THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 2: Review of Basic Concepts
(with thanks to Harald Søndergaard)

Toby Murray

✉ toby.murray@unimelb.edu.au

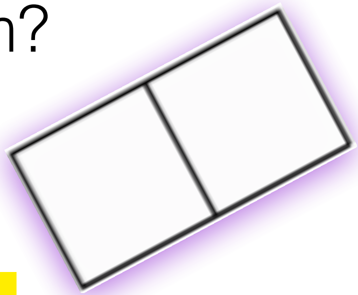
DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

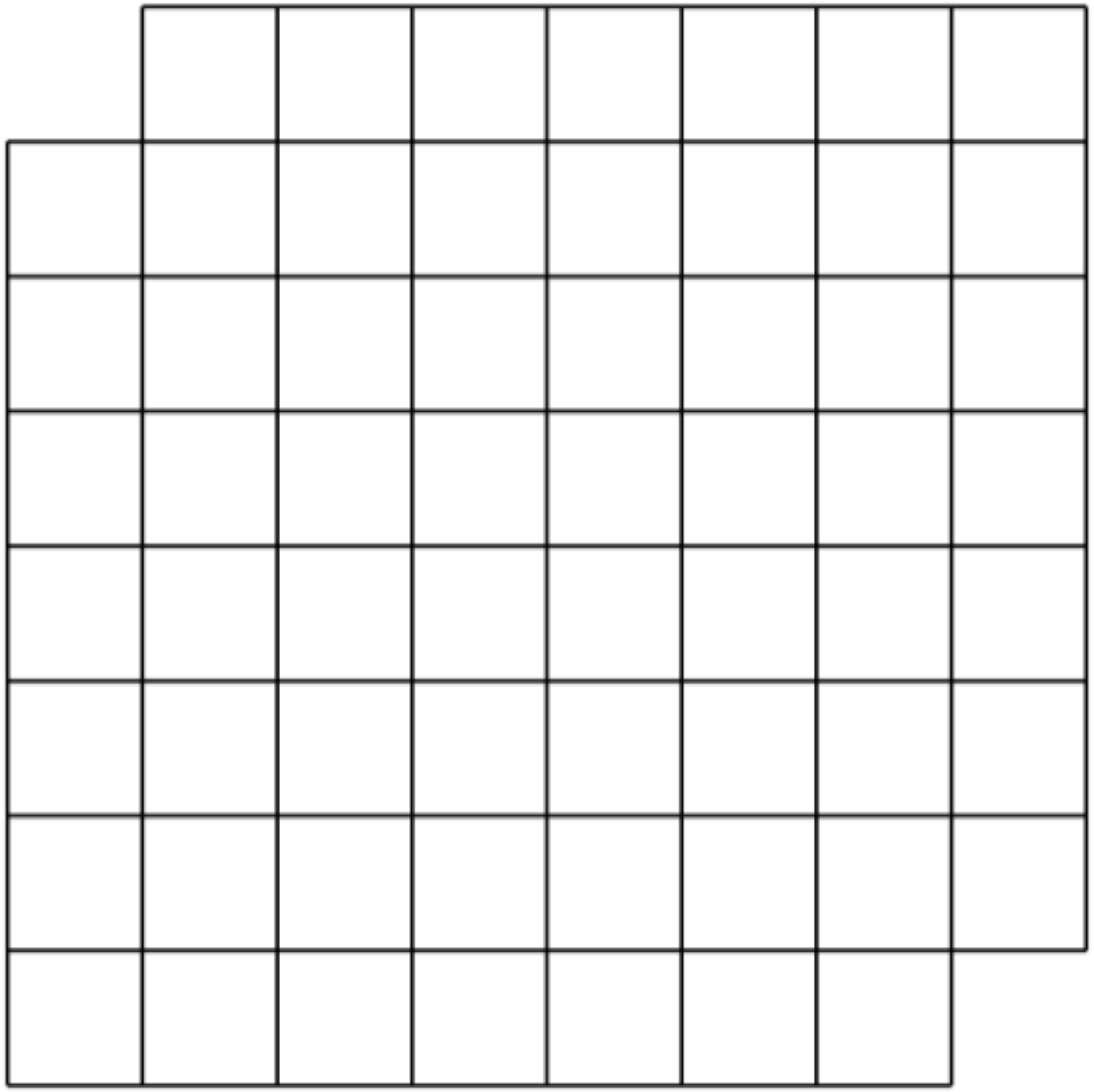
@tobycmurray

Approaching a problem

- Can we cover this board with 31 tiles of the following form?

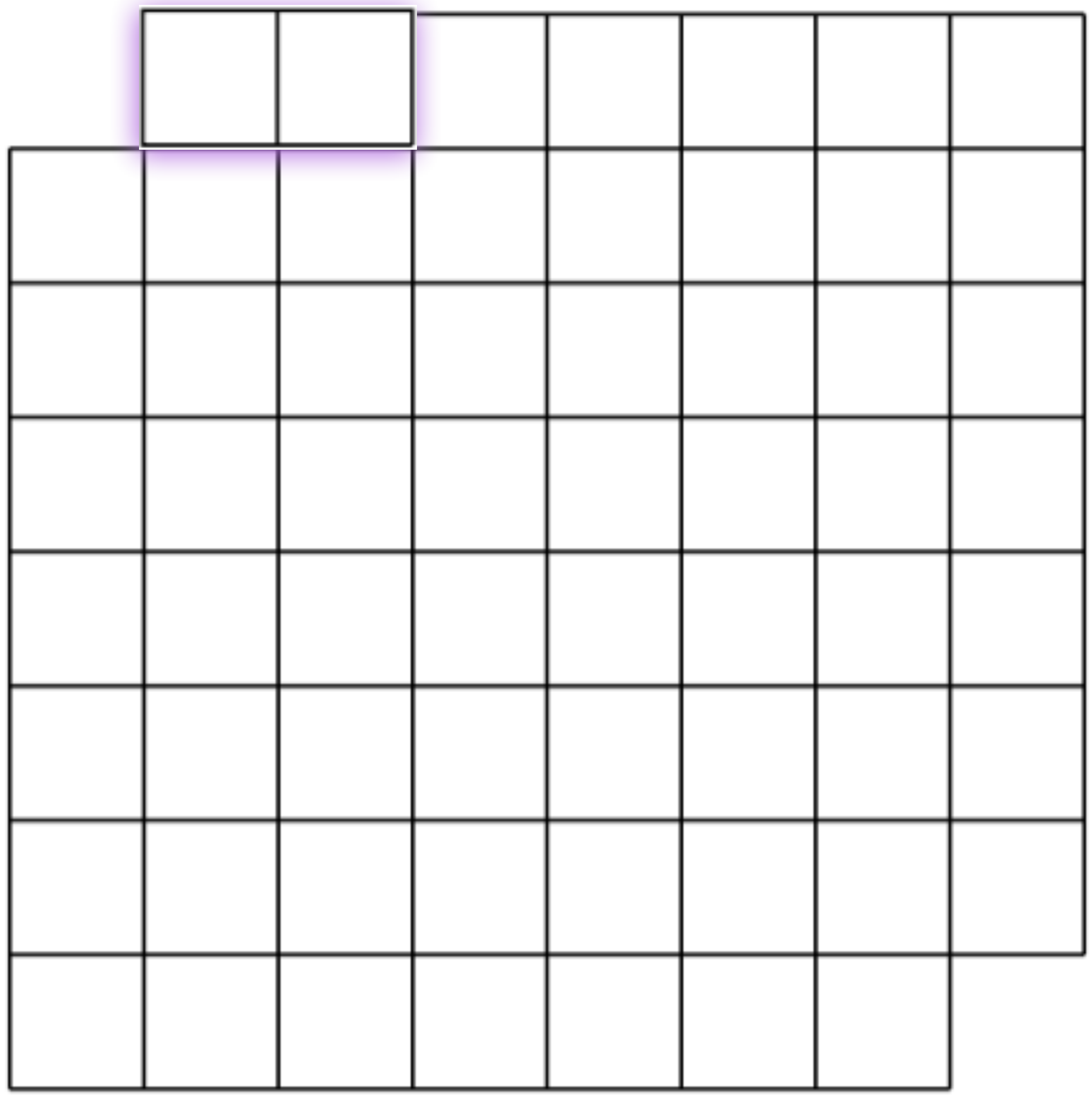


- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.



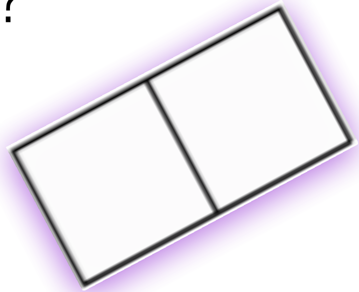
Approaching a problem

- Can we cover this board with 31 tiles of the following form?
- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.

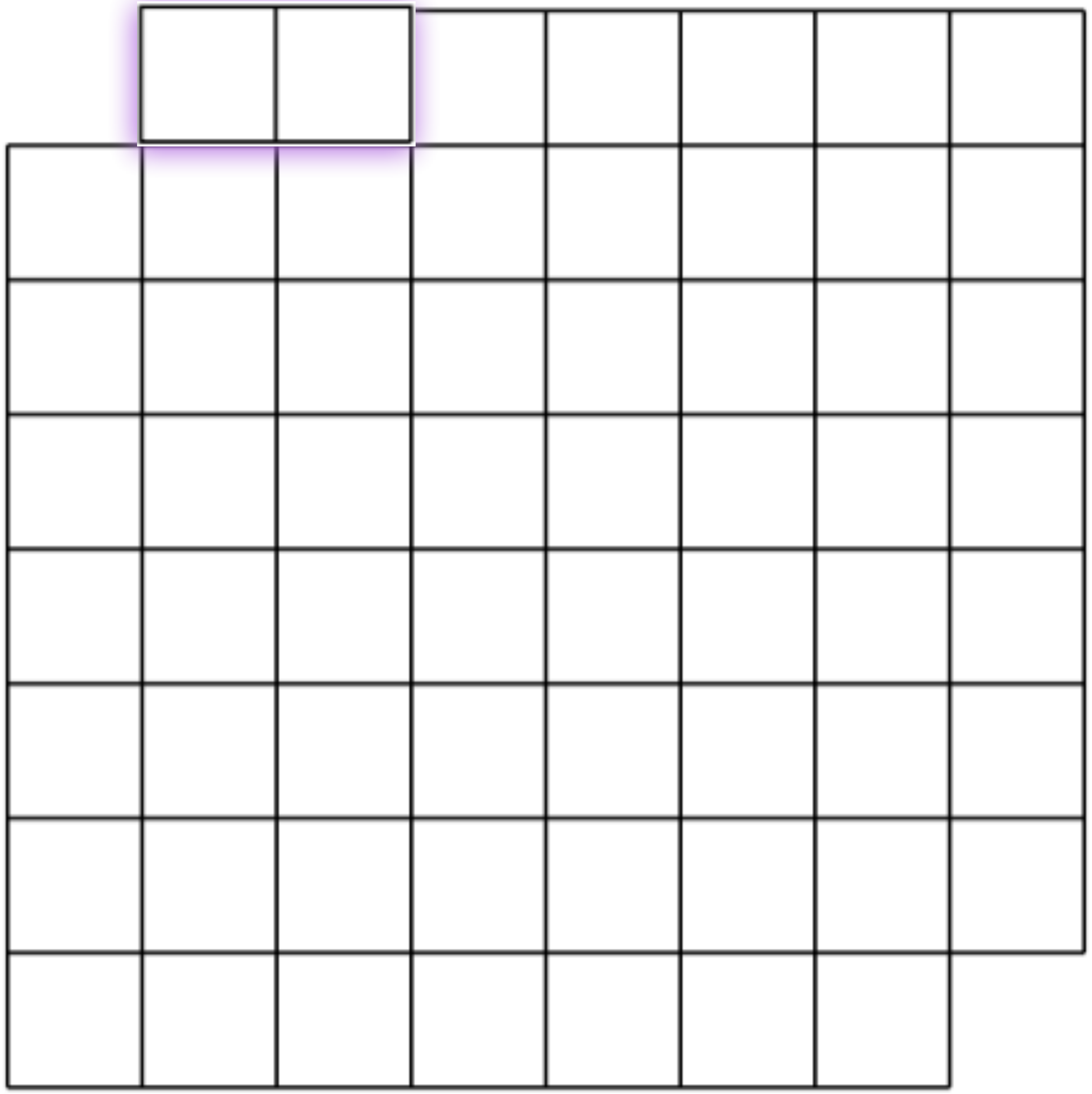


Approaching a problem

- Can we cover this board with 31 tiles of the following form?

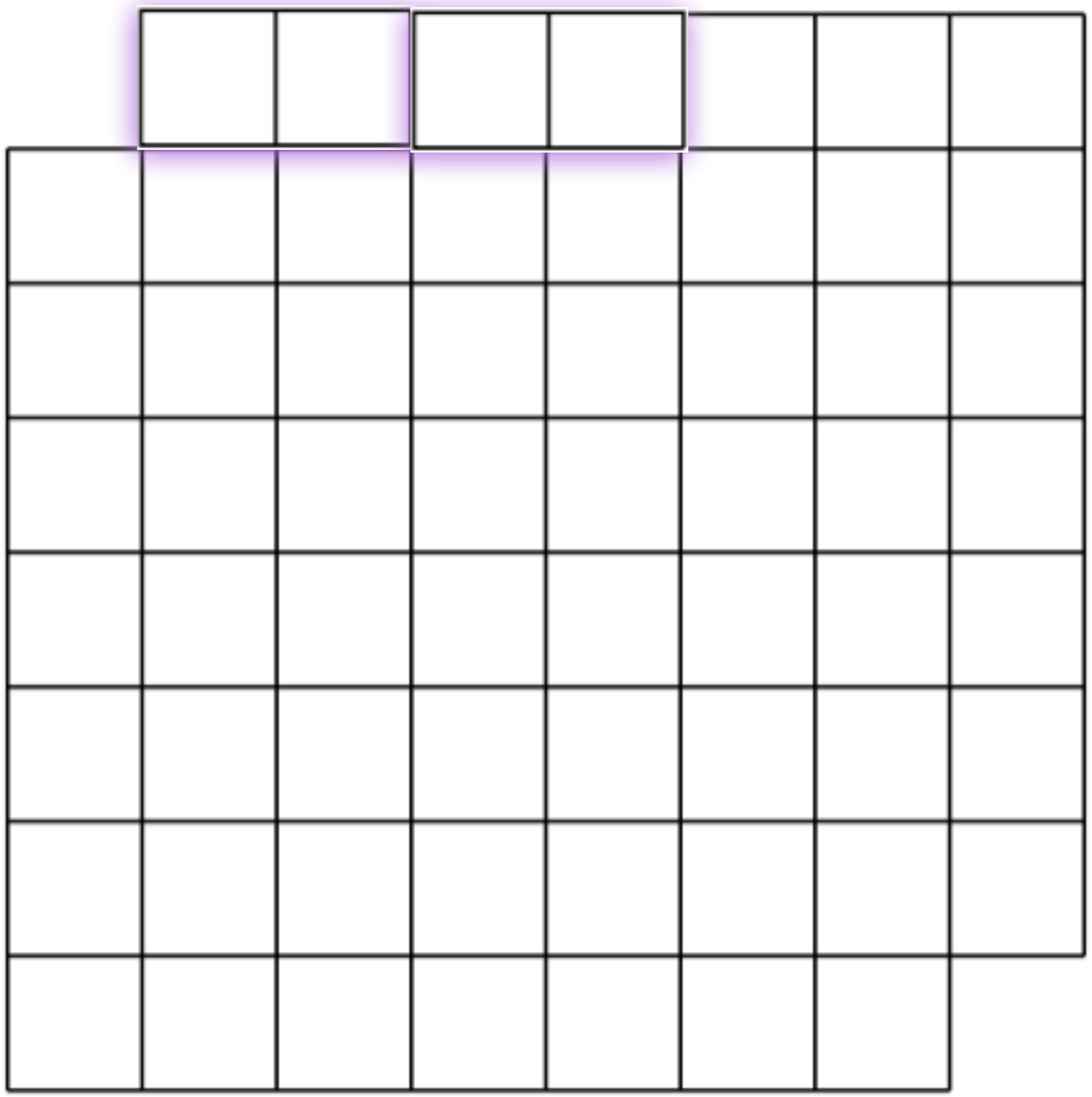


- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.



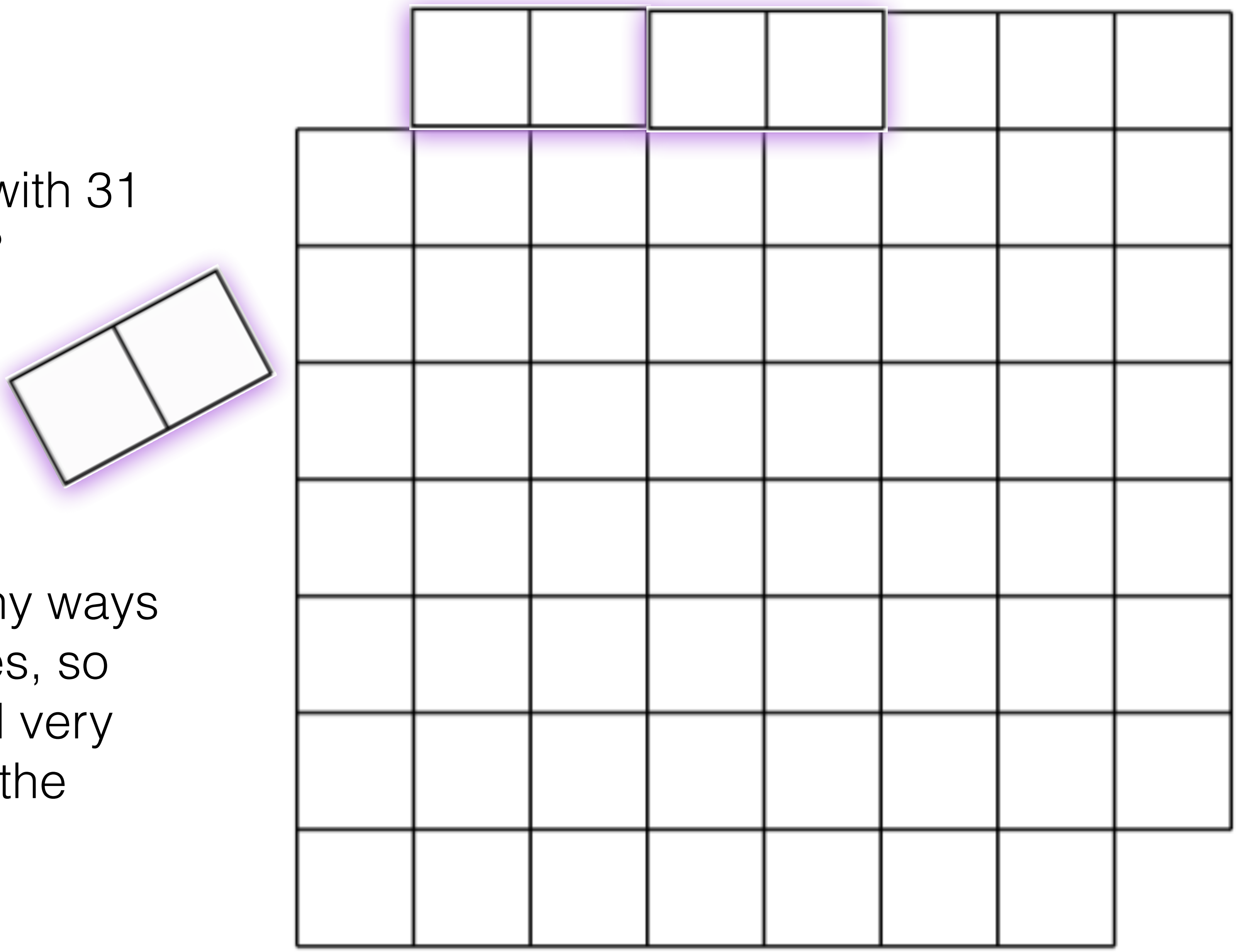
Approaching a problem

- Can we cover this board with 31 tiles of the following form?
- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.



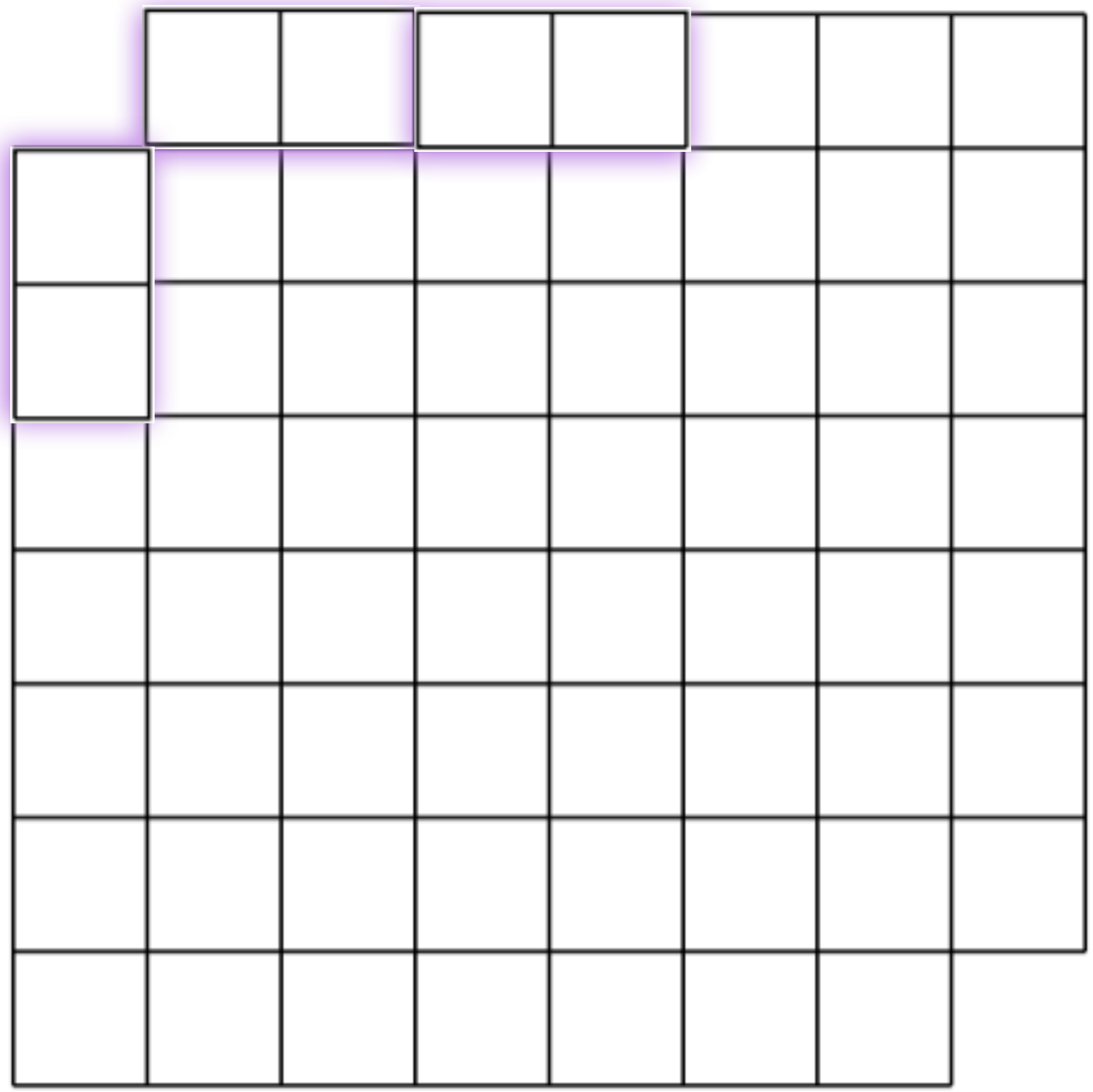
Approaching a problem

- Can we cover this board with 31 tiles of the following form?
- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.

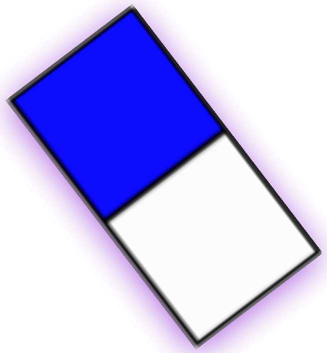


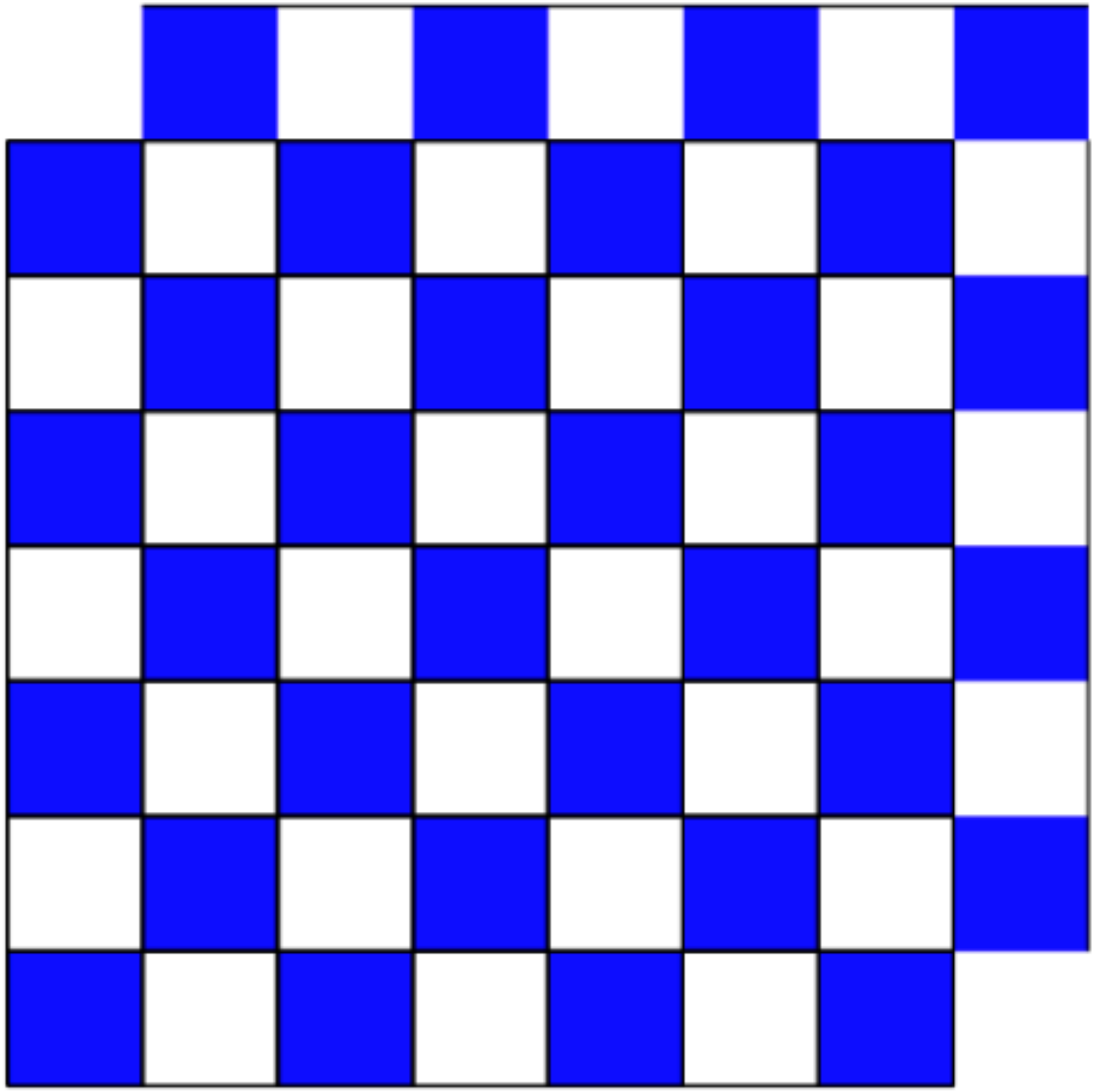
Approaching a problem

- Can we cover this board with 31 tiles of the following form?
- This is the **mutilated checkerboard problem**.
- There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.



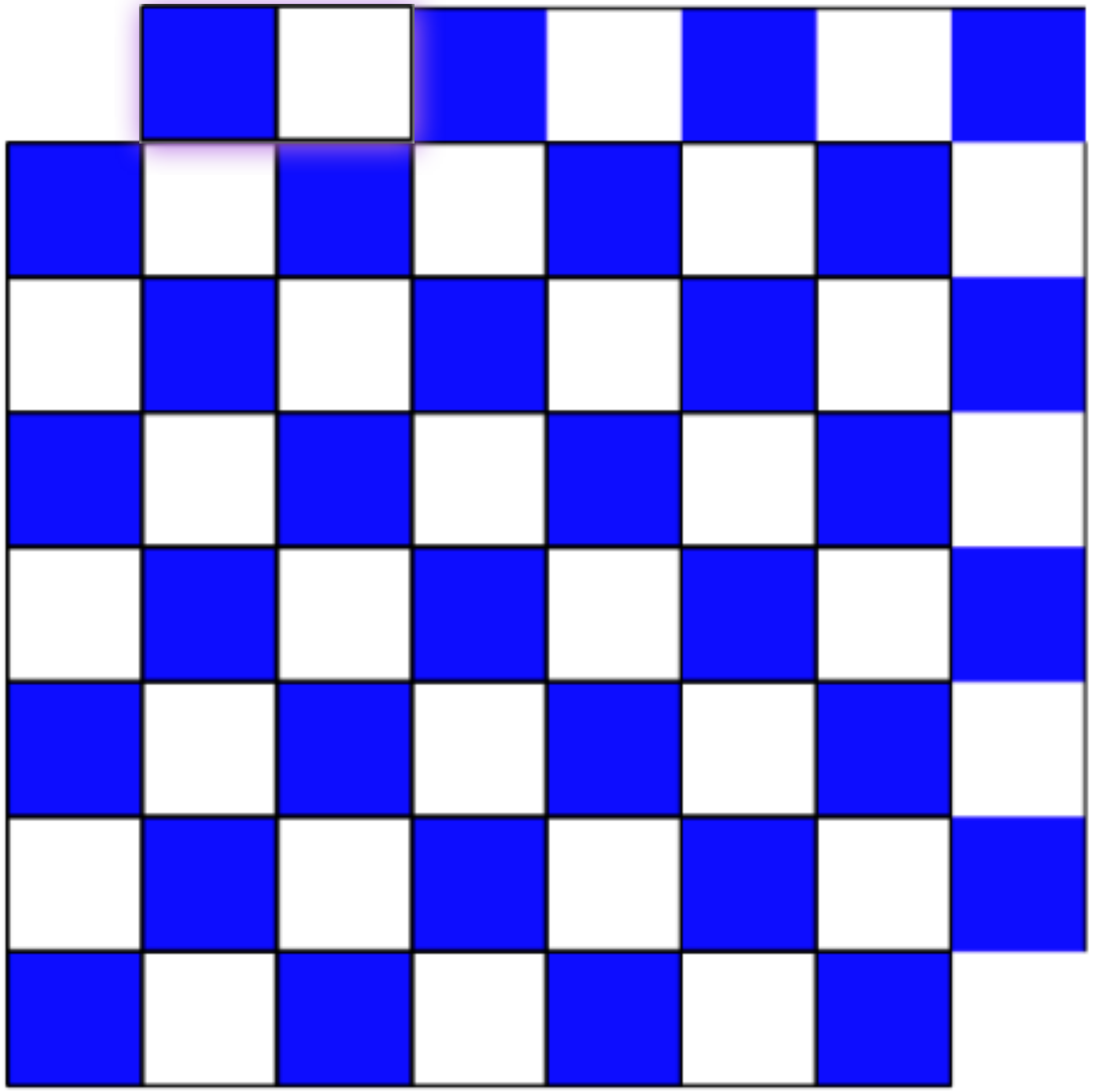
Transform and Conquer?
Use abstraction?





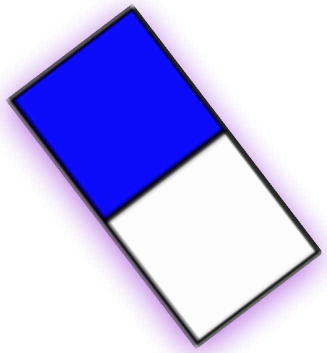
- Can we cover this board with 31 tiles of the form shown?
- Why can we quickly determine that the answer is no?
- **Hint:** Using the way the squares are coloured helps.

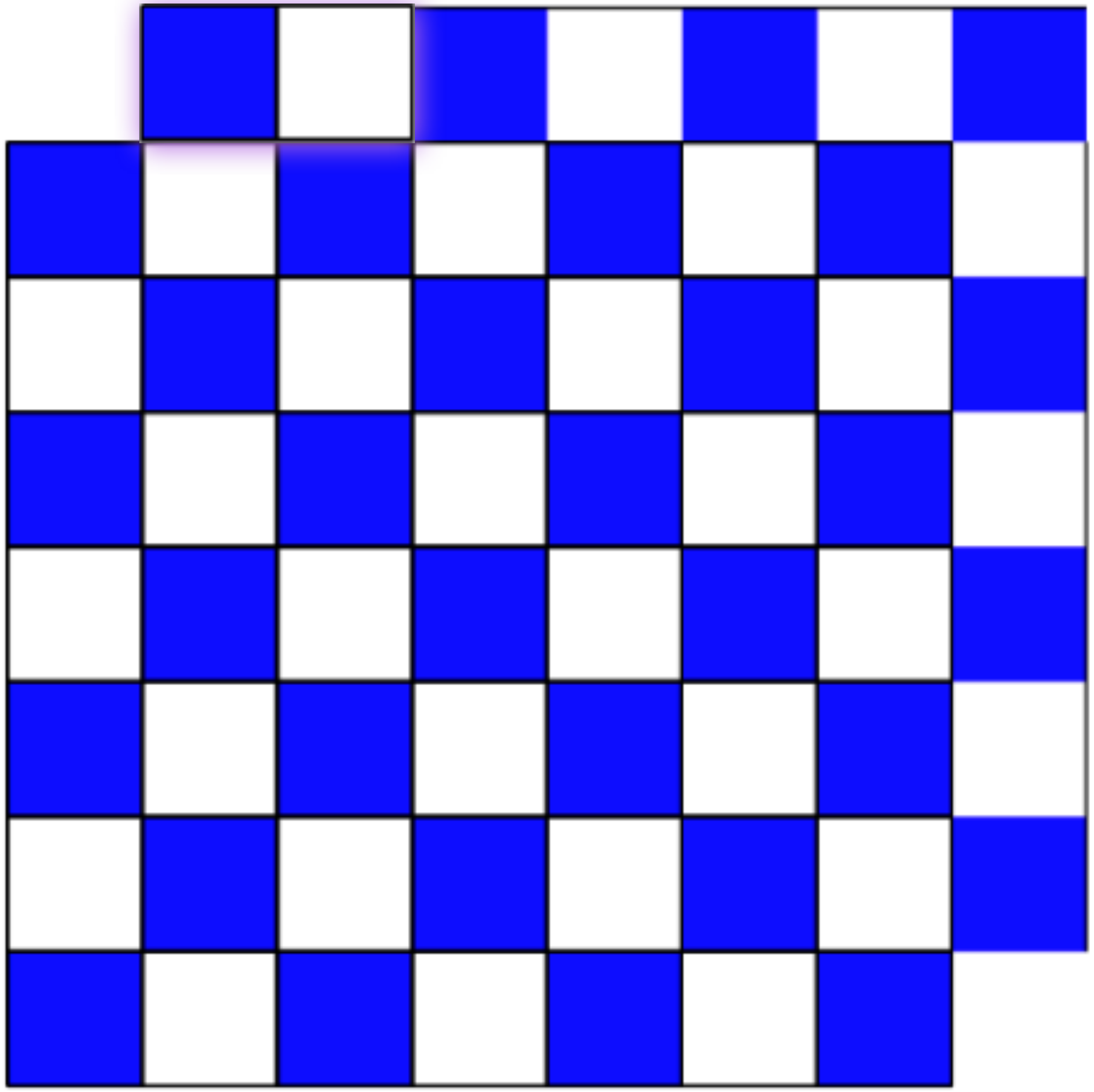
Transform and Conquer?
Use abstraction?



- Can we cover this board with 31 tiles of the form shown?
- Why can we quickly determine that the answer is no?
- **Hint:** Using the way the squares are coloured helps.

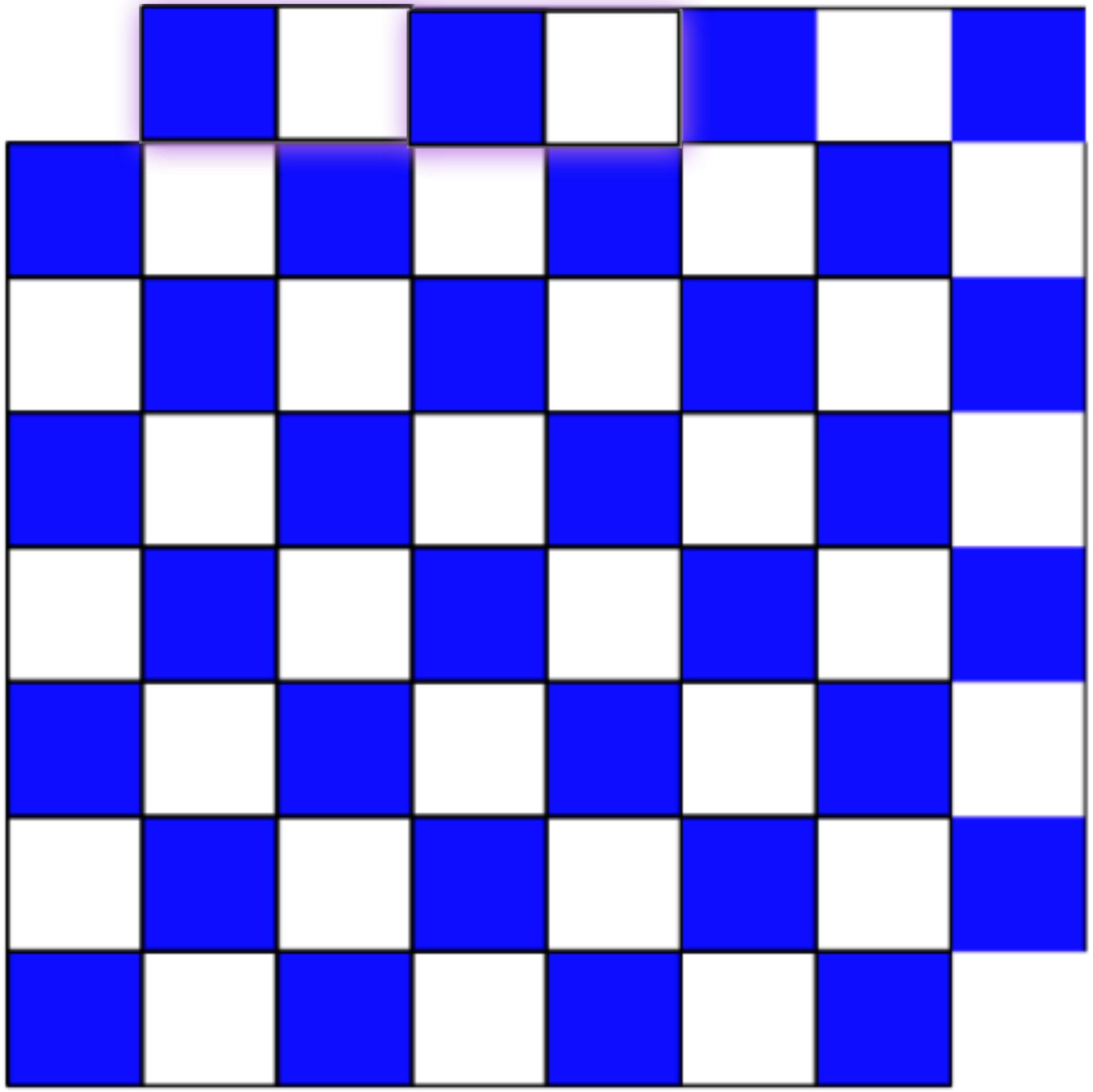
Transform and Conquer?
Use abstraction?





- Can we cover this board with 31 tiles of the form shown?
- Why can we quickly determine that the answer is no?
- **Hint:** Using the way the squares are coloured helps.

Transform and Conquer?
Use abstraction?



- Can we cover this board with 31 tiles of the form shown?
- Why can we quickly determine that the answer is no?
- **Hint:** Using the way the squares are coloured helps.

Algorithms and Data Structures

- **Algorithms**: for solving problems, transforming data.

- **Data structures**: for storing data; arranging data in a way that suits an algorithm.
  - **Linear** data structures: stacks and queues
  - Trees and graphs
  - Dictionaries
- Which data structures are you familiar with?

Exercise

- Pick you favourite data structure and describe:
  - How to insert and item into the data structure
  - How to find an item
  - How to handle duplicate items

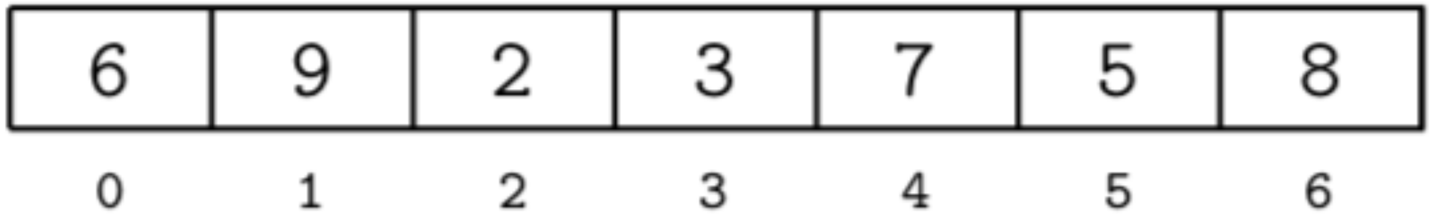
Primitive Data Structures: The Array

- An array corresponds to a sequence of consecutive cells in memory.

- Depending on programming language: `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`.

- Locating a cell, and storing or retrieving data at that cell is very fast.

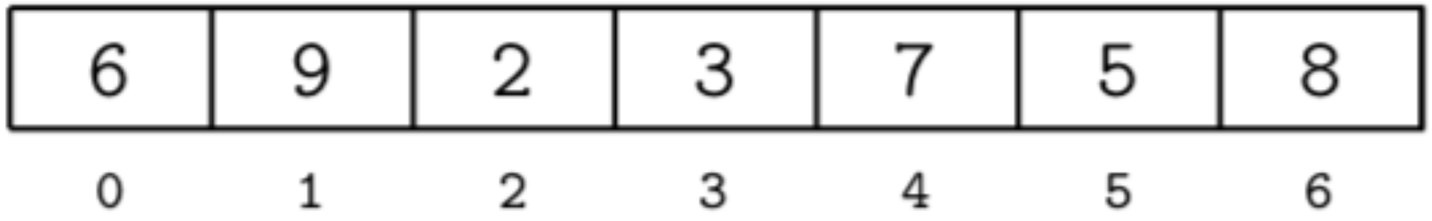
- The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.

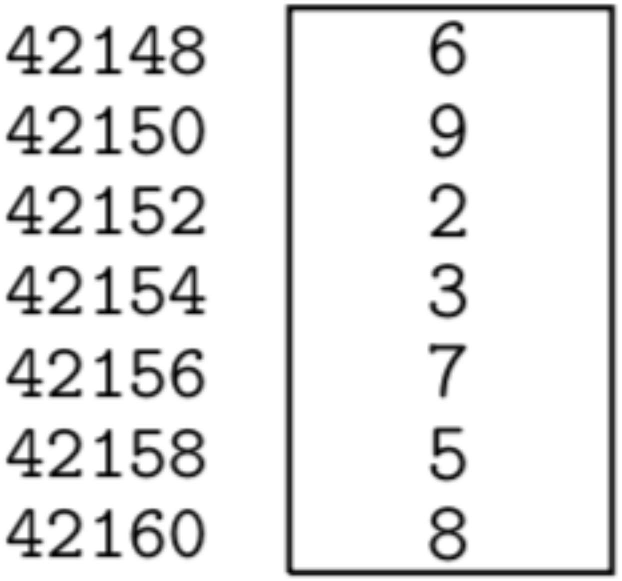


6	9	2	3	7	5	8
0	1	2	3	4	5	6

Primitive Data Structures: The Array

- An array corresponds to a sequence of consecutive cells in memory.
- Depending on programming language: `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`.
- Locating a cell, and storing or retrieving data at that cell is very fast.
- The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.





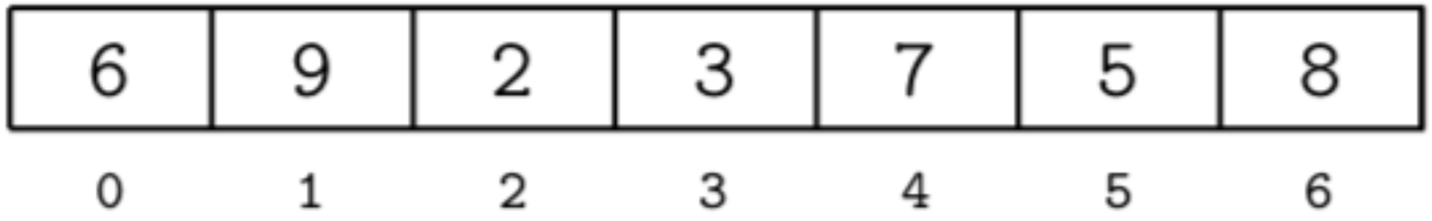
Primitive Data Structures: The Array

- An array corresponds to a sequence of consecutive cells in memory.

- Depending on programming language: `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`.

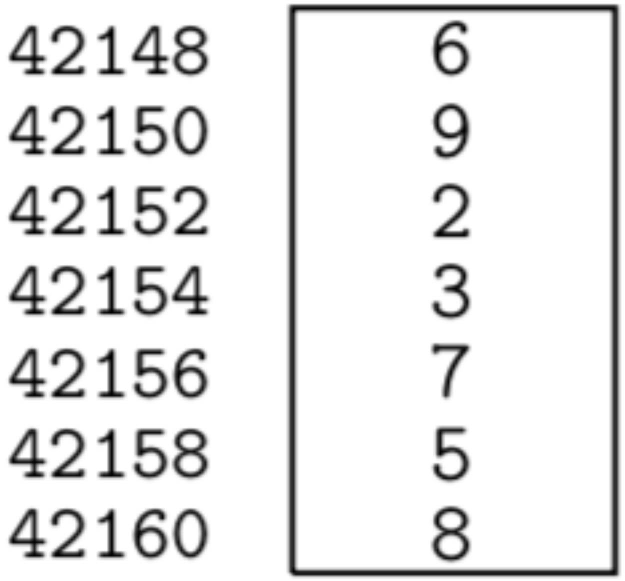
- Locating a cell, and storing or retrieving data at that cell is very fast.

- The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.



6	9	2	3	7	5	8
0	1	2	3	4	5	6

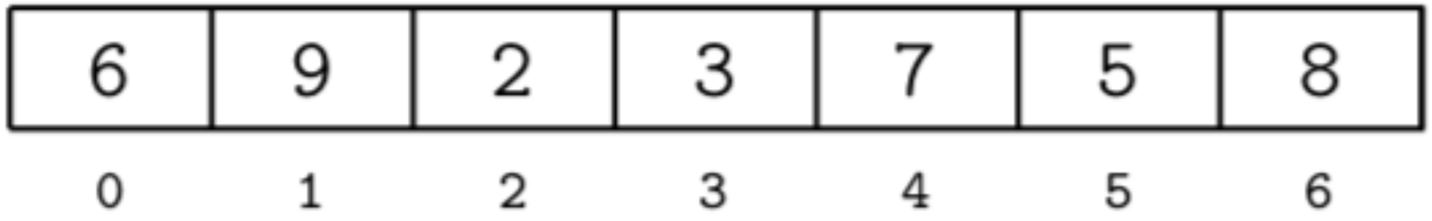
How many bytes does each integer occupy here?



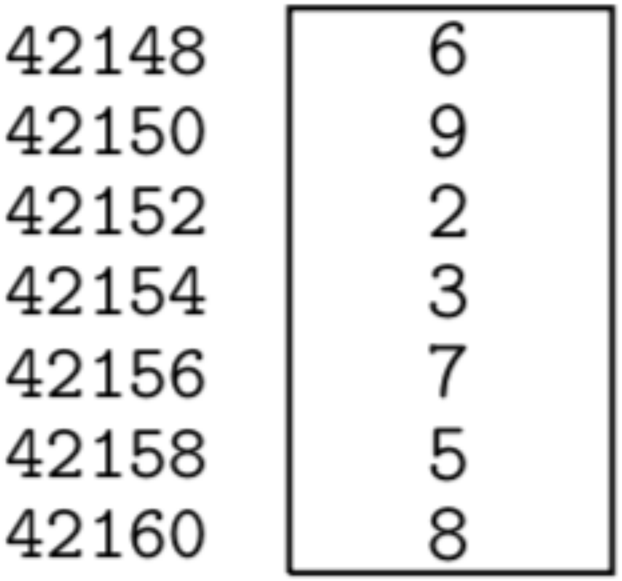
42148	6
42150	9
42152	2
42154	3
42156	7
42158	5
42160	8

Primitive Data Structures: The Array

- An array corresponds to a sequence of consecutive cells in memory.
- Depending on programming language: `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`.
- Locating a cell, and storing or retrieving data at that cell is very fast.
- The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.



6	9	2	3	7	5	8
0	1	2	3	4	5	6



42148	6
42150	9
42152	2
42154	3
42156	7
42158	5
42160	8

How many bytes does each integer occupy here?

Answer: 2 (16-bit integers)

Primitive Data Structures: The Linked List

An array `x`:

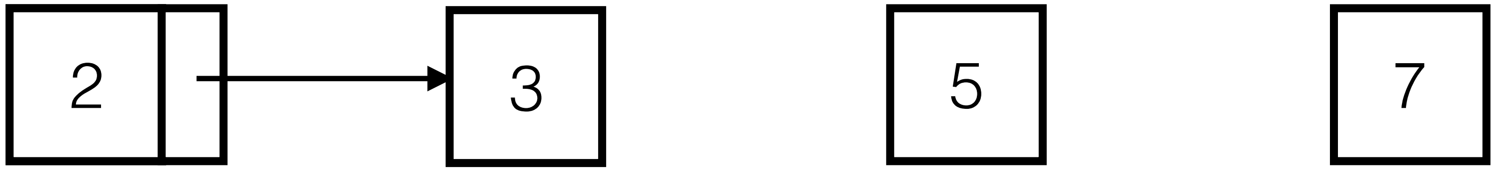
2	3	5	7

Primitive Data Structures: The Linked List

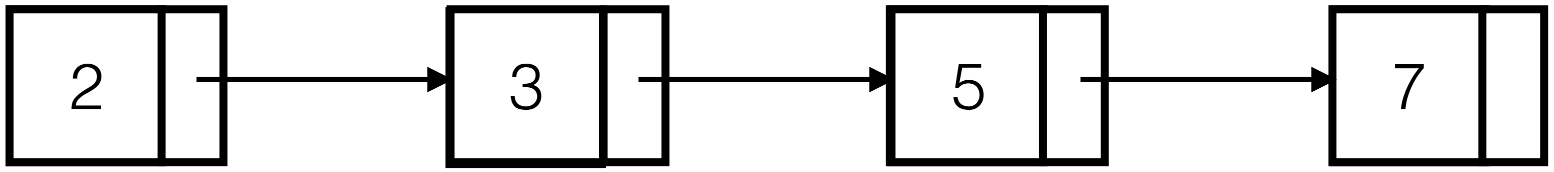
2 3 5 7

Primitive Data Structures: The Linked List

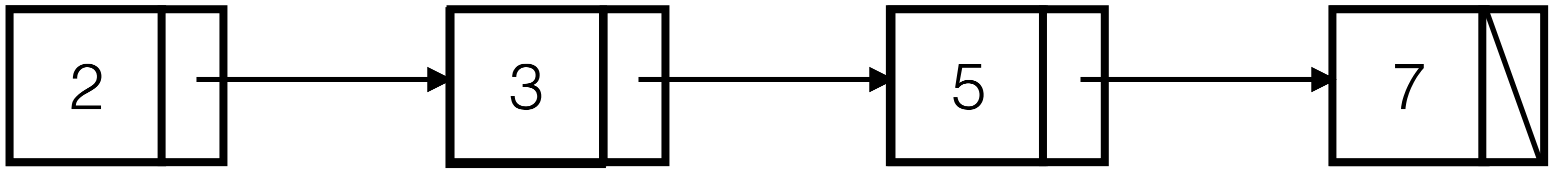
Primitive Data Structures: The Linked List



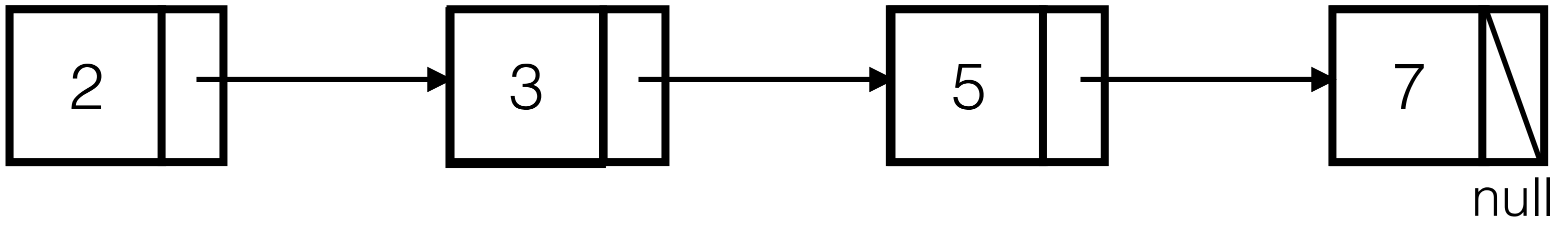
Primitive Data Structures: The Linked List



Primitive Data Structures: The Linked List

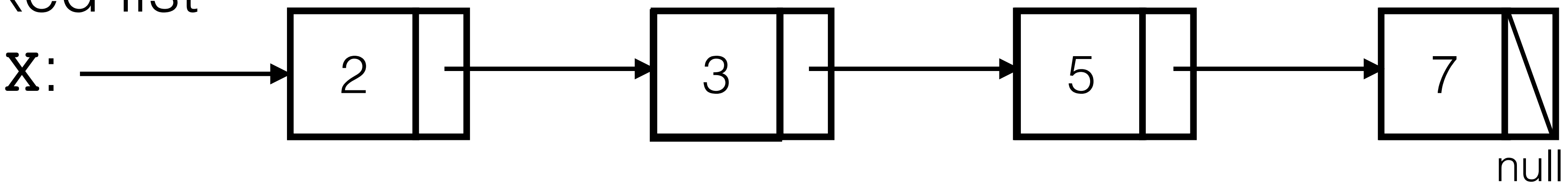


Primitive Data Structures: The Linked List



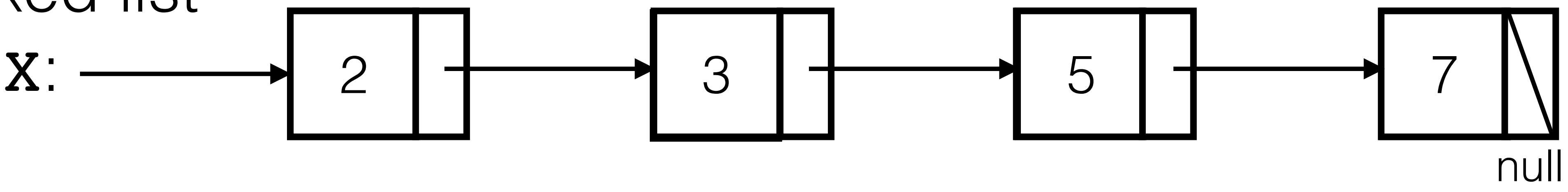
Primitive Data Structures: The Linked List

A linked list



Primitive Data Structures: The Linked List

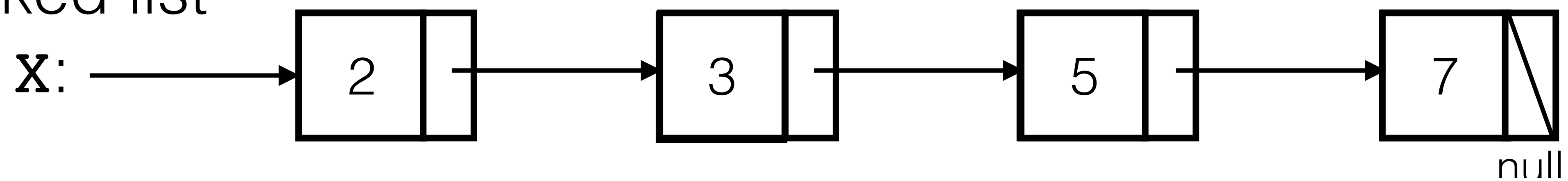
A linked list



Suppose variable **x** holds the address 42160, then the list could look like this in memory:

Primitive Data Structures: The Linked List

A linked list

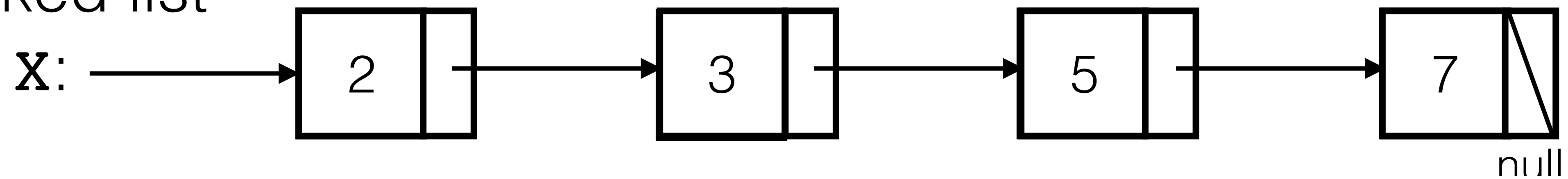


Suppose variable **x** holds the address 42160, then the list could look like this in memory:

42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list

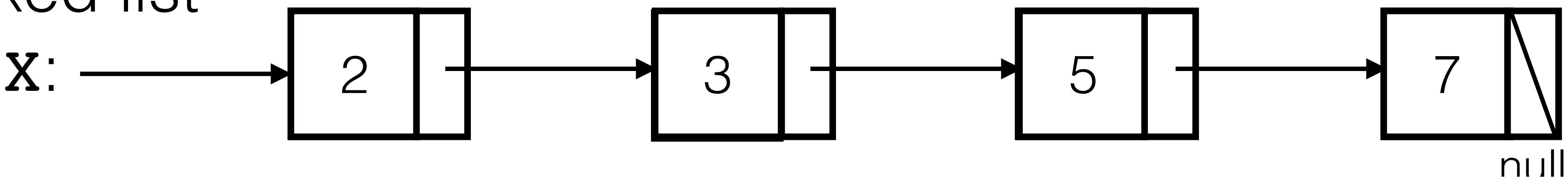


Suppose variable **x** holds the address 42160, then the list could look like this in memory:

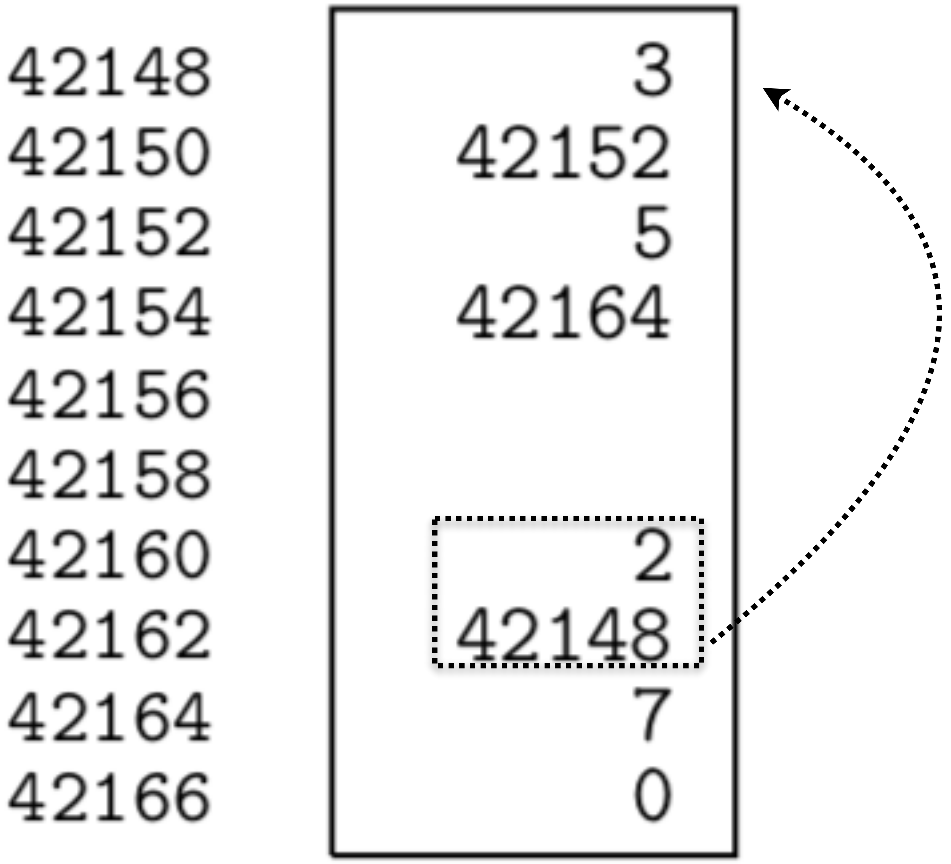
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



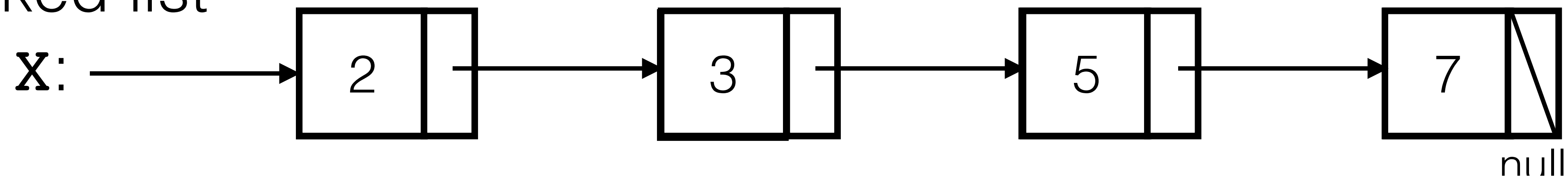
Suppose variable **x** holds the address 42160, then the list could look like this in memory:



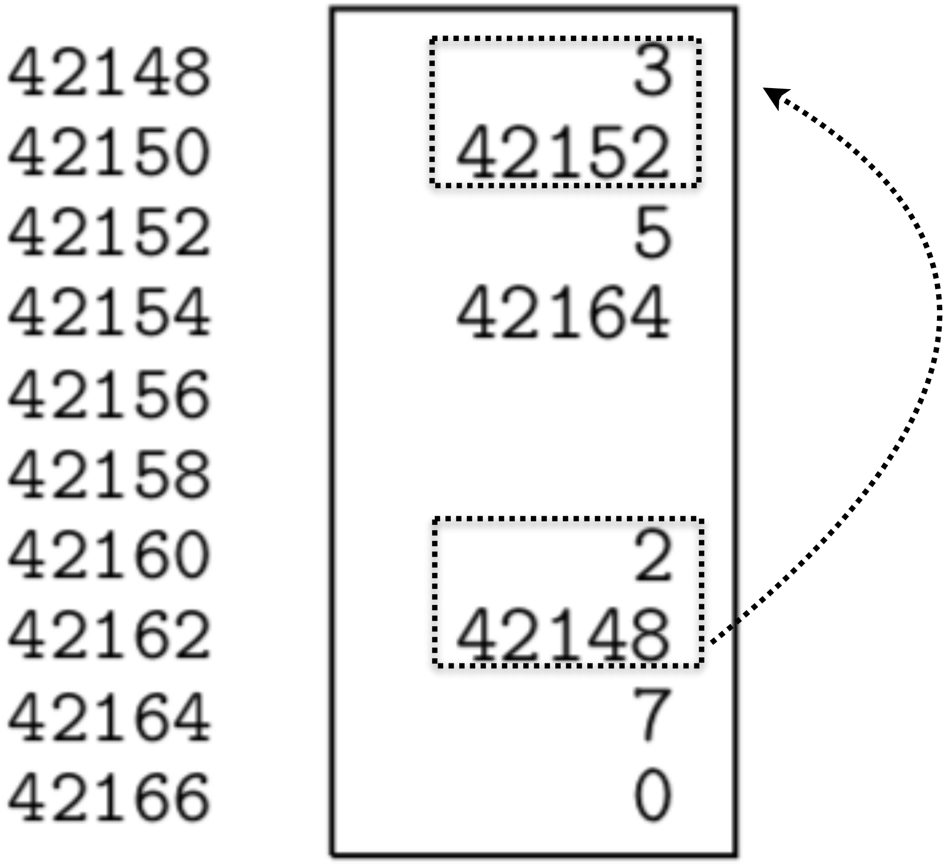
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



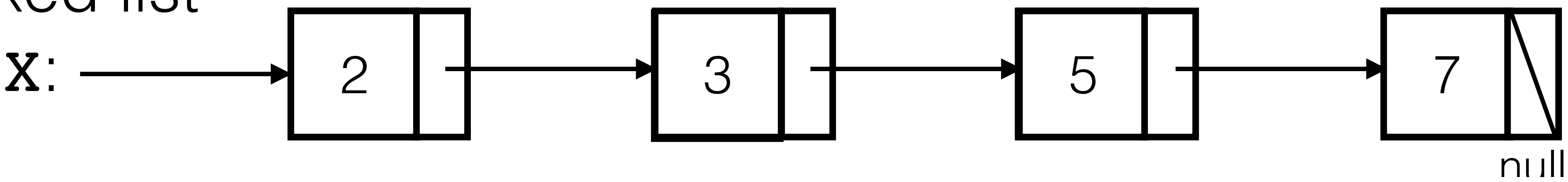
Suppose variable **x** holds the address 42160, then the list could look like this in memory:



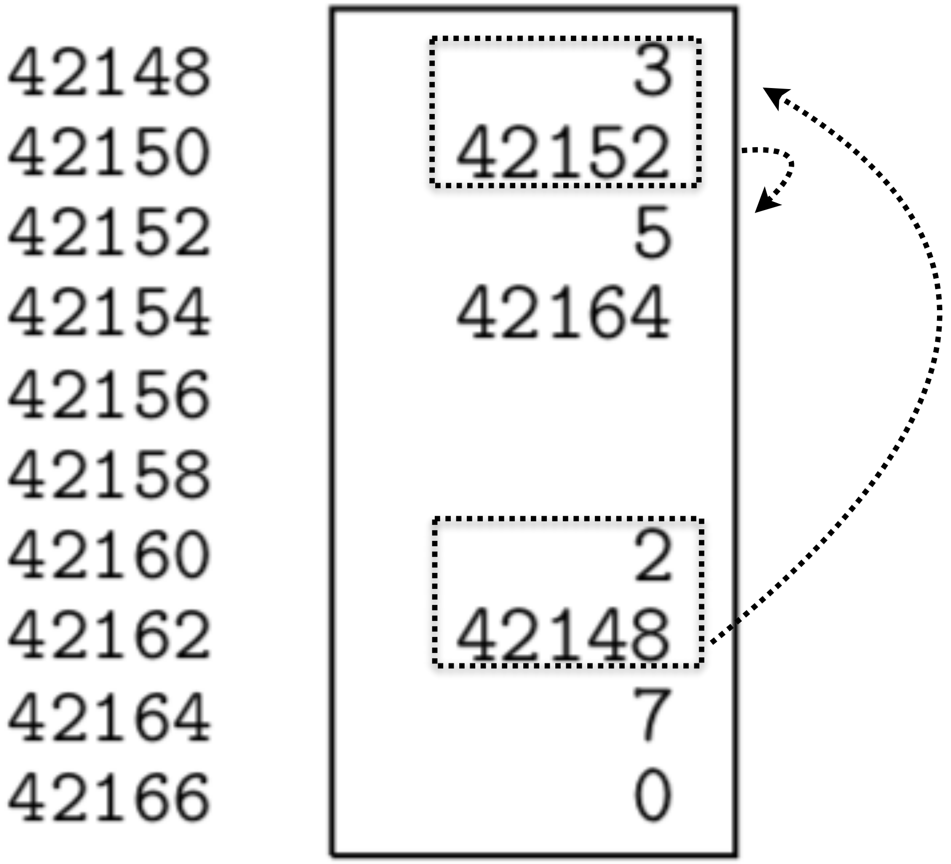
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



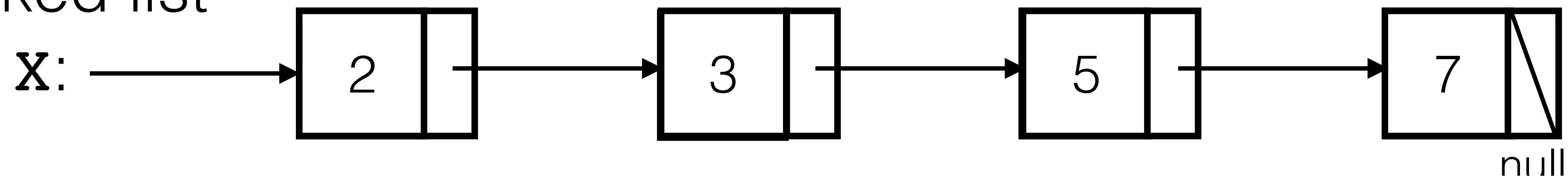
Suppose variable **X** holds the address 42160, then the list could look like this in memory:



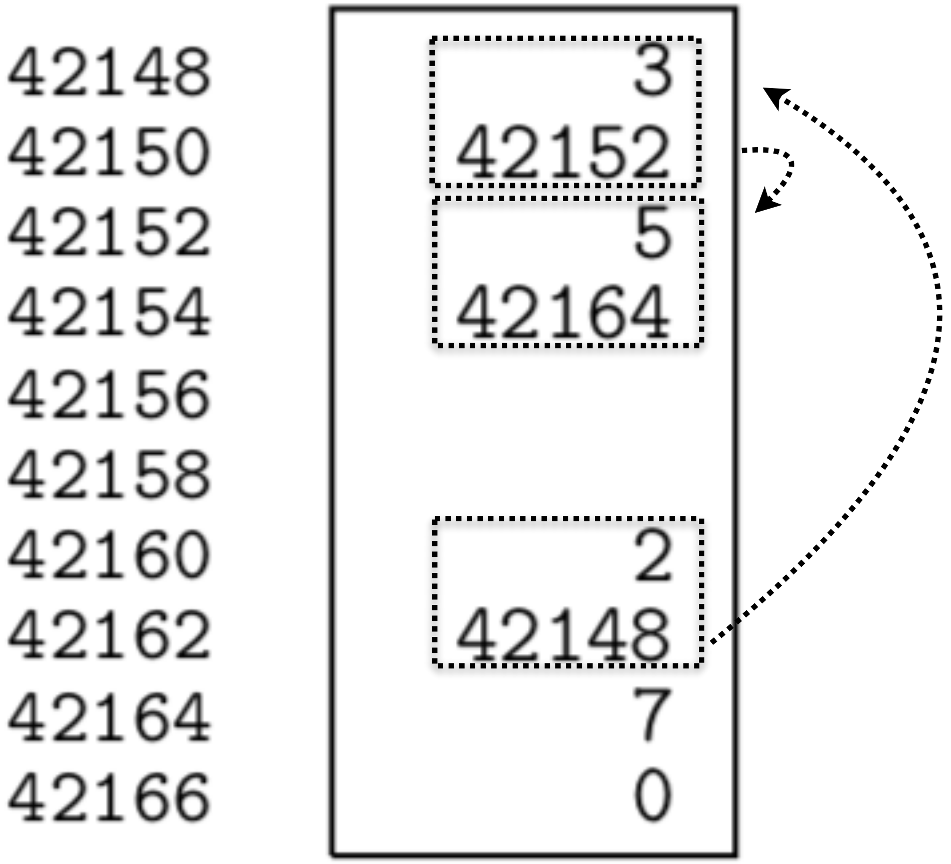
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



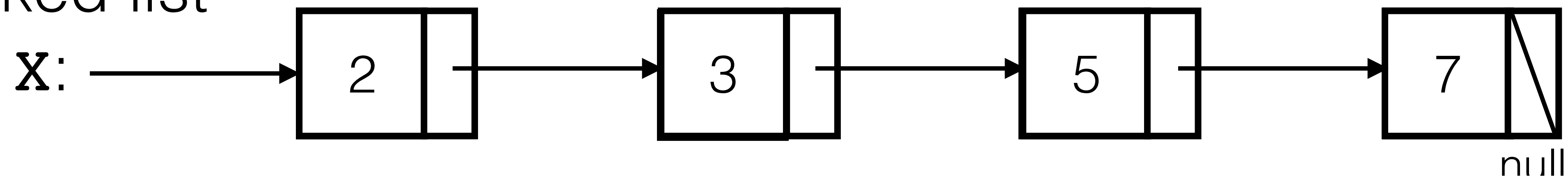
Suppose variable **x** holds the address 42160, then the list could look like this in memory:



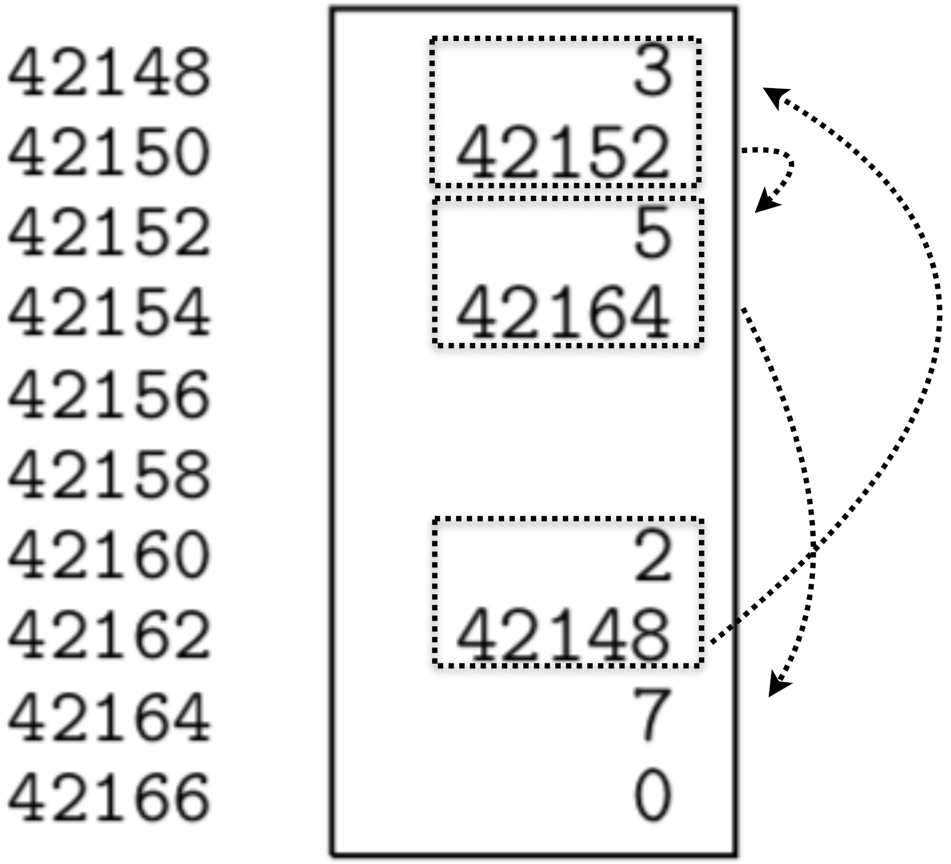
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



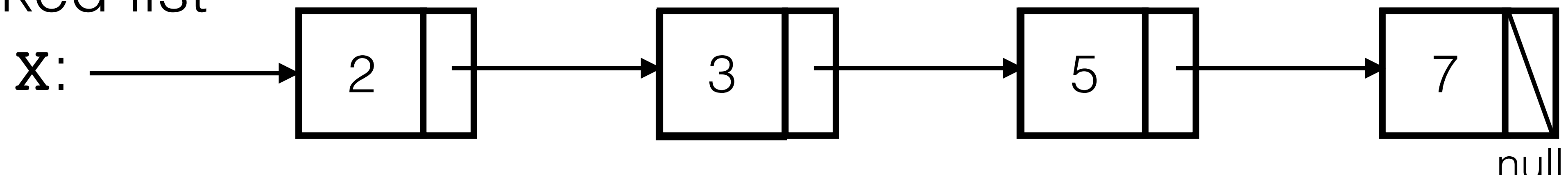
Suppose variable **x** holds the address 42160, then the list could look like this in memory:



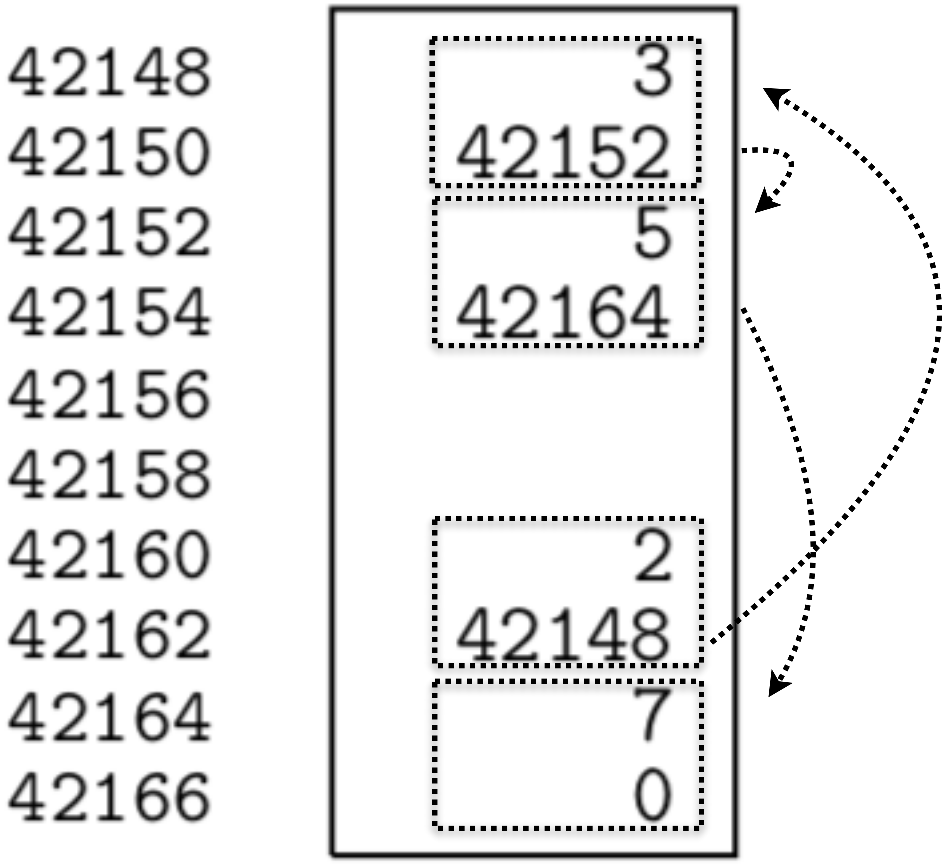
42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

Primitive Data Structures: The Linked List

A linked list



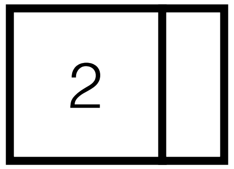
Suppose variable **x** holds the address 42160, then the list could look like this in memory:



42148	3
42150	42152
42152	5
42154	42164
42156	
42158	
42160	2
42162	42148
42164	7
42166	0

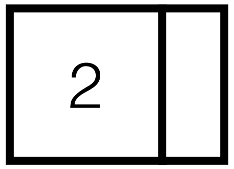
Terminology

Terminology



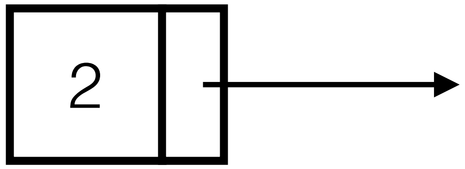
Terminology

node

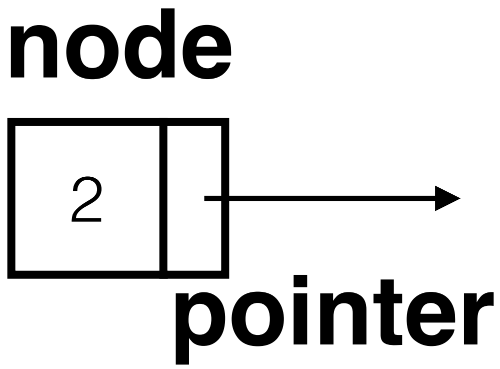


Terminology

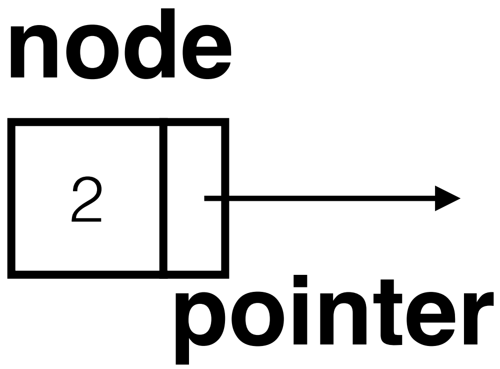
node



Terminology



Terminology



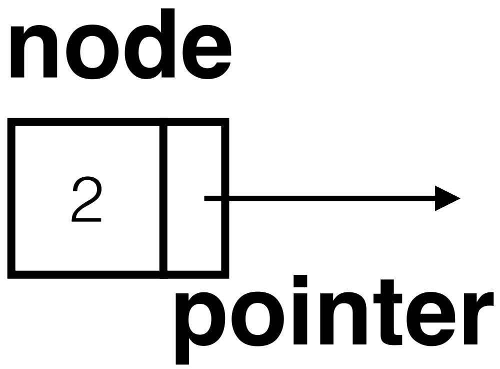
node

2

pointer

(in Java: “reference”)

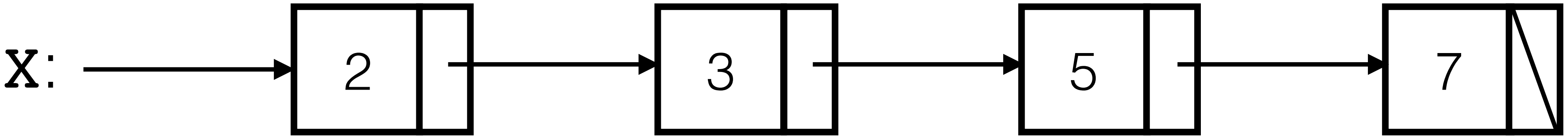
Terminology



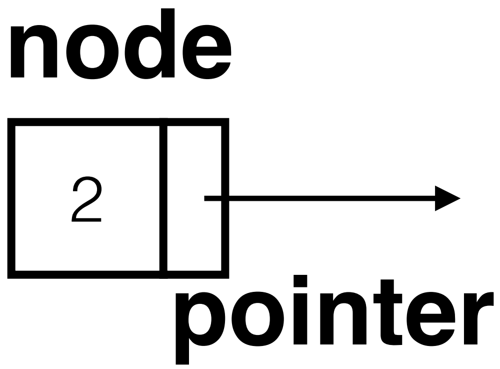
node

pointer

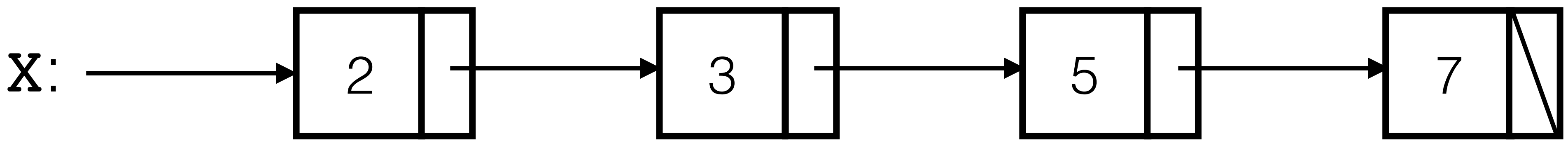
(in Java: “reference”)



Terminology



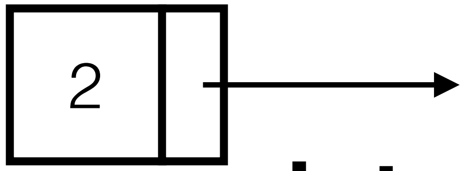
(in Java: “reference”)



`x` is (a pointer to) the **head node** of the list

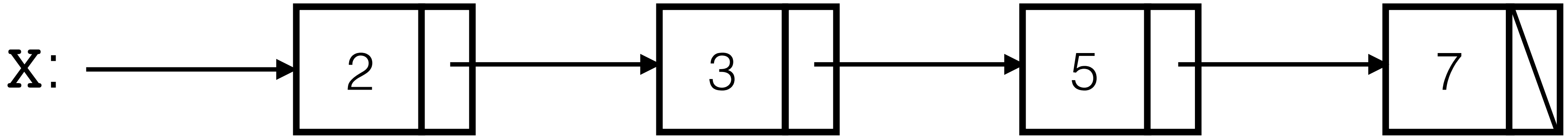
Terminology

node

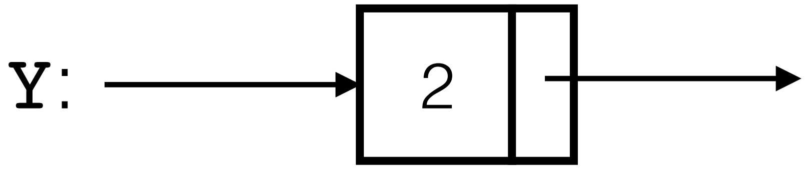


pointer

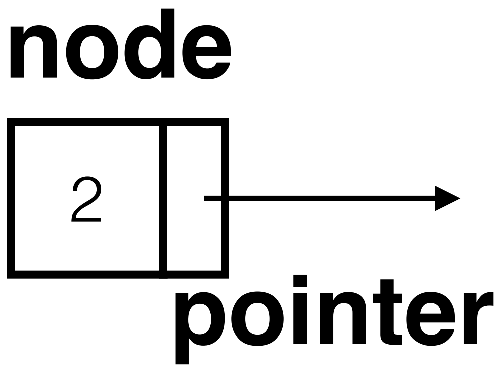
(in Java: “reference”)



`X` is (a pointer to) the **head node** of the list



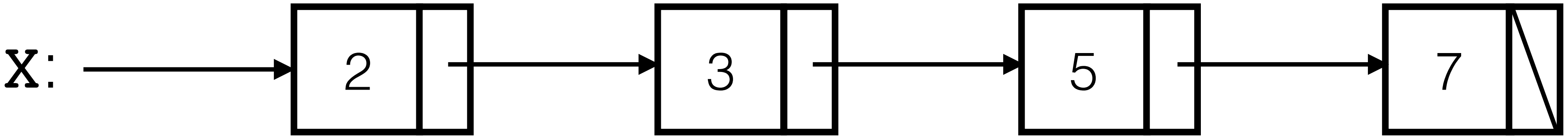
Terminology



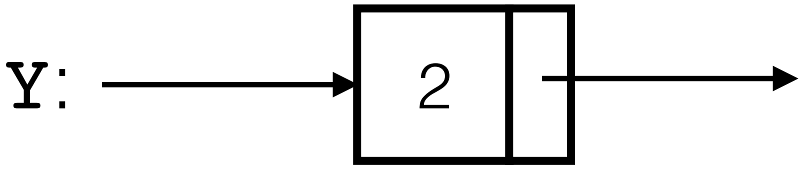
node

pointer

(in Java: “reference”)



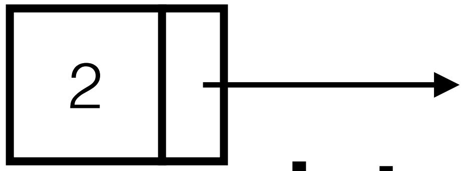
`X` is (a pointer to) the **head node** of the list



“`Y.val`” *refers to*

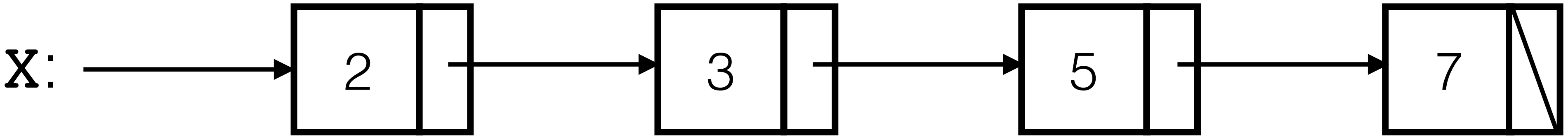
Terminology

node

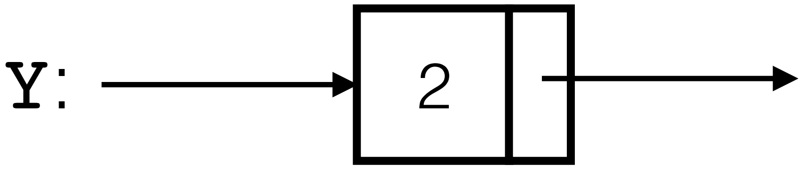


pointer

(in Java: “reference”)



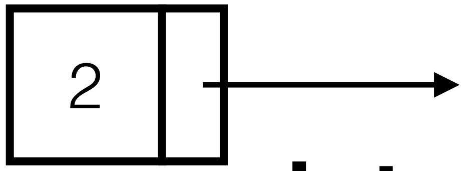
X is (a pointer to) the **head node** of the list



“Y.val” *refers to*

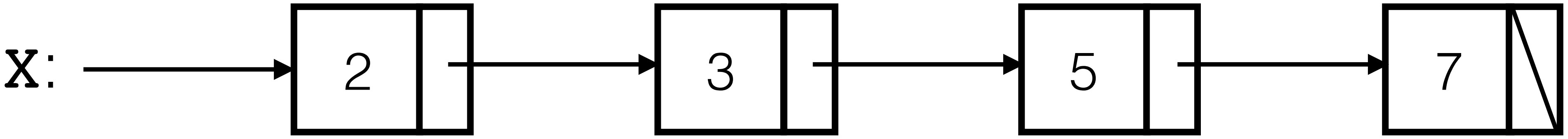
Terminology

node

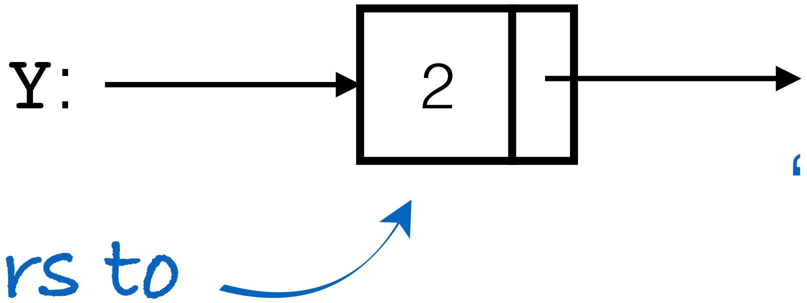


pointer

(in Java: “reference”)



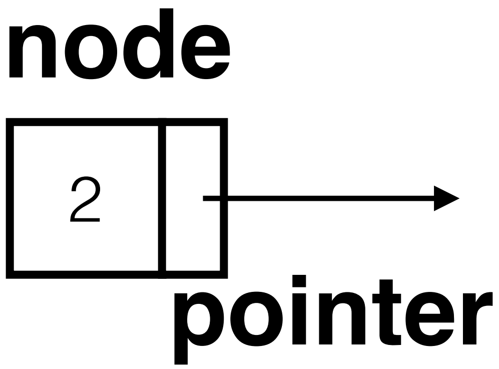
`x` is (a pointer to) the **head node** of the list



“`Y.val`” *refers to*

“`Y.next`” *refers to*

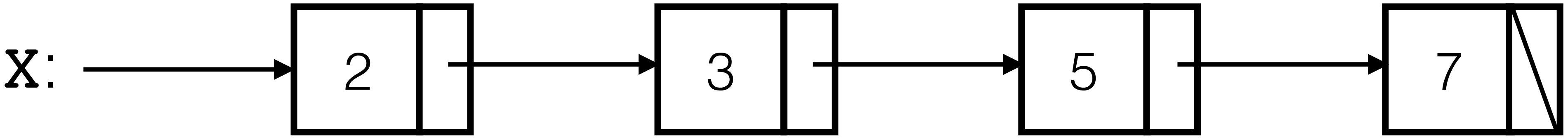
Terminology



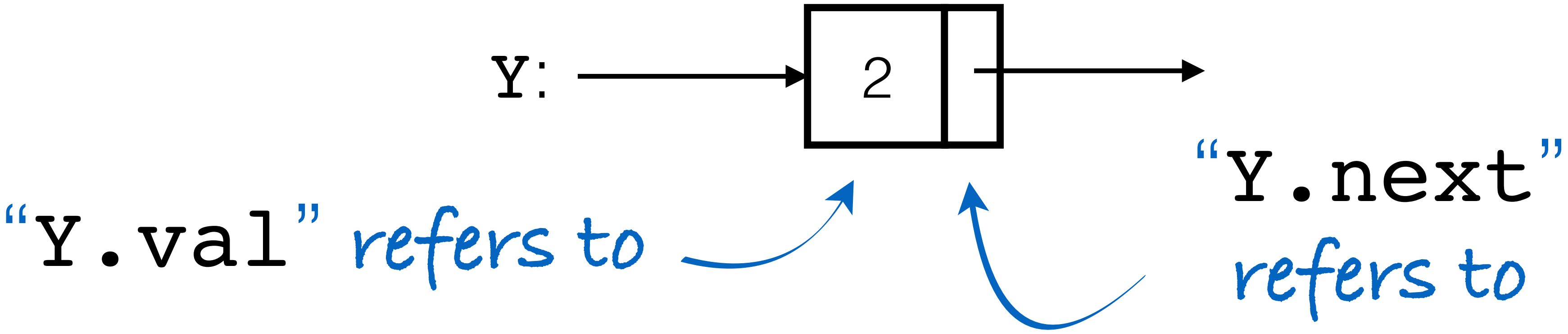
node

pointer

(in Java: “reference”)



`X` is (a pointer to) the **head node** of the list



“`Y.val`” refers to

“`Y.next`” refers to

Linked List

- Often we use a dummy head node that points to the first object, or to a special `null` object that represents an empty list. This makes it easier to write functions that insert or delete elements.

- Inserting and deleting elements is very fast: just move a few links around.

- Finding the *i*th element can be time-consuming.

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

```
function find(A,x,n)
    $j \leftarrow 0$
    while $j < n$
        if $A[j] = x$
            return j
        $j \leftarrow j+1$
    return -1
```

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item `x`.

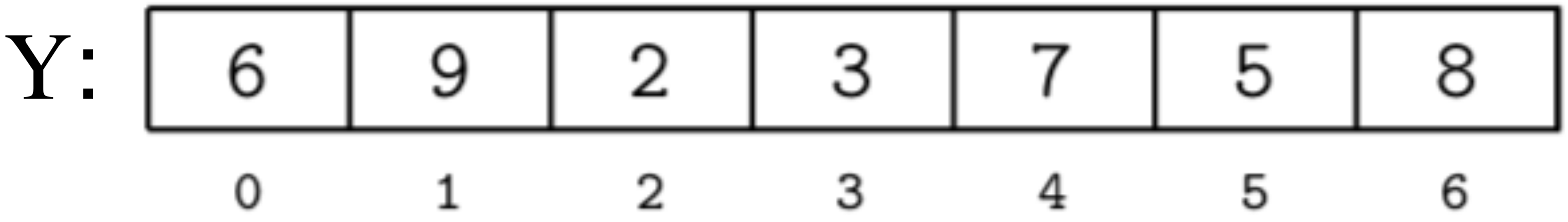
```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Iterative Processing: Array

- Walk through the array (of length n)
- For example, to locate item **x**.

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Y:							
	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Iterative Processing: Array

- Walk through the array (of length n)
- For example, to locate item **x**.

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Let's trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Let's trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y   x: 7

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Let’s trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y x: 7 n: 6

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Let’s trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y x: 7 n: 6 j: 0

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

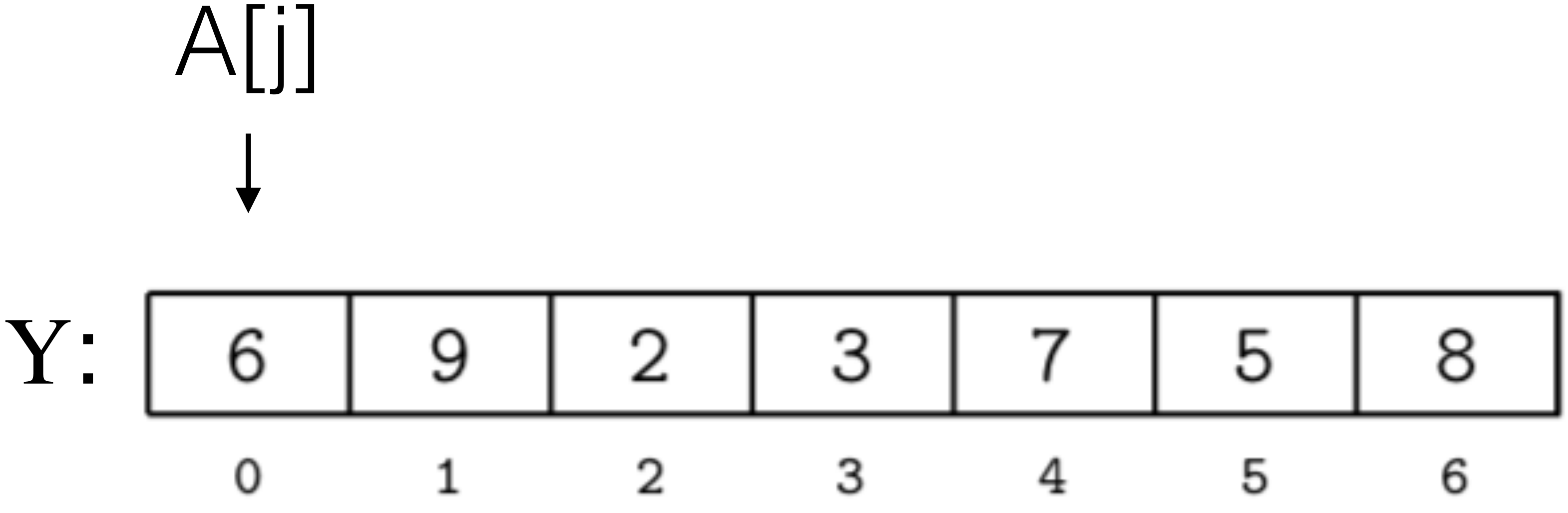
Let’s trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)
- For example, to locate item **x**.

A: Y   x: 7   n: 6   j: 0

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



A[j]

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

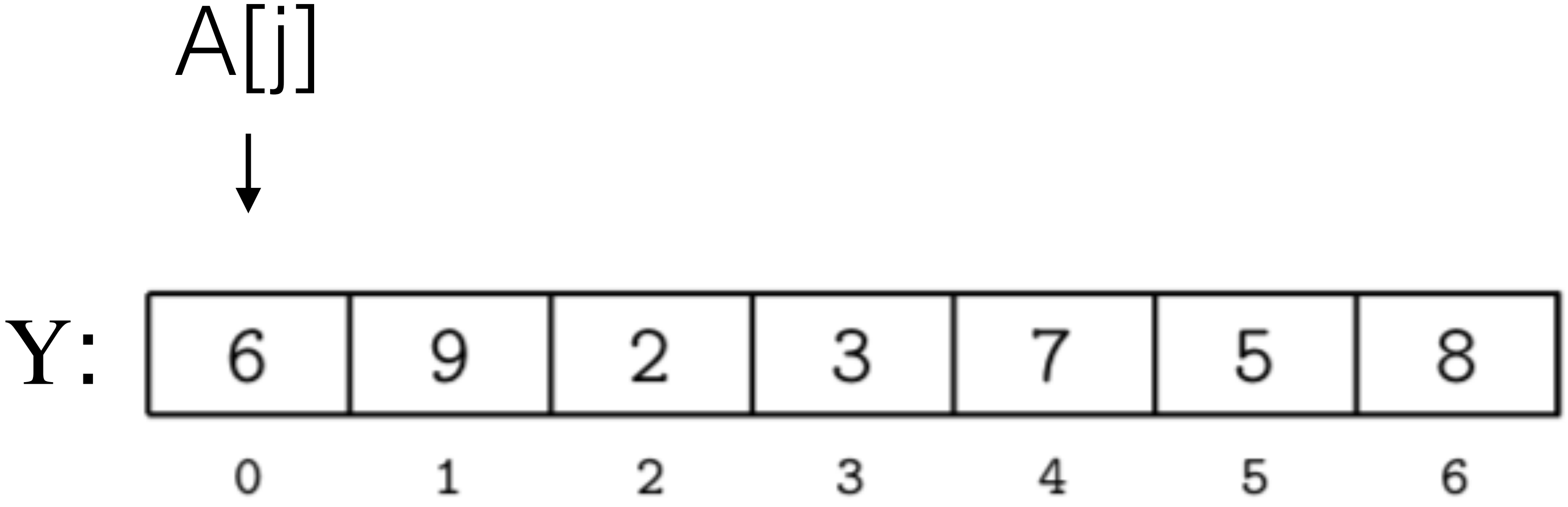
Let’s trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)
- For example, to locate item **x**.

A: Y x: 7 n: 6 j: 1

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Let's trace the execution of find(Y,7,6)

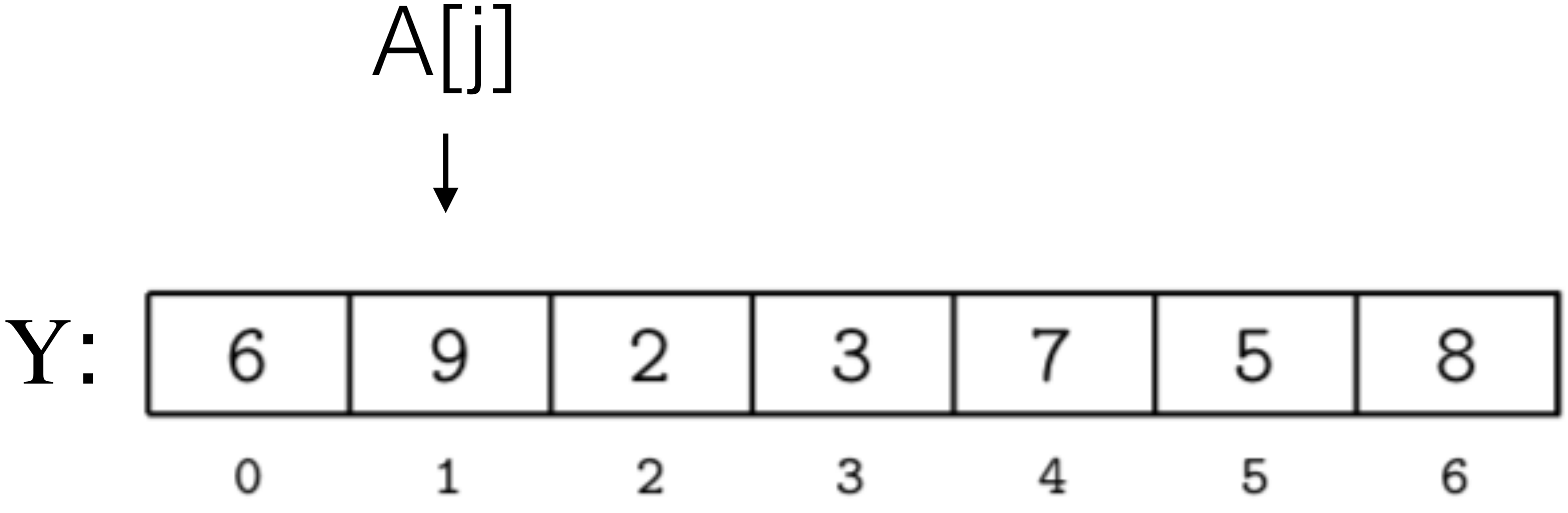
Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y   x: 7   n: 6   j: 1

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



A[j]

	0	1	2	3	4	5	6
Y:	6	9	2	3	7	5	8

Let’s trace the execution of find(Y,7,6)

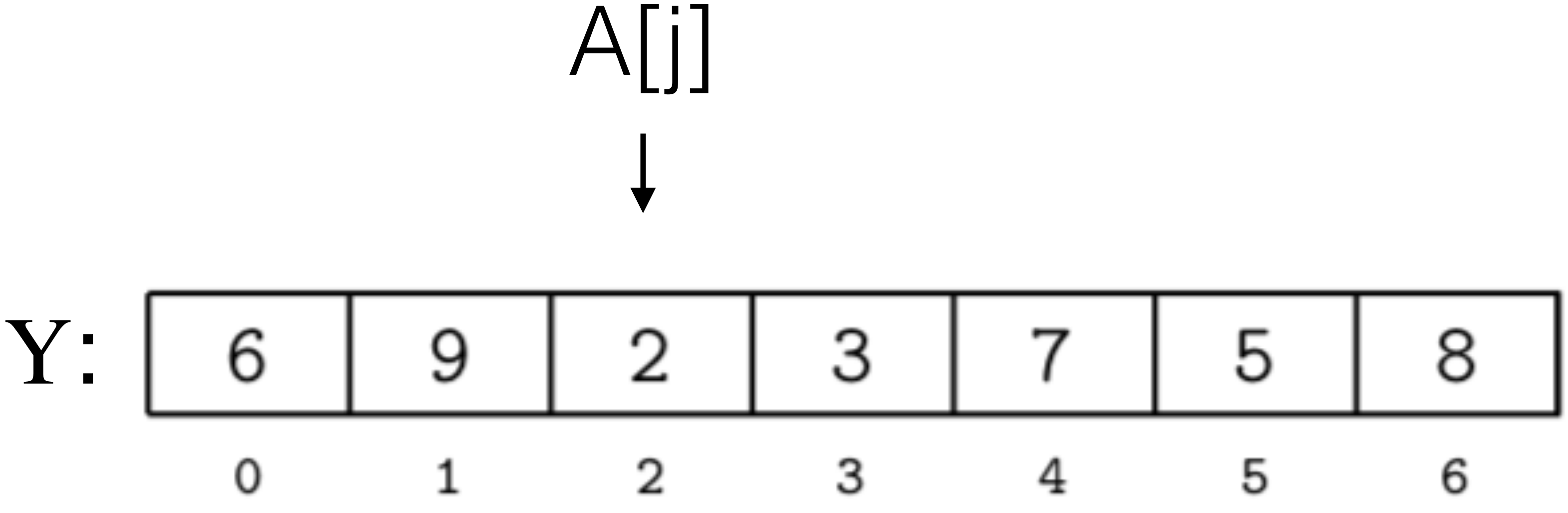
Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y   x: 7   n: 6   j: 2

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



A[j]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Let's trace the execution of find(Y,7,6)

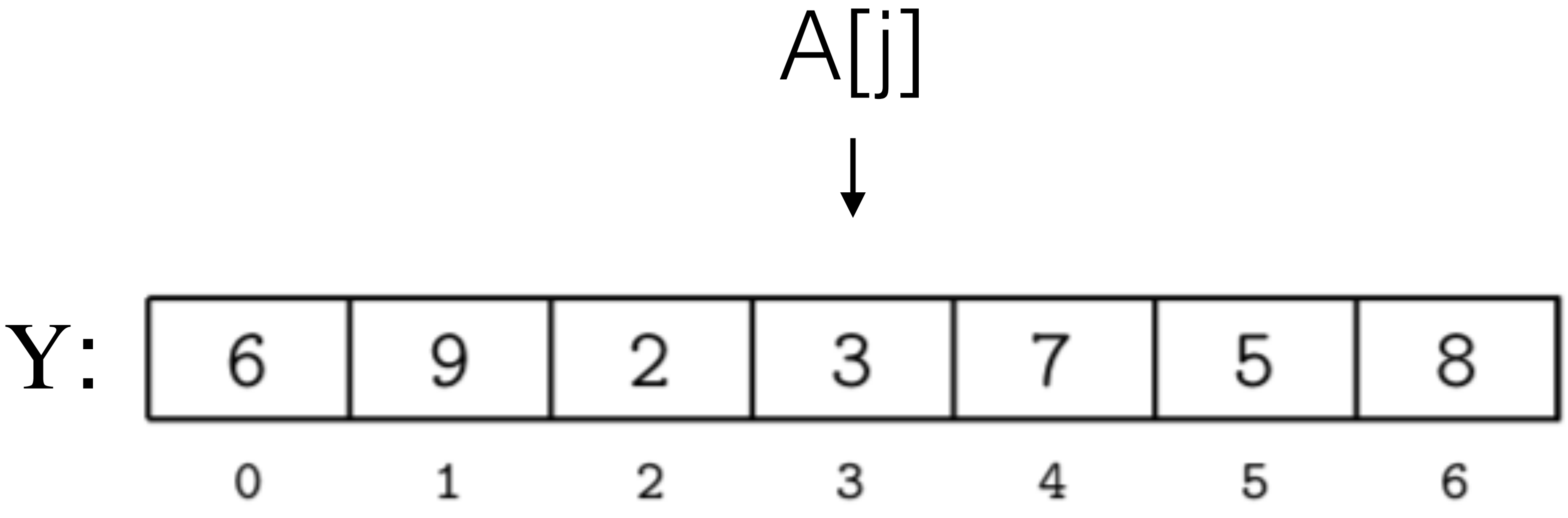
Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y   x: 7   n: 6   j: 3

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



A[j]

Y:							
	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

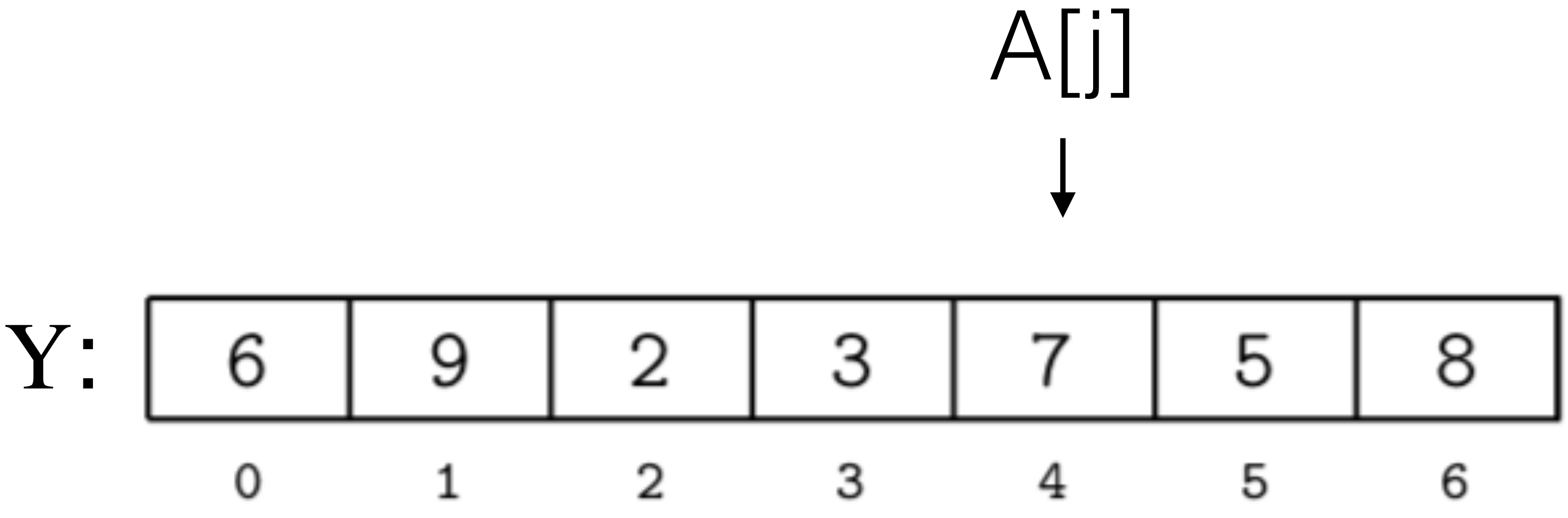
Let’s trace the execution of find(Y,7,6)

Iterative Processing: Array

- Walk through the array (of length n)
- For example, to locate item **x**.

A: Y x: 7 n: 6 j: 4

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



A[j]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Let’s trace the execution of find(Y,7,6)

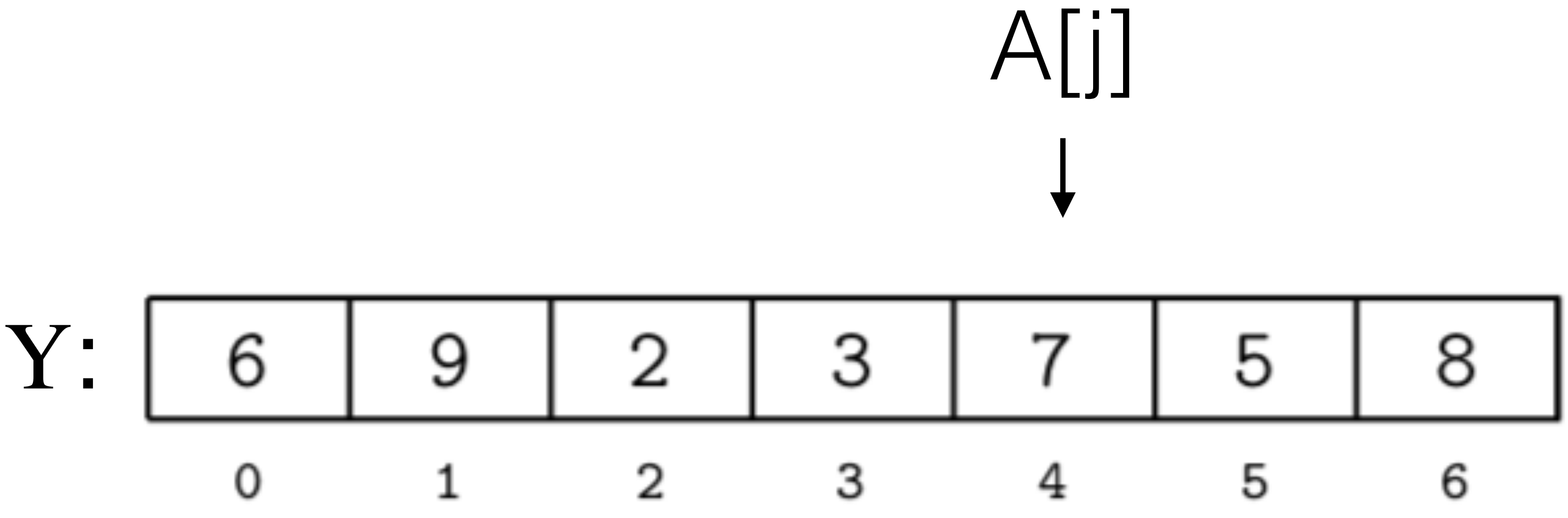
Iterative Processing: Array

- Walk through the array (of length n)

- For example, to locate item **x**.

A: Y x: 7 n: 6 j: 4

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Let’s trace the execution of find(Y,7,6)

(returns 4)

Iterative Processing: List

- Walk through a linked list.

- For example, to locate item `x`.

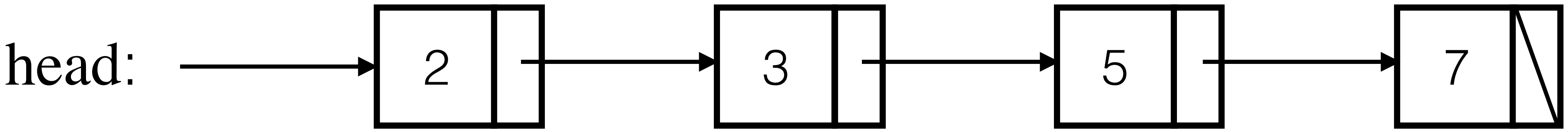
```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

Iterative Processing: List

- Walk through a linked list.

- For example, to locate item `x`.

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```



Iterative Processing: List

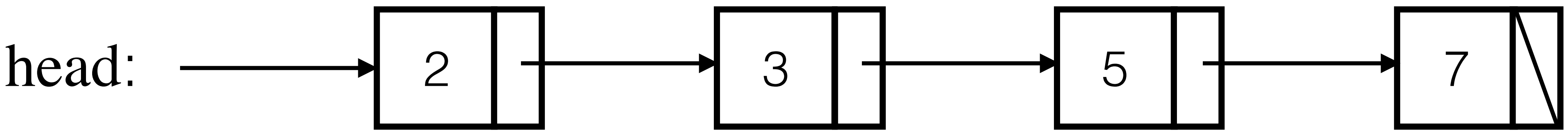
- Walk through a linked list.

- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Iterative Processing: List

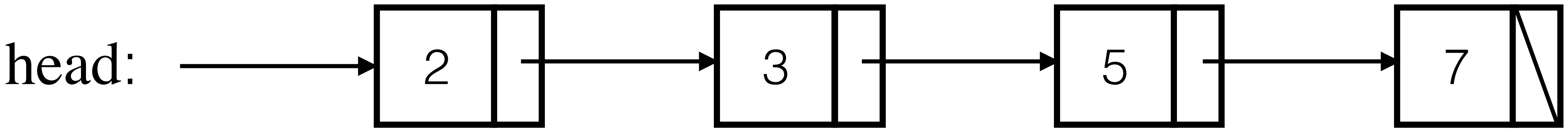
- Walk through a linked list.

- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    $p \leftarrow head$
    while $p \neq null$
        if $p.val = x$
            return p
        $p \leftarrow p.next$
    return null
```

```
function find(A,x,n)
    $j \leftarrow 0$
    while $j < n$
        if $A[j] = x$
            return j
        $j \leftarrow j+1$
    return -1
```



Iterative Processing: List

- Walk through a linked list.

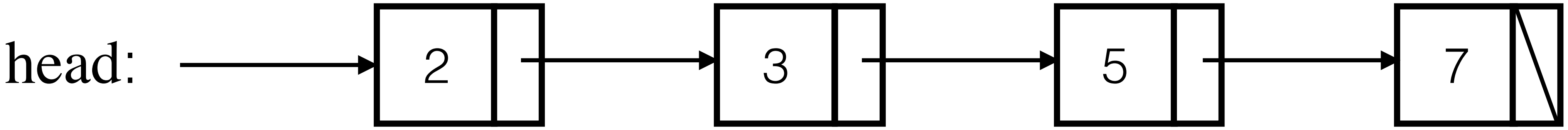
- For example, to locate item `x`.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

p:

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



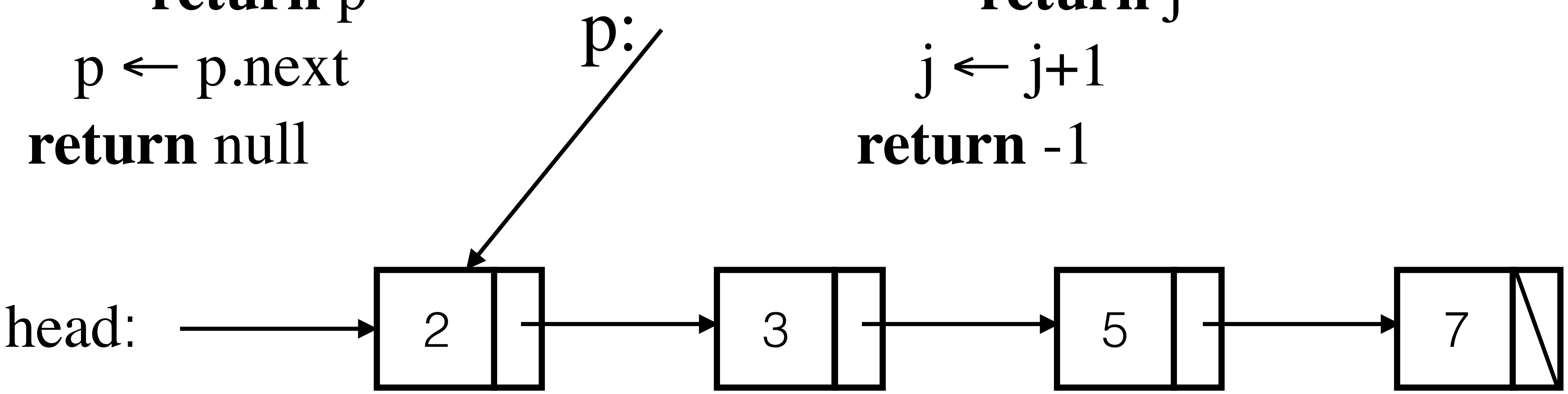
Iterative Processing: List

- Walk through a linked list.
- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Iterative Processing: List

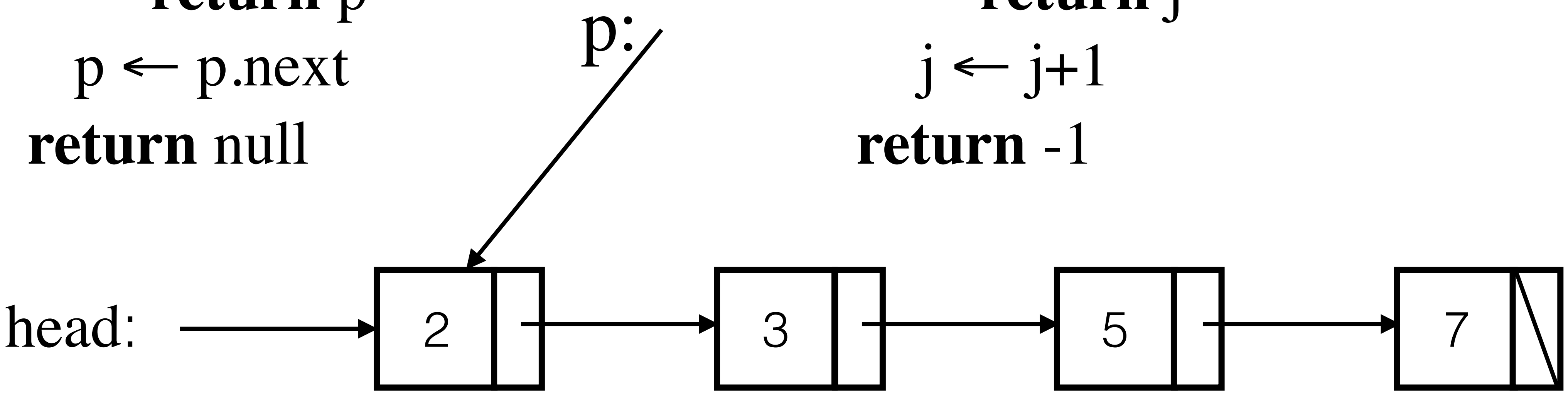
- Walk through a linked list.

- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Iterative Processing: List

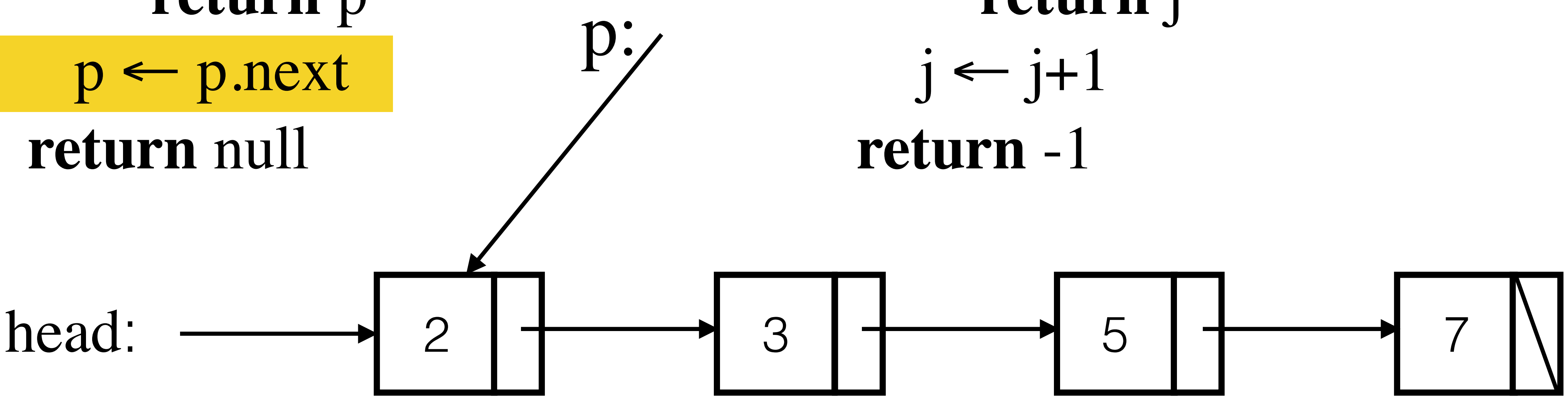
- Walk through a linked list.

- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



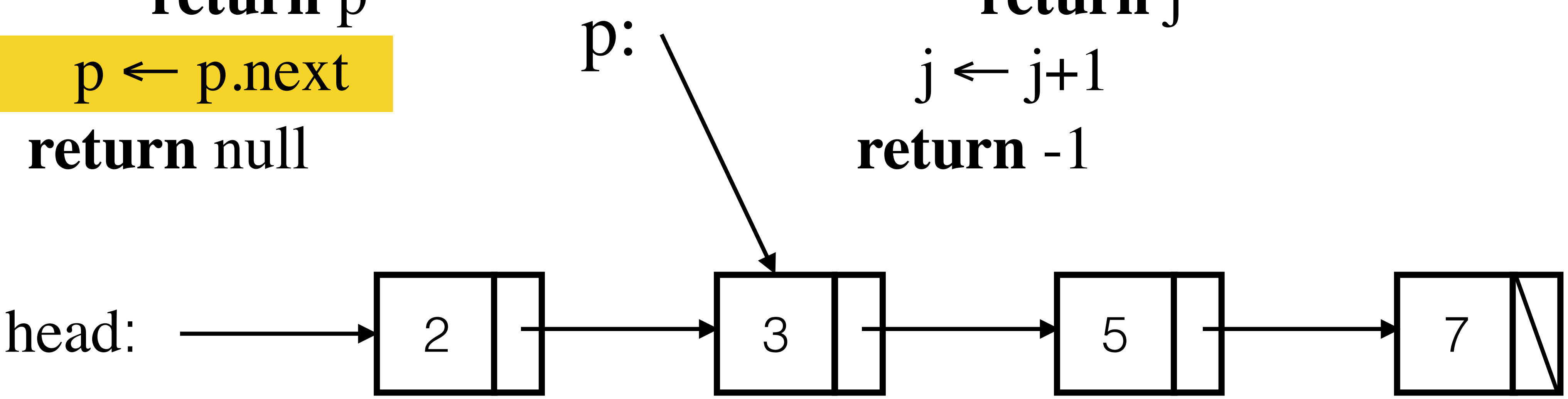
Iterative Processing: List

- Walk through a linked list.
- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



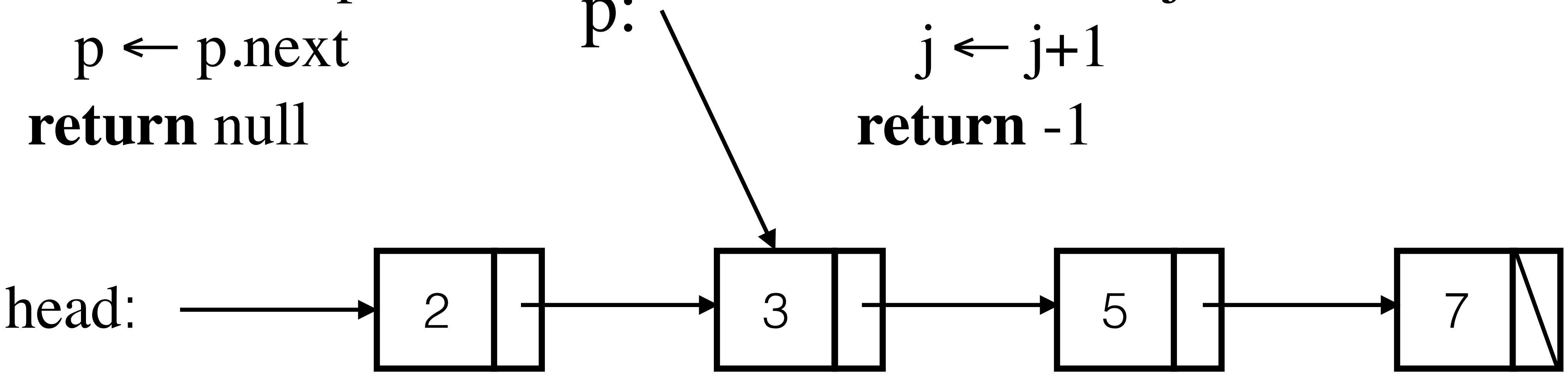
Iterative Processing: List

- Walk through a linked list.
- For example, to locate item **x**.

(note similarity to array version)

```
function find(head,x)
    p ← head
    while p ≠ null
        if p.val = x
            return p
        p ← p.next
    return null
```

```
function find(A,x,n)
    j ← 0
    while j < n
        if A[j] = x
            return j
        j ← j+1
    return -1
```



Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item `x`.

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(A,x,0,n-1)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item `x`.

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance
- For example, to locate item `x`.

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:

6	9	2	3	7	5	8
0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y x: 7

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y x: 7 lo: 0

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y   x: 7   lo: 0   hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

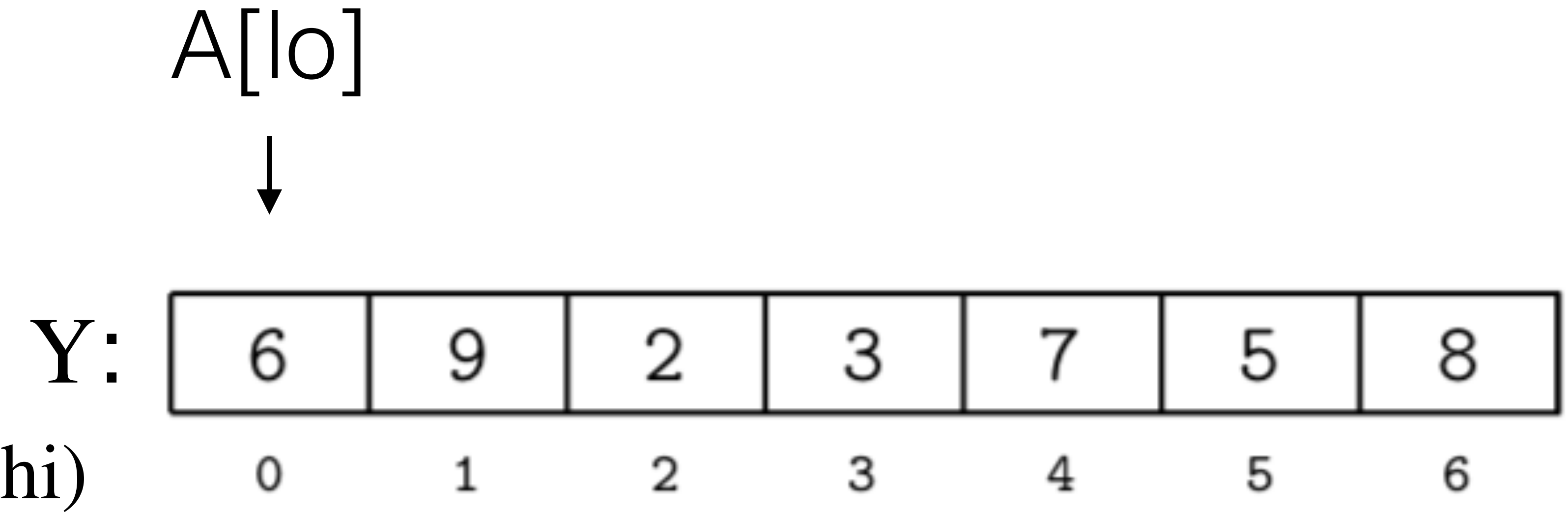
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item `x`.

A: Y x: 7 lo: 0 hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

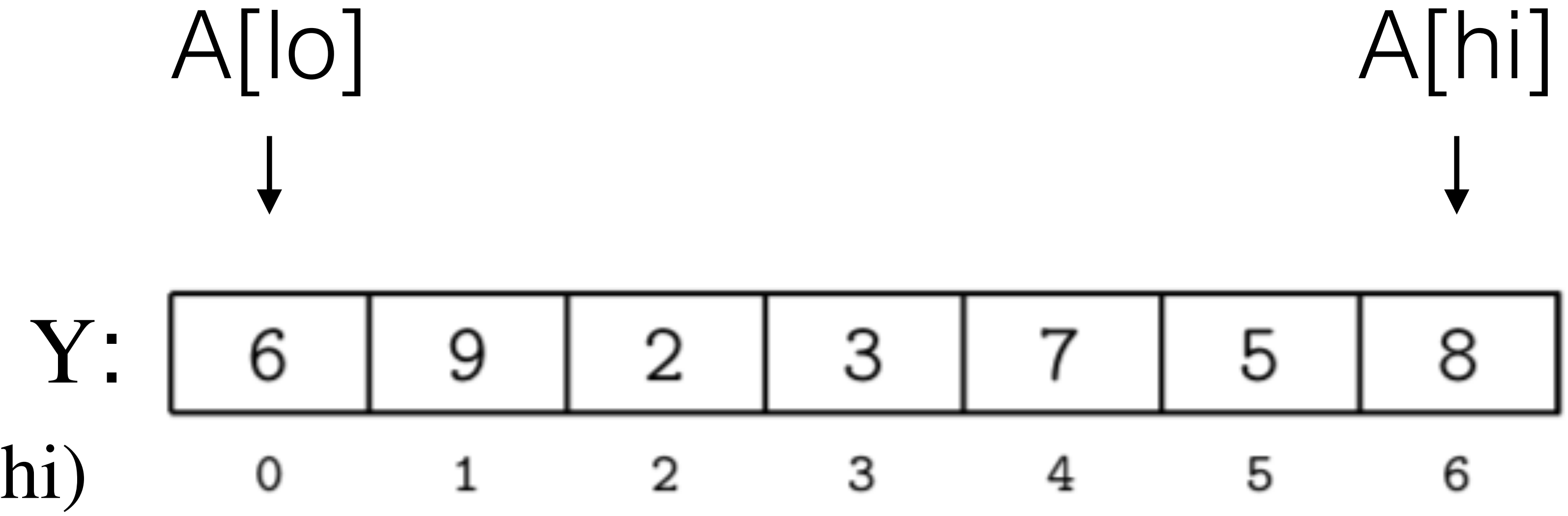
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y   x: 7   lo: 0   hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo]   A[hi]

	0	1	2	3	4	5	6
Y:	6	9	2	3	7	5	8

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

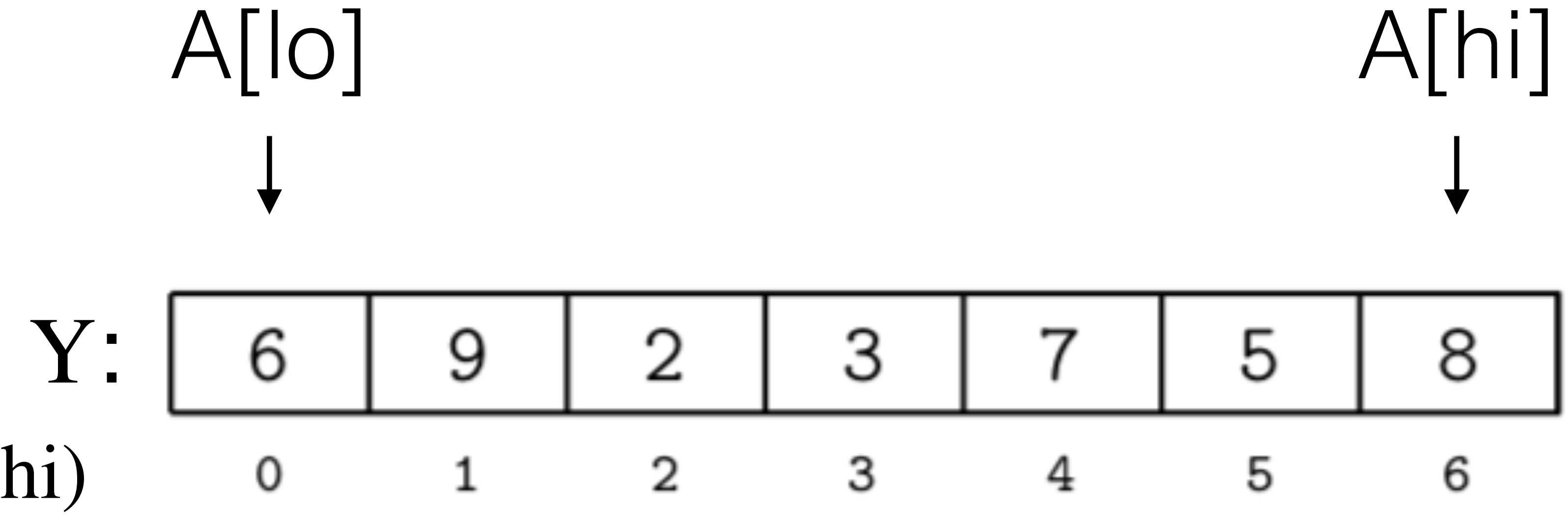
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y   x: 7   lo: 1   hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo]

A[hi]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let’s trace the execution of find(Y,7,0,6)

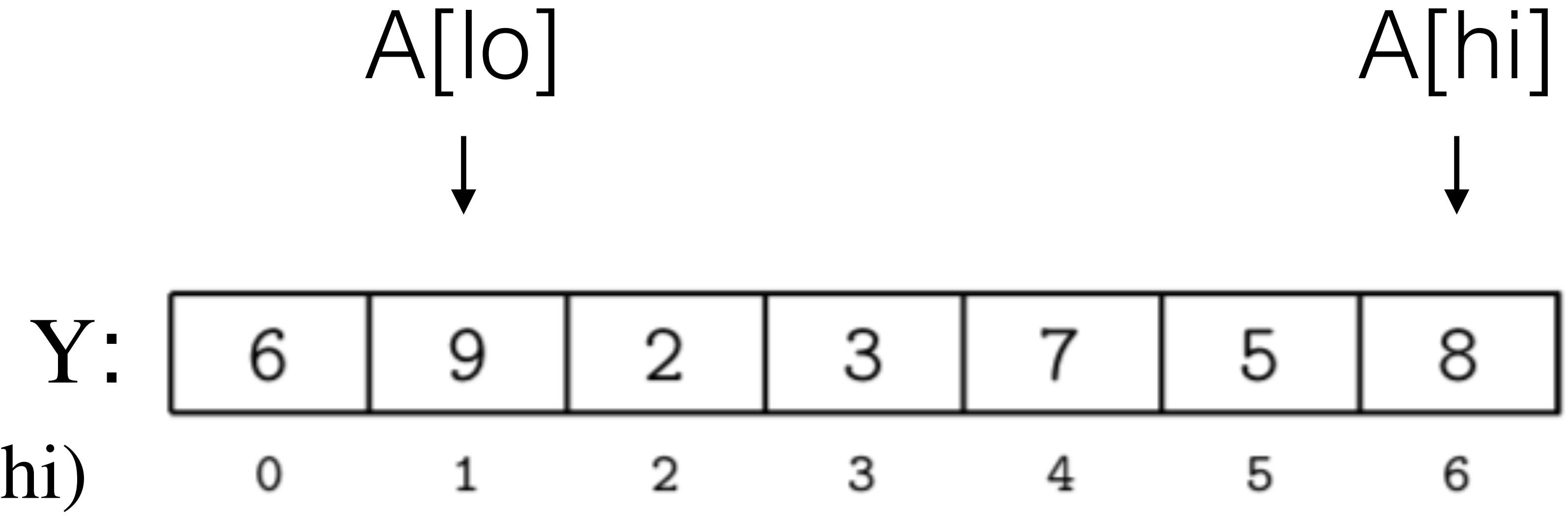
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item `x`.

A: Y   x: 7   lo: 1   hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo]

A[hi]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let’s trace the execution of find(Y,7,0,6)

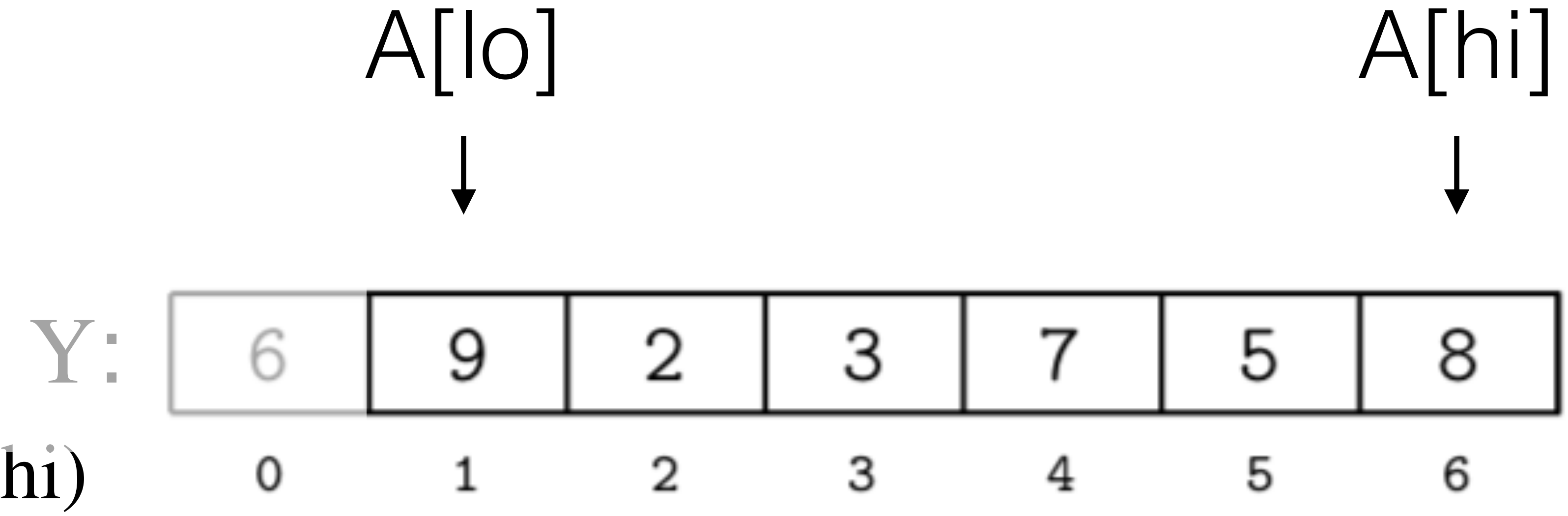
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y x: 7 lo: 1 hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

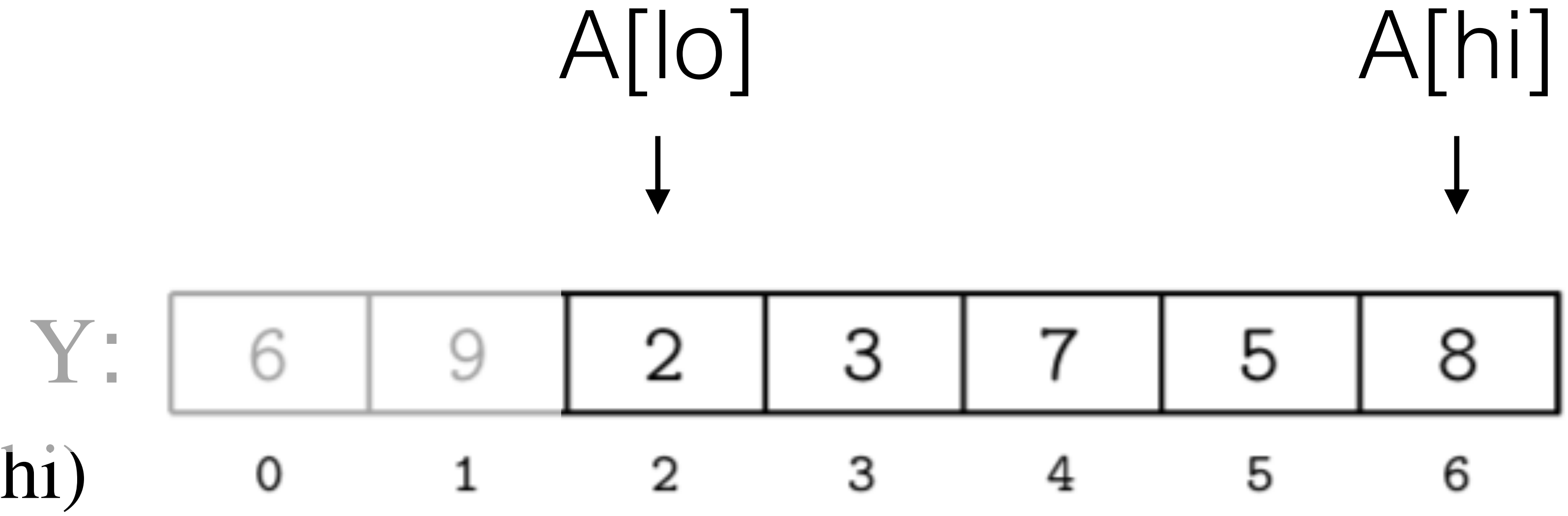
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y x: 7 lo: 2 hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo] ↓ A[hi] ↓

Y:							
	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

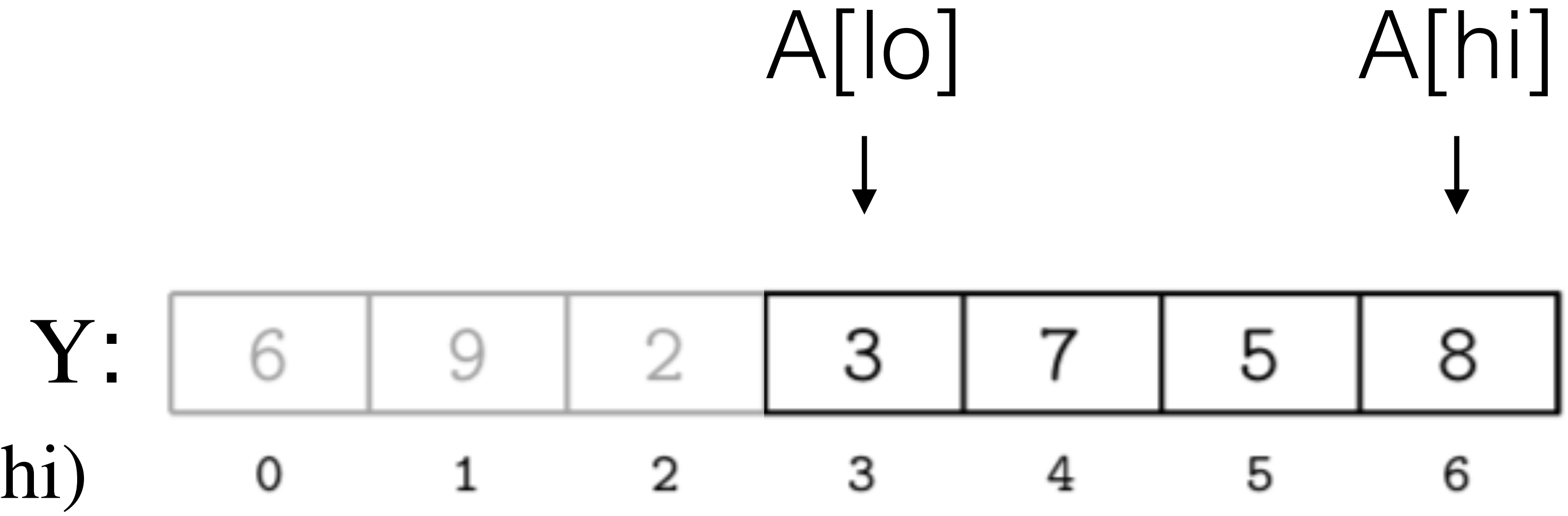
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y   x: 7   lo: 3   hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo]   A[hi]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let’s trace the execution of find(Y,7,0,6)

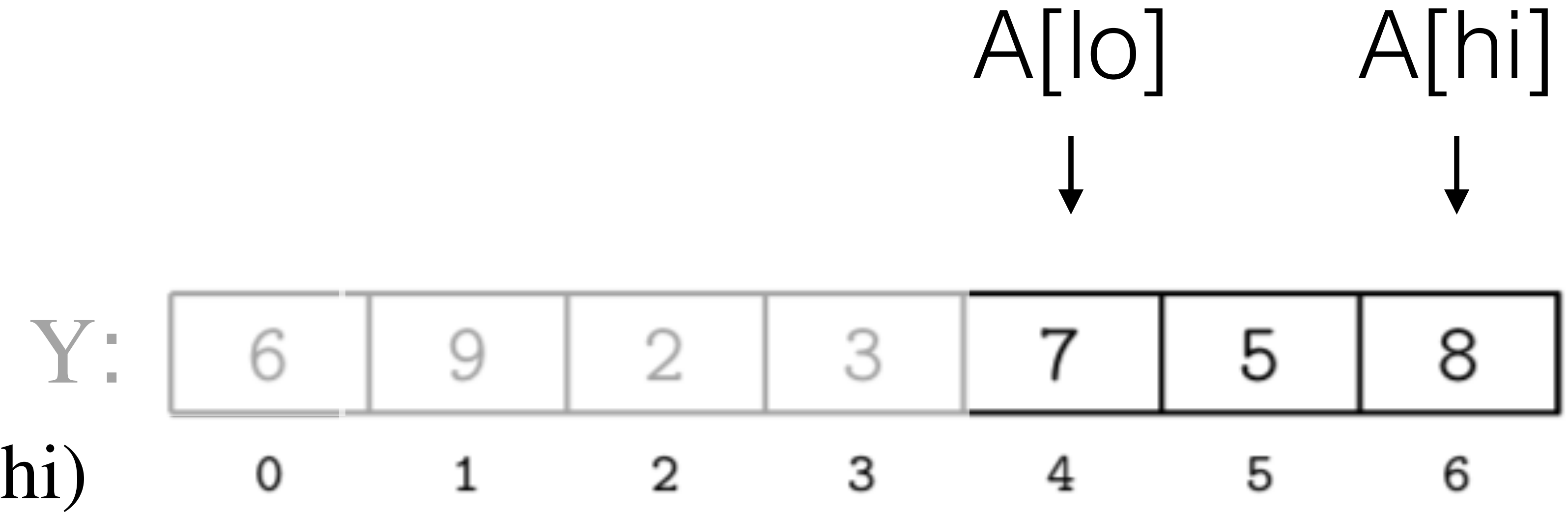
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item **x**.

A: Y x: 7 lo: 4 hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo] A[hi]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

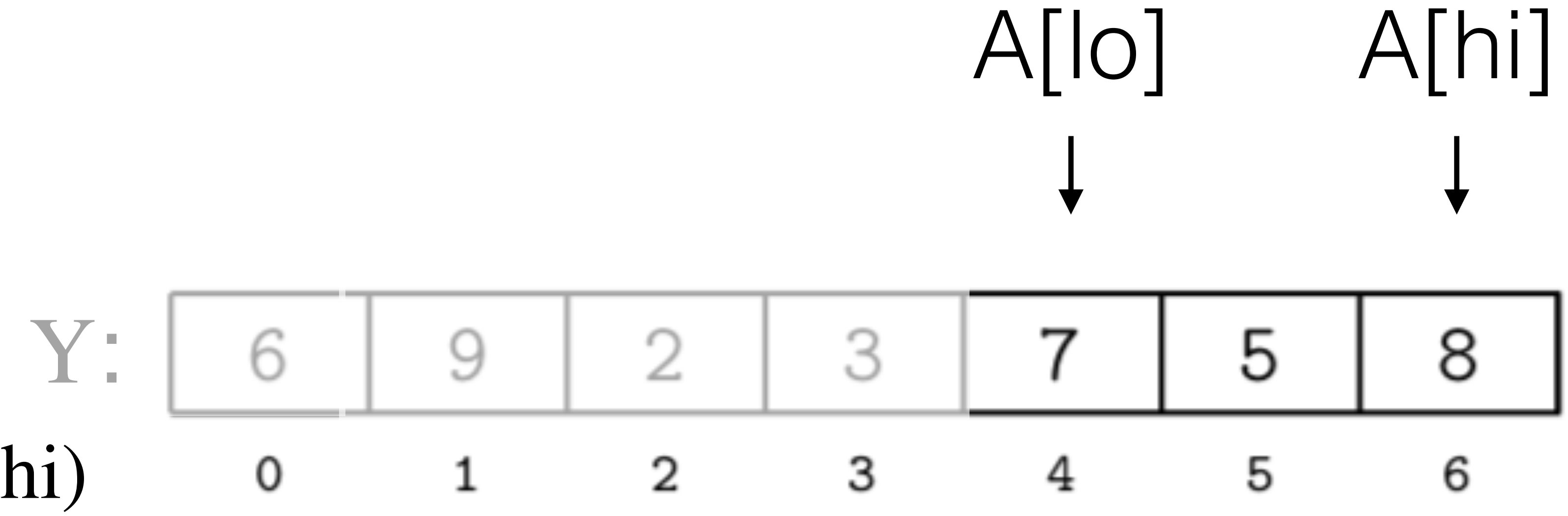
Recursive Processing: Array

- Solve the problem for a sub-instance and use the solution to solve the full instance

- For example, to locate item `x`.

A: Y x: 7 lo: 4 hi: 6

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```



A[lo] A[hi]

Y:	6	9	2	3	7	5	8
	0	1	2	3	4	5	6

Initial call: find(A,x,0,n-1)

Let's trace the execution of find(Y,7,0,6)

(returns 4)

Recursive Processing: List

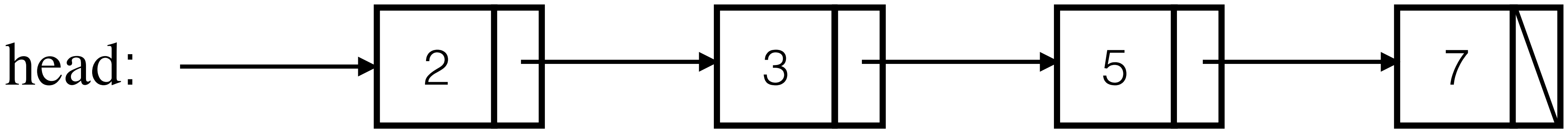
- Solve the problem for a sub-instance and use the solution to solve the full instance

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

Recursive Processing: List

- Solve the problem for a sub-instance and use the solution to solve the full instance

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

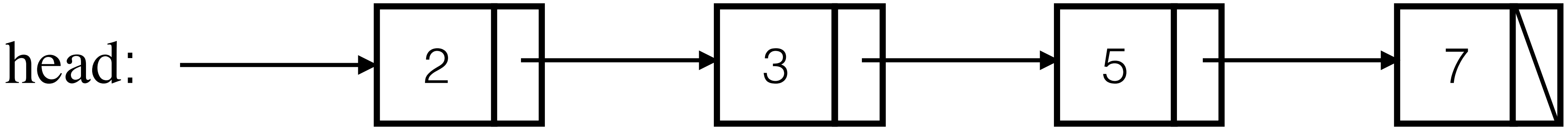


Recursive Processing: List

- Solve the problem for a sub-instance and use the solution to solve the full instance

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

Initial call: find(head,x)



Recursive Processing: List

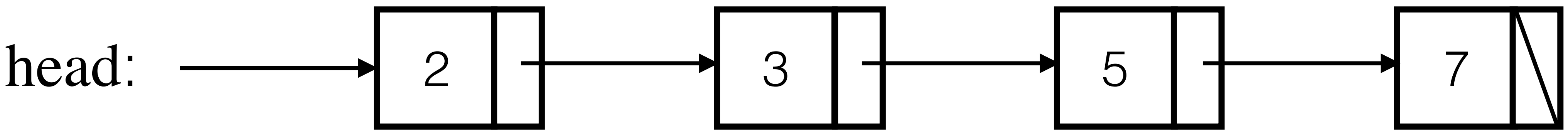
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

- Solve the problem for a sub-instance and use the solution to solve the full instance

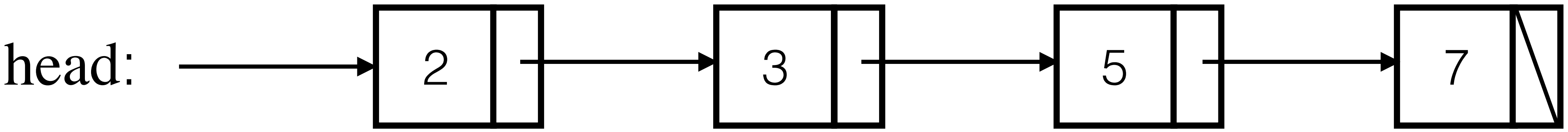
(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

p:

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

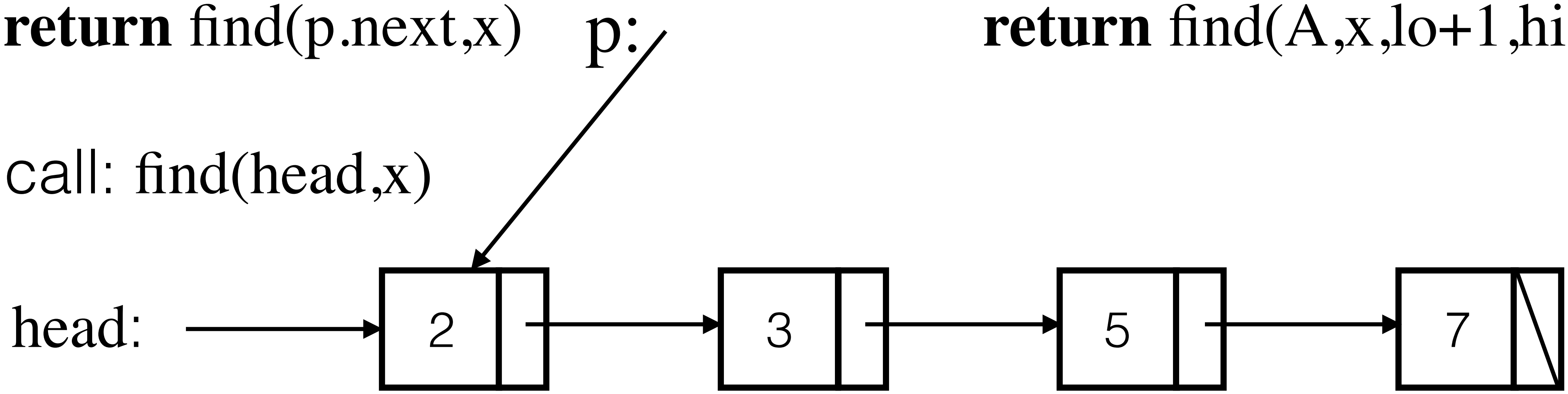
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

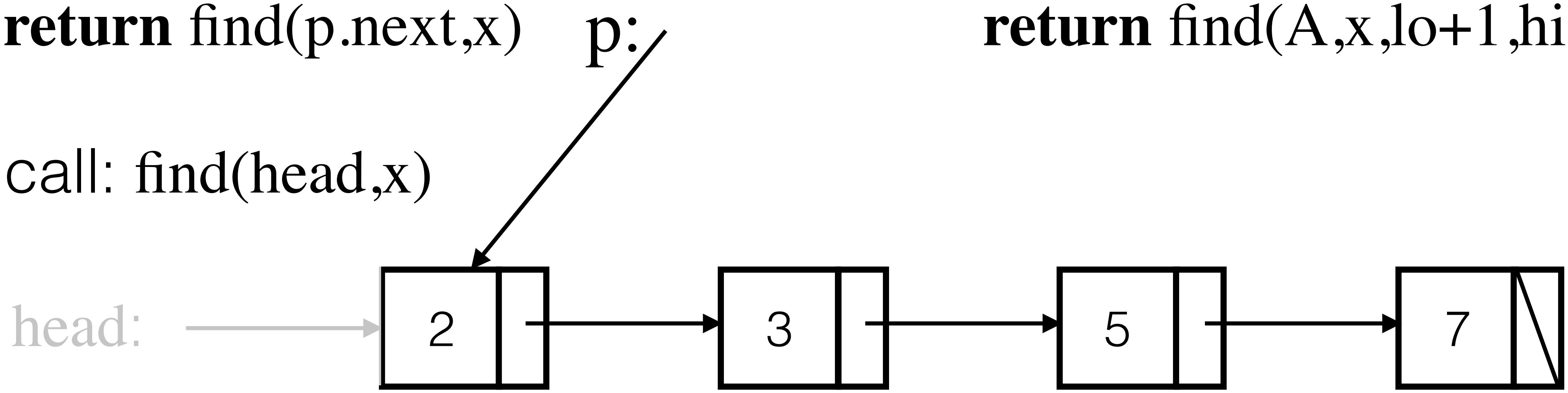
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

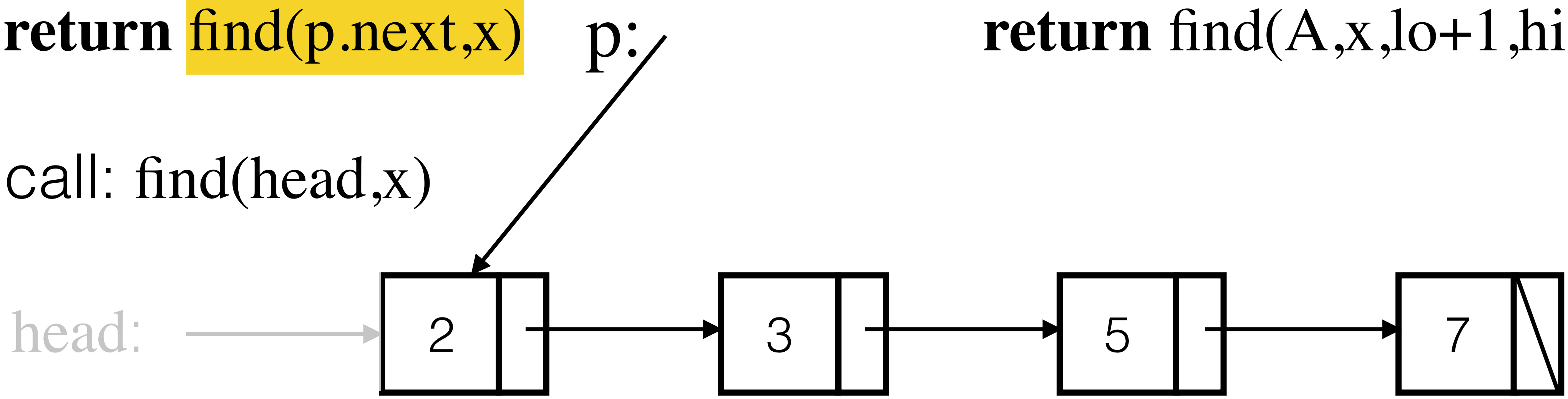
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

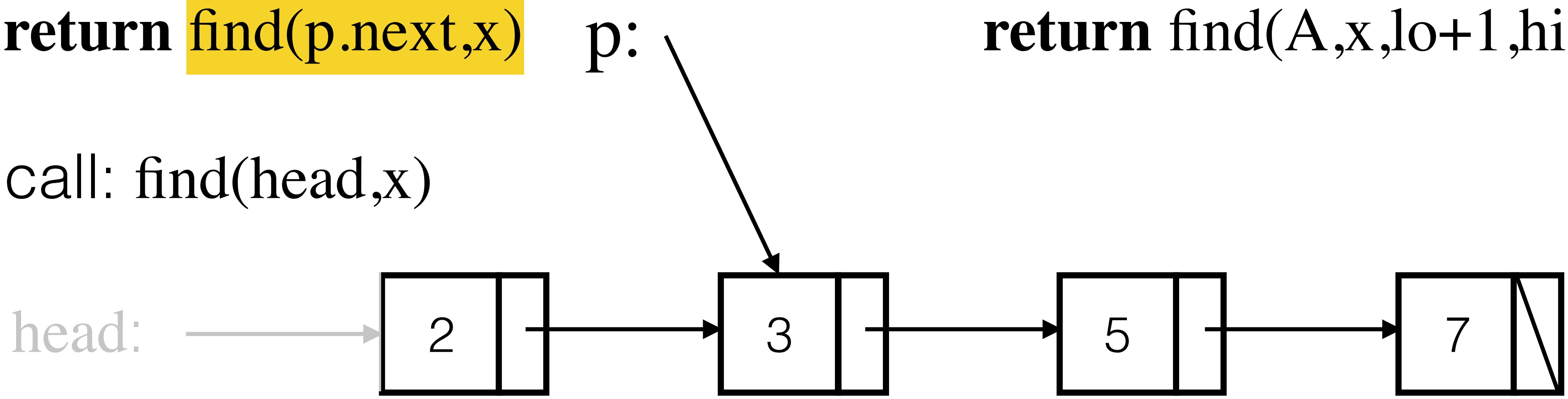
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

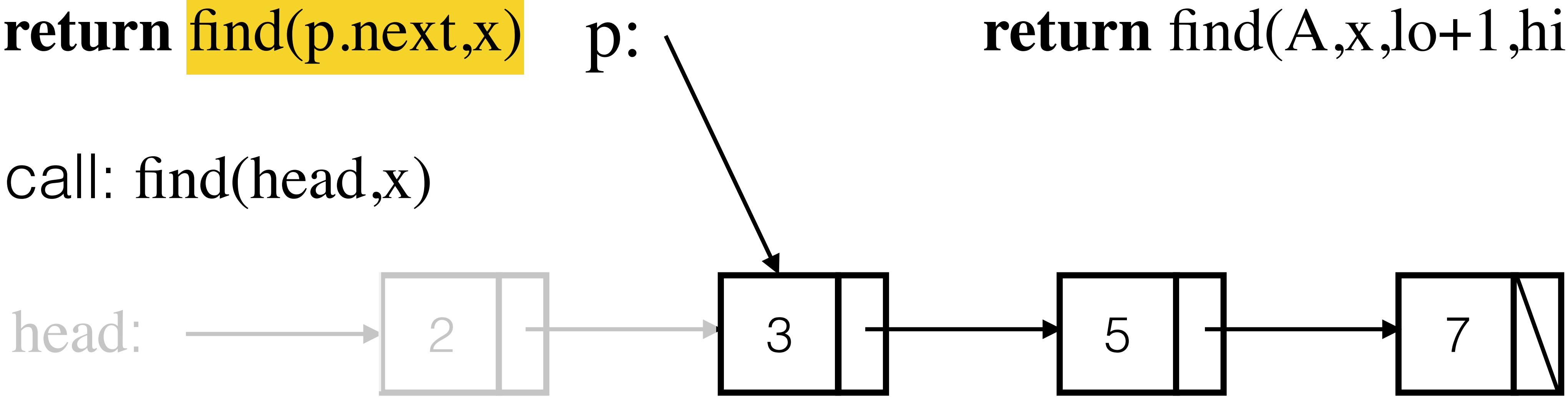
- Solve the problem for a sub-instance and use the solution to solve the full instance

(note similarity to array version)

```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Recursive Processing: List

- Solve the problem for a sub-instance and use the solution to solve the full instance

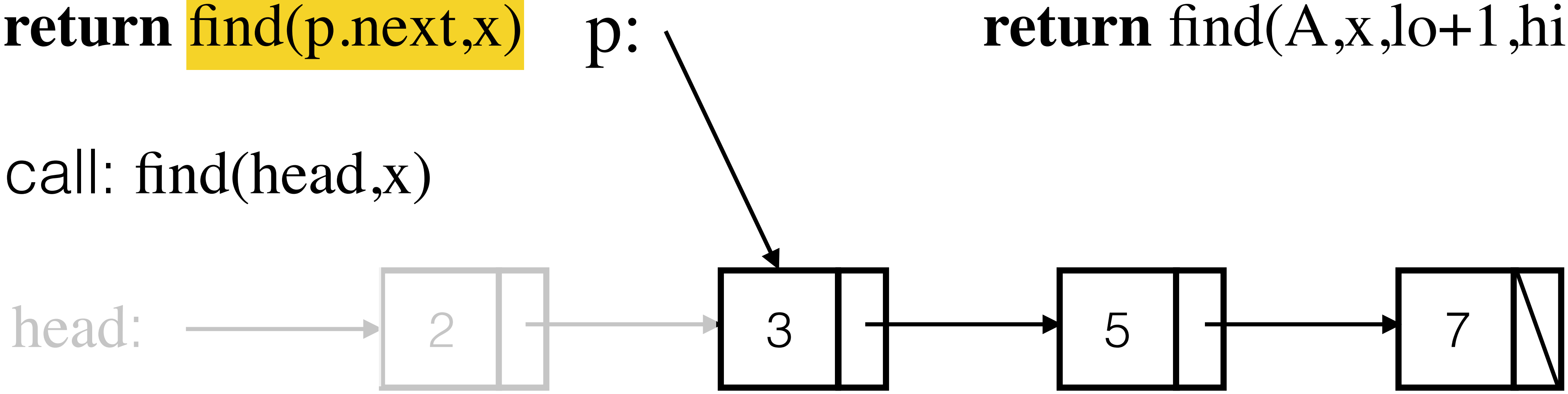
```
function find(p,x)
    if p = null
        return p
    else if p.val = x
        return p
    else
        return find(p.next,x)
```

We will discuss recursion properly in Week 3

(note similarity to array version)

```
function find(A,x,lo,hi)
    if lo > hi
        return -1
    else if A[lo] = x
        return lo
    else
        return find(A,x,lo+1,hi)
```

Initial call: find(head,x)



Abstract DataTypes

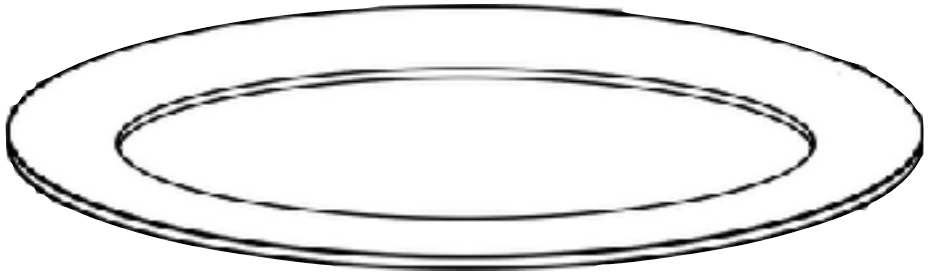
- A collection of data items, and a family of operations that operate on that data
- Think of an ADT as a set of contracts, an **interface**
- We must still **implement** these promises, but it is an advantage to separate the implementation of the ADT from the “concept” (i.e. the interface it provides)
- Good programming practice is to support this separation
  - Nothing outside of the definition of the ADT should refer to anything inside, except through function calls and basic operations

Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT

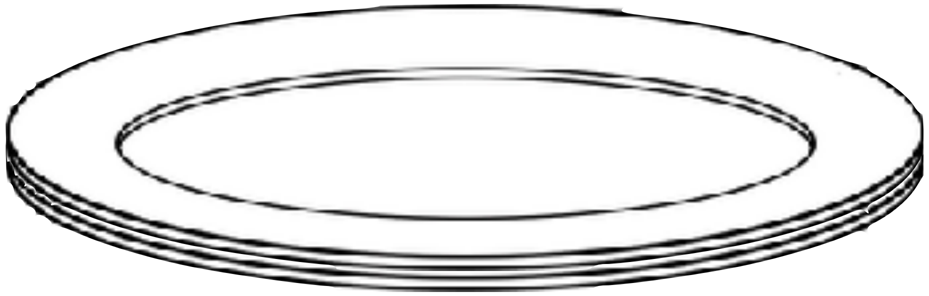
Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



Fundamental Data Structure: The Stack

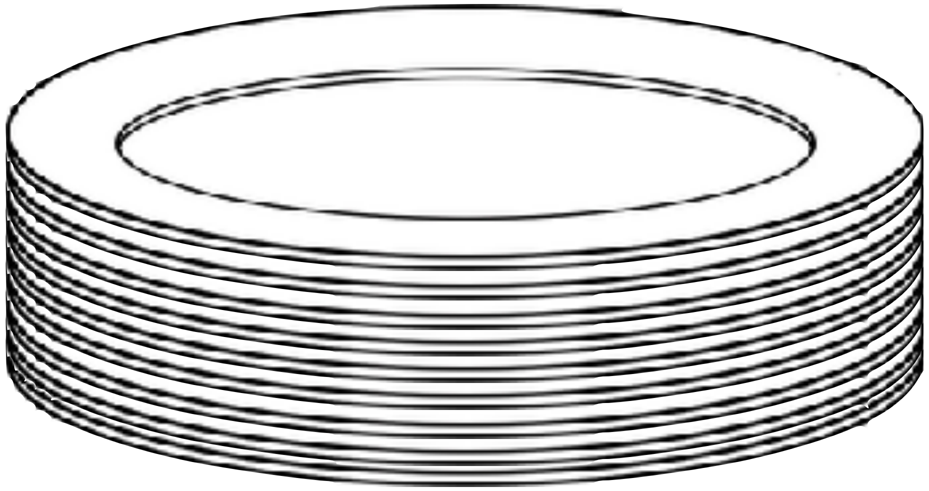
- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …



- Usually implemented as an ADT

Fundamental Data Structure: The Stack

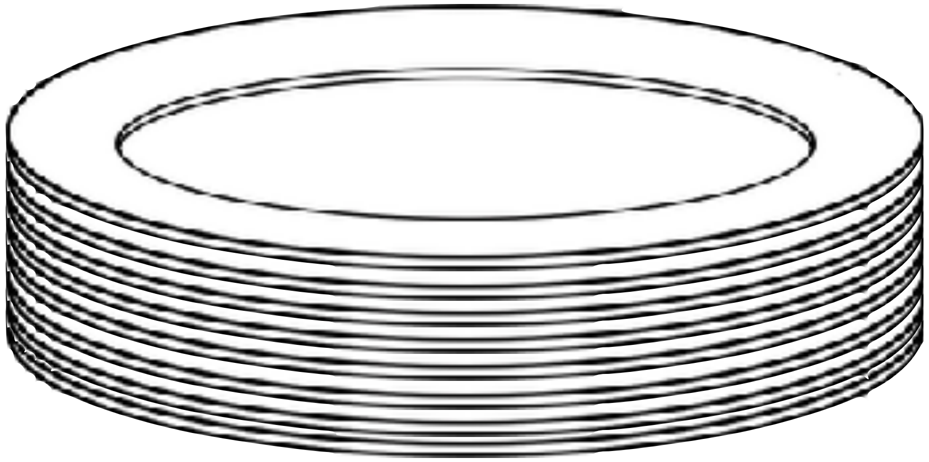
- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …



- Usually implemented as an ADT

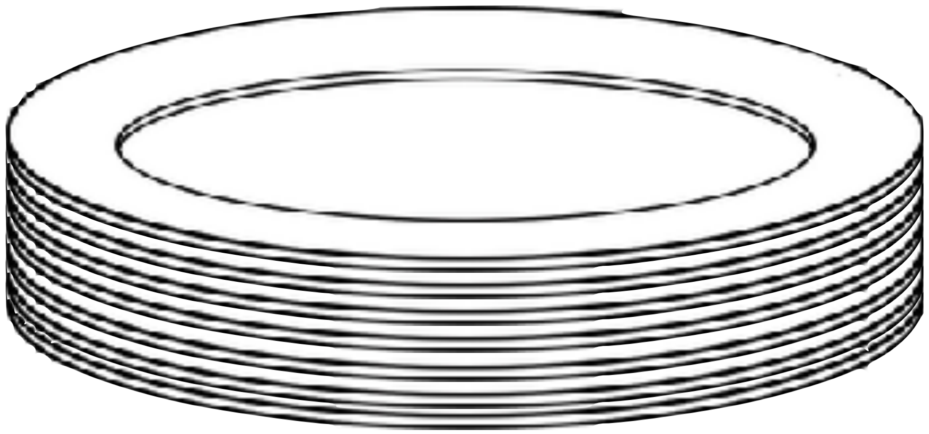
Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



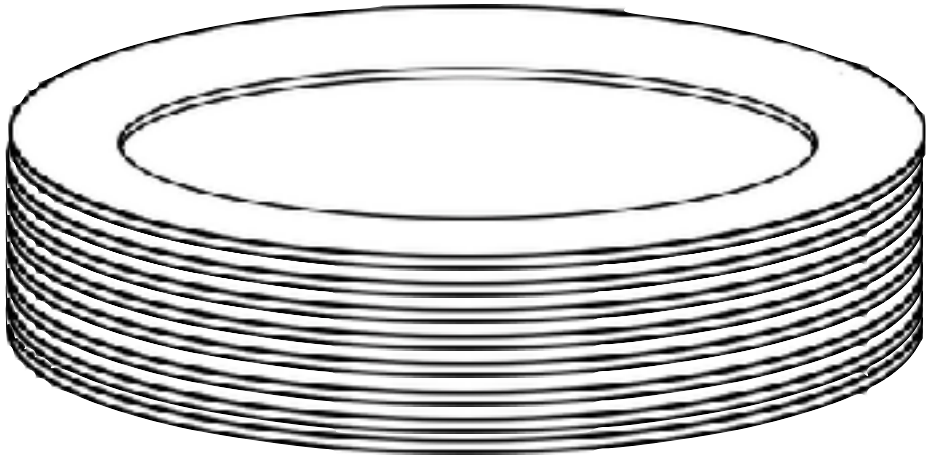
Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



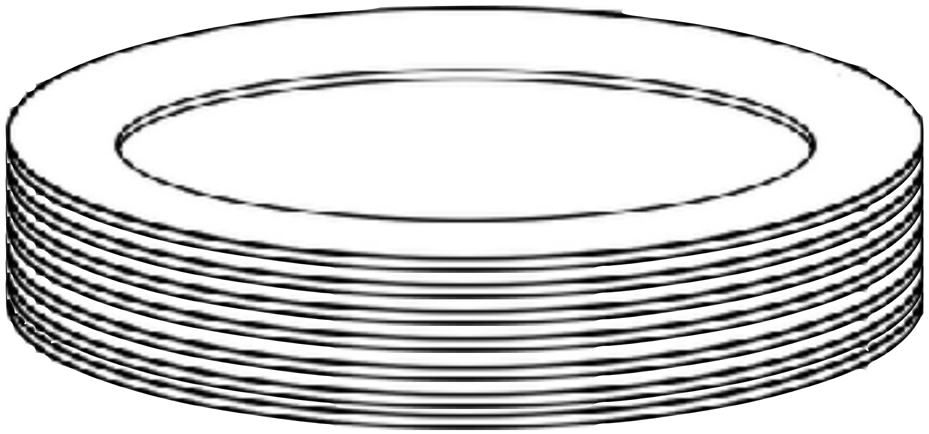
Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



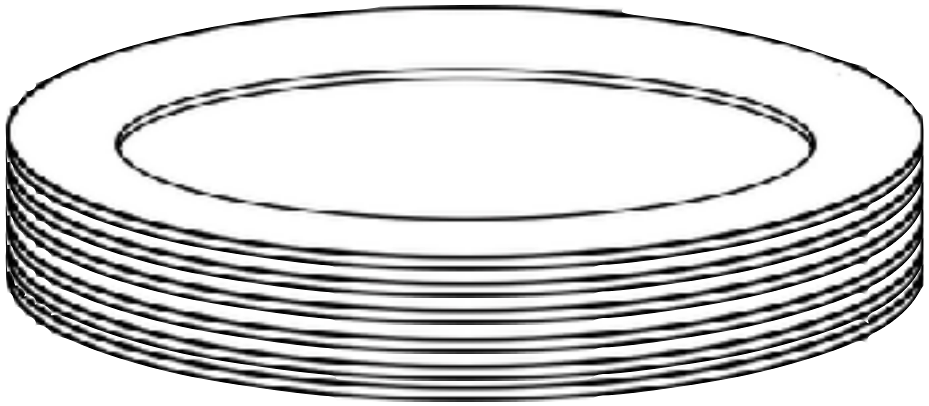
Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



Fundamental Data Structure: The Stack

- Last-In-First-Out (LIFO)
- Operations:
  - CreateStack
  - Push
  - Pop
  - Top
  - EmptyStack?
  - …
- Usually implemented as an ADT



Stack Implementation: Array

Stack Implementation: Array

6	9	2	3	7		
0	1	2	3	4	5	6

Stack Implementation: Array

6	9	2	3	7		
0	1	2	3	4	5	6

top: 5

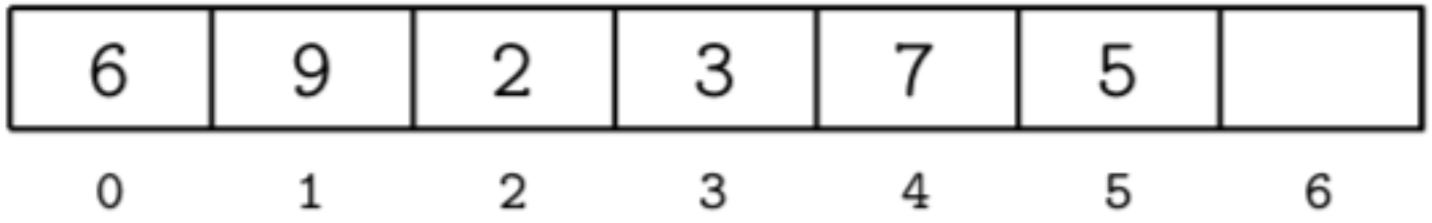
Stack Implementation: Array

6	9	2	3	7		
0	1	2	3	4	5	6

top: 5

Push(5)

Stack Implementation: Array



6	9	2	3	7	5	
0	1	2	3	4	5	6

top: 5

Push(5)

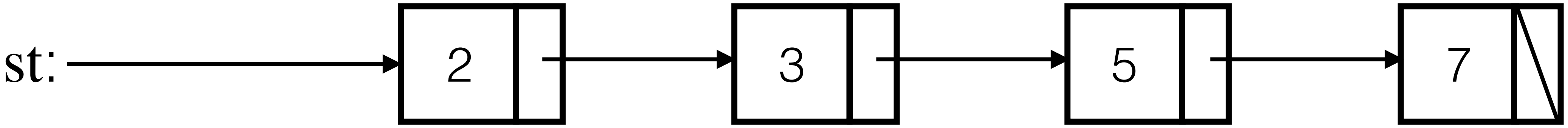
Stack Implementation: Array

6	9	2	3	7	5	
0	1	2	3	4	5	6

top: 6

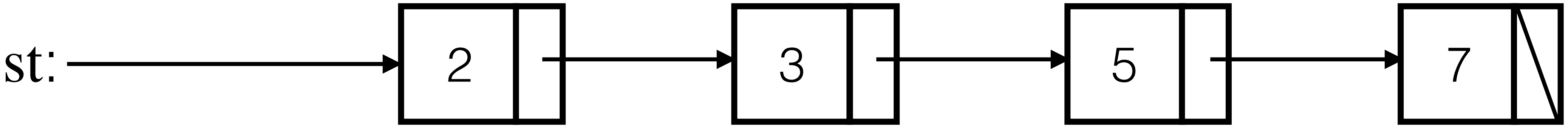
Push(5)

Stack Implementation: Linked List



```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

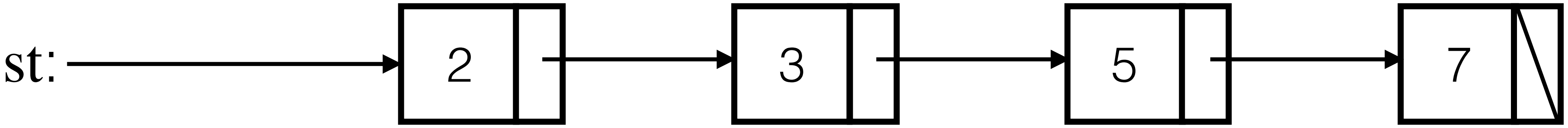
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

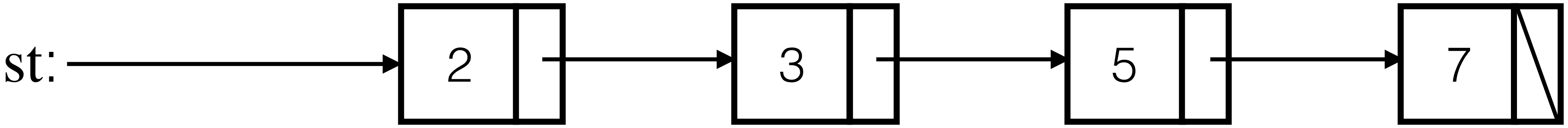
Stack Implementation: Linked List

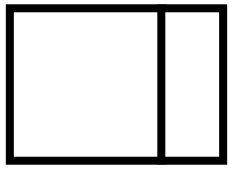


Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

Stack Implementation: Linked List

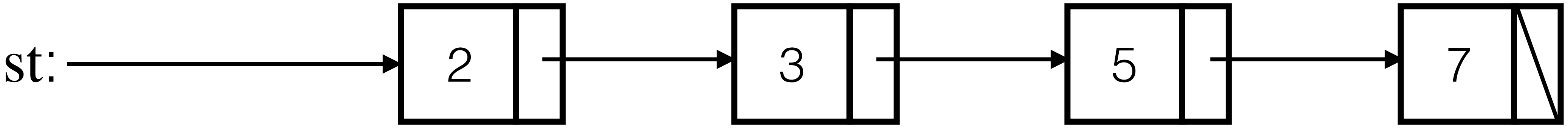


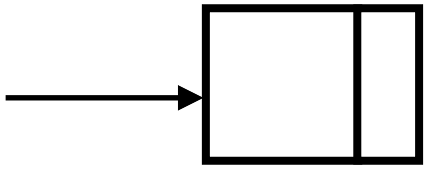


Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

Stack Implementation: Linked List

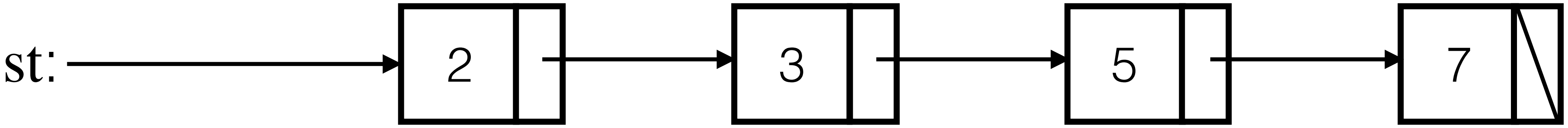


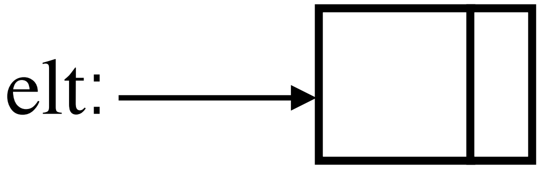


Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

Stack Implementation: Linked List

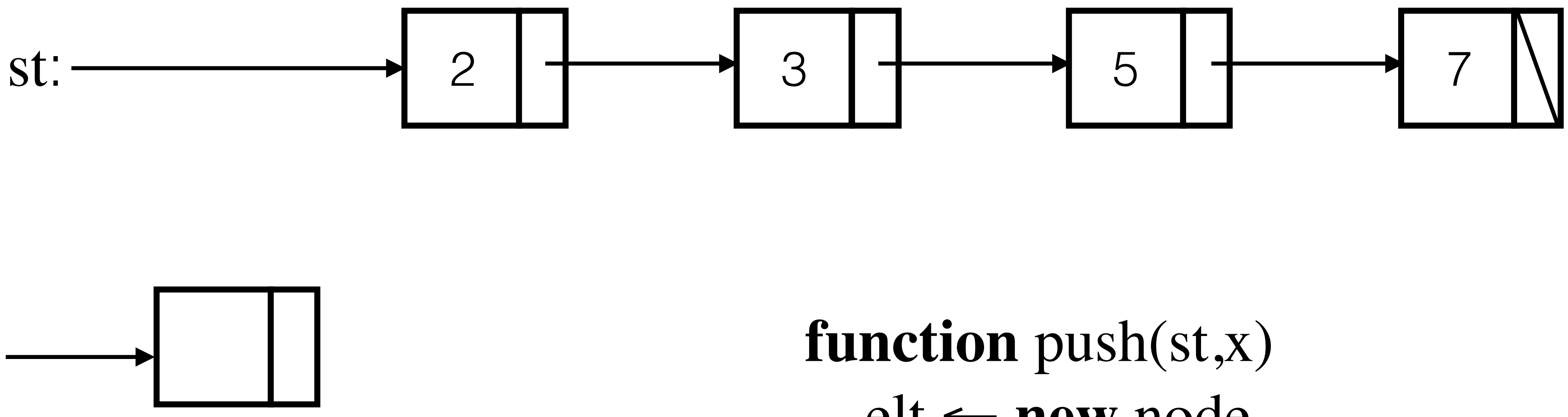




Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

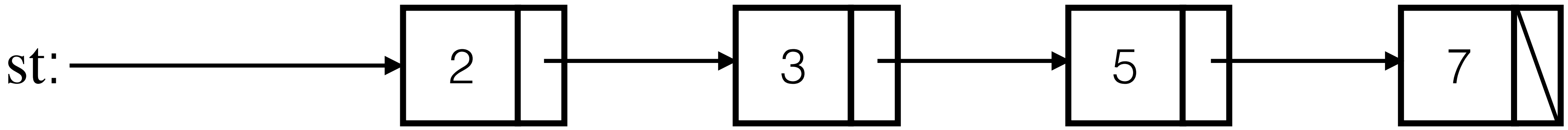
Stack Implementation: Linked List

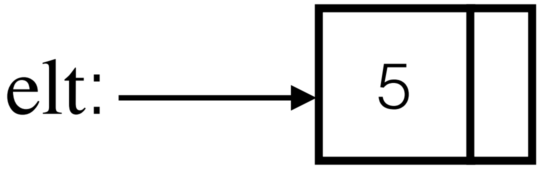


Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

Stack Implementation: Linked List

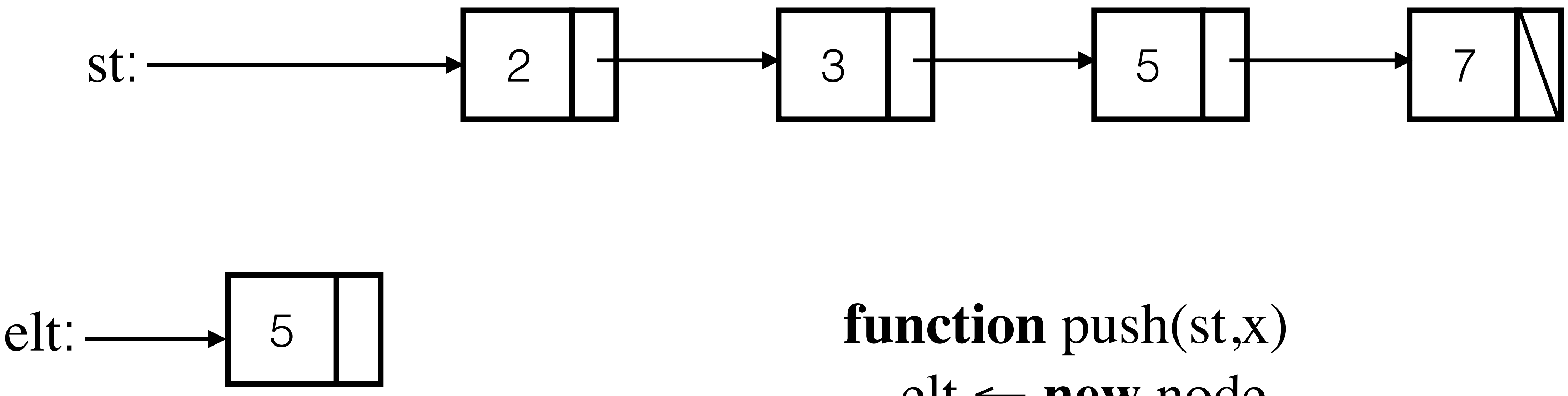




Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

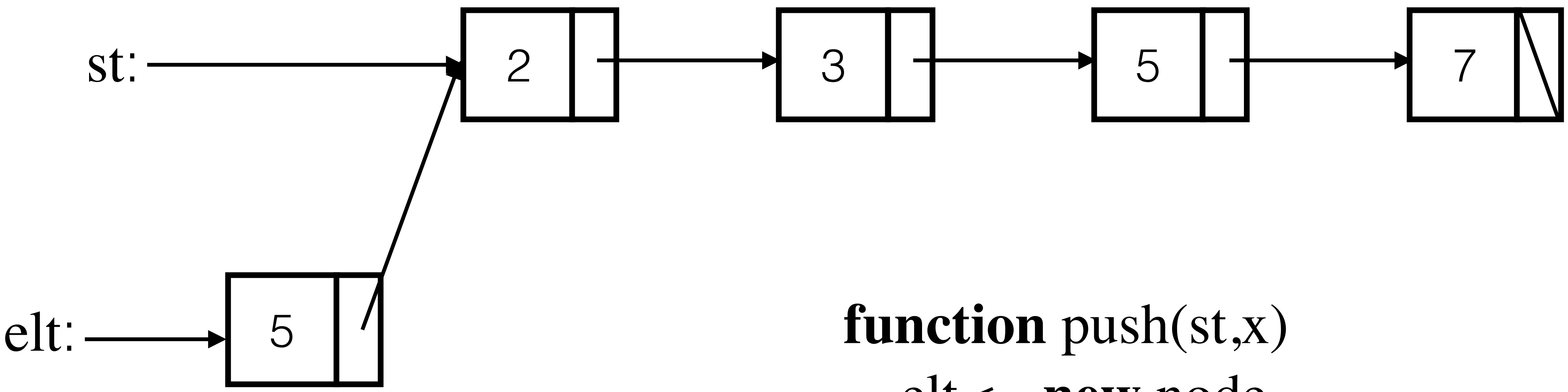
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

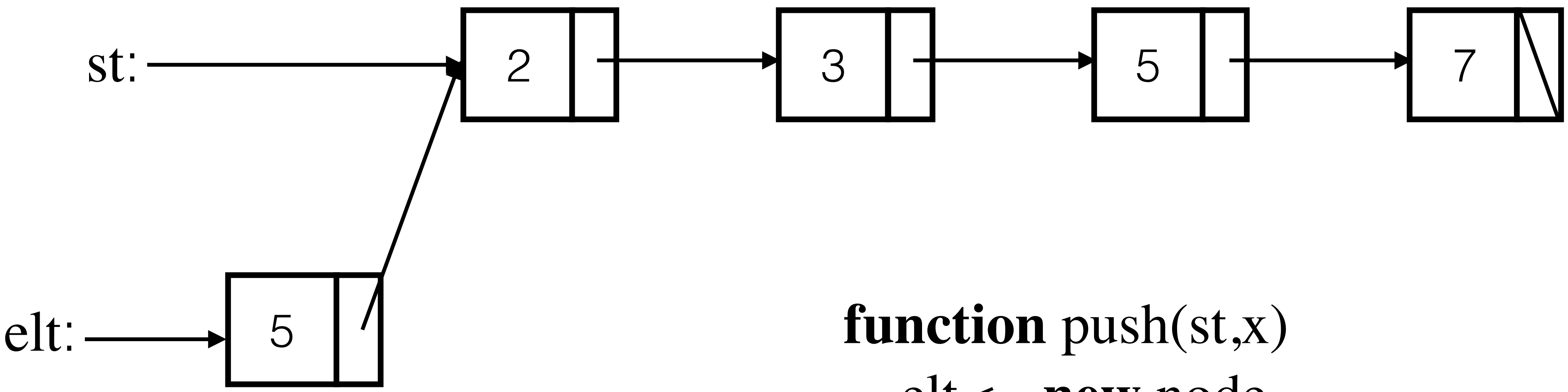
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

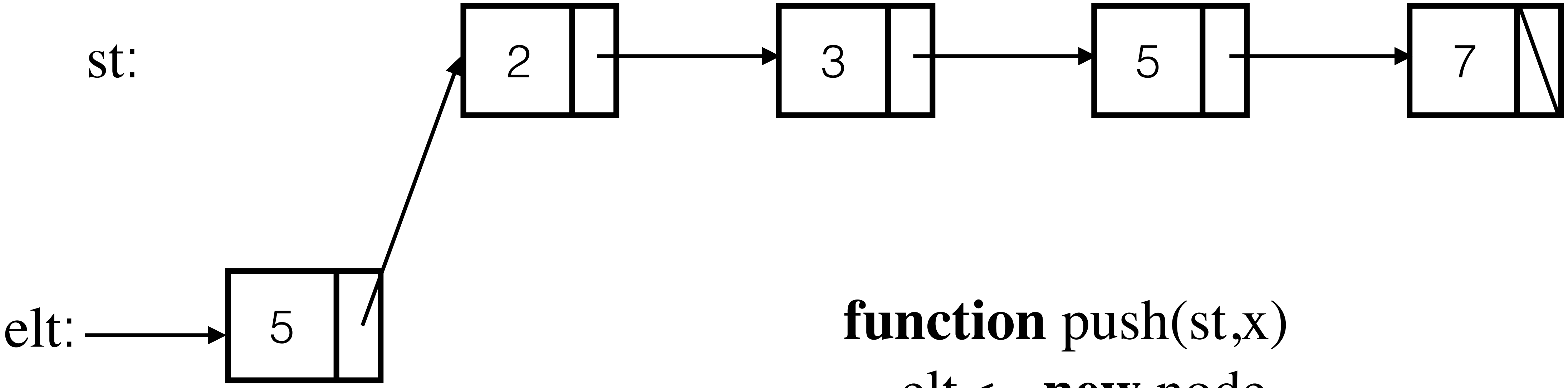
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

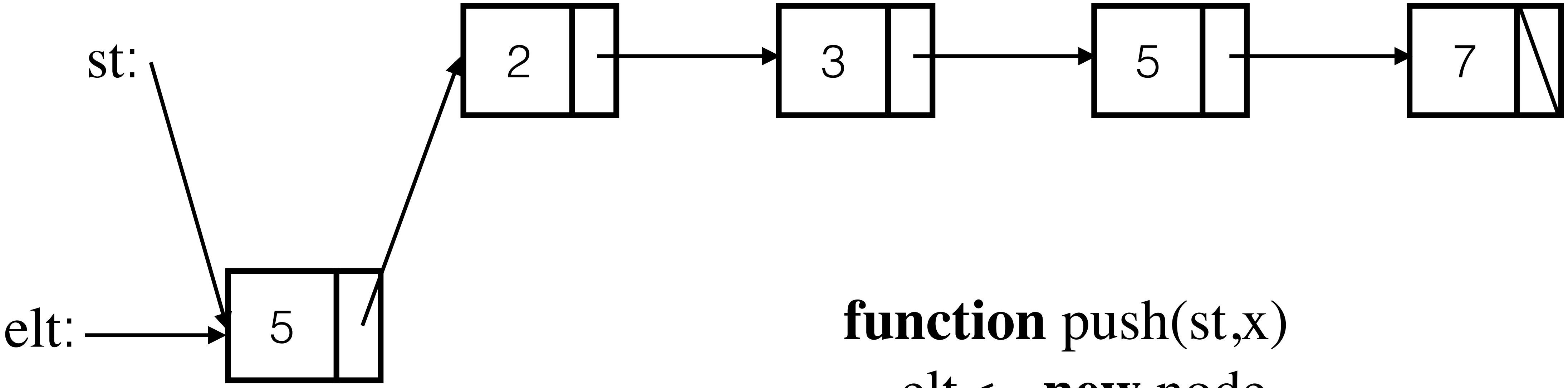
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

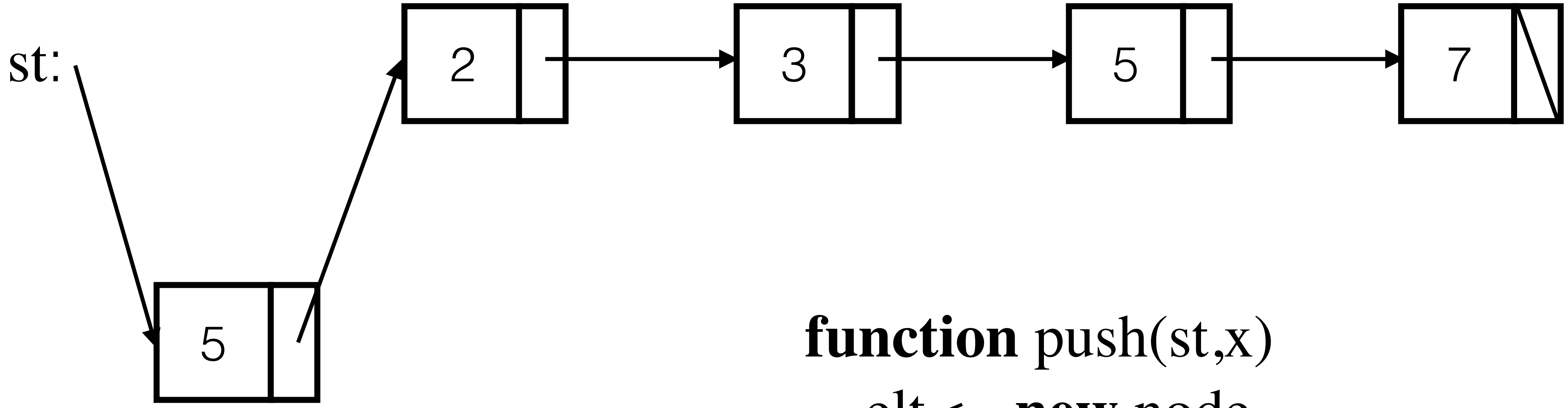
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

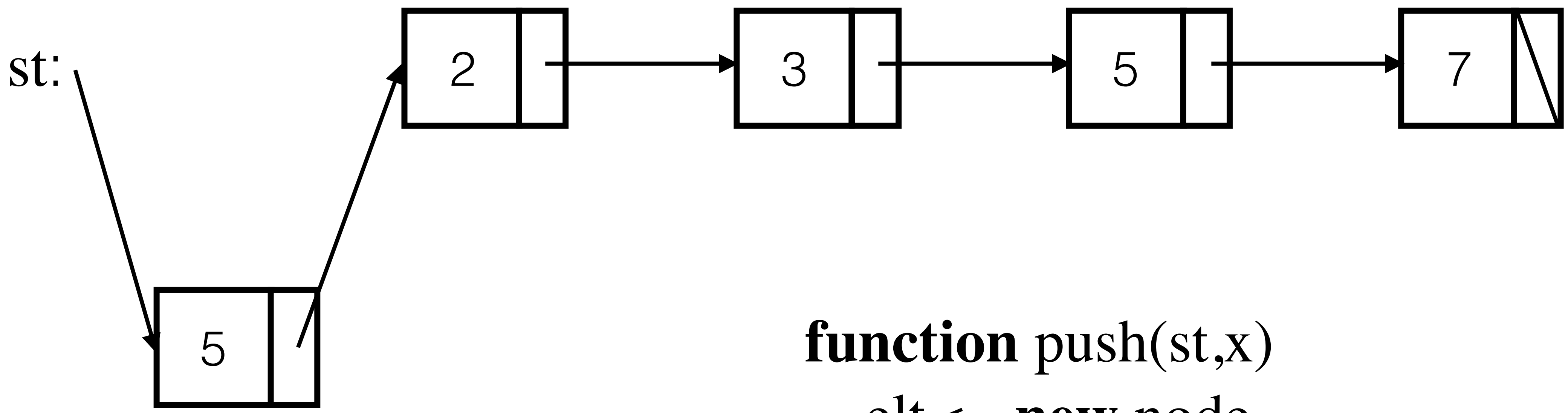
Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

Stack Implementation: Linked List



Push(5)

```
function push(st,x)
    elt ← new node
    elt.val ← x
    elt.next ← st
    st ← elt
    return st
```

See https://www.cs.usfca.edu/~galles/visualization/Algorithms.html for more visualisations

Pseudo Code

- On the previous slide, we assumed that a “node” has two attributes: a “val” which is its value, and a “next” which points to the rest of the list.

- There is no standard for pseudo-code. Use the examples in Levitin as a guide. Cormen et al. pages 20–22 (in Reading Resources) has a list of standard conventions used with pseudo-code which are good to follow, except we use ← as the assignment operator.

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Fundamental Data Structure: Queues

- First-In-First-Out (FIFO)
- Operations:
  - CreateQueue
  - Enqueue
  - Dequeue
  - Head
  - EmptyQueue?
  - …

Other Data Structures

- We will meet many other (abstract) data structures, e.g.
  - The priority queue
  - Various types of “tree”
  - Various types of “graph”
- If you check out algorithm animation tools or advanced algorithm books, you will meet exotic data structures such as splay trees and skip lists.

Next Week

- Algorithm analysis—how to reason about an algorithm’s resource consumption.