

A Concurrent Program Logic with a Future and History

Roland Meyer, Thomas Wies, Sebastian Wolff

[OOPSLA'22]

Motivation

"*Programs = Algorithms + Data Structures*"

–Niklaus Wirth, 1978

Motivation

"*Programs = Algorithms + Data Structures*"

–Niklaus Wirth, 1978

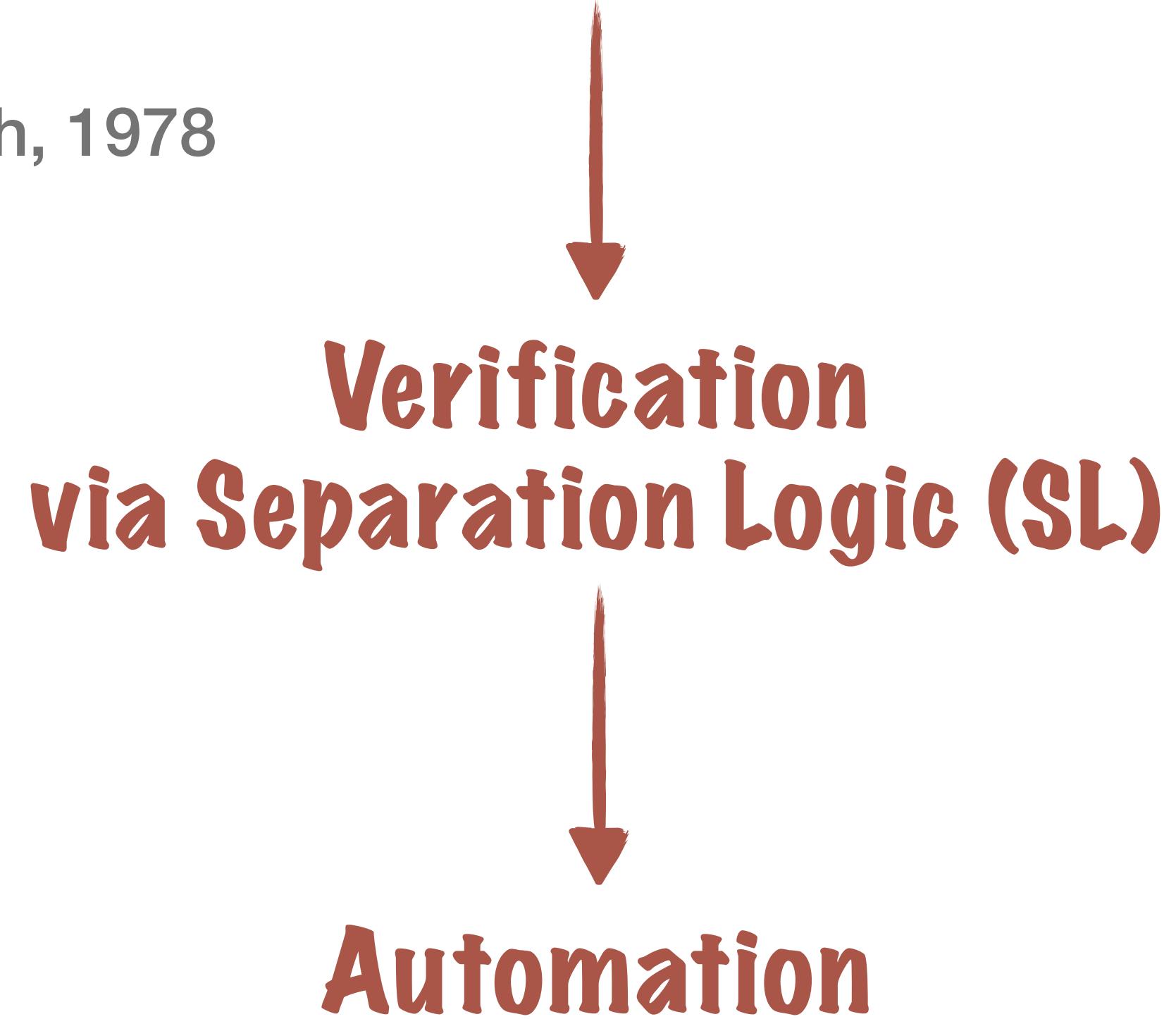


**Verification
via Separation Logic (SL)**

Motivation

"*Programs = Algorithms + Data Structures*"

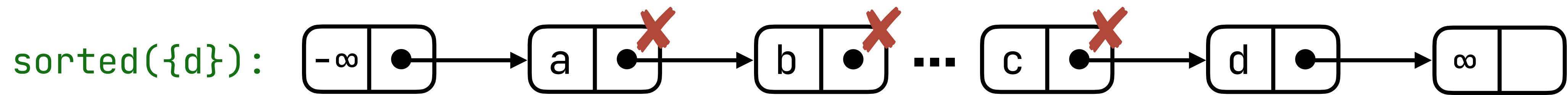
–Niklaus Wirth, 1978



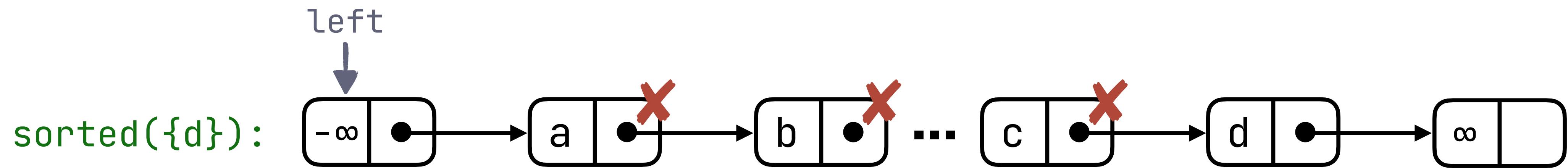
Verification Challenge 1: **Complex Unbounded Updates**

Verification Challenge 2: **Linearizability**

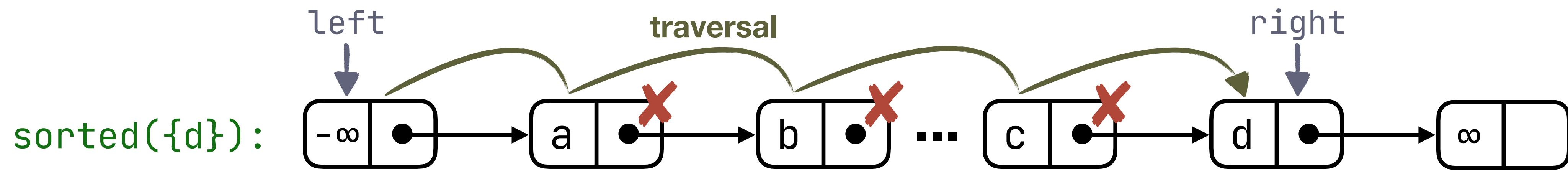
Example: Unbounded Removal



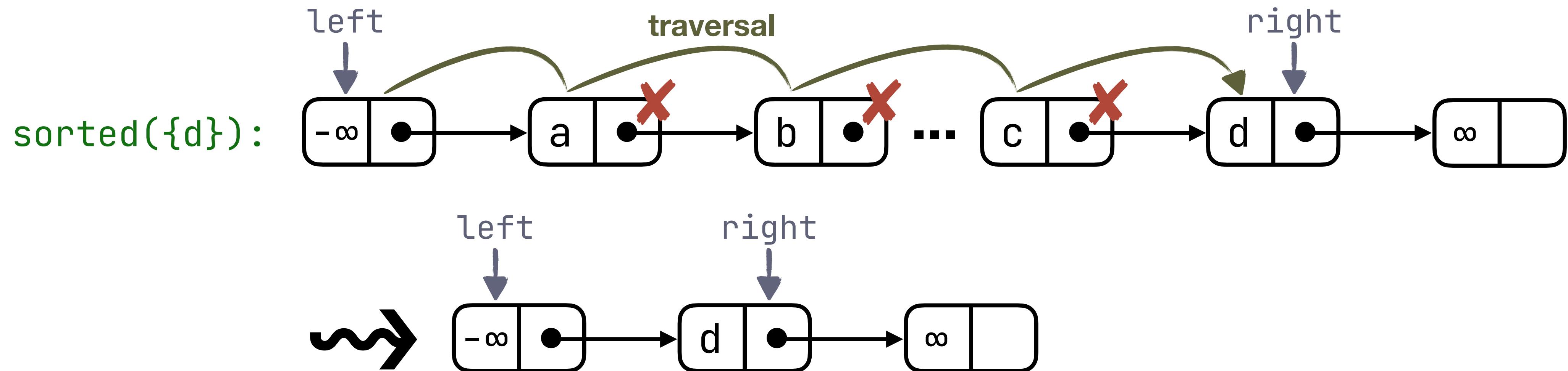
Example: Unbounded Removal



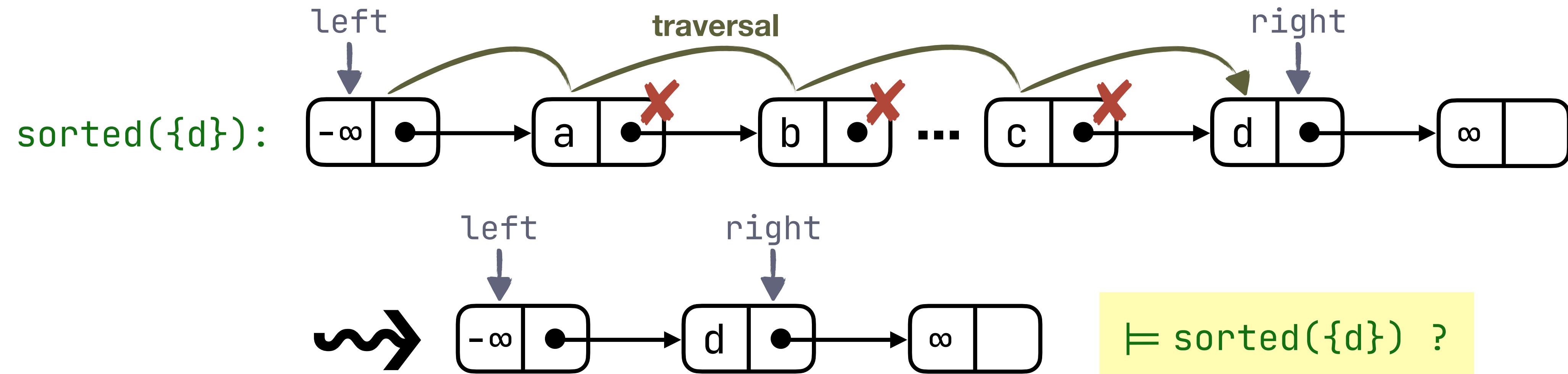
Example: Unbounded Removal



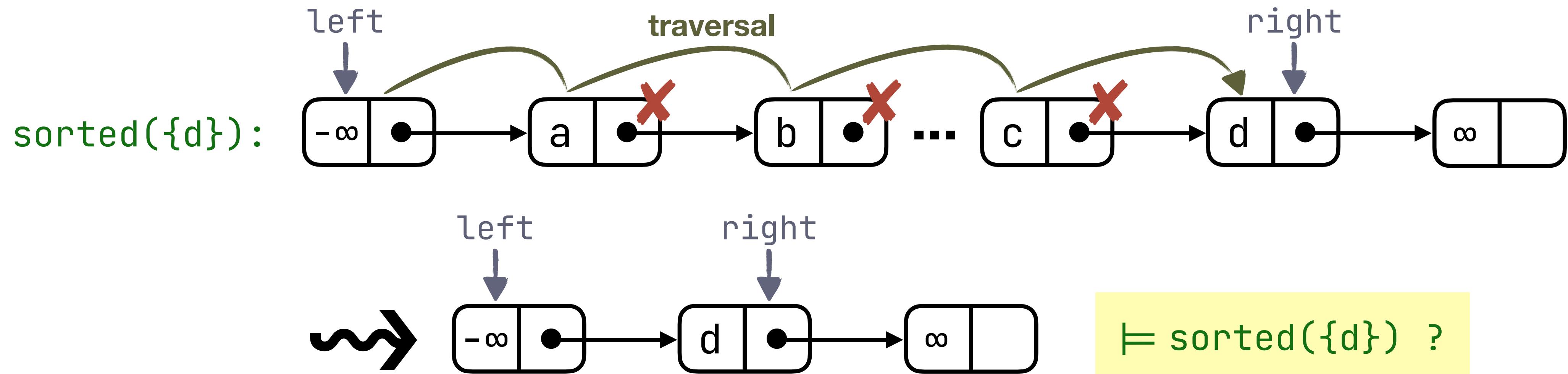
Example: Unbounded Removal



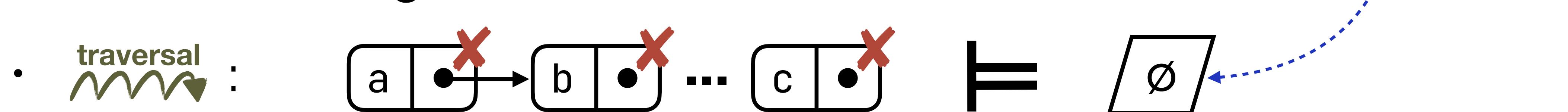
Example: Unbounded Removal



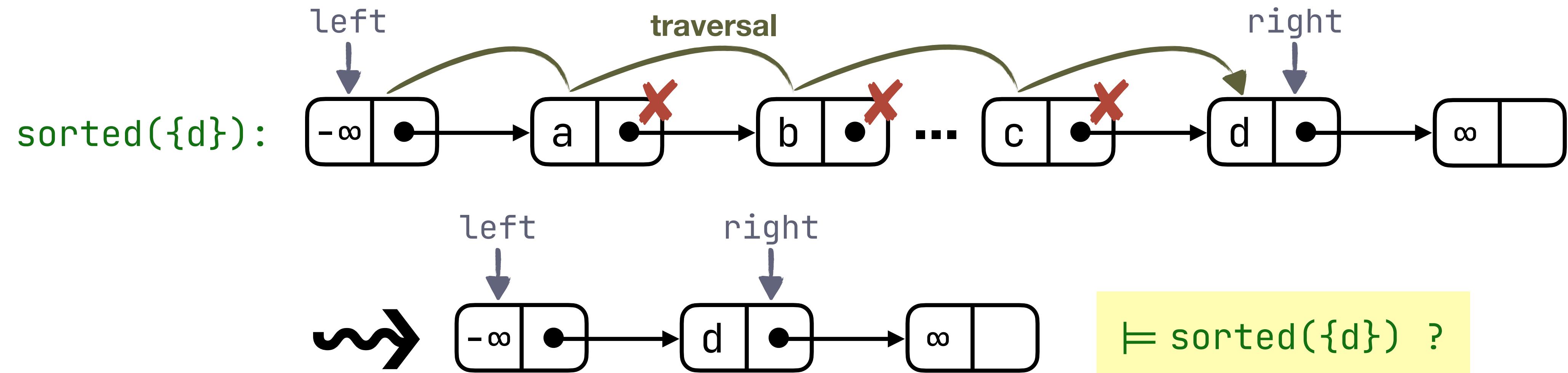
Example: Unbounded Removal



Classical Reasoning



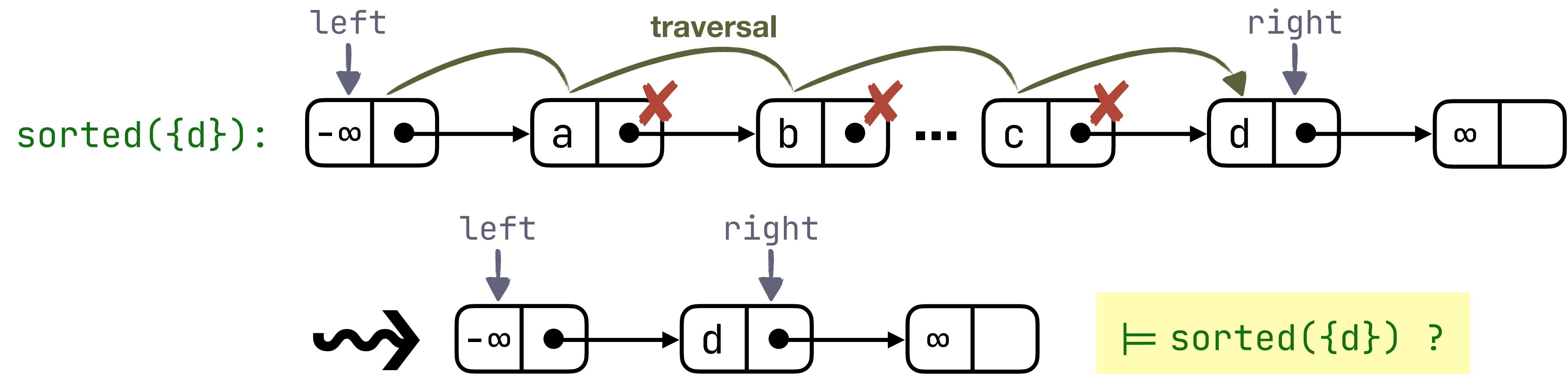
Example: Unbounded Removal



Classical Reasoning

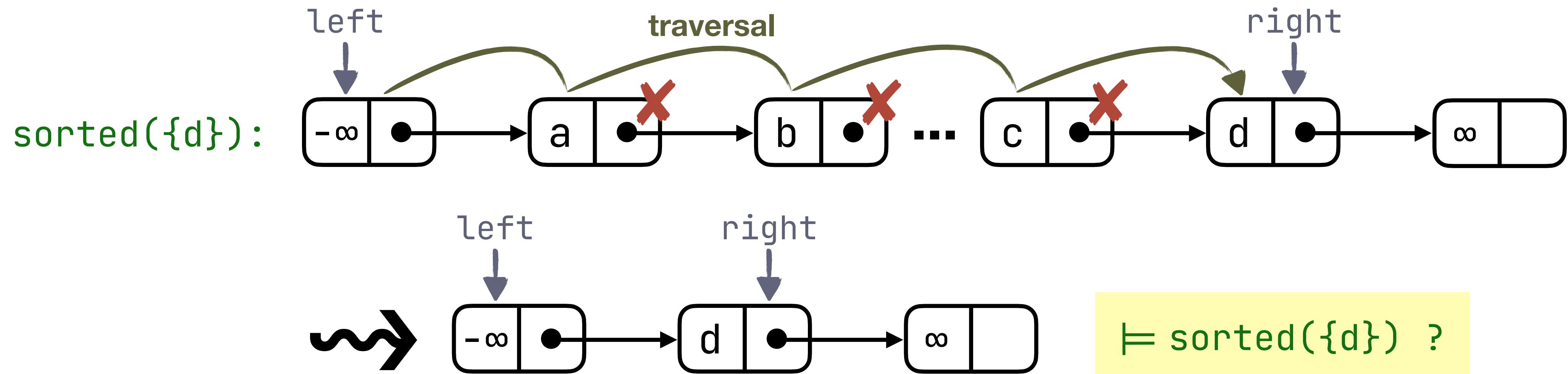
- : \models e.g. recursive SL predicates
- : \models induction over recursive predicates

Example: Unbounded Removal

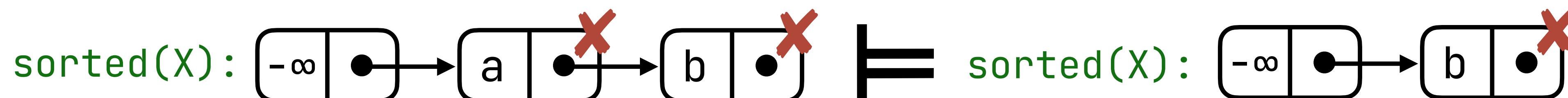


Contribution: Ghost Update Chunks

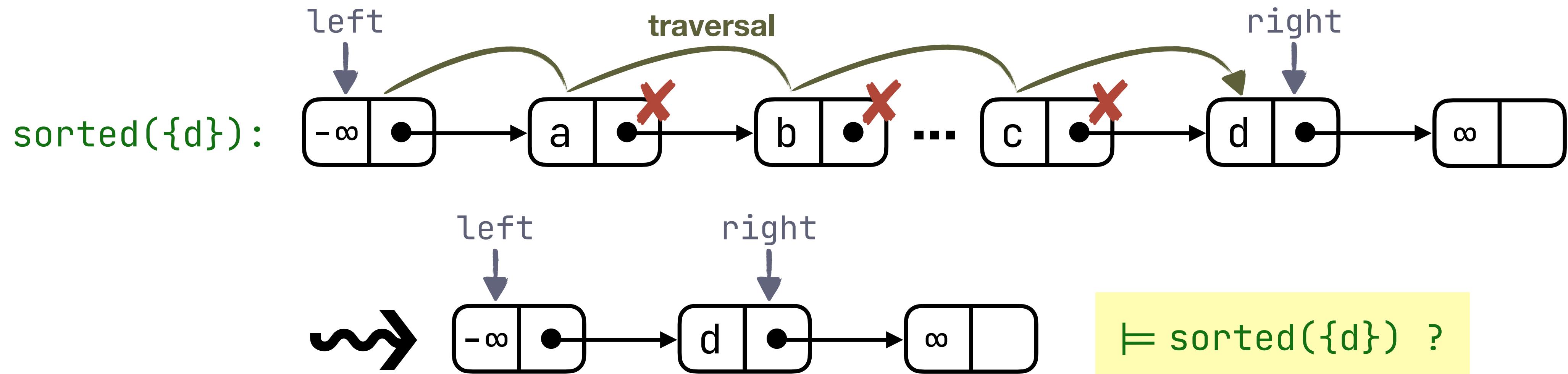
Example: Unbounded Removal



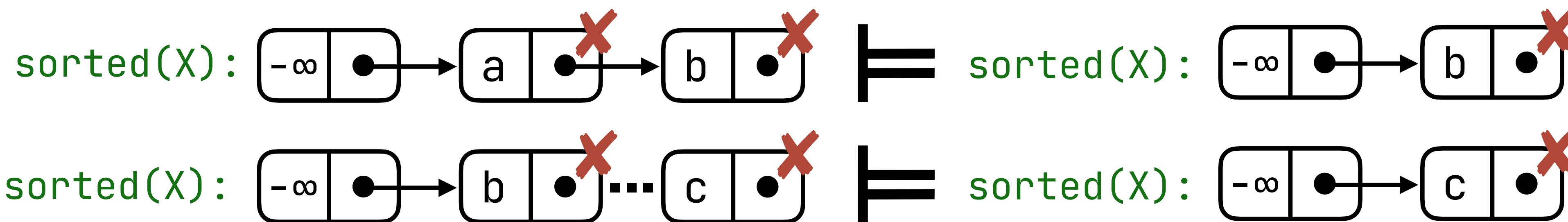
Contribution: Ghost Update Chunks



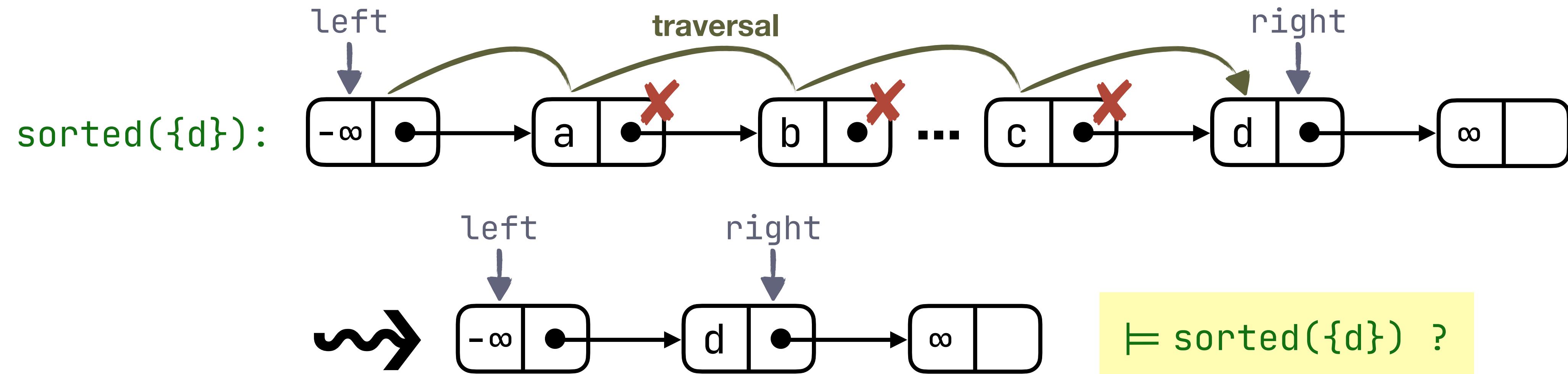
Example: Unbounded Removal



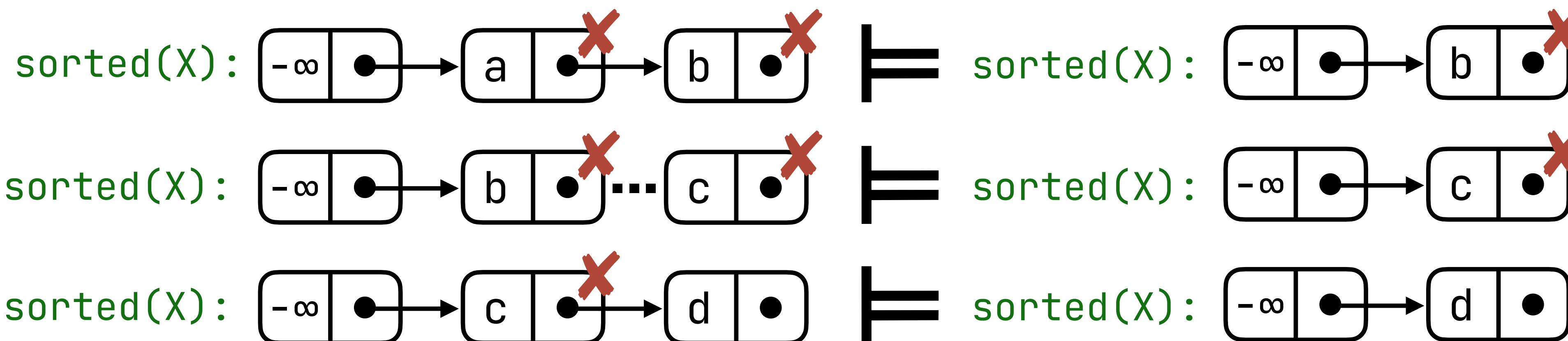
Contribution: Ghost Update Chunks



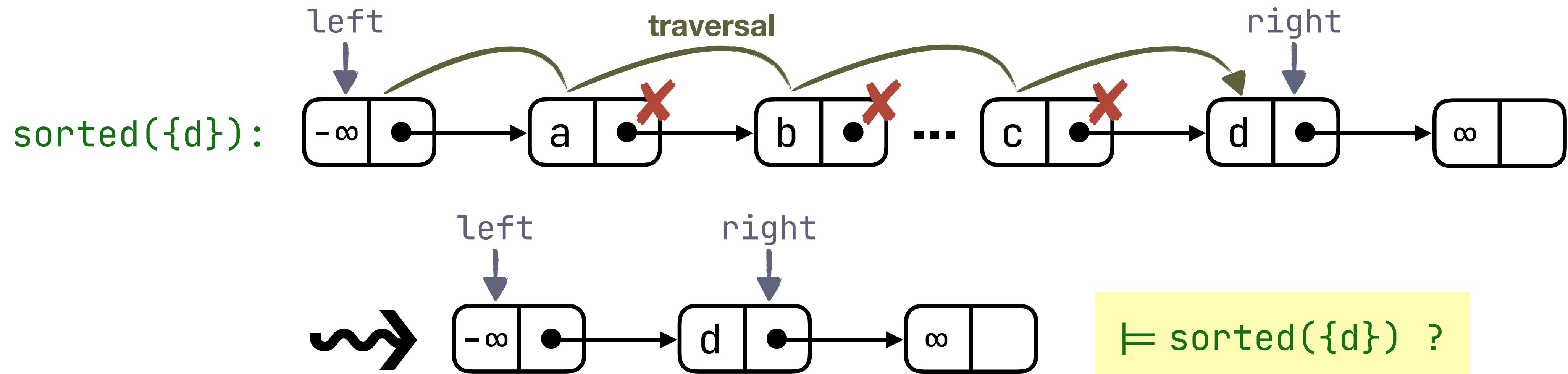
Example: Unbounded Removal



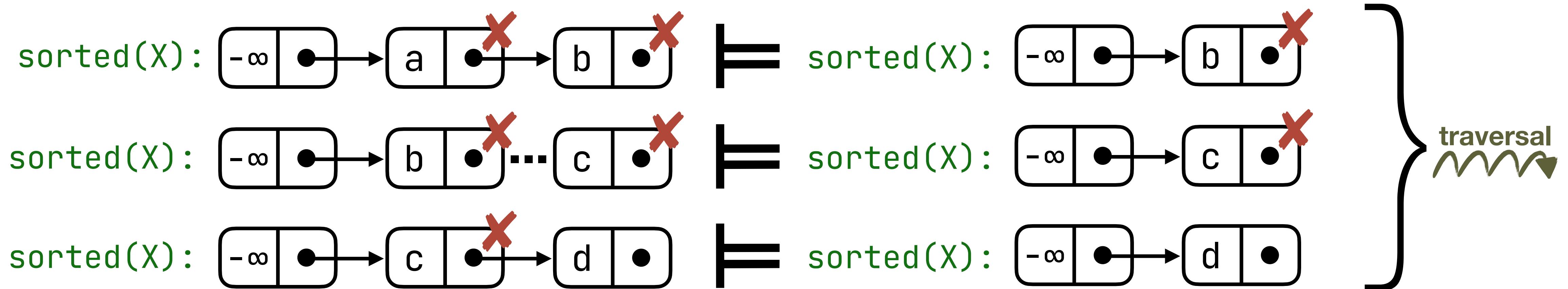
Contribution: Ghost Update Chunks



Example: Unbounded Removal

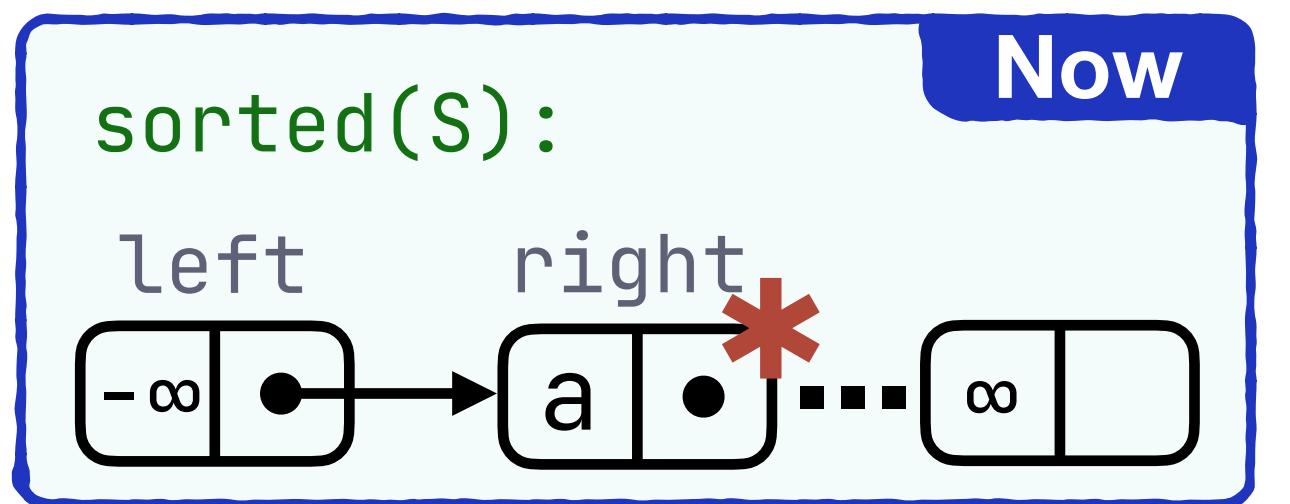


Contribution: Ghost Update Chunks



Ghost Update Chunks

```
// traversal step 1
```



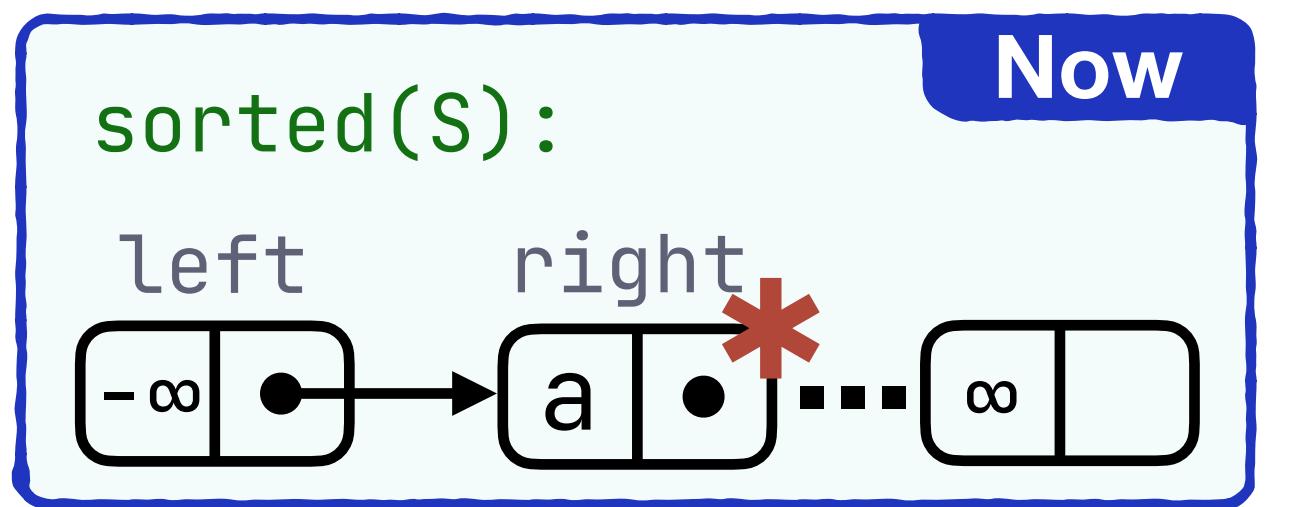
```
assume(right→mark);
```

```
next := right→next;
```

```
right := next;
```

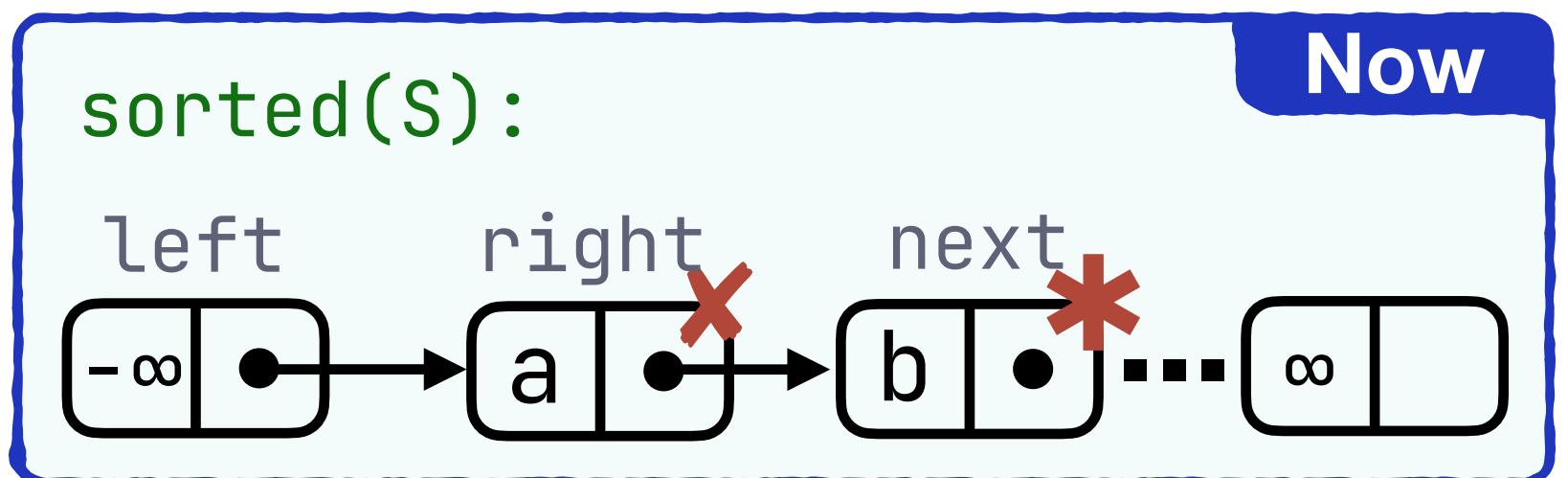
Ghost Update Chunks

```
// traversal step 1
```



```
assume(right→mark);
```

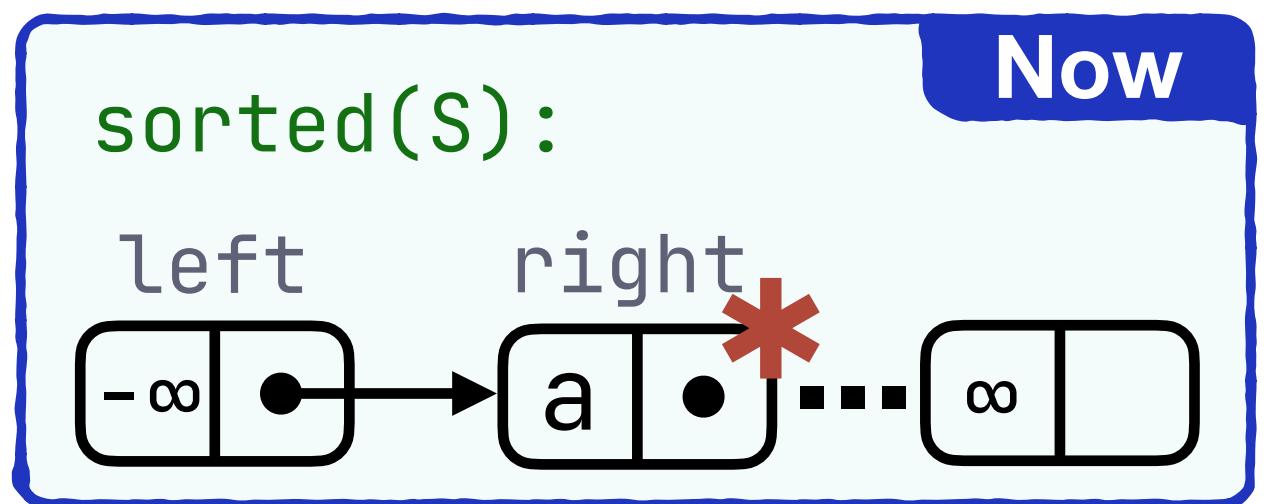
```
next := right→next;
```



```
right := next;
```

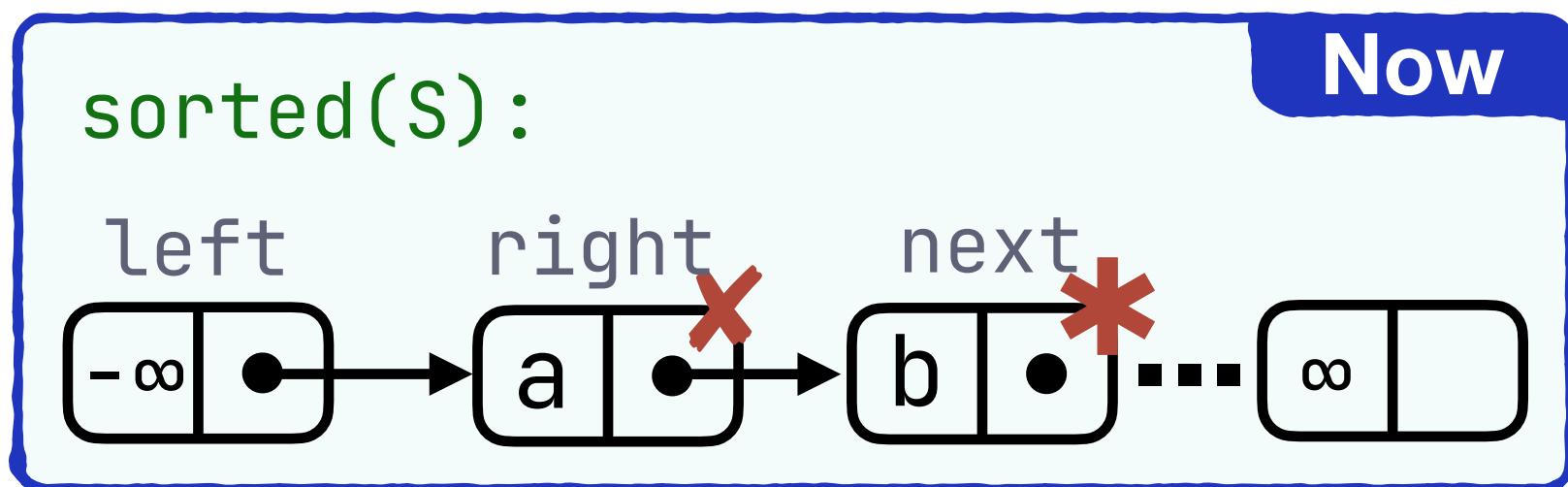
Ghost Update Chunks

```
// traversal step 1
```



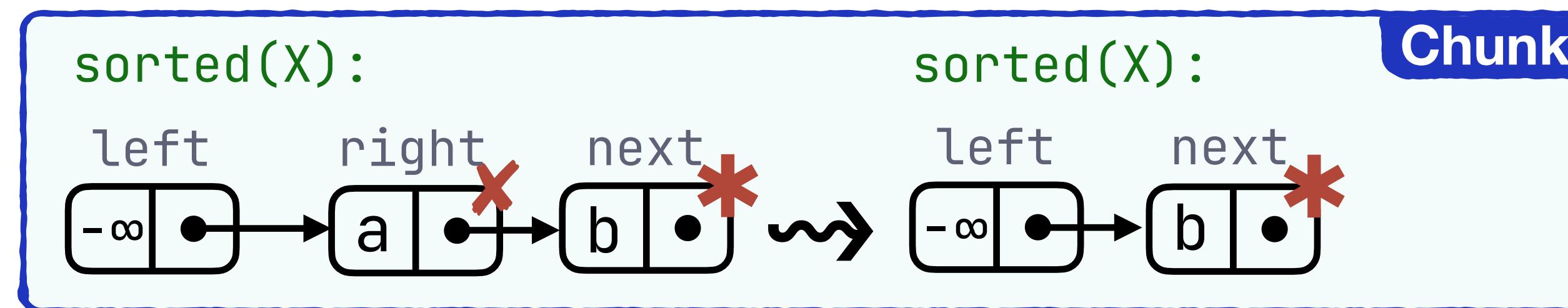
```
assume(right→mark);
```

```
next := right→next;
```



```
right := next;
```

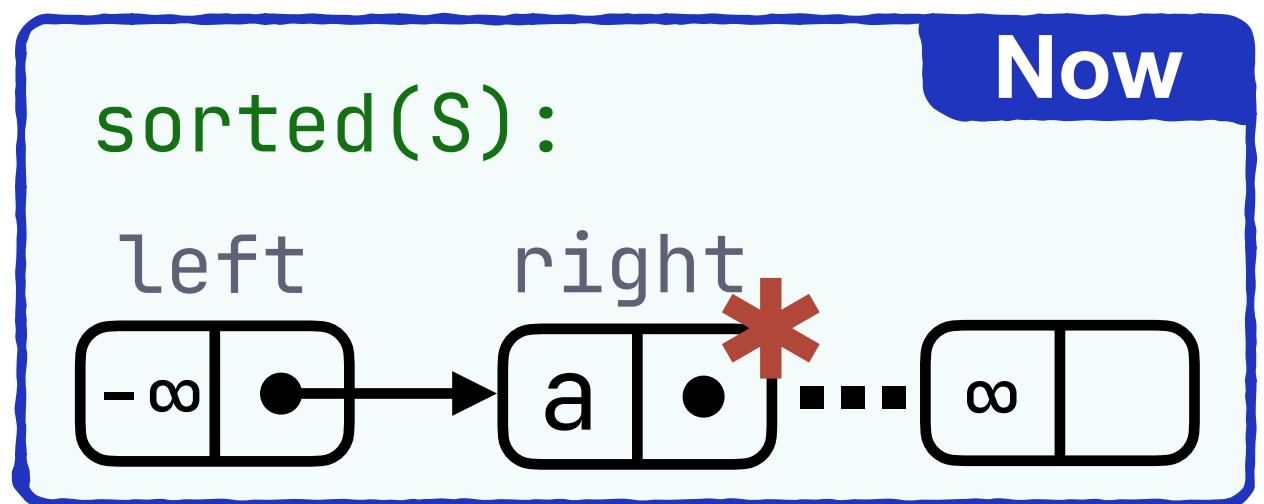
*



INTRO

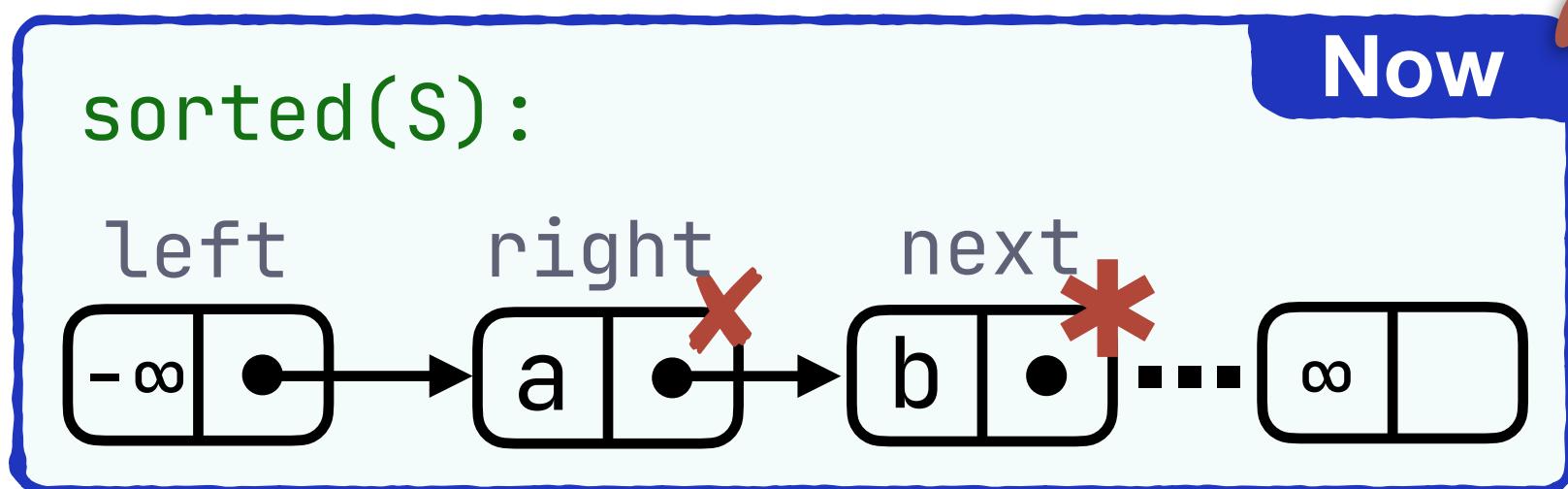
Ghost Update Chunks

```
// traversal step 1
```



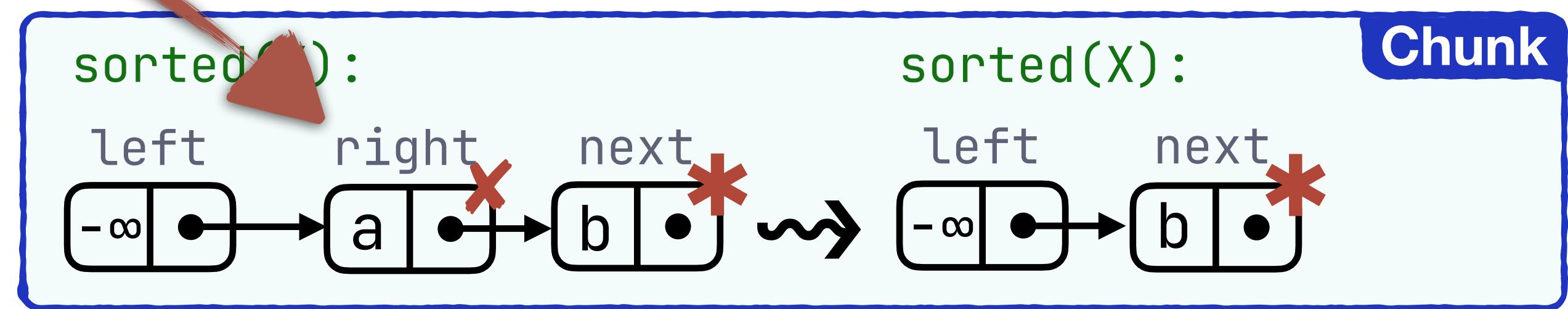
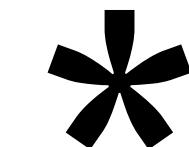
```
assume(right→mark);
```

```
next := right→next;
```

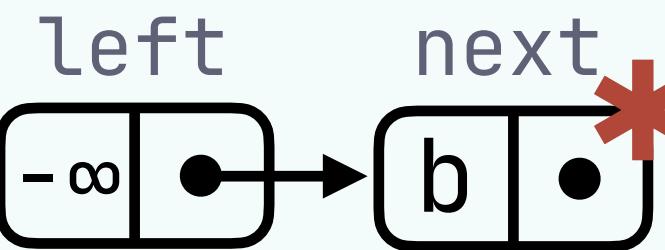


```
right := next;
```

DISCHARGE



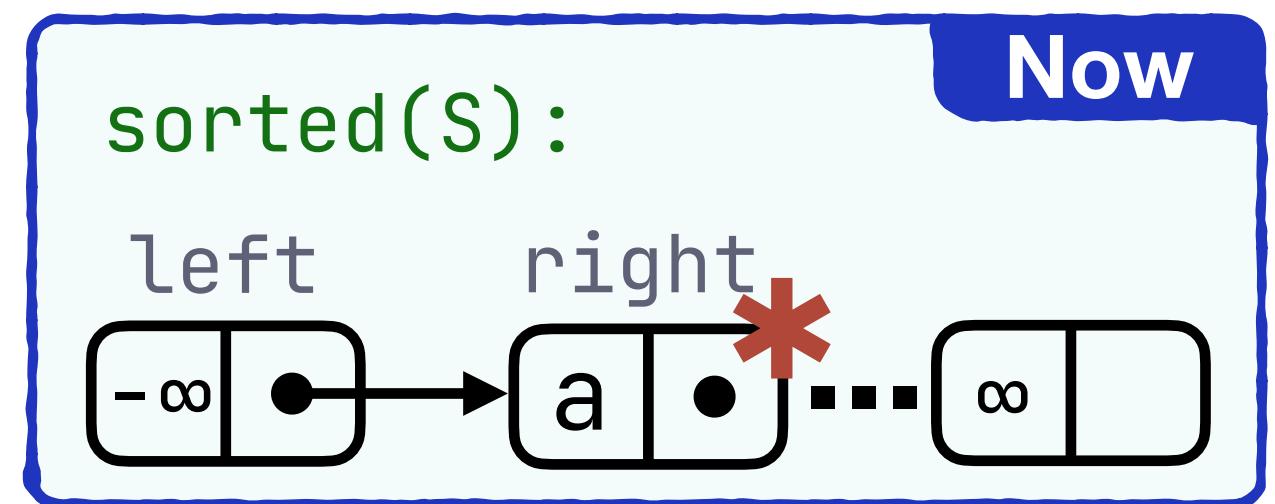
sorted(X):



Chunk

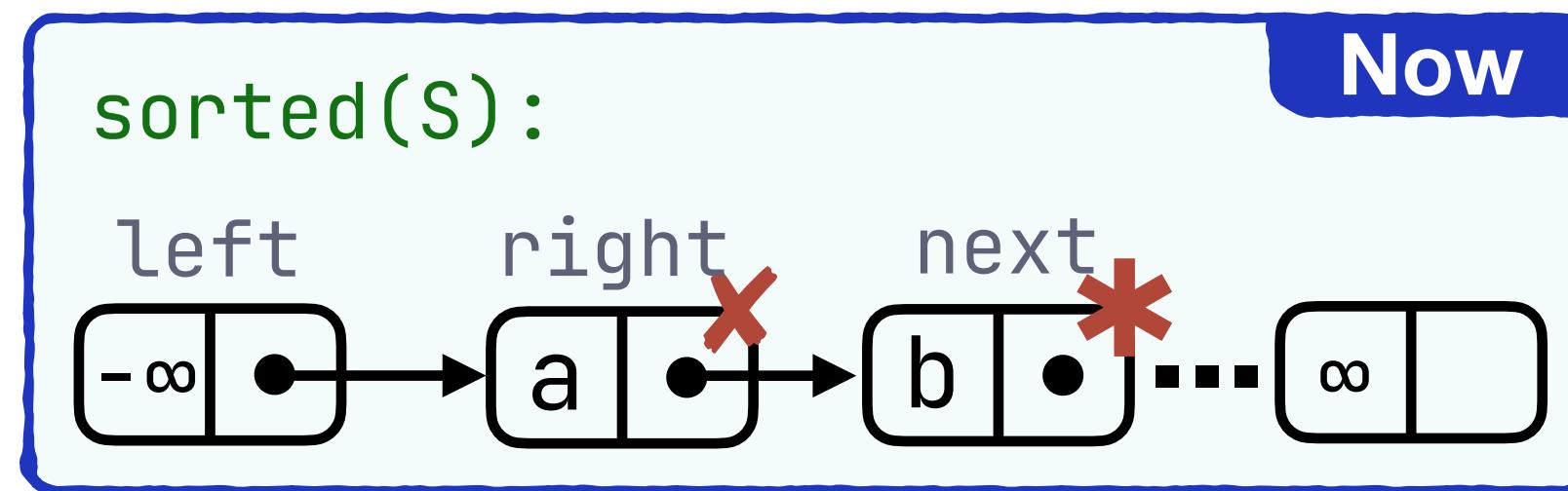
Ghost Update Chunks

// traversal step 1



assume(right→mark);

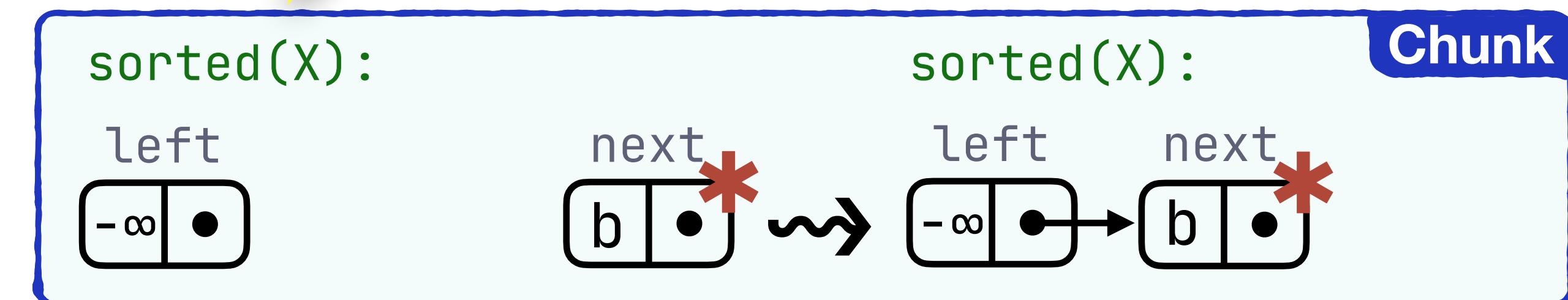
next := right→next;



right := next;

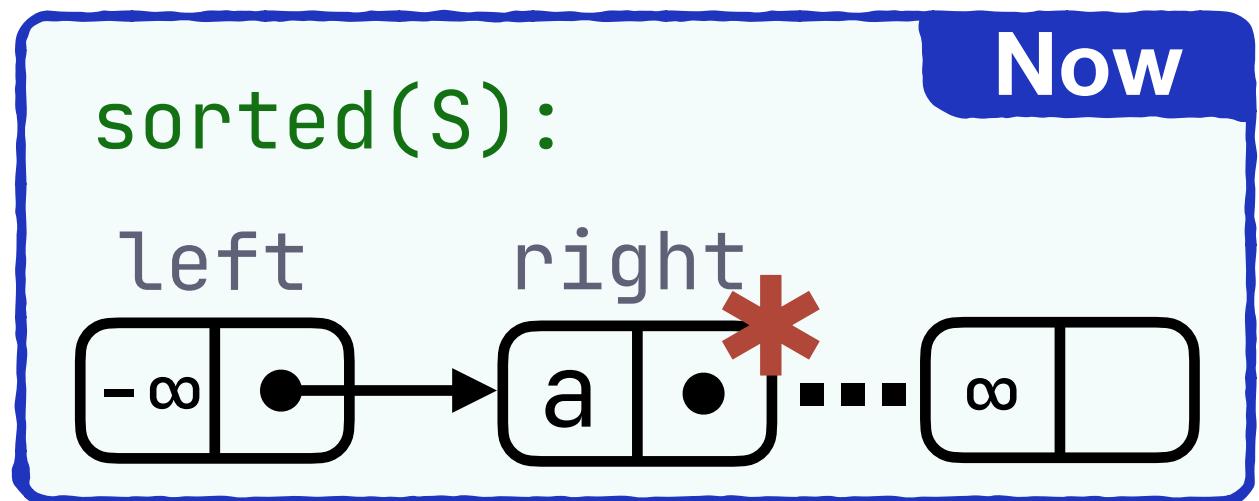
Chunk := Hoare Tripel assuming Now

*



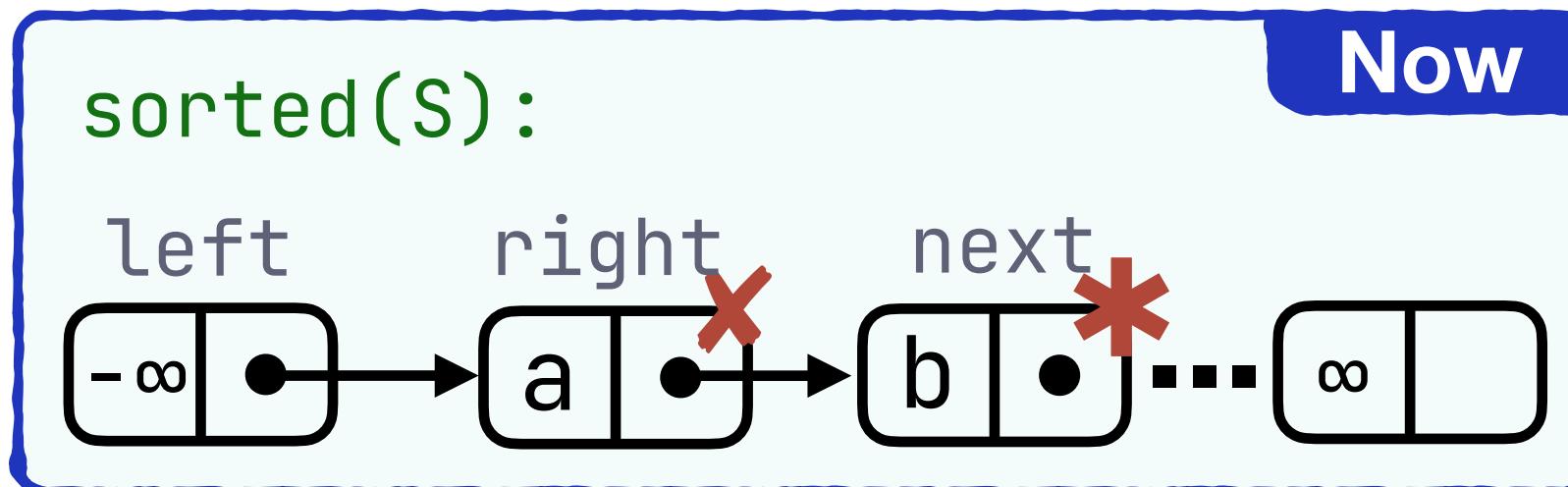
Ghost Update Chunks

```
// traversal step 1
```

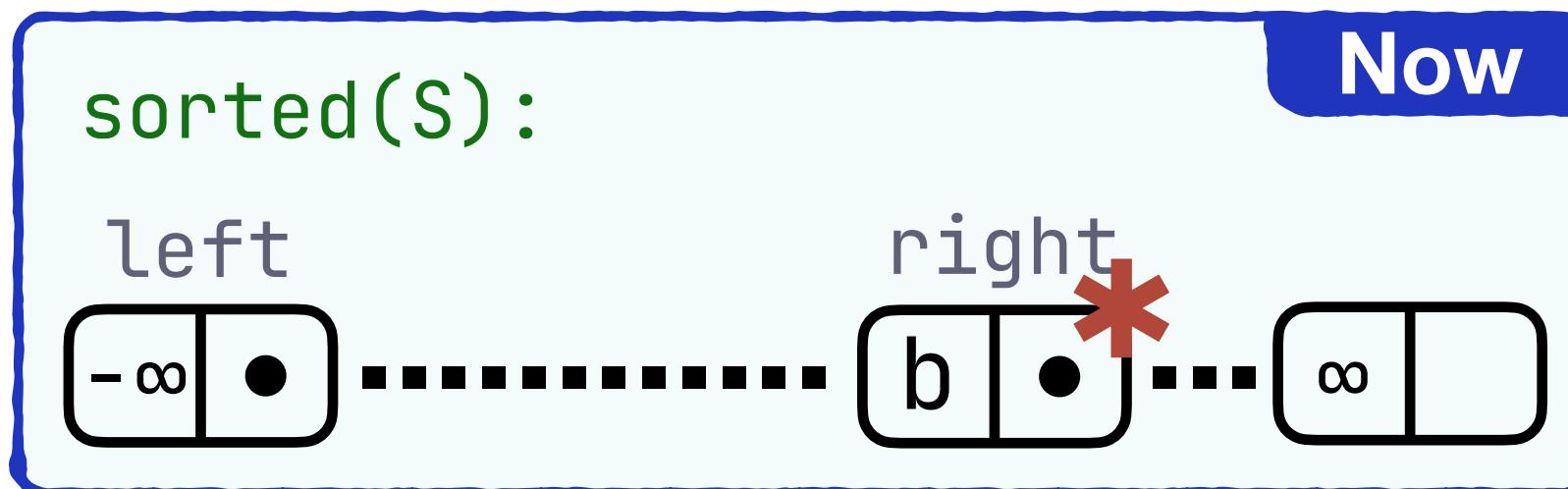


```
assume(right→mark);
```

```
next := right→next;
```

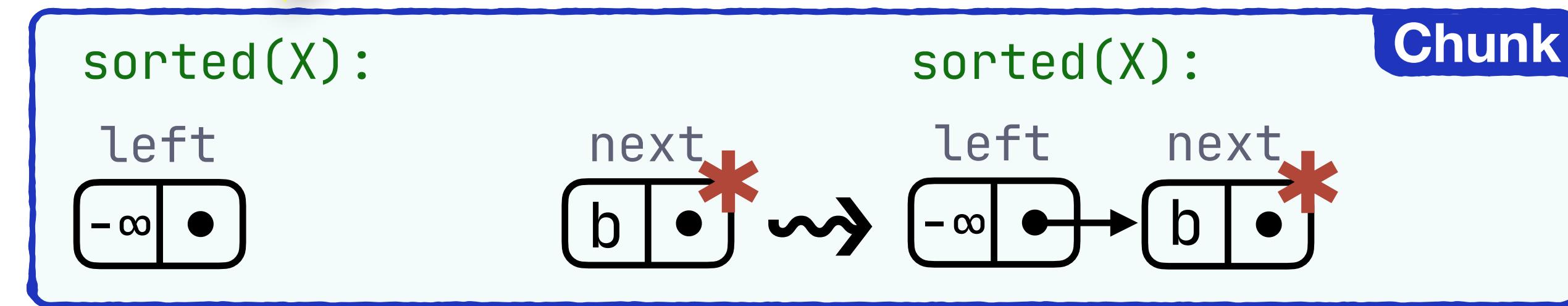


```
right := next;
```

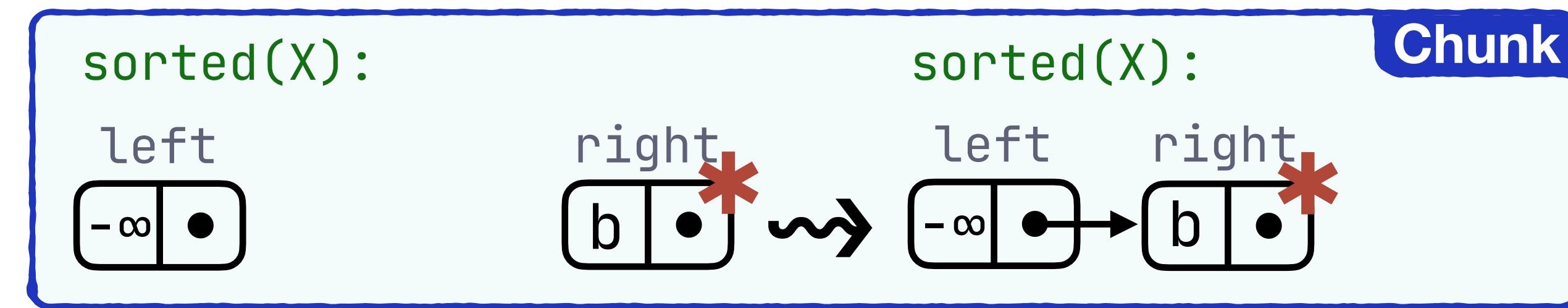


Chunk := Hoare Tripel assuming Now

*

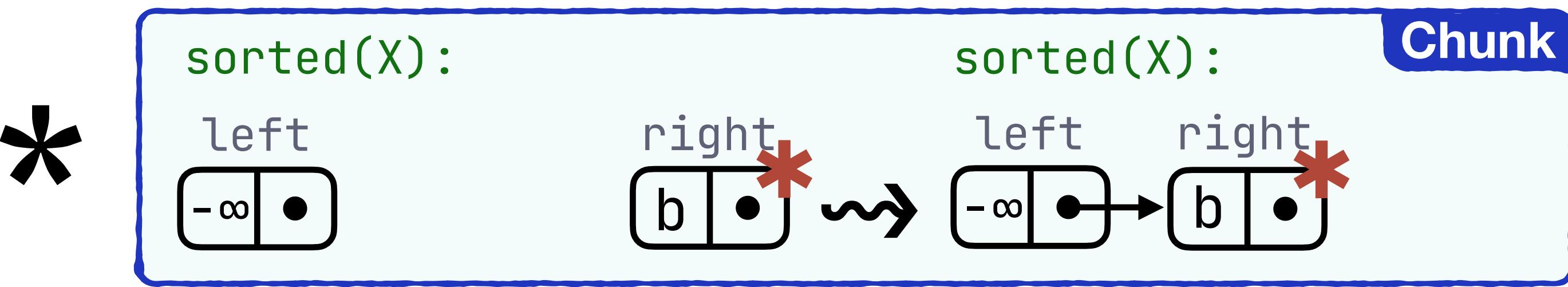
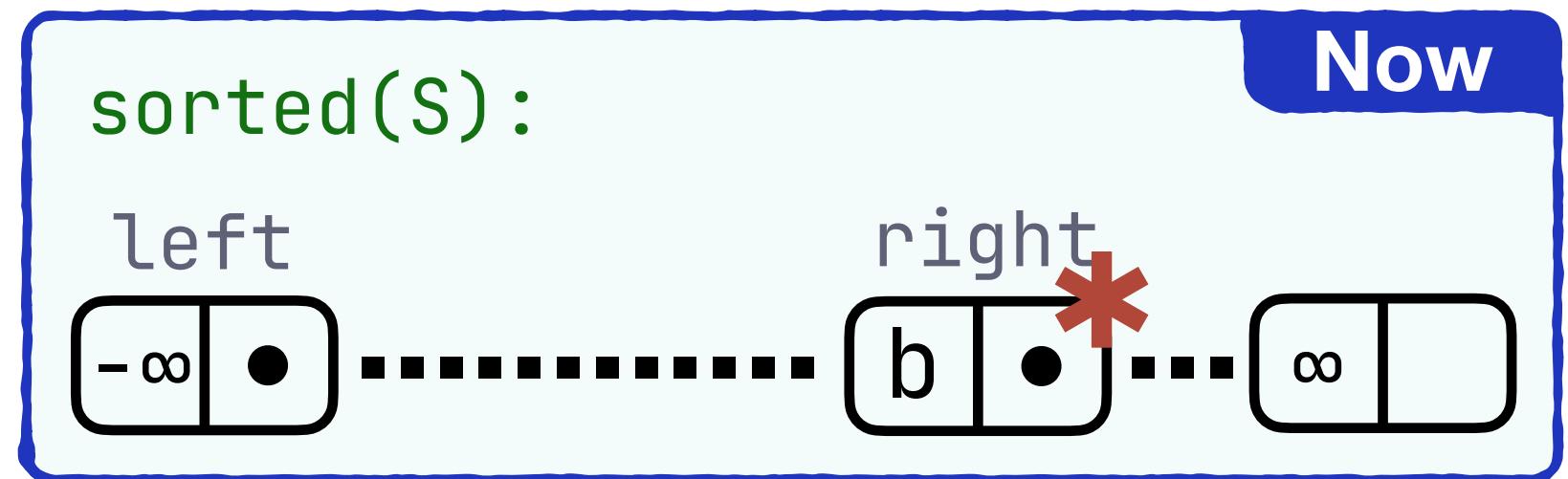


*



Ghost Update Chunks

// traversal step 1 2



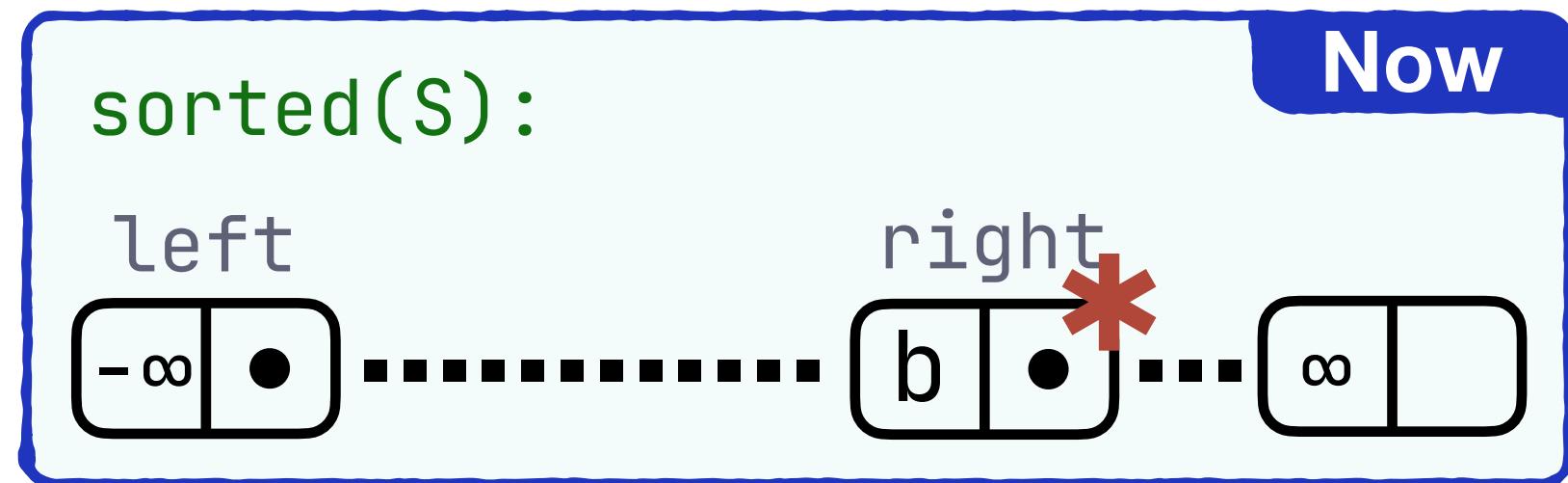
assume(right→mark);

next := right→next;

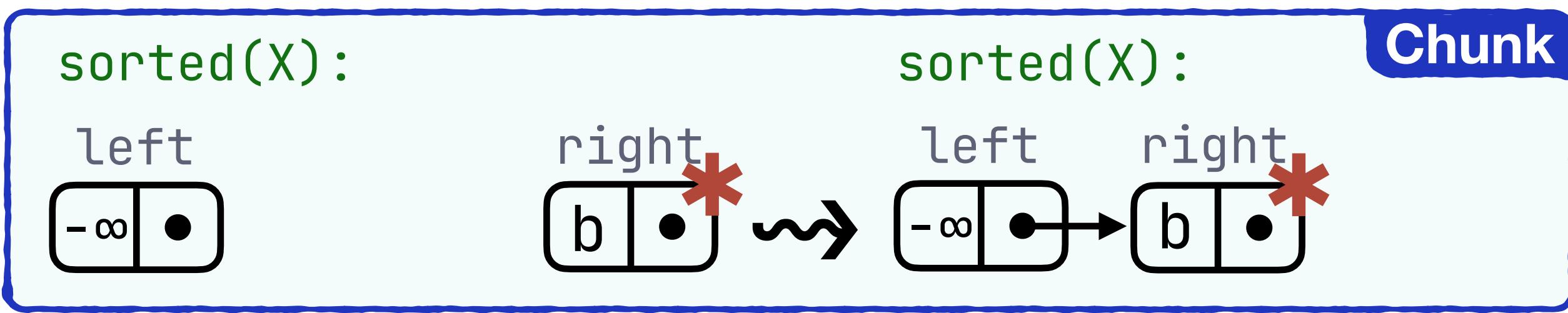
right := next;

Ghost Update Chunks

// traversal step 1 2

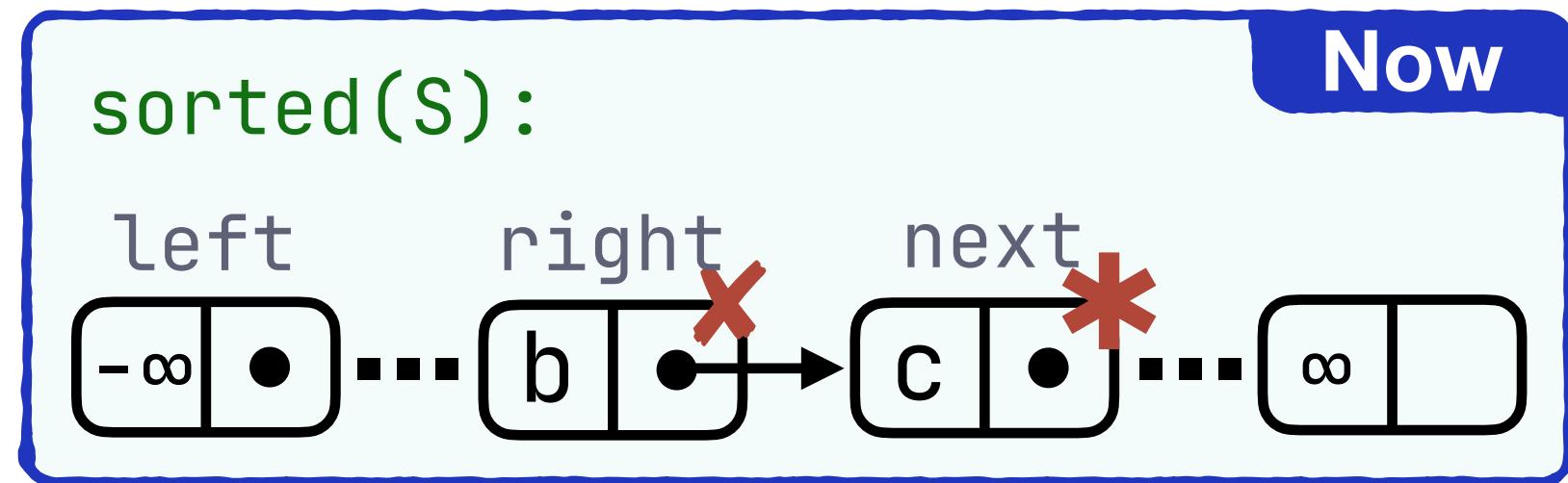


*

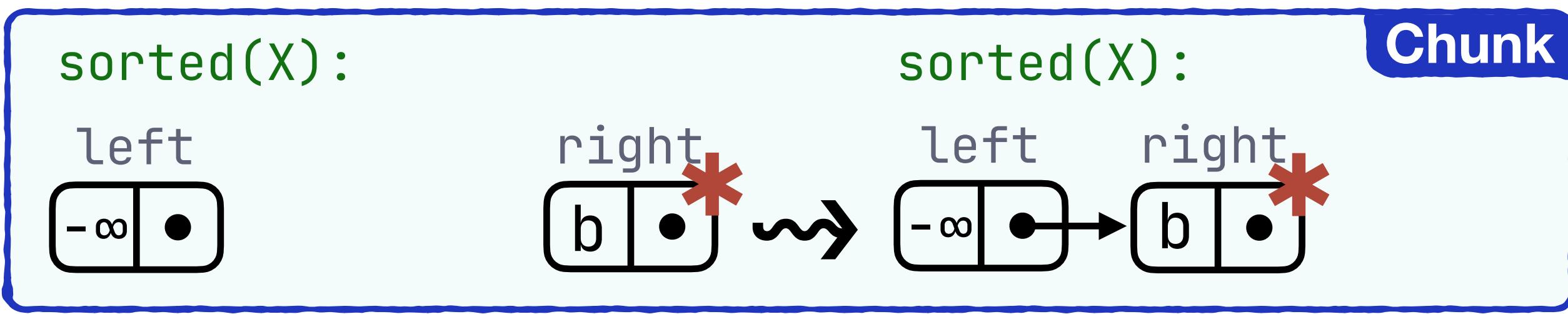


assume(right→mark);

next := right→next;



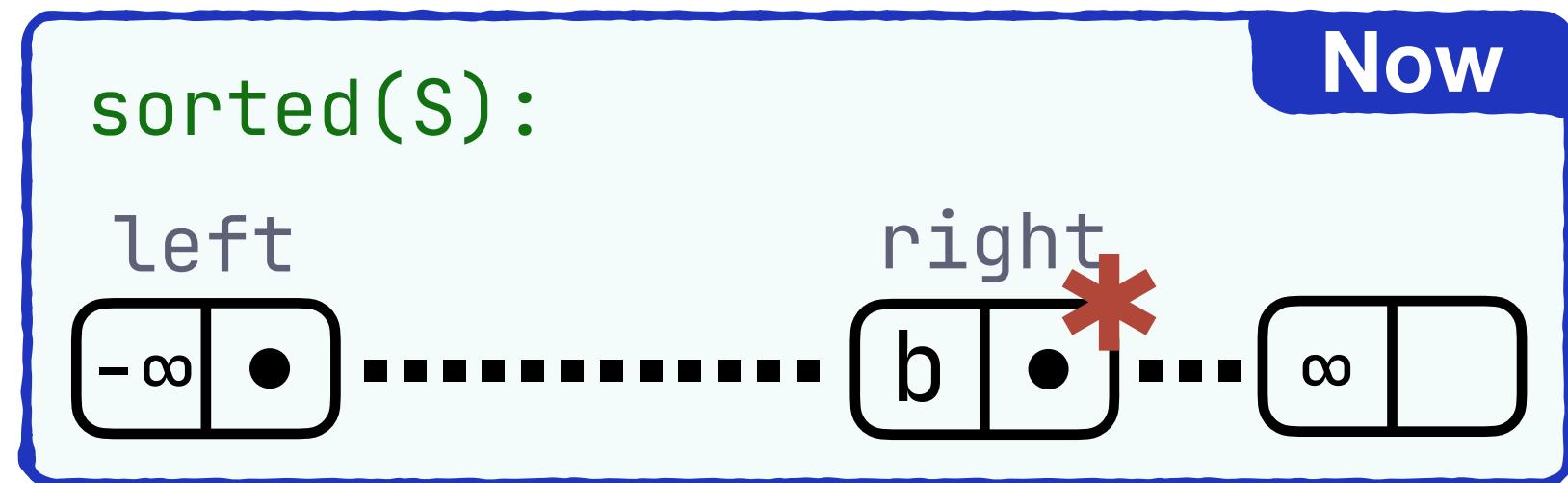
*



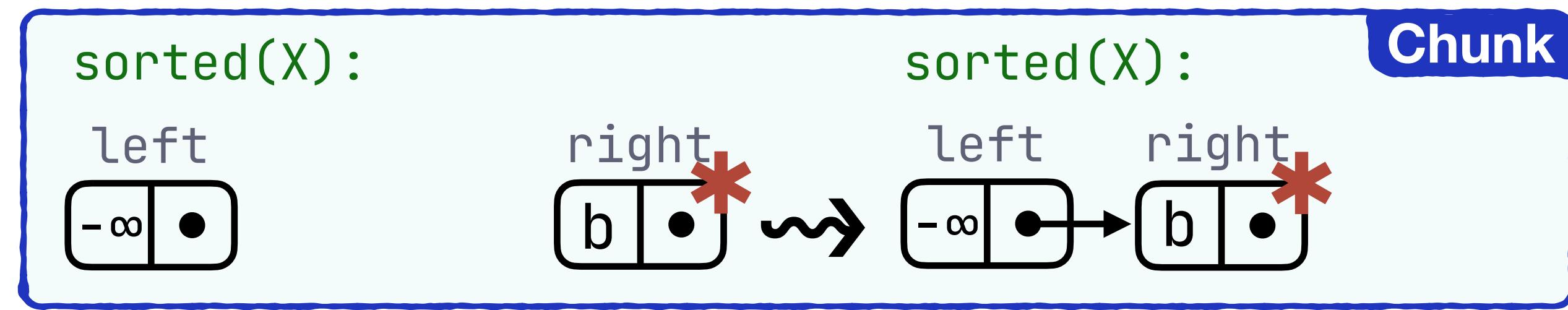
right := next;

Ghost Update Chunks

// traversal step 1 2

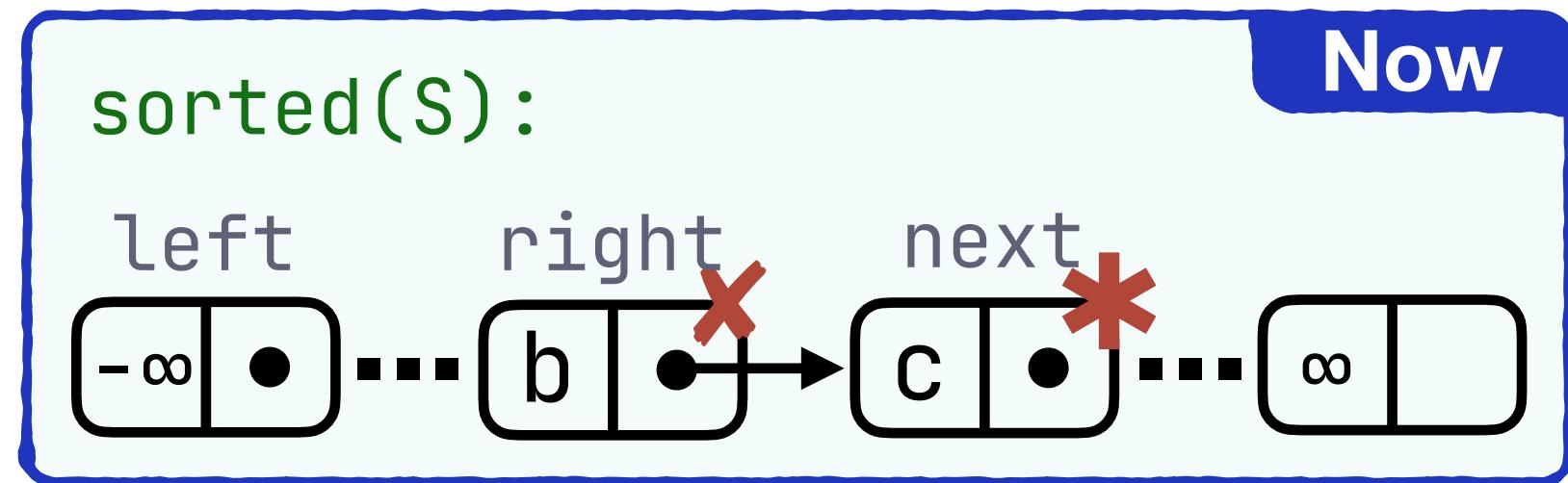


*

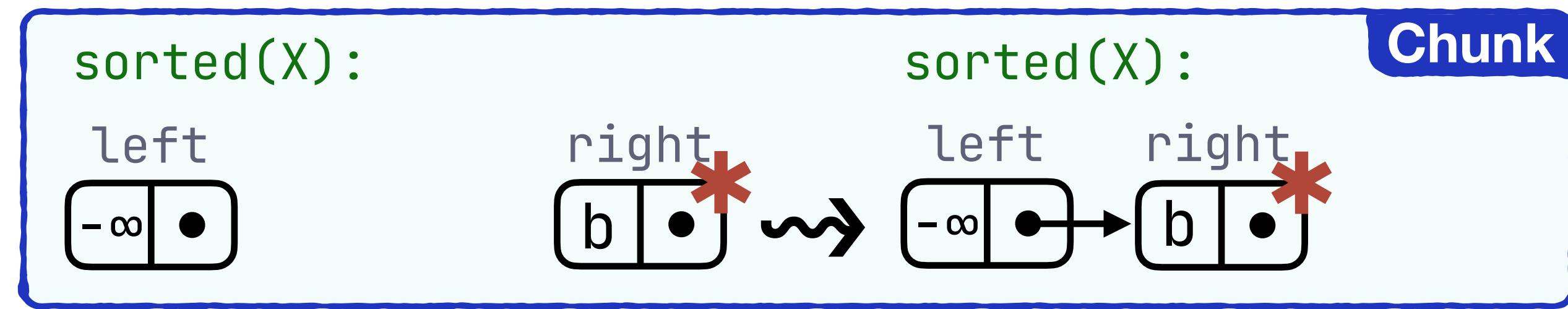


assume(right→mark);

next := right→next;

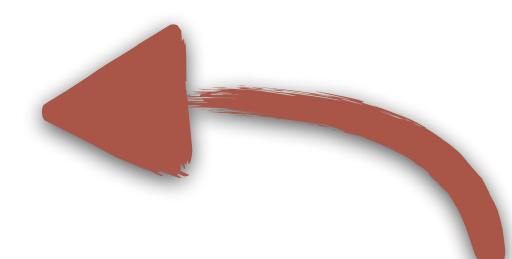
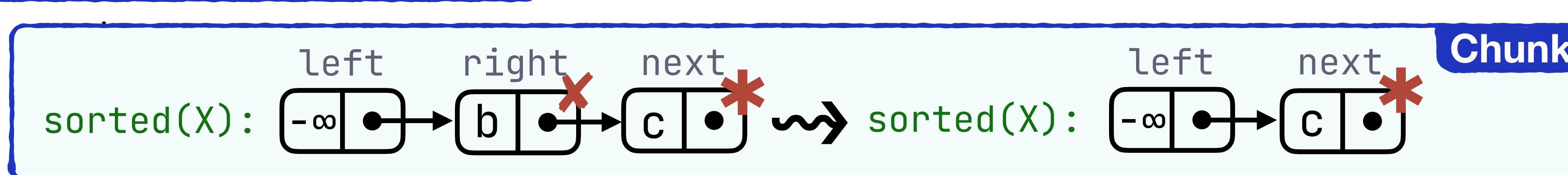


*



right :=

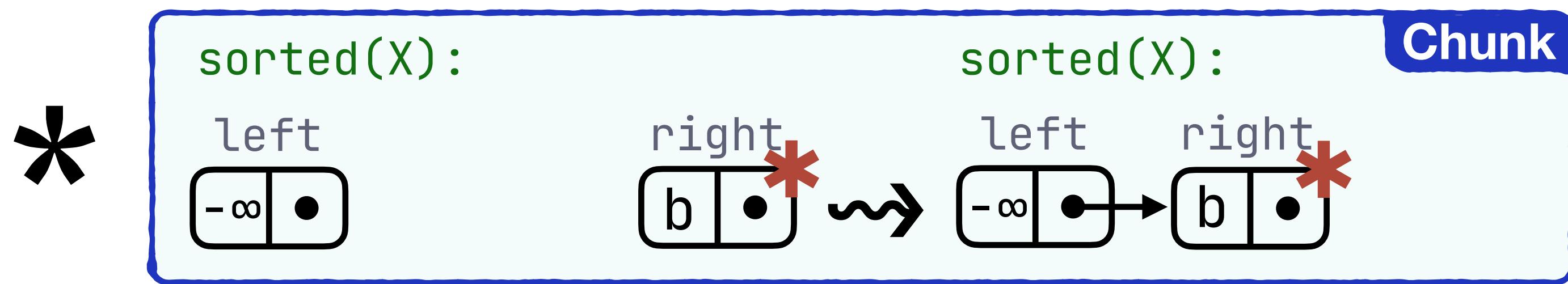
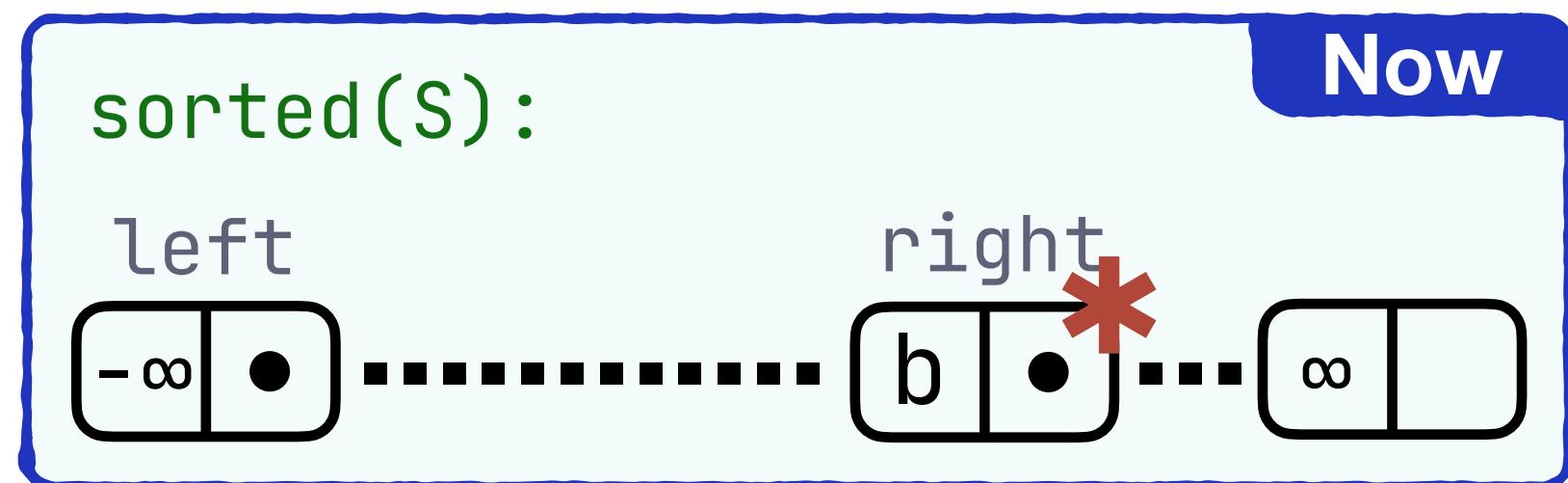
*



INTRO

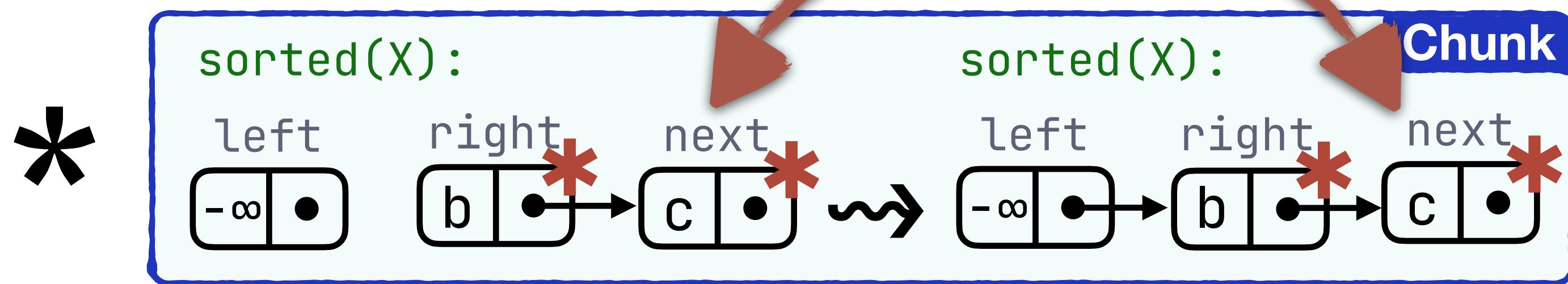
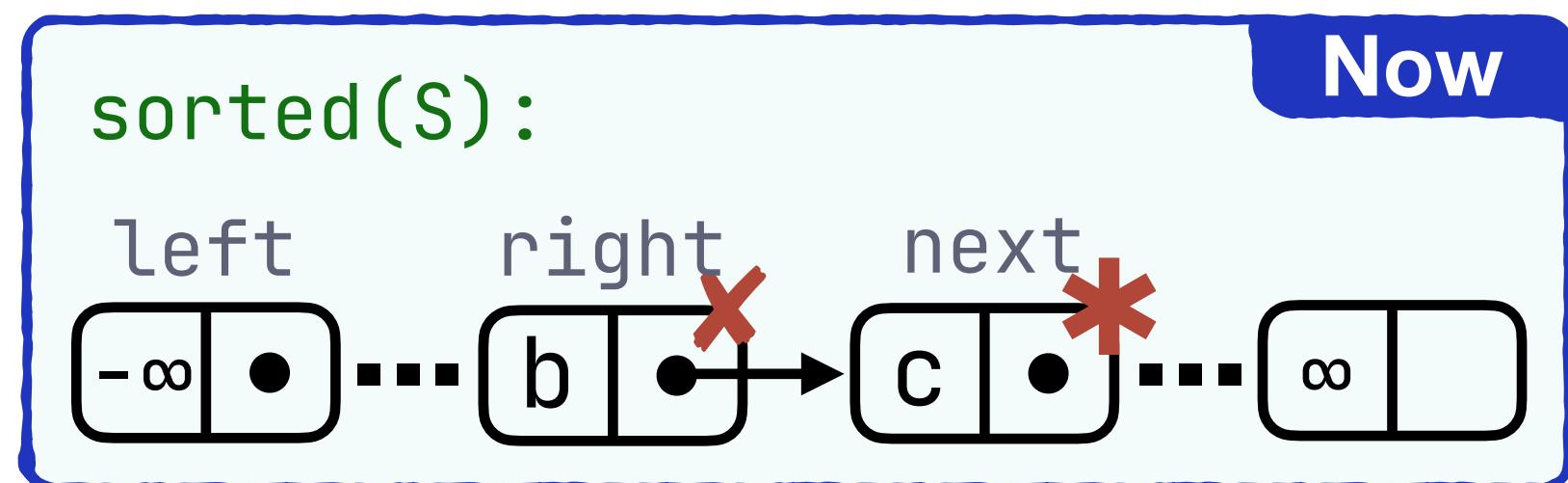
Ghost Update Chunks

// traversal step 1 2

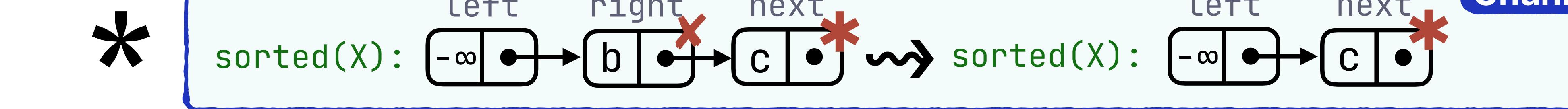
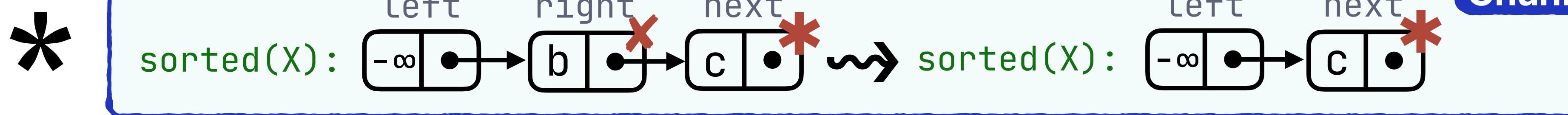


assume(right→mark);

next := right→next;



right :=



FRAME

Chunk

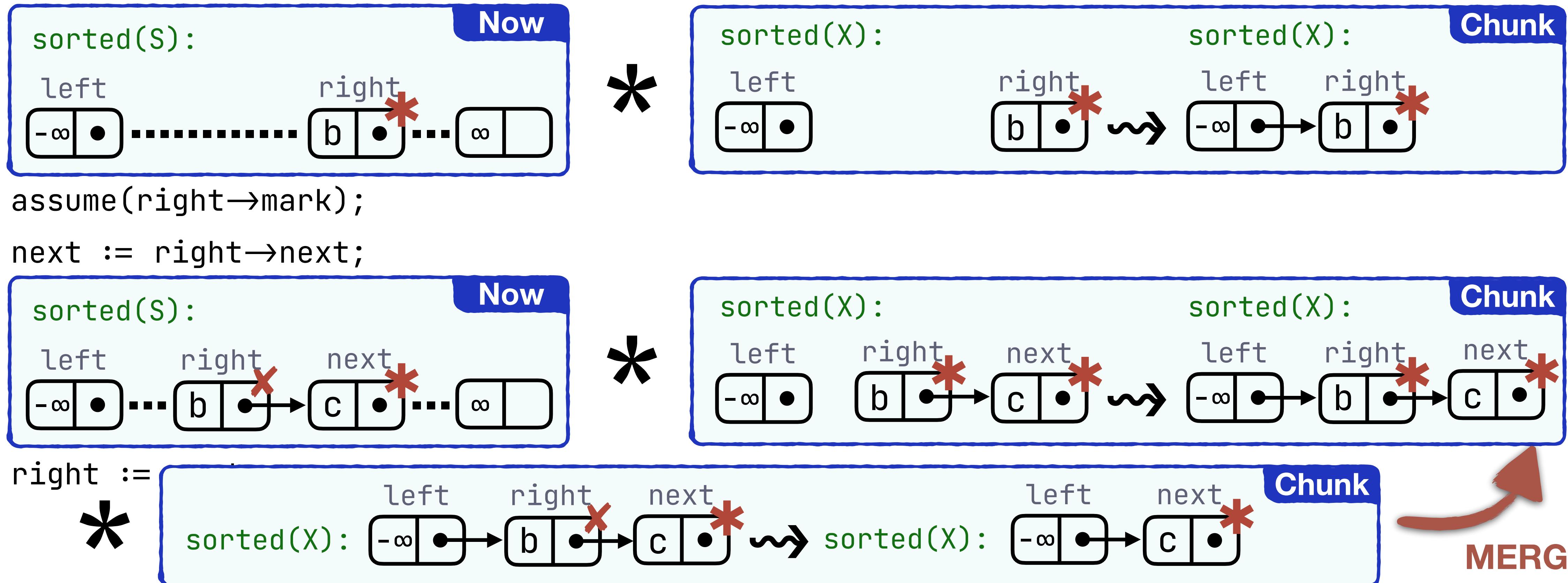
Chunk

Chunk

Chunk

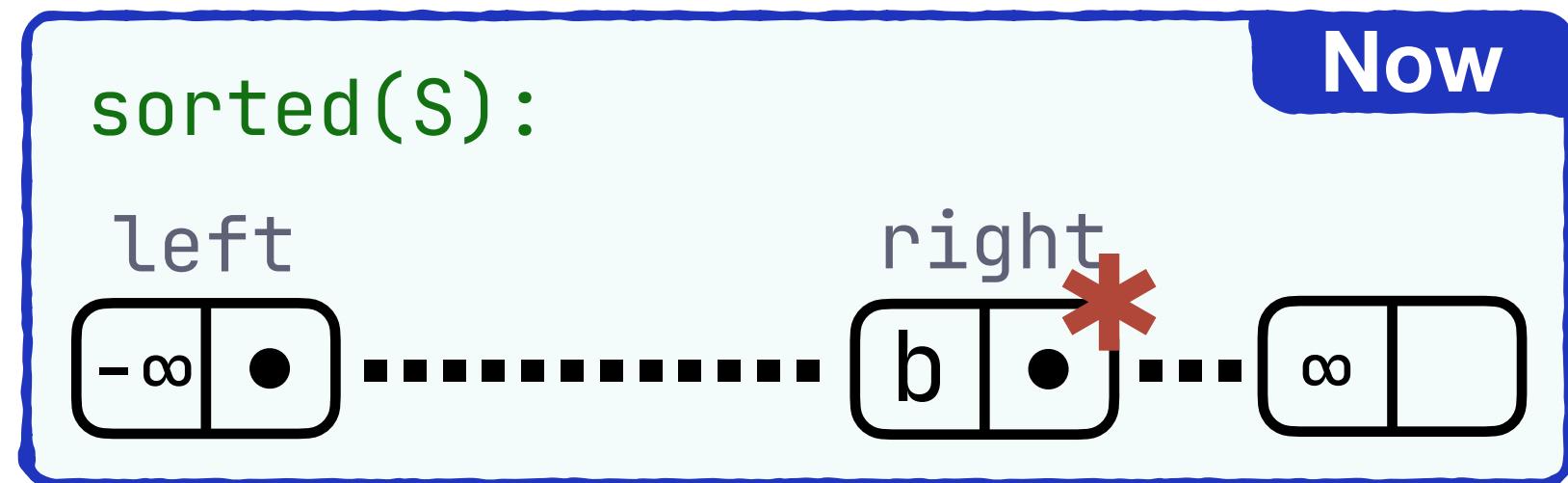
Ghost Update Chunks

// traversal step 1 2

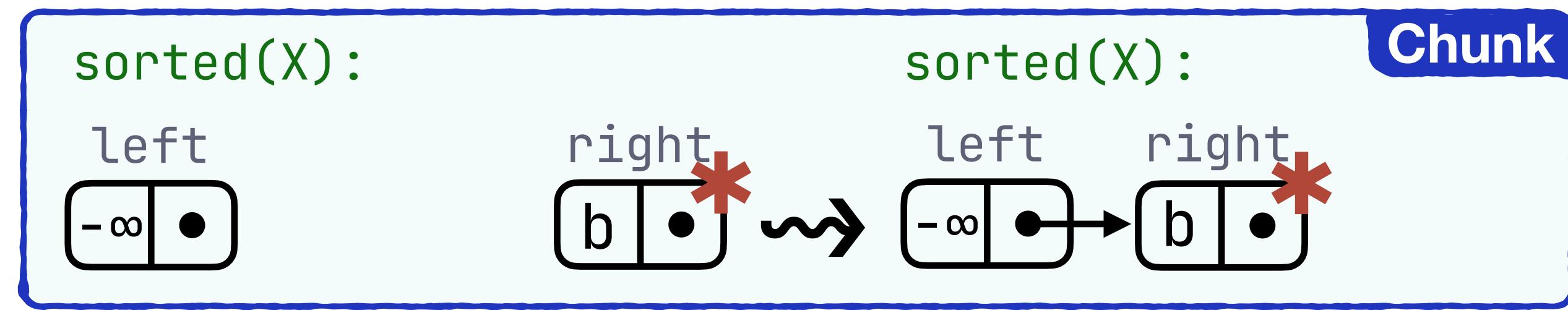


Ghost Update Chunks

// traversal step 1 2

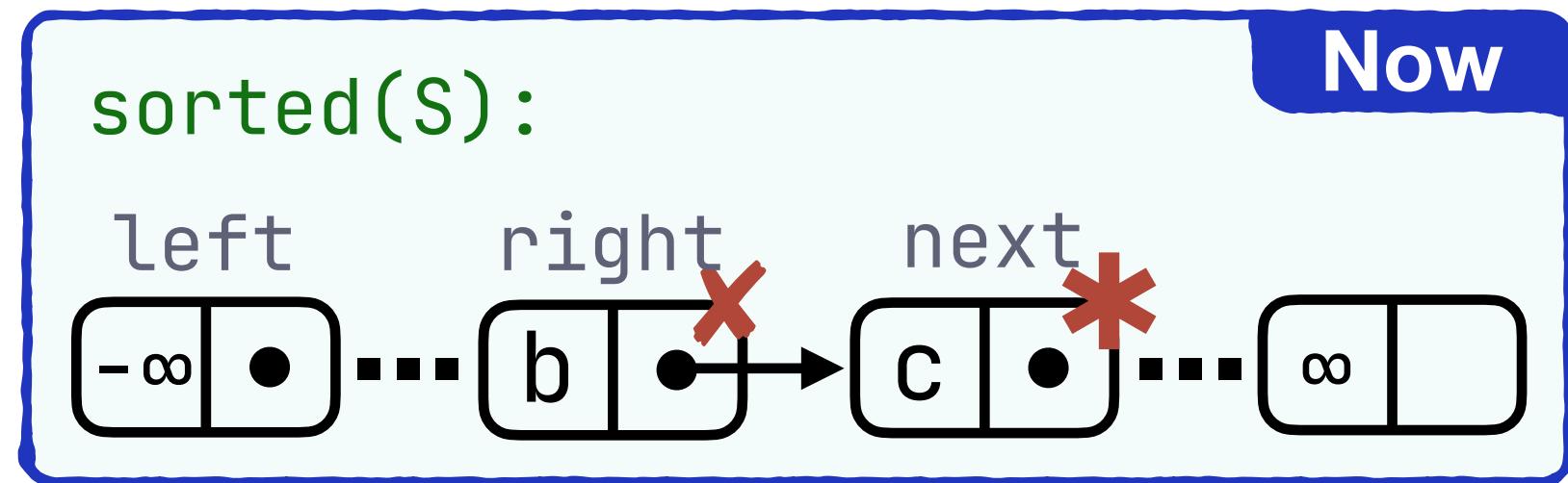


*

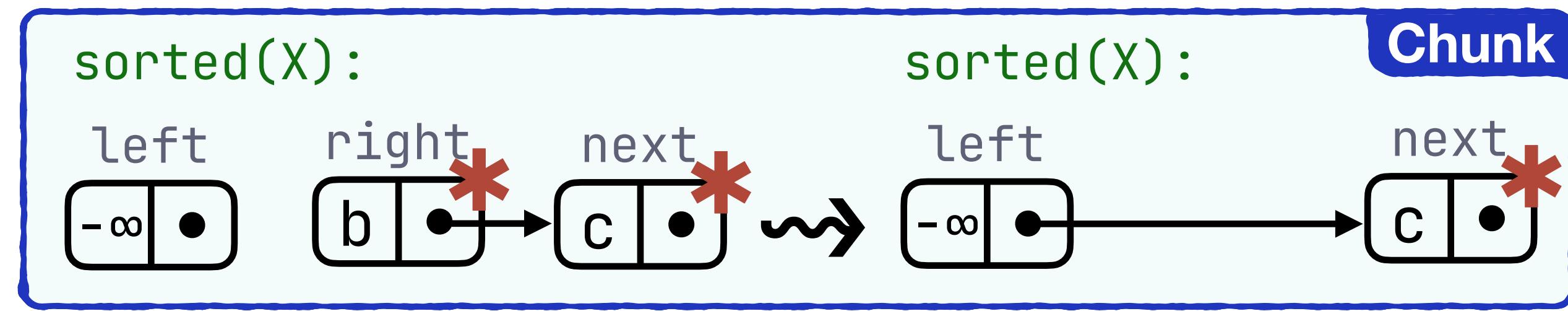


assume(right→mark);

next := right→next;



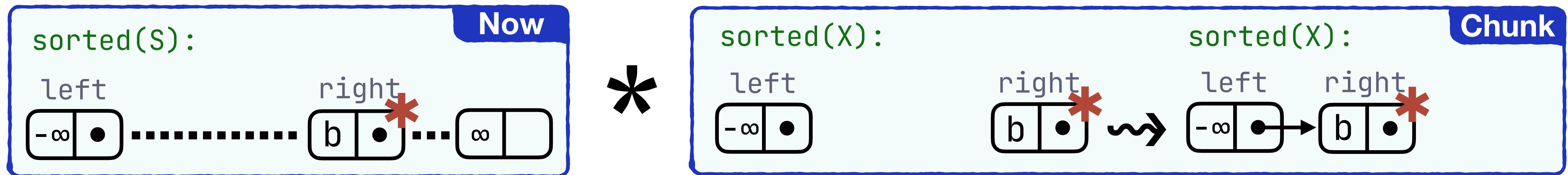
*



right := next;

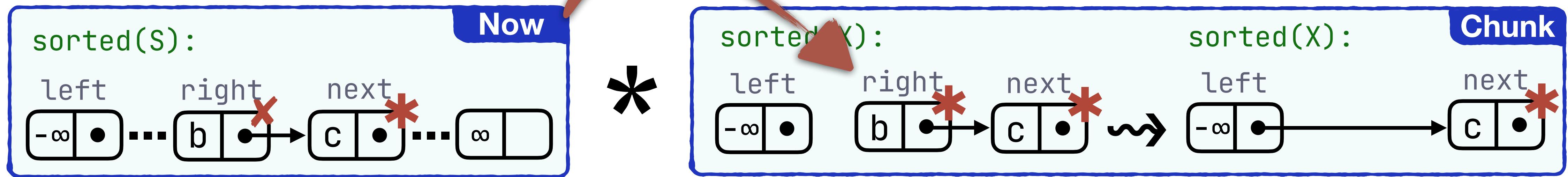
Ghost Update Chunks

// traversal step 1 2



assume(right→mark);

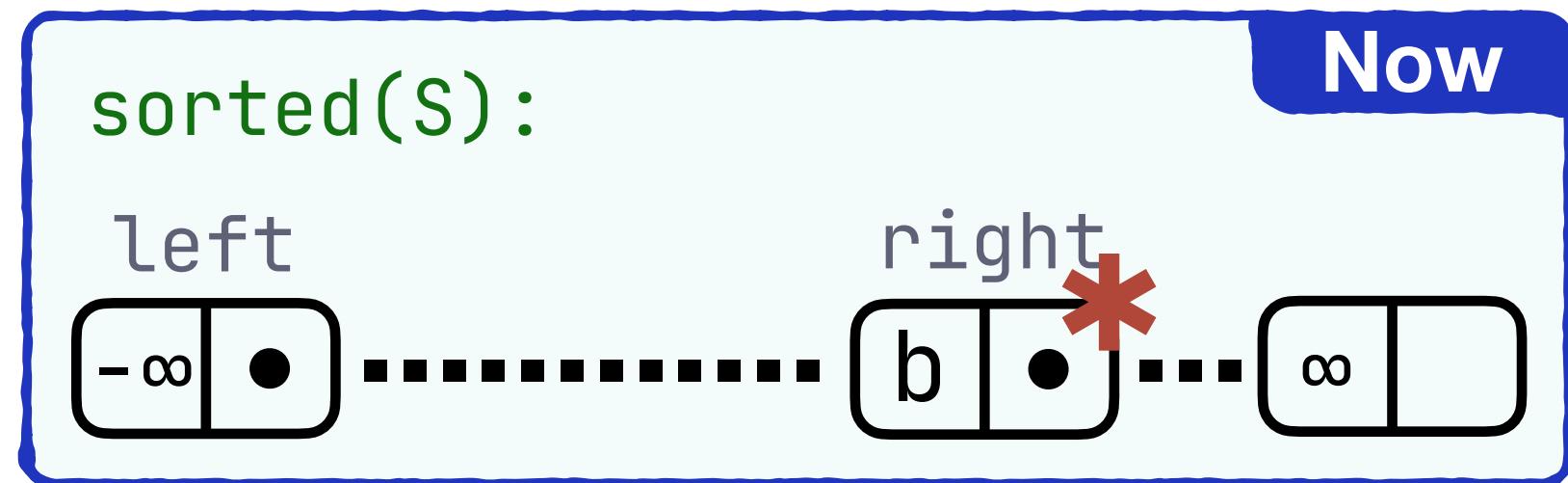
next := right→next;



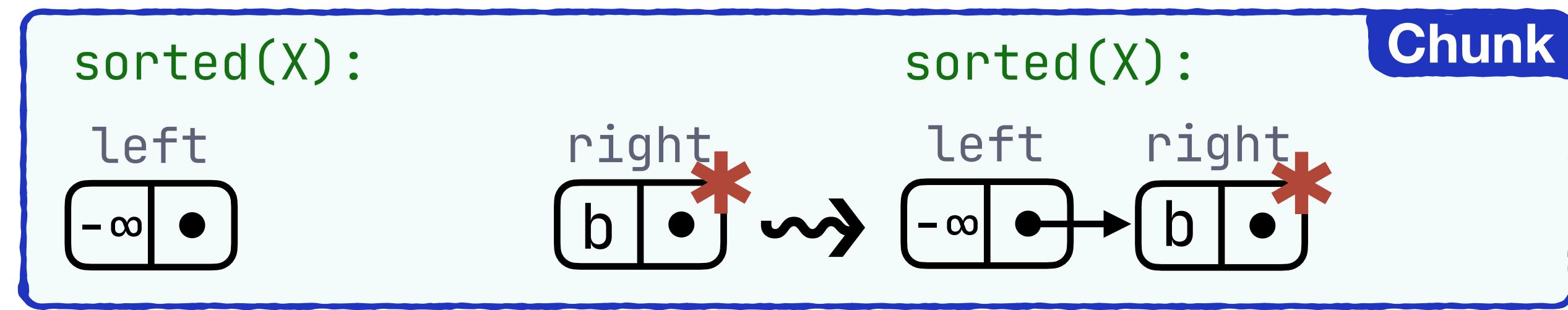
right := next;

Ghost Update Chunks

// traversal step 1 2

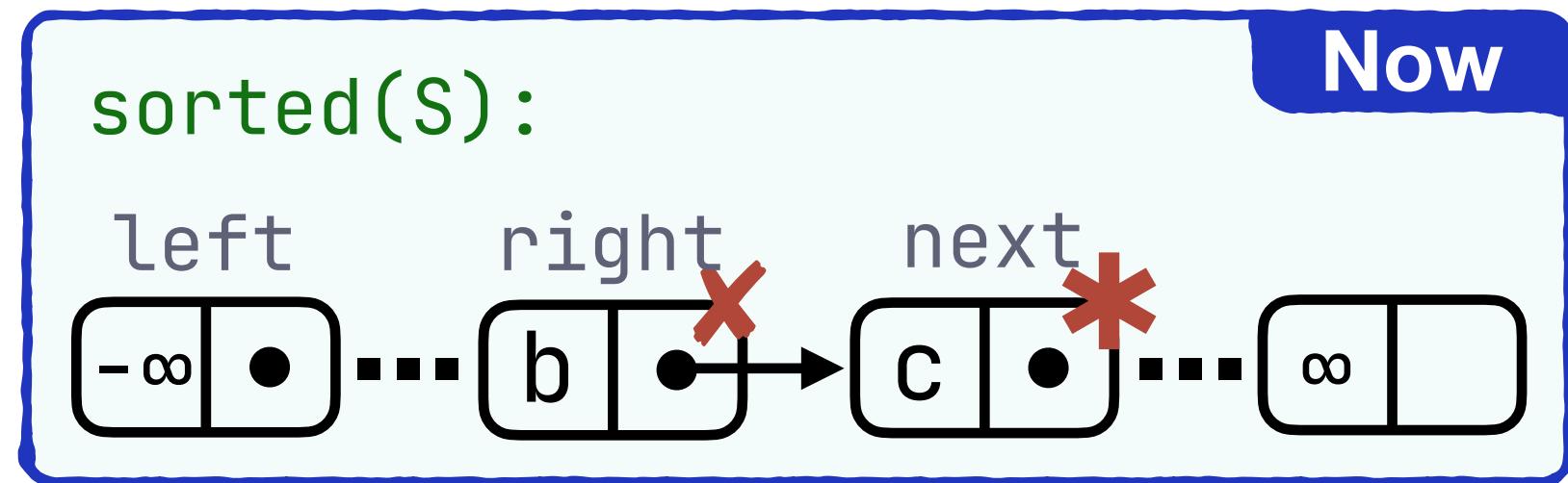


*

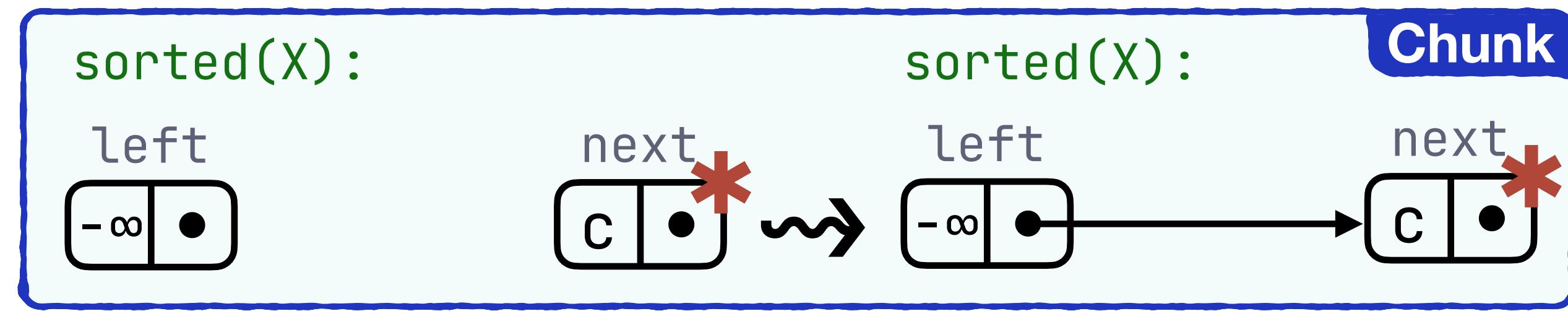


assume(right→mark);

next := right→next;



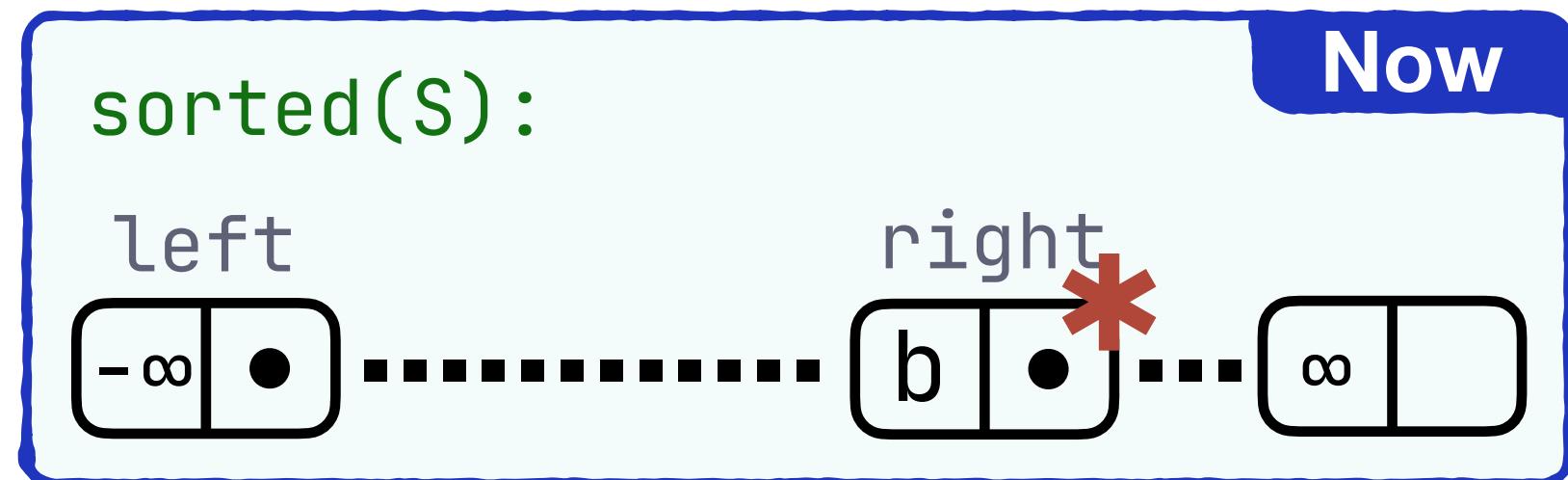
*



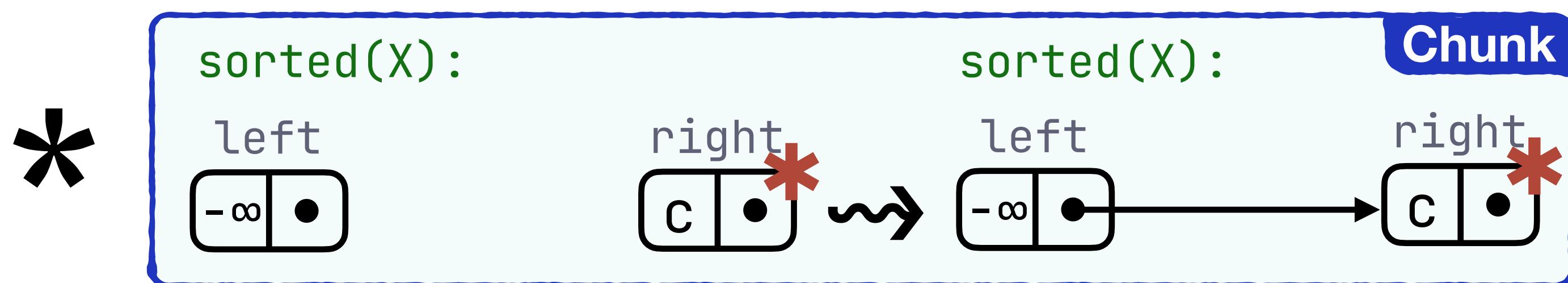
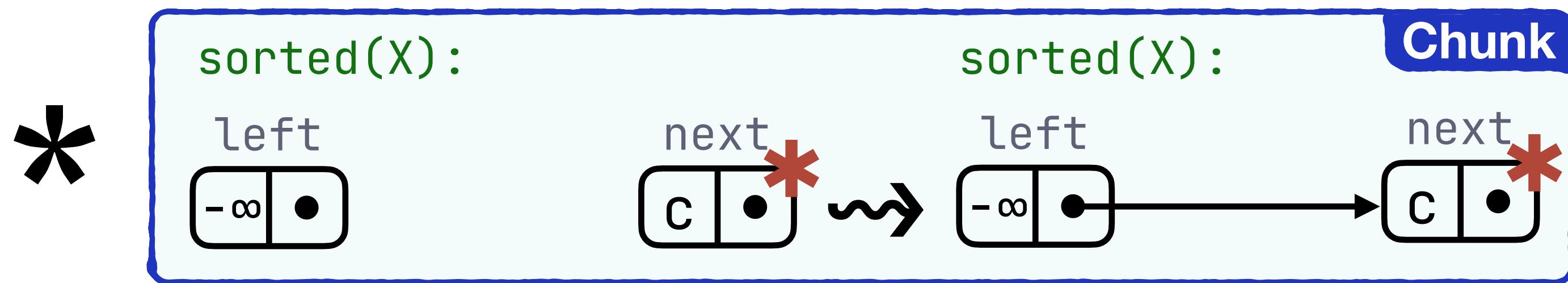
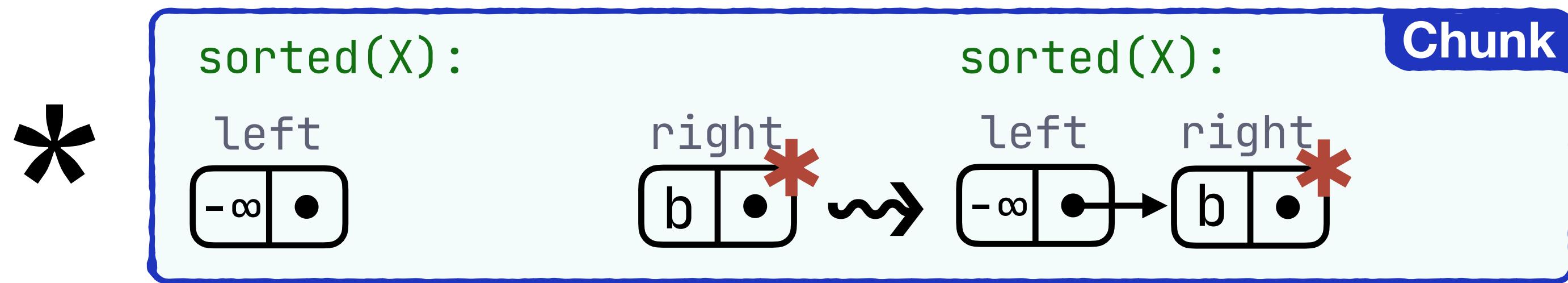
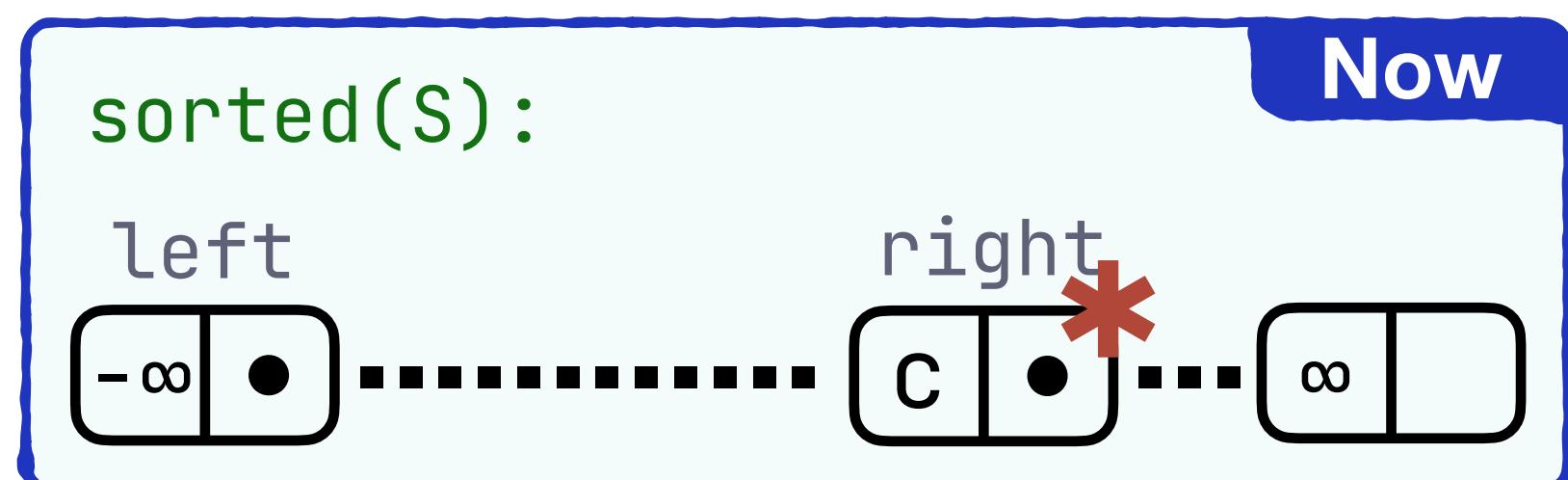
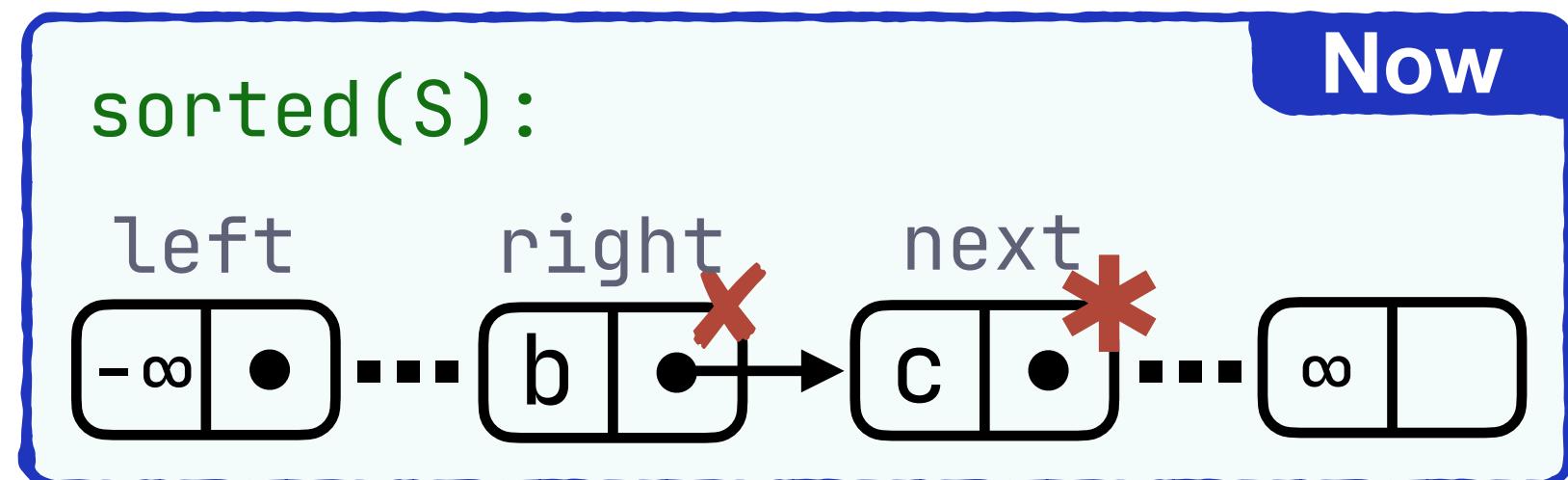
right := next;

Ghost Update Chunks

// traversal step 1 2

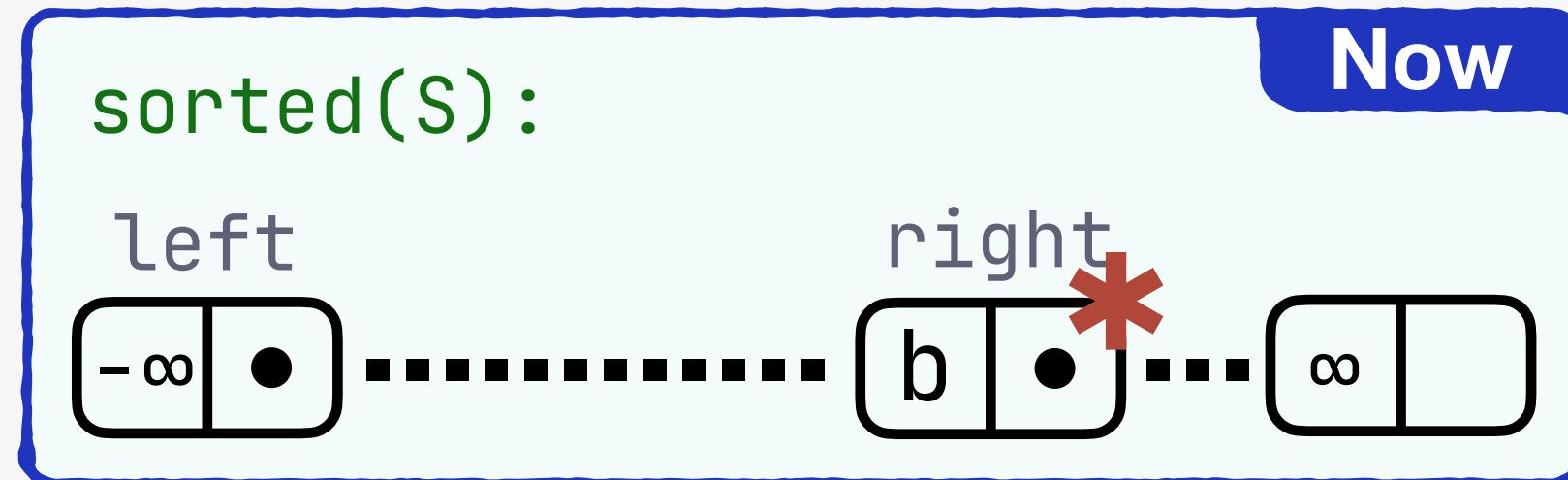


next := right→next;

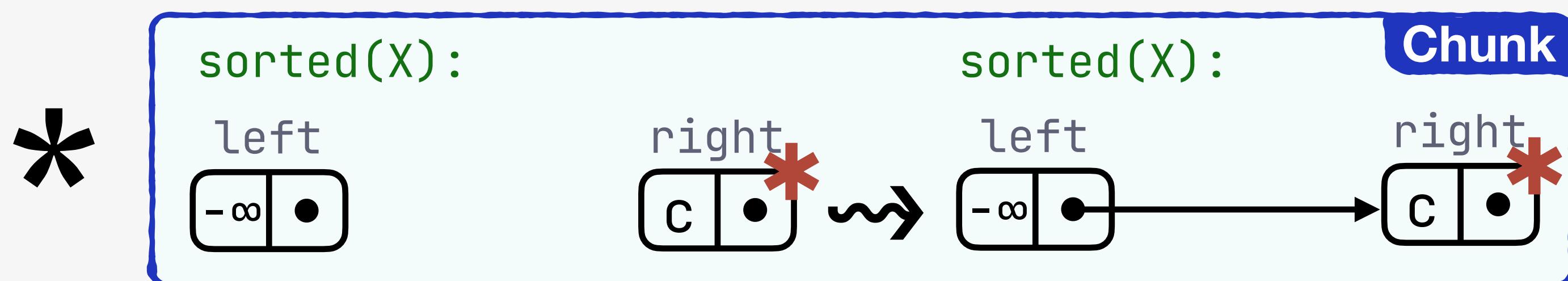
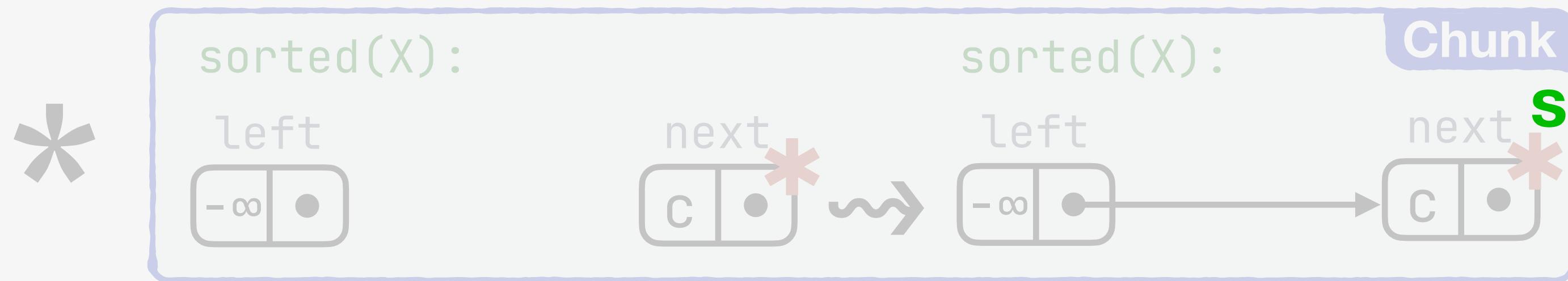
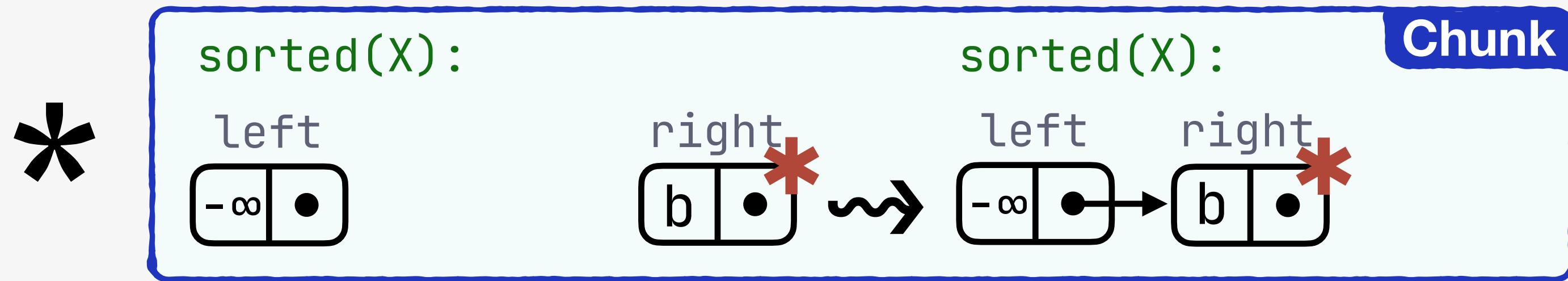
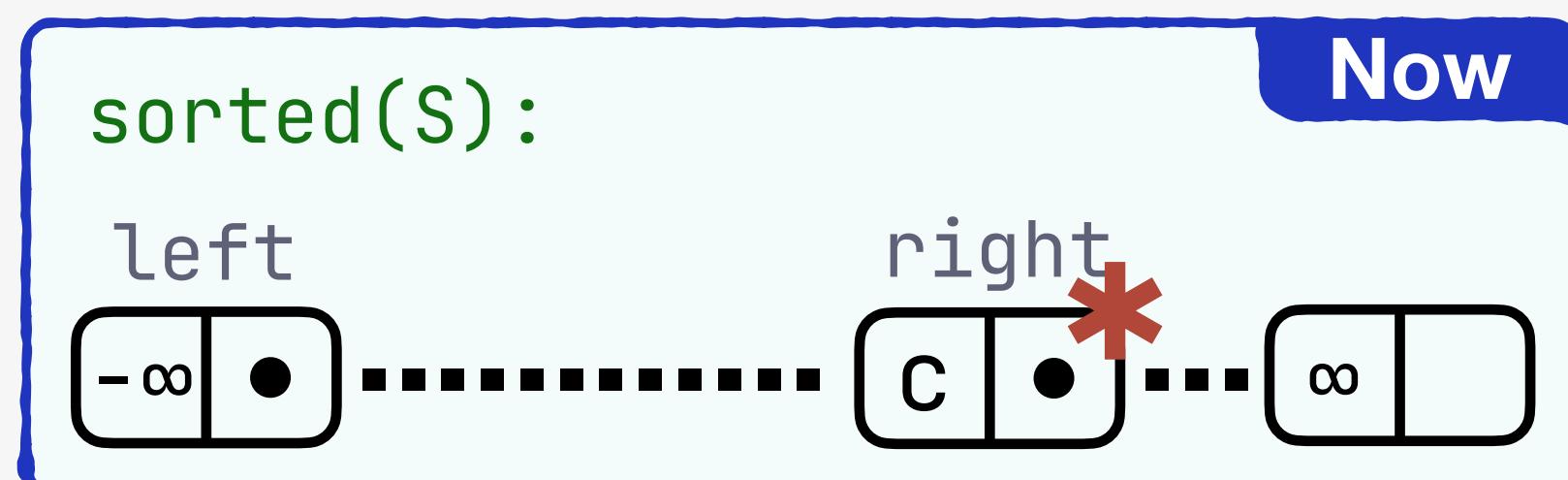
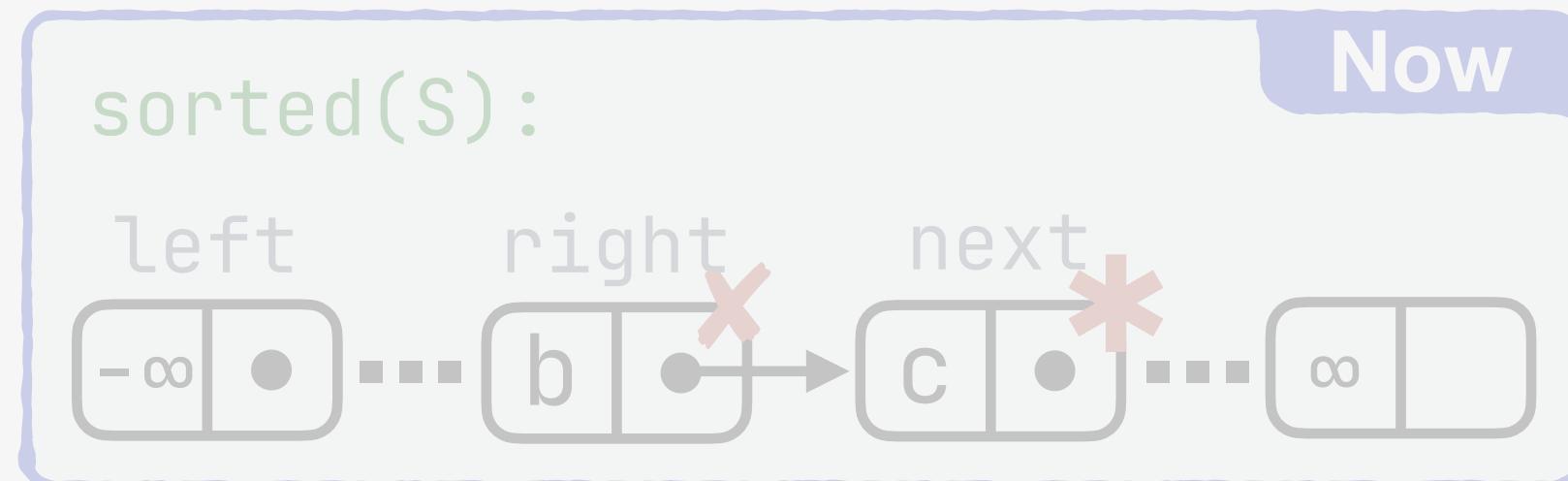


Ghost Update Chunks

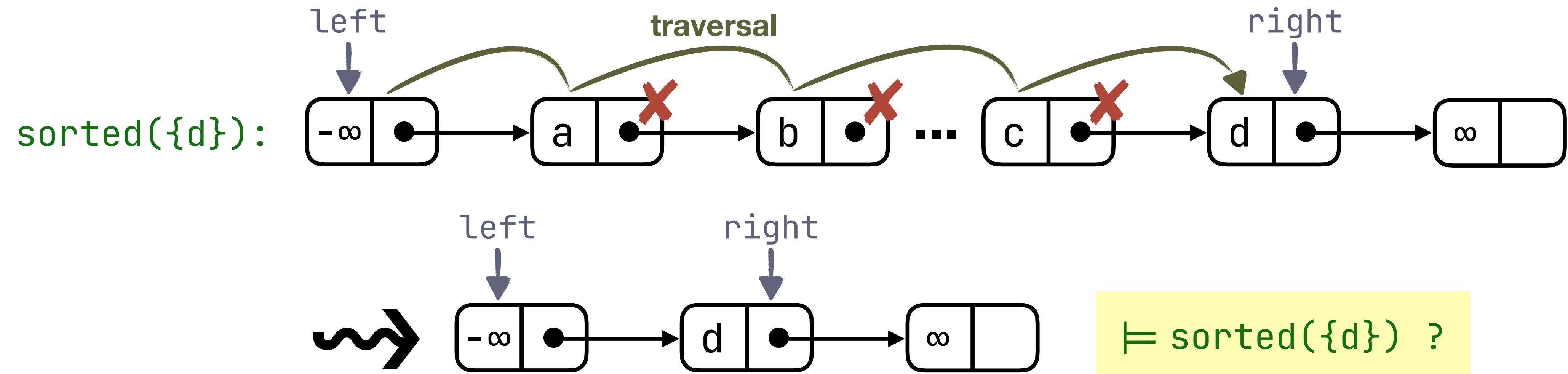
```
// traversal step 1 2 k+1
```



```
next := right→next;
```



Example: Unbounded Removal



Contribution: Ghost Update Chunks

- formalization in abstract SL
- usage patterns: INTRO, MERGE, FRAME, DISCHARGE
- implementation

Verification Challenge 1: Complex Unbounded Updates

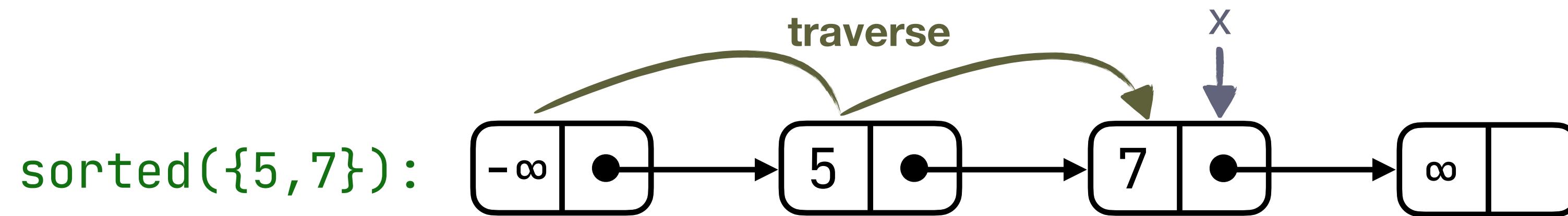
Verification Challenge 2: **Linearizability**

Example: contains(7)

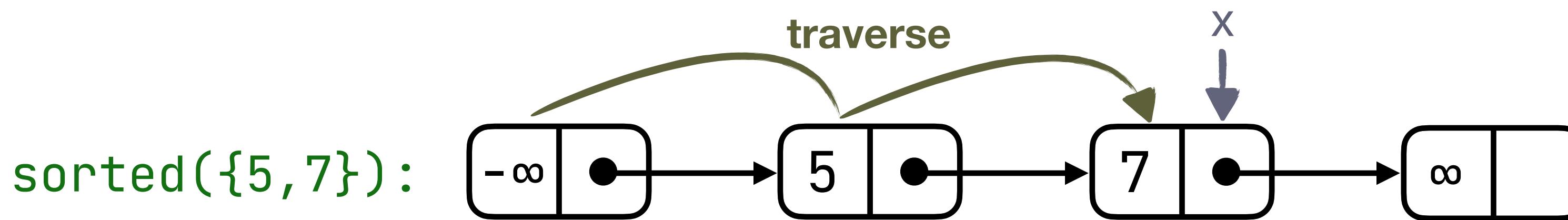
sorted({5, 7}):

The diagram shows a linked list with four nodes. Each node is a rectangle divided into two horizontal sections by a vertical line. The left section contains either $-\infty$, the value 5, or the value 7. The right section is empty. A black dot representing a pointer is positioned at the boundary between the two sections of each node. Arrows connect these pointers from one node to the next, forming a chain: $-\infty \rightarrow 5 \rightarrow 7 \rightarrow \infty$.

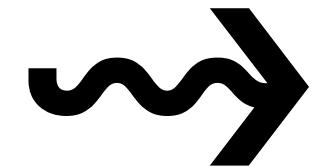
Example: contains(7)



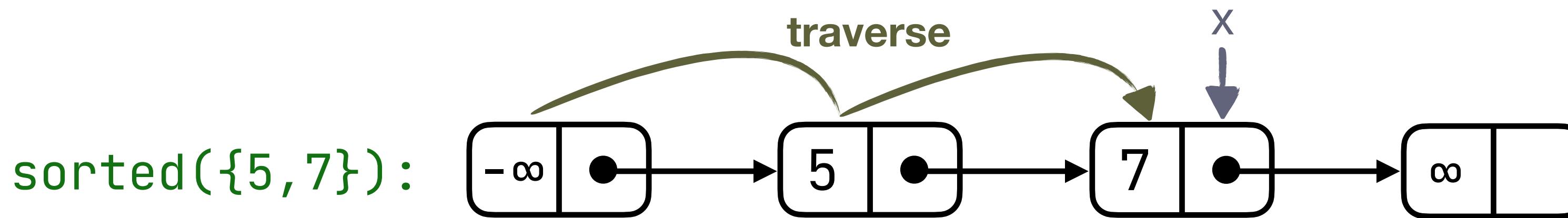
Example: contains(7)



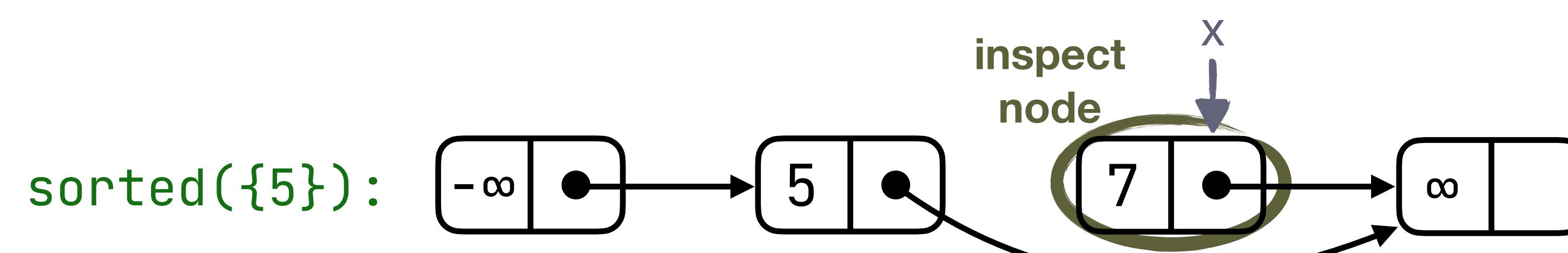
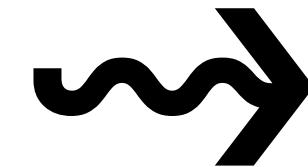
interference



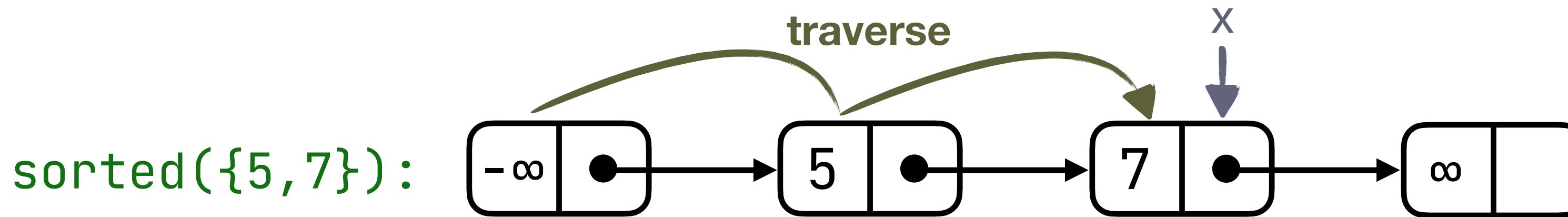
Example: contains(7)



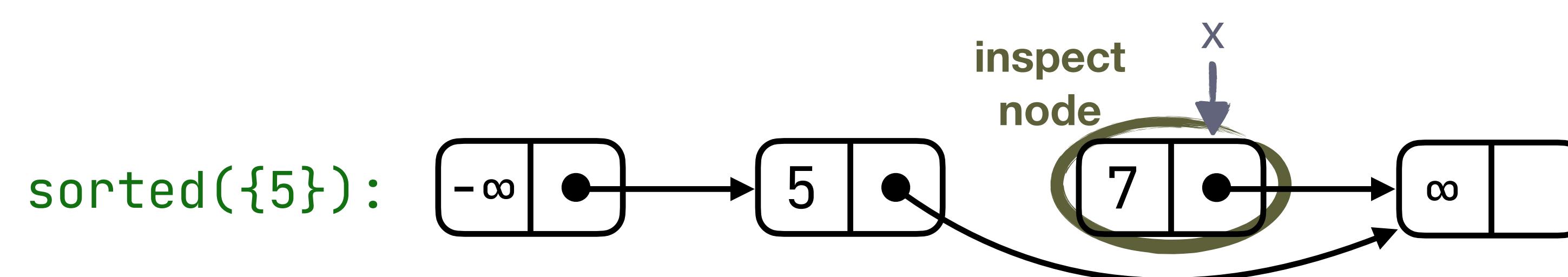
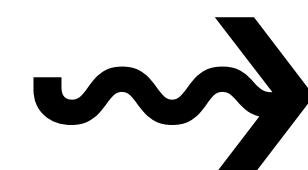
interference



Example: contains(7)

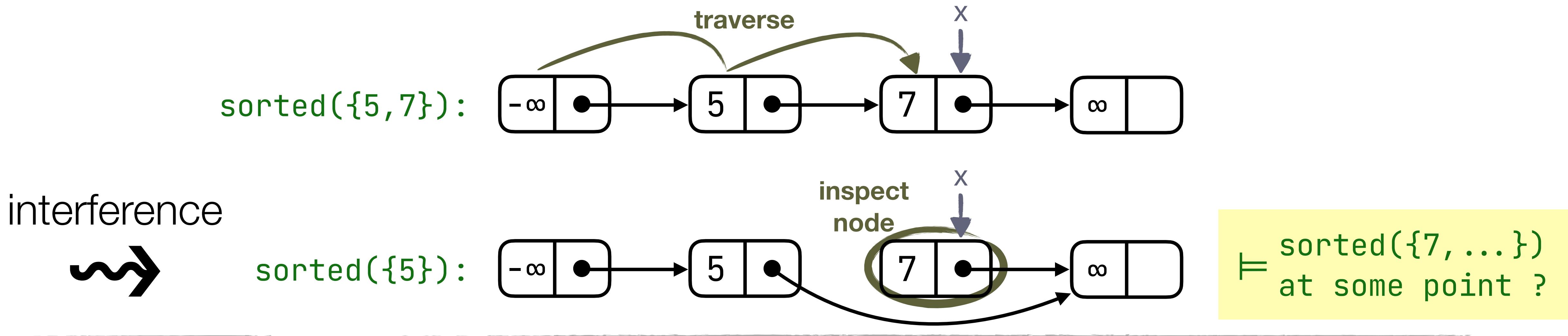


interference



\models sorted({7, ...})
at some point ?

Example: contains(7)

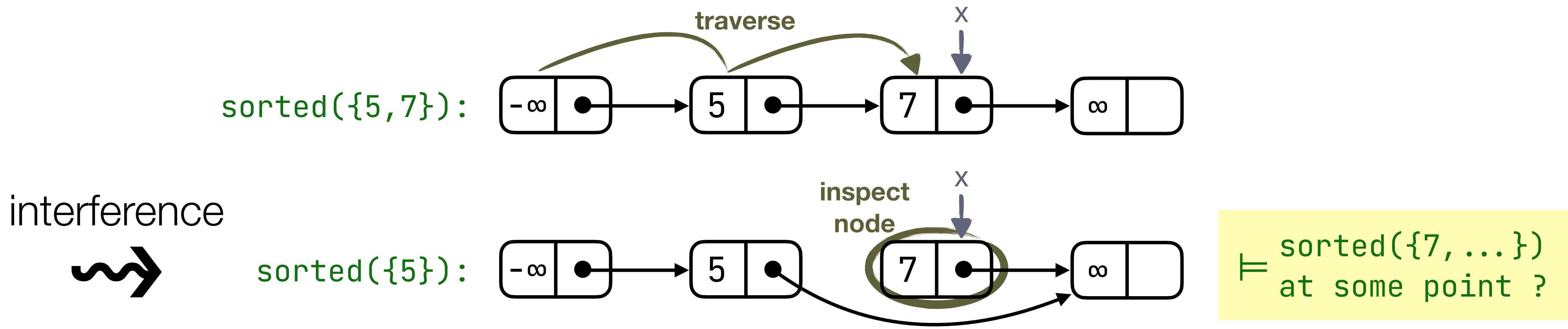


Classical Reasoning

- : prophecy about outcome of ("case split")

automation ?

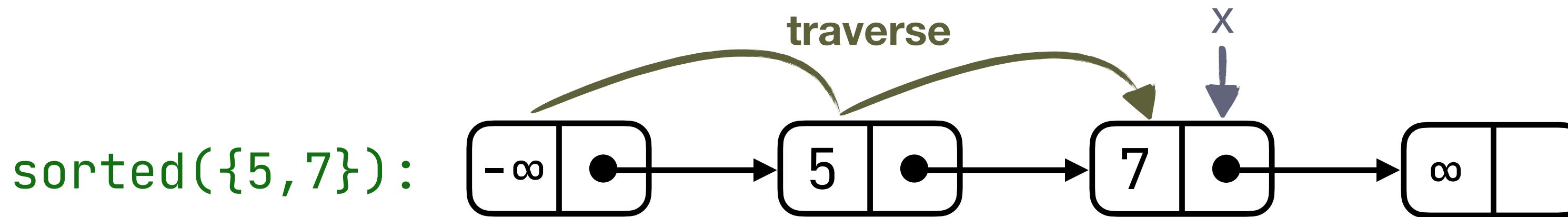
Example: contains(7)



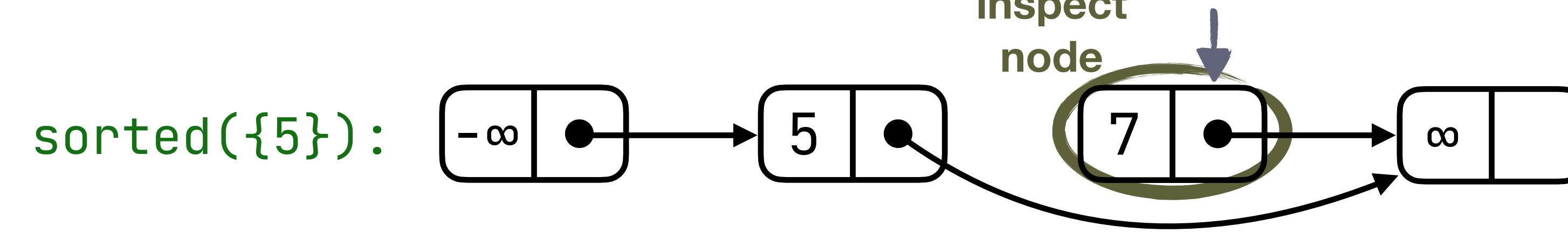
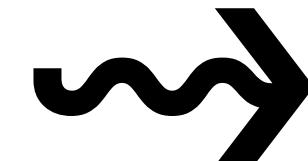
Classical Reasoning

- : prophecy about outcome of ("case split") ← automation ?
- : reconcile outcome with prophecy

Example: contains(7)



interference

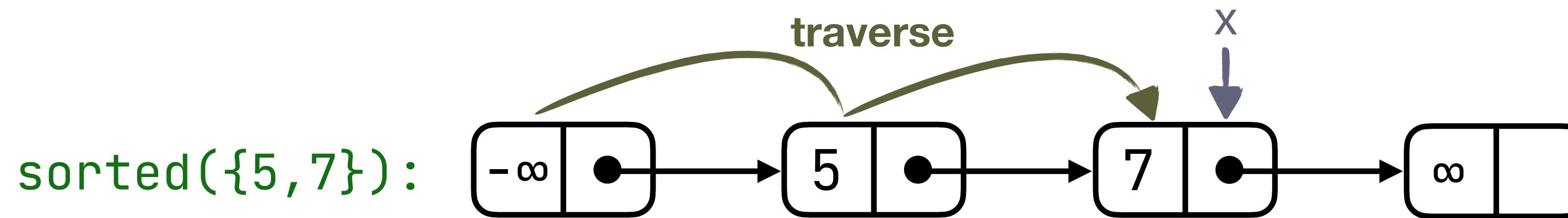


\models sorted({7, ...})
at some point ?

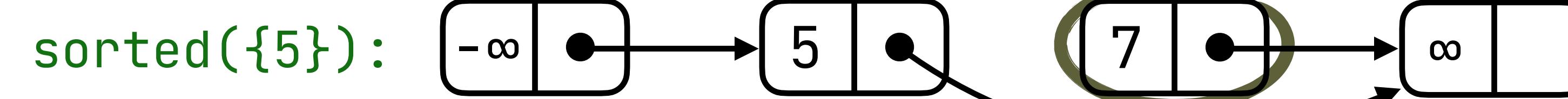
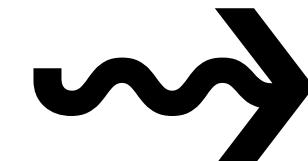
Contribution: Past Reasoning

- : record past state

Example: contains(7)



interference

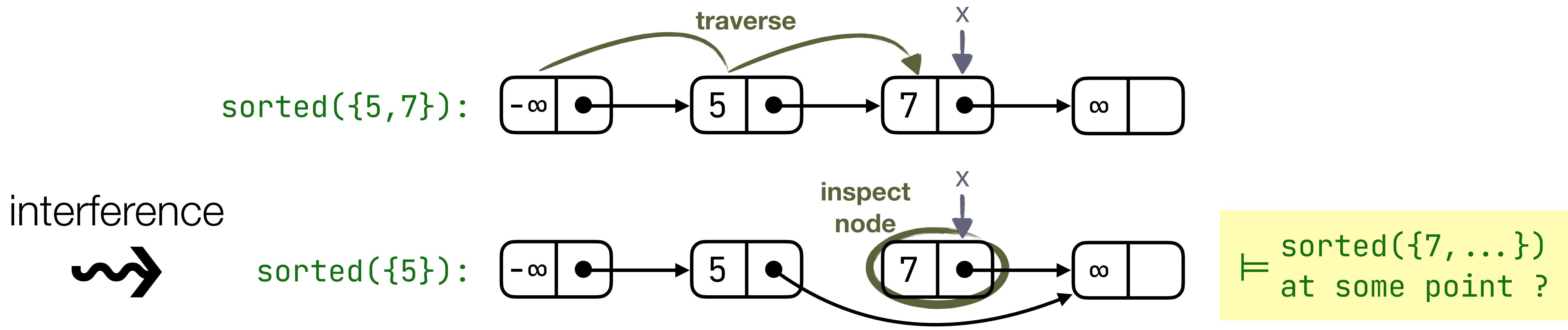


\models sorted({7, ...})
at some point ?

Contribution: Past Reasoning

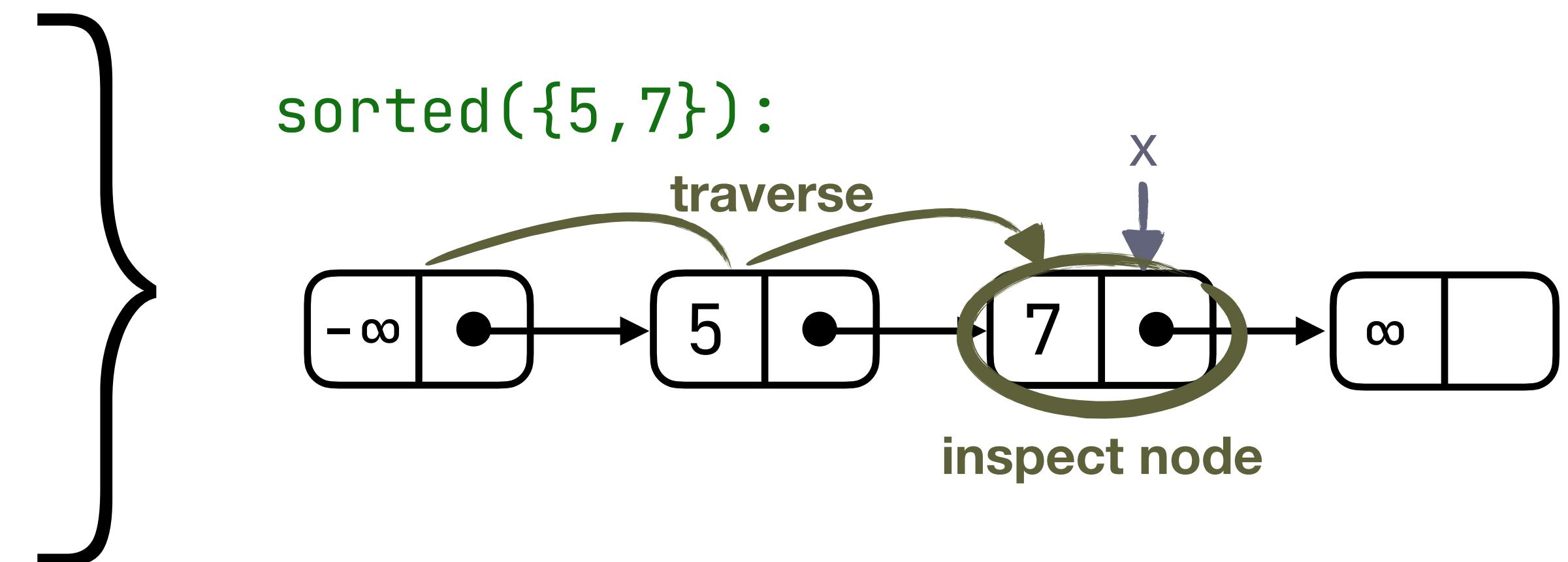
- : record past state
- : propagate outcome into past state

Example: contains(7)



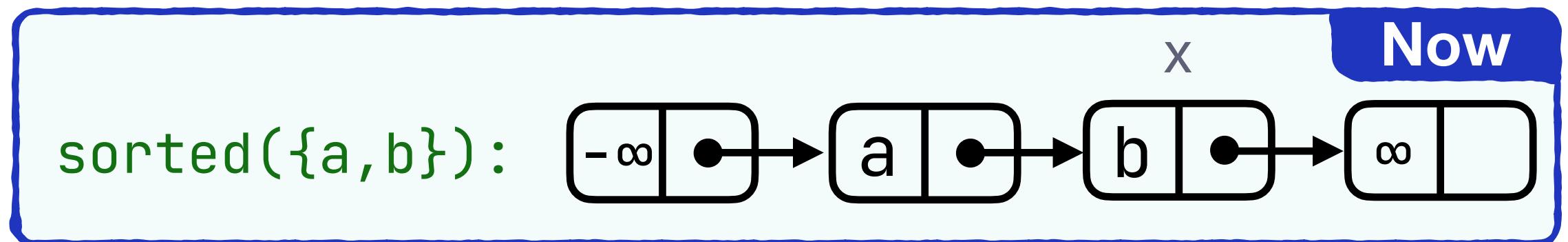
Contribution: Past Reasoning

- : record past state
- : propagate outcome into past state



Past Reasoning

```
// ...
x = /* a→next */
```

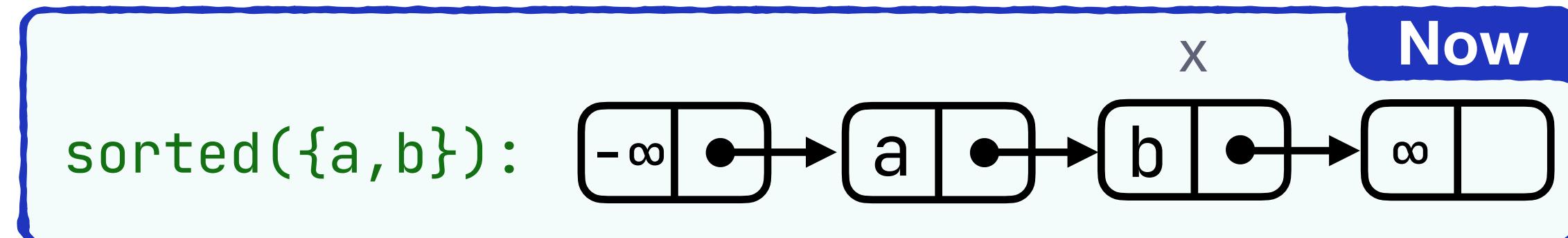


```
if (x→key == k) {
    return true;
} else { /* continue traversal */ }
```

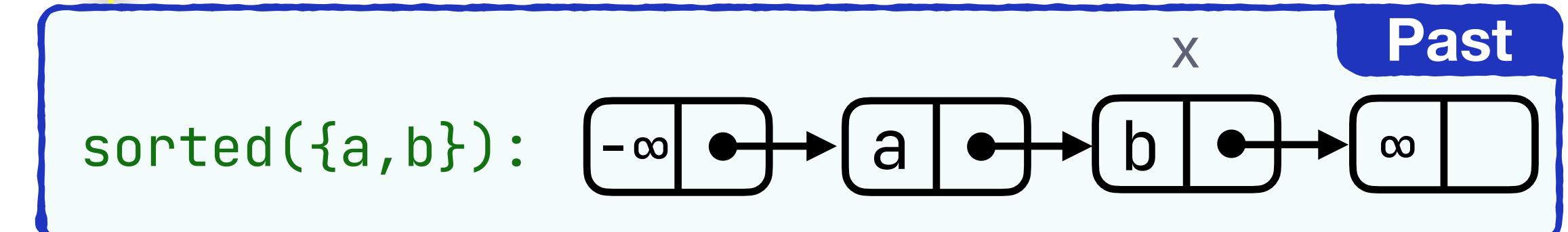
Past Reasoning

// ...

x = /* a→next */



*



if (x→key = k) {

 return true;

} else { /* continue traversal */ }

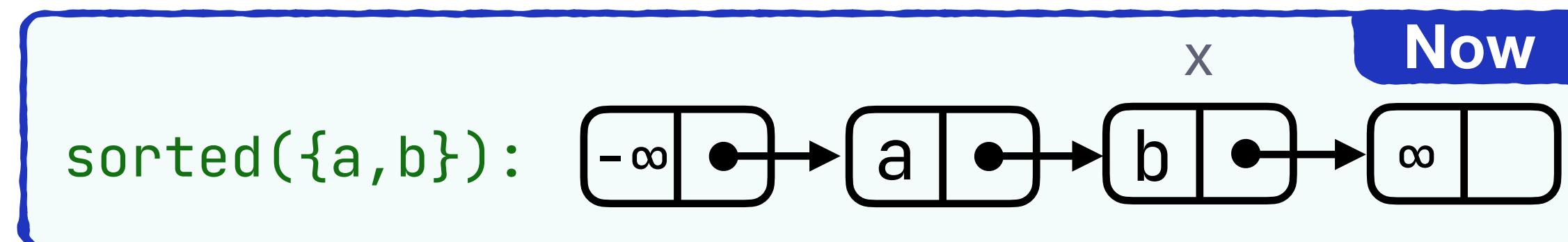
Past := Now

in some (previous)
state of the execution

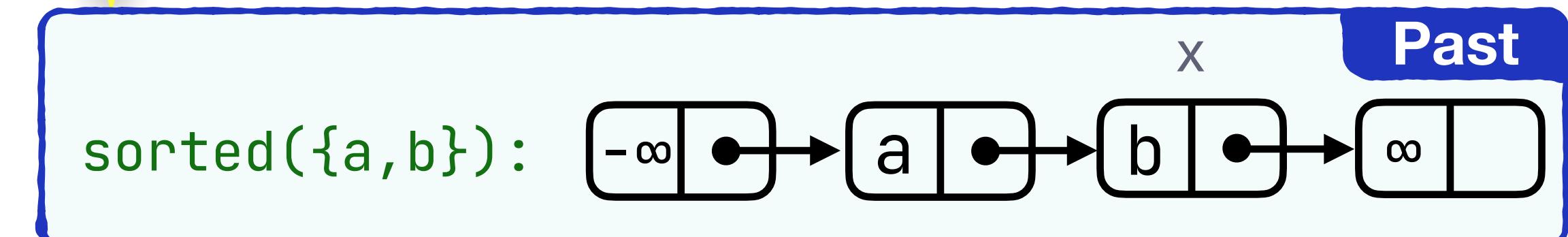
Past Reasoning

// ...

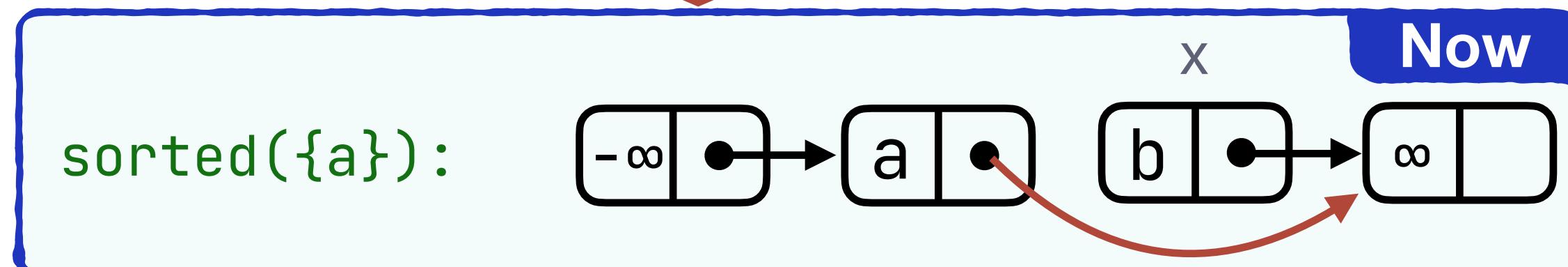
x = /* a→next */



*



interference



if (x→key = k) {

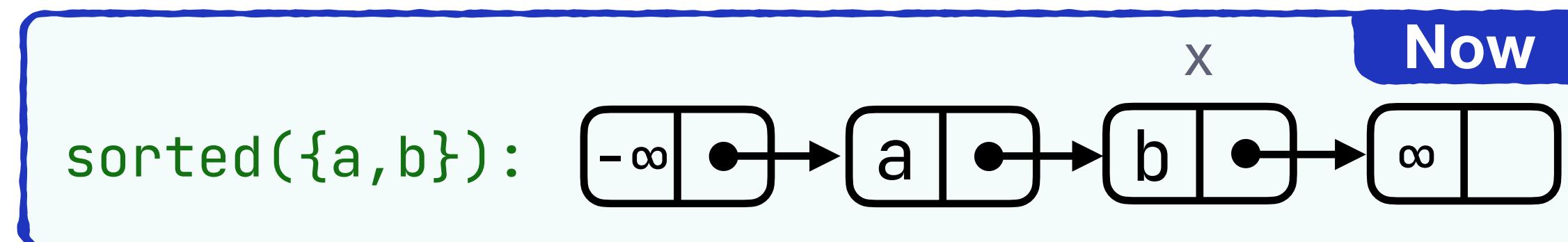
return true;
} else { /* continue traversal */ }

Past := Now in some (previous) state of the execution

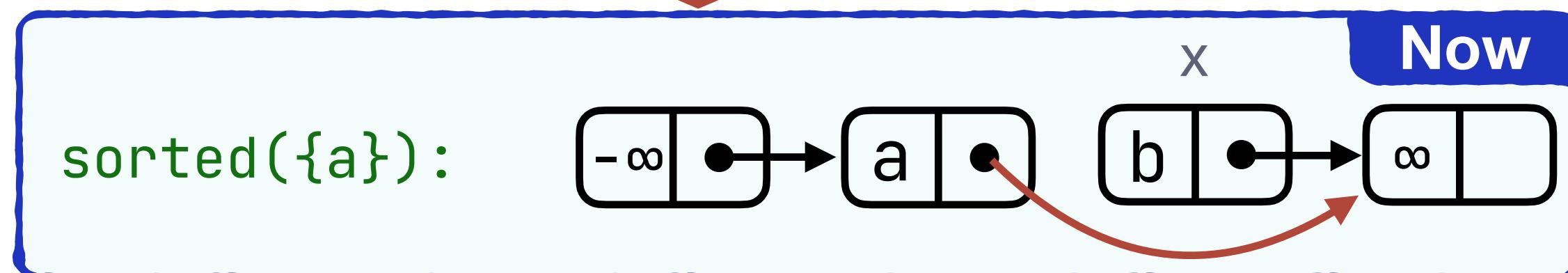
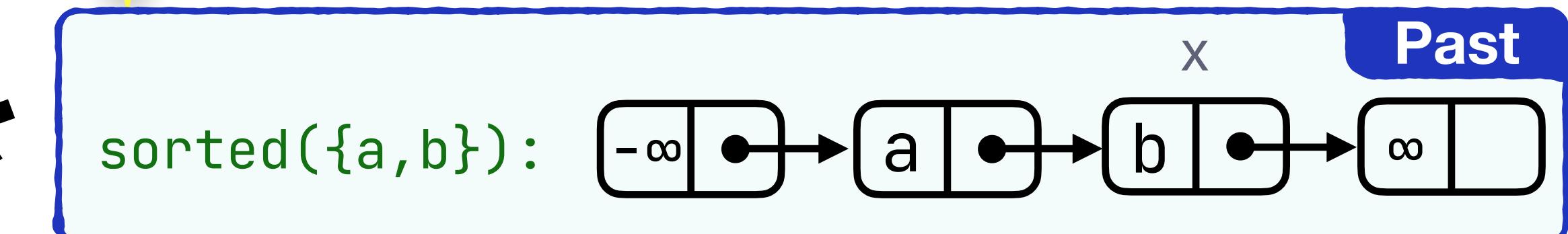
Past Reasoning

// ...

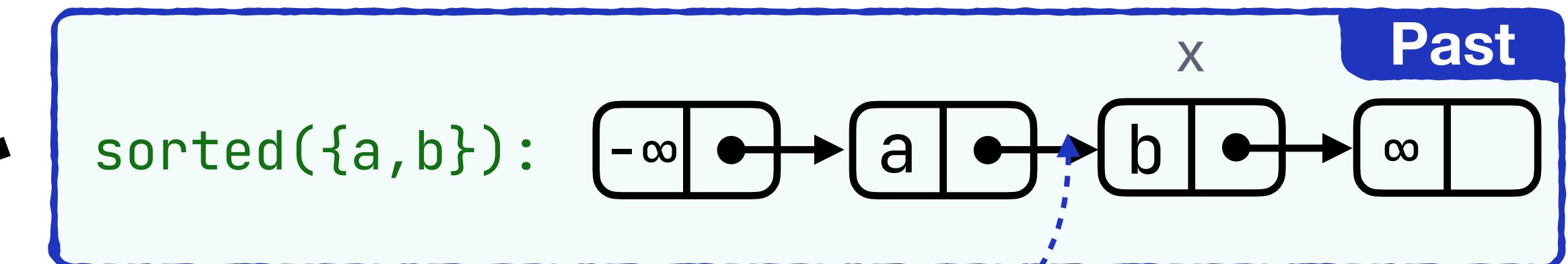
x = /* a→next */



*

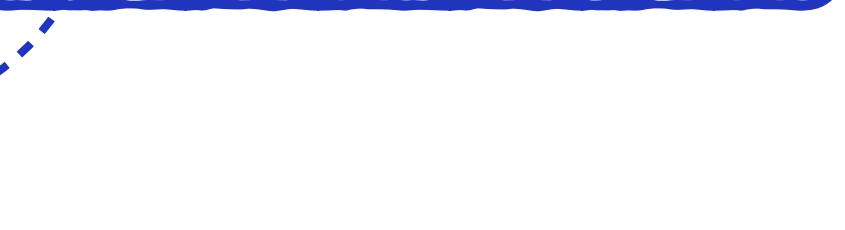
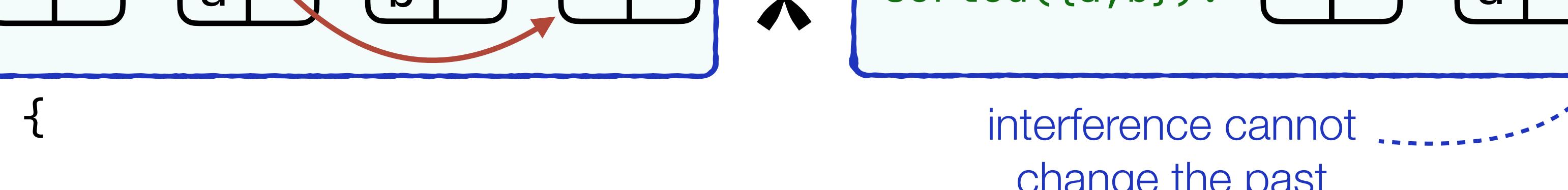


*



if (x→key = k) {

interference cannot
change the past



```
    return true;  
} else { /* continue traversal */ }
```

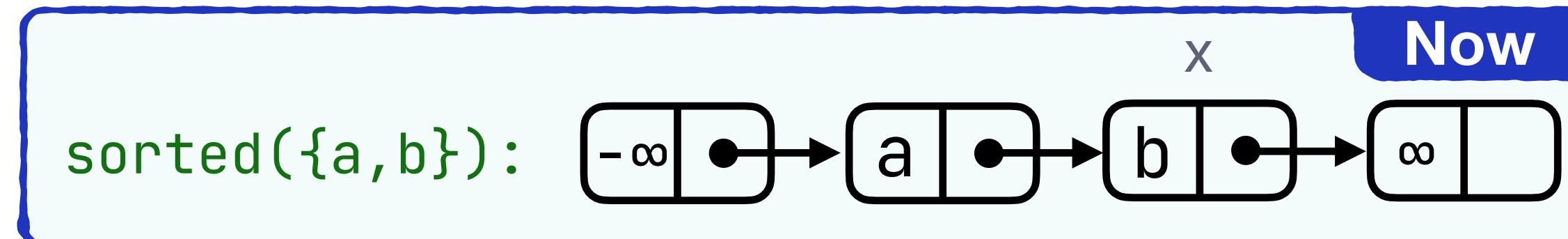
Past := Now

in some (previous)
state of the execution

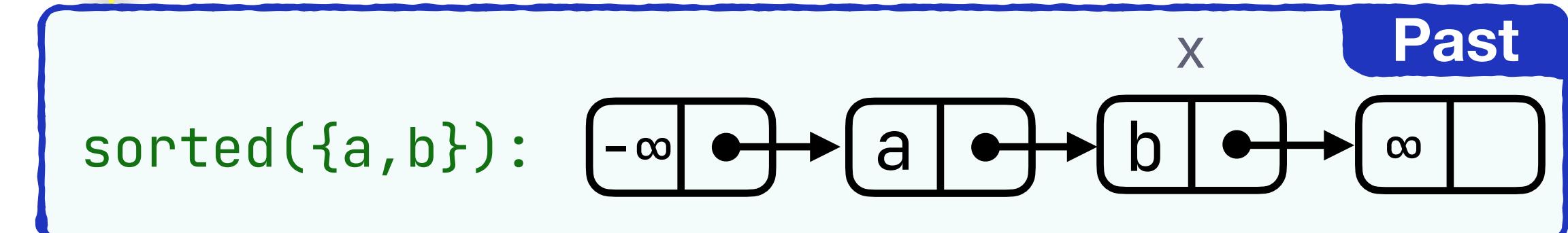
Past Reasoning

// ...

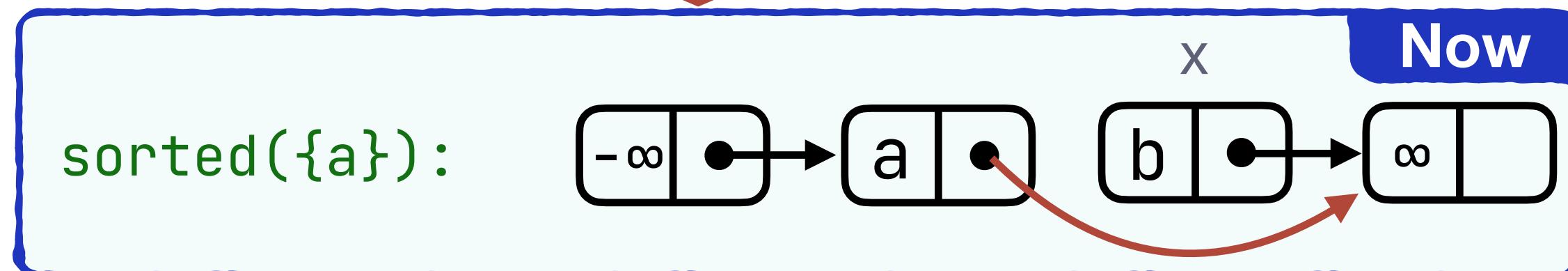
x = /* a→next */



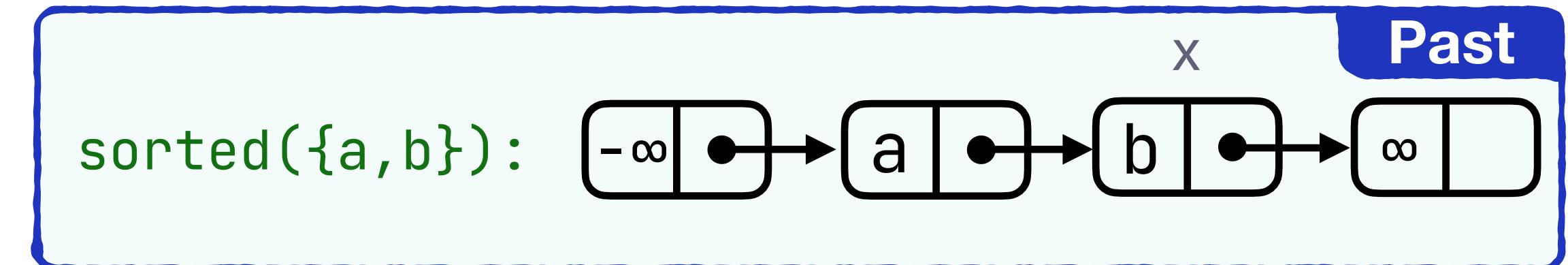
*



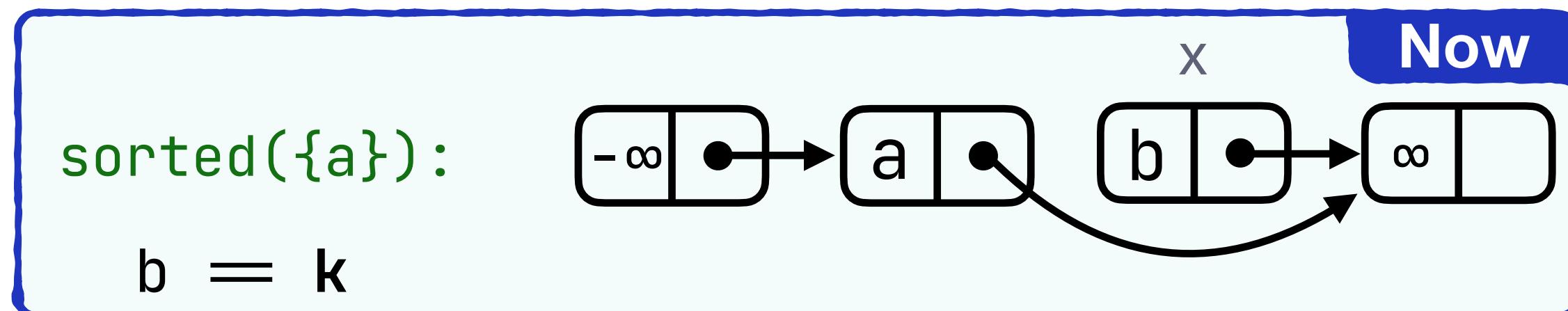
interference



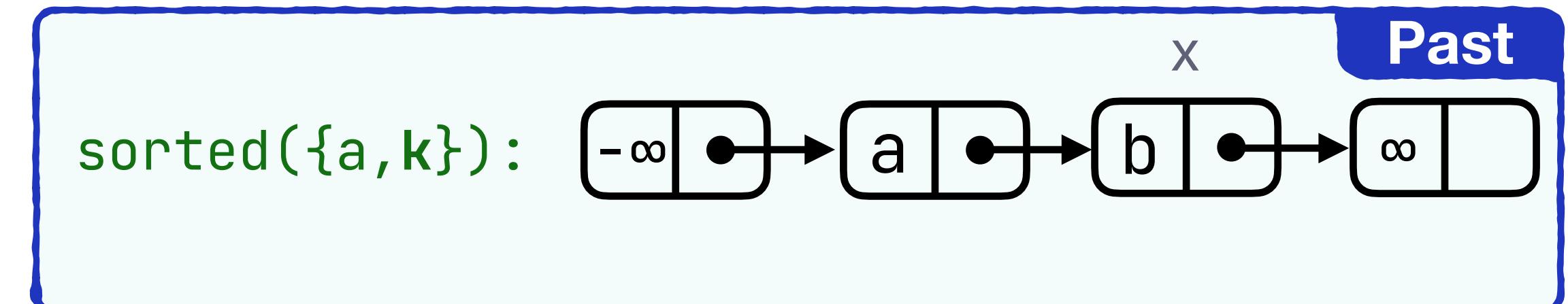
*



if (x→key = k) {



*



 return true;

} else { /* continue traversal */ }

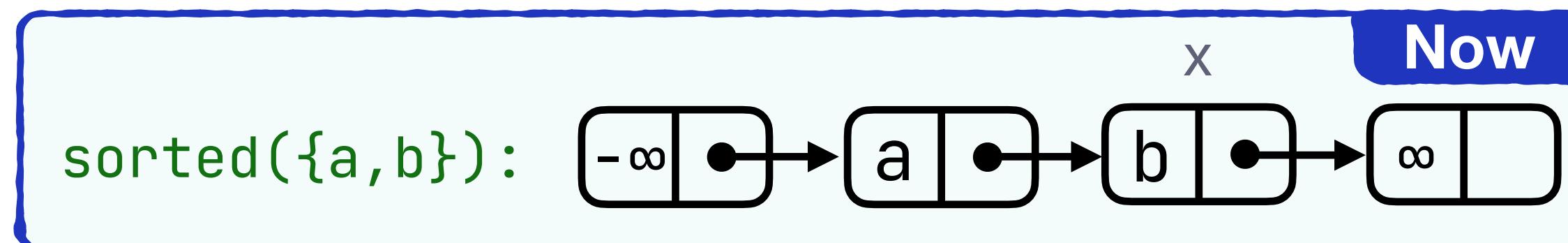
Past := Now

in some (previous)
state of the execution

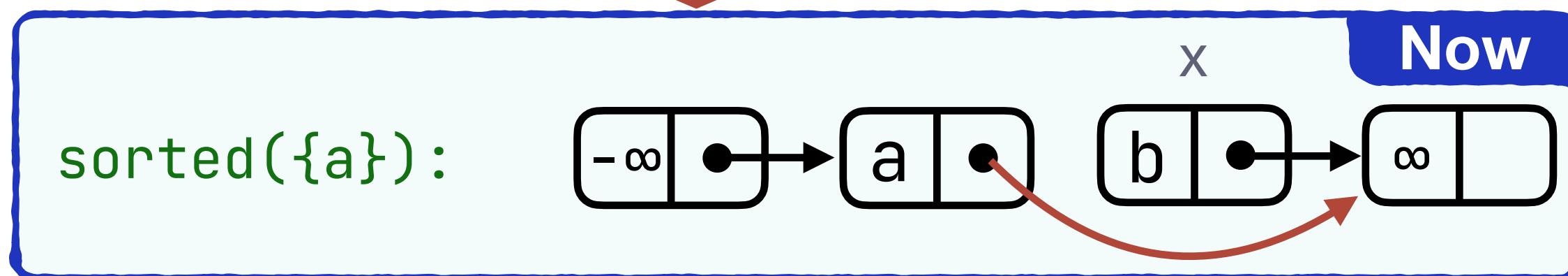
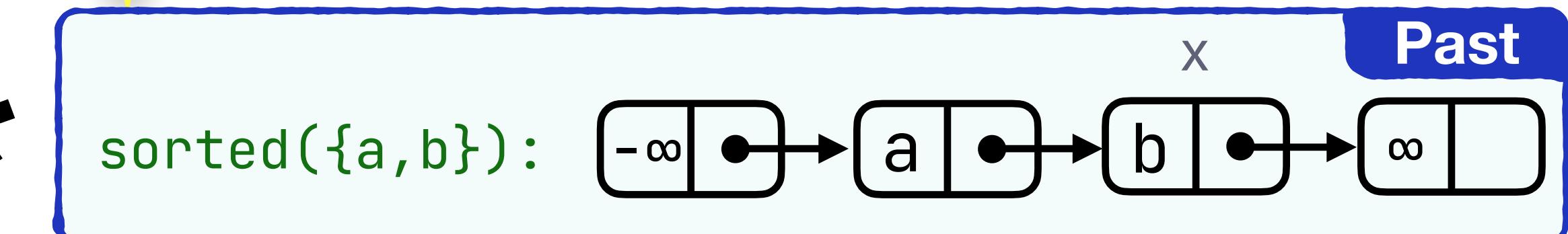
Past Reasoning

// ...

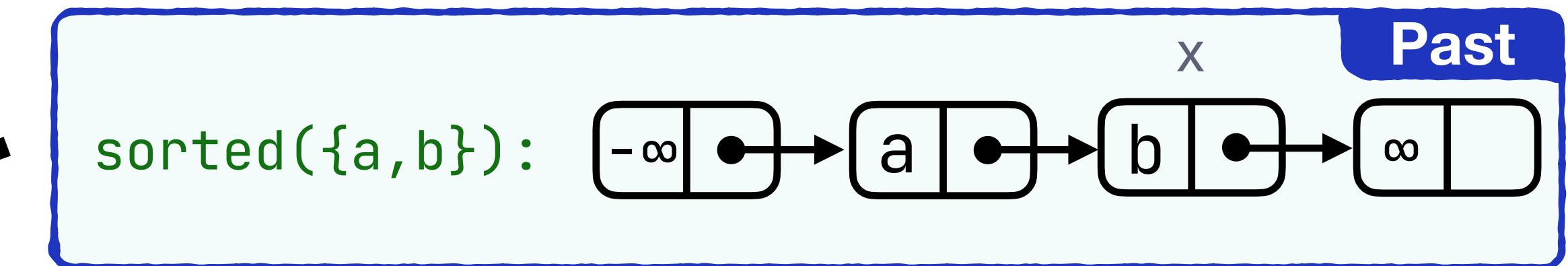
x = /* a→next */



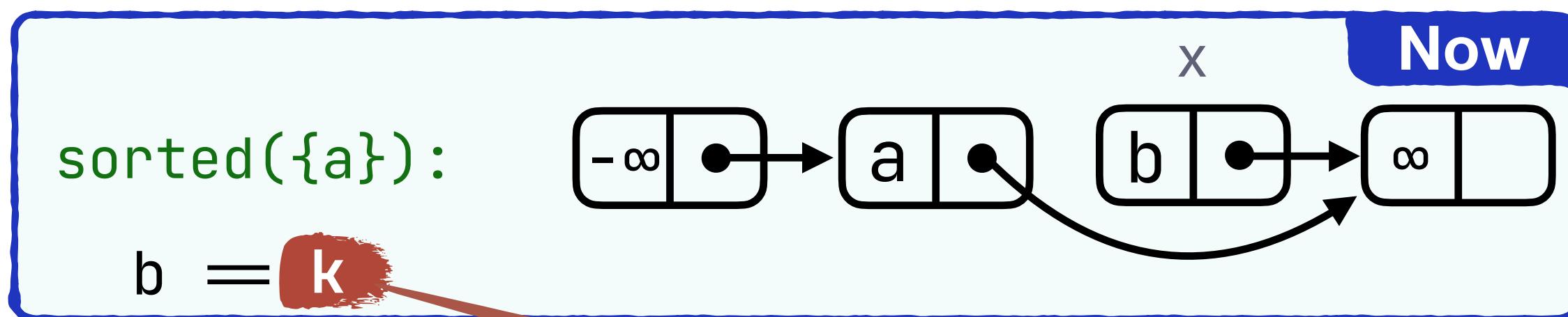
*



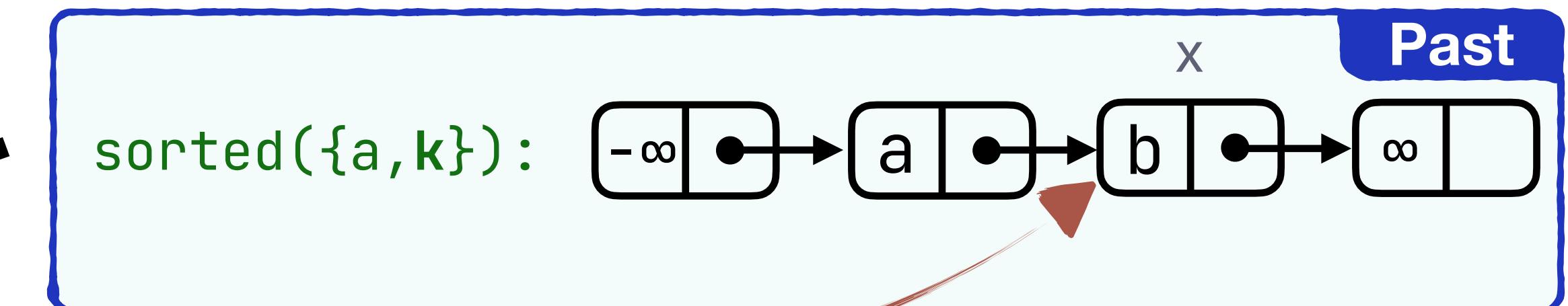
*



if (x→key = k) {



*



return true;

} else { /* continue traversal */ }

Past

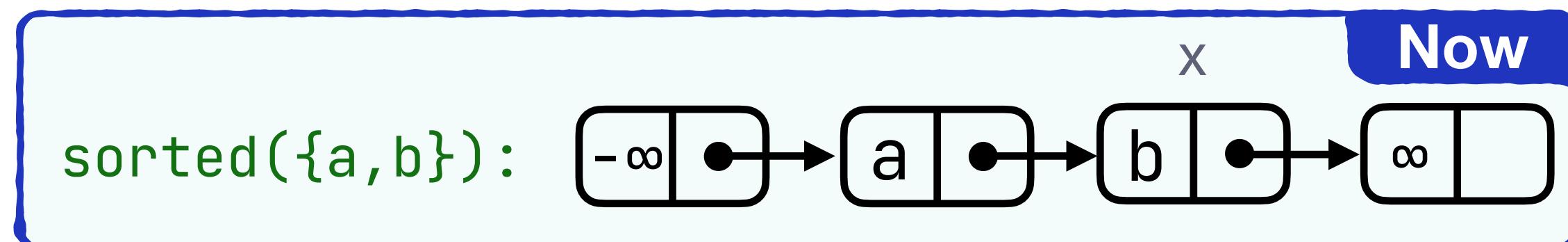
:= Now

in some (previous)
state of the execution

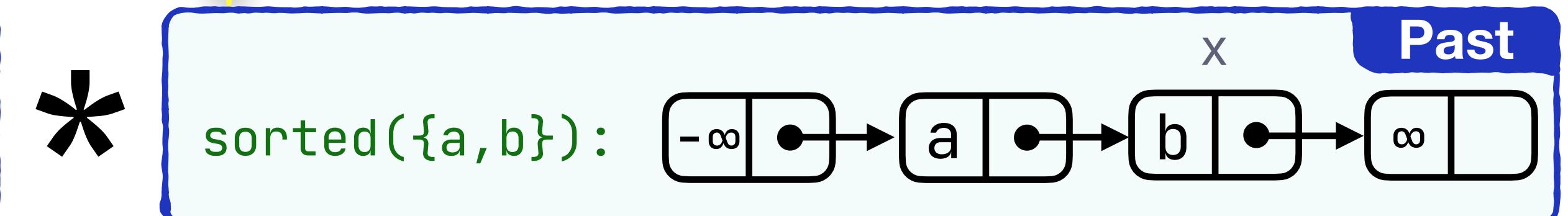
Past Reasoning

// ...

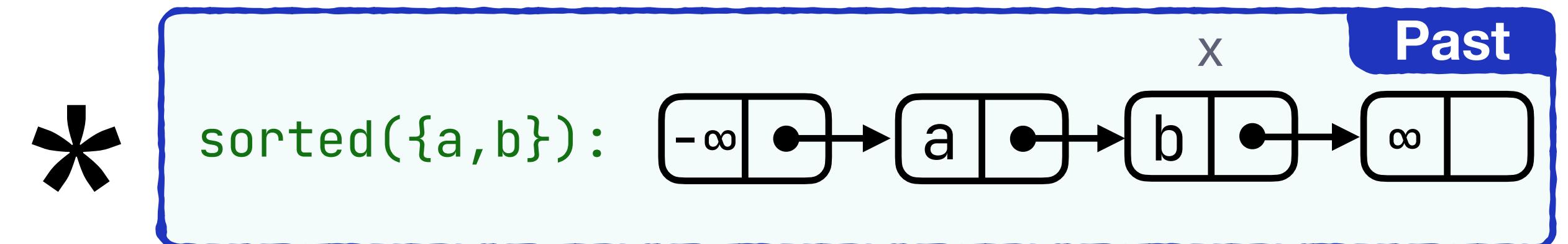
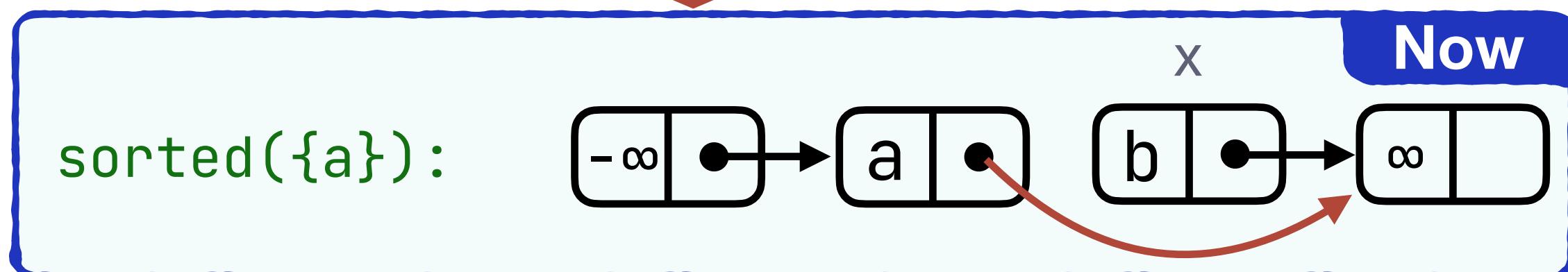
x = /* a→next */



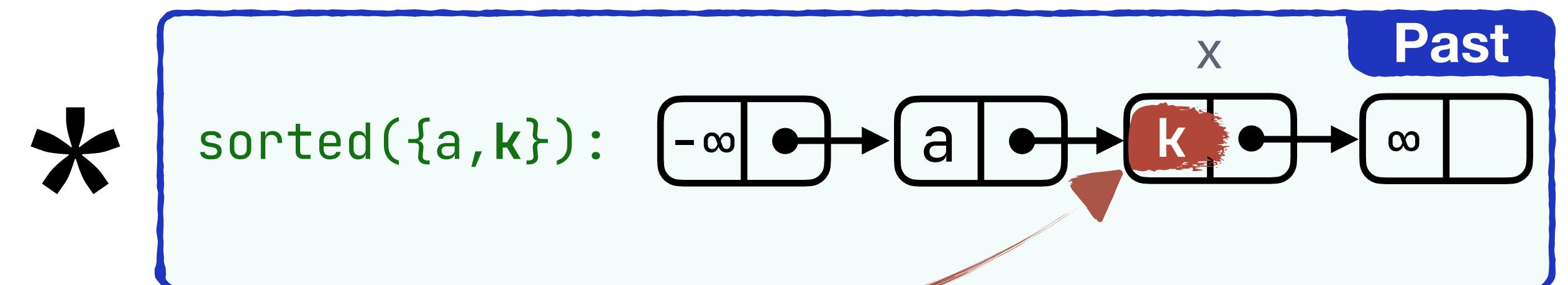
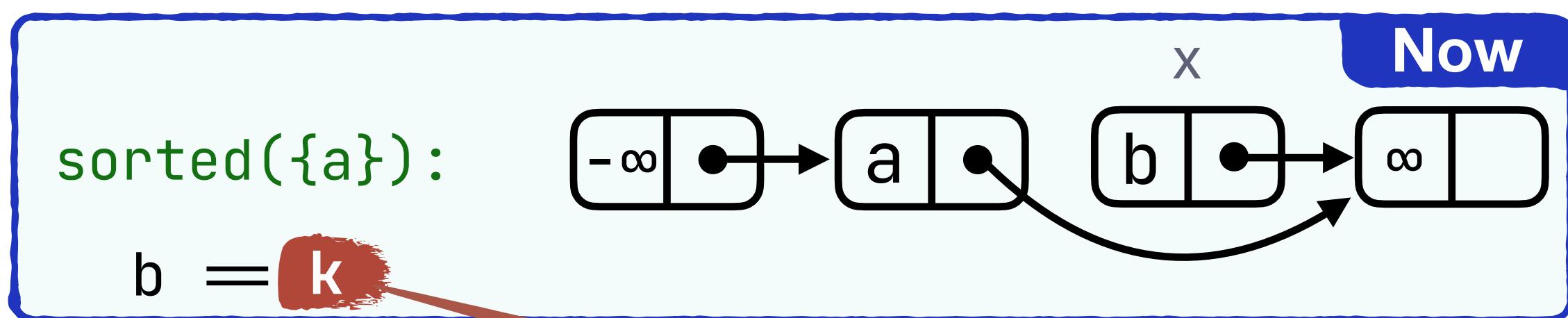
Past := **Now** in some (previous) state of the execution



interference



if (x→key = k) {

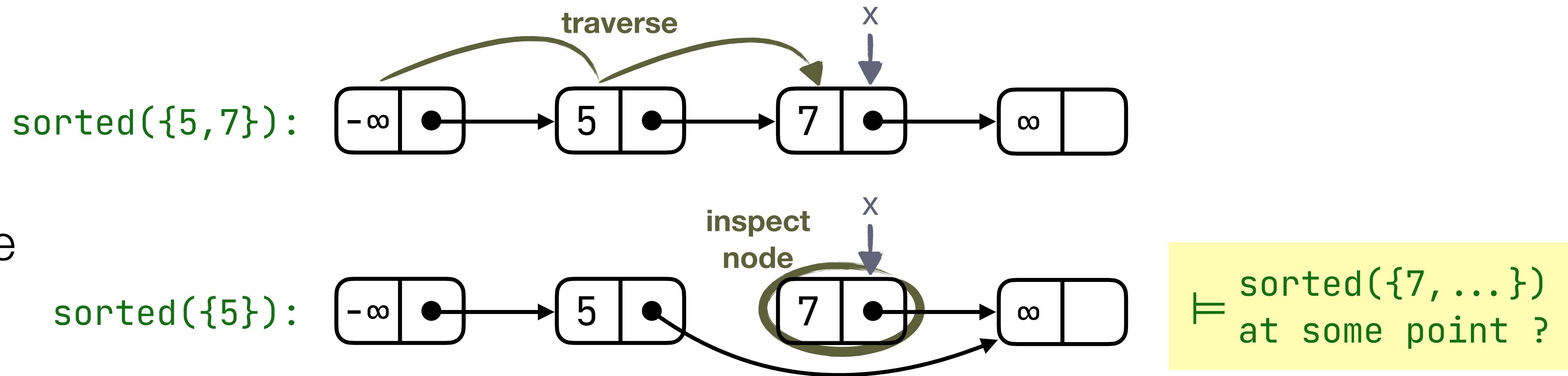


 return true;

} else { /* continue traversal */ }

\models sorted({k, ...}) at some point !

Example: contains(7)



Contribution: Past Reasoning

- formalization in abstract SL
- light-weight instance of hindsight reasoning
- implementation

Recap:

A Concurrent Program Logic with a Future and History

- program logic with
 - chunks ↪ complex unbounded updates via simple ones
 - pasts ↪ lightweight hindsight reasoning in SL
 - tool ↪ first to verify challenging examples like Harris' set

Benchmark	Lineariz.
Fine-Grained set	46s ✓
Lazy set	77s ✓
FEMRS tree (no maintenance)	130s ✓
Vechev&Yahav 2CAS set	125s ✓
Vechev&Yahav CAS set	54s ✓
ORVYY set	47s ✓
Michael set	306s ✓
Harris set	1378s ✓



Thanks

Recap:

A Concurrent Program Logic with a Future and History

- program logic with
 - chunks ↪ complex unbounded updates via simple ones
 - pasts ↪ lightweight hindsight reasoning in SL
 - tool ↪ first to verify challenging examples like Harris' set

Benchmark	Lineariz.
Fine-Grained set	46s ✓
Lazy set	77s ✓
FEMRS tree (no maintenance)	130s ✓
Vechev&Yahav 2CAS set	125s ✓
Vechev&Yahav CAS set	54s ✓
ORVYY set	47s ✓
Michael set	306s ✓
Harris set	1378s ✓

