Sorting Algorithms, Binary Search, Recursion

02-201 / 02-601

Recursion & The Stack

Computing Power(x,y)

- Write a function power(x,y) that returns x^y.

```
func power(x,y int) int {
    ans := 1
    for i := 1; i <= y; i++ {
        ans *= x
    }
    return ans
}
```

- How long will this take to run?

- Can write a function that will be faster?

Computing Power(x,y)

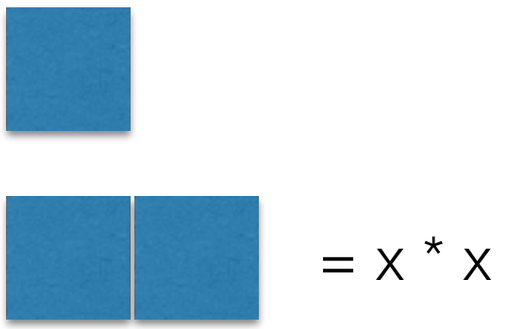
- Write a function power(x,y) that returns x^y.

```
func power(x,y int) int {
    ans := 1
    for i := 1; i <= y; i++ {
        ans *= x
    }
    return ans
}
```

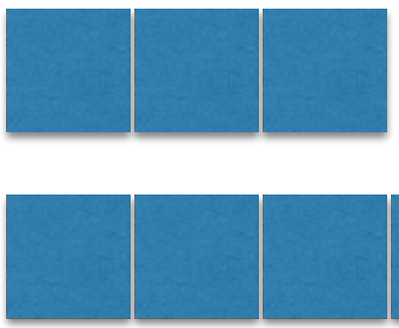
- How long will this take to run? **About y steps**

- Can write a function that will be faster?

Our previous version:



= x * x



= x * x * x

= x * x * x * x

= x * x * x * x * x

= x * x * x * x * x * x

= x * x * x * x * x * x * x

Multiple by x each time through the loop.

At the end, need to multiply x together y times — is there an way to do this with fewer than y multiplications?

Recursively Solve power(x, y/2)

= power(x, y/2) * power(x, y/2)

= power(x, y/2)

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    return z*z
}
```

What if y is odd?

Complete Function

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}
```

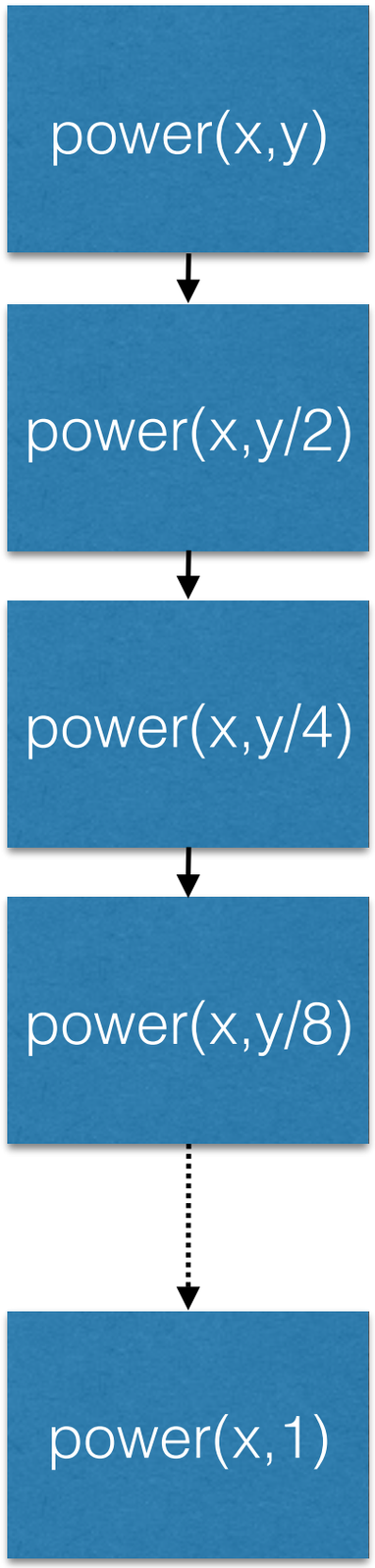
If y is odd, need to do one more multiplication

Running time of modified version

- Inside this function we do a constant amount of work (independent of x and y):
  - 3 if statements
  - 1 or 2 multiplications
  - 1 function call

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}
```

How many times is power called?

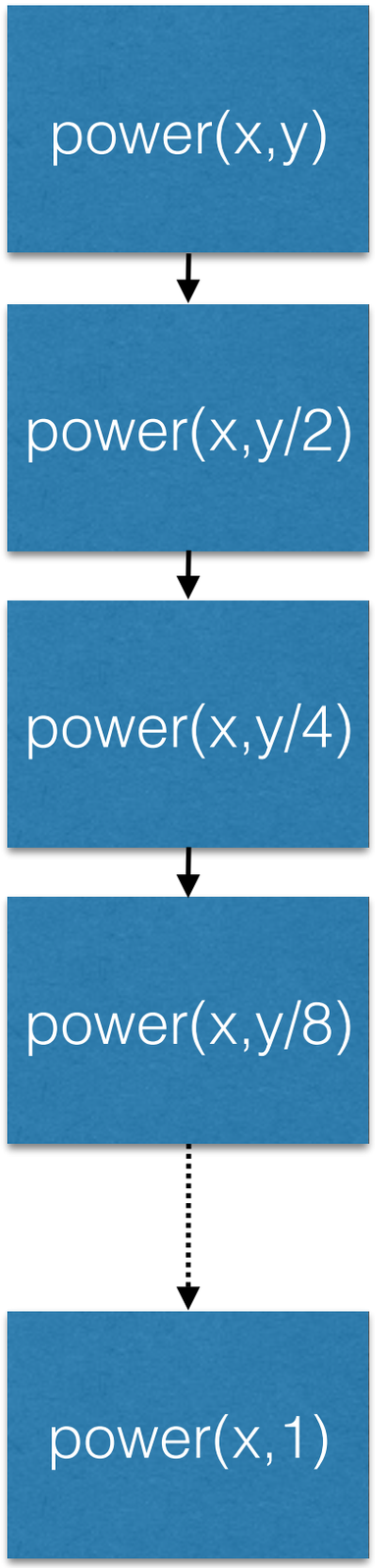


Every time we recurse, y is halved:

Base case: when y = 1, we stop the recursion:

- How many times can you halve a number y before you get to 1?

How many times is power called?



Every time we recurse, y is halved:

Base case: when y = 1, we stop the recursion:

- How many times can you halve a number y before you get to 1?

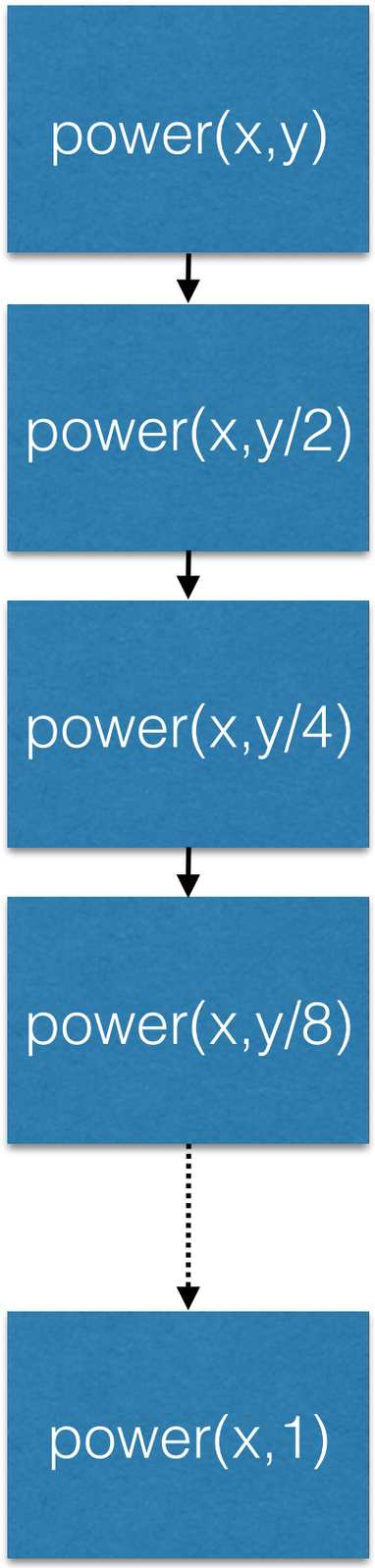
- Want *i* such that: $2^i = y$

- When this happens, the denominator will equal y and $y / 2^i$ will equal 1.

- Take log of both sides: $\log_2 2^i = \log_2 y$

- Therefore, $i = \log_2 y$

How many times is power called?



Every time we recurse, y is halved:

Base case: when y = 1, we stop the recursion:

- How many times can you halve a number y before you get to 1?

- Want *i* such that: $2^i = y$
- When this happens, the denominator will equal y and y / 2^i will equal 1.
- Take log of both sides: $\log_2 2^i = \log_2 y$
- Therefore, $i = \log_2 y$

Will recurse ≈$\log_2$ y times.

So total work is about $\log_2 y$.

Recursion

- How does it work for power to call itself?

- On one hand: nothing special is going on here. Power is a function and we can call the function like any other:

```
fibb(i)    = fibb(i-1) + fibb(i-2)
fibb(1)    = 1
fibb(2)    = 1
```

- This works out so long as eventually we get to a case where the function doesn’t call itself.

- On the other hand: each time you call the function, you need to create new variables (x, y, and z in power). How is this done?

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(2) factorial(10)

```
x = 10
f = 0 … 3628800
i = 1 … 10
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(3) factorial(10)

```
x = 3
f = 0 … 6
i = 1 … 3
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(4) factorial(10)

```
x = 3
f = 0 … 5040
i = 1 … 3
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

(1) nChooseK(10,3)

```
n1 = 10
k1 = 3
numerator = 0
denominator = 0
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

THE Stack

- Behind the scenes, Go (and all other programming languages) maintain a stack that contains the variables associated with the functions you are calling

```
func factorial(x int) int {
    var f int = 0
    for i := 1; i <= x; i++ {
        f = f * i
    }
    return f
}

func nChooseK(n1, k1 int) int {
    var numerator, denominator int
    numerator = factorial(n1) // (2)
    denominator = factorial(k1) // (3)
    denominator = denominator * factorial(n1-k1) // (4)
    return numerator / denominator
}

func main() {
    var n, k, nCk int
    n, k = 10, 3
    nCk = nChooseK(n, k) // (1)
    fmt.Println(nCk)
}
```

main()

```
n = 10
k = 3
nCk = 0
```

- A function call issues a *push* of a record that contains the local variables of the called function.
- A return issues a *pop* of that record (since those local variables are no longer needed)

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

power(10, 4)

```
x = 10
y = 4
z = 0
```

power(10, 8)

```
x = 10
y = 8
z = 0
```

main()

```
no local vars
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

power(10, 2)

```
x = 10
y = 2
z = 0
```

power(10, 4)

```
x = 10
y = 4
z = 0
```

power(10, 8)

```
x = 10
y = 8
z = 0
```

main()

```
no local vars
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

power(10, 1)

```
x = 10
y = 2
z = 0
```

power(10, 2)

```
x = 10
y = 2
z = 0
```

power(10, 4)

```
x = 10
y = 4
z = 0
```

power(10, 8)

```
x = 10
y = 8
z = 0
```

main()

```
no local vars
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

power(10, 2)

```
x = 10
y = 2
z = 0
```

power(10, 4)

```
x = 10
y = 4
z = 0
```

power(10, 8)

```
x = 10
y = 8
z = 0
```

main()

```
no local vars
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

power(10, 4)

```
x = 10
y = 4
z = 0
```

power(10, 8)

```
x = 10
y = 8
z = 0
```

main()

```
no local vars
```

Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

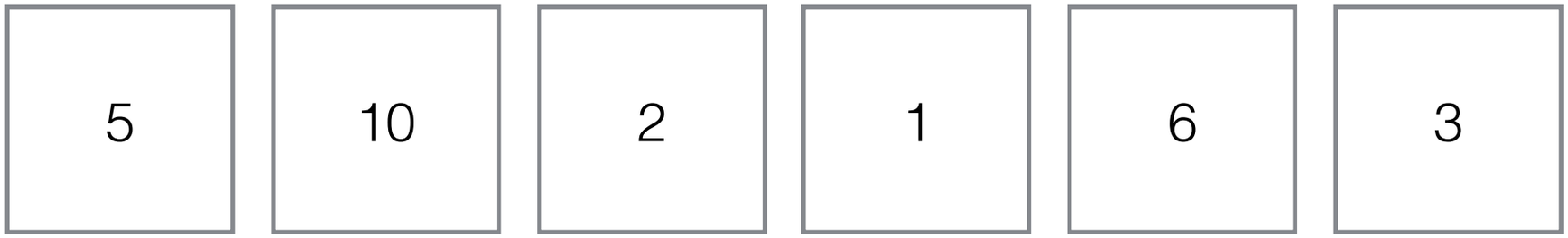
Recursion works the same way

```
func power(x,y int) int {
    if y == 0 { return 1 }
    if y == 1 { return x }
    z := power(x, y/2)
    z = z * z
    if y % 2 == 1 {
        z *= x
    }
    return z
}

func main() {
    power(10, 8)
}
```

Sorting

The Sorting Problem

Given: a set of items k_1, k_2,… k_n, re-order them so that $k_{i1} < k_{i2} < \ldots < k_{in}$



Don’t have to be integers: can sort anything where < is defined.

Insertion Sort & Linked Lists

inList = list of items to sort
outList = empty list
for every item *k* in inList:
    walk down outList finding where *k* should go
    insert *k* into the middle of outList

```
func insertSort1(inList []int) []int {
  var outList []int = make([]int, 0)

  // for every item
  for j, k := range inList {
    if j == 0 {
      outList = append(outList, k)
    } else {
      // walk down outList
      for i := 0; i < len(outList); i++ {
        if outList[i] > k {
          // k belongs at position i
          outList = append(outList, 0)
          copy(outList[i+1:], outList[i:])
          outList[i] = k
          break
        }
      }
    }
  }
  return outList
}
```

copy(x,y) is a builtin function that copies the items from y into x

Here, we use it to make a "hole" at position *i* in order to store *k*

break stops the current **for** loop

How many steps does this insertSort() take?

Time for insertSort1

```
func insertSort1(inList []int) []int {
  var outList []int = make([]int, 0)

  // for every item
  for j, k := range inList {
    if j == 0 {
      outList = append(outList, k)
    } else {
      // walk down outList
      for i := 0; i < len(outList); i++ {
        if outList[i] > k {
          // k belongs at position i
          outList = append(outList, 0)
          copy(outList[i+1:], outList[i:])
          outList[i] = k
          break
        }
      }
    }
  }
  return outList
}
```

- Let n = len(inList)

n times through this loop

possibly n times through this loop

copy could have to copy n items

- About n^3 steps in total

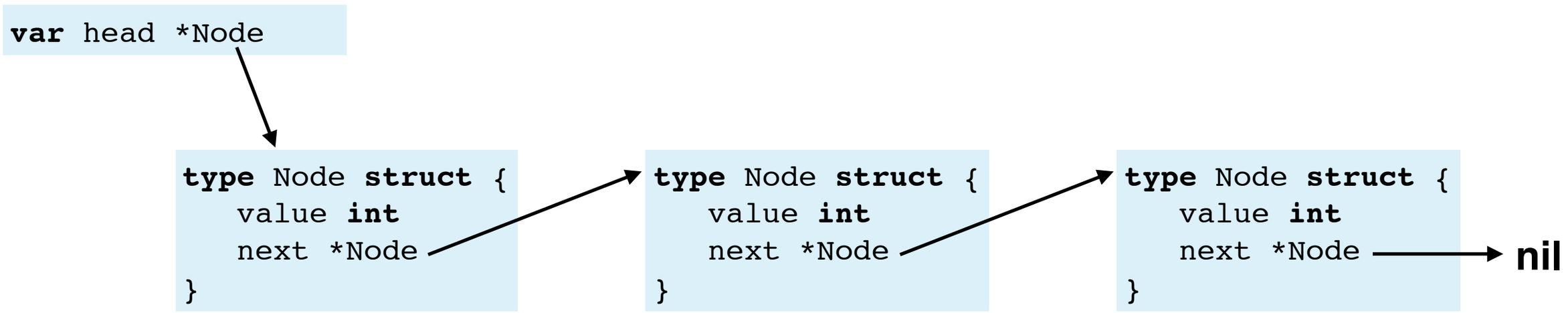
- This is pretty slow: to sort 100 items, might take 1 million steps!

Can we do better?

Linked Lists

- One big problem: we have to move all items after position *i* out of the way to insert something.

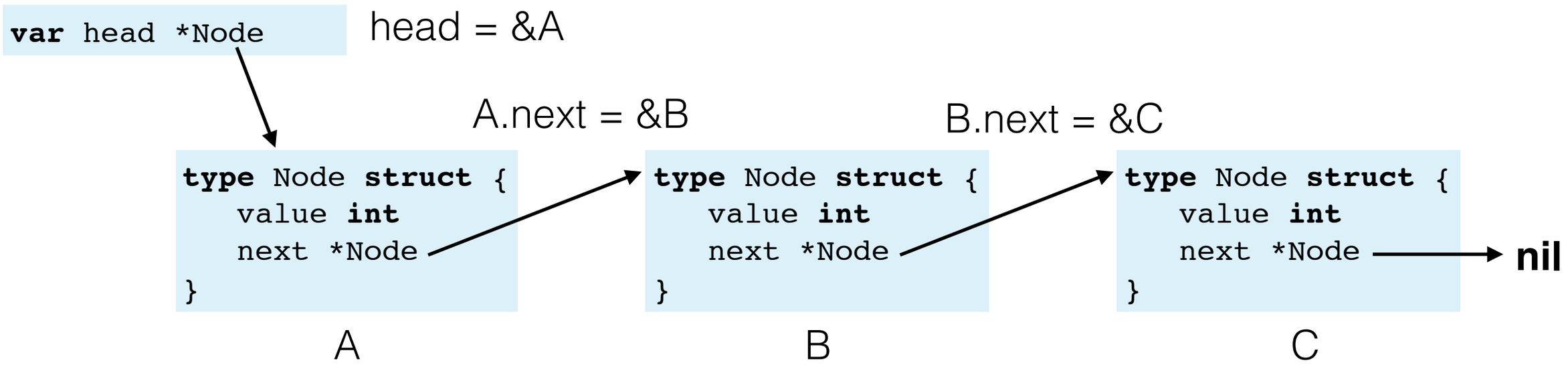
- Linked lists avoid this problem (and are another great example of the utility of pointers)



Linked Lists

- One big problem: we have to move all items after position *i* out of the way to insert something.

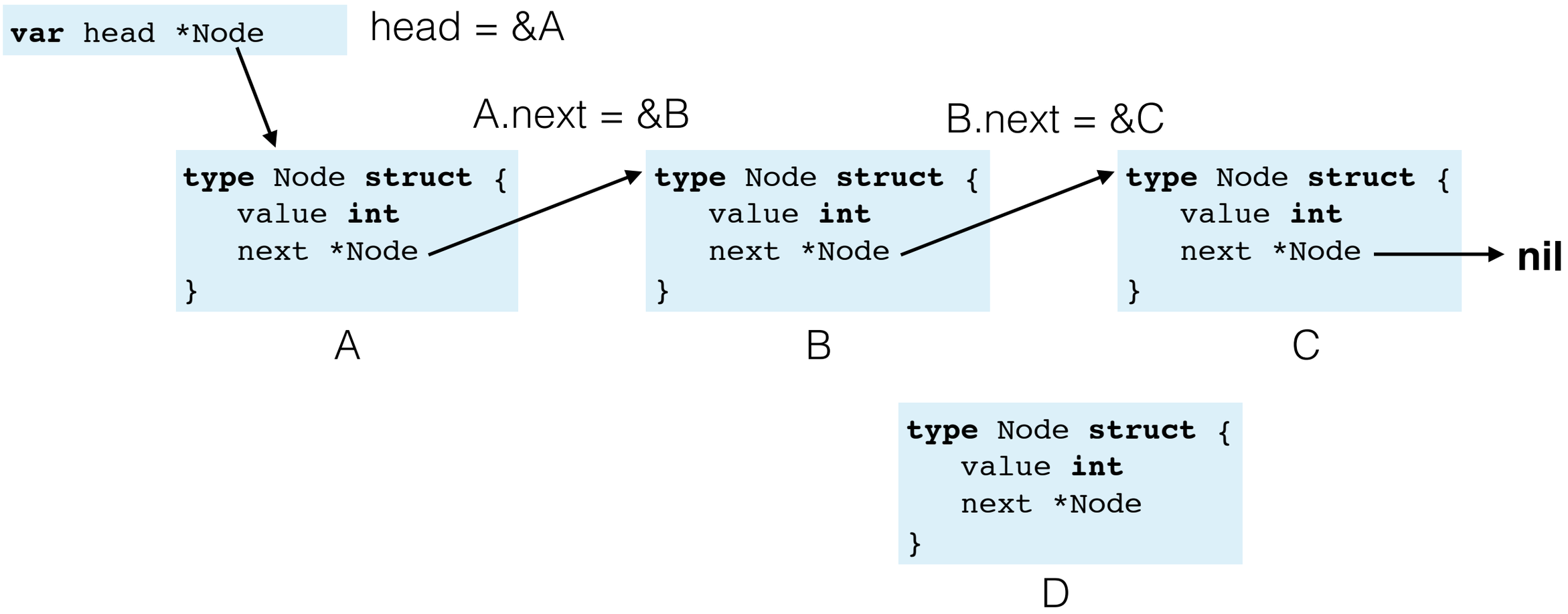
- Linked lists avoid this problem (and are another great example of the utility of pointers)



Linked Lists

- One big problem: we have to move all items after position *i* out of the way to insert something.

- Linked lists avoid this problem (and are another great example of the utility of pointers)

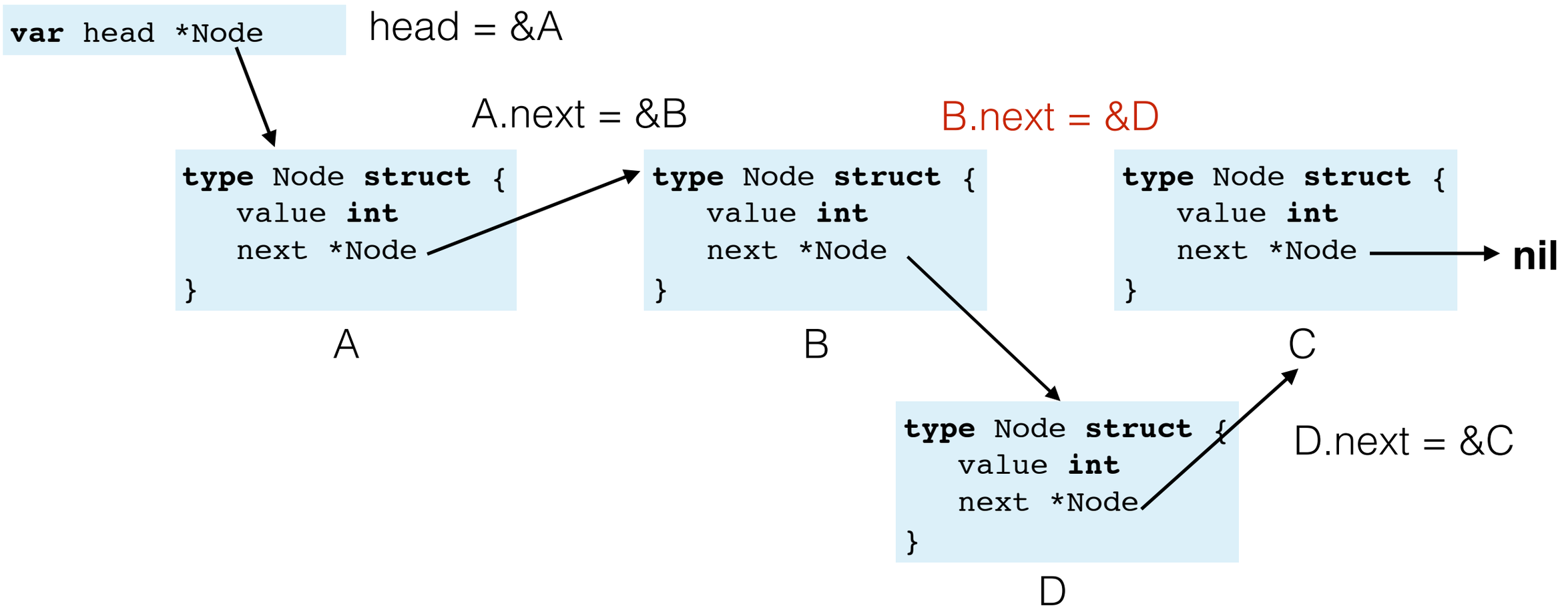


- How can we insert a node between B and C (say)?

Linked Lists

- One big problem: we have to move all items after position *i* out of the way to insert something.

- Linked lists avoid this problem (and are another great example of the utility of pointers)



- How can we insert a node between B and C (say)?

Linked List Insertion Sort

```
func insertionSort(inList []int) []int {
   // create a linked list with just one item in it
   var head *Node = createNode(inList[0])

   // for every remaining item
   for _, k := range inList[1:] {
      newNode := createNode(k)

      // walk down the linked list
      for prev, cur := (*Node)(nil), head; cur != nil; prev, cur = cur, cur.next {

         // if this is where we should insert
         if cur.value > k {

            // if not at the start of the list
            if prev != nil {
               prev.next = newNode
            } else {
               // otherwise, we're at the start of the list
               head = newNode
            }
            newNode.next = cur
            break
         }
      }
   }

   return convertLinkedListToSlice(head)
}
```

try to insert each item

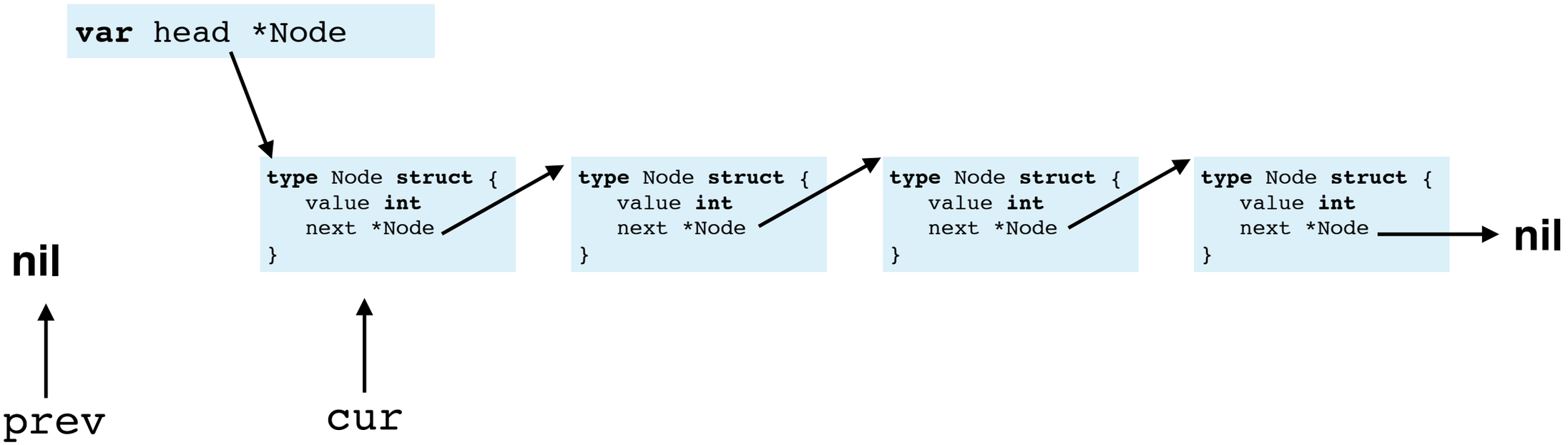
walk down the list, maintaining *two* pointers into it: one for the current node and one for the previous node

If we're not trying to put this at the start

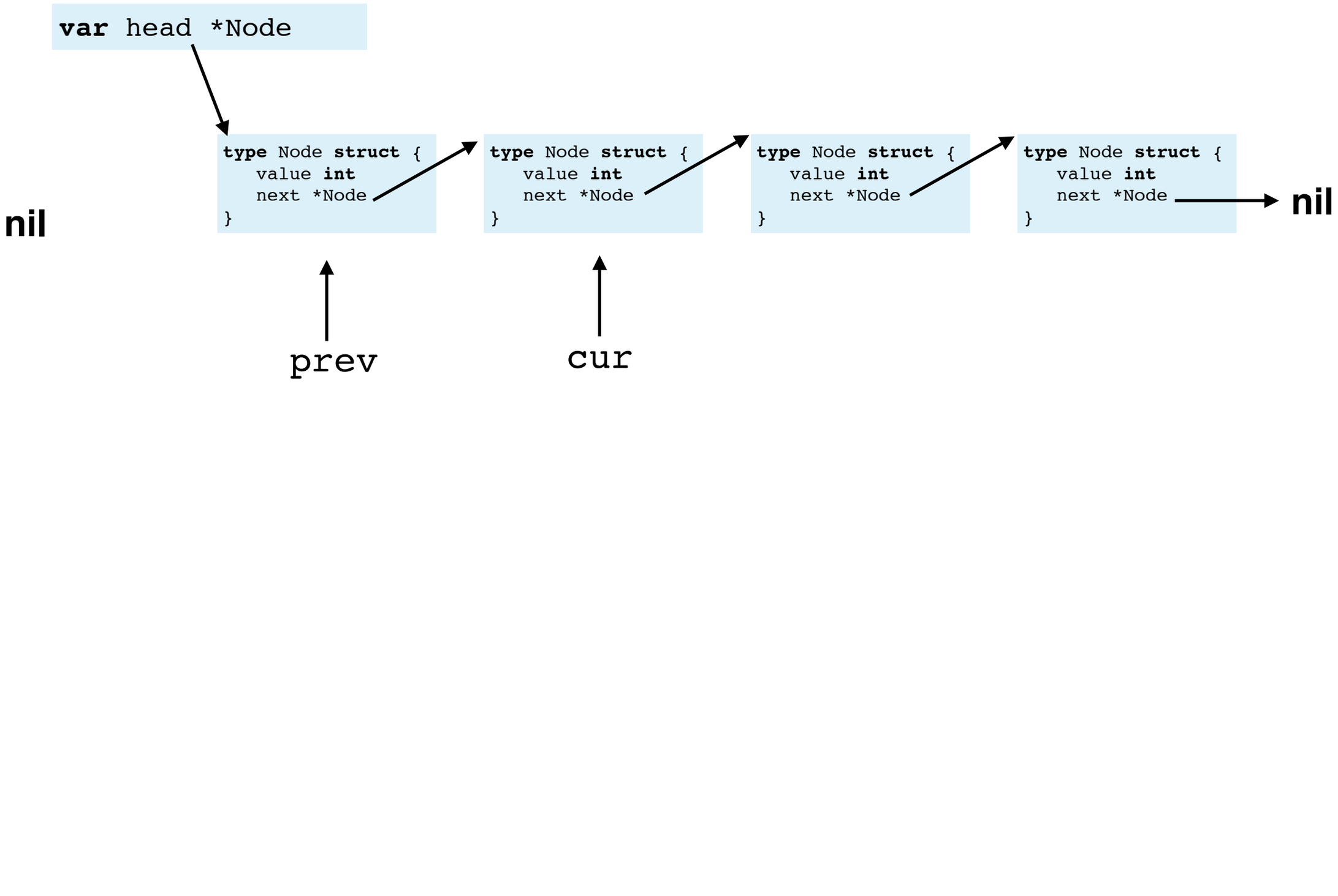
If we need to change head because k belongs at the start

link newNode to cur so that it points to the rest of the list

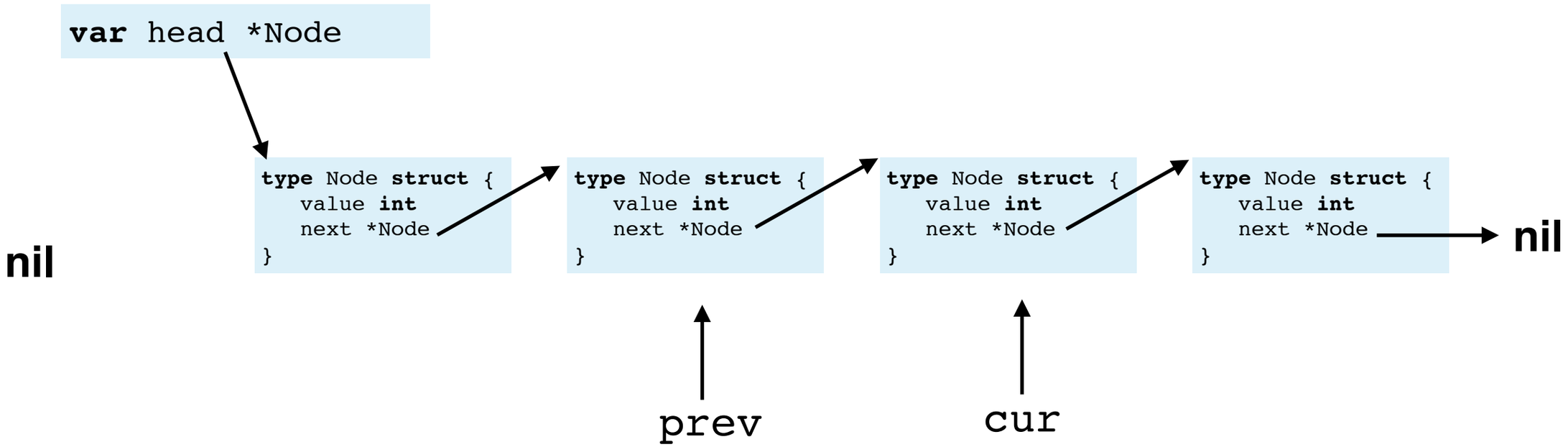
Linked List Insertion In Pictures



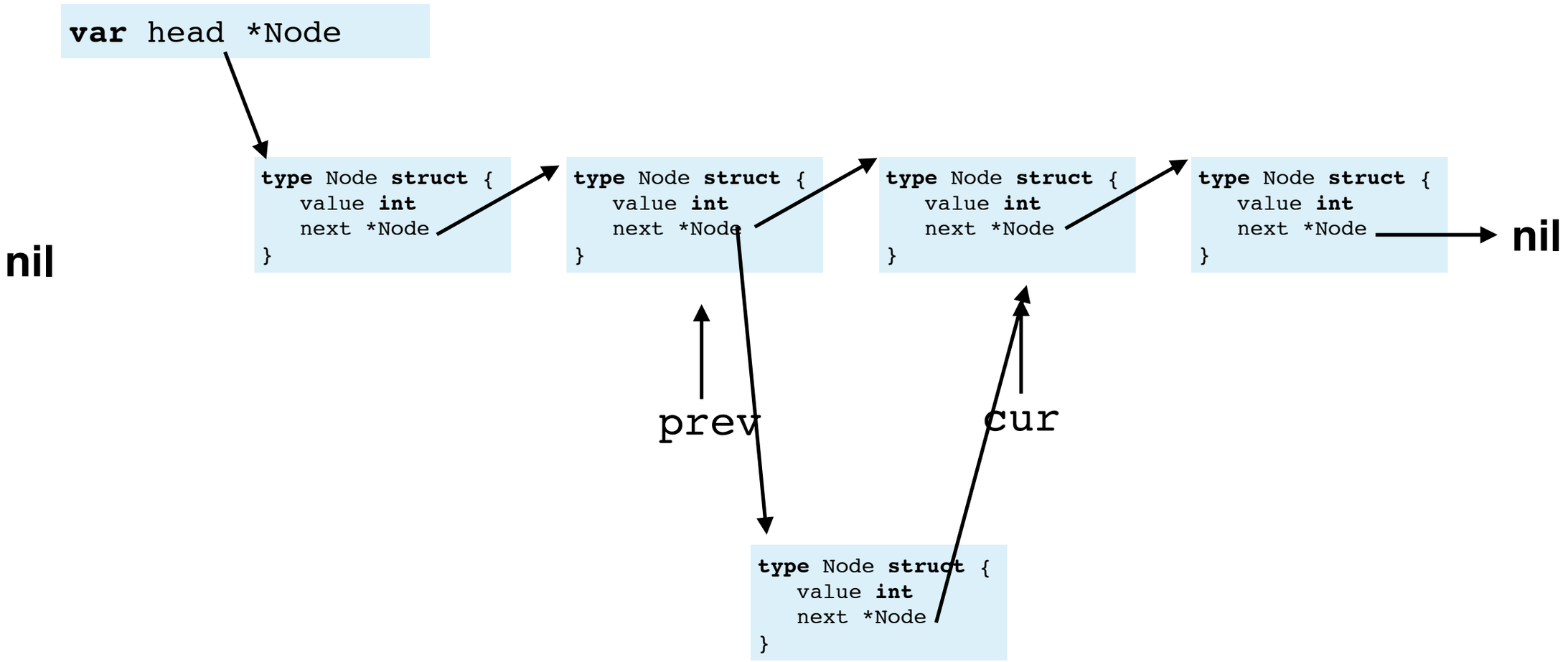
Linked List Insertion In Pictures



Linked List Insertion In Pictures



Linked List Insertion In Pictures



Worst-case runtime for Linked List Insertion Sort

```
func insertionSort(inList []int) []int {
   // create a linked list with just one item in it
   var head *Node = createNode(inList[0])

   // for every remaining item
   for _, k := range inList[1:] {
      newNode := createNode(k)

     // walk down the linked list
     for prev, cur := (*Node)(nil), head; cur != nil; prev, cur = cur, cur.next {

        // if this is where we should insert
        if cur.value > k {

           // if not at the start of the list
           if prev != nil {
              prev.next = newNode
           } else {
              // otherwise, we're at the start of the list
              head = newNode
           }
           newNode.next = cur
           break
        }
     }
  }

  return convertLinkedListToSlice(head)
}
```

about n times through this loop

possibly n times through this loop

A constant amount of work inside here

This insertion sort implementation takes about n^2 steps:
about 10,000 steps to stort 100 numbers.

createNode and convertLinkedListToSlice

```
func createNode(v int) *Node {
    return &Node{value: v, next: nil}
}

func convertLinkedListToSlice(head *Node) []int {
    out := make([]int, 0)
    for p := head; p != nil; p = p.next {
        out = append(out, p.value)
    }
    return out
}
```

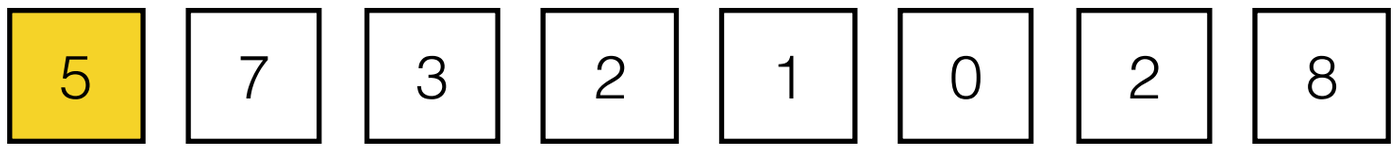
walk down the list with a single pointer

Can we sort faster?

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

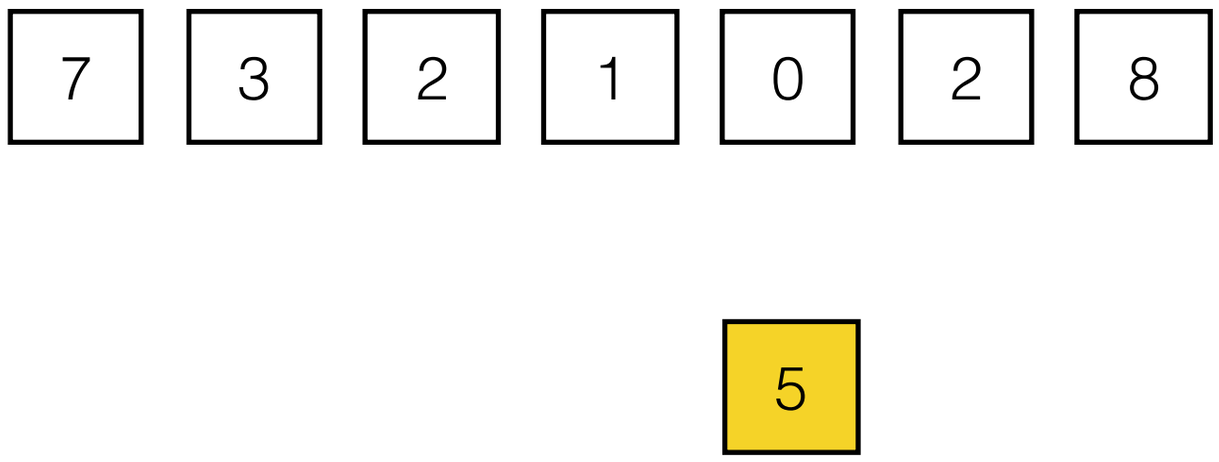


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

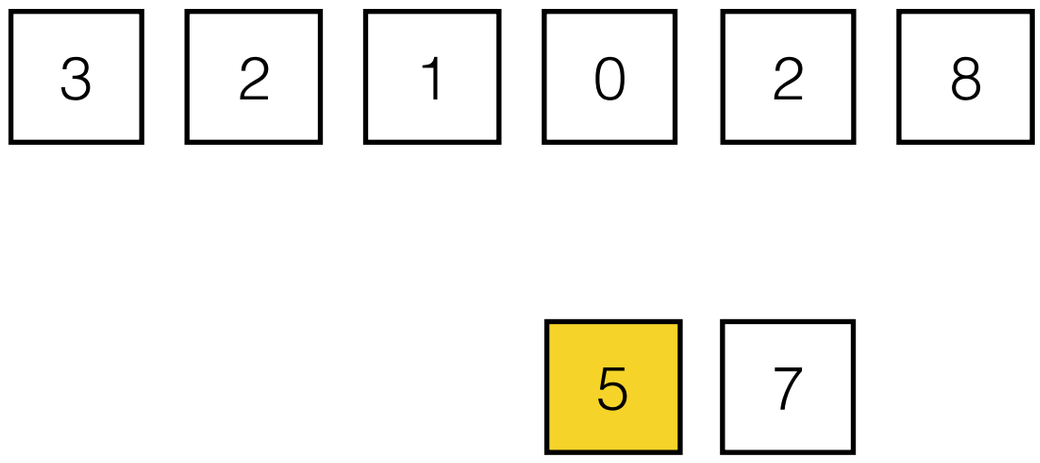


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

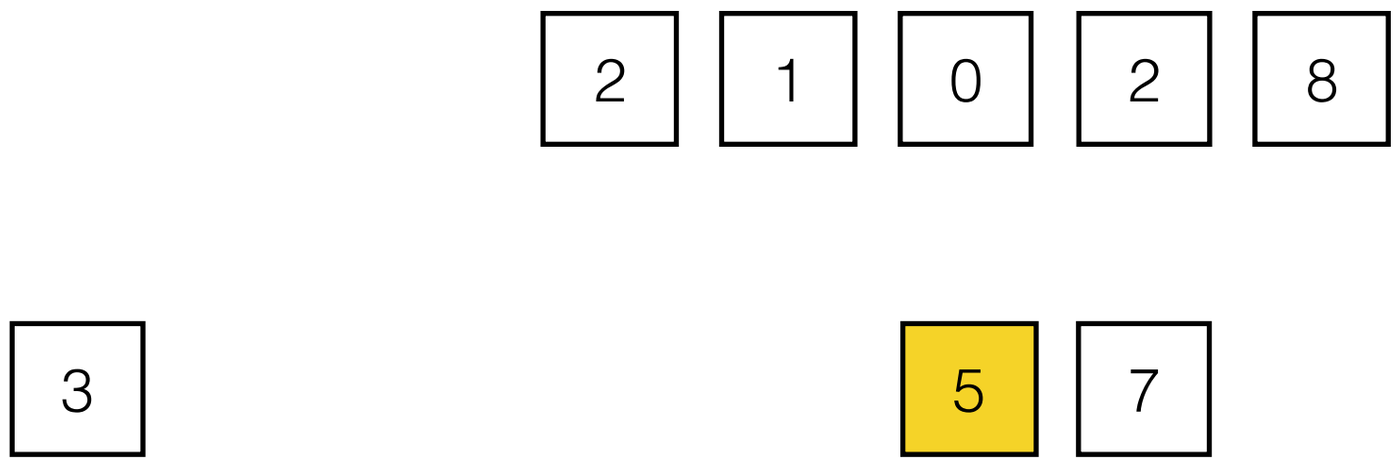


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

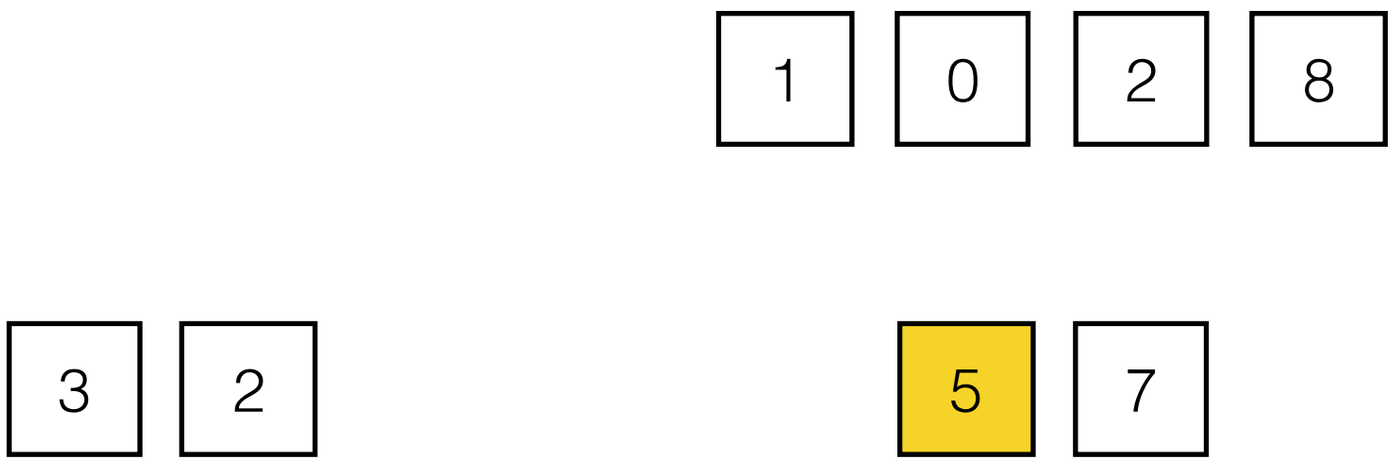


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

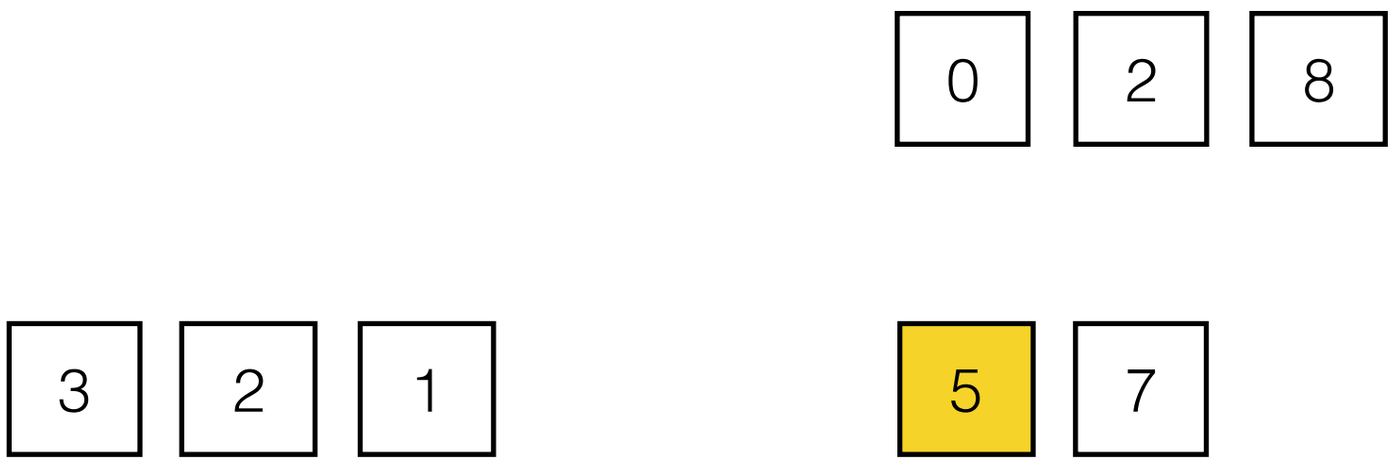


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

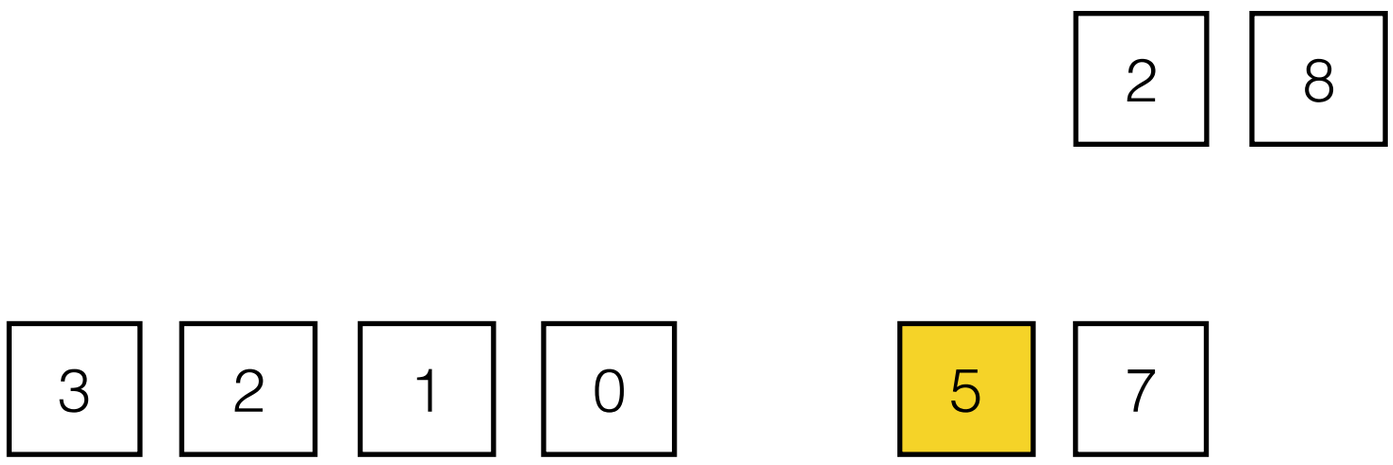


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

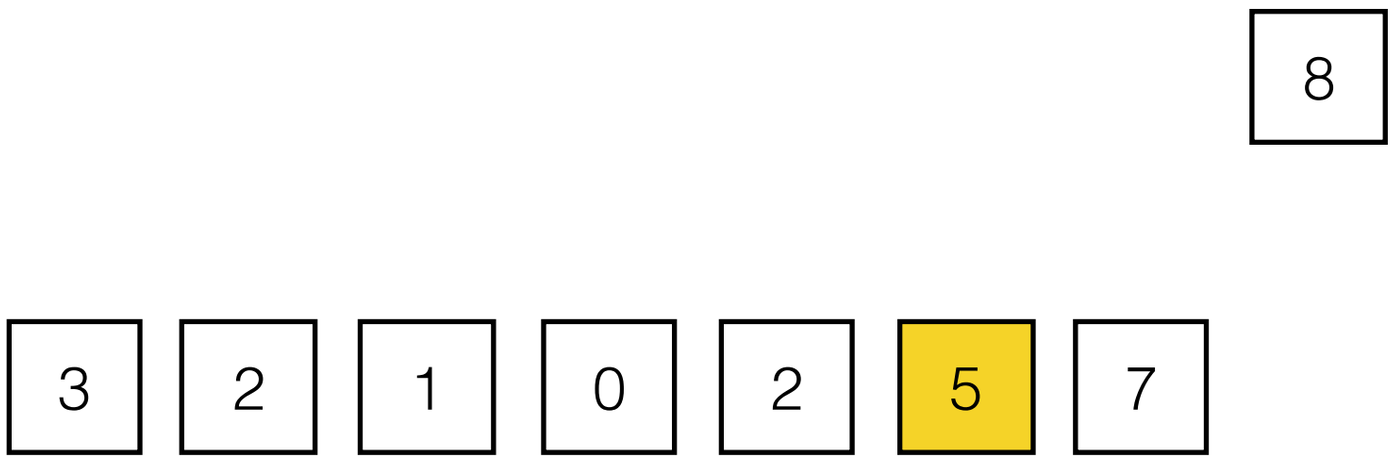


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right

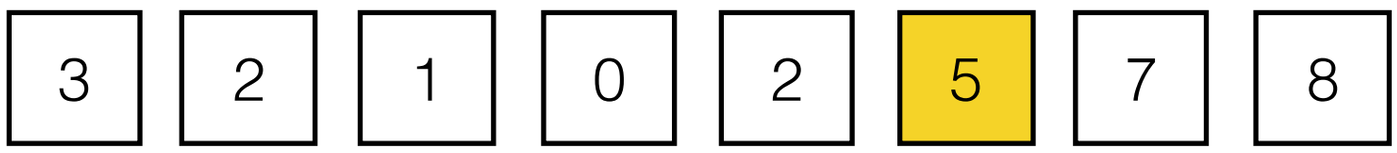


- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort

- Quicksort is often the fastest sort in practice.
- Based on the idea of “divide and conquer”: break the problem of sorting *n* numbers into two subproblems of sorting fewer numbers
- Based on the *partition* operation:

partition: Let p be the first item in the list.
Rearrange the list so that items < p are to the left of p and items > p are to the right



- After this, 5 is in the right place in the sorted order
- And everything that should be before 5 is before it, and everything that is after it should be after it.

Quicksort Using partition

```
func quickSort(inList []int) {
    if len(inList) > 1 {
        p := partition(inList)
        quickSort(inList[:p])
        quickSort(inList[p+1:])
    }
}
```

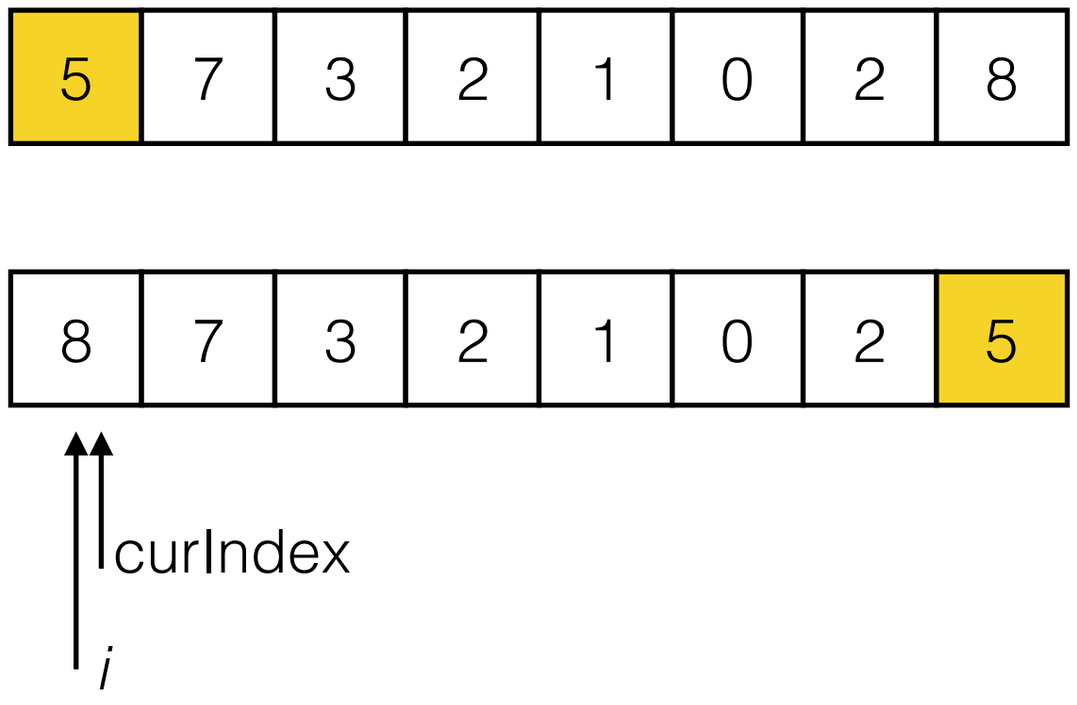
If the list contains 0 or 1 items, it's already sorted

Otherwise, partition, putting inList[0] in the right place

Recursively partition the left half and the right half

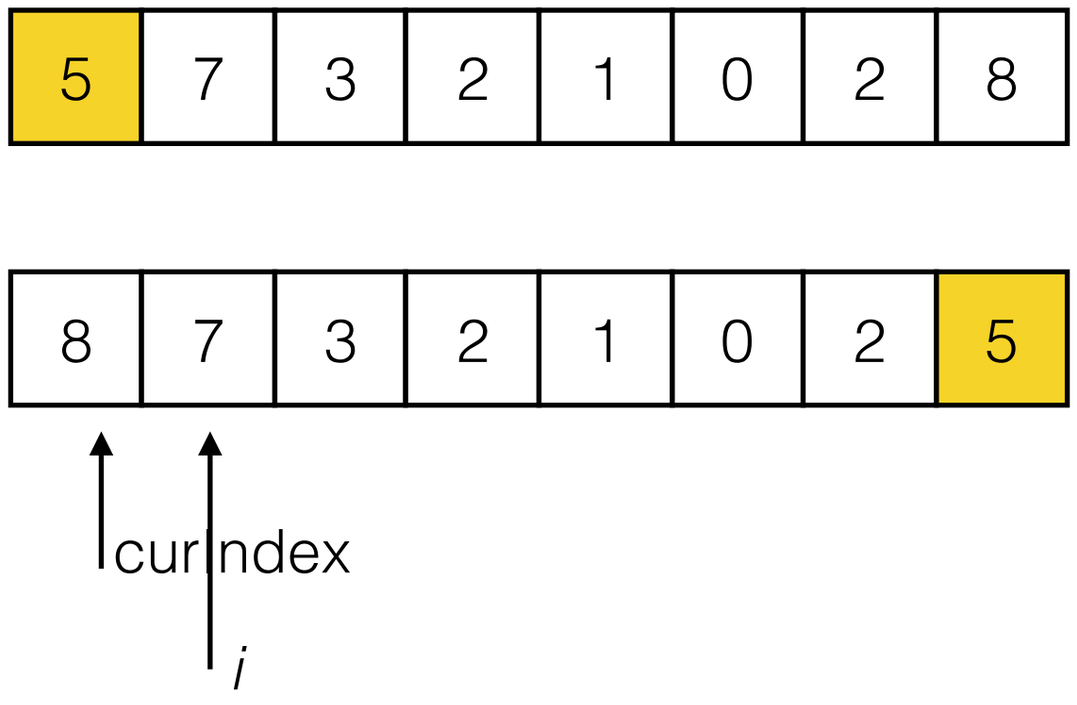
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



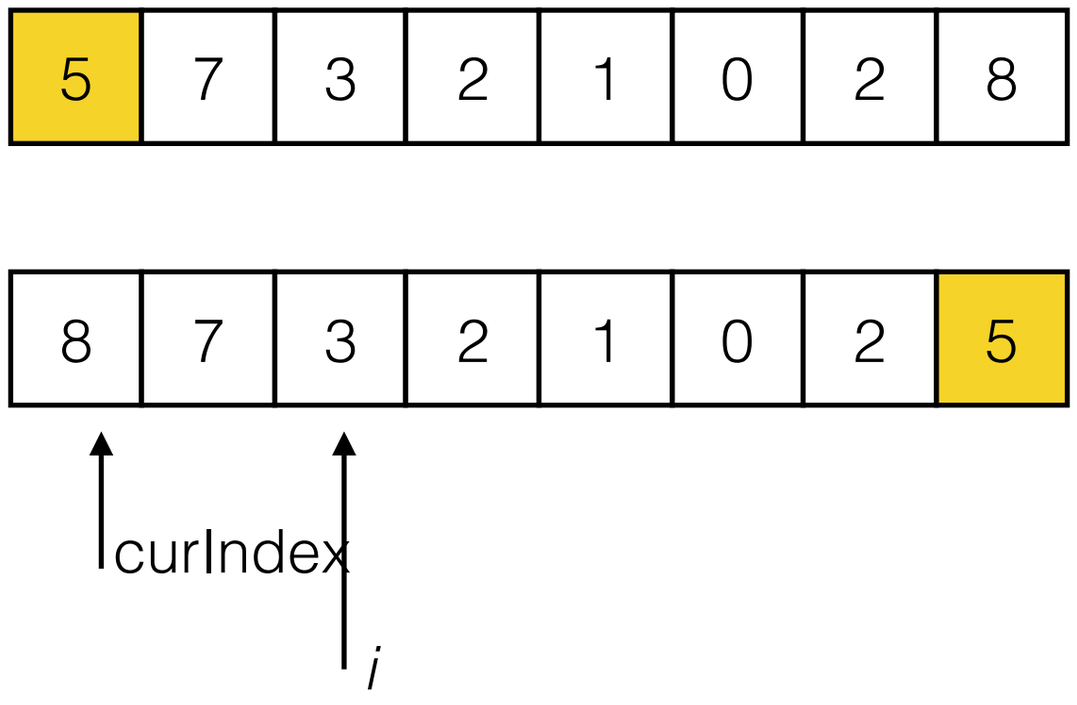
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



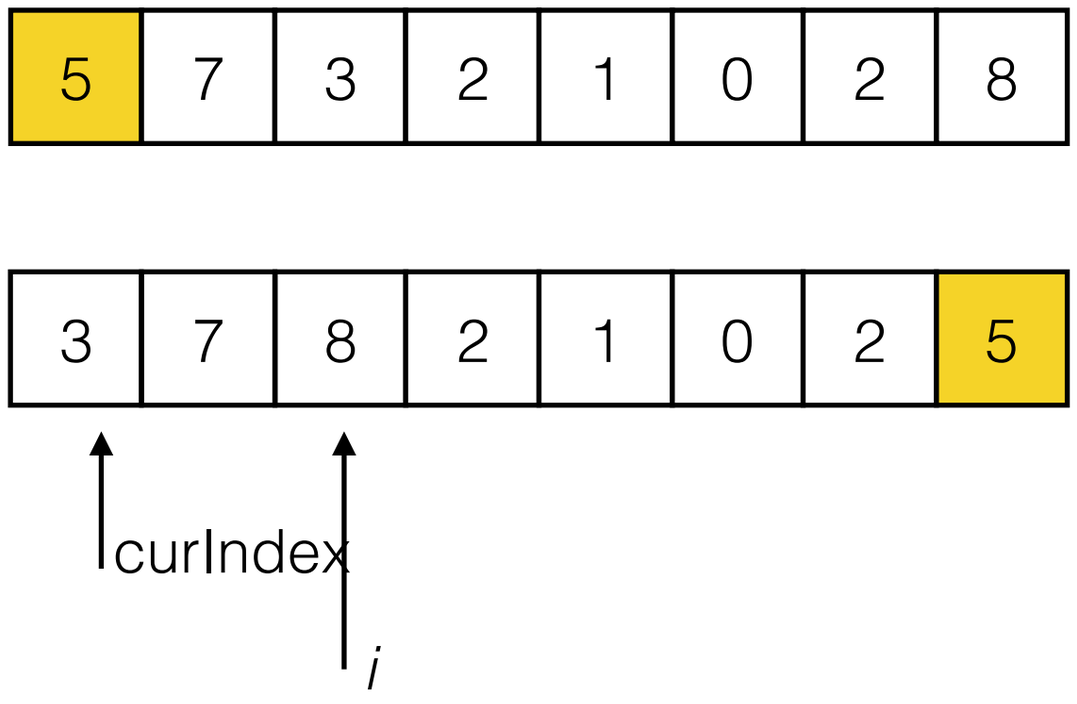
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



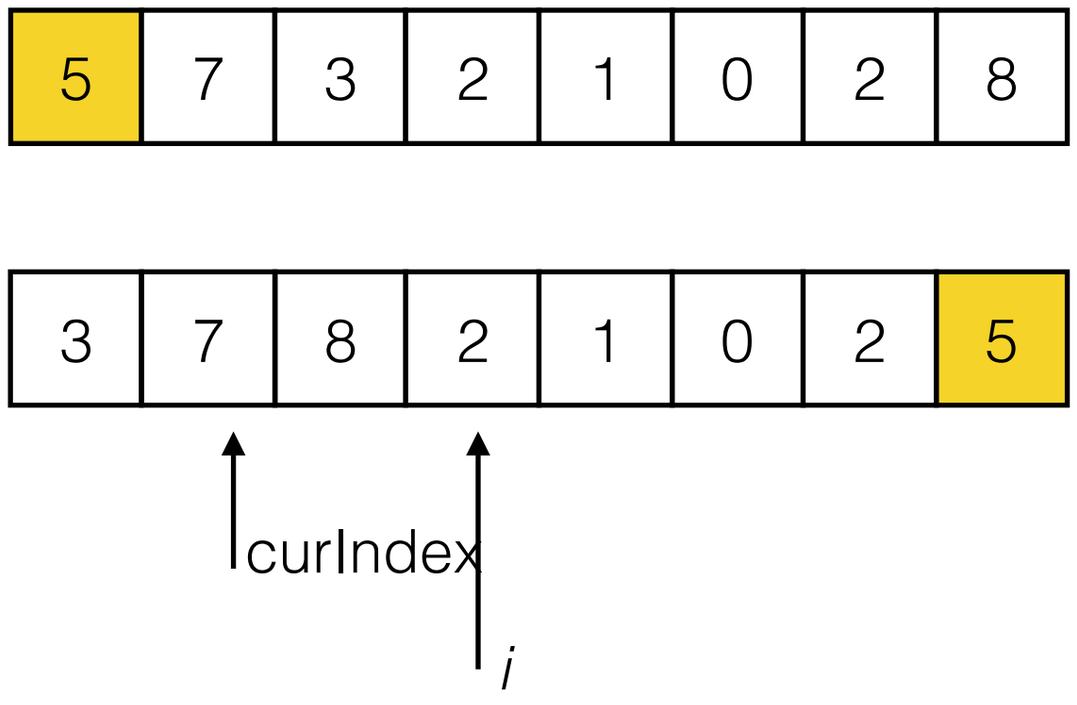
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



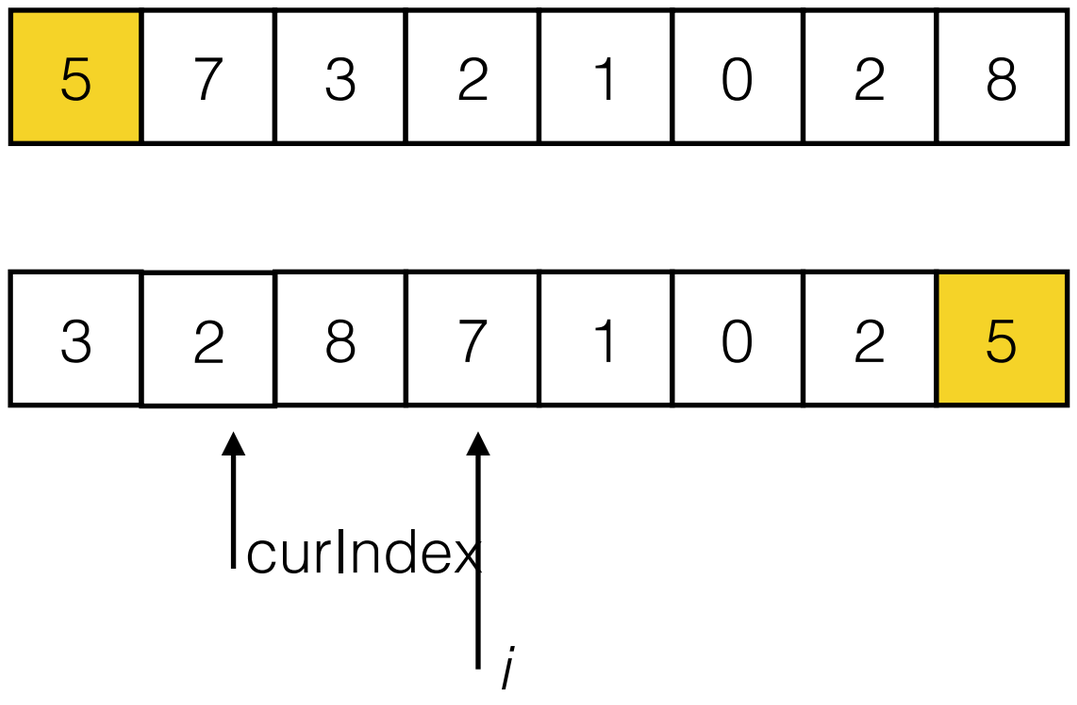
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



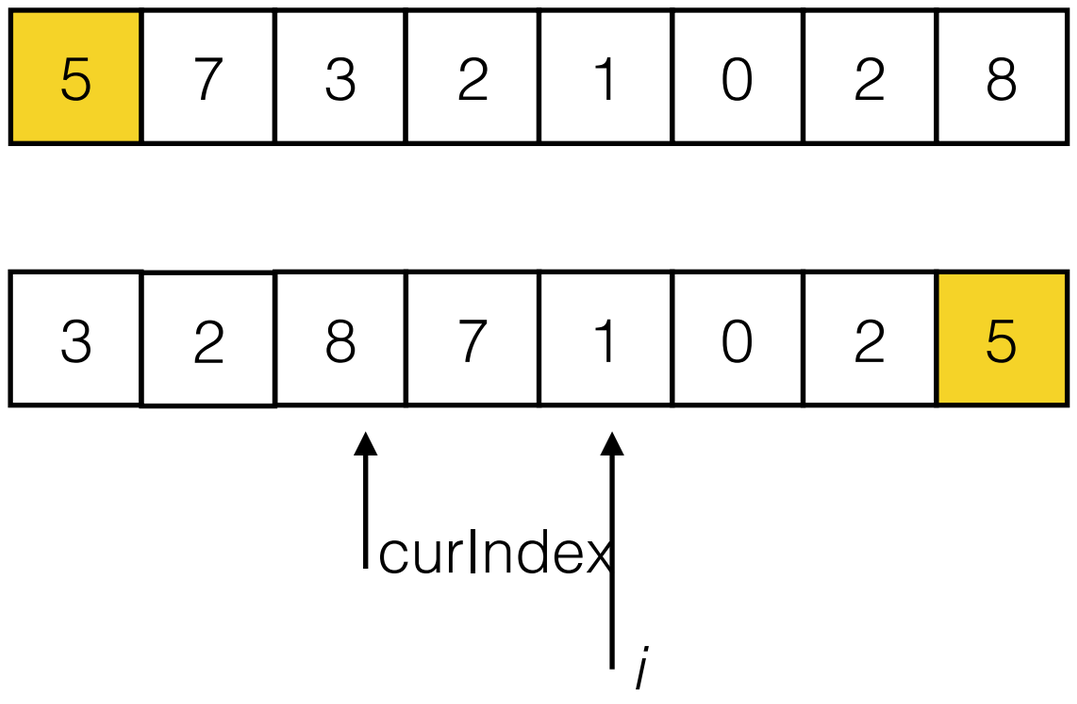
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



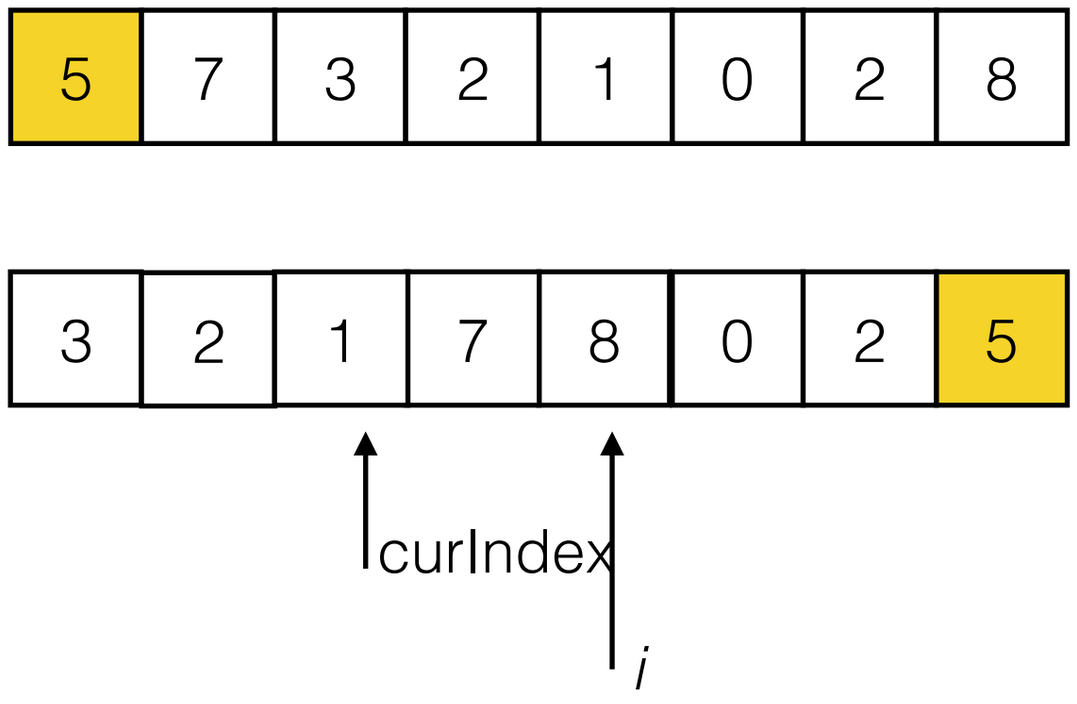
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



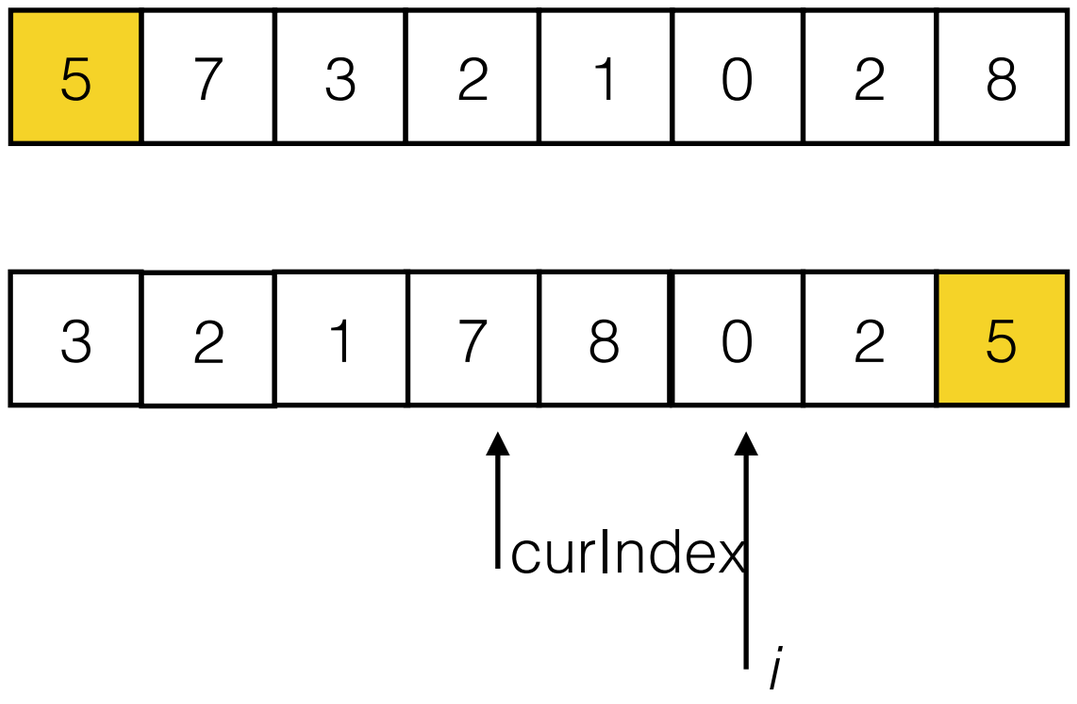
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



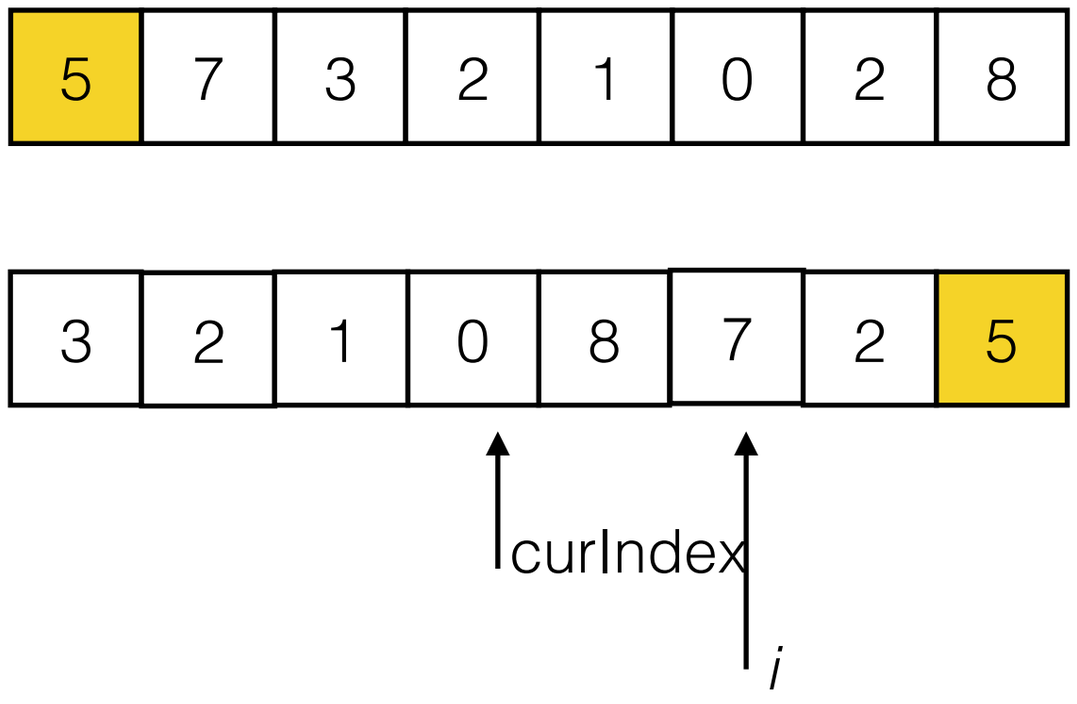
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



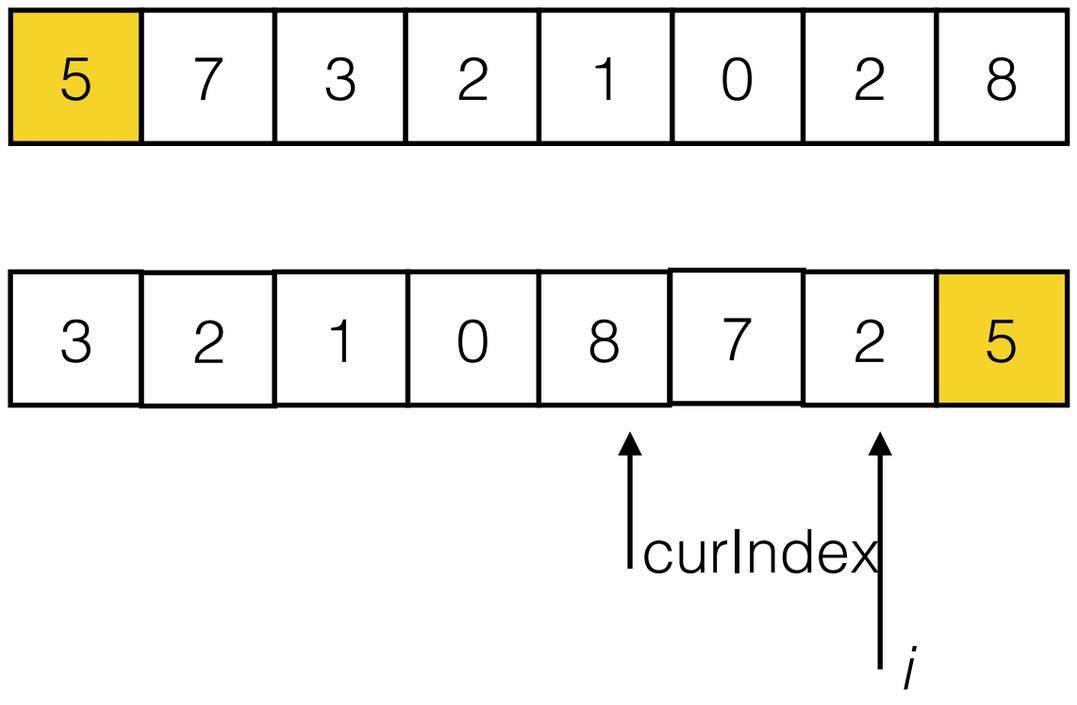
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



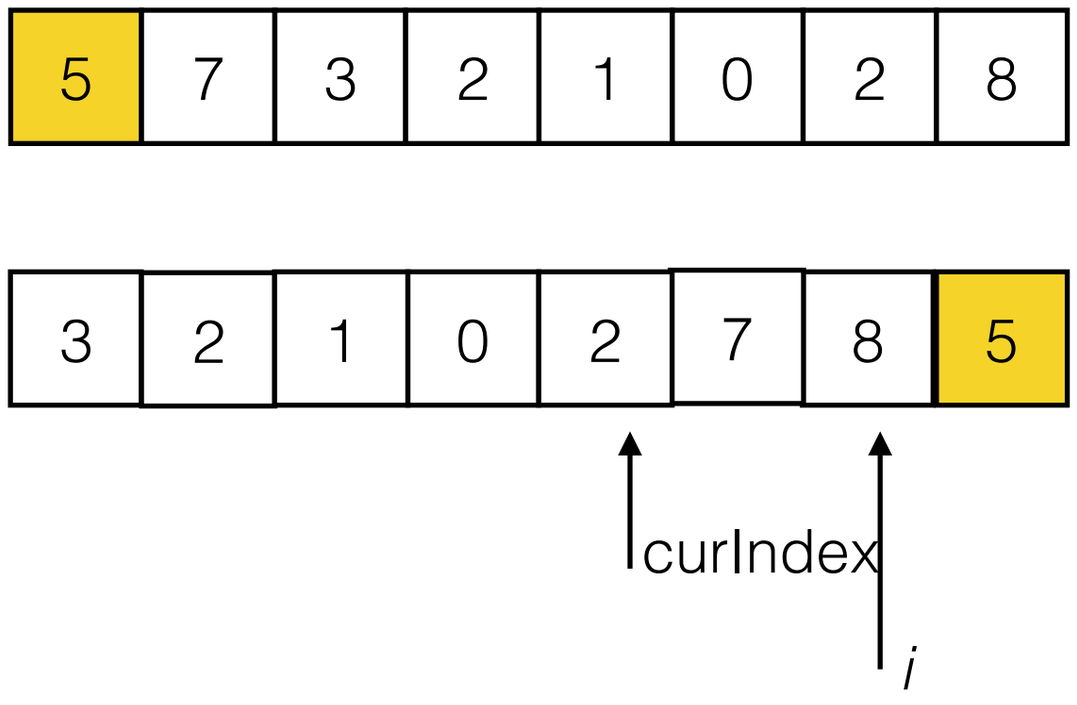
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



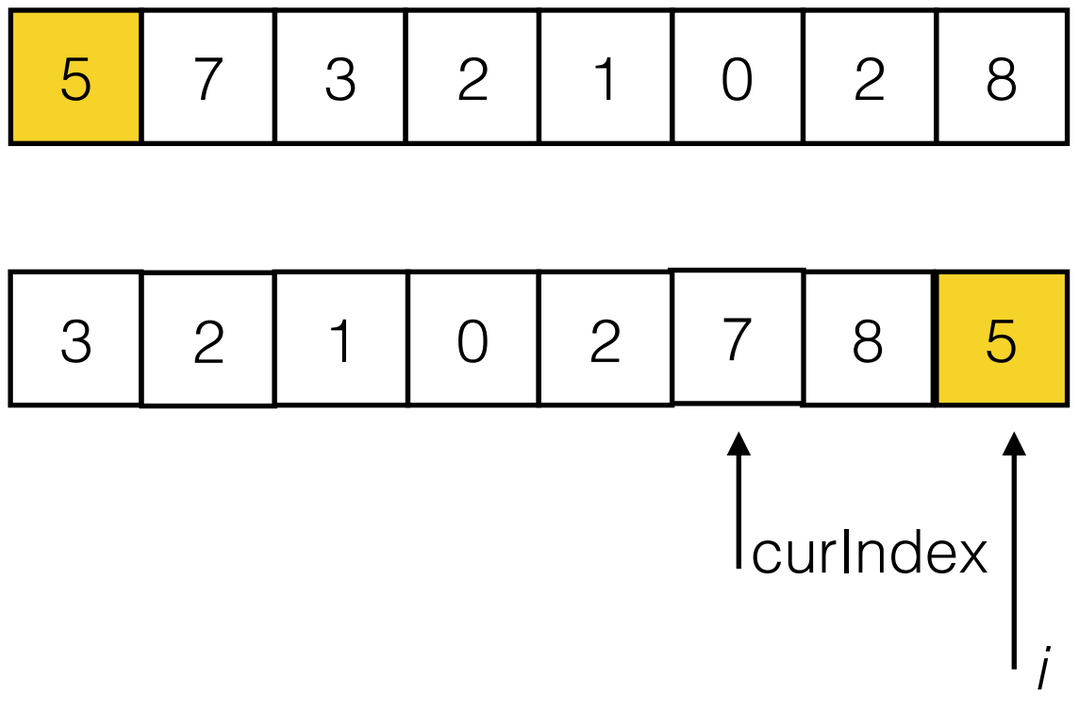
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



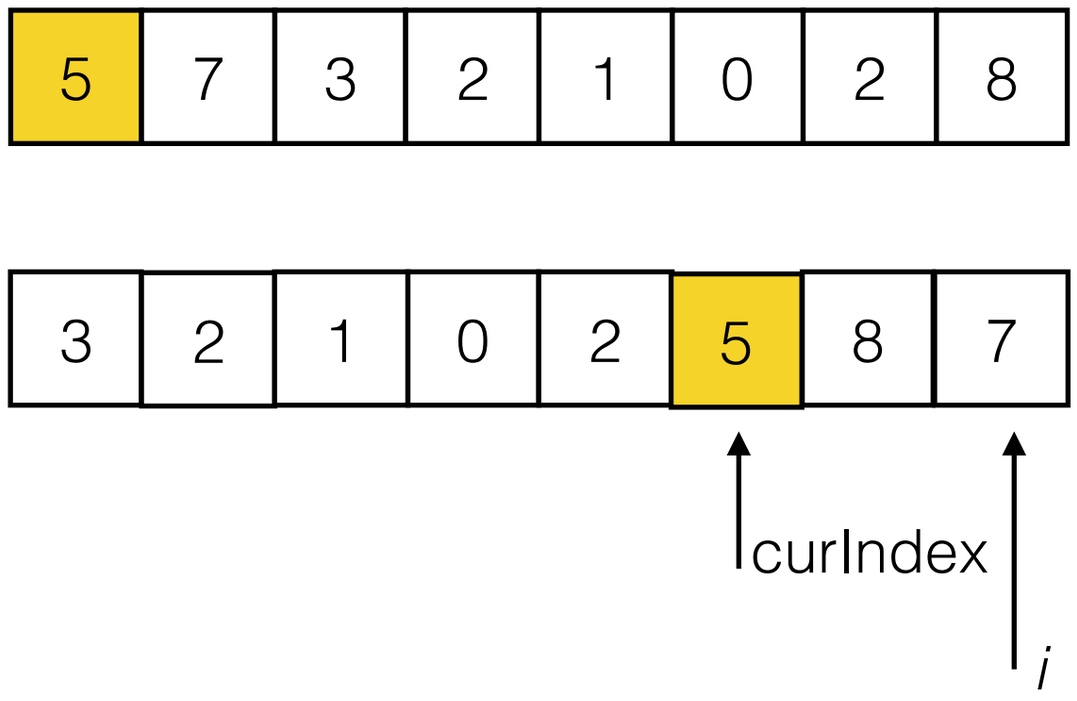
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



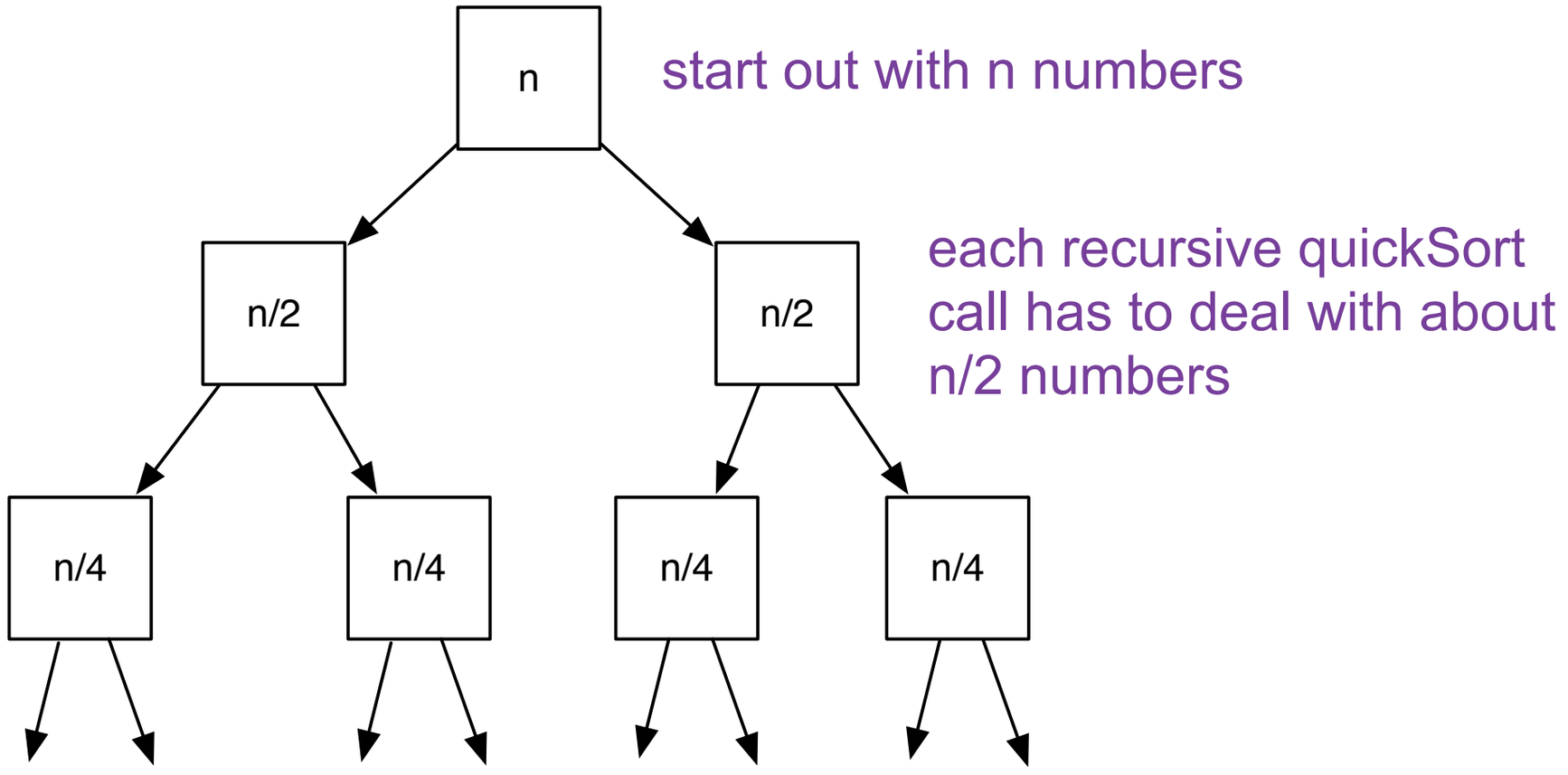
Partition

```
func partition(inList []int) int {
    pivot := inList[0]
    lastPos := len(inList)-1

    // swap the first and list items
    inList[0], inList[lastPos] = inList[lastPos], inList[0]

    curIndex := 0
    for i := 0; i < lastPos; i++ {
        if inList[i] < pivot {
            inList[i], inList[curIndex] = inList[curIndex], inList[i]
            curIndex++
        }
    }
    inList[curIndex], inList[lastPos] = inList[lastPos], inList[curIndex]
    return curIndex
}
```



Runtime of Quicksort

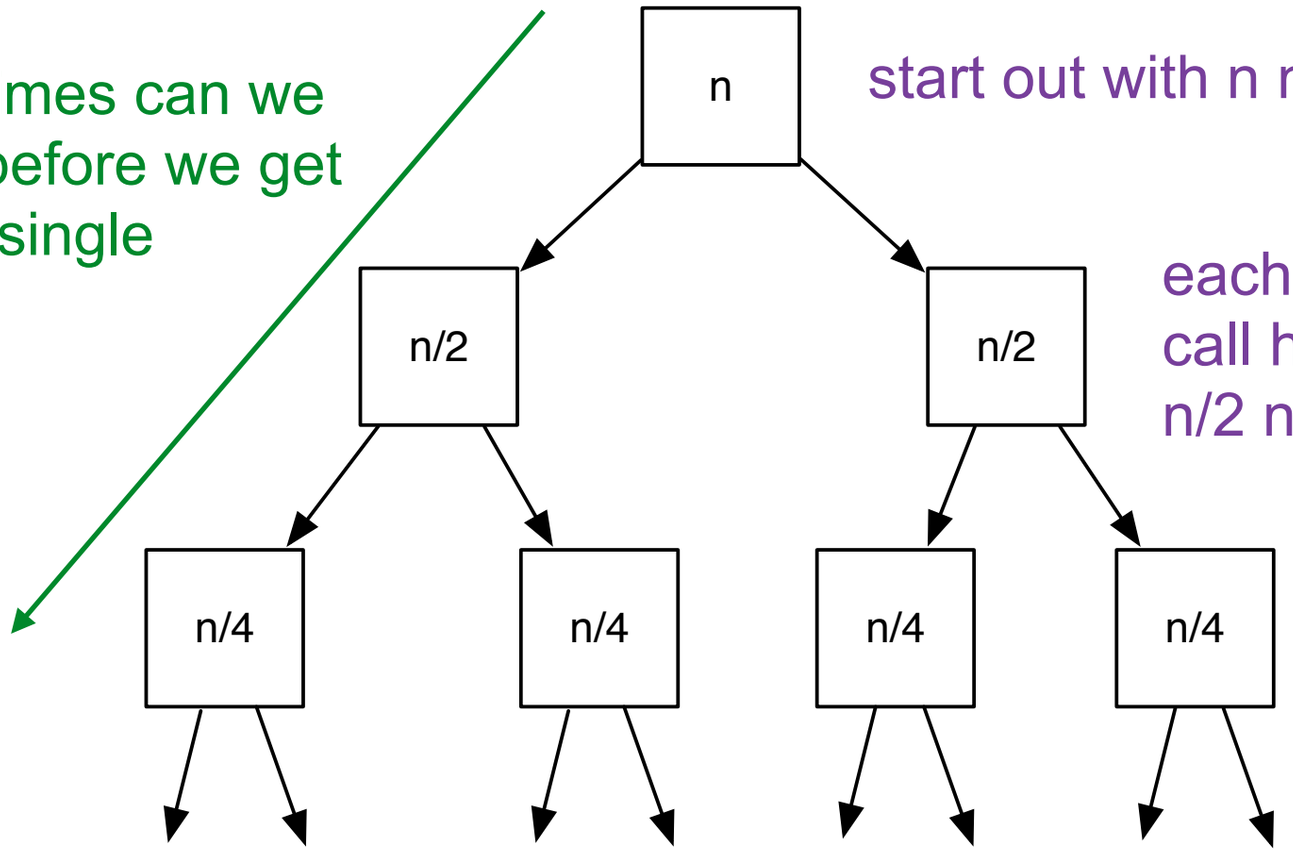
- What if luckily inList[0] was always the median of the remaining numbers?



Runtime of Quicksort

- What if luckily inList[0] was always the median of the remaining numbers?

How many times can we divide by 2 before we get to a list of a single number?



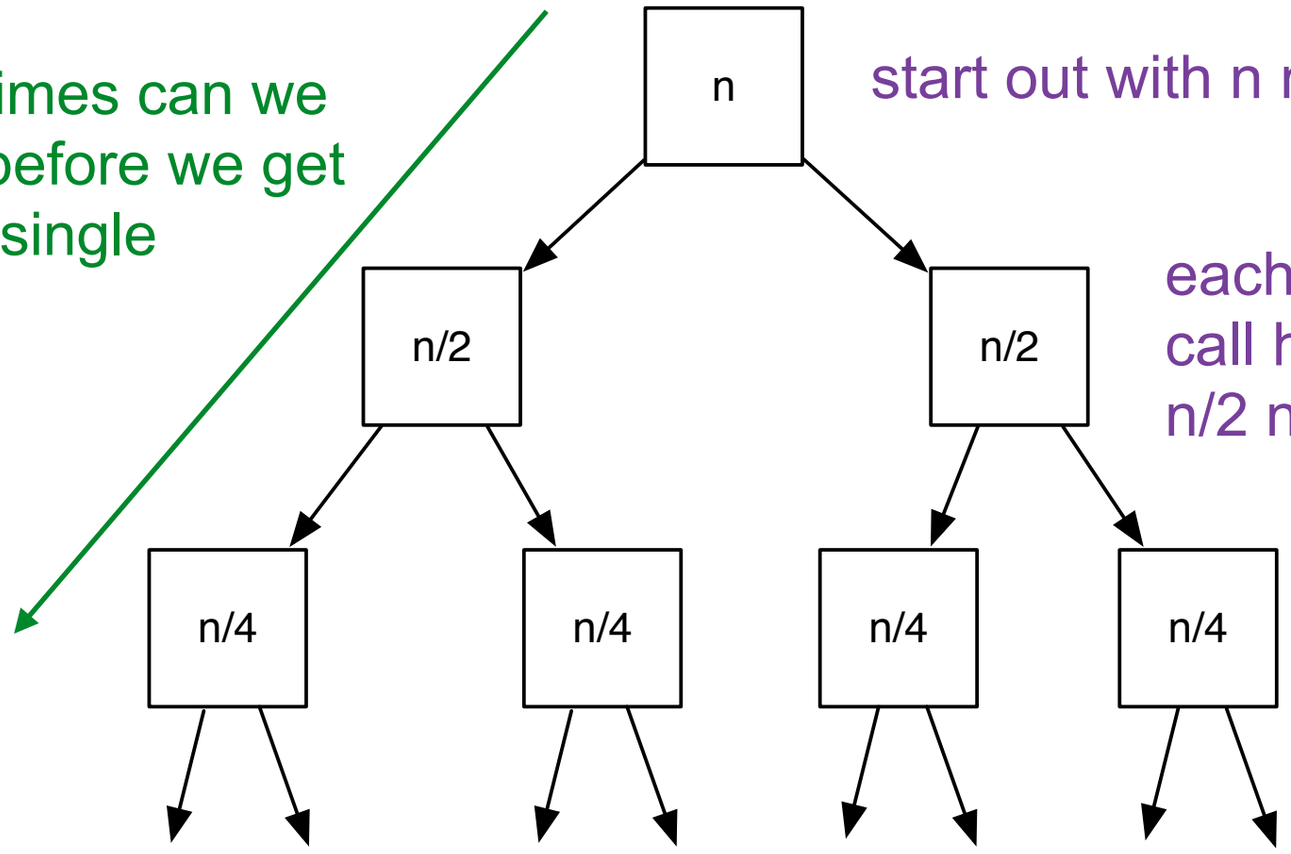
start out with n numbers

each recursive quickSort call has to deal with about n/2 numbers

Runtime of Quicksort

- What if luckily inList[0] was always the median of the remaining numbers?

How many times can we divide by 2 before we get to a list of a single number?



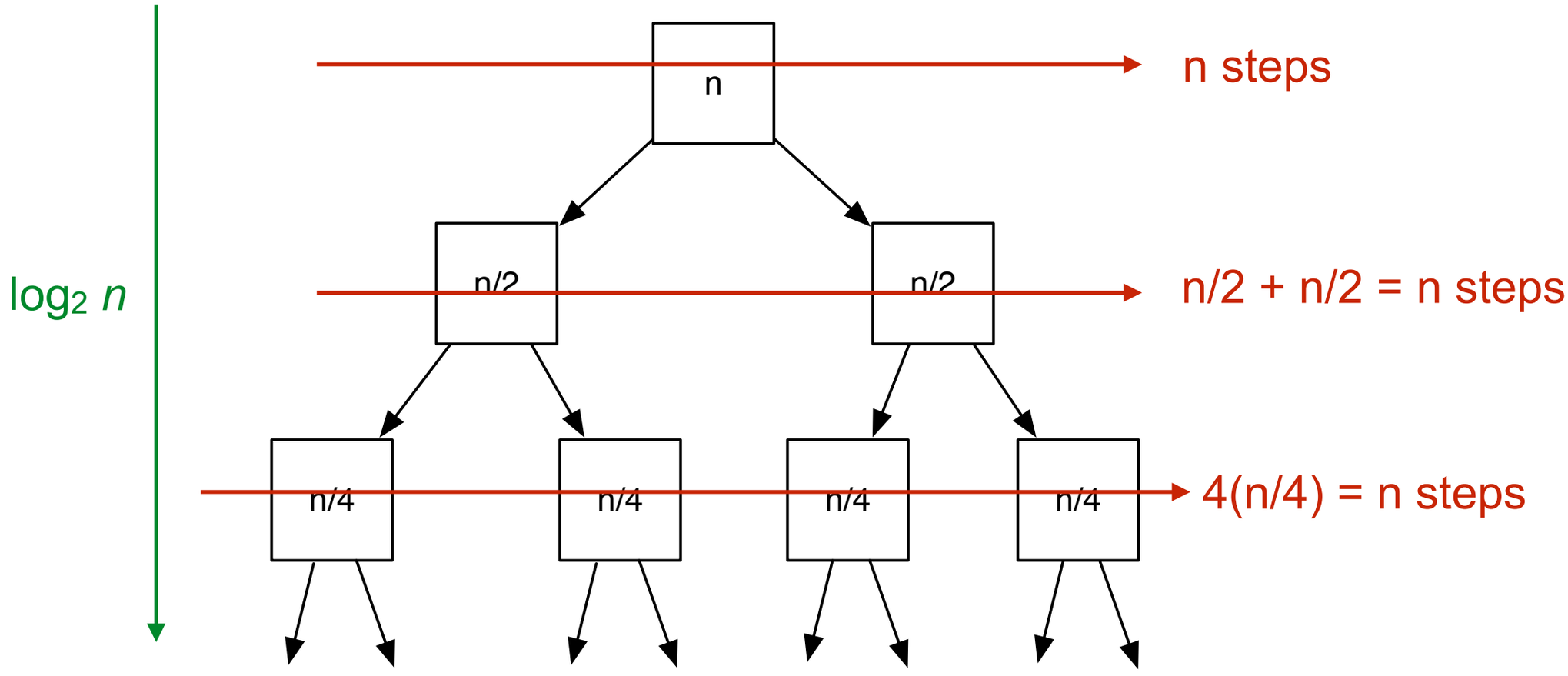
start out with n numbers

each recursive quickSort call has to deal with about n/2 numbers

About $\log_2 n$ by the same reasoning as with power()

Runtime of Quicksort, 2

Each quickSort call calls partition() which does work proportional to the size of the remaining list.



$\log_2 n$ levels, each with about n steps = $n \log n$ total steps

WORST-CASE Runtime of Quicksort

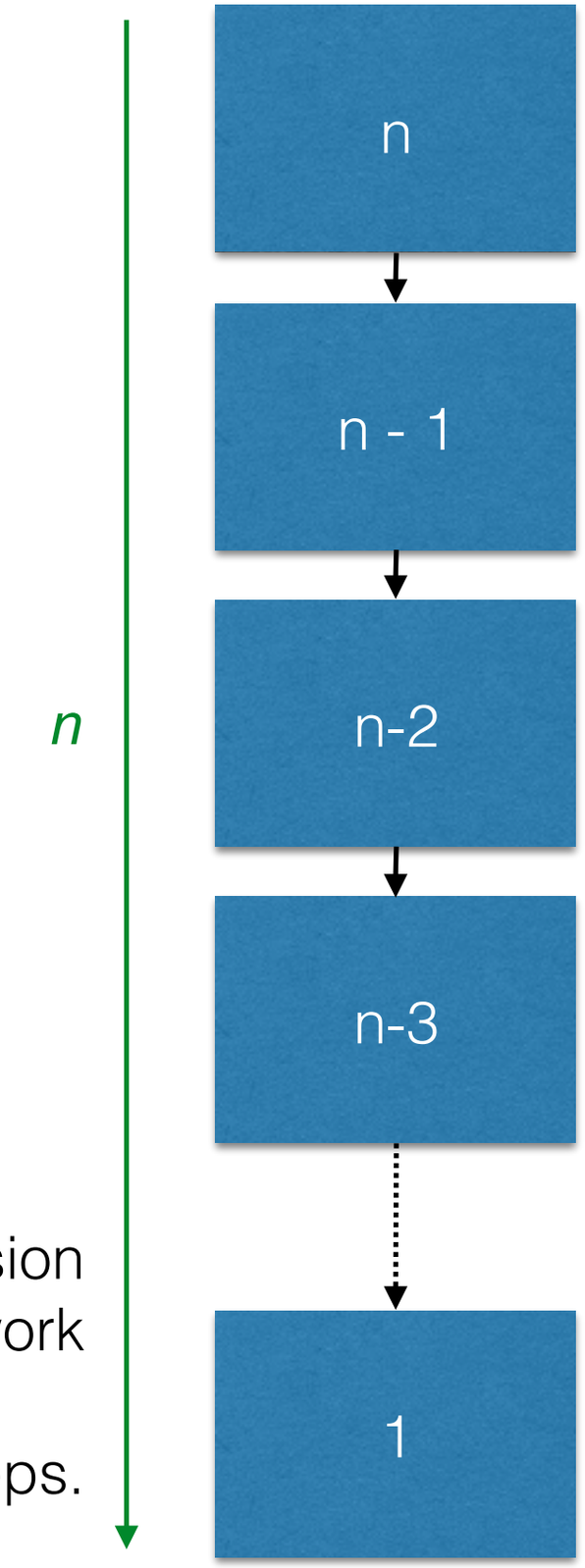
- What if we didn't get lucky?
  What would the worst pattern of partitions be?

WORST-CASE Runtime of Quicksort

- What if we didn't get lucky?
  What would the worst pattern of partitions be?

When the partitions are not balanced (say empty vs. everything else)

Say: when the input is already sorted.



n levels of recursion
each doing about *n* work

= n^2 steps.

Binary Search

Searching for an item in a sorted list

- Let S be a sorted slice
- How would we find the item with value *k* ?

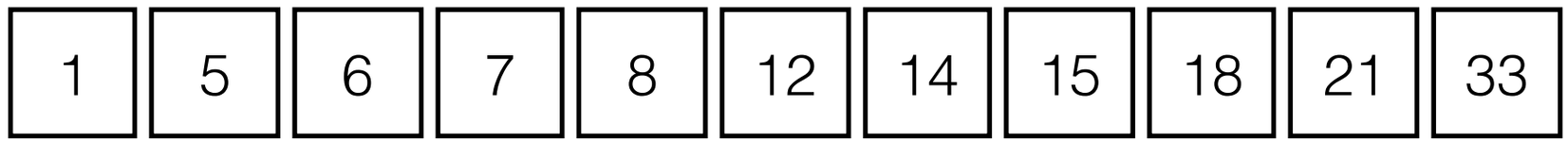
Option 1: Start at the beginning of S and walk through it until you find k:

```
for i, x := range S {
   if x == k {
      return i
   }
}
```

Option 2: the phone book algorithm: open the phone book at the middle, if the item you're looking for is in the first half, go to the middle of the first half, and so on:

Binary Search: The Phone Book Algorithm

Find 8:



At each step:

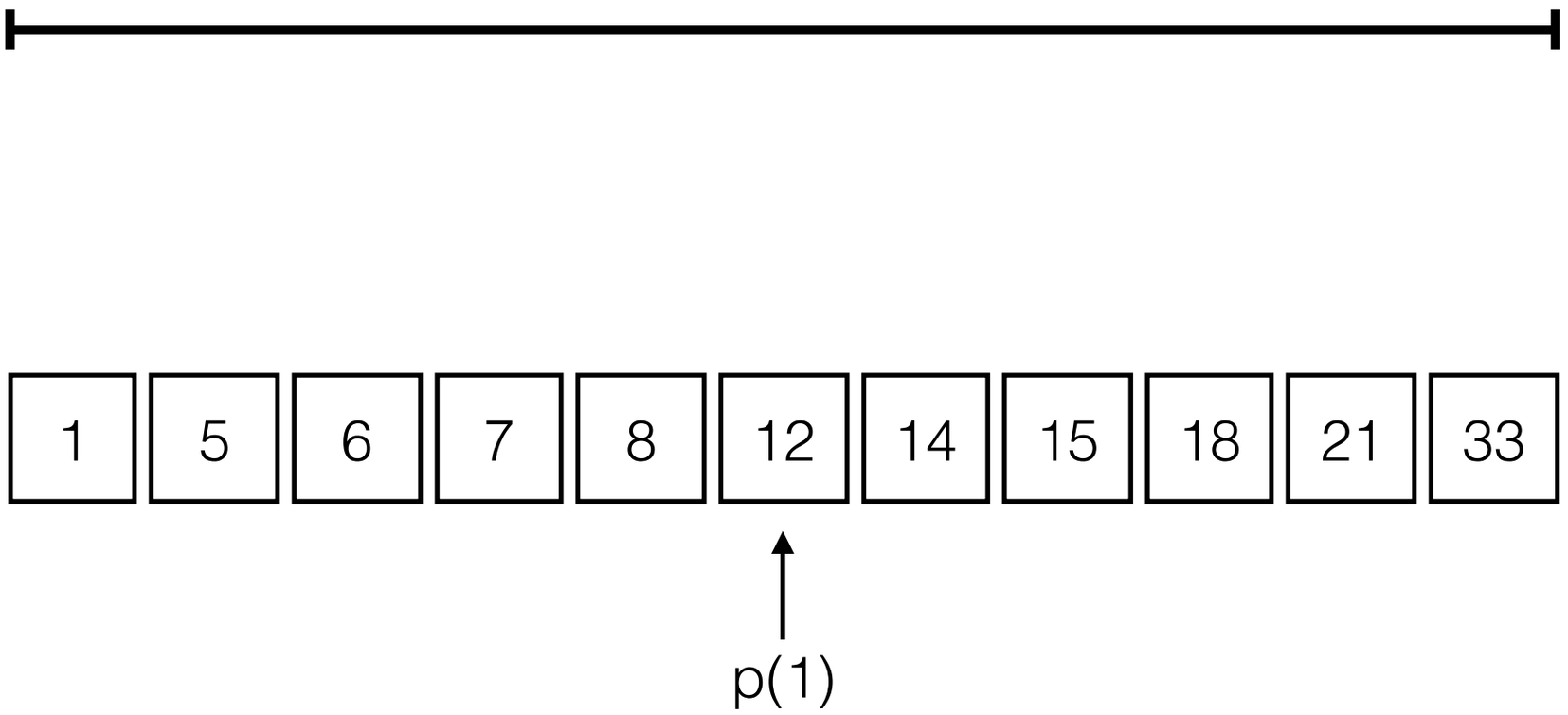
If *p == item, report it

if *p > item, look to the left half

if *p < item, look to the right half

Binary Search: The Phone Book Algorithm

Find 8:



At each step:

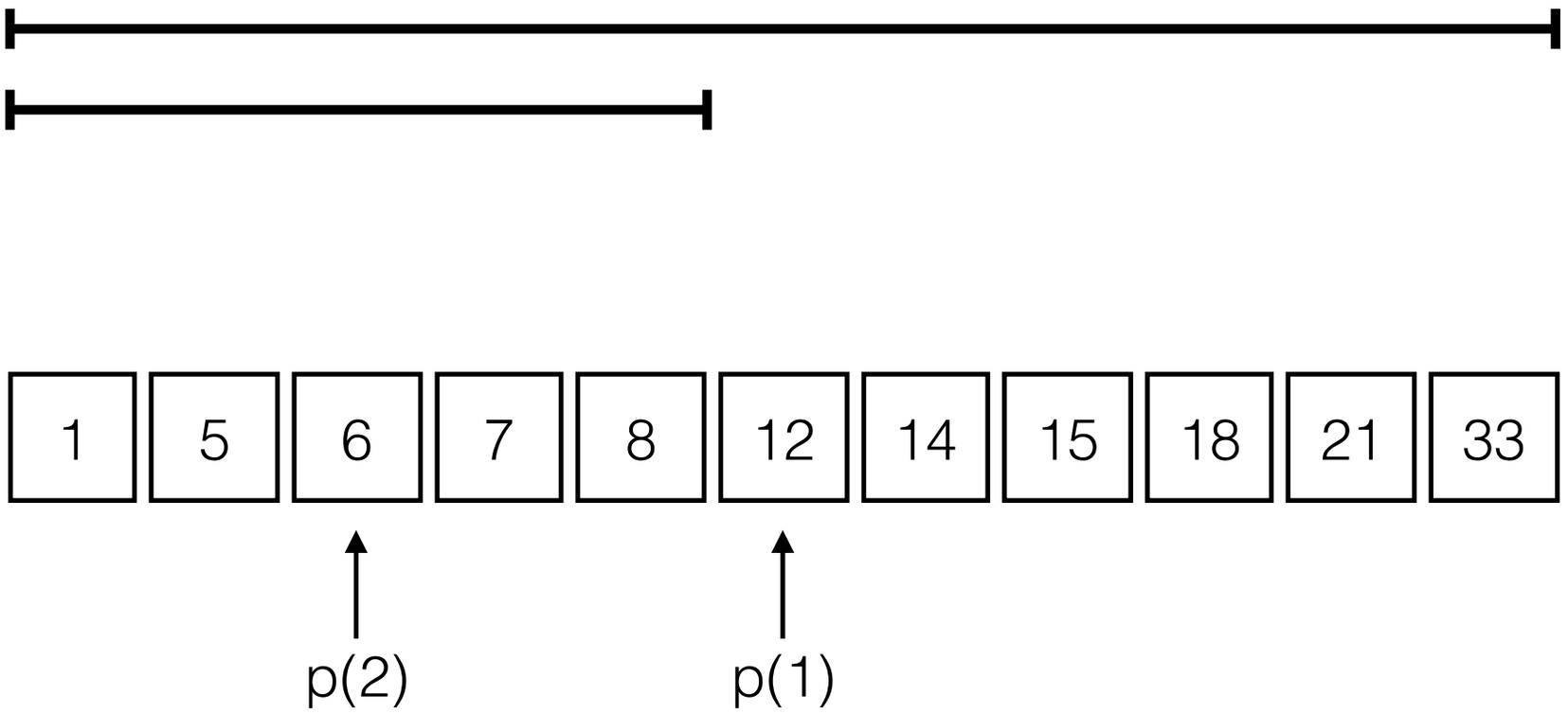
If *p == item, report it

if *p > item, look to the left half

if *p < item, look to the right half

Binary Search: The Phone Book Algorithm

Find 8:



At each step:

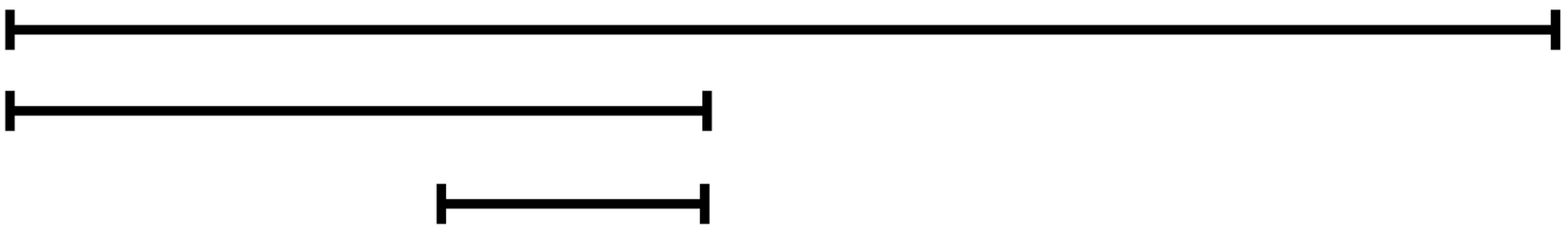
If *p == item, report it

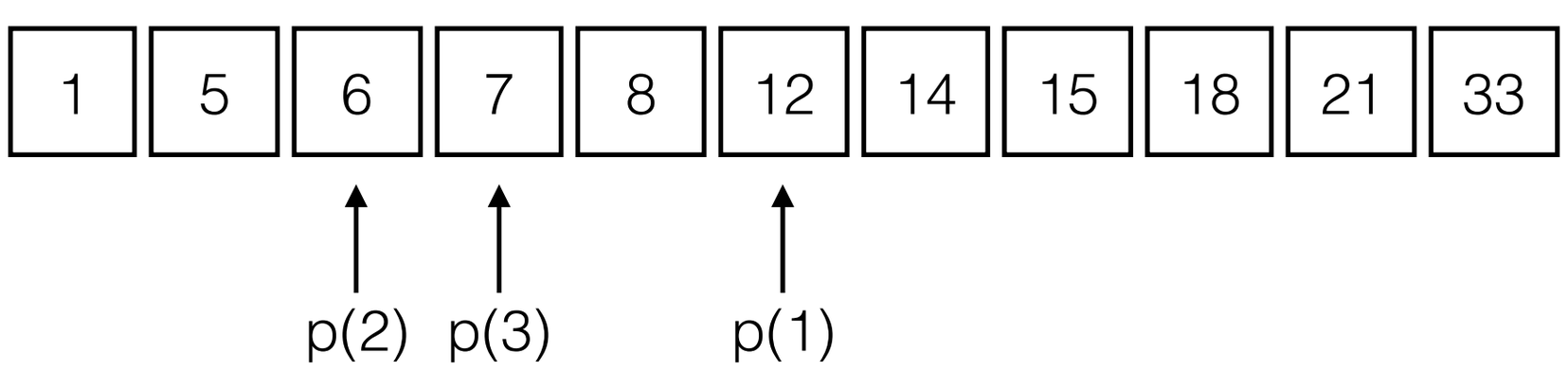
if *p > item, look to the left half

if *p < item, look to the right half

Binary Search: The Phone Book Algorithm

Find 8:





At each step:

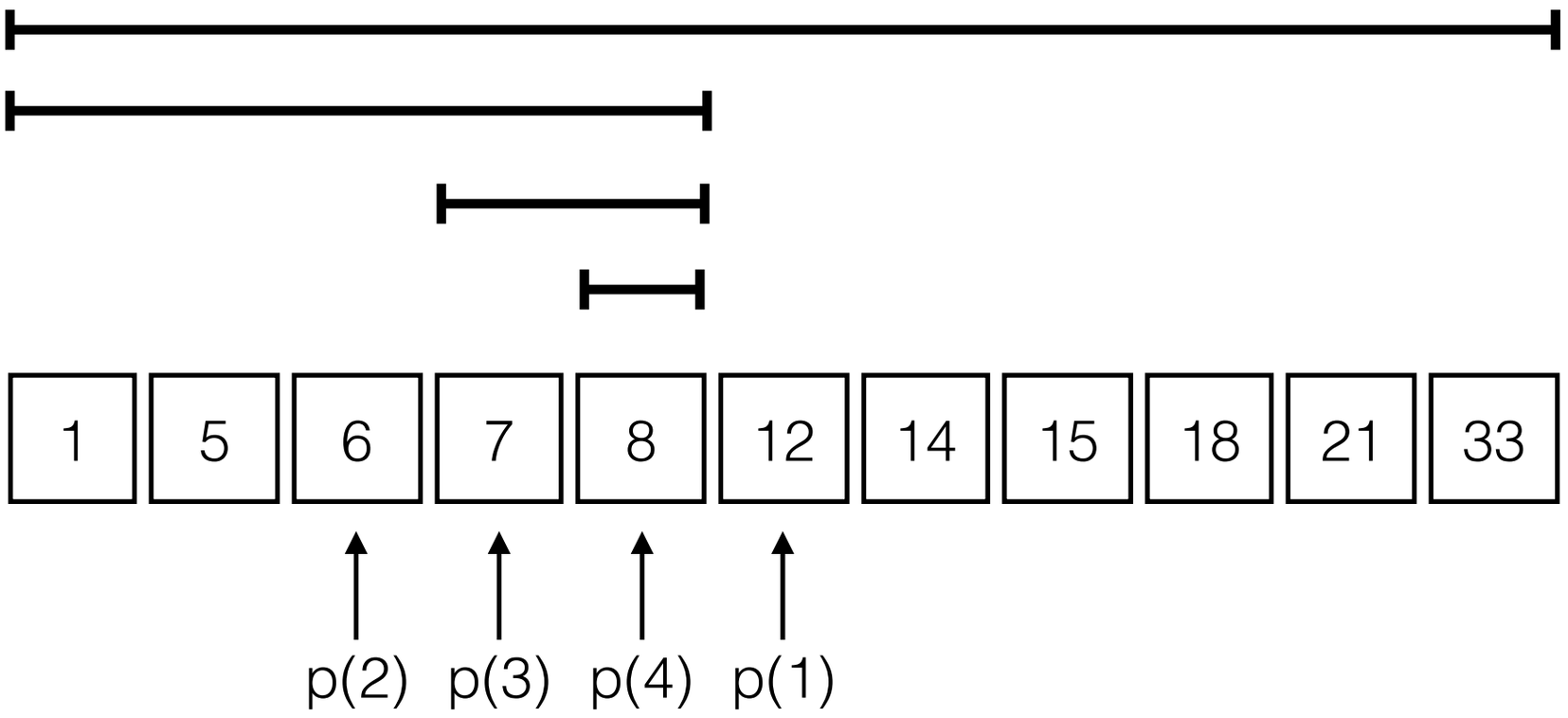
If *p == item, report it

if *p > item, look to the left half

if *p < item, look to the right half

Binary Search: The Phone Book Algorithm

Find 8:



At each step:

If *p == item, report it

if *p > item, look to the left half

if *p < item, look to the right half

Binary Search, Code

```
func binarySearch(inList []int, k int) (int, bool) {
    left, right := 0, len(inList)-1
    for left <= right {
        mid := (right + left) / 2
        if inList[mid] == k {
            return mid, true
        } else if inList[mid] > k {
            right = mid - 1
        } else if inList[mid] < k {
            left = mid + 1
        }
    }
    return 0, false
}
```

Binary Search, Recursive

```
func binarySearchRecur(inList []int, k int) (int, bool) {
    if len(inList) == 0 {
        return 0, false
    }
    mid := len(inList)/2
    if inList[mid] == k {
        return mid, true
    } else if inList[mid] > k {
        if mid == 0 {
            return 0, false
        }
        return binarySearchRecur(inList[:mid-1], k)
    } else {
        p, f := binarySearchRecur(inList[mid+1:], k)
        return mid+1+p, f
    }
}
```

If the middle of the list = k, we found it.

If k must be in the left half

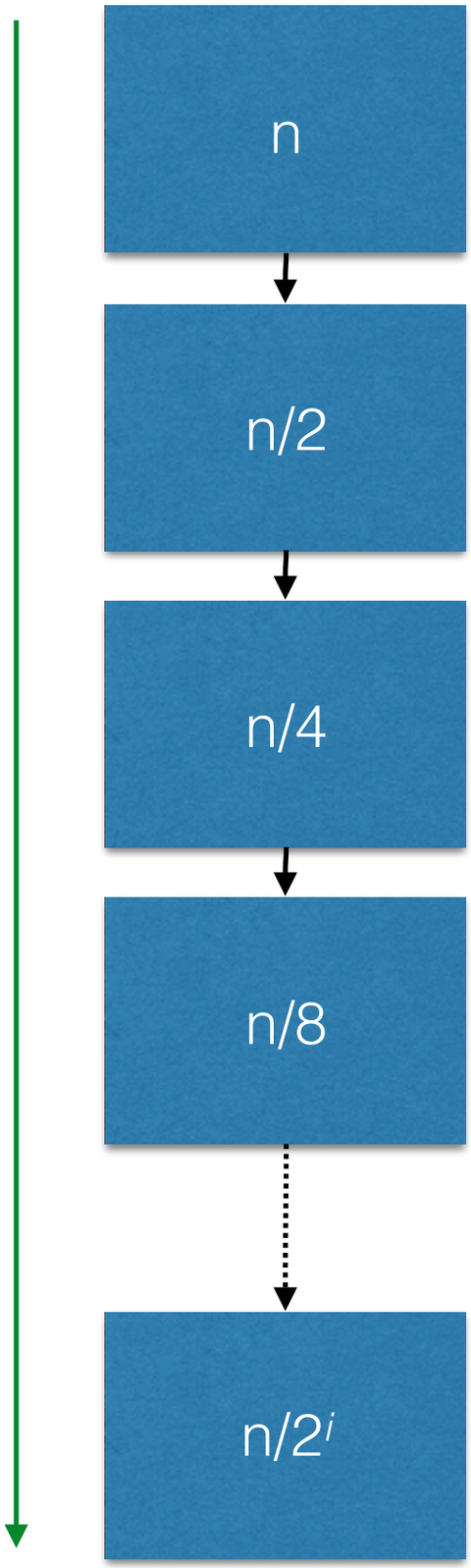
check whether we're falling off the left end of the array

Binary Search Runtime

- What similar thing have see that could tell us the runtime?

Binary Search Runtime

- What similar thing have see that could tell us the runtime?



Will “recurse” $\log_2 n$ times.
At each level, we do a constant amount of work.

Runtime for binary search $\approx \log_2 n$

Summary

- Recursion is implemented in the same way as any other function call: the local variables are stored on the stack.

- Divide and conquer: good way to speed up algorithms (binary search, power, quicksort)

- Worst-case runtime for insertion sort is about n^2 steps

- Worst-case runtime for quick sort is about n^2 steps, but in practice, you expect around n log n steps.