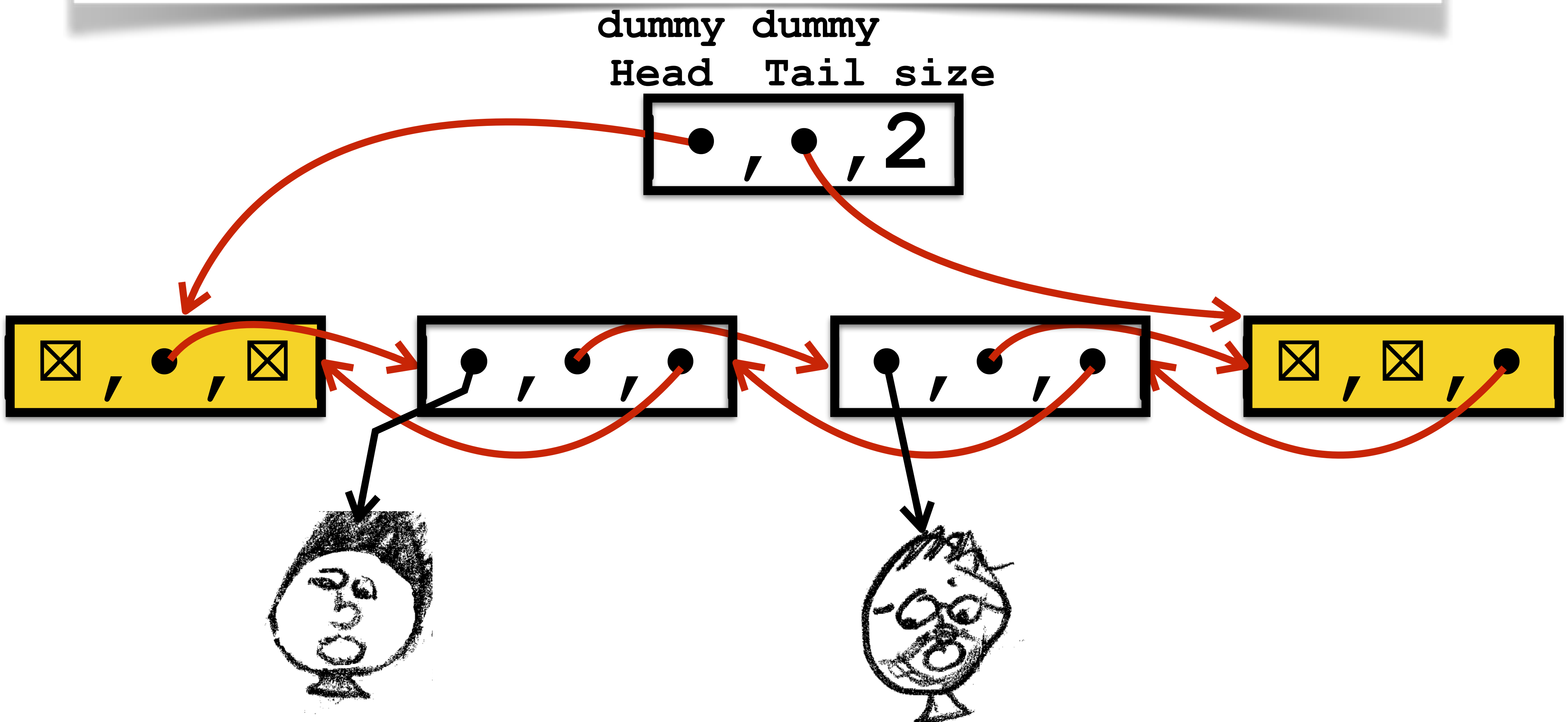


```
add(i,element)  //  Inserts element into the i-th position
                //  (and increments the indices of elements that were
                //  previously at index i or up)
set(i,element)  //  Replaces the element in the i-th position
remove(i)       //  Removes the i-th element from list
get(i)          //  Returns the i-th element (but doesn't alter list)
clear()         //  Empties list.
isEmpty()       //  Returns true if empty, false if not empty.
size()          //  Returns number of elements in the list
  :
```

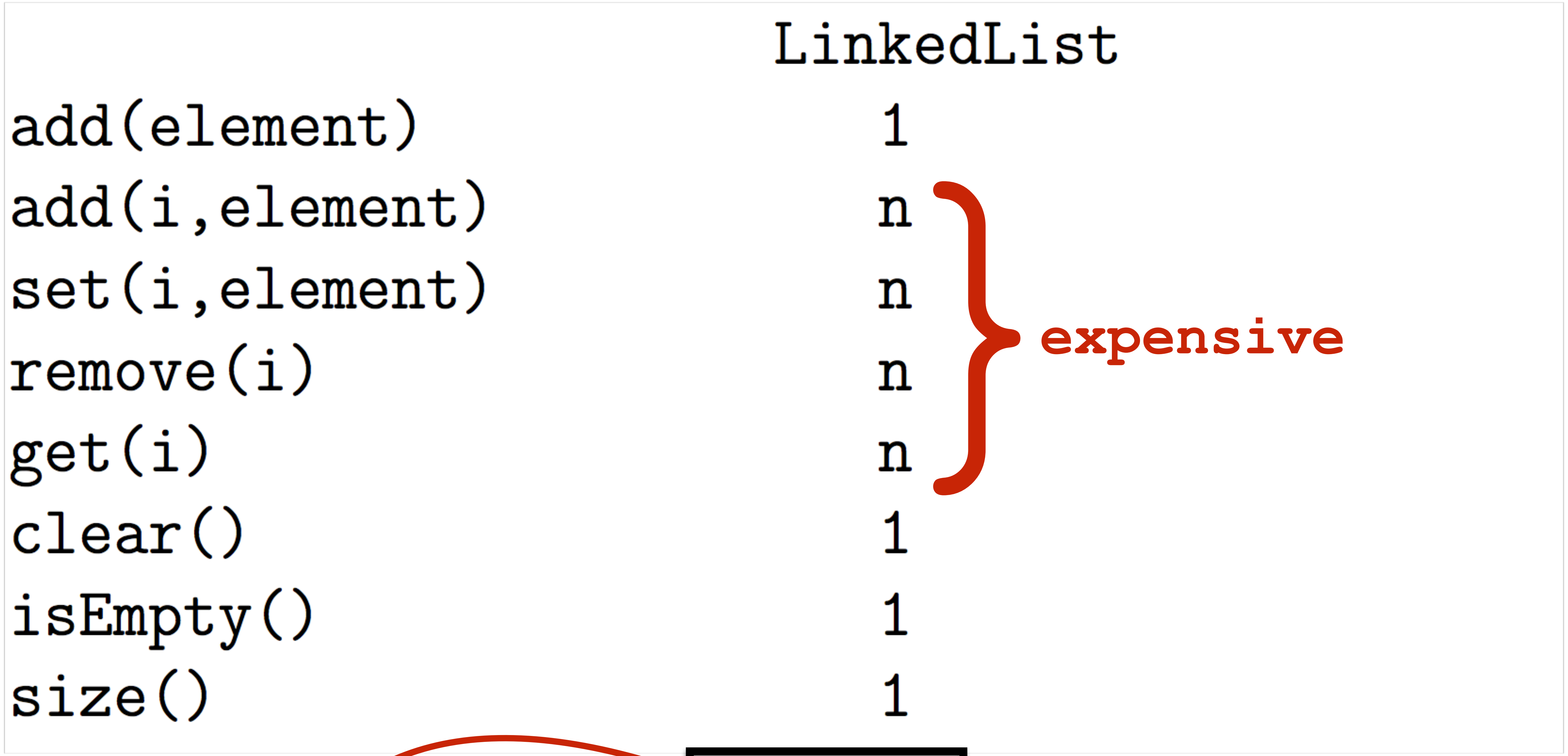
```
LinkedList<Student>
studentList  =  new  LinkedList<Student>();
```

Java LinkedList

- implemented as doubly linked list (with dummies)
- Node class is private

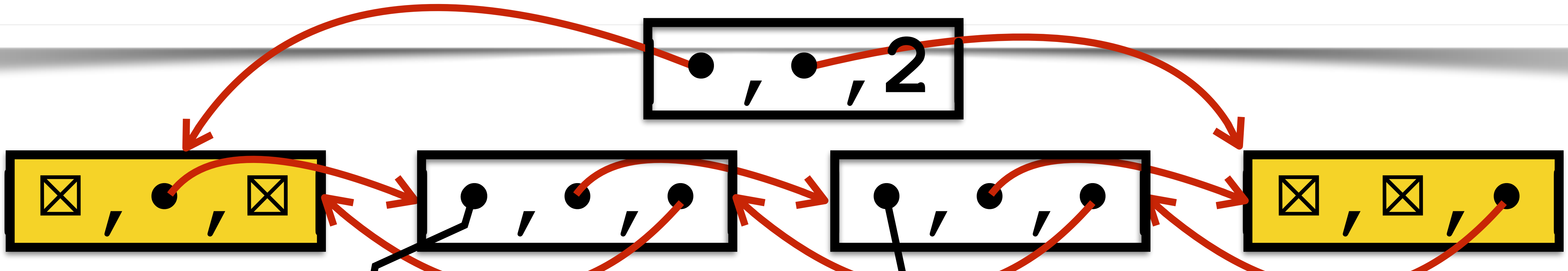


Java LinkedList



	LinkedList
add(element)	1
add(i,element)	n
set(i,element)	n
remove(i)	n
get(i)	n
clear()	1
isEmpty()	1
size()	1

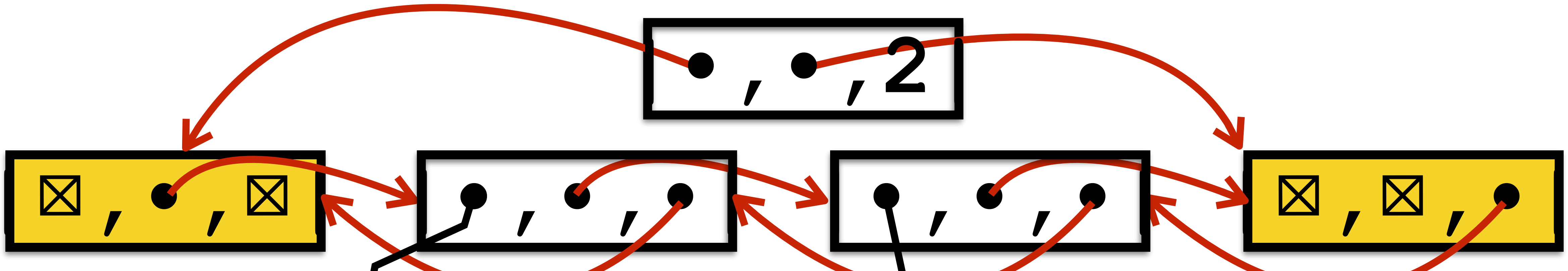
expensive



Java LinkedList

```
for (j = 1; j < n;   j++)
    print( studentList.get(j) )
```

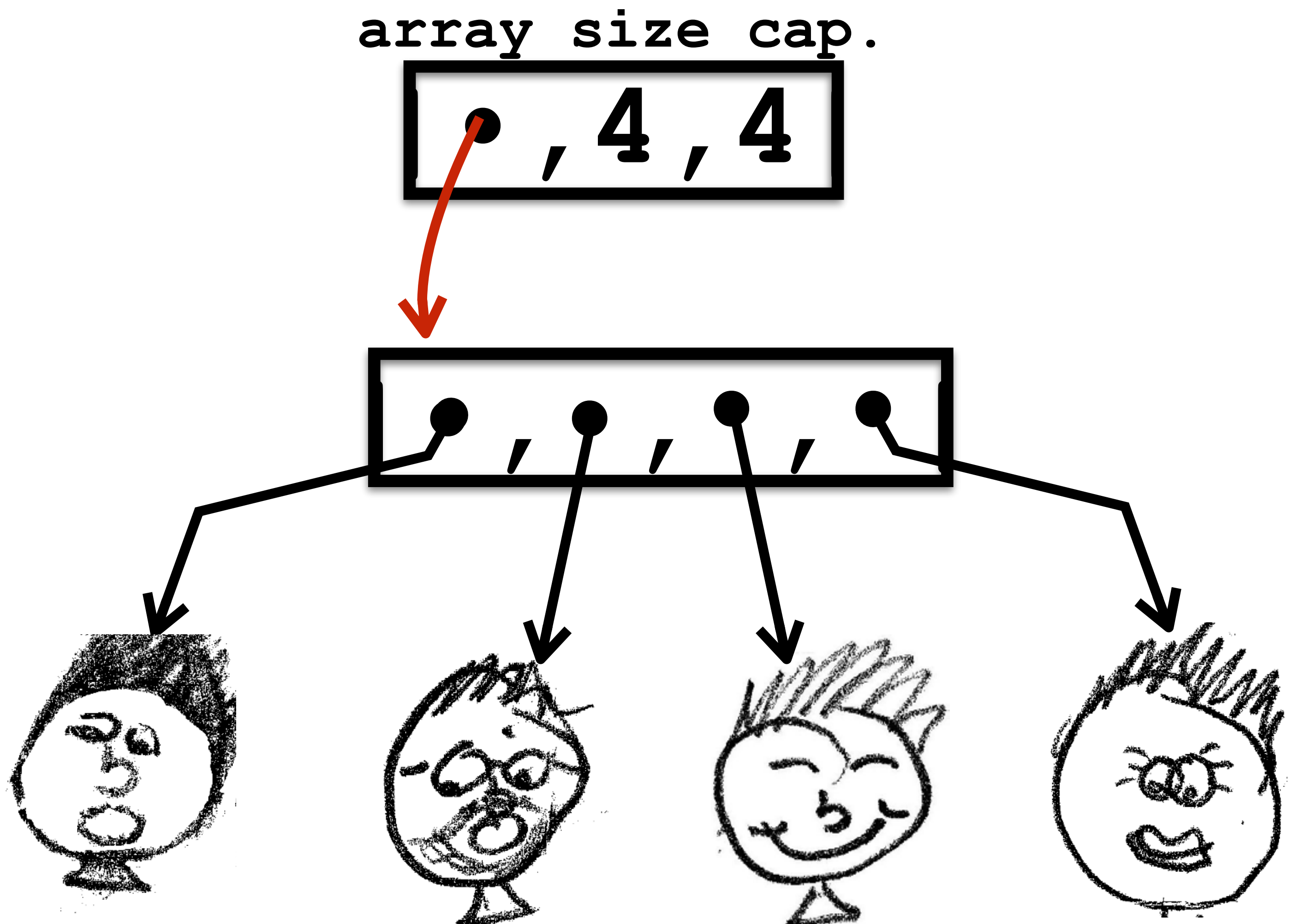
Time(n) **is** $\Omega(n^2)$



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

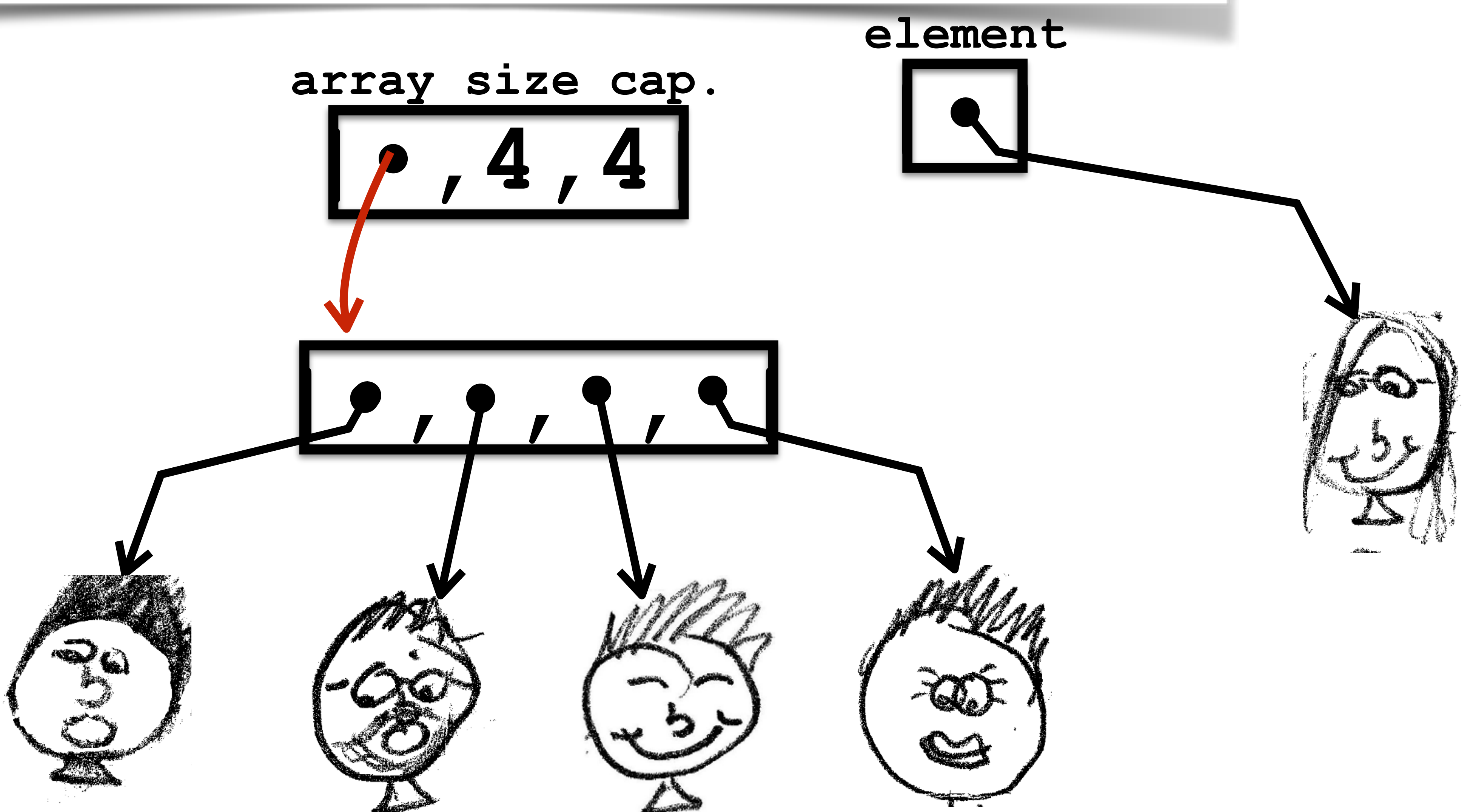
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

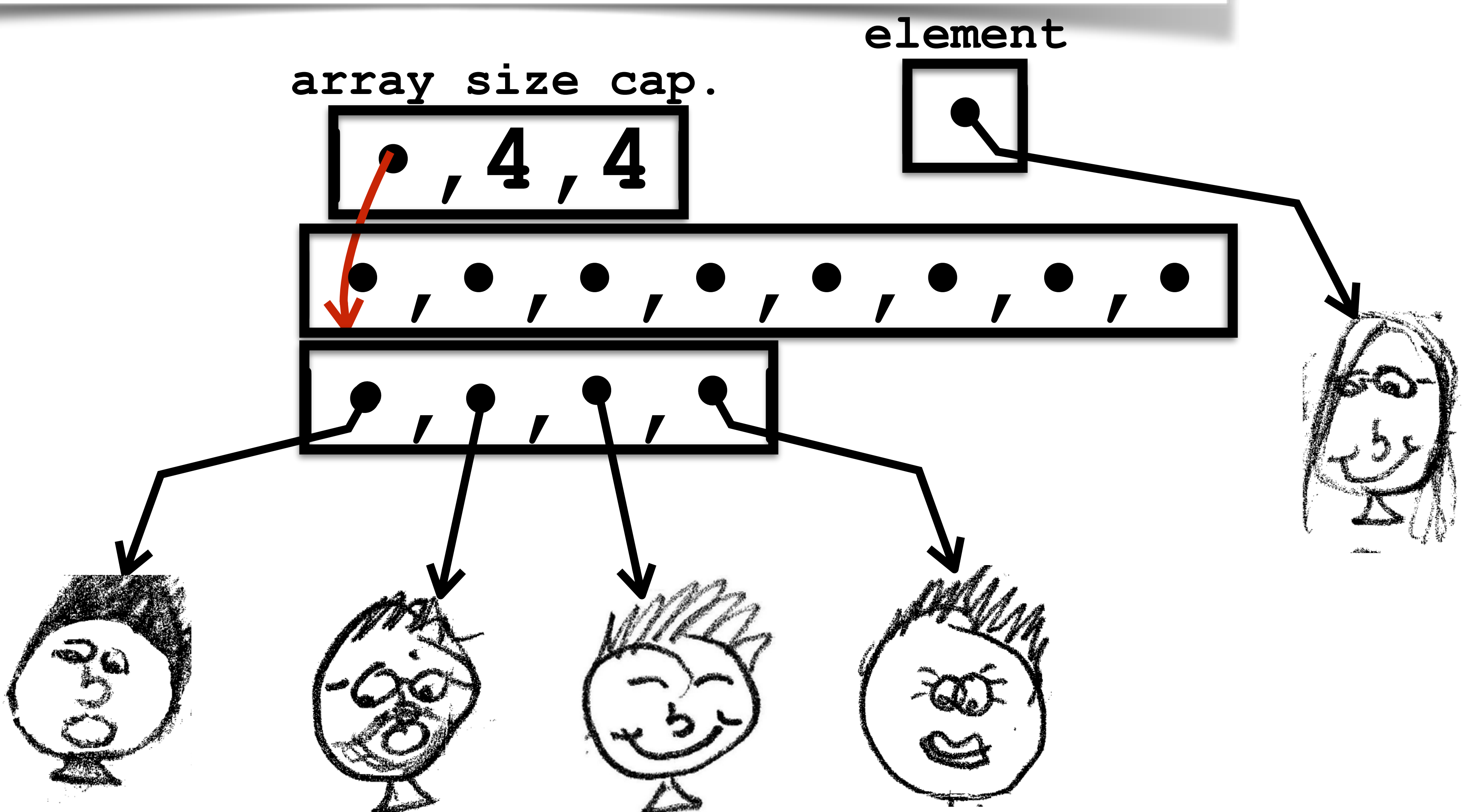
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- when new space is needed - doubles the array size.

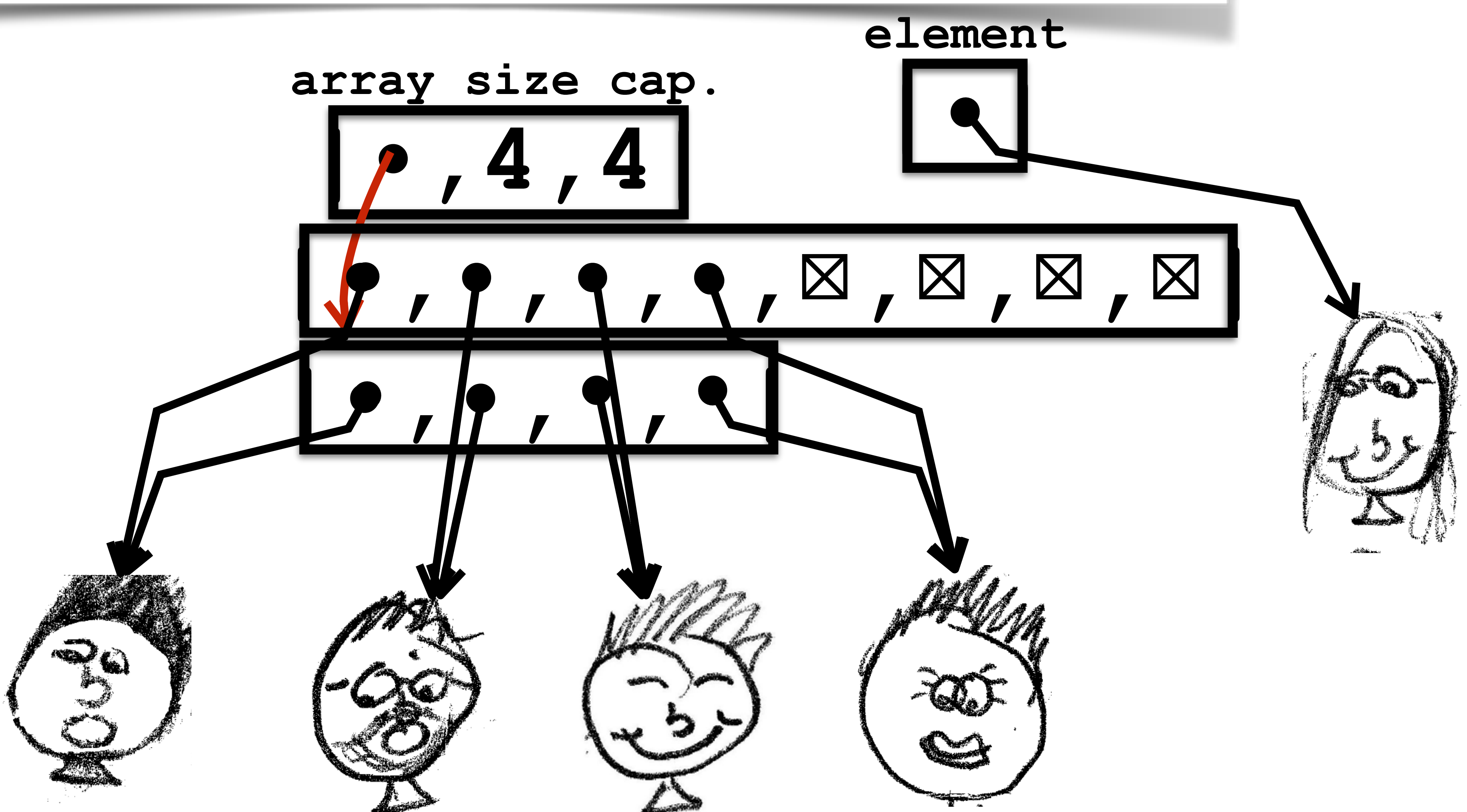
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- smaller array is copied into bigger array.

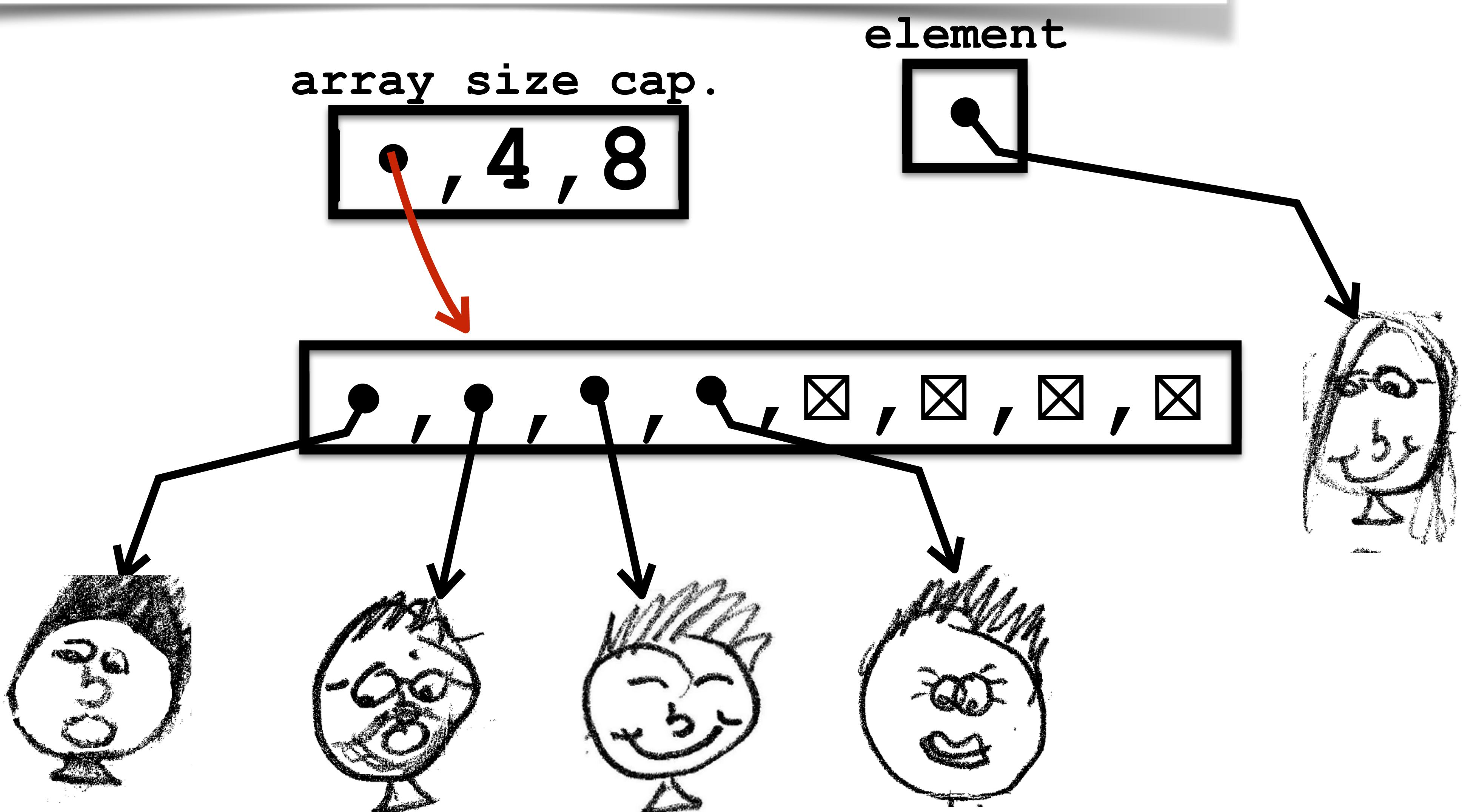
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- new element added to new array

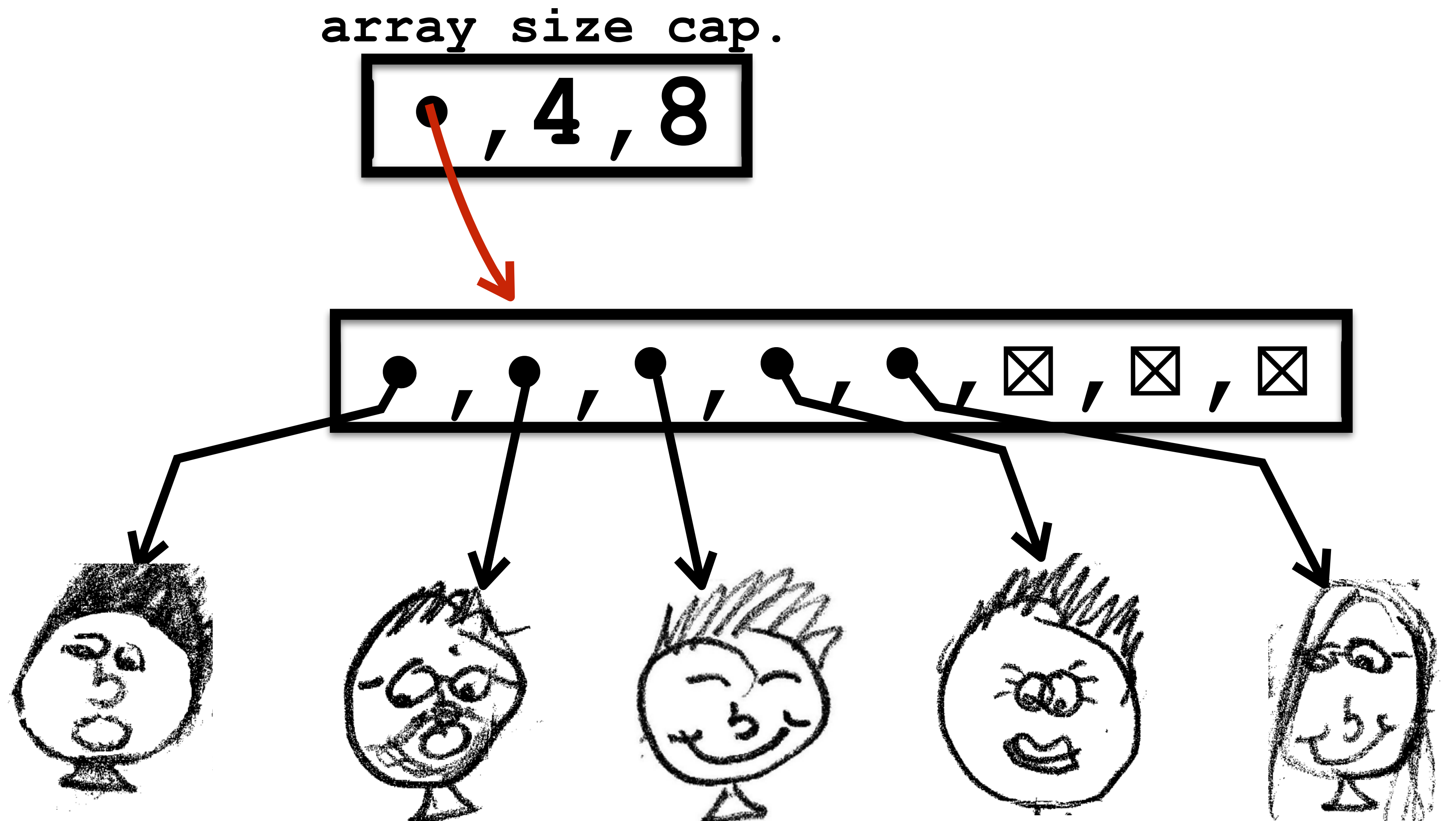
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of growing sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



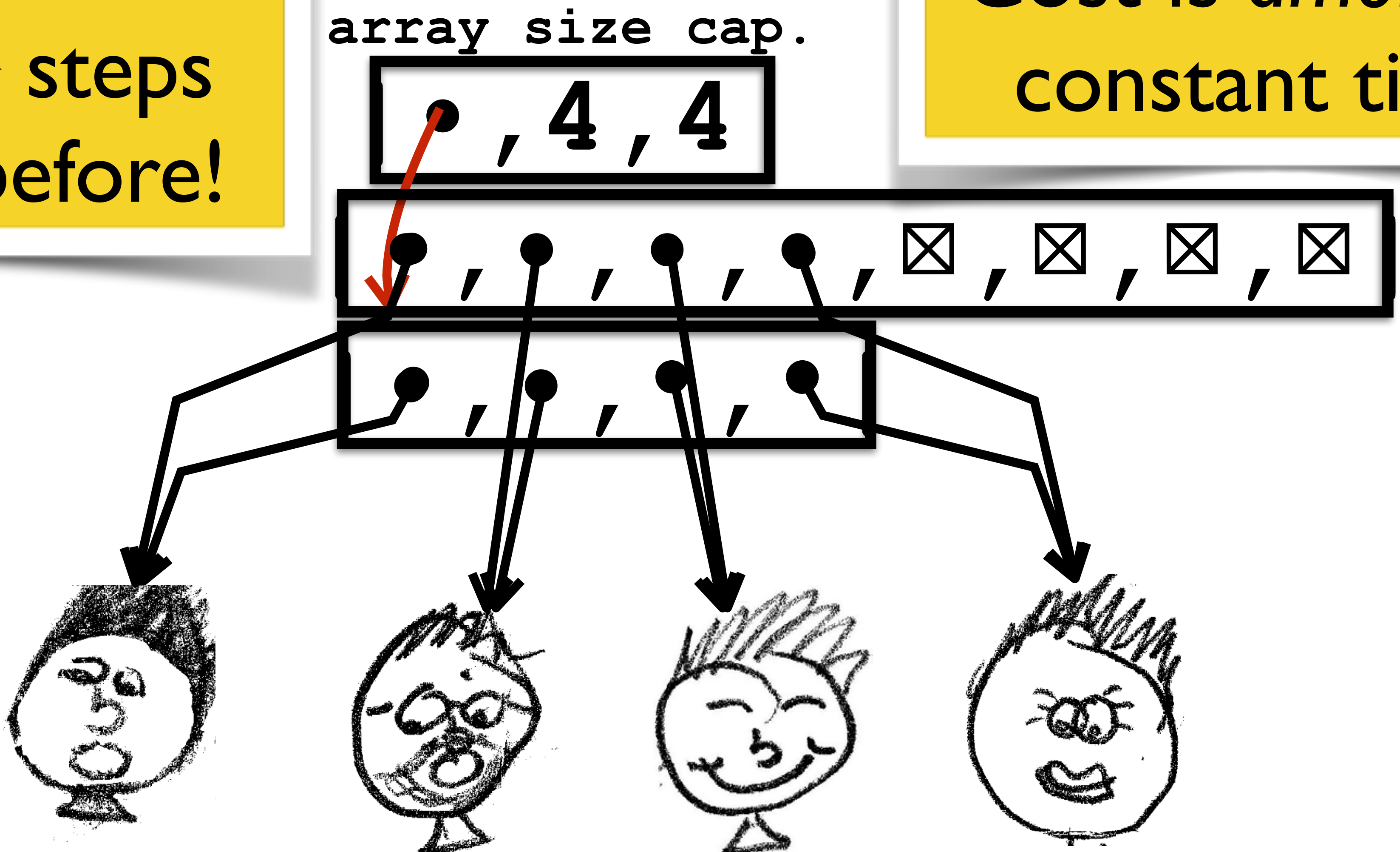
Cost ?

Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

Extra cost of 2^{k+1} new steps only if 2^k steps already spent before!

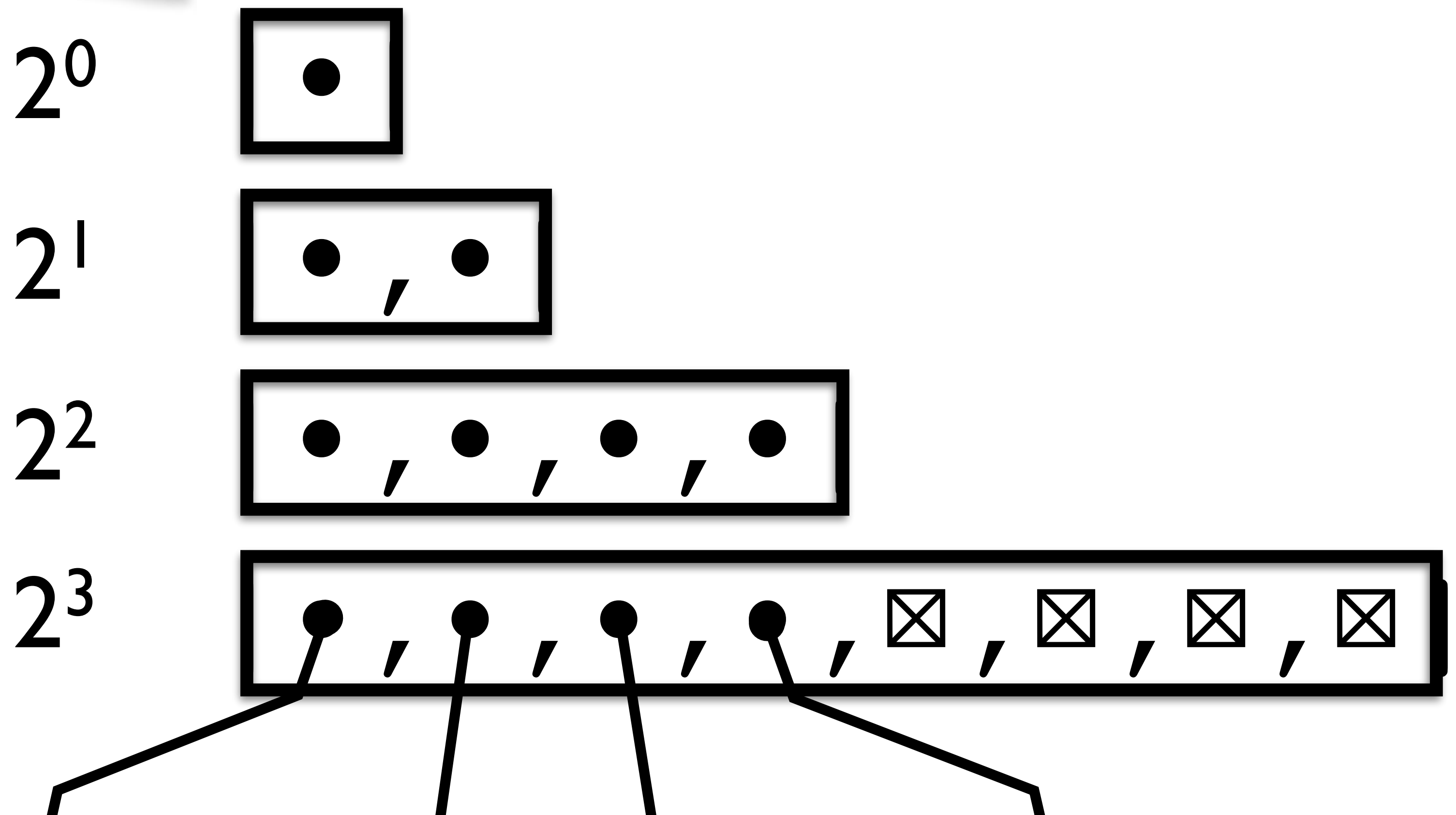
Cost is *amortized* constant time!



Java ArrayList

To build an array of 2^k elements you need $2^{k+1}-1$ steps!

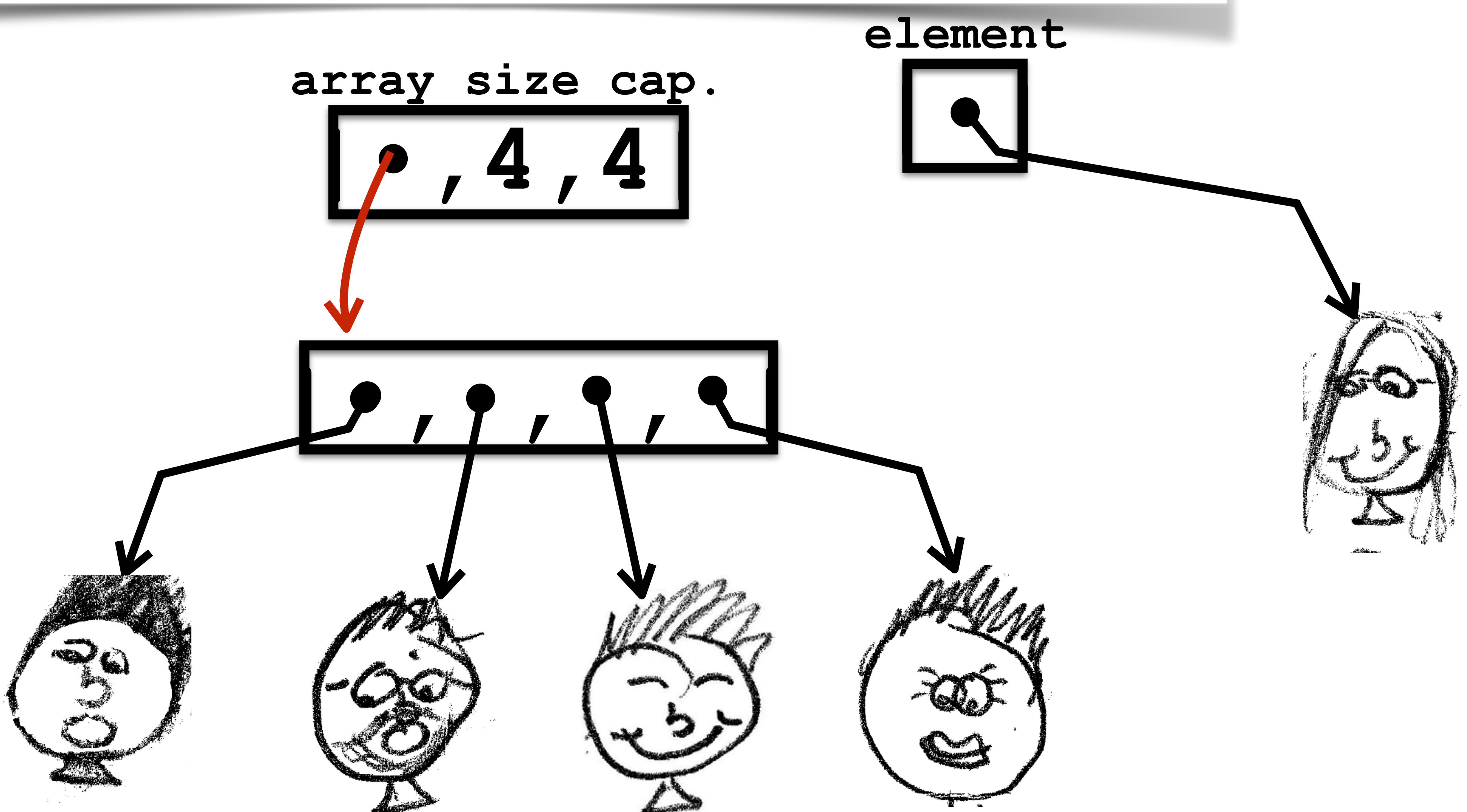
Cost is *amortized* constant time!



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

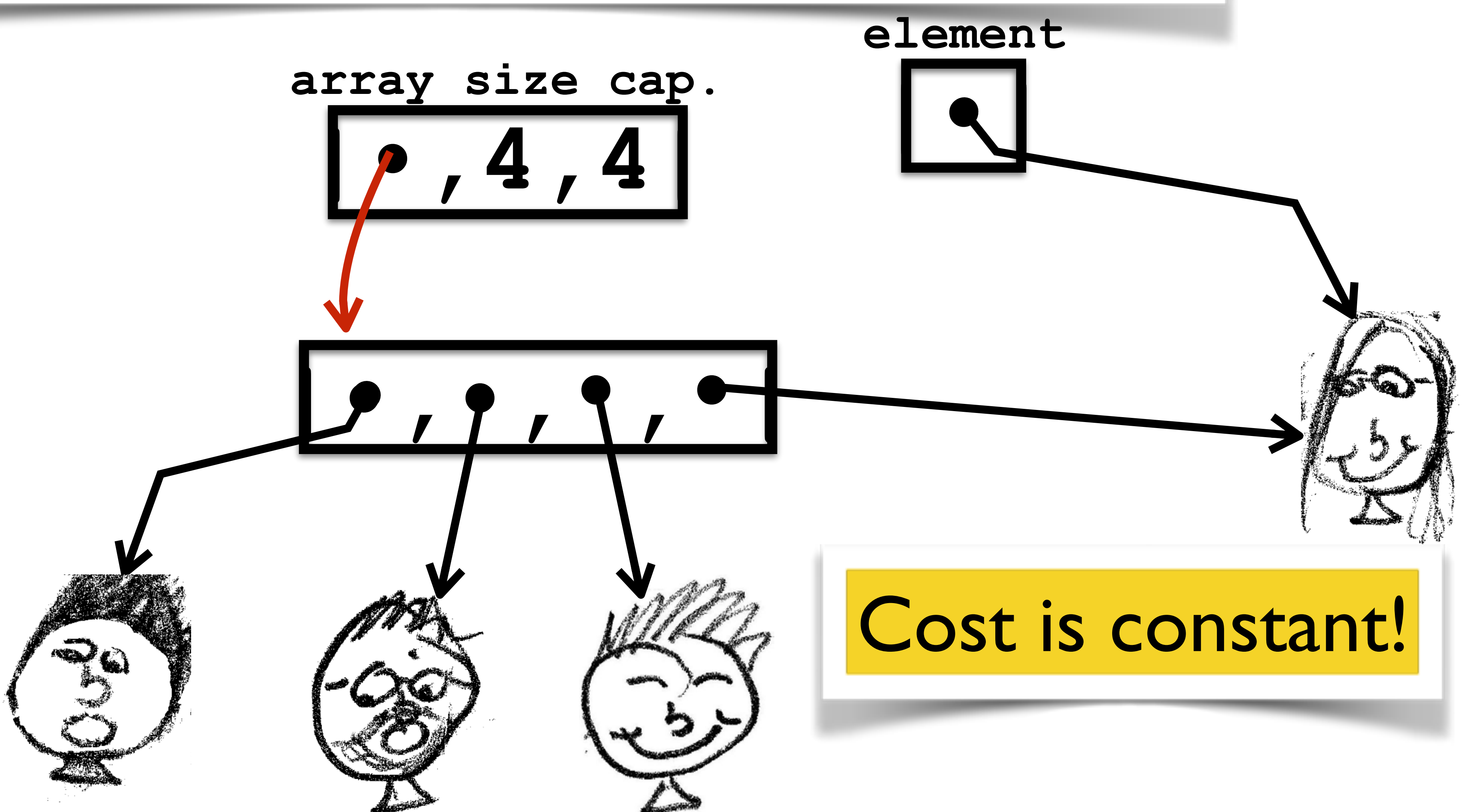
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```

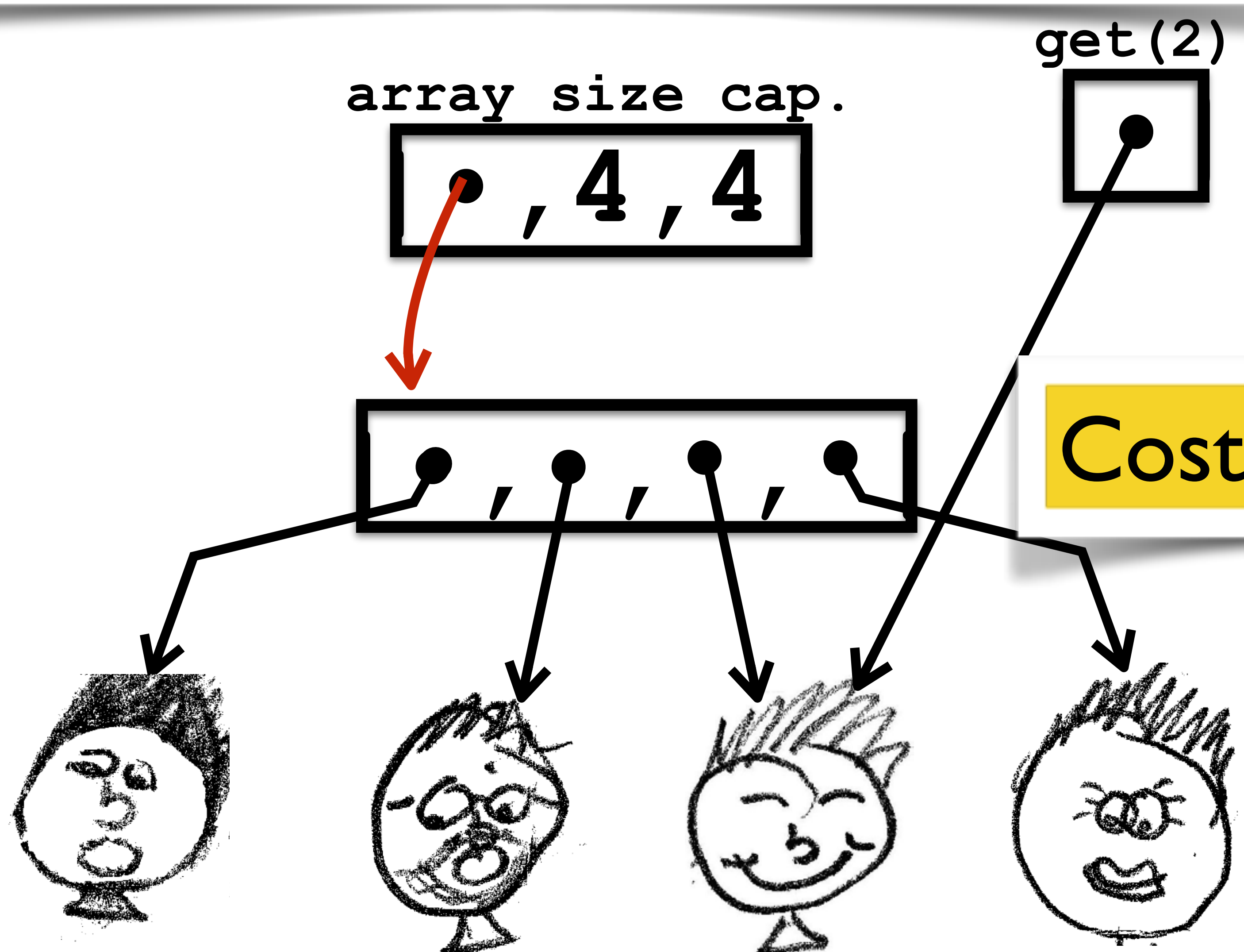


Cost is constant!

Java ArrayList

- implementation using arrays of growing sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```

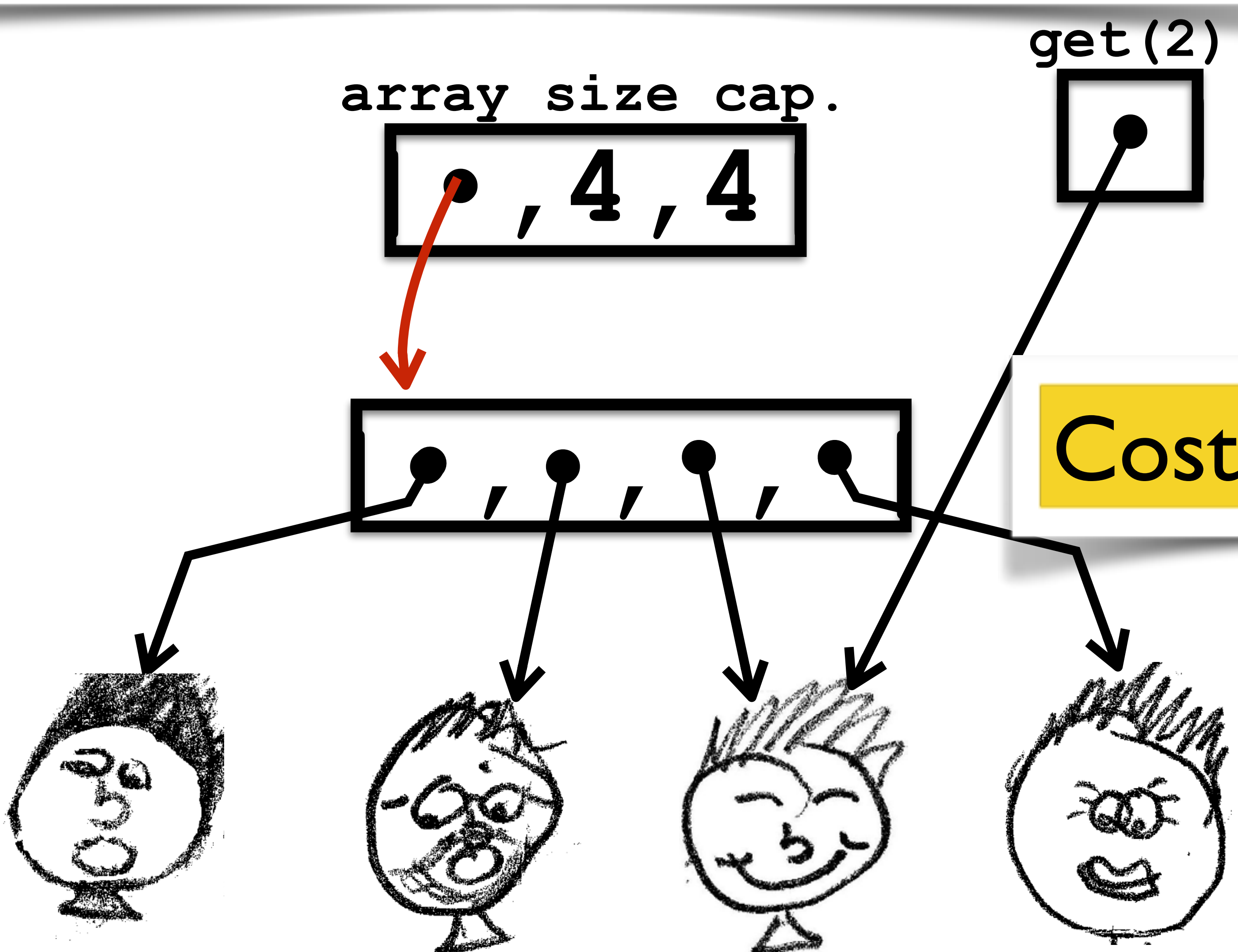


Cost is constant!

Java ArrayList

- implementation using arrays of growing sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```

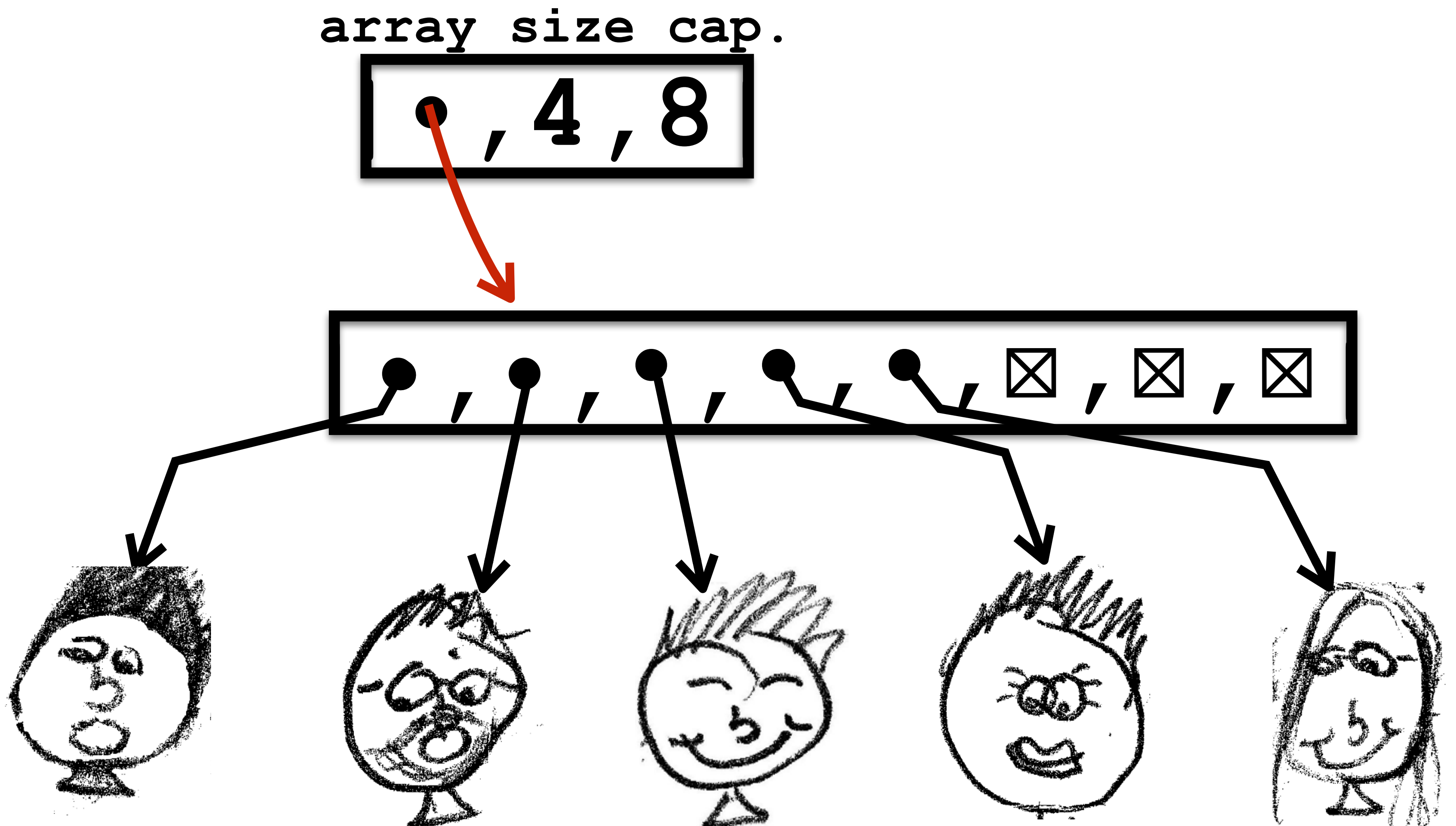


Cost is constant!

Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

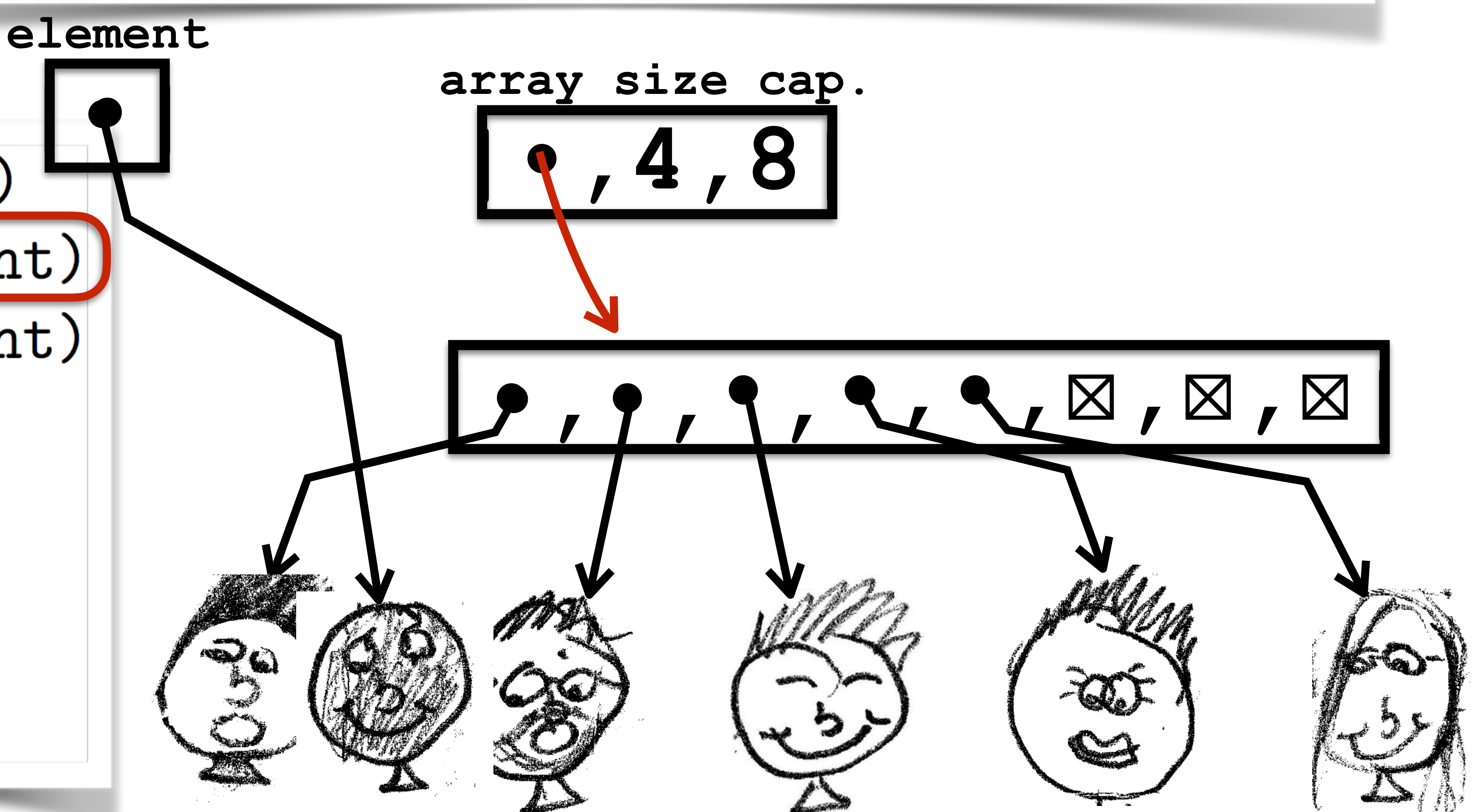
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

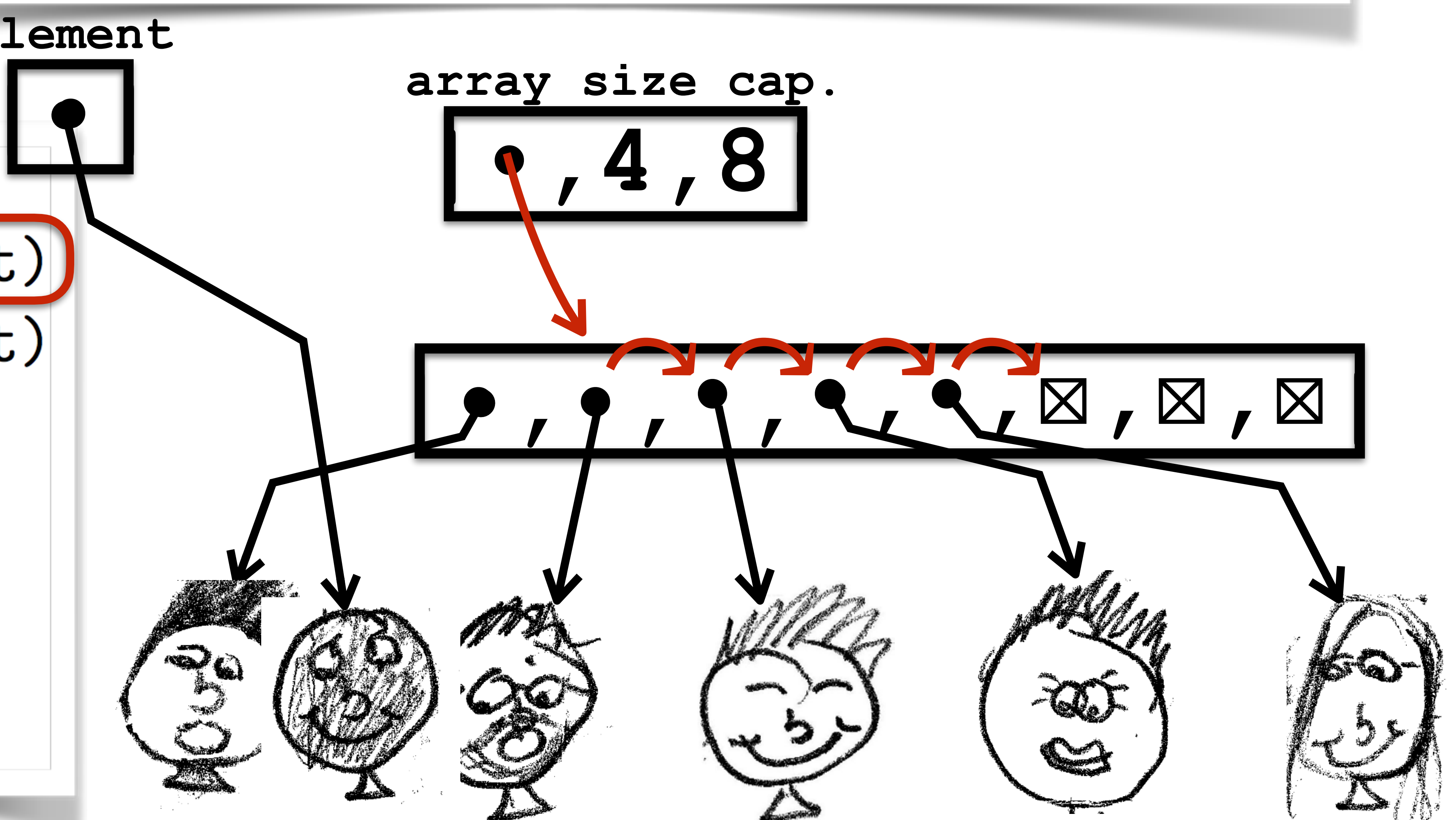
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of growing sizes
- cannot access using a[i] notation

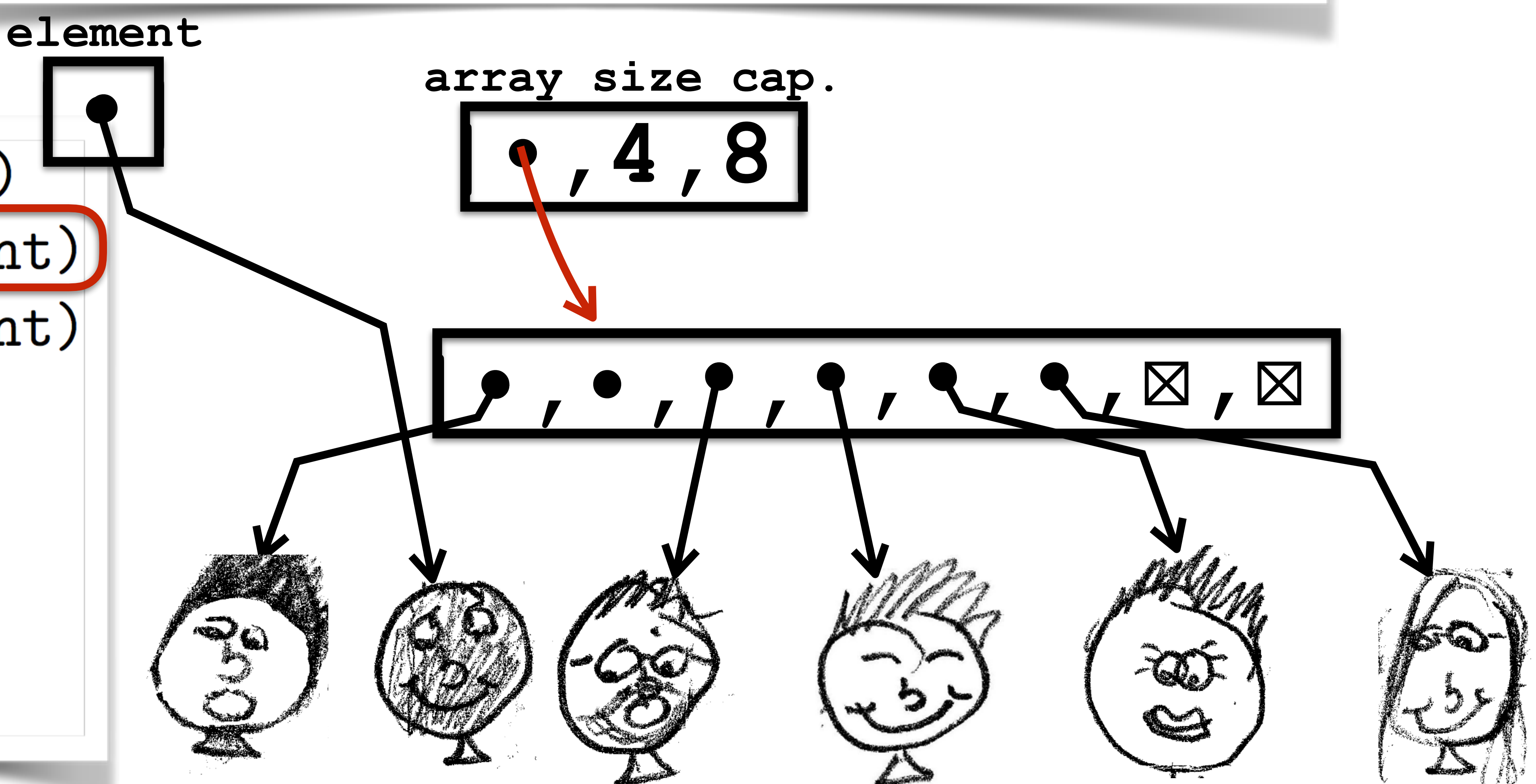
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

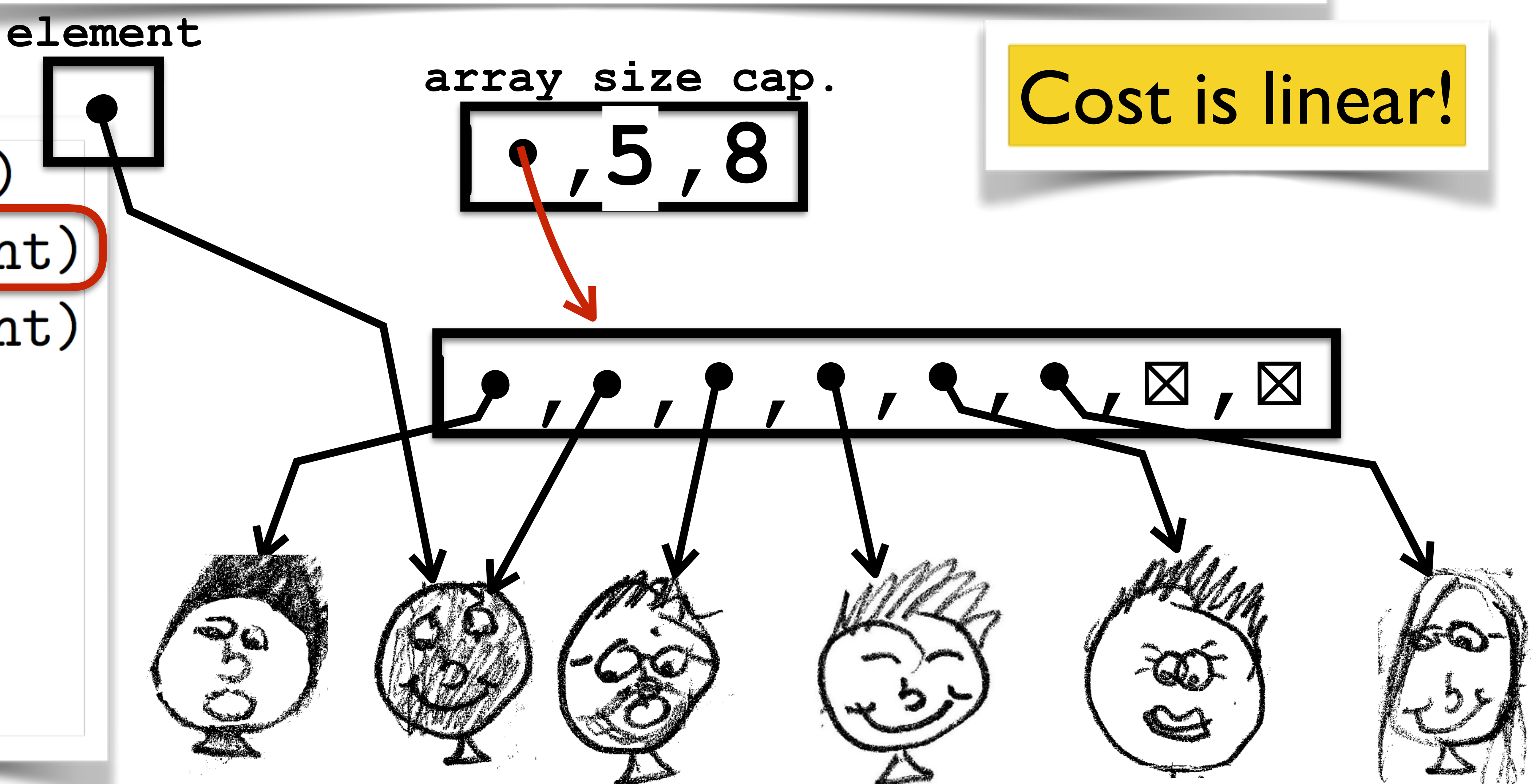
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Java ArrayList

- implementation using arrays of growing sizes
- cannot access using a[i] notation

```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```



Cost is linear!

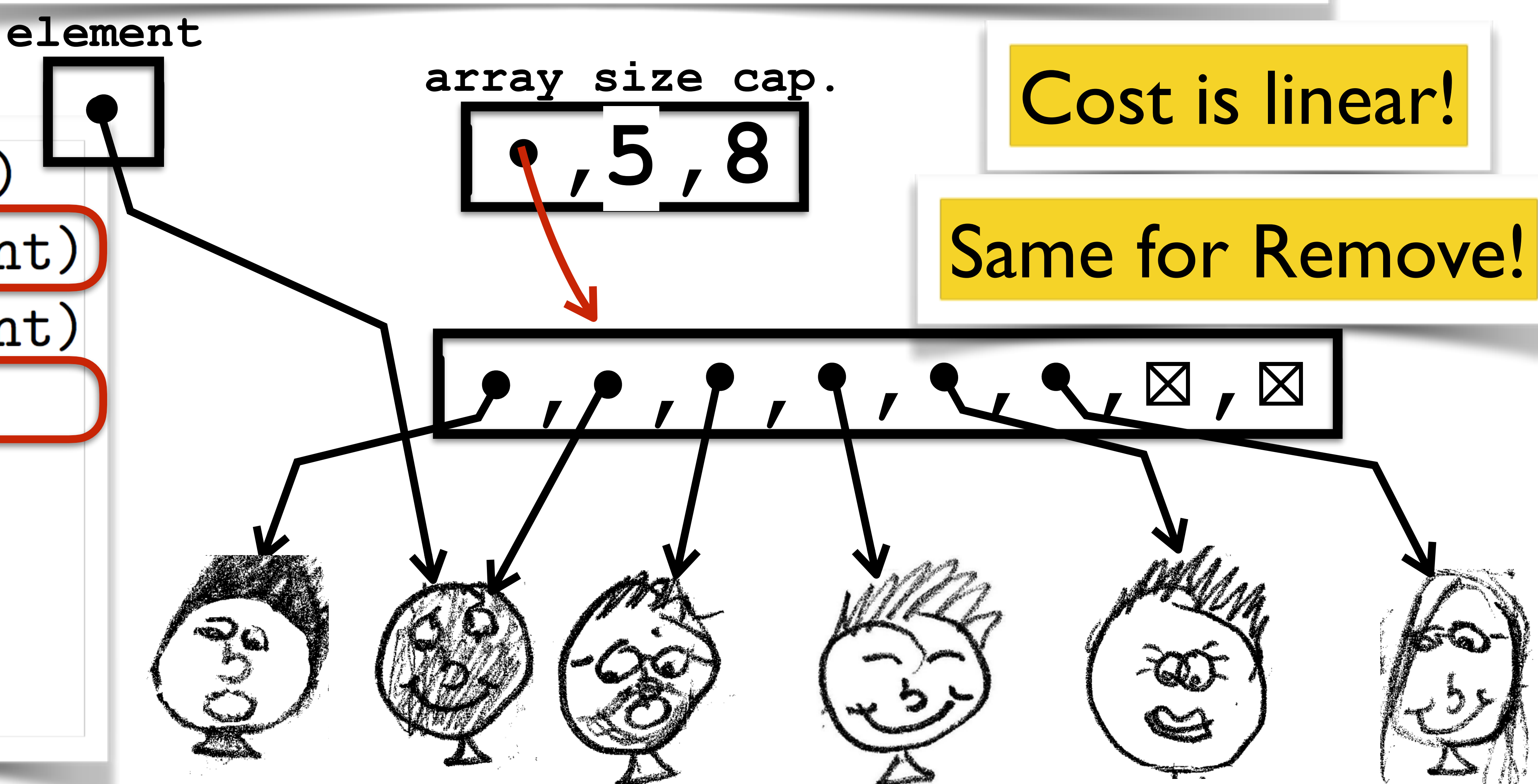
Java ArrayList

- implementation using arrays of <u>growing</u> sizes
- cannot access using a[i] notation

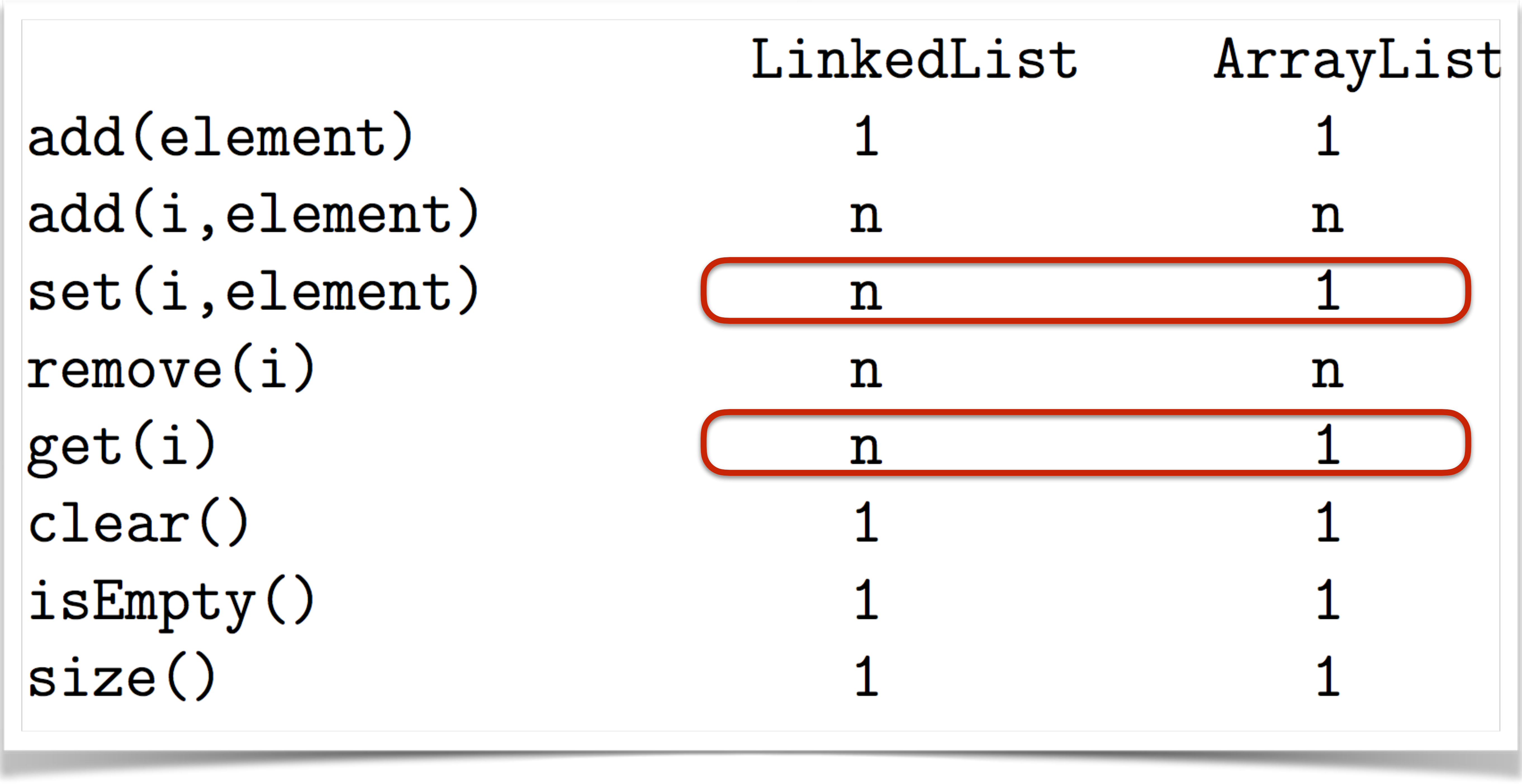
```
add(element)
add(i,element)
set(i,element)
remove(i)
get(i)
clear()
isEmpty()
size()
```

Cost is linear!

Same for Remove!



LinkedList vs ArrayList



	LinkedList	ArrayList
add(element)	1	1
add(i,element)	n	n
set(i,element)	n	1
remove(i)	n	n
get(i)	n	1
clear()	1	1
isEmpty()	1	1
size()	1	1

Winter 2016
COMP-250: Introduction to Computer Science

Lecture 6, January 28, 2016