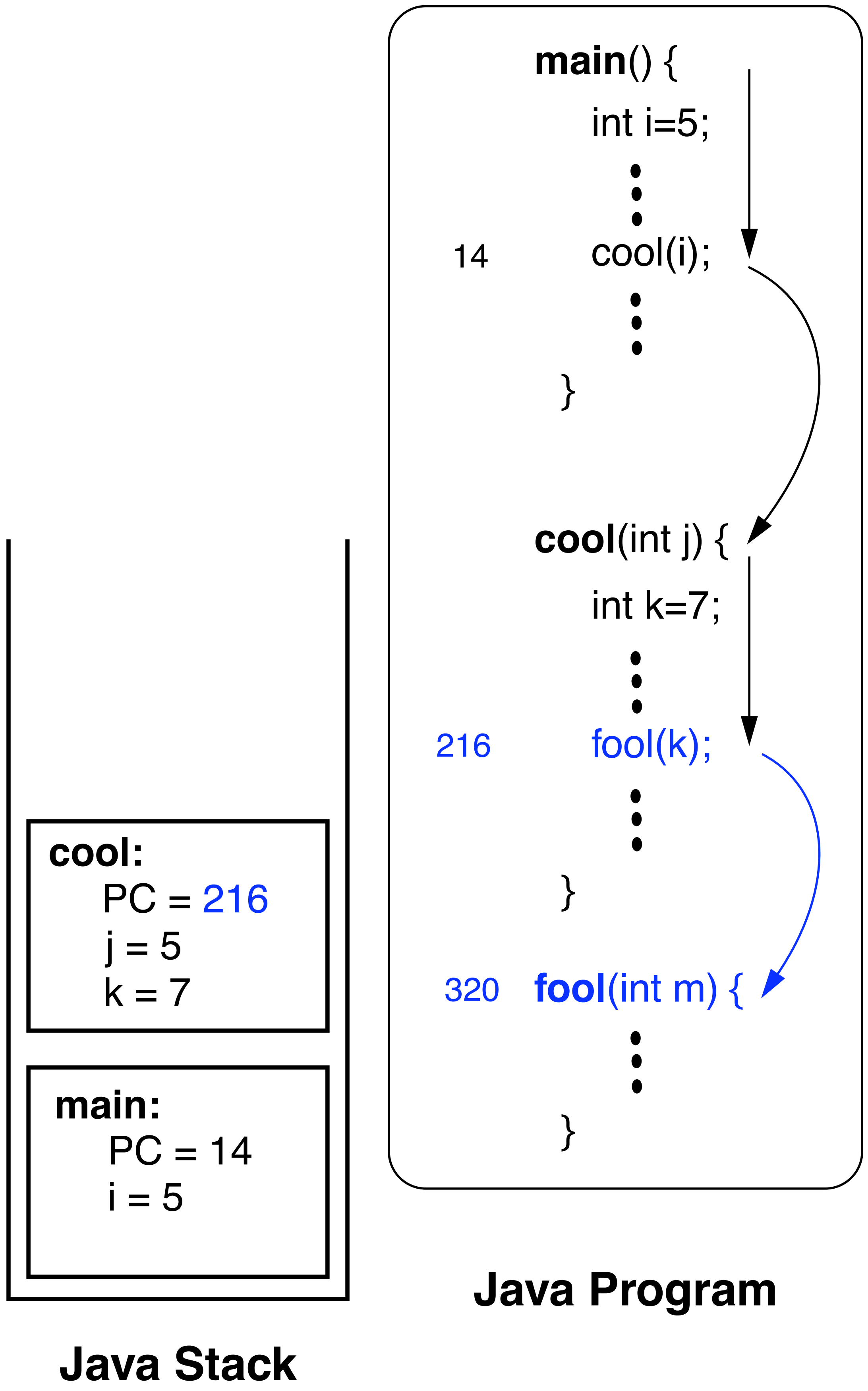
# Winter 2016
# COMP-250: Introduction to Computer Science

Lecture 8, February 4, 2016

# Stacks in the Java Virtual Machine

- Each process running in a Java program has its own Java Method Stack.
- Each time a method is called, it is pushed onto the stack.
- The choice of a stack for this operation allows Java to do several useful things:
  - Perform recursive method calls
  - Print stack traces to locate an error



**cool:**
PC = 216
j = 5
k = 7

**main:**
PC = 14
i = 5

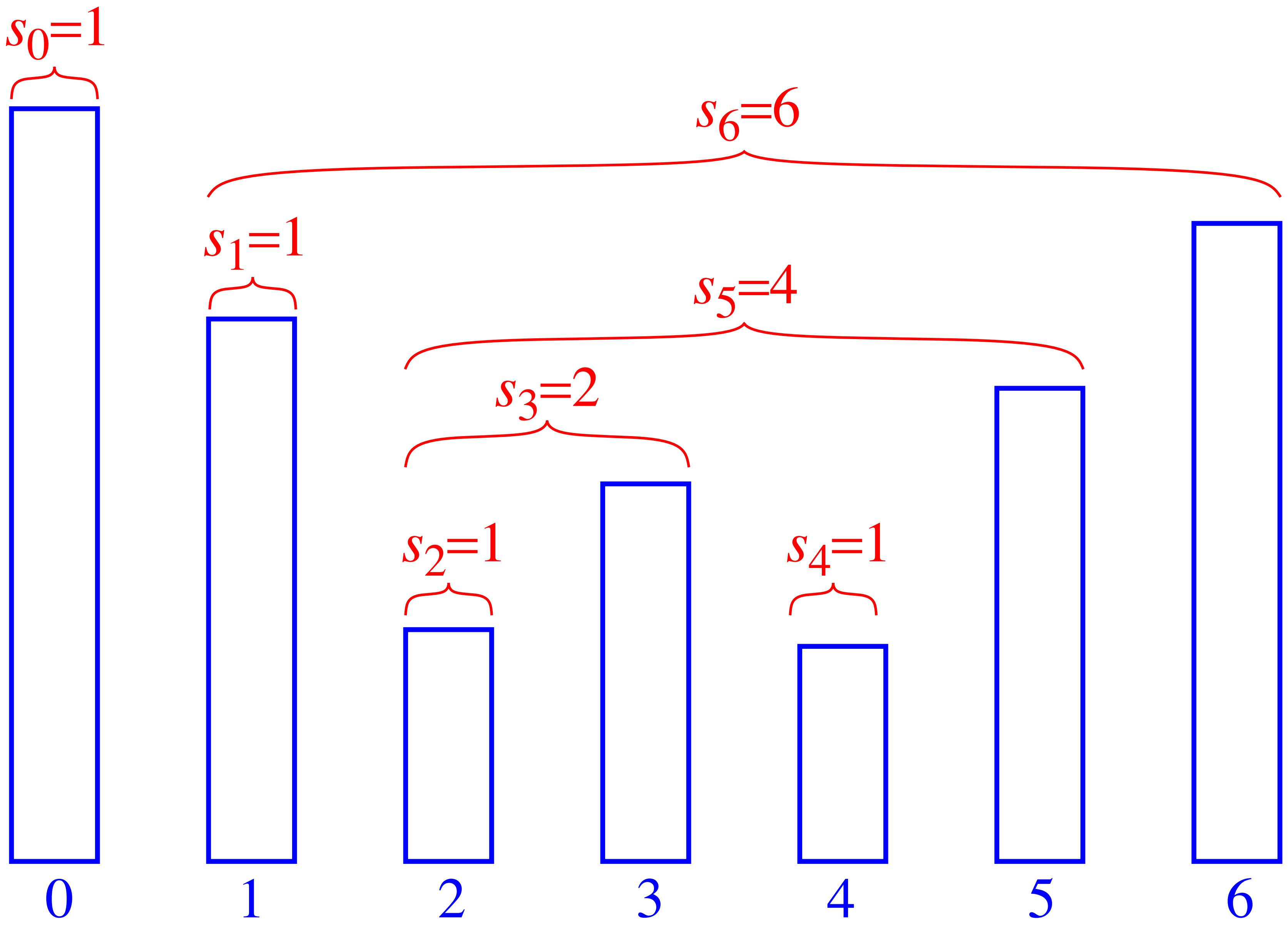
**Java Stack**

```
      main() {
          int i=5;
             ⋮
14        cool(i);
             ⋮
      }

      cool(int j) {
          int k=7;
             ⋮
216       fool(k);
             ⋮
      }

320   fool(int m) {
             ⋮
      }
```

**Java Program**

# Application: Time Series

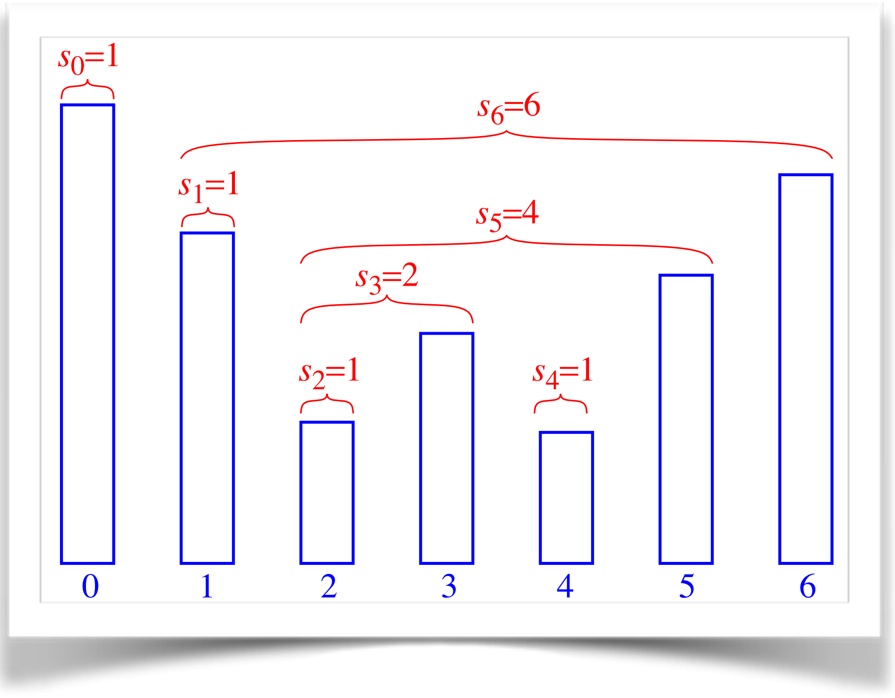
- The ***span*** $s_i$ of a stock's price on a certain day $i$ is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day $i$.



# An Inefficient Algorithm

- There is a straightforward way to compute the span of a stock on each of *n* days:

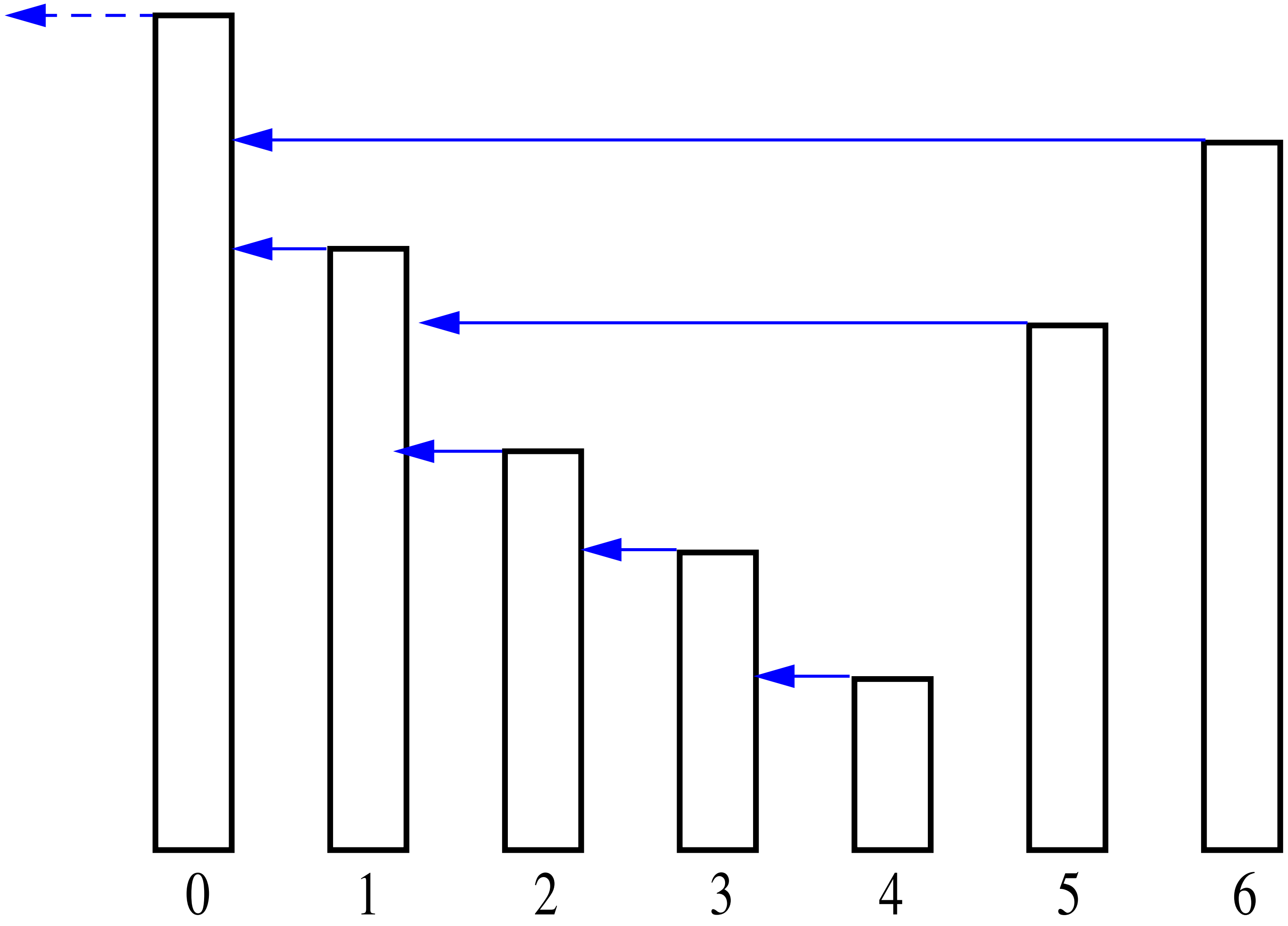
```
Algorithm computeSpans1($P$):
  Input: an $n$-element array $P$ of numbers such that
         $P[i]$ is the price of the stock on day $i$
  Output: an $n$-element array $S$ of numbers such that
         $S[i]$ is the span of the stock on day $i$
  for $i \leftarrow 0$ to $n - 1$ do
    $k \leftarrow 0$
    $done \leftarrow$ false
    repeat
      if $P[i - k] \leq P[i]$ then
        $k \leftarrow k + 1$
      else
        $done \leftarrow$ true
    until $(k = i)$ or $done$
    $S[i] \leftarrow k$
  return $S$
```



- The running time of this algorithm is (ugh!) $O(n^2)$. Why?

# A Stack Can Help

- We see that $s_i$ on day $i$ can be easily computed if we know the closest day preceding $i$, such that the price is greater than on that day than the price on day $i$. If such a day exists, let's call it $h(i)$, otherwise, we conventionally define $h(i) = -1$
- The span is now computed as $s_i = i - h(i)$

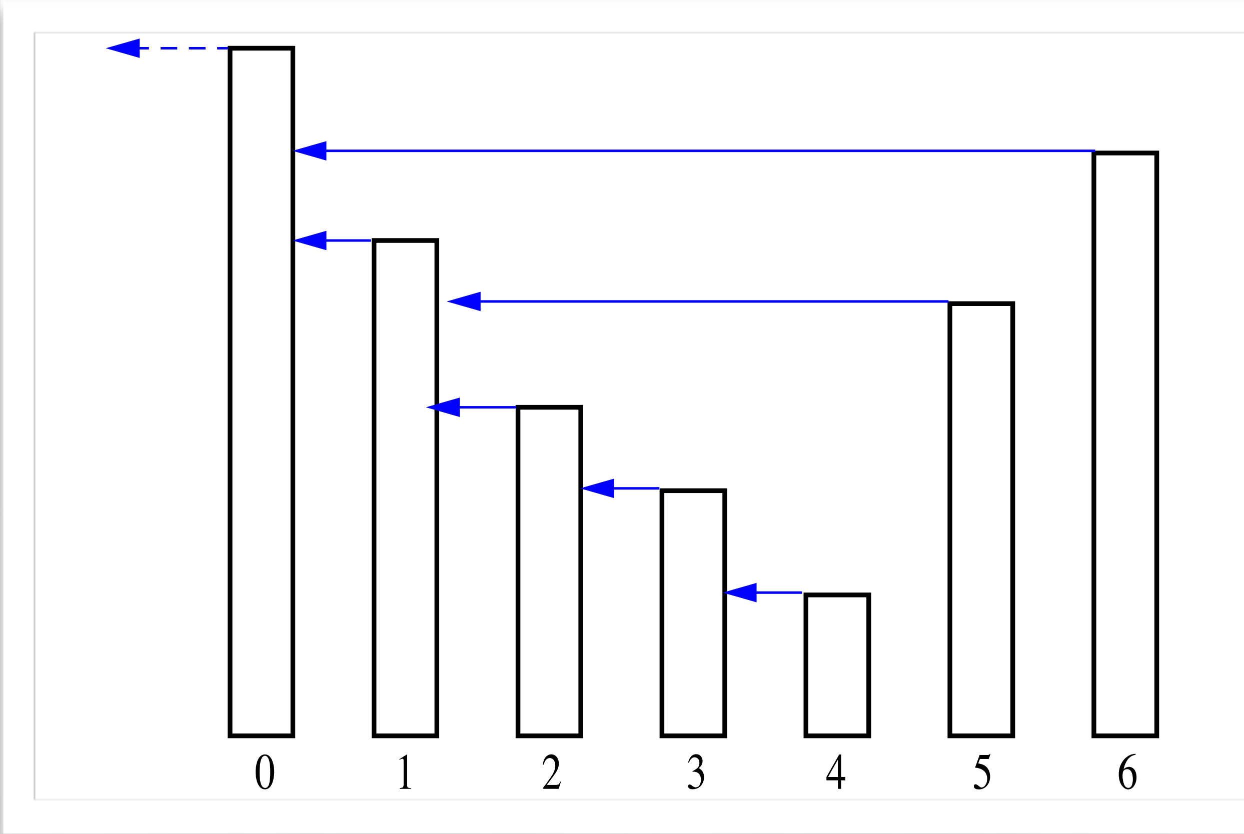


We use a ***stack*** to keep track of $h(i)$

# An Efficient Algorithm

- The code for our new algorithm:

```
Algorithm computeSpan2($P$):
  Input: An $n$-element array $P$ of numbers representing
         stock prices
  Output: An $n$-element array $S$ of numbers such that
          $S[i]$ is the span of the stock on day $i$
  Let $D$ be an empty stack
  for $i \leftarrow 0$ to $n - 1$ do
    $done \leftarrow$ false
    while not($D$.isEmpty() or $done$) do
      if $P[i] \geq P[D.\text{top}()]$ then
        $D$.pop()
      else
        $done \leftarrow$ true
      if $D$.isEmpty() then
        $h \leftarrow -1$
      else
        $h \leftarrow D.\text{top}()$
      $S[i] \leftarrow i - h$
      $D$.push($i$)
  return $S$
```



# Queue ADT

# Queue

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the rear (enqueued) and removed from the front (dequeued).

# Queue

- The queue has two fundamental methods:
  - enqueue(*o*): Inserts object *o* at rear of the queue
  - dequeue(): Removes object from front of queue and returns it; an error occurs if queue is empty.
- These support methods should also be defined:
  - size(): Returns number of objects in the queue
  - isEmpty(): Returns a boolean value that indicates whether the queue is empty
  - front(): Returns, but not remove, the front object in the queue; an error occurs if queue is empty.

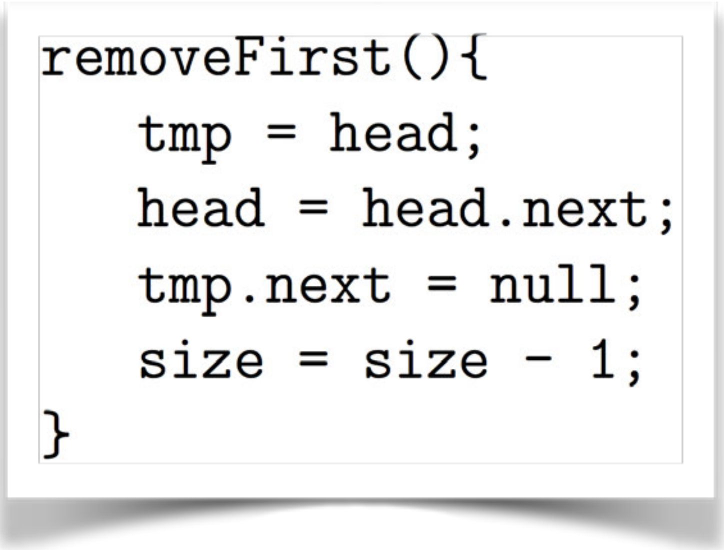
# Queue

| OPERATION | STATE |
|---|---|
| | – |
| add(a) | a |
| add(b) | ab |
| remove() | b |
| add(c) | bc |
| add(d) | bcd |
| add(e) | bcde |
| remove() | cde |
| add(f) | cdef |
| remove() | def |
| add(g) | defg |

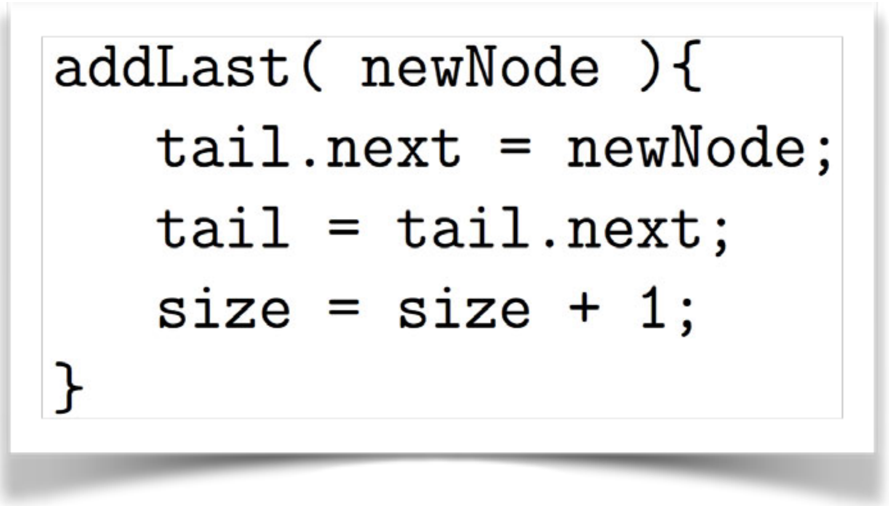
```
                 ------------------------------------
                 0123456789        head    size
OPERATION
                                    0       0
add(a)           a                  0       1
add(b)           ab                 0       2
remove()          b                 1       1
add(c)            bc                1       2
add(d)            bcd               1       3
add(e)            bcde              1       4
remove()           cde              2       3
add(f)             cdef             2       4
remove()            def             3       3
add(g)              defg            3       4
                 -------------------------------------
```

```
                -----------------------------------
                0123456[redacted] head    size
OPERATION
                                    0      0
add(a)          a                   0      1
add(b)          ab                  0      2
remove()         b                  1      1
add(c)           bc                 1      2
add(d)           bcd                1      3
add(e)           bcde               1      4
remove()          cde               2      3
add(f)            cdef              2      4
remove()           def              3      3
add(g)             defg             3      4
                ------------------------------------
```

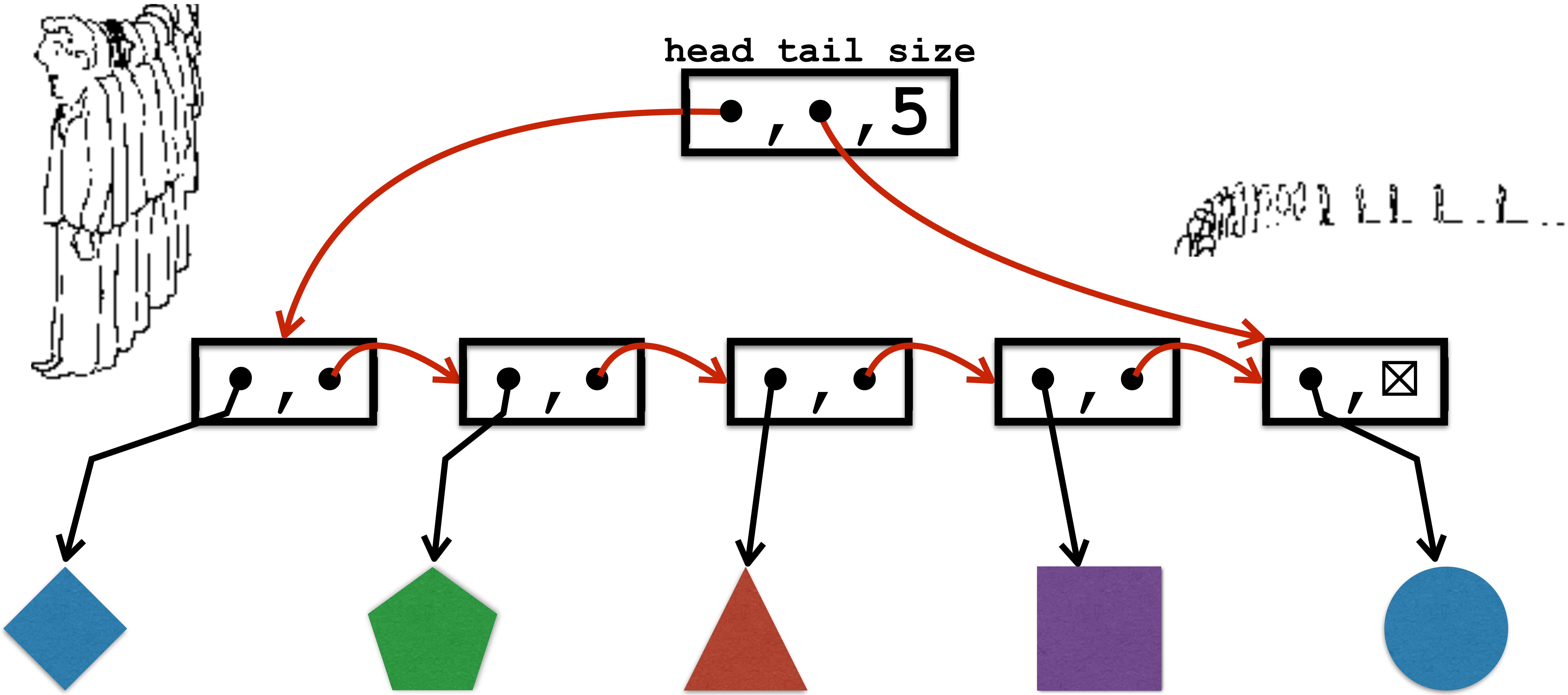
# Queue as List



```
removeFirst(){
    tmp = head;
    head = head.next;
    tmp.next = null;
    size = size - 1;
}
```

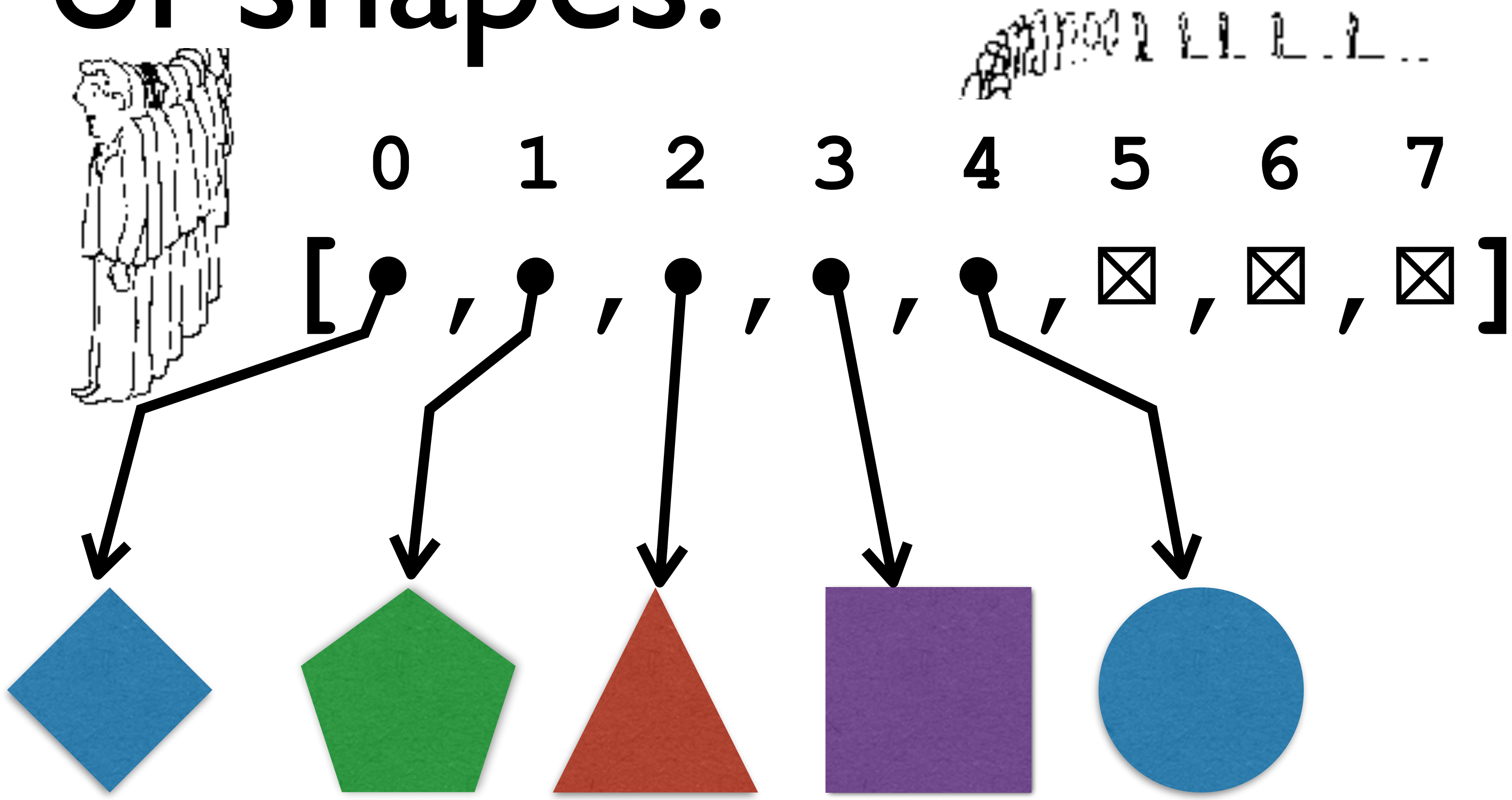


```
addLast( newNode ){
    tail.next = newNode;
    tail = tail.next;
    size = size + 1;
}
```



# Queue as Array

Array of shapes:

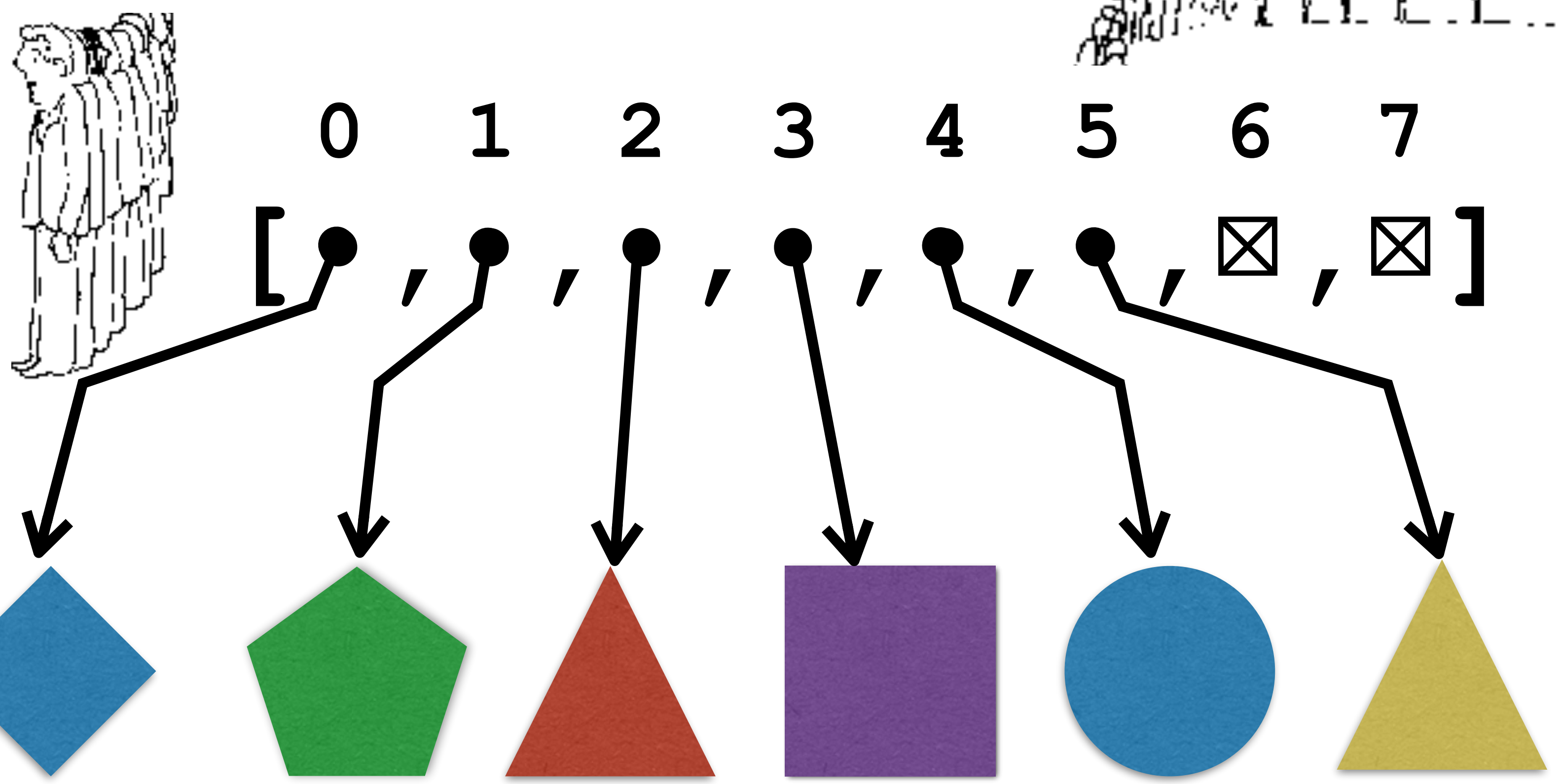


`0 1 2 3 4 5 6 7`

`[●,●,●,●,●,⊠,⊠,⊠]`

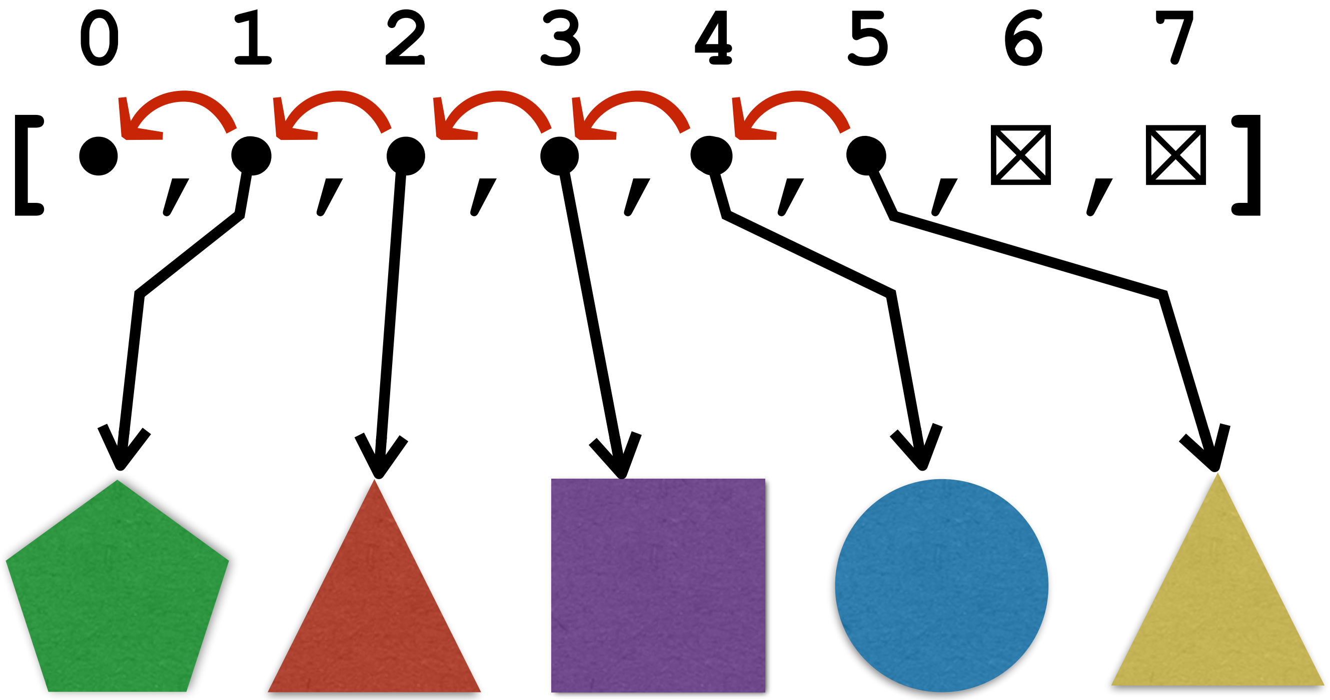
Size=5

# Queue as Array



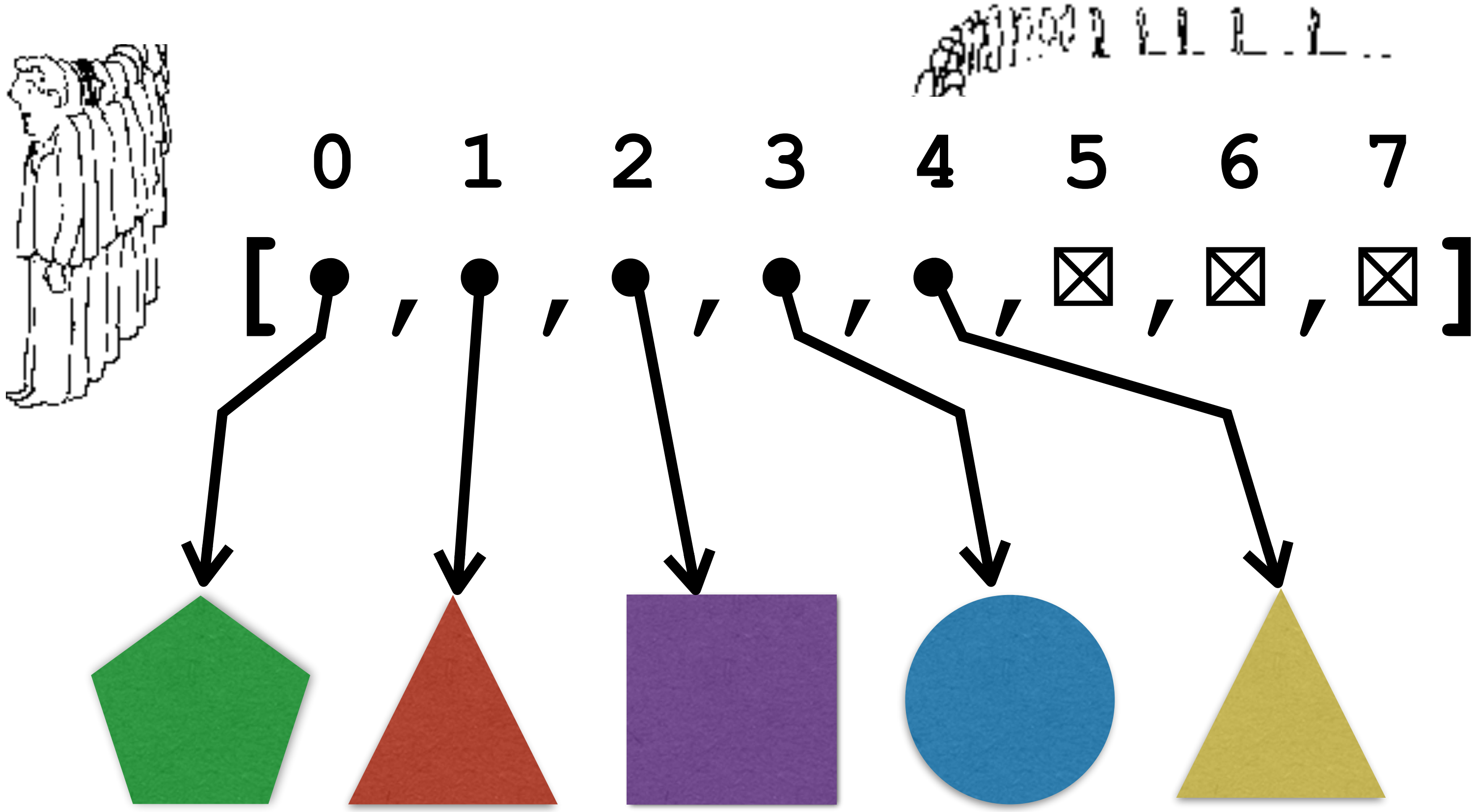
Size=6

# Queue as Array



Size=5

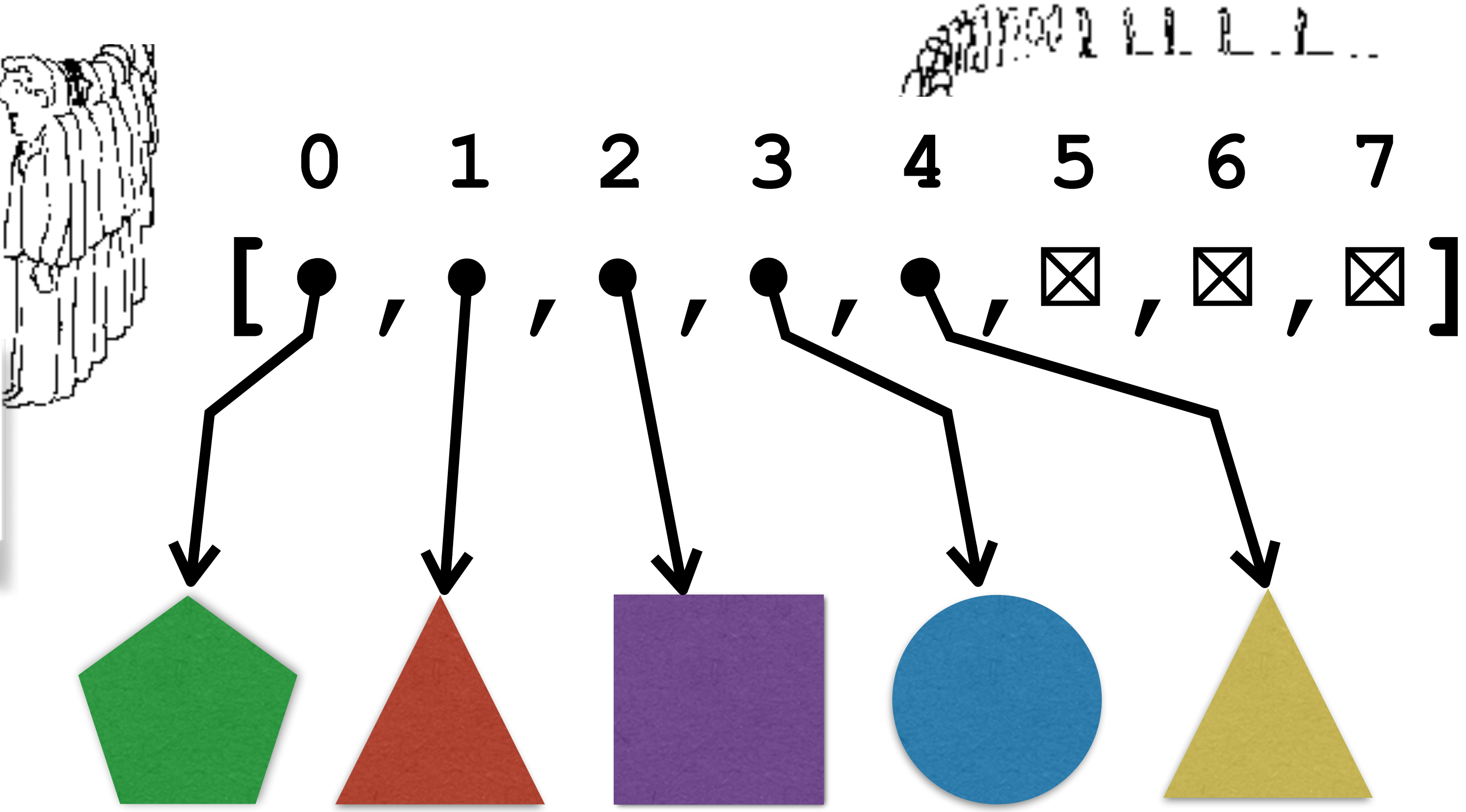
# Queue as Array



```
  0   1   2   3   4   5   6   7
[ ● , ● , ● , ● , ● , ☒ , ☒ , ☒ ]
```

Size=5

# Queue as Array



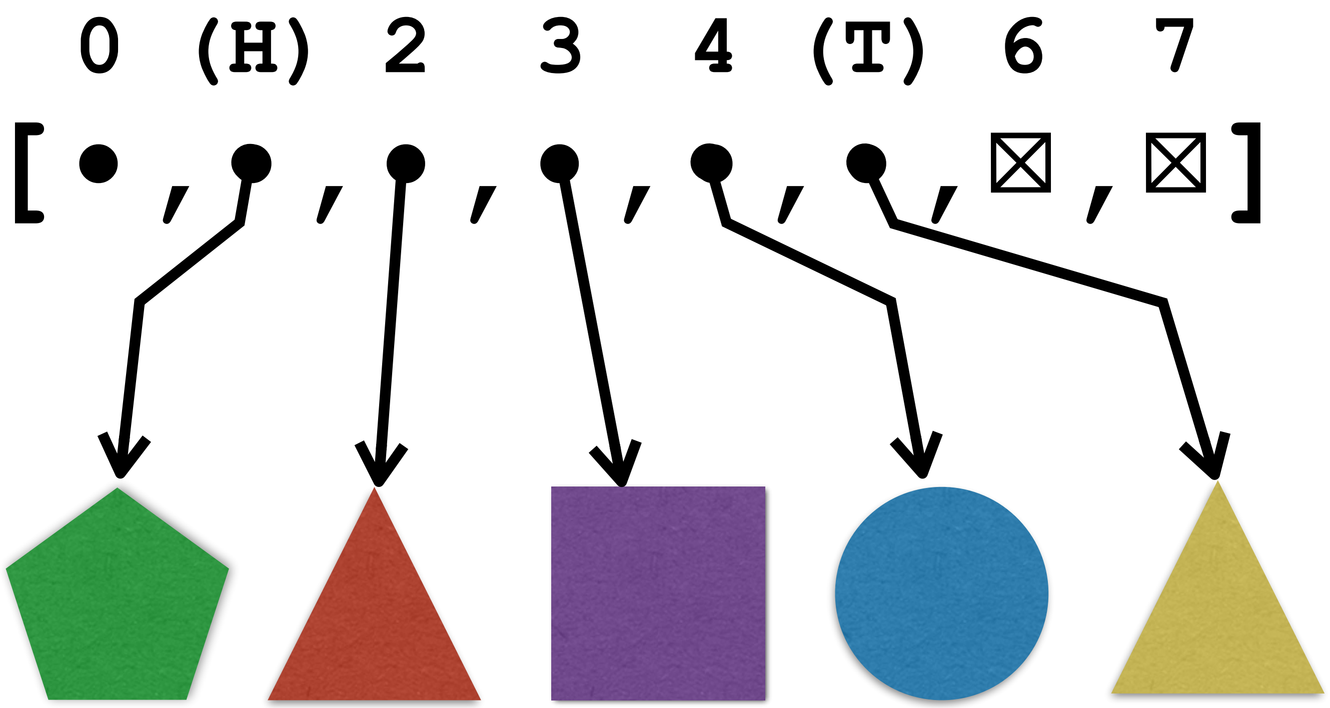
0 1 2 3 4 5 6 7

[•,•,•,•,•,⊠,⊠,⊠]

But why ??

Size=5

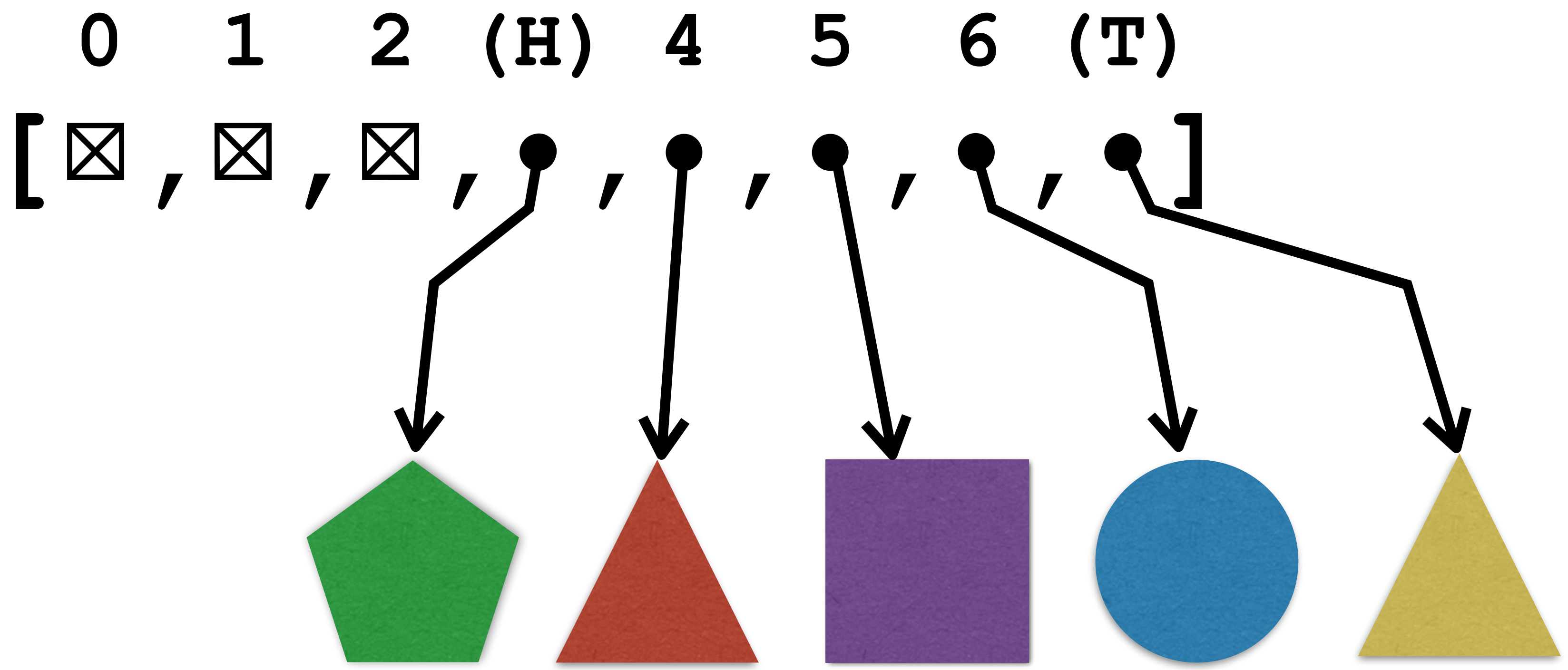
# Queue as Array



```
 0  (H)  2   3   4  (T)  6   7
[• , • , • , • , • , • , ⊠ , ⊠]
```

head=1, tail=5, (size=5)

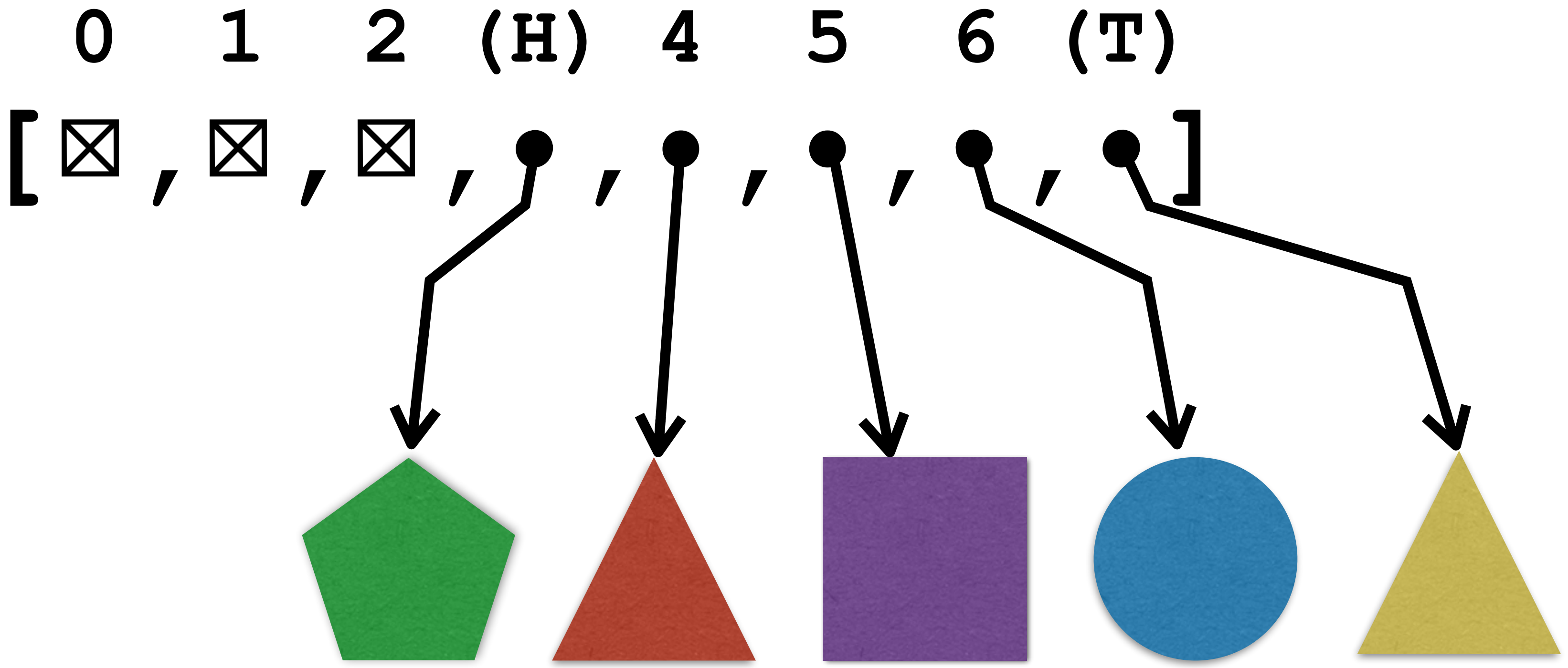
# Queue as Array



```
  0   1   2  (H)  4   5   6  (T)
[☒ , ☒ , ☒ , ● , ● , ● , ● , ● ]
```

head=3, tail=7, (size=5)

# Queue as Array



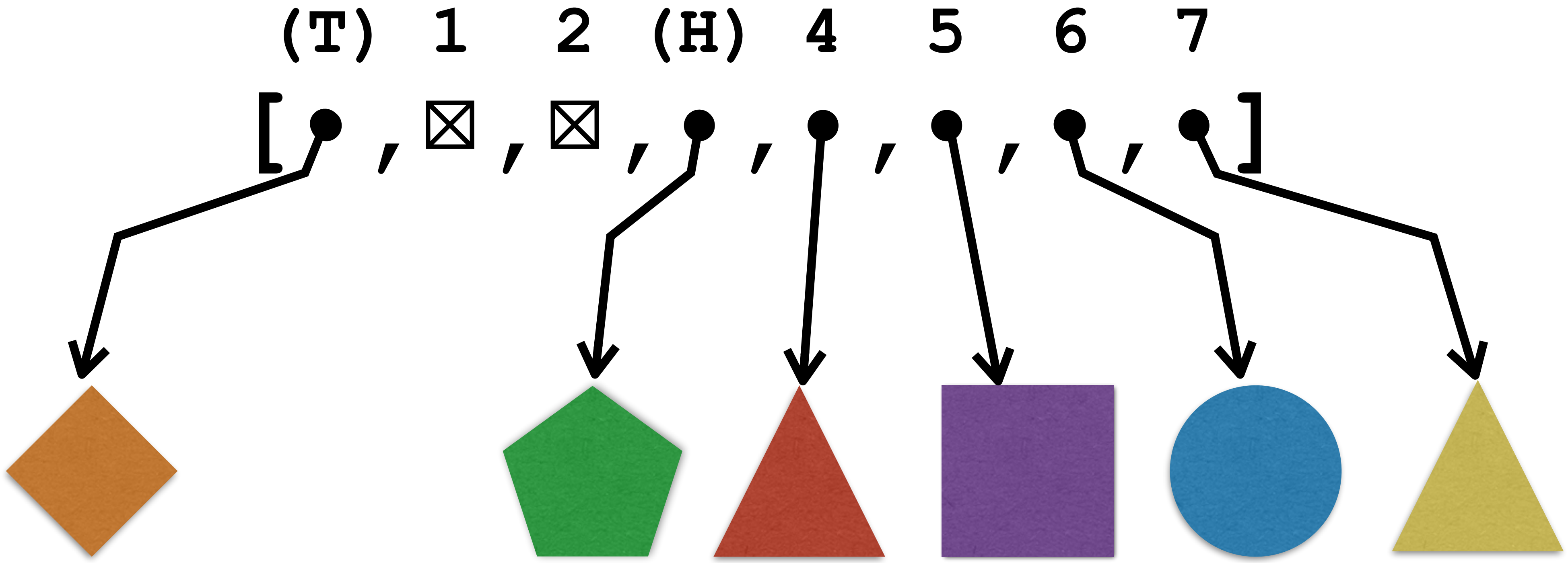
`0   1   2  (H)  4   5   6  (T)`

`[☒,☒,☒,●,●,●,●,●]`

FULL ??

head=3, tail=7, (size=5)

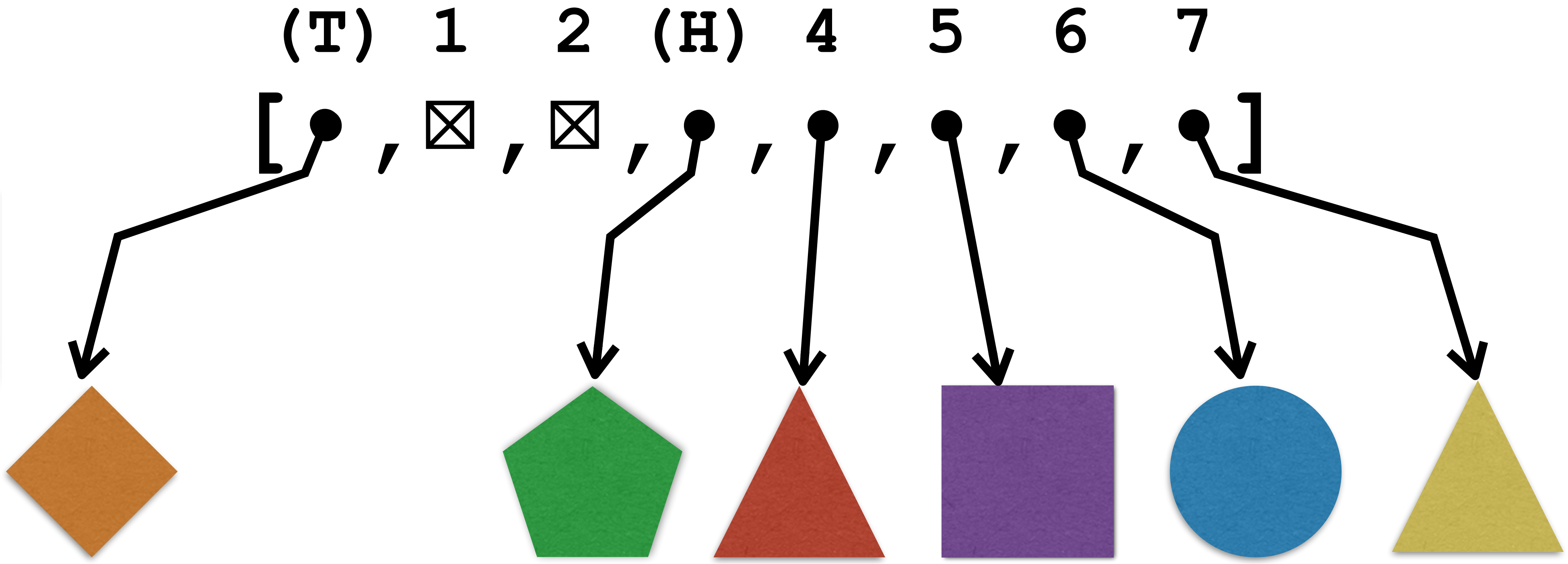
# Queue as Array



```
(T) 1   2  (H)  4   5   6   7
[ ● , ⊠ , ⊠ , ● , ● , ● , ● , ● ]
```

head=3, tail=0, (size=6)

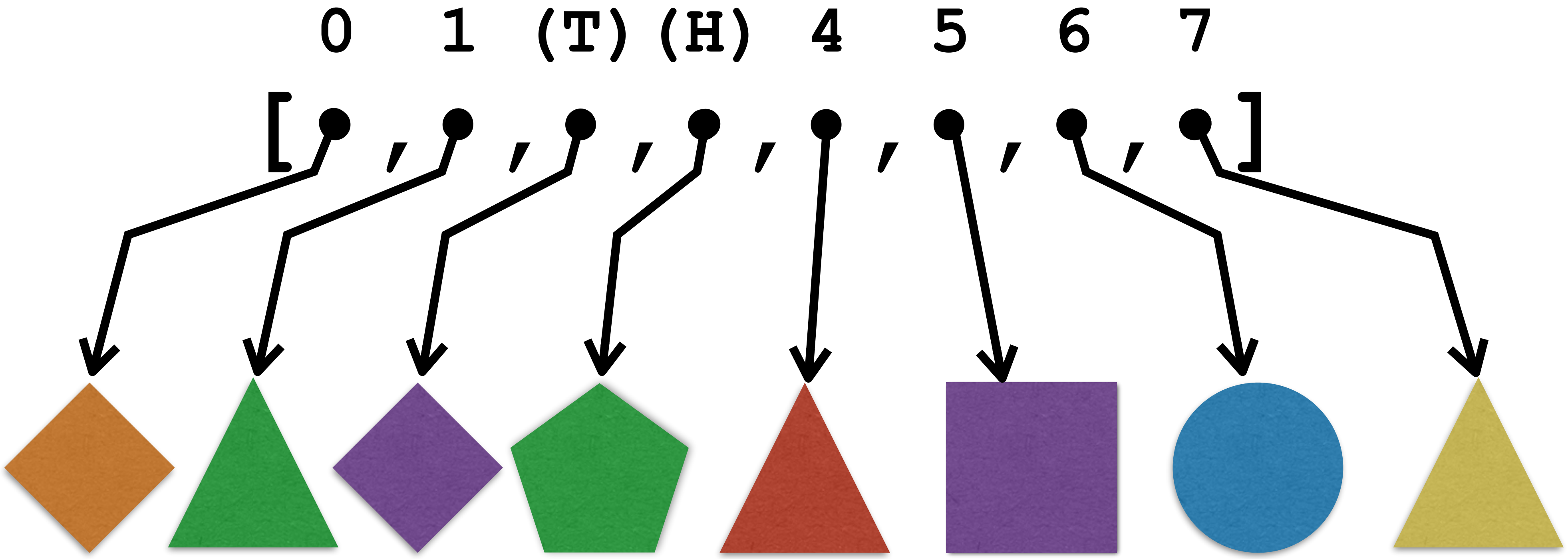
# Queue as Array



NO !!

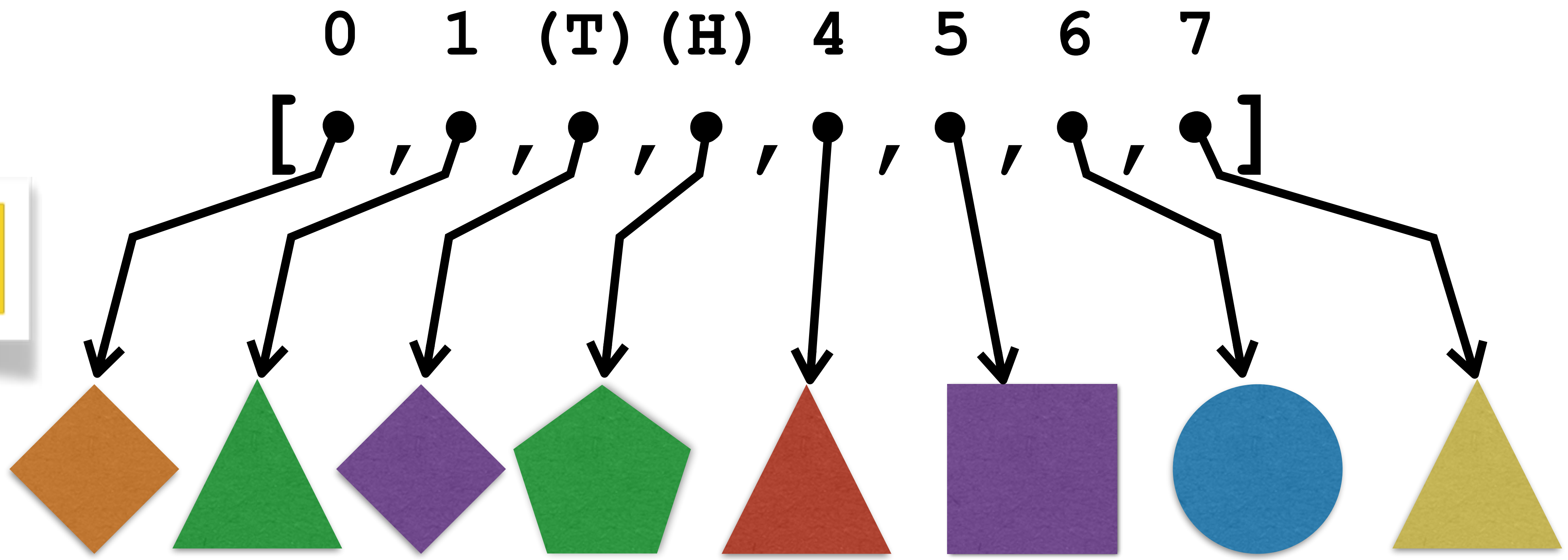
head=3, tail=0, (size=6)

# Queue as Array



head=3, tail=2, (size=8)

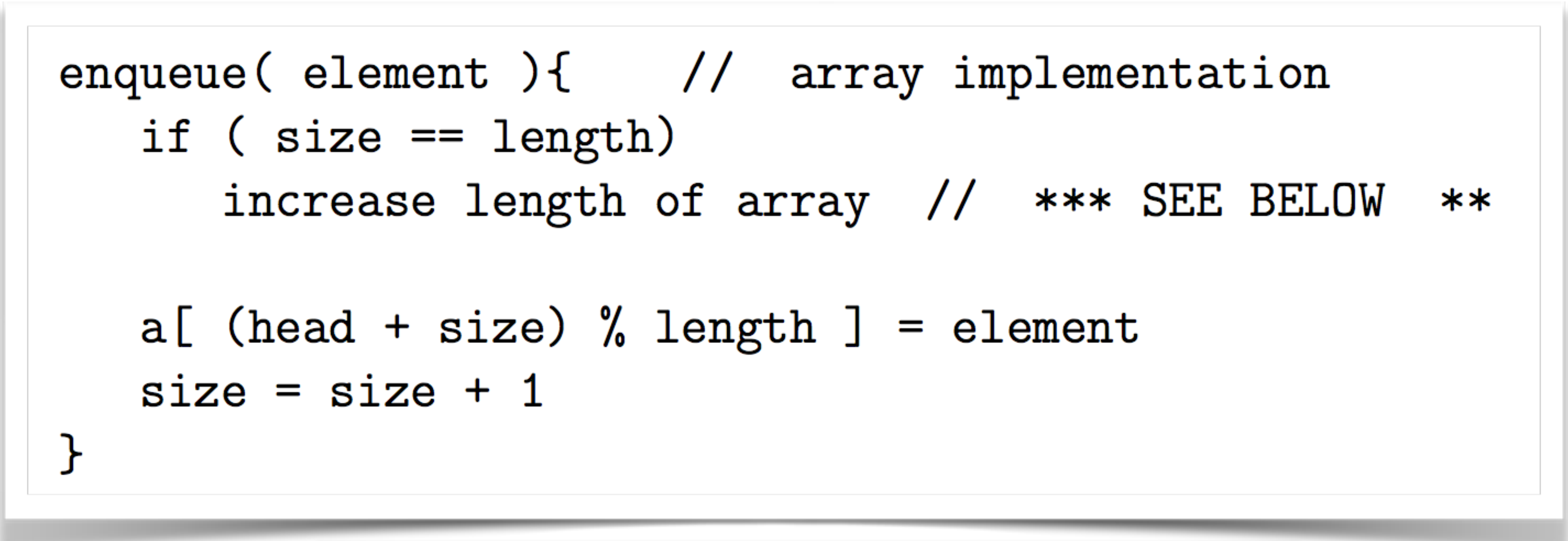
# Queue as Array



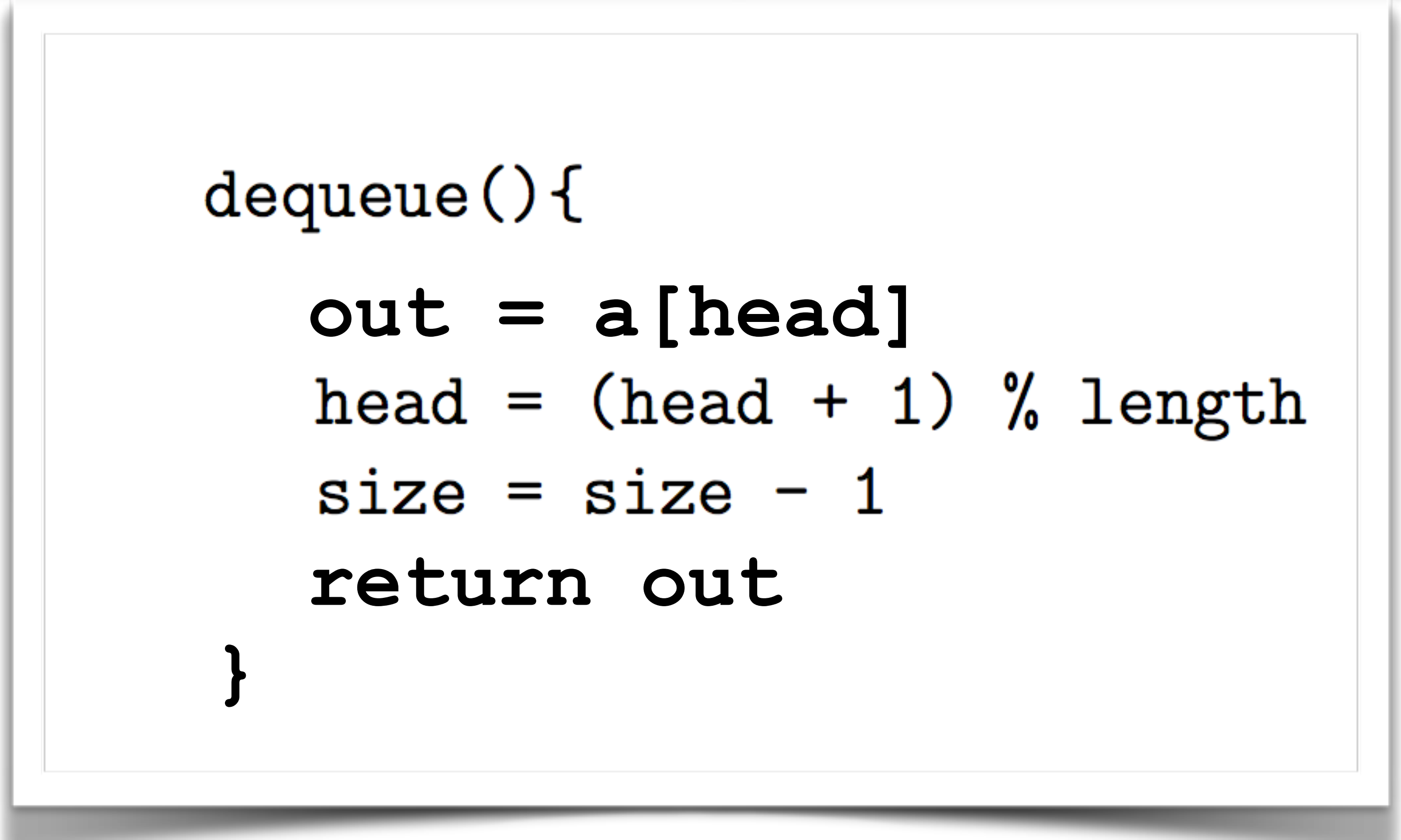
FULL !!

head=3, tail=2, (size=8)

# Queue as Array

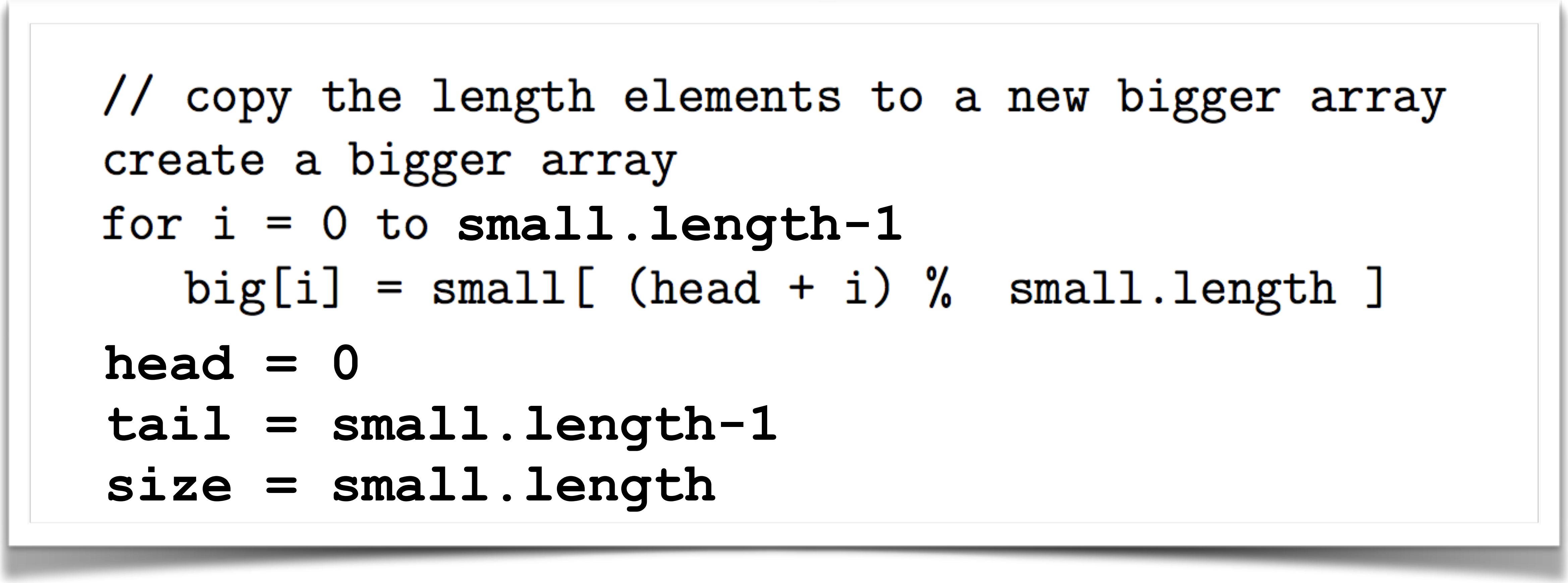


```
enqueue( element ){    //  array implementation
   if ( size == length)
      increase length of array  //  *** SEE BELOW  **

   a[ (head + size) % length ] = element
   size = size + 1
}
```



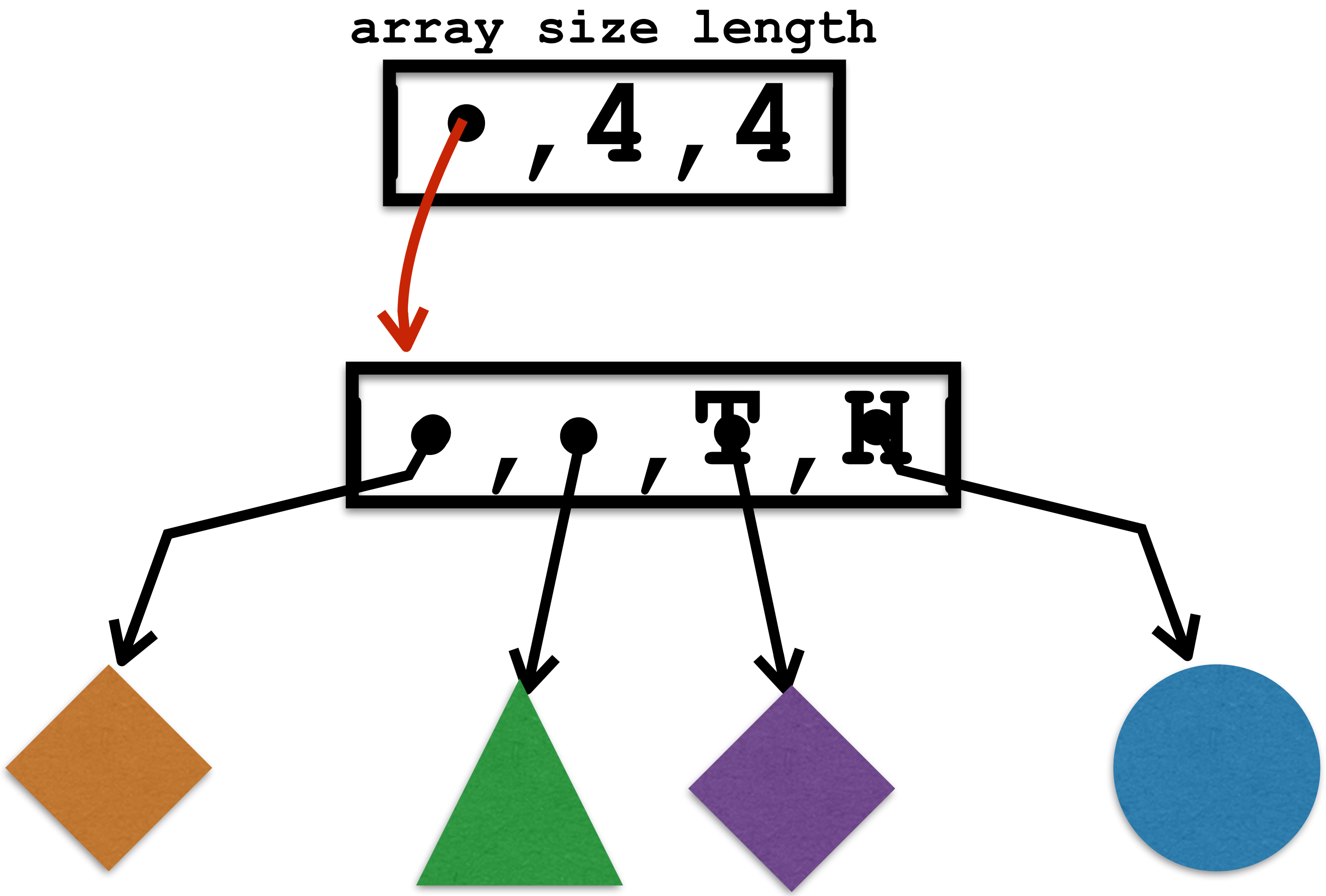
```
dequeue(){
   out = a[head]
   head = (head + 1) % length
   size = size - 1
   return out
}
```

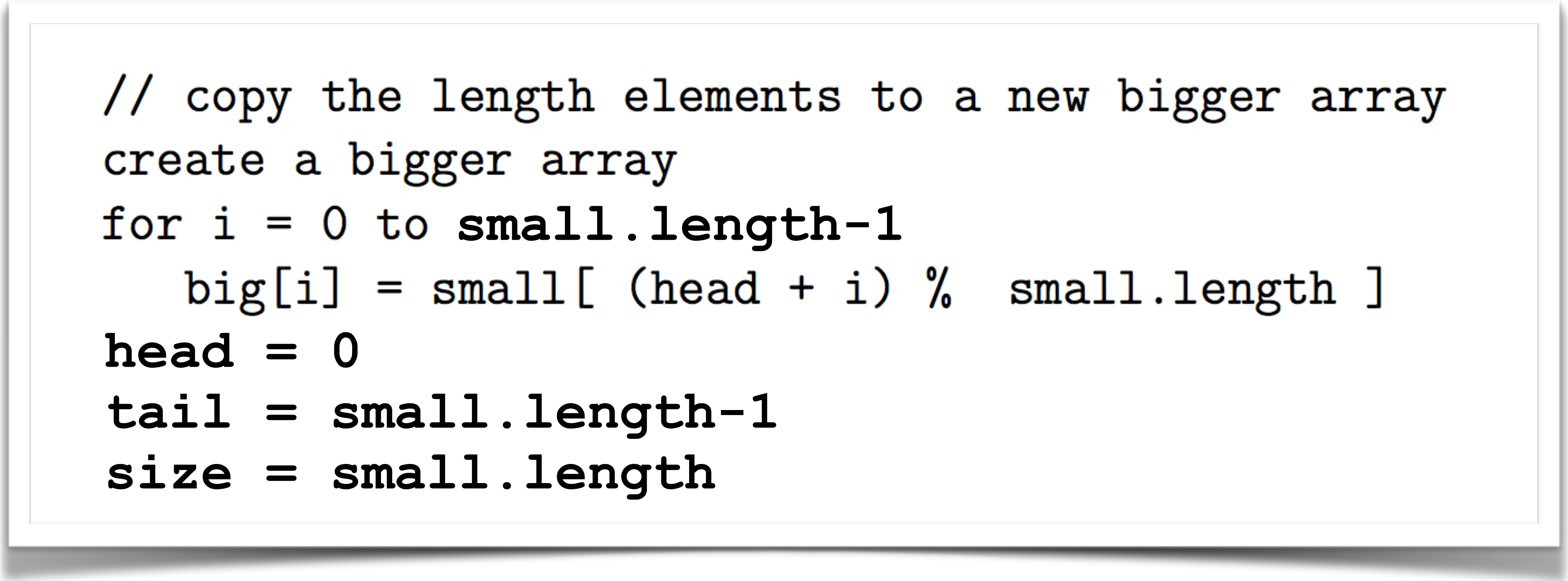
# Queue as Array



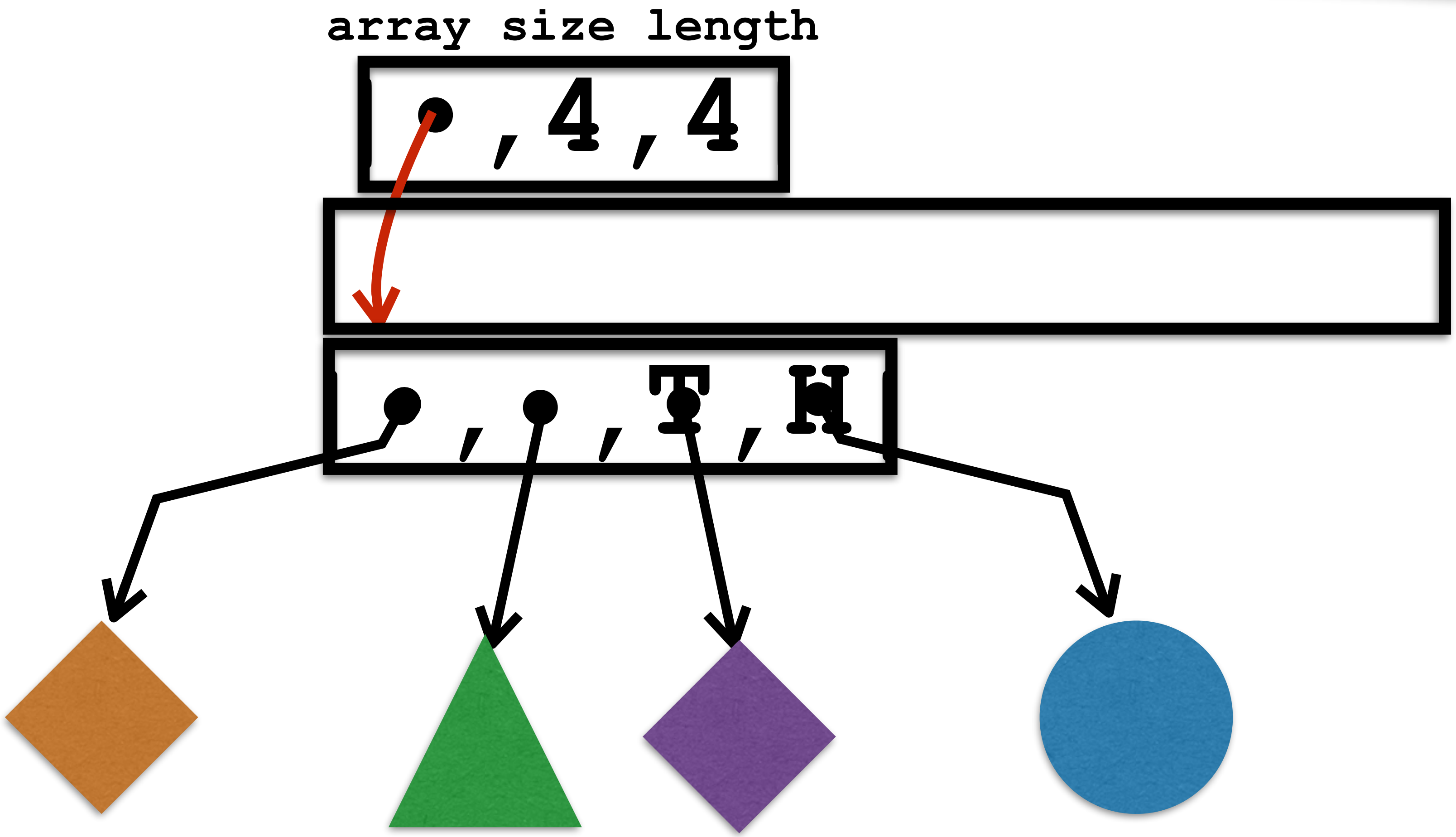
```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
```

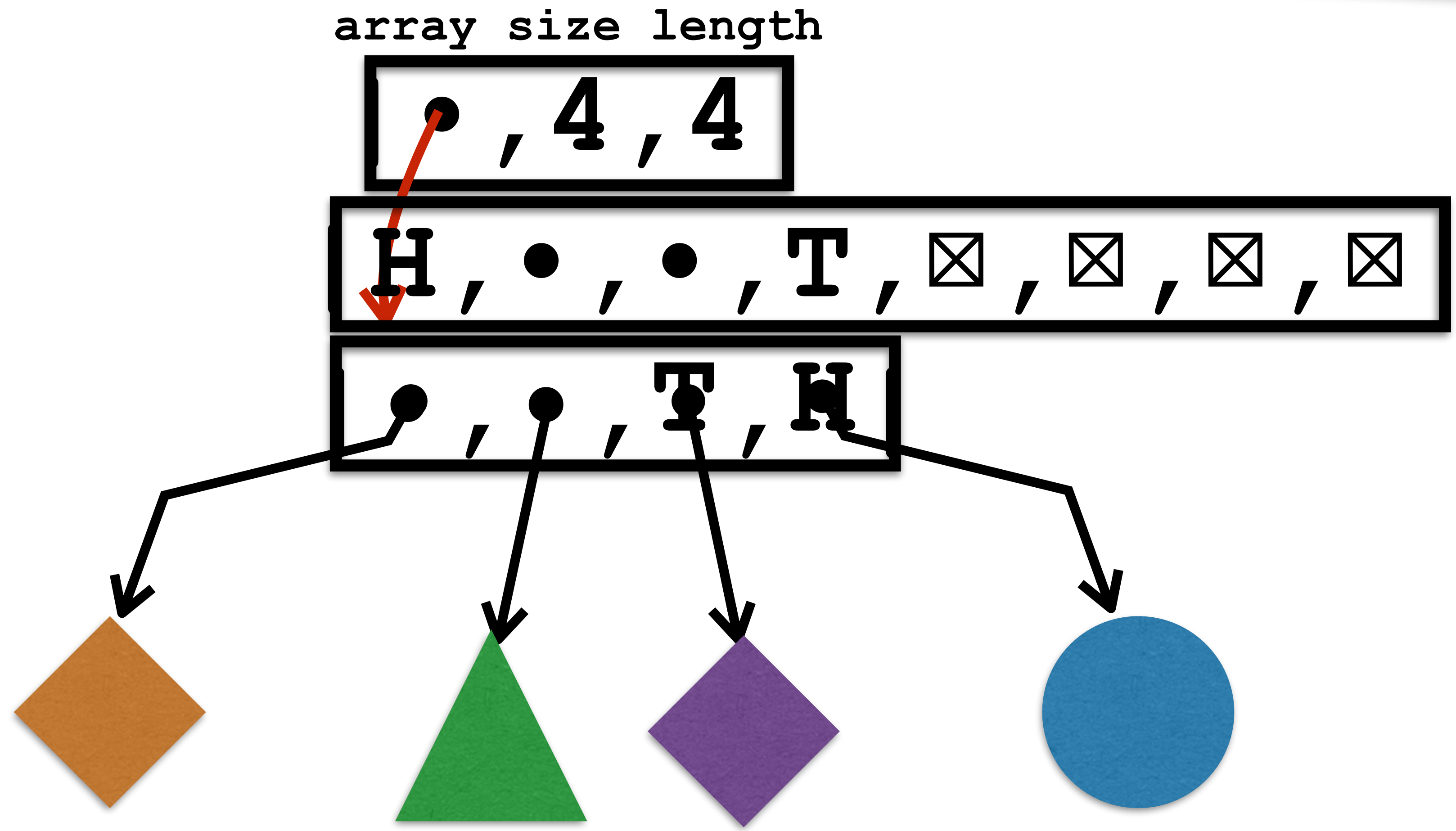


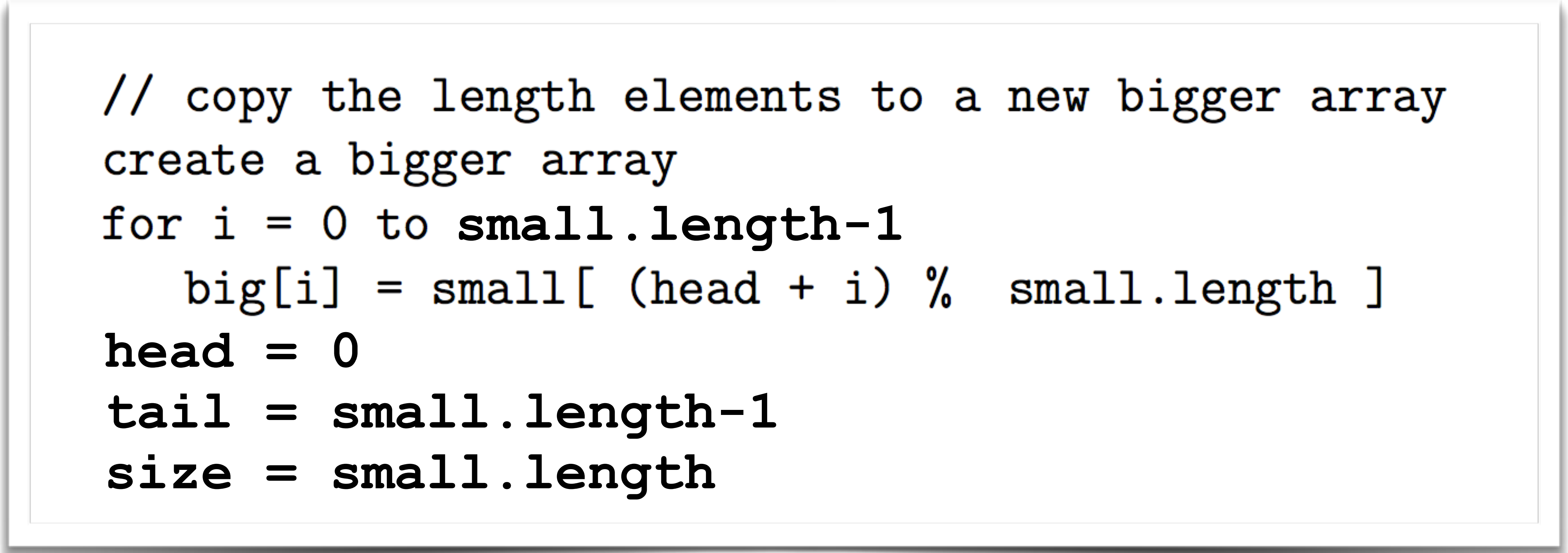


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

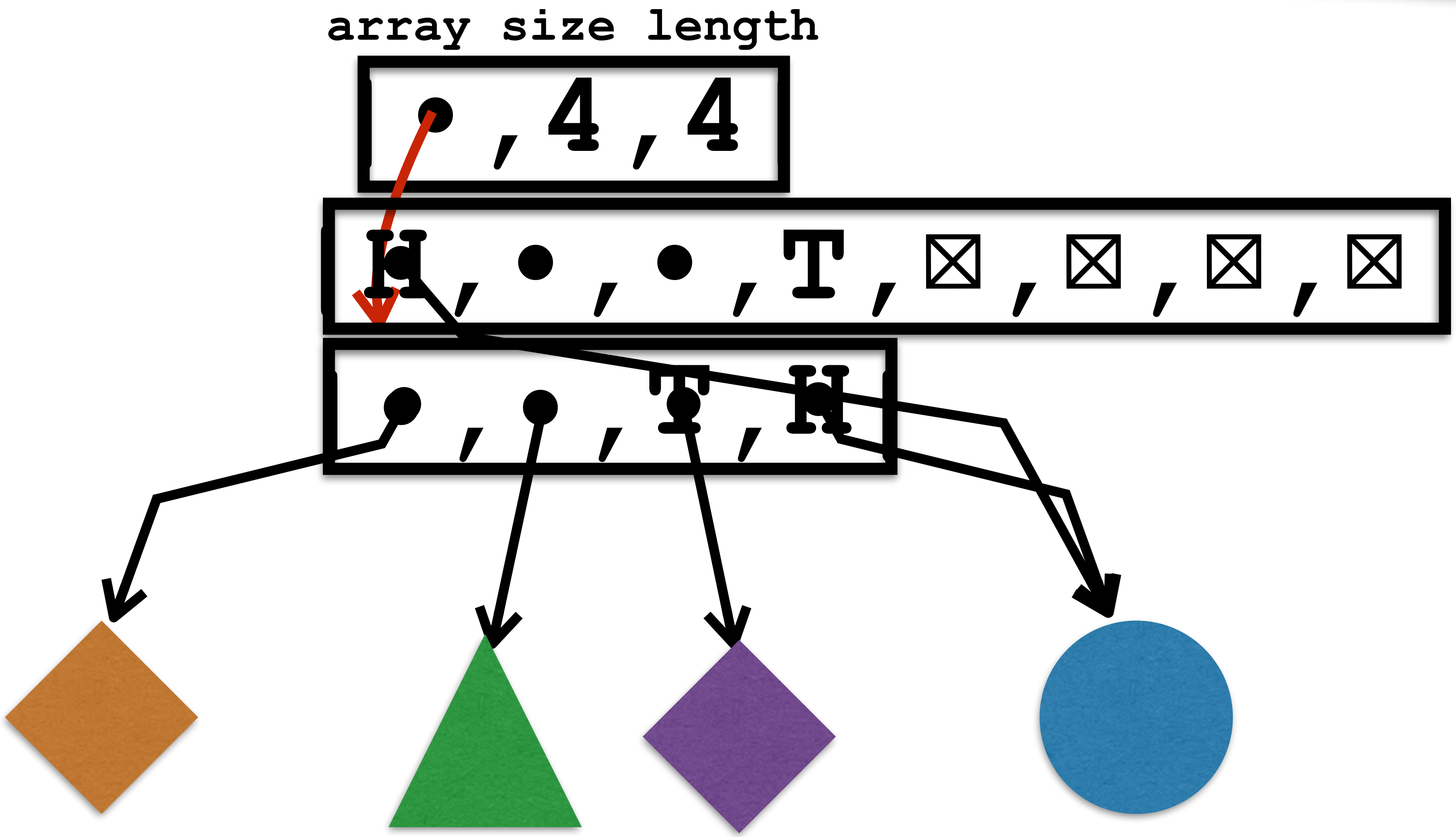


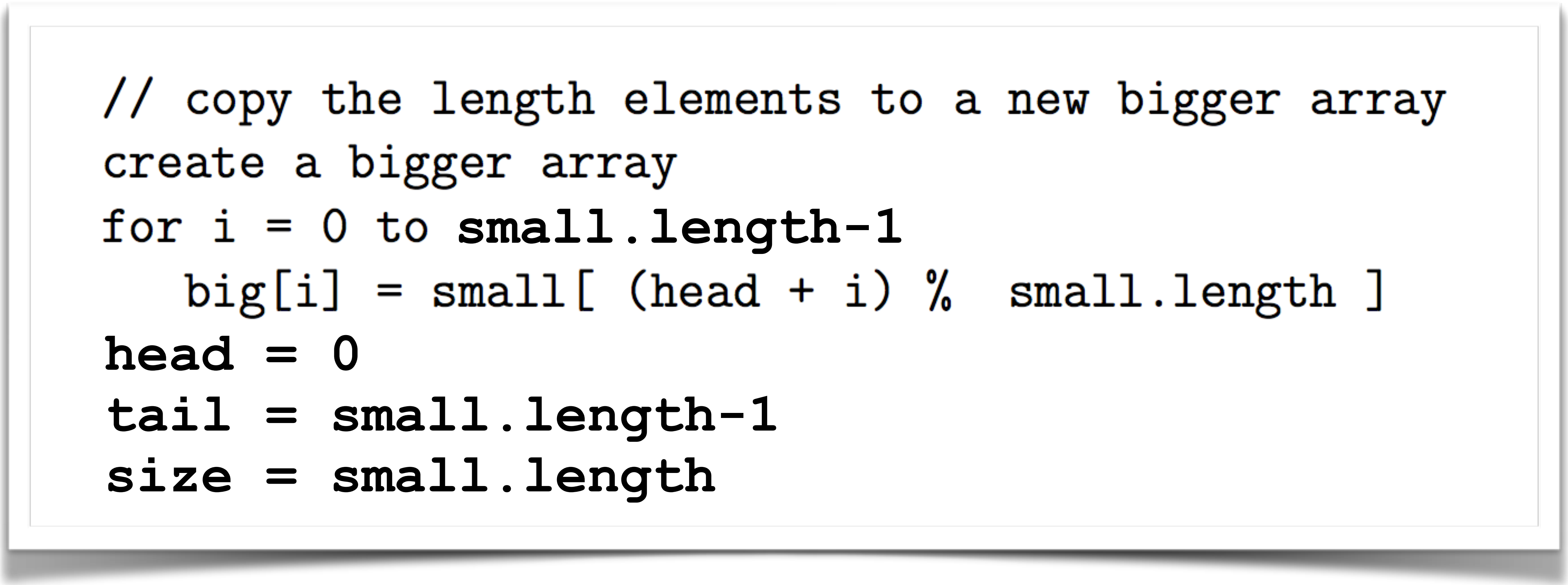
```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```



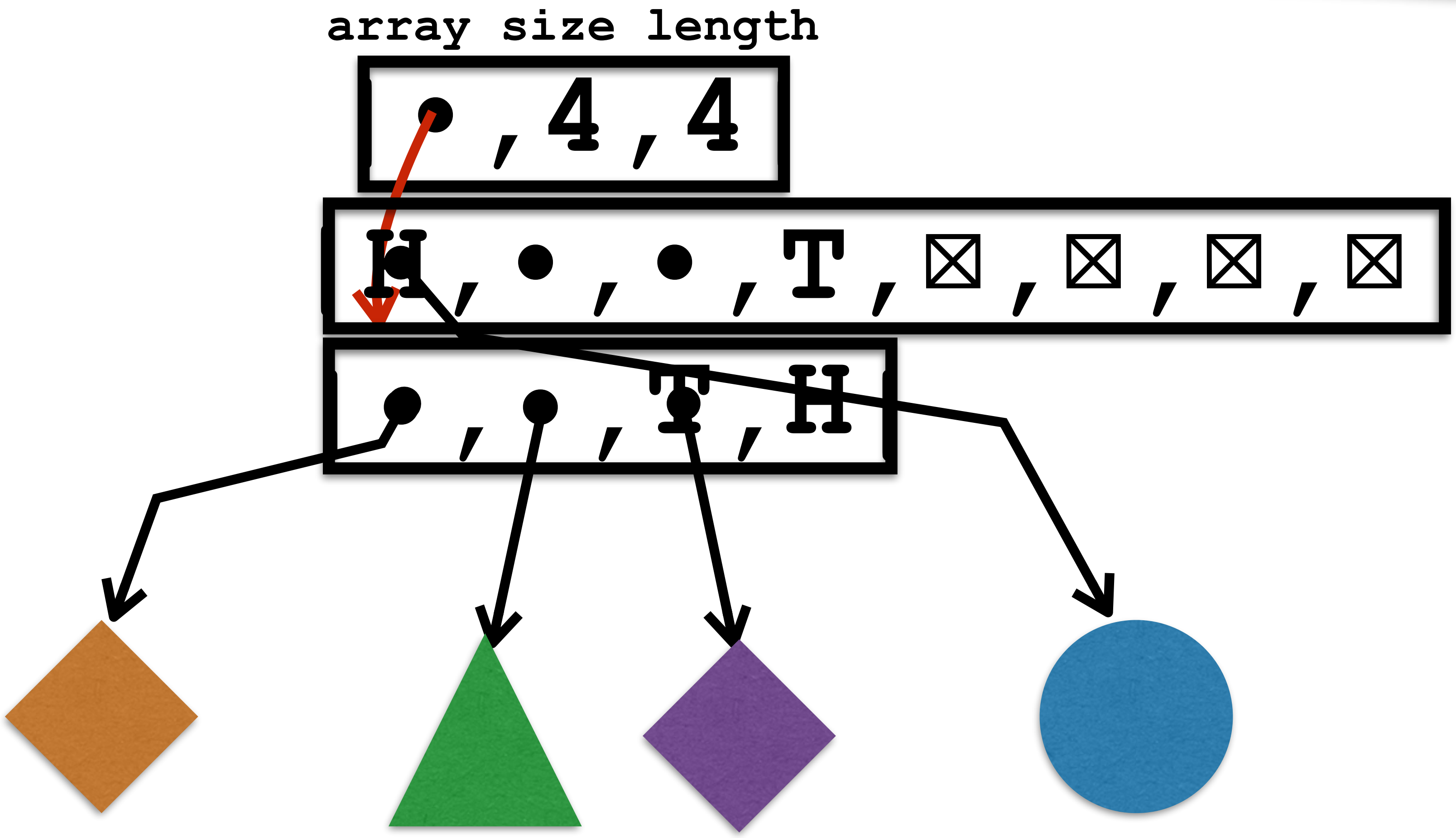


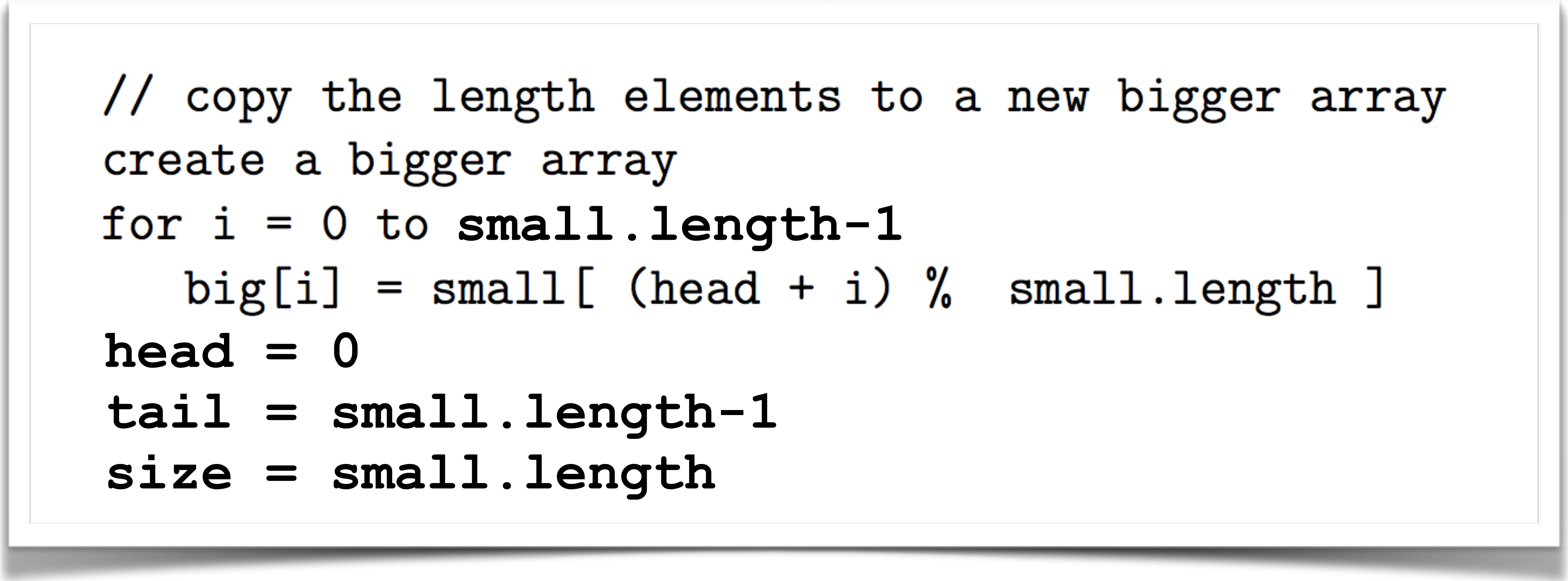
```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```



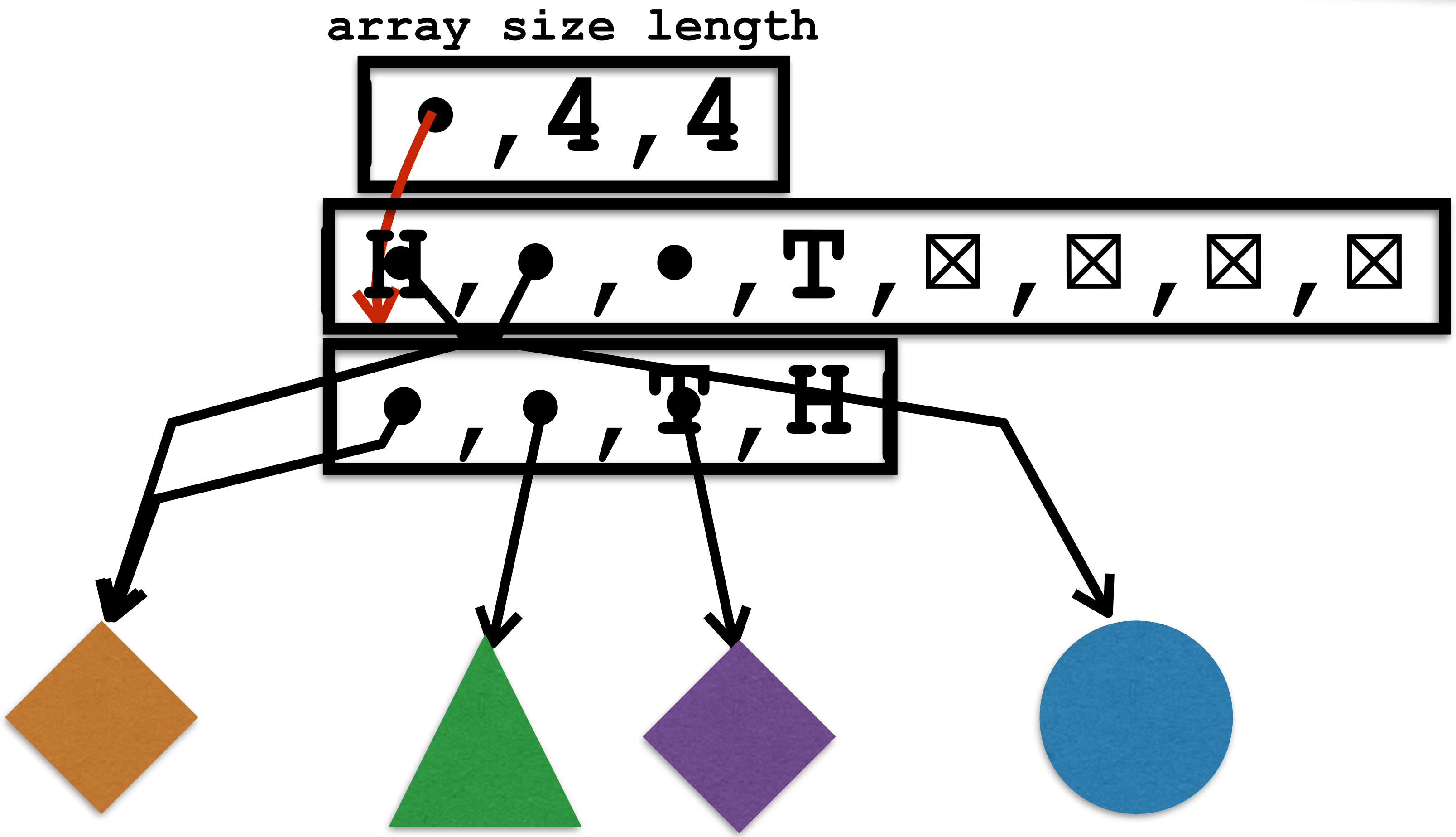


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

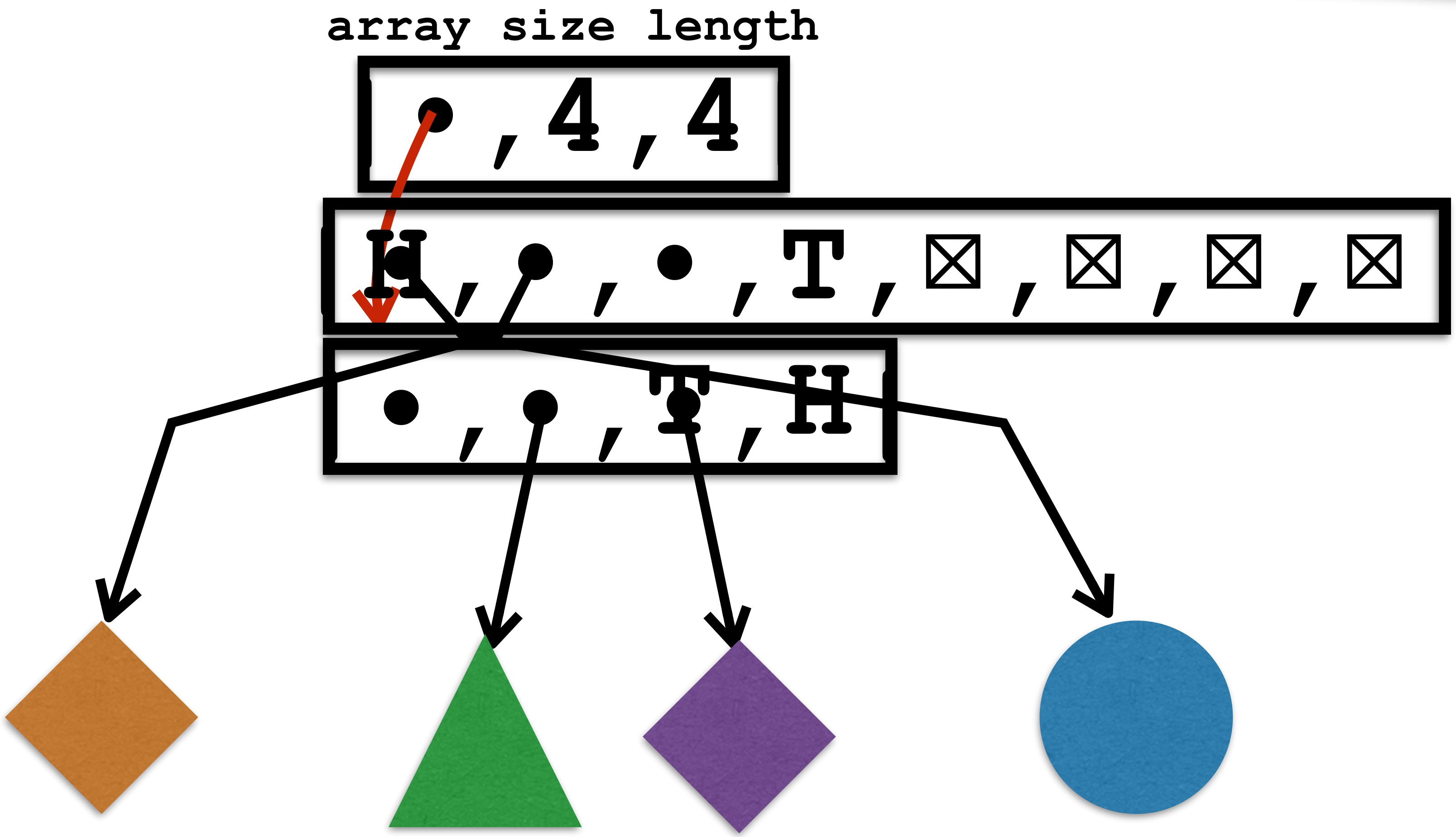




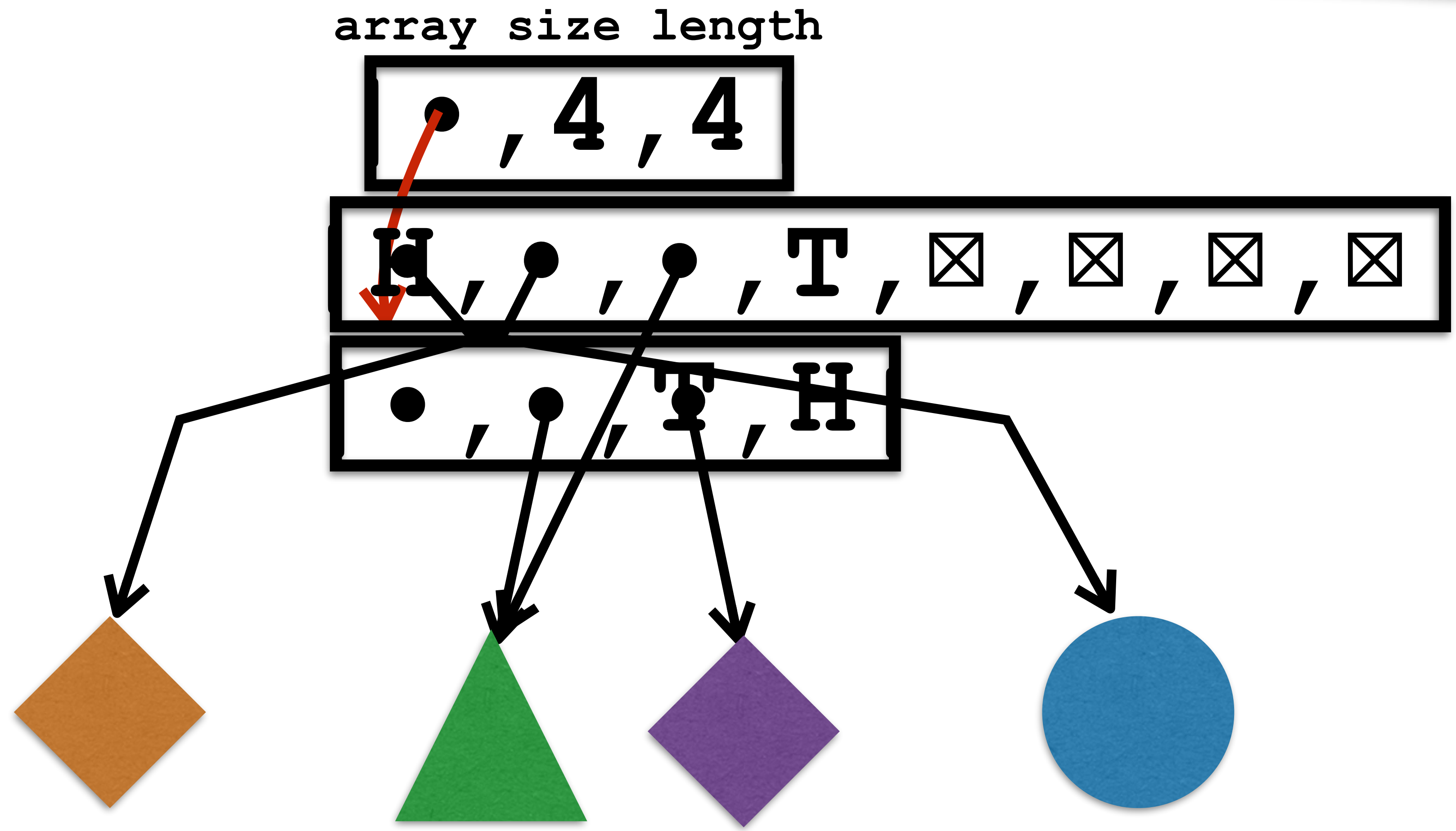
```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

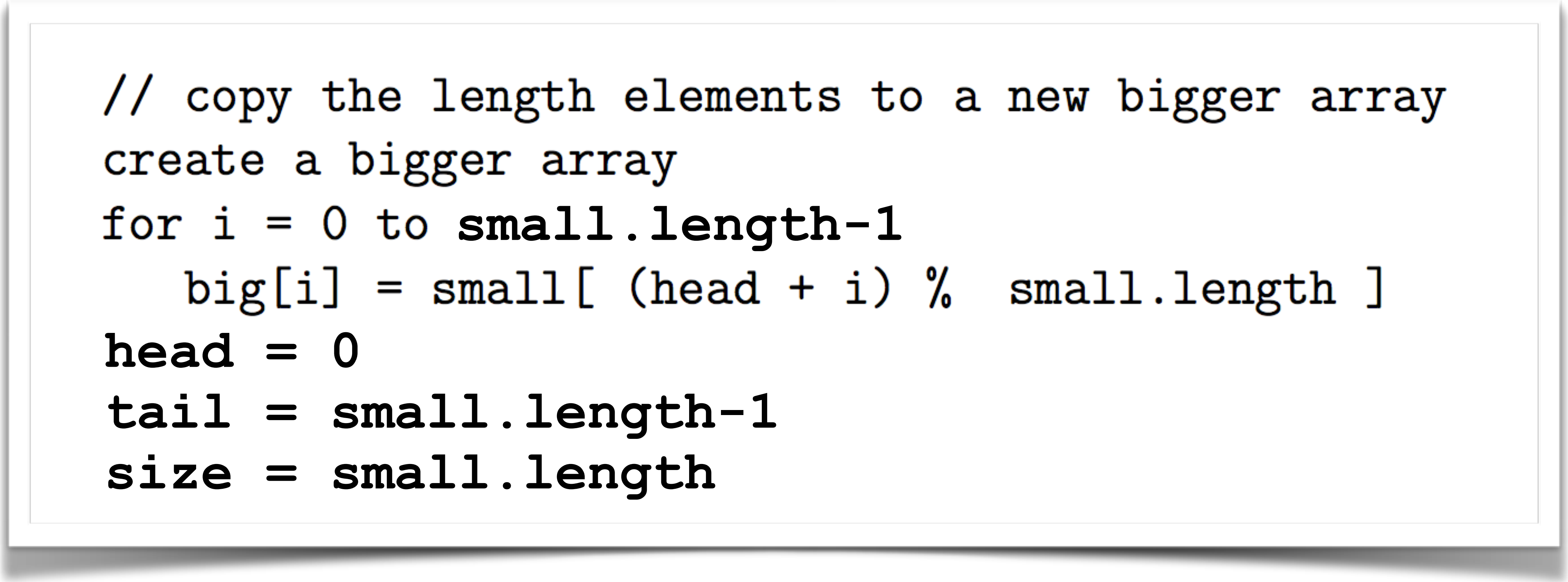


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

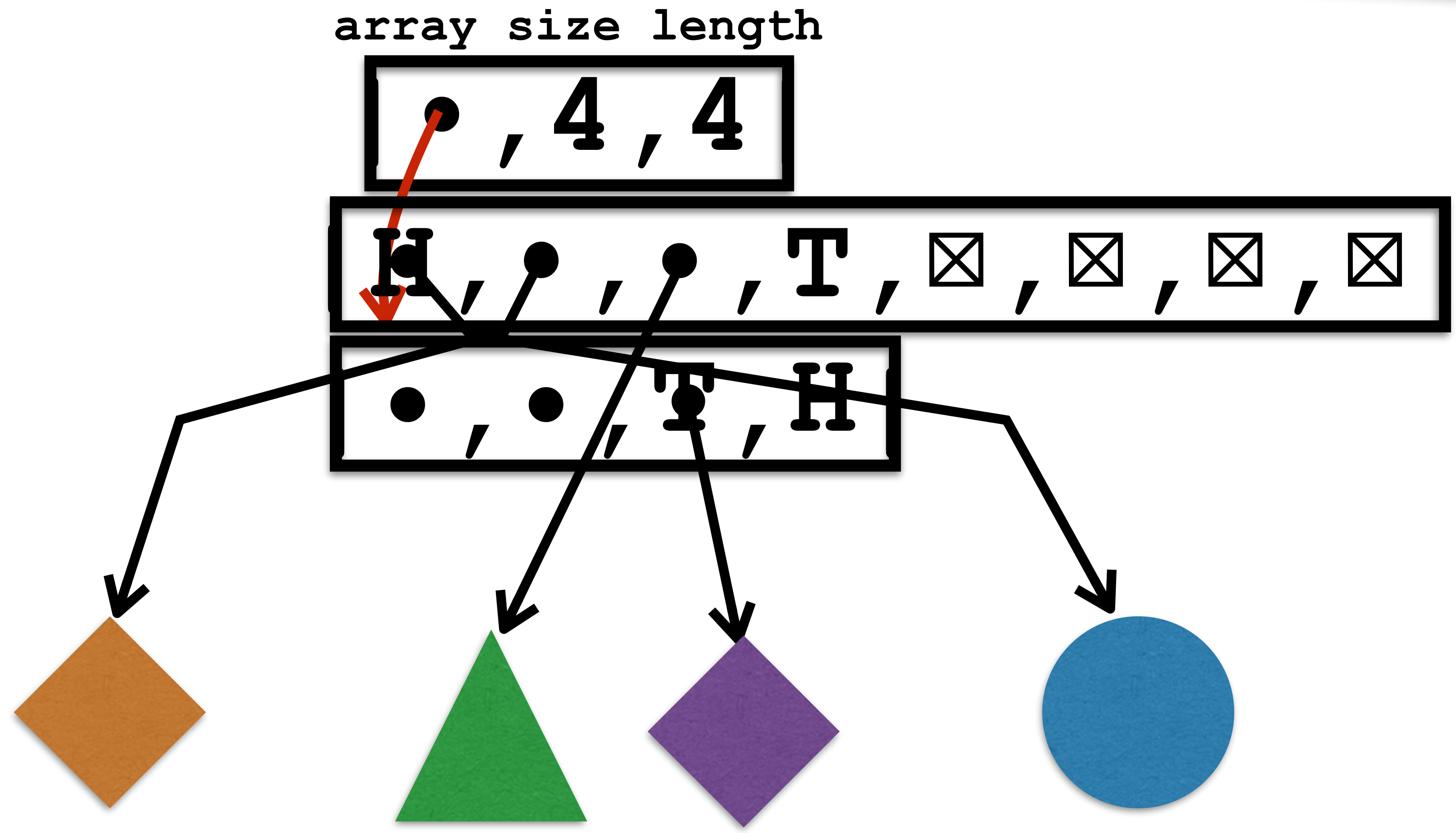


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
```

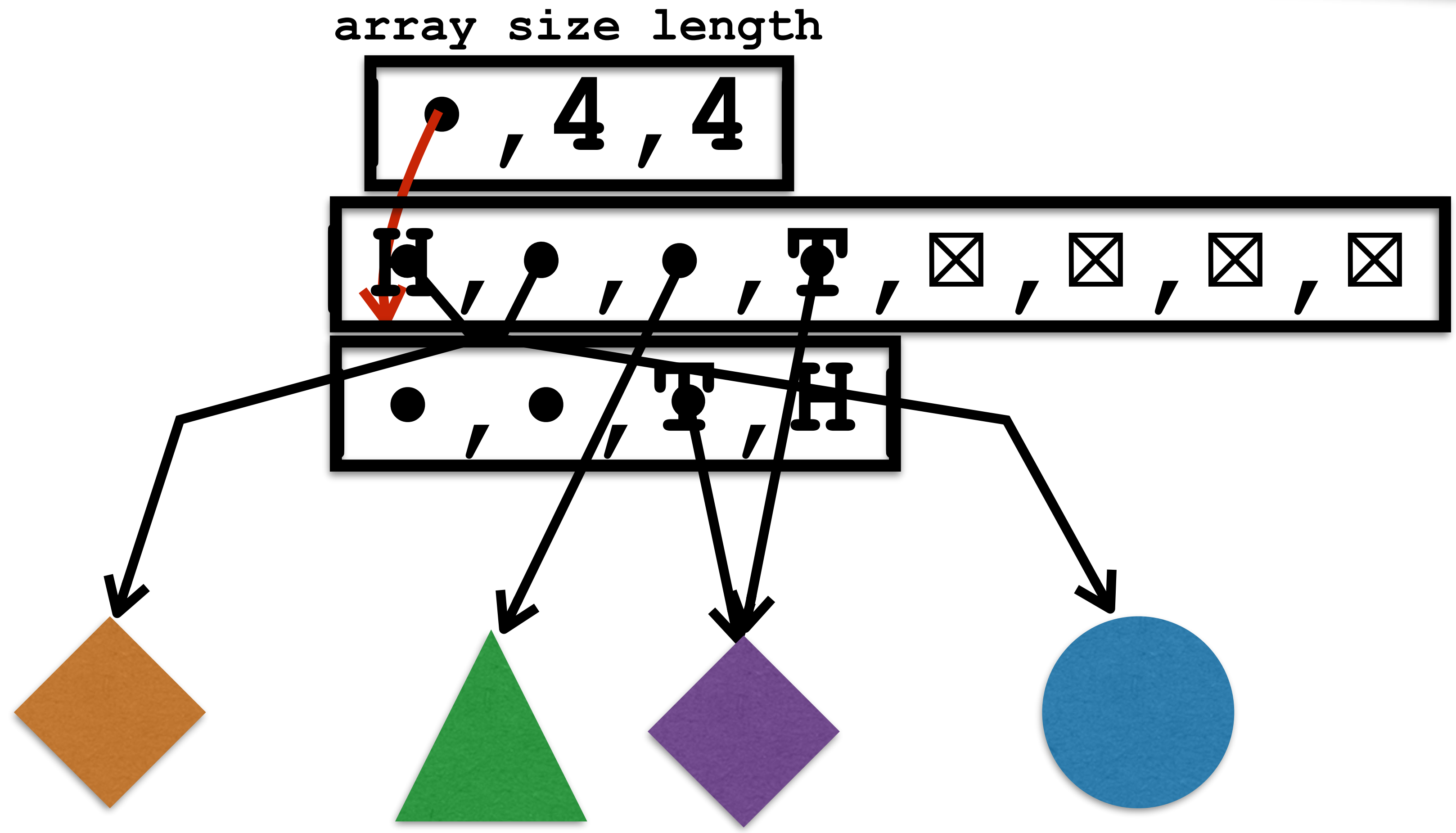


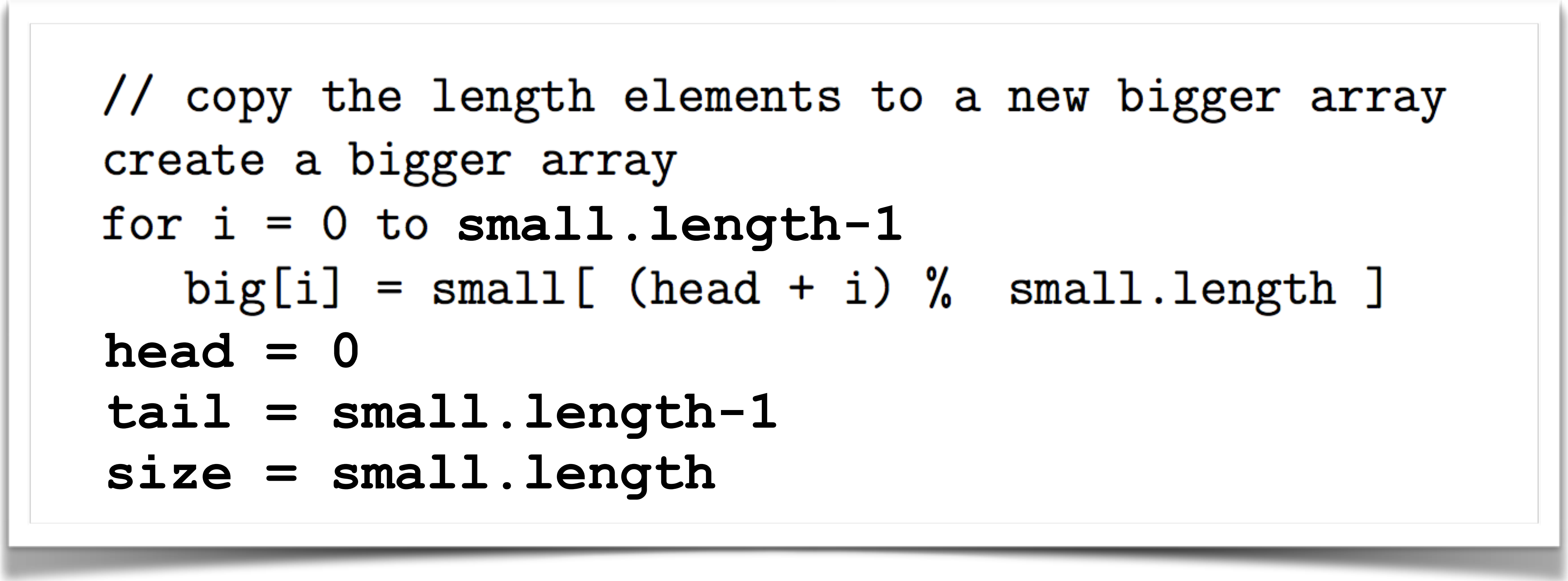


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

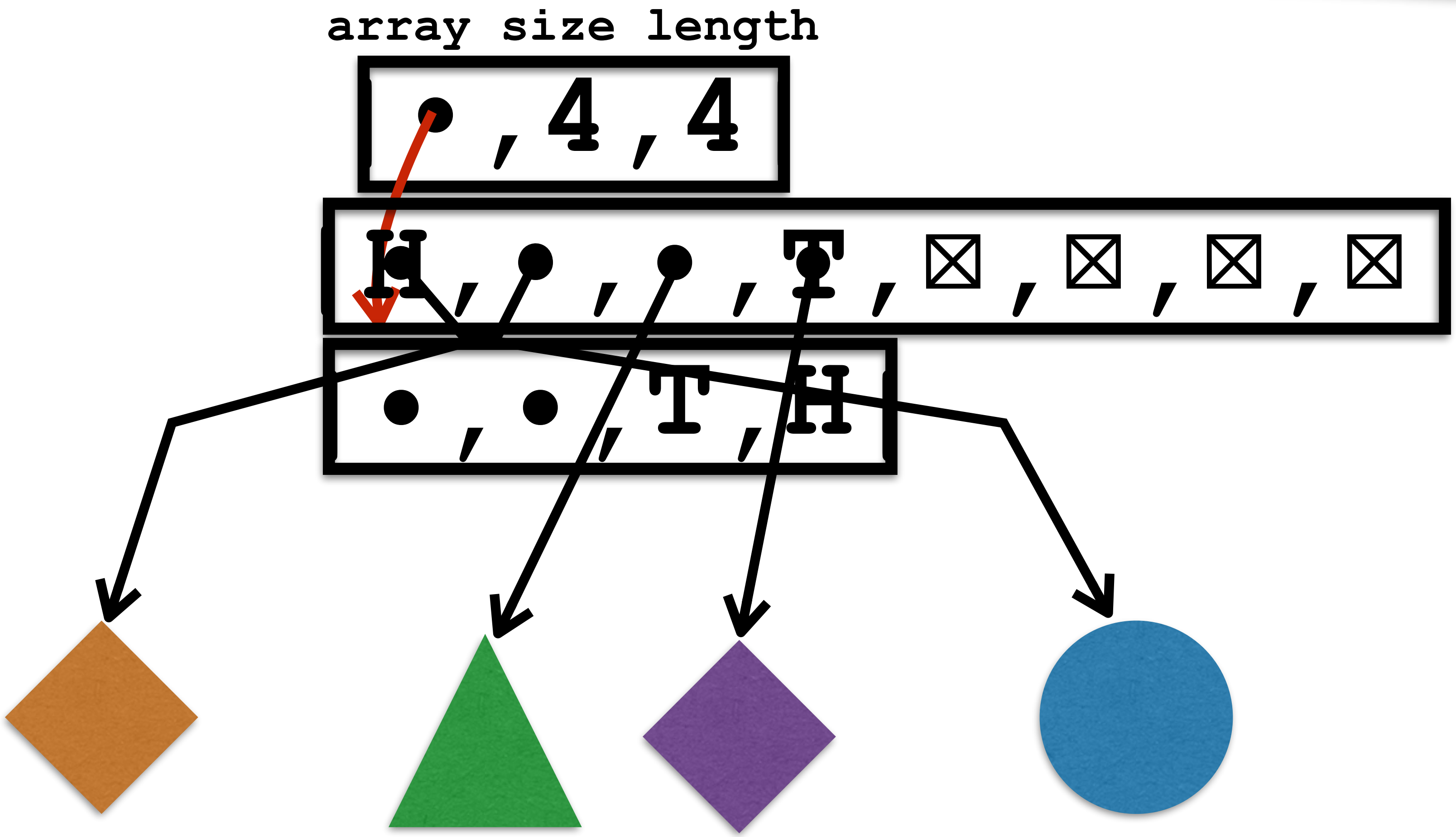


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

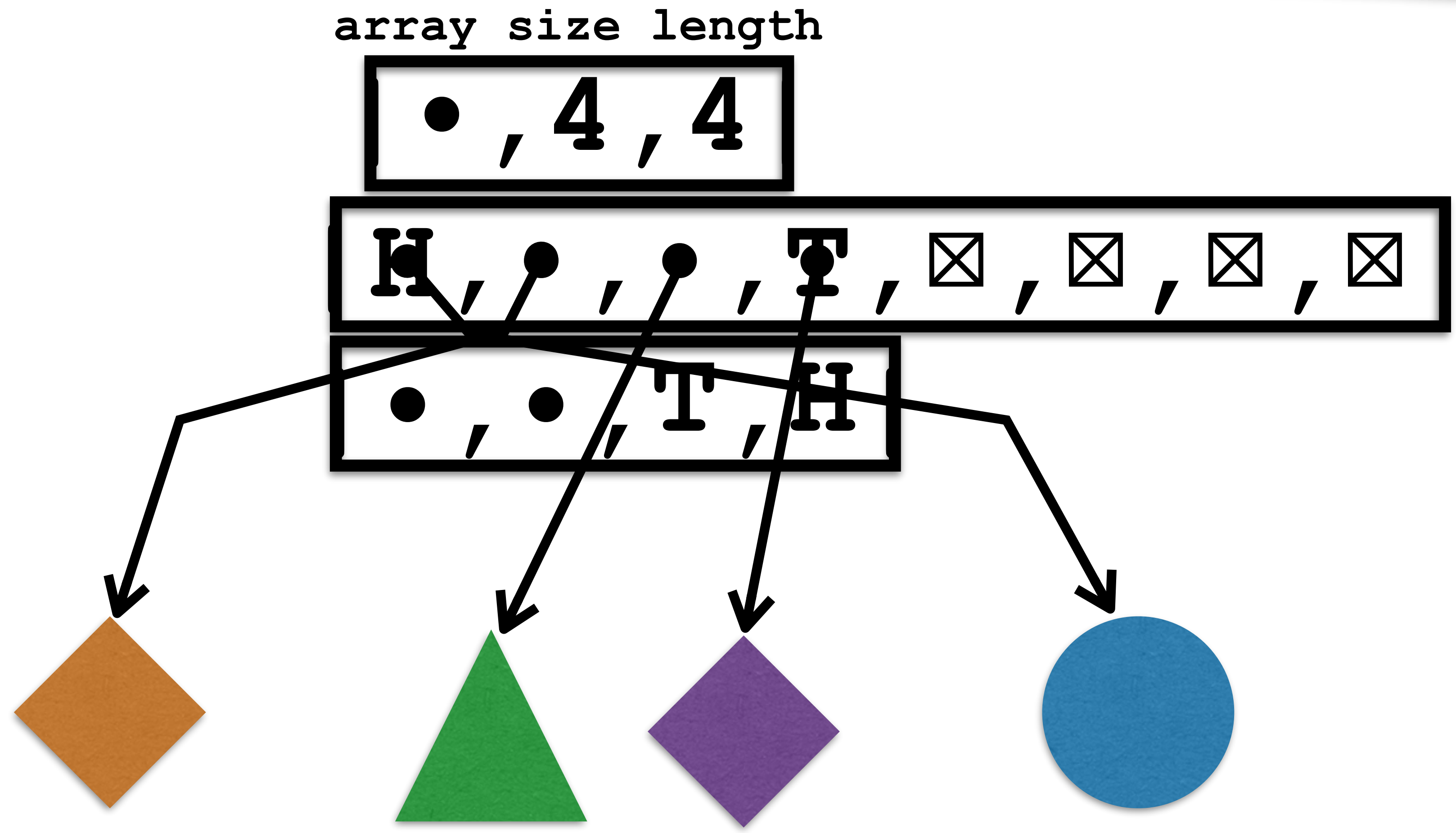


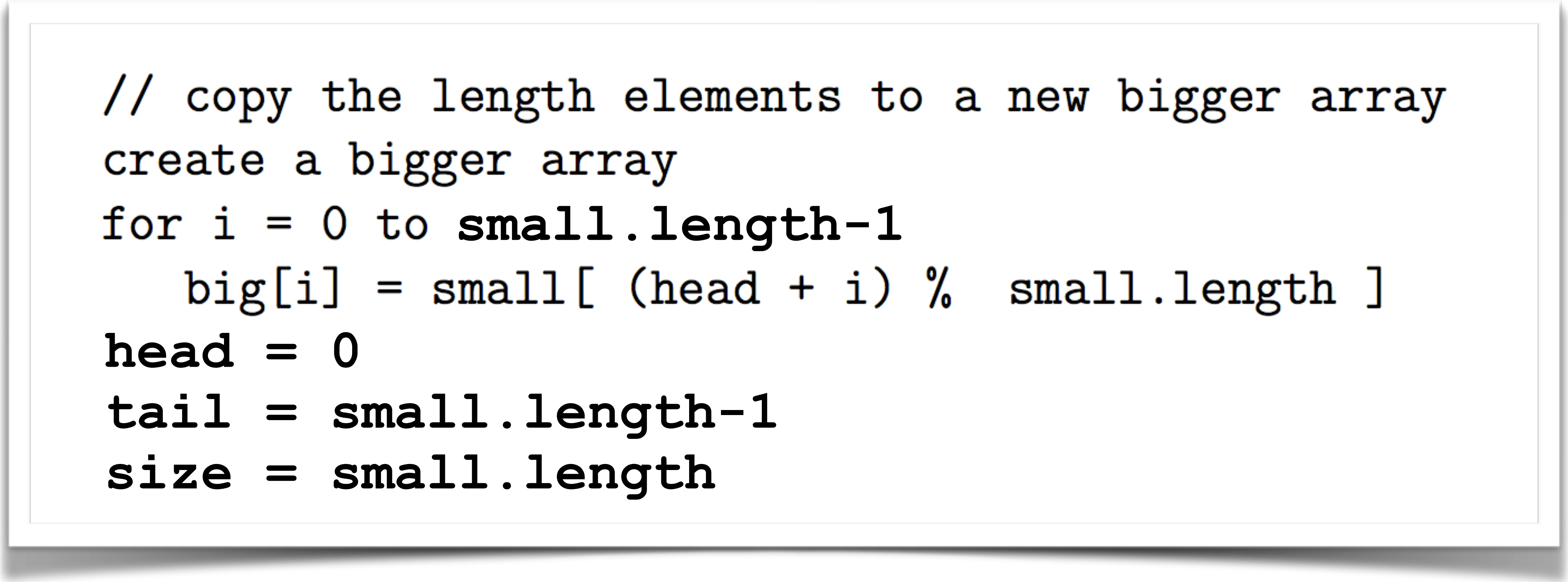


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

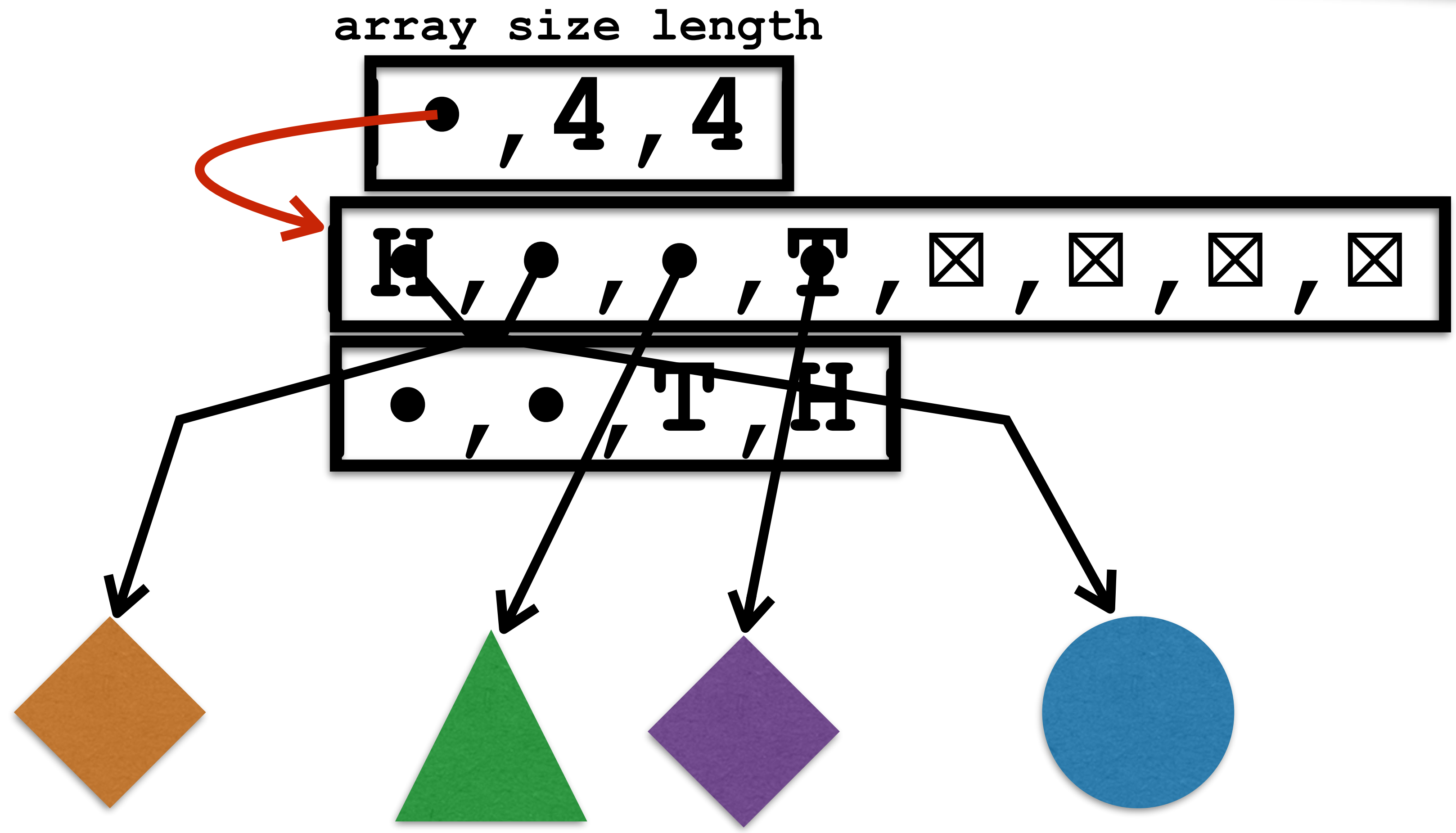


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) % small.length ]
head = 0
tail = small.length-1
size = small.length
```

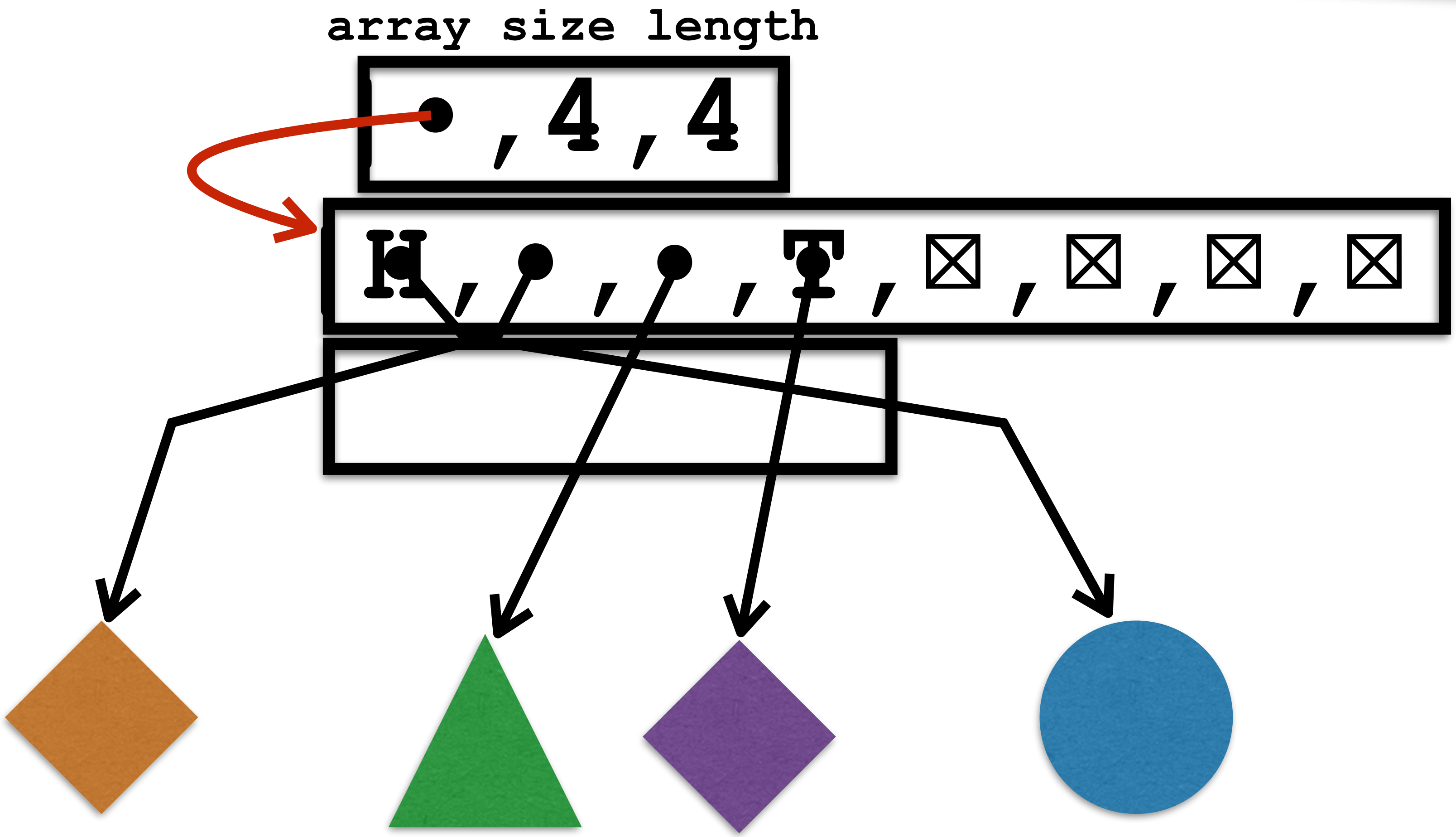


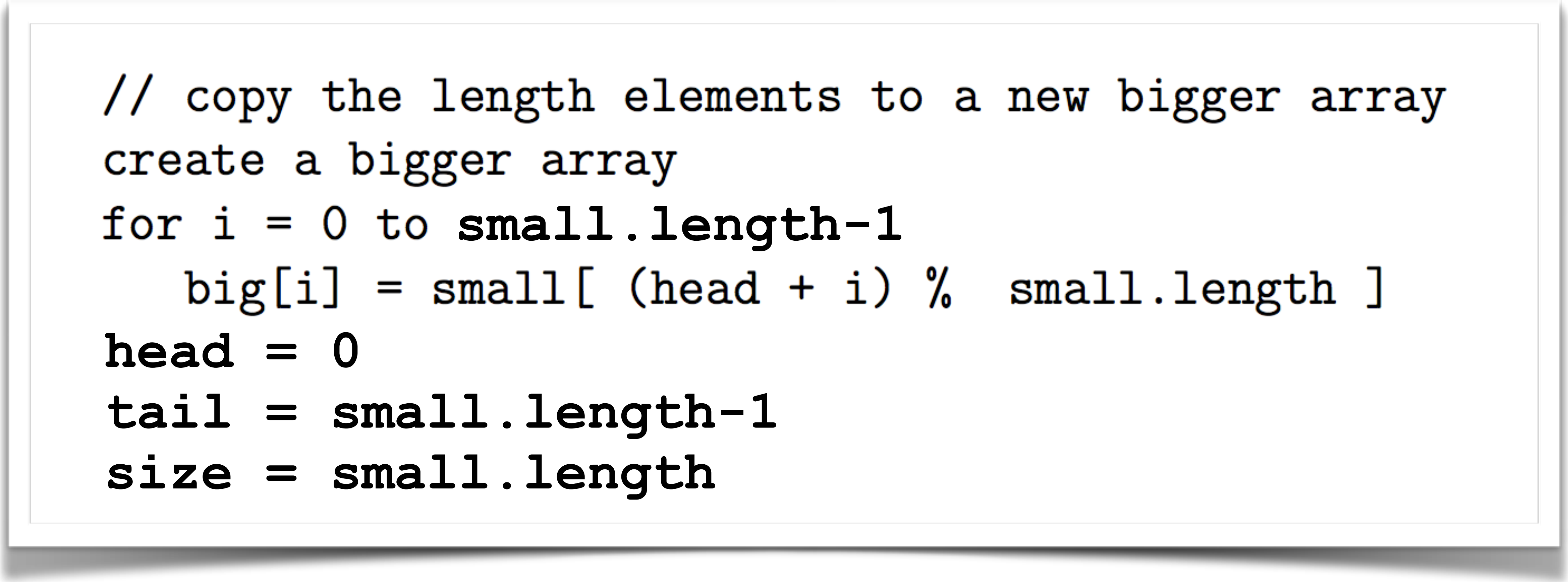


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

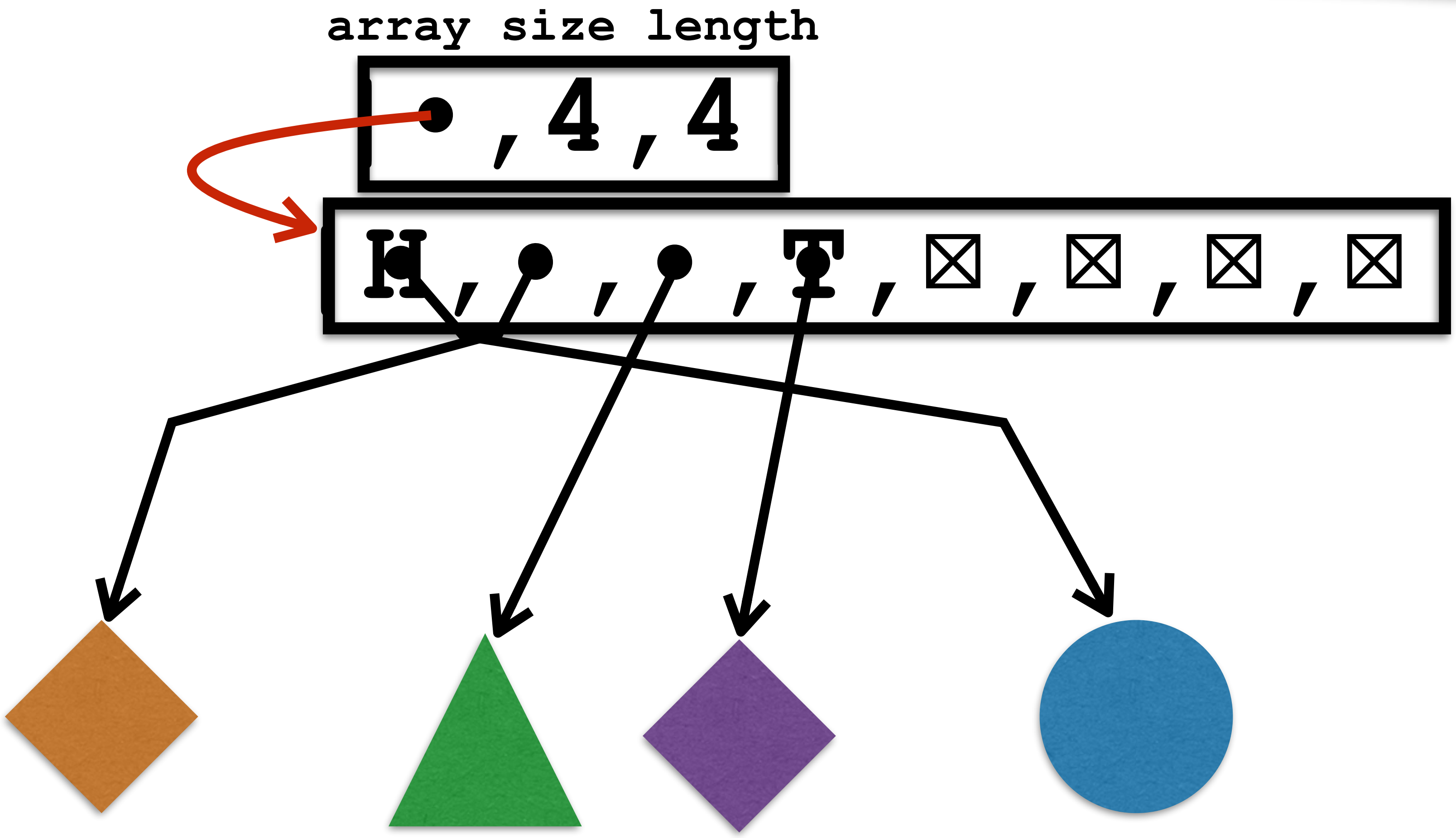


```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

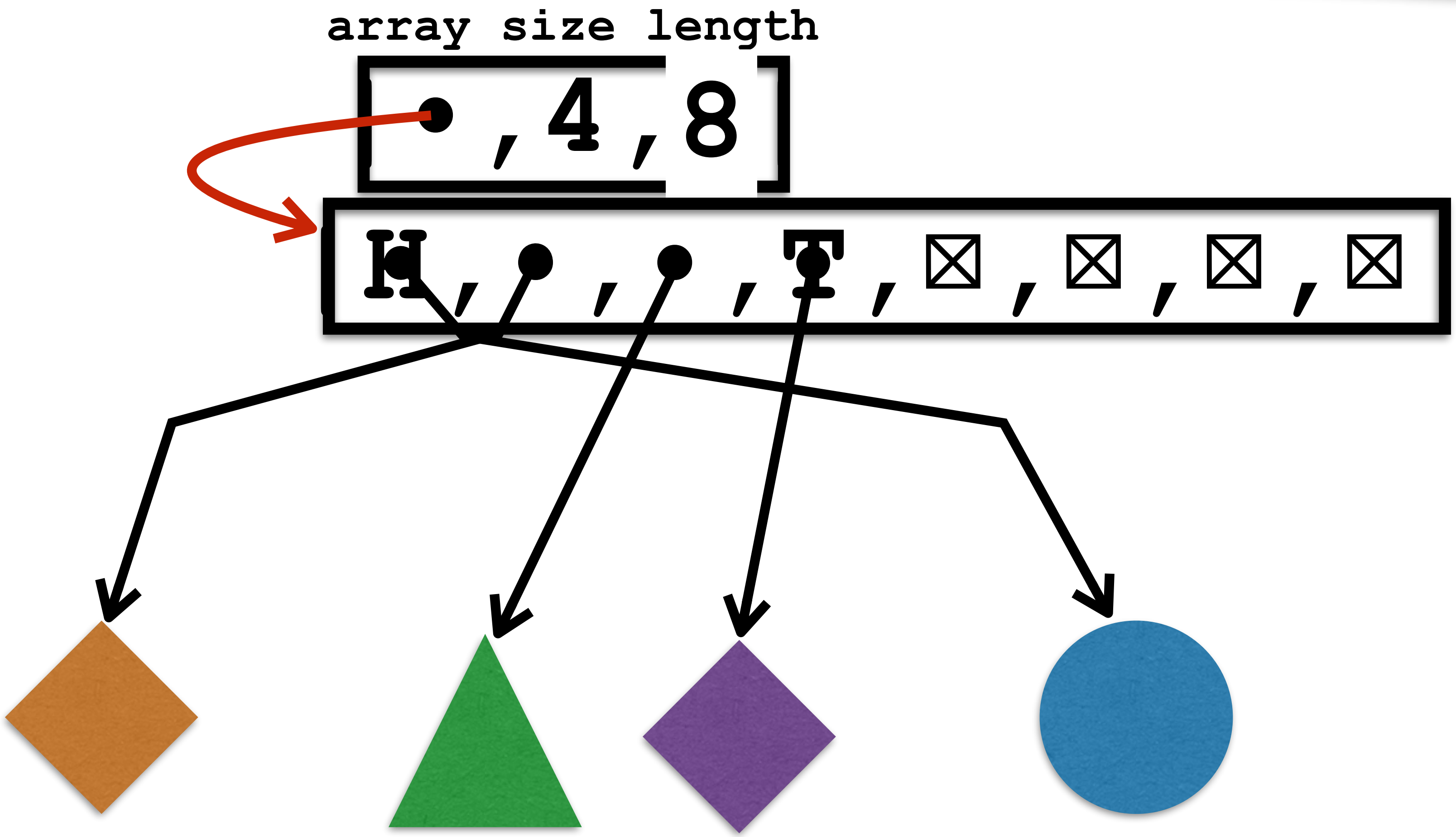




```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
   big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```



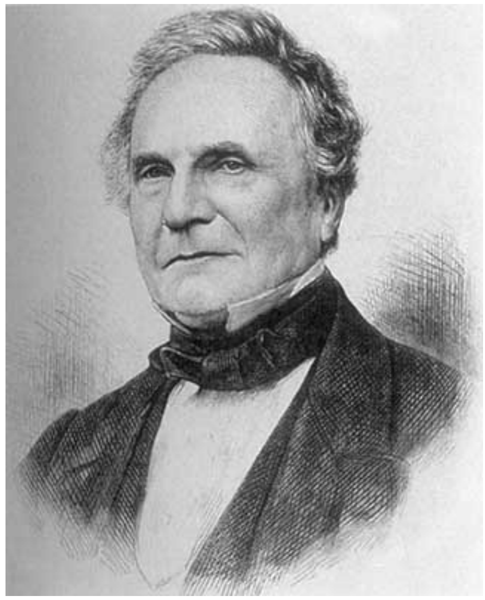
```
// copy the length elements to a new bigger array
create a bigger array
for i = 0 to small.length-1
    big[i] = small[ (head + i) %  small.length ]
head = 0
tail = small.length-1
size = small.length
```

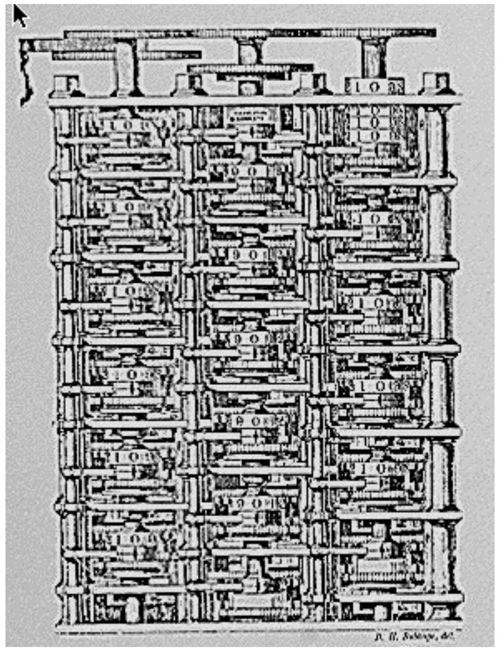


# Running Times and Asymptotic Notation

# Computational Tractability

> As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?
>
> \- Charles Babbage


Charles Babbage (1864)


Analytic Engine (schematic)

# Computational Tractability

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that tries every possible solution.

- Typically takes $2^N$ time or worse for inputs of size N.
- Unacceptable in practice.

↖ even worse : N ! for some problems

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants a > 0 and d > 0 such that on every input of size N, its running time is bounded by $a N^d$ steps.

**Def.** An algorithm is poly-time if the above scaling property holds.

↖ choose $C = 2^d$

# Worst Case Analysis

# Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

# Winter 2016
# COMP-250: Introduction to Computer Science

Lecture 8, February 4, 2016