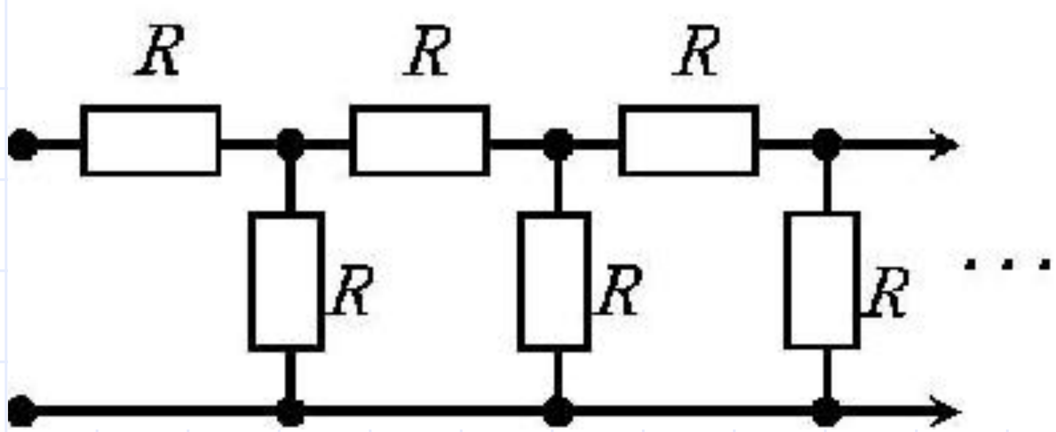
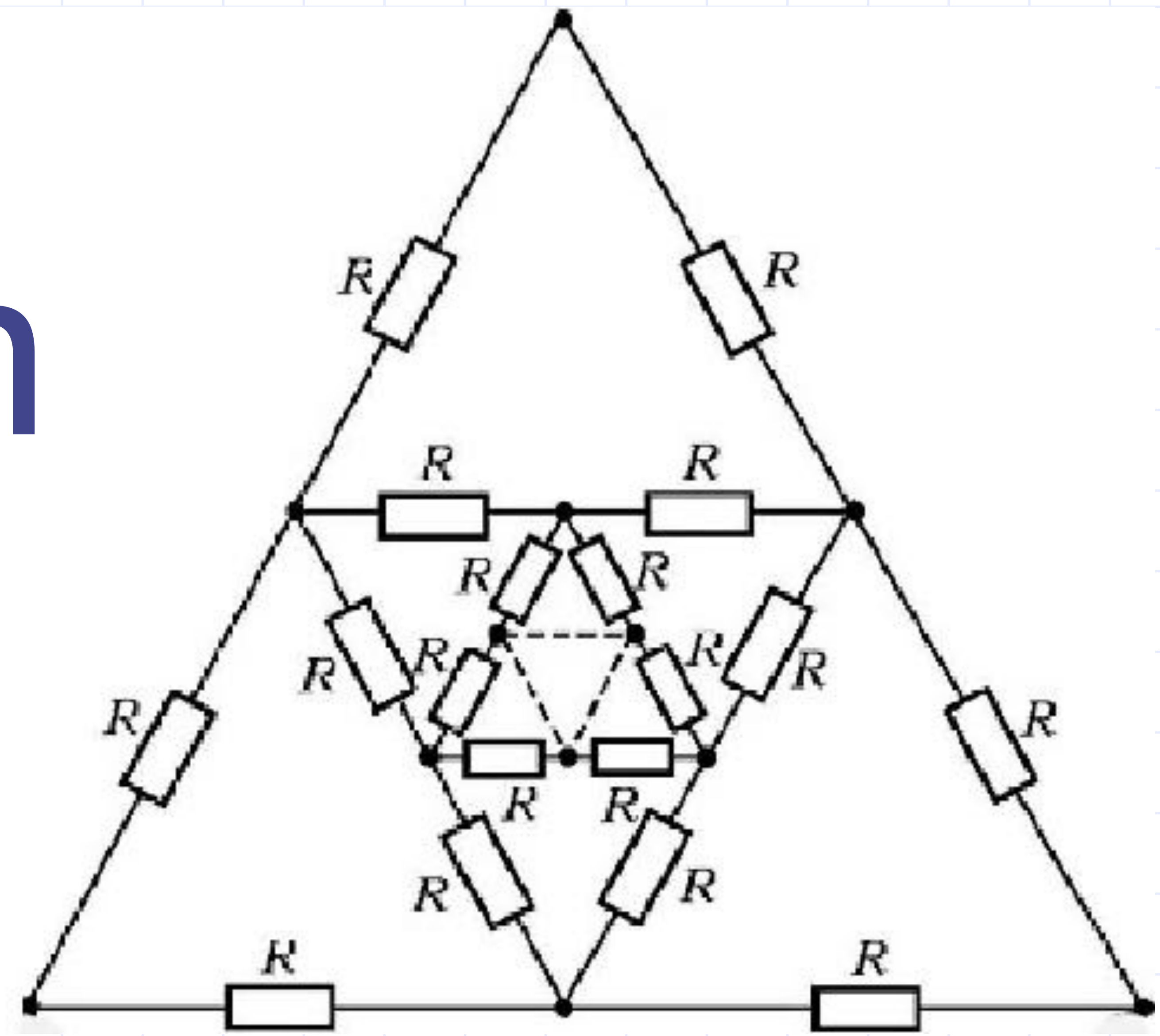
ESC101: Introduction to Computing

Recursion





Predict the output of the following code

```
#include <stdio.h>
int fun( int a[], int n)
{
   if( n == 1)
      if(a[0]%2 == 0)
         return -a[0];
      else
         return a[0];
   else
      if(a[n-1]%2 == 0)
         return -a[n-1]*fun(a, n-1);
      else
         return a[n-1]*fun(a,n-1);
}

int main()
{
   int a[]={10,4,25};
   int b[]={5,8,25};
   printf("%d\n",fun(a, 3) );
   printf("%d\n",fun(b, 3) );
   return 0;
}
```

Predict the output of the following code

```
#include <stdio.h>
int fun( int a[], int n)
{
   if( n == 1)
      if(a[0]%2 == 0)
         return -a[0];
      else
         return a[0];
   else
      if(a[n-1]%2 == 0)
         return -a[n-1]*fun(a, n-1);
      else
         return a[n-1]*fun(a,n-1);
}

int main()
{
   int a[]={10,4,25};
   int b[]={5,8,25};
   printf("%d\n",fun(a, 3) );
   printf("%d\n",fun(b, 3) );
   return 0;
}
```

Output
1000
-1000

Array's Maximum, once again

a

25	4	3	7	5	23	-3	9

max_array(a,8)

alls max_array(a+1,7)

calls max_array(a+2,6)

calls max_array(a+3,5)

calls max_array(a+4,4)

calls max_array(a+5,3)

calls max_array(a+6,2)

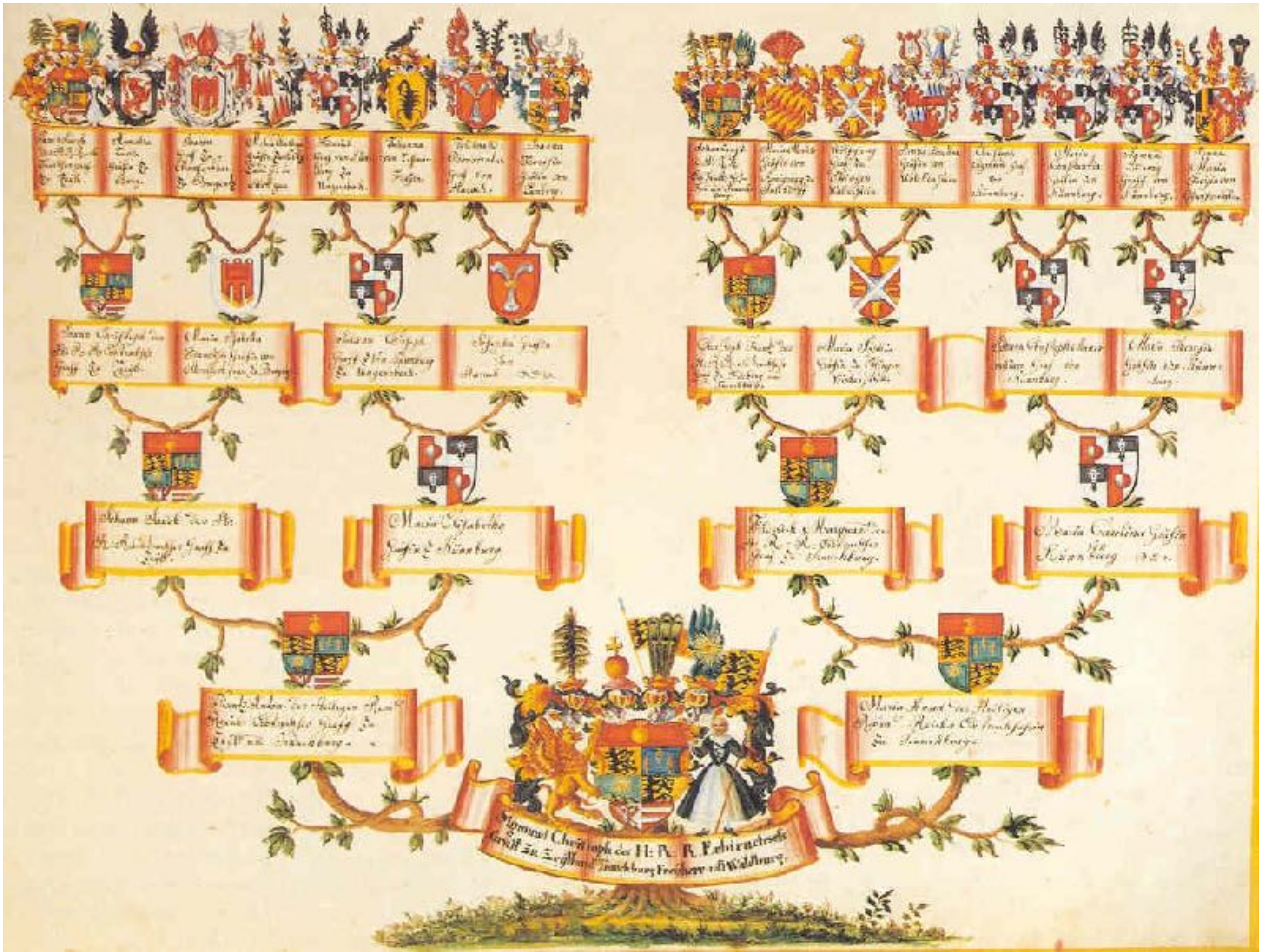
calls max_array(a+7,1)

```
int max_array(int a[], int n) {
    int maxval;
    if (n == 0) return -99999;
    if (n==1) return a[0];
    maxval = max_array(a, n-1);
    return max(a[n-1],maxval);
}
```

Stack Depth is n

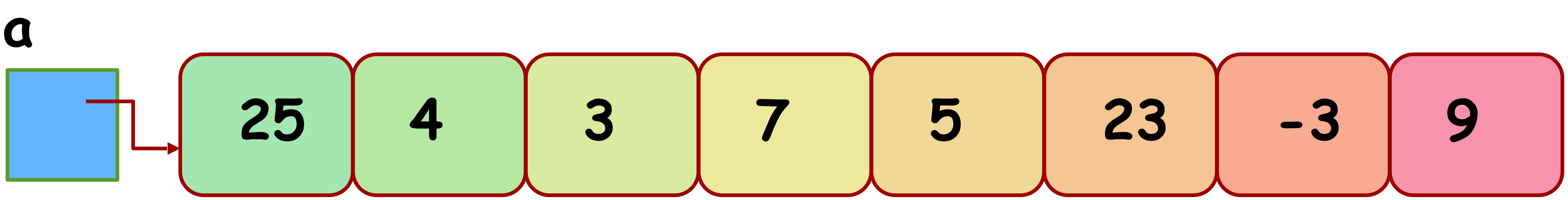
Can we reduce the stack depth?

Recursion II – Two-way Recursion

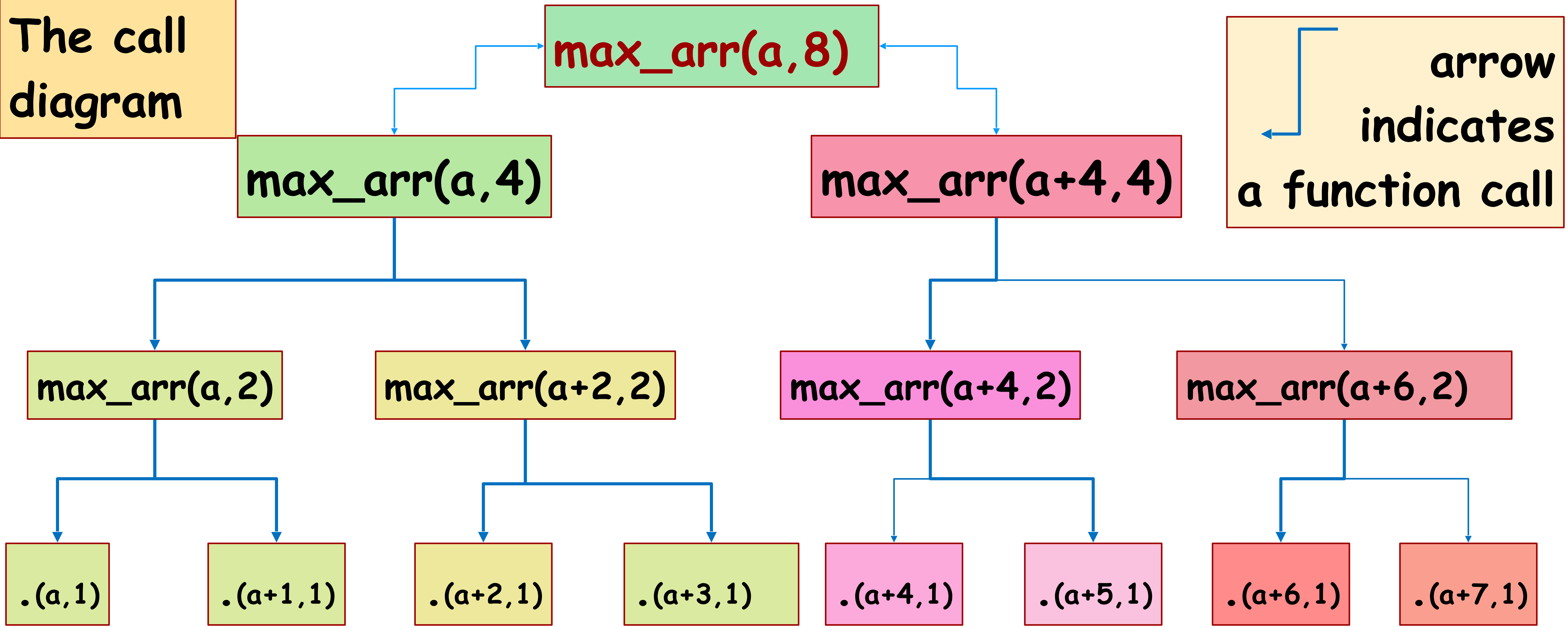


Can we reduce the stack depth?

1. Divide the array a into about two equal halves: a[s … e/2 -1] and a[e/2 … n-1].
2. Recursively find the maximum element in each half and return the larger of the two maxima.
3. As before: recursion exits when n is 0 or n is 1.
4. If n is 1 then return end element, if n is 0 return –INFTY.



The call diagram



Stack depth (length of the longest path in call stack) ≈ 1+log n.

```c
#include <stdio.h>
#define MNEG -9999
int max( int a, int b)
{
   if(a>b) return a;
   return b;
}

int maxarray(int a[], int s, int e)
{
   int maxv;
   if( e==s )
      return a[s];
   if( (s>e))
      return MNEG;
   maxv=max( maxarray(a,s,e/2),maxarray(a,e/2, e) );
   return maxv;
}

int main()
{
   int arr[]={10,100,4,20,45,56,72,43,33};
   printf("maxarray = %d\n",maxarray(arr, 0, 8) );
   return 0;
}
```

```c
#include <stdio.h>
#define MNEG -9999
int max( int a, int b)
{
   if(a>b) return a;
   return b;
}

int maxarray(int a[], int s, int e)
{
   int maxv;
   if( e==s )
      return a[s];
   if( (s>e))
      return MNEG;
   maxv=max( maxarray(a,s,e/2),maxarray(a,e/2, e) );
   return maxv;
}

int main()
{
   int arr[]={10,100,4,20,45,56,72,43,33};
   printf("maxarray = %d\n",maxarray(arr, 0, 8) );
   return 0;
}
```

WRONG

```c
#include <stdio.h>
#define MNEG -9999
int max( int a, int b)
{
   if(a>b) return a;
   return b;
}

int maxarray(int a[], int s, int e)
{
        int m = (e+s)/2;
        if( e==s )
                return a[s];
        if( (s>e))
                return MNEG;
        return  max( maxarray(a, s,m), maxarray(a, m+1, e) );
}
int main()
{
   int arr[]={10,100,4,20,45,56,72,43,33};
   printf("maxarray = %d\n",maxarray(arr, 0, 8) );
   return 0;
}
```

Estimating the Time taken

- Two types of operations
  - Function calls
  - Other operations (call them simple operations)
- Assume each simple operation takes fixed amount of time (1 unit) to execute
  - Really a very crude assumption, but will simplify calculations
- Time taken by a function call is proportional to the number of operations performed by the call before returning.

Recursive search

Task: Given a key, return 1 if it is in an integer array or -1 if not

```
Function find_key(int a[], int key, int n)
if (n == 0)
    return -1;
if (a[n-1] == key)
    return 1;
else
   return find_key(a,key, n-1);
```

Estimating the Time taken

- Search1

```
1. if (n == 0) return 0;
2. if (a[n-1] == key) return 1;
3. return find_key(a, key, n-1);
```

  - $T(n)$ denote the time taken by search on an array of size n.

$$T(n) = T(n-1) + C \,,\ T(0) = C$$

$$T(n) = Cn$$

  - The worst case run time of find_key() is proportional to the size of array
    - Bigger the array, slower the search
  - What is the best case run time?
  - Which one is more important to consider?

Estimating the Time taken

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
return find_key(a, start, mid-1, key)
        || find_key(a, mid+1, end, key);
```

- Search2
  - Recurrence?

$$T(n) \leq T(n/2) + T(n/2) + c$$

  - Solution?

$$T(n) \propto n$$

  - The worst case run time of Search2 is also proportional to the size of array
    - Can we do better?

Can we search faster?

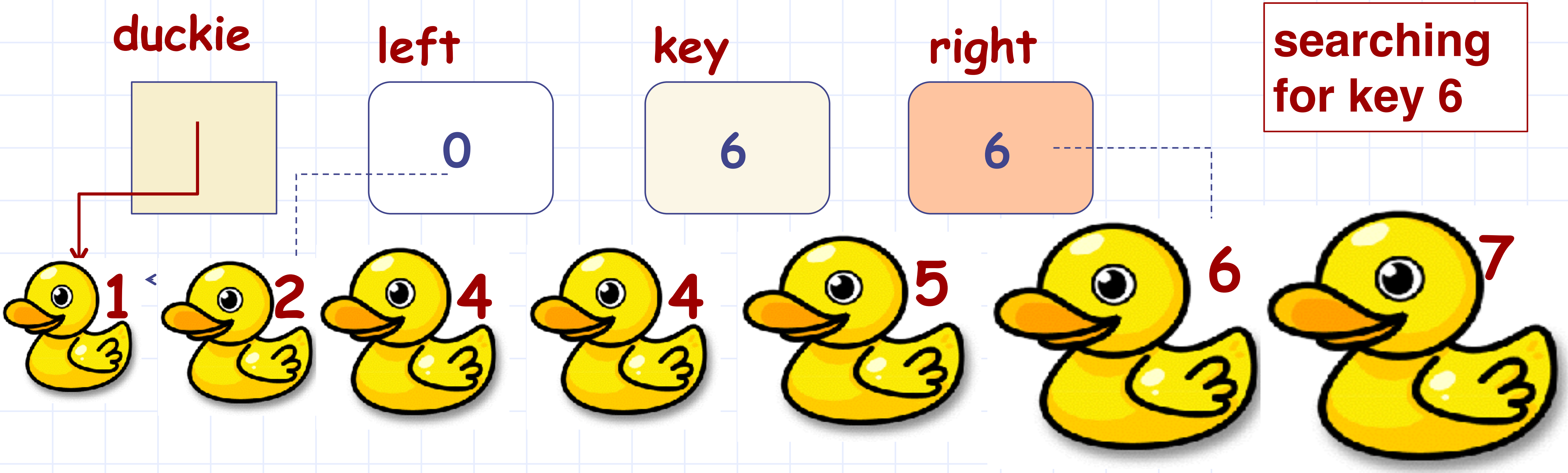
- Yes, provided the elements in the array are sorted
  - in either ascending or descending order

Let us take an example. We have an array of numbers, sorted in non-descending order.

int duckie [] = {1,2,4,4,5,6,7};

some numbers can be repeated, like 4 in duckie[]

To illustrate the idea, consider searching for the number **6** in the array.



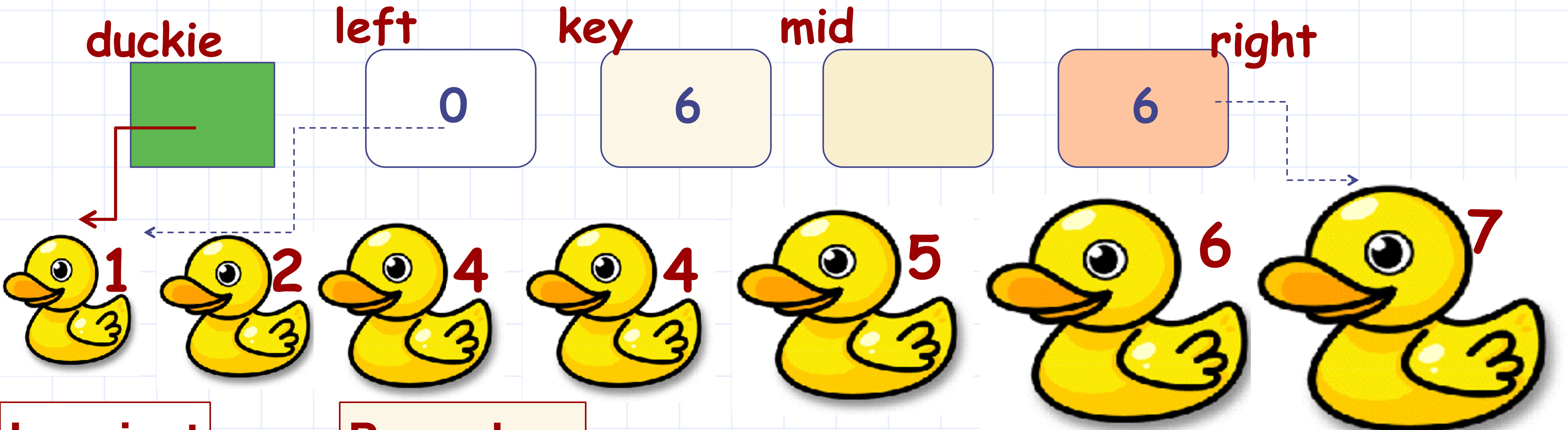
searching for key 6

Initialization

Keep two indices, left and right.
Initially left is 0 and right is the rightmost index in the array. Here right is 6.

Invariant

The key that is being searched for lies in between the indices left and right in the array duckie[] (both ends included), if at all it is in the array.



Invariant

The key lies between the indices left and right in the array duckie[], if at all it is in the array.

Procedure

Calculate the middle index of left and right

mid =(left + right)/2

Compare duckie[mid] with key. There are 3 possible outcomes.

BINARY SEARCH

duckie[mid] == key

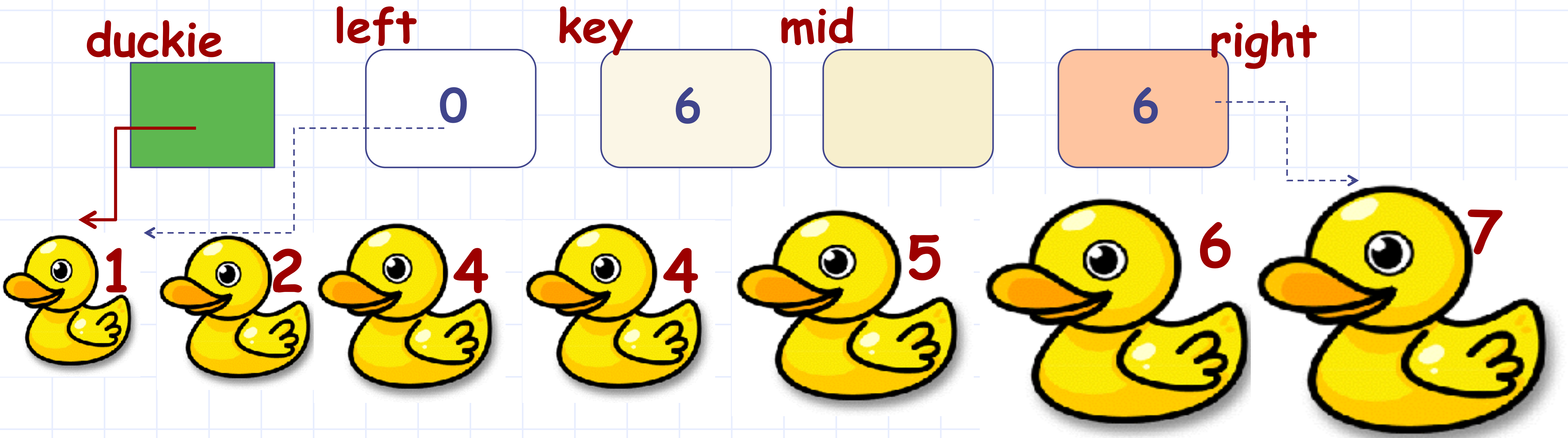
duckie[mid] == key
Key is found. return mid.

duckie[mid] < key

Key may lie between duckie[mid+1] and duckie[right].

duckie[mid] > key

Key may lie between duckie[left] and duckie[mid-1].



Let us trace the procedure on the duckie array

Calculate the middle index of left and right

mid =(left + right)/2

BINARY SEARCH

Compare duckie[mid] with key. There are 3 possible outcomes.

duckie[mid] == key

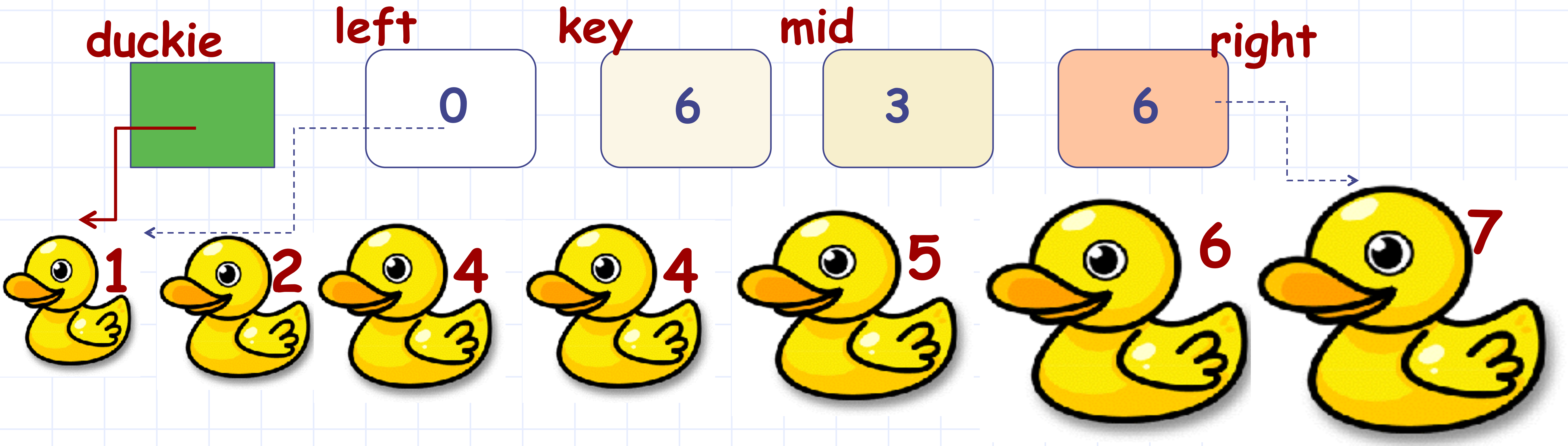
duckie[mid] == key
Key is found. return mid.

duckie[mid] < key

Key may lie between duckie[left] and duckie[mid-1].

duckie[mid] > key

Key may lie between duckie[mid+1] and duckie[right].



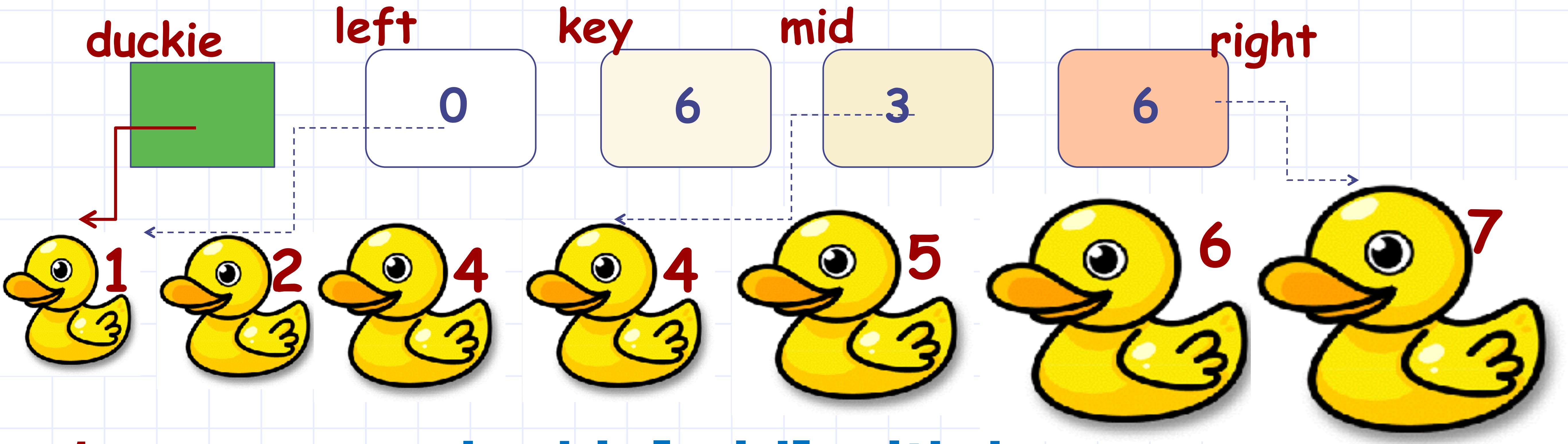
BINARY SEARCH

Calculate mid:

mid = (0 + 6)/2

mid is 3

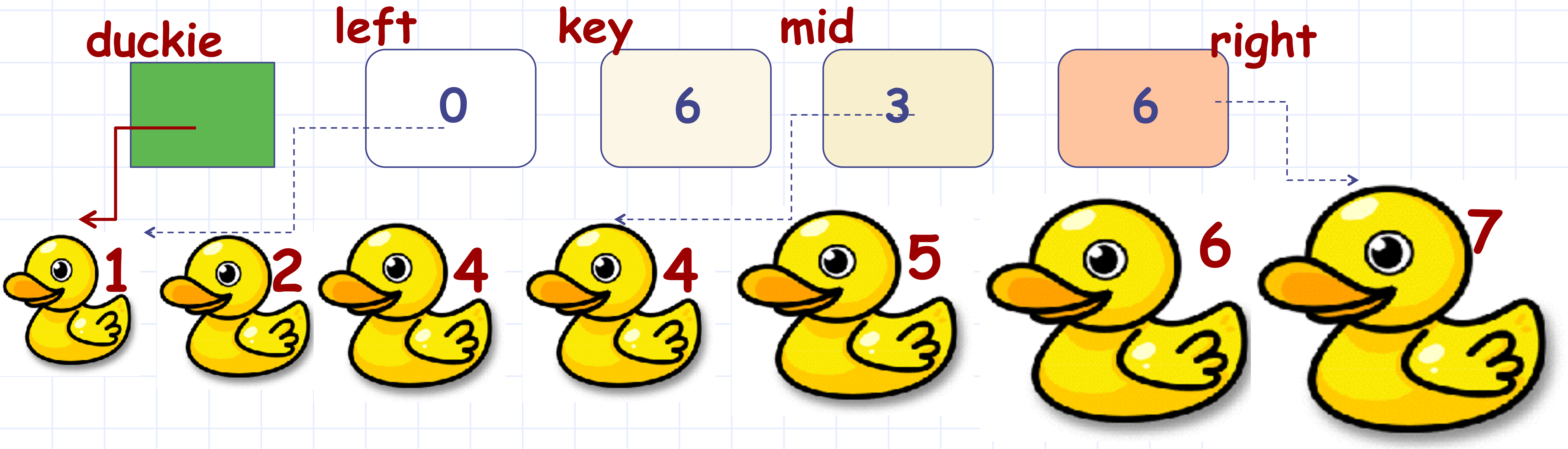
Compare duckie[3] with key.



1. compare duckie[mid] with key.
2. duckie[mid] is 4,key is 6.
3. 4 < 6 so key may only lie among duckie[mid+1] to duckie[right]

BINARY SEARCH

What to do now?

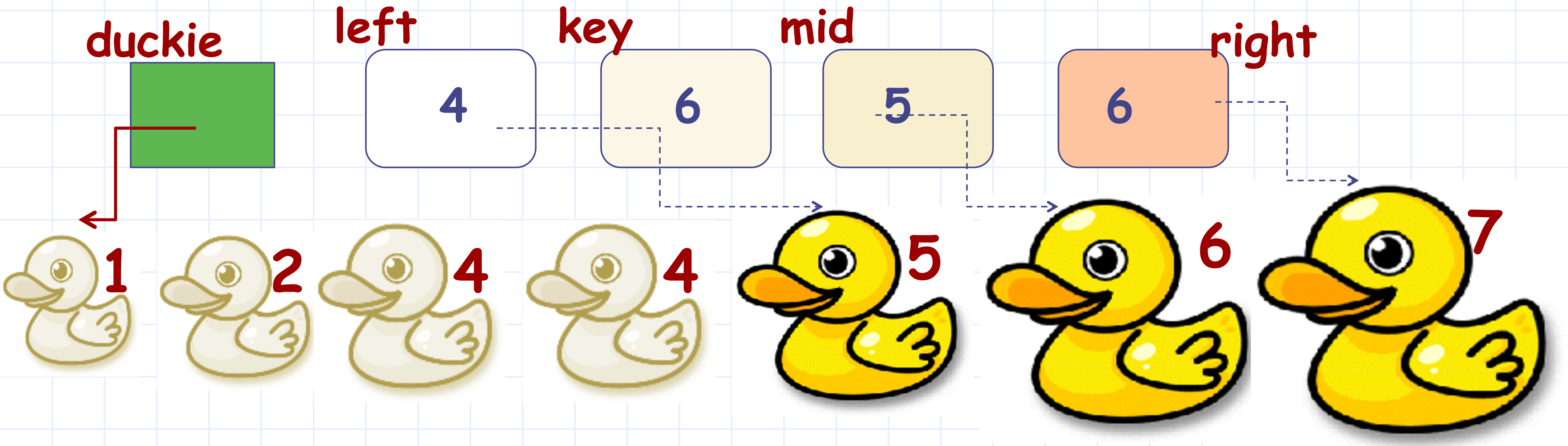


BINARY SEARCH

1. We know for sure that key does not lie in the index range 0..mid.

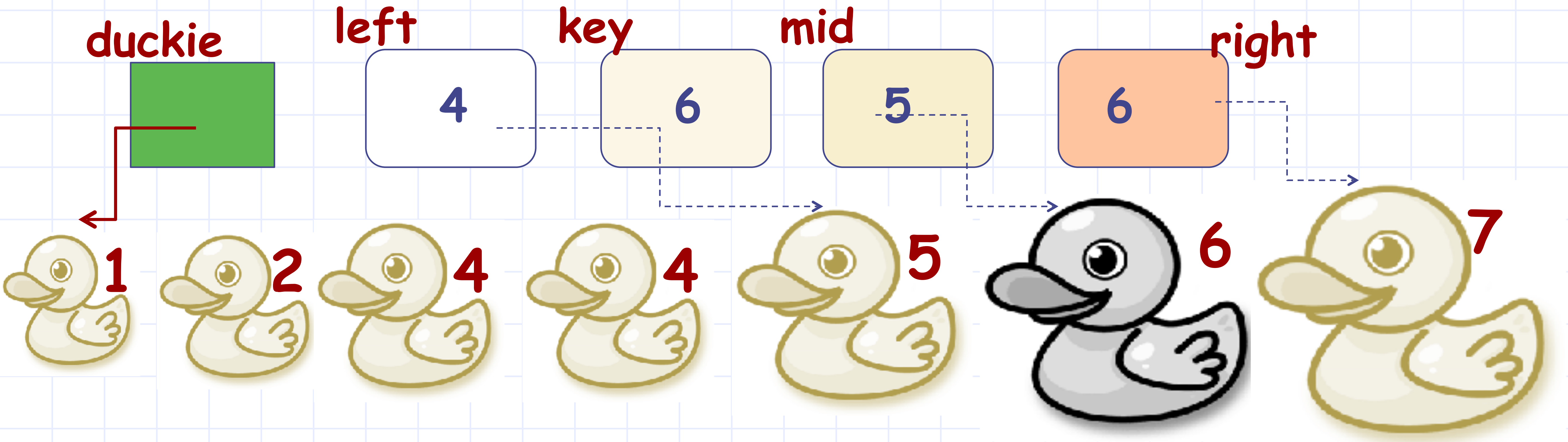
2. So we can set left to mid+1.

3. Repeat the process.



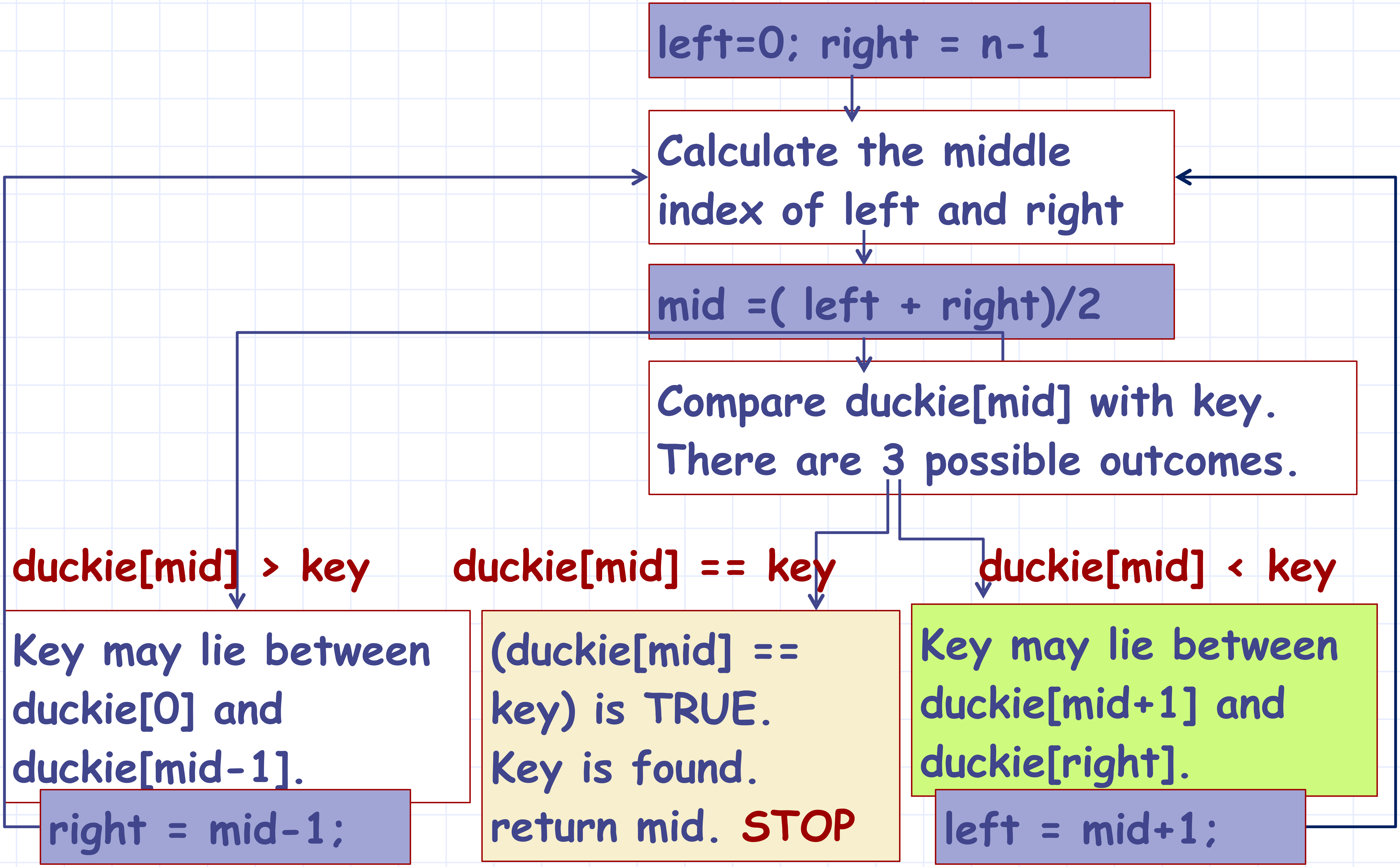
1. Continuing…
2. mid = (left+right)/2. So mid is 5.
3. Now duckie[5] is 6, so we have found the key.

BINARY SEARCH



BINARY SEARCH

1. We have found the key, so we can return the index 5.
2. Let us complete the flow of procedure.

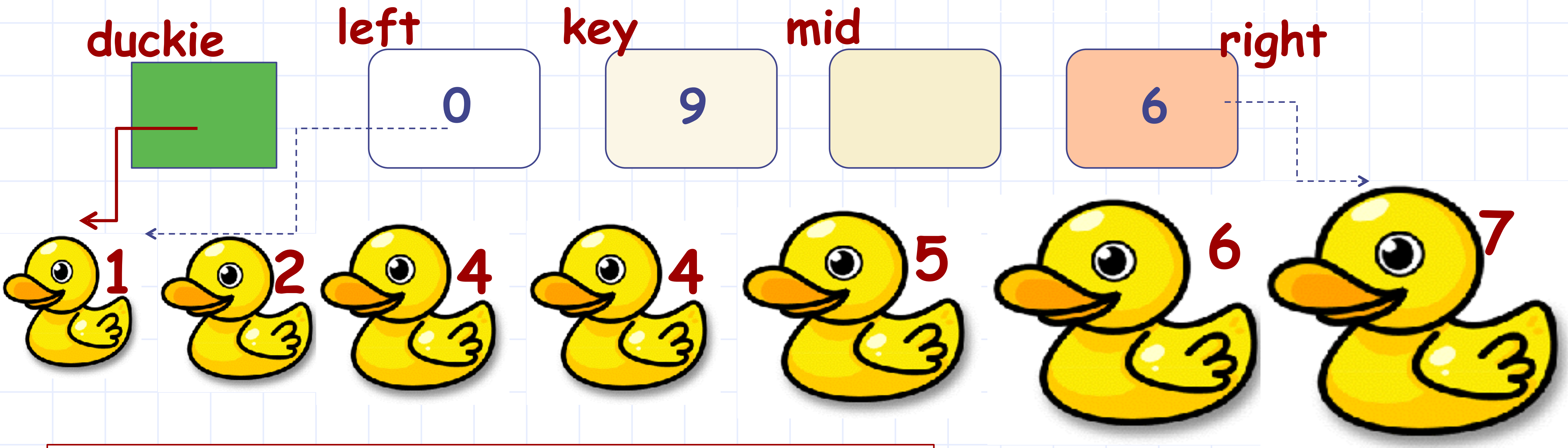


Something is still missing…

BINARY SEARCH

Let us search for 9 in this array.

an example of a search that does not succeed

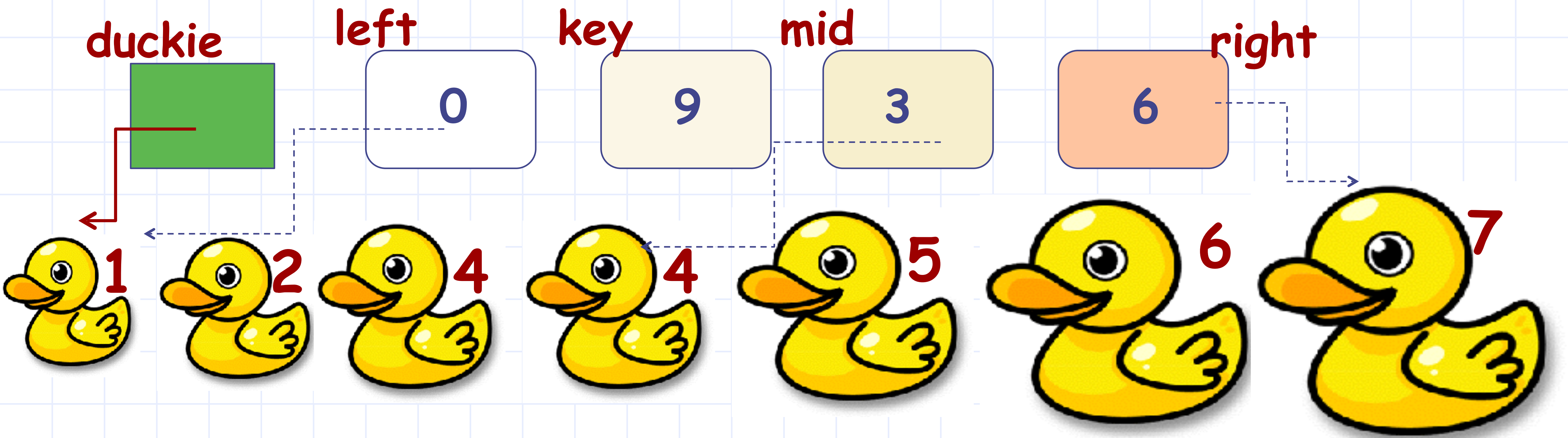


Key is 9

Initial values:

left is 0, mid is undefined, right is 6.

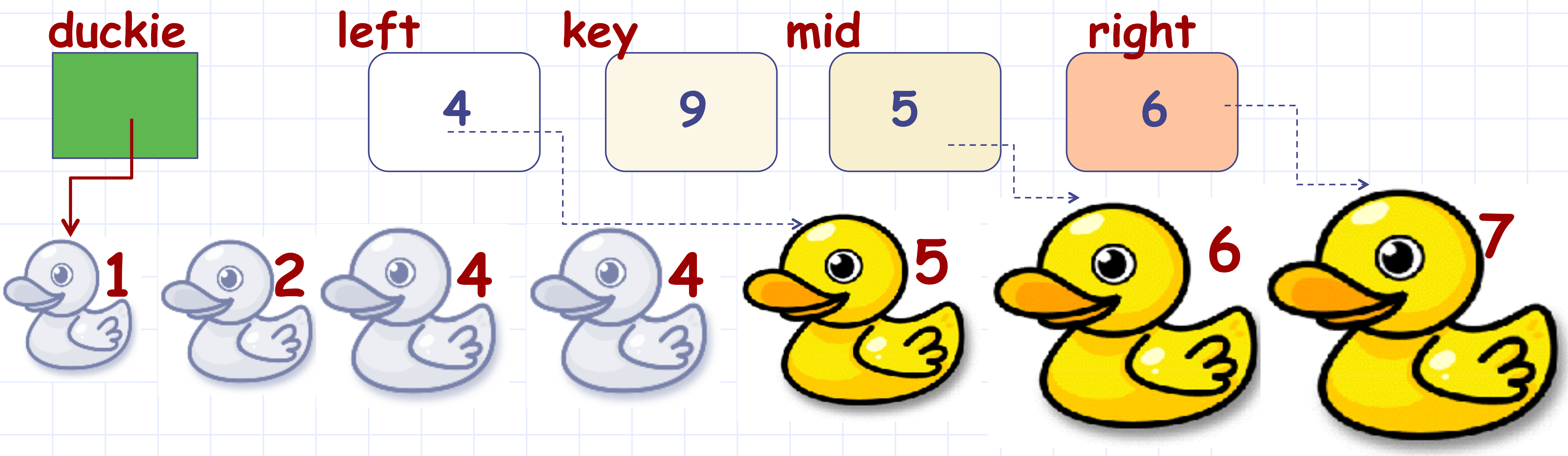
searching for 9.



Set mid to mid = (left+right)/2.
mid is 3. Compare duckie[mid] with key.
duckie[mid] is 4, key is 9 and 4<9.

So we have to move right, meaning left is set to mid+1.
So left will be 4.

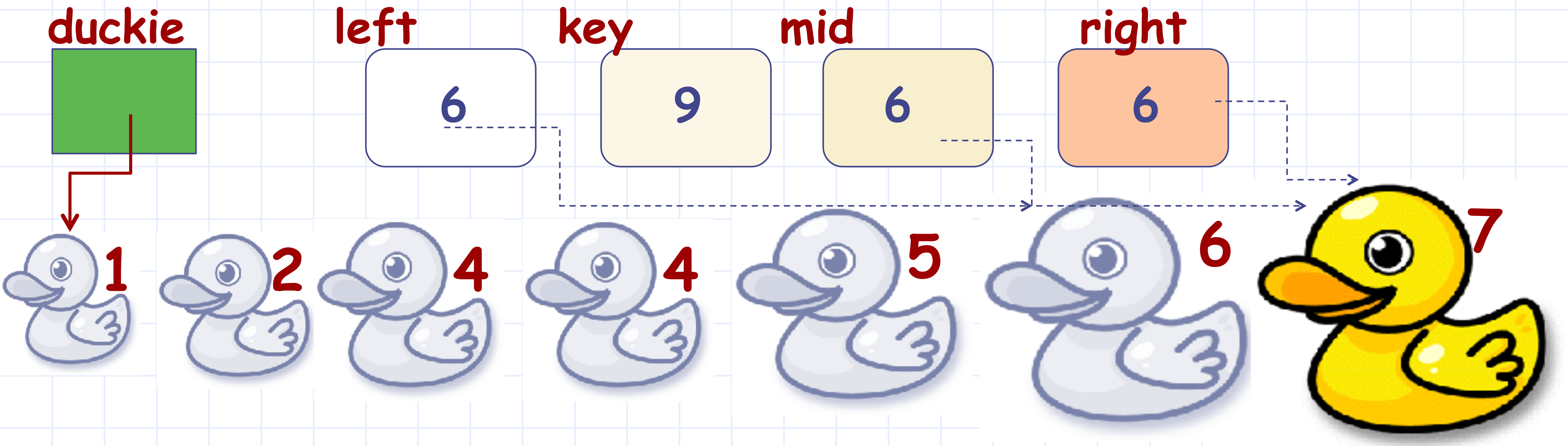
search for 9



Set mid to (left+right)/2. So mid is (4+6)/2 equals 5.
Compare duckie[mid] with key.
duckie[mid] is 6, key is 9 and 6<9.

So, we have to move right again.
Set left to mid +1, so left becomes 6.

search for 9

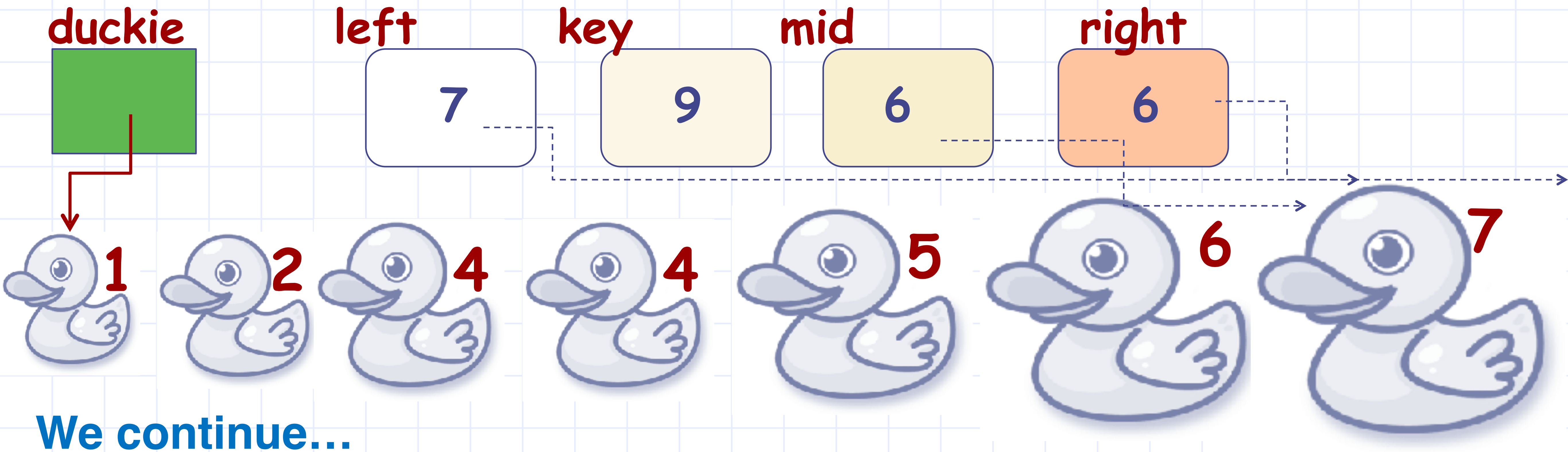


We continue…

Set mid to mid = (left+right)/2.
So mid is (6+6)/2 equals 6.
Compare duckie[mid] with key. duckie[mid] is 7, key is 9 and 7<9.

So, we have to move right again.
Set left to mid +1, so left becomes 7.

search for 9



We continue…

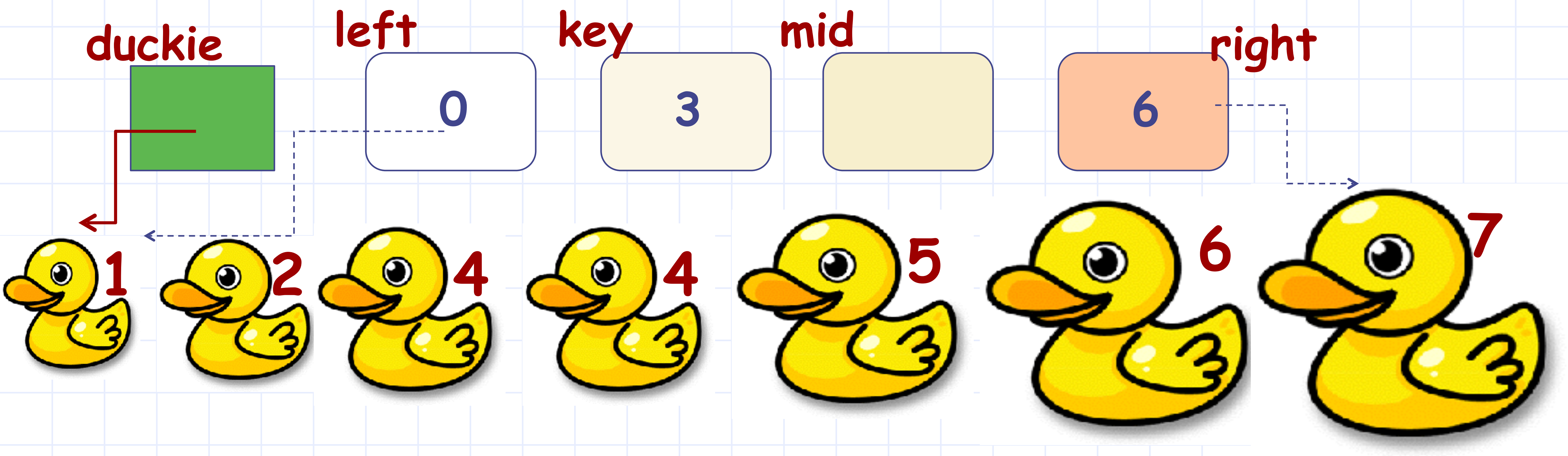
left is 7, right is 6.
The two ends have crossed over.
So the item is not there in the array!

NOT FOUND!

By invariant, item is there between duckie[left] and duckie[right] so long as left <= right.

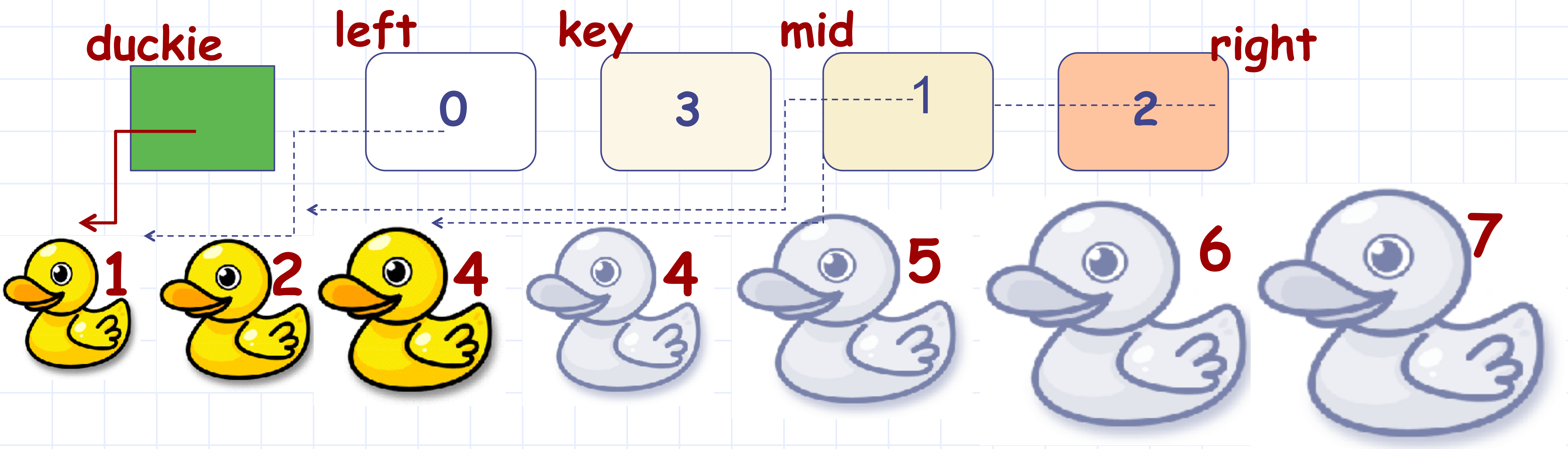
OK, so another condition for termination is **left > right**.
Is there any other termination condition? Can we search for 3?

Searching for 3 in this array.



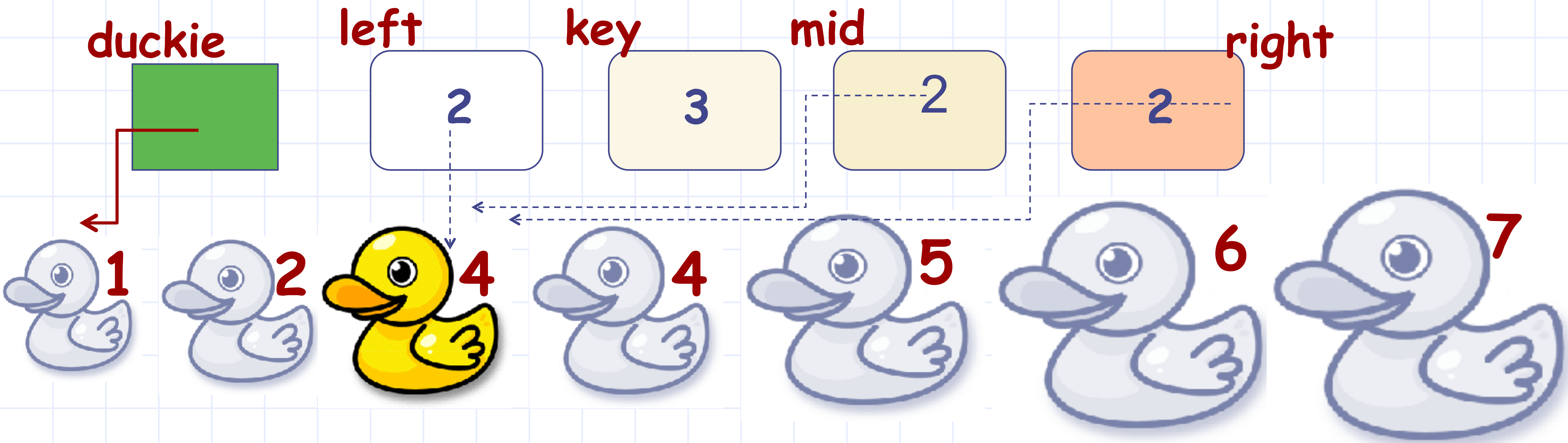
1. left is 0, right is 6.
2. mid is (0+6)/2 which is 3.
3. duckie[mid] is 4, key is 3, so we have to move left.
4. right will be set to mid-1.

Searching for 3 in this array.



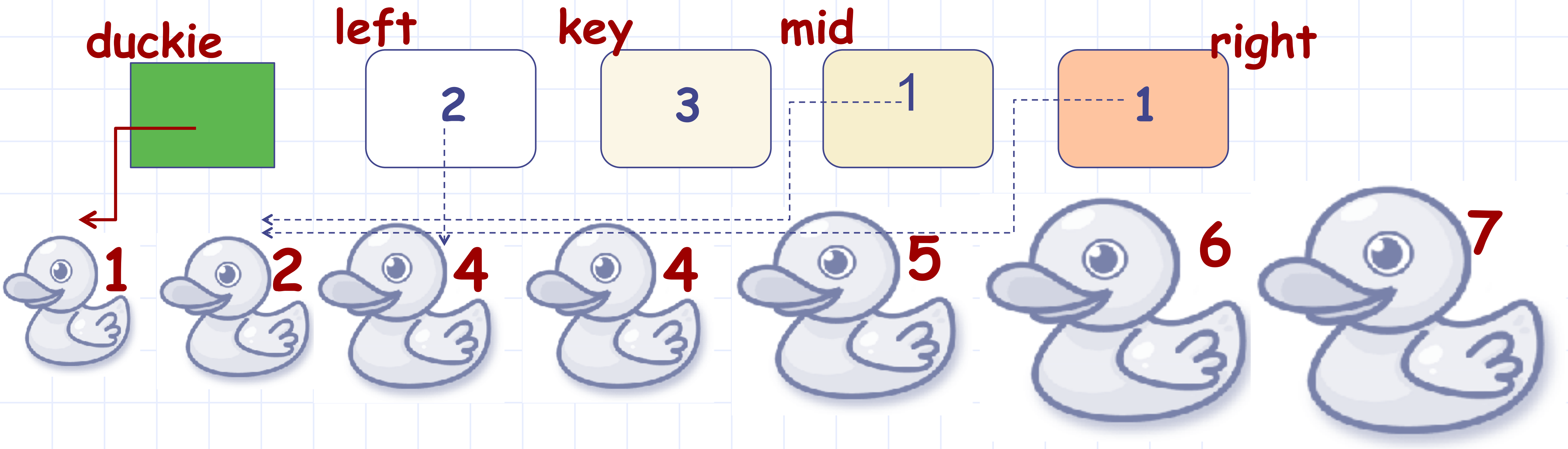
1. left is 0, right is 2.
2. mid is (0+2)/2 which is 1.
3. duckie[mid] is 2 , key is 3, so we have to move right.
4. left will be set to mid+1.

Searching for 3 in this array.



1. left is mid+1 which is 2, right is 2.
2. Now mid is (2+2)/2 which is 2.
3. duckie[mid] is 4 , key is 3, so we have to move left.
4. right will be set to mid-1,So right will be 1.

Searching for 3 in this array.



1. left is 2, right is 1.
2. Left and right have crossed over, **NOT FOUND!**

Binary Search for Sorted Arrays

- binsearch(a, start, end, key)
  - Search key between a[start]…a[end], where a is a sorted (non-decreasing) array

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
        return binsearch(a, start, mid-1, key);
else  return binsearch(a, mid+1, end, key);
```

Isn't this same as search2?

- Let us look closely

```
int search2(a, start, end, key) {
  if start > end, return 0;
  mid = (start + end)/2 ;
  if a[mid]==key, return 1;
  return search2(a, start, mid-1, key)
           || search2(a, mid+1, end,
 ey);
```

In worst case,
Both search2 may fire.
But, **only ONE** of the two binsearch will fire.

```
f start > end, return 0;
nid = (start + end)/2 ;
f a[mid]==key, return 1;
f (a[mid] > key)
       return binsearch(a, start, mid-1,
key);
  else  return binsearch(a, mid+1, end,
key);
}
```

Estimating the Time taken

```
if start > end, return 0;
mid = (start + end)/2 ;
if a[mid]==key, return 1;
if (a[mid] > key)
      return binsearch(a, start, mid-1, key);
else  return binsearch(a, mid+1, end, key);
```

- binsearch
  - Recurrence?

$$T(n) = T(n/2) + C$$

  - Solution?

$$T(n) \propto \log n$$

  - The worst case run time of binsearch is proportional to the log of the size of array
    - Much faster than Search/Search1/Search2 for large arrays
    - Remember: It works for sorted arrays