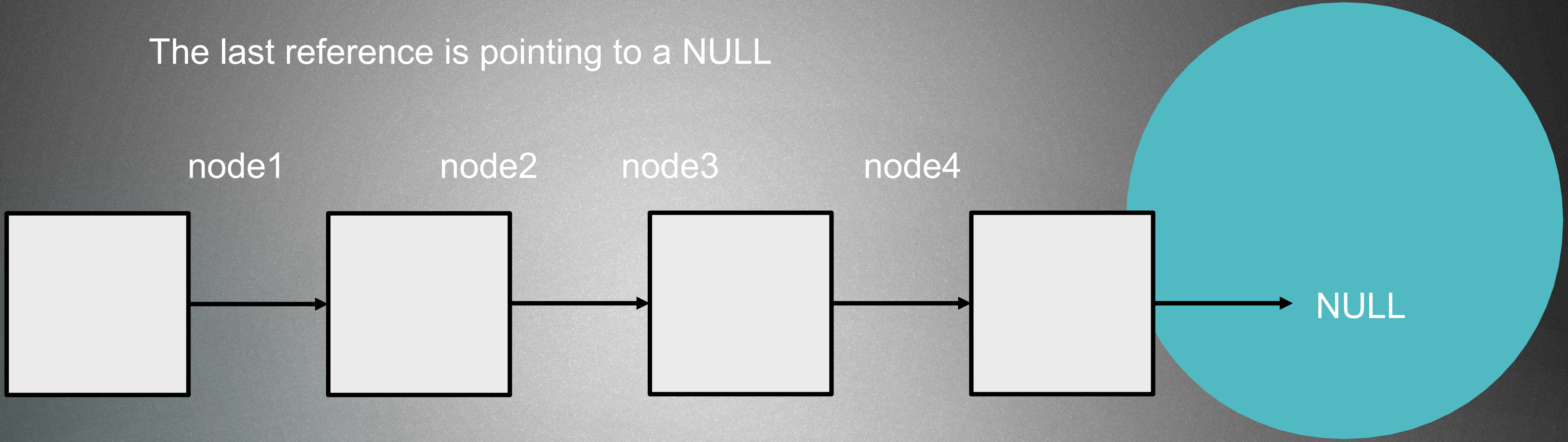
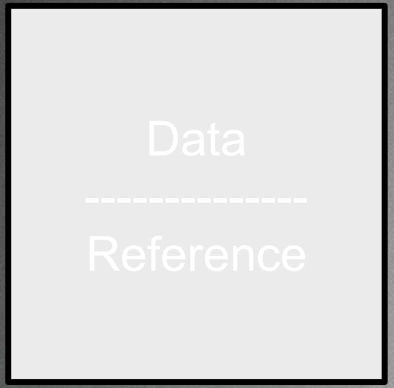
LINKED LISTS

LINKED LISTS

Linked lists are composed of nodes and references / pointers pointing from one node to the other !!!

The last reference is pointing to a NULL





Data

Reference

A single node:

- contains data → integer, double or custom object
- contains a reference pointing to the next node in the linked list

```
class Node {

       data
       Node nextNode

       ...
}
```

- Each node is composed of a data and a reference/link to the next node in the sequence
- Simple and very common data structure !!!
- They can be used to implement several other common data types: stacks, queues
- Simple linked lists by themselves do not allow random access to the data // so we can not use indexes ... getItem(int index) !!!
- Many basic operations such as obtaining the last node of the list or finding a node that contains a given data or locating the place where a new node should be inserted — require sequential scanning of most or all of the list elements

Advantages

- Linked lists are dynamic data structures (arrays are not !!!)
- It can allocate the needed memory in run-time
- Very efficient if we want to manipulate the first elements
- EASY IMPLEMENTATION
- Can store items with different sizes: an array assumes every element to be exactly the same
- It's easier for a linked list to grow organically. An array's size needs to be known ahead of time, or re-created when it needs to grow

Disadvantages

- Waste memory because of the references
- Nodes in a linked list must be read in order from the beginning as linked lists have sequential access (array items can be reached via indexes in **O(1)** time !!!)
- Difficulties arise in linked lists when it comes to reverse traversing. Singly linked lists are extremely difficult to navigate backwards,
- Solution: doubly linked lists → easier to read, but memory is wasted in allocating space for a back pointer

Linked list operations insertion

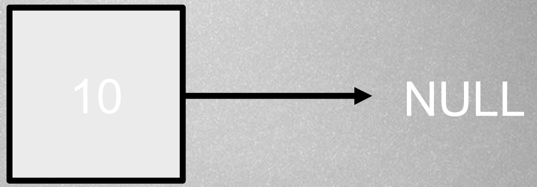
Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

linkedList.insertAtStart(10);

Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

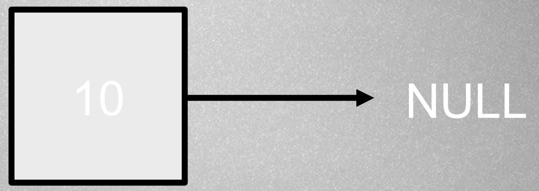
linkedList.insertAtStart(10);



Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

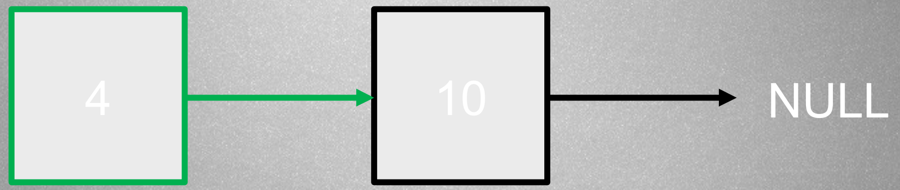
linkedList.insertAtStart(4);



Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

```
linkedList.insertAtStart(4);
```

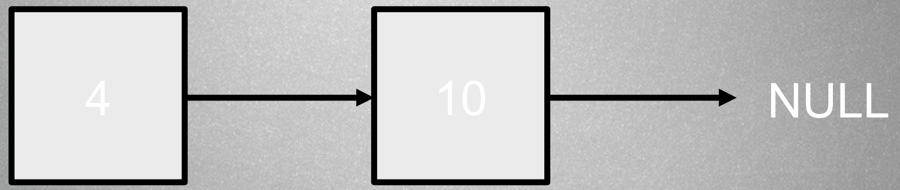


We just have to set the pointer to point to the next node

Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

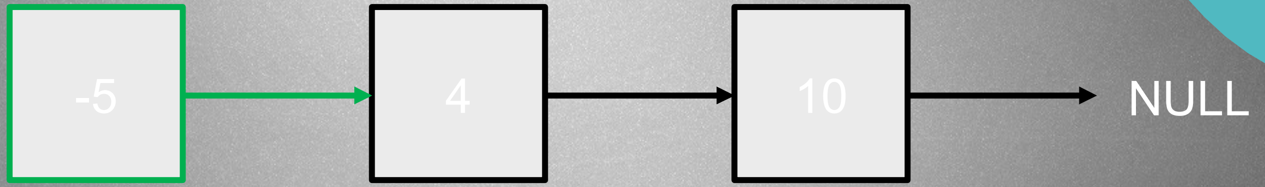
```
linkedList.insertAtStart(-5);
```



Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

linkedList.insertAtStart(-5);

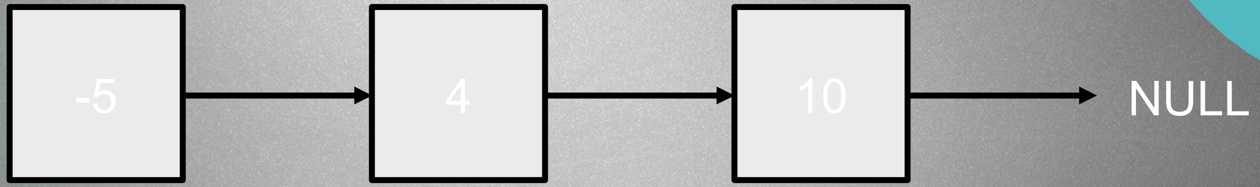


We just have to set the pointer to point to the next node

Linked list operations insertion

Inserting items at the beginning of the linked list: very simple, we just have to update the references → O(1) time complexity

```
linkedList.insertAtStart(-5);
```



-5

4

10

NULL

We just have to set the pointer to point to the next node

SO THIS OPERATION IS VERY FAST, THIS IS WHY WE LIKE LINKED LISTS !!!

Linked list operations insertion

Inserting items at the end of the linked list: not that very simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

Linked list operations insertion

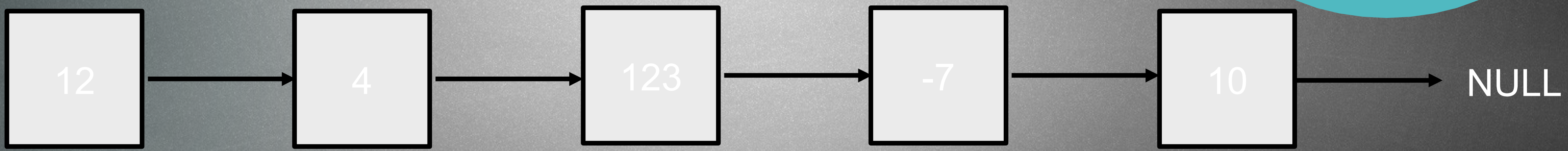
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



12

4

123

-7

10

NULL

Linked list operations insertion

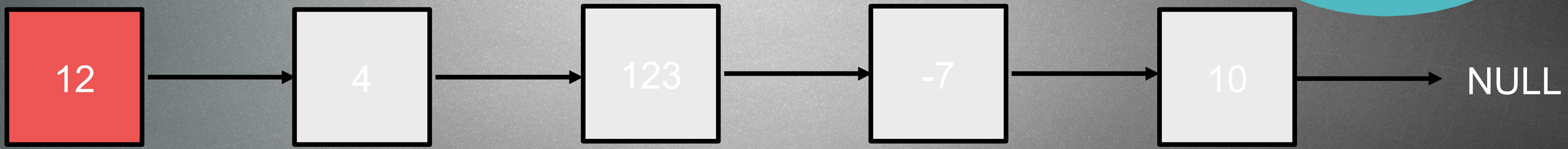
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? No, keep going

Linked list operations insertion

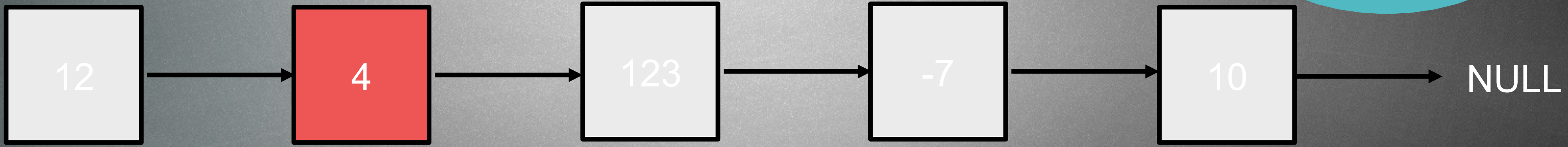
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? No, keep going

Linked list operations insertion

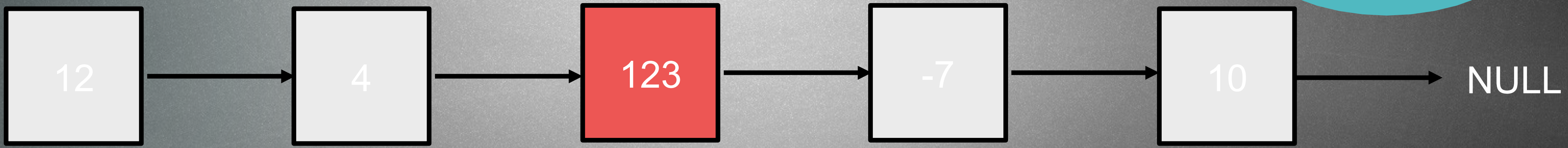
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? No, keep going

Linked list operations insertion

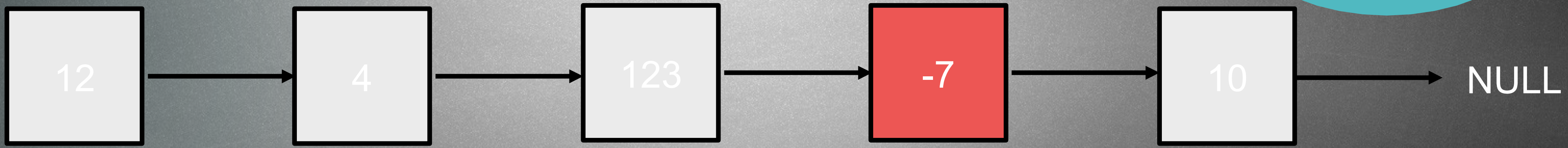
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? No, keep going

Linked list operations insertion

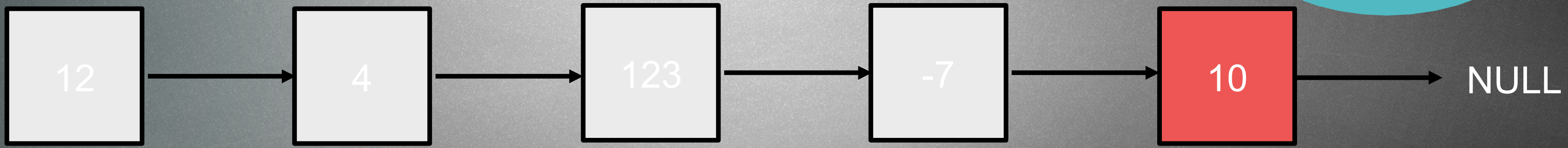
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? Yes, this is the last node

We have to update the references

Linked list operations insertion

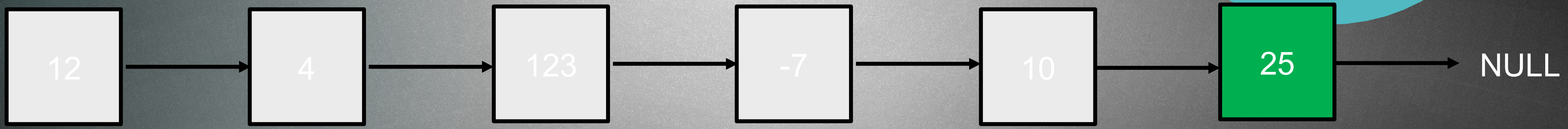
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



We have to get to the last node: so is it pointing to a NULL? Yes, this is the last node

We have to update the references

Linked list operations insertion

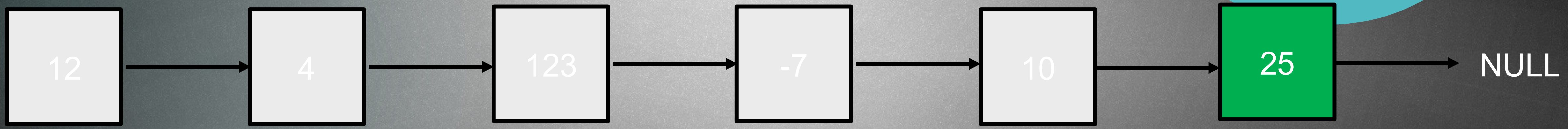
Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

linkedList.insertAtEnd(25);



Very important: updating the references again takes O(1) BUT we have to traverse the list itself and that what takes O(N) !!! // O(1) + O(N) = O(N)

Linked list operations insertion

Inserting items at the end of the linked list: not thatvery simple, we have to traverse the whole linked list to find the last node

How do we find the last node? We know the last node is pointing to a NULL !!!

+ we have to update the references when we get there

O(N) time complexity

Insert at the beginning	O(1)
Inserting at the end	O(N)

Linked list operations remove

Remove item at the beginning of the list is always very fast: we do not have to search

- the item, we just have to update the references accordingly
  - O(1) time complexity

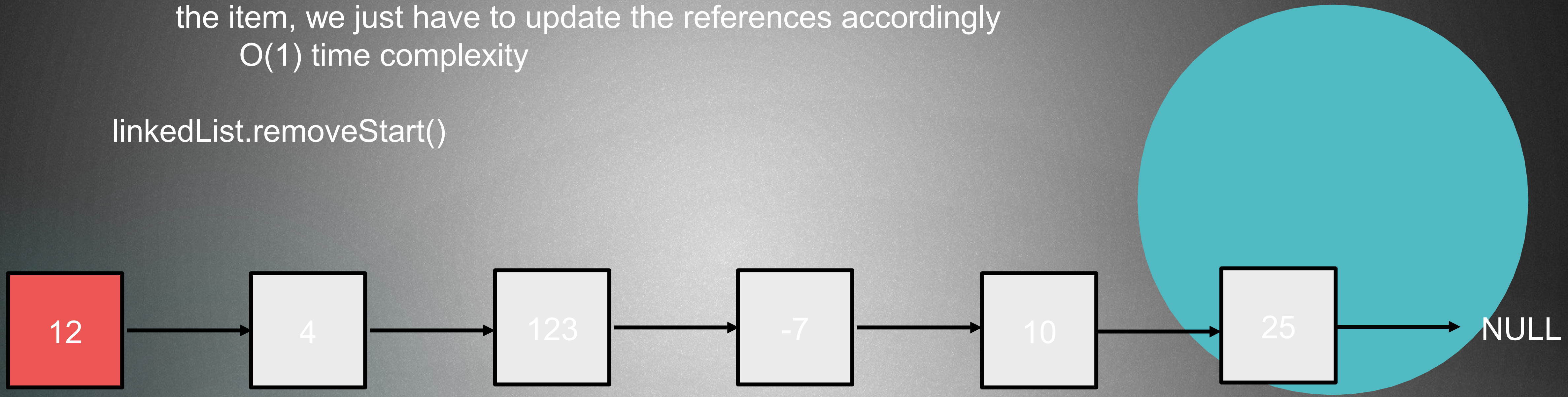
linkedList.removeStart()

Linked list operations   remove

Remove item at the beginning of the list is always very fast: we do not have to search the item, we just have to update the references accordingly

O(1) time complexity

linkedList.removeStart()

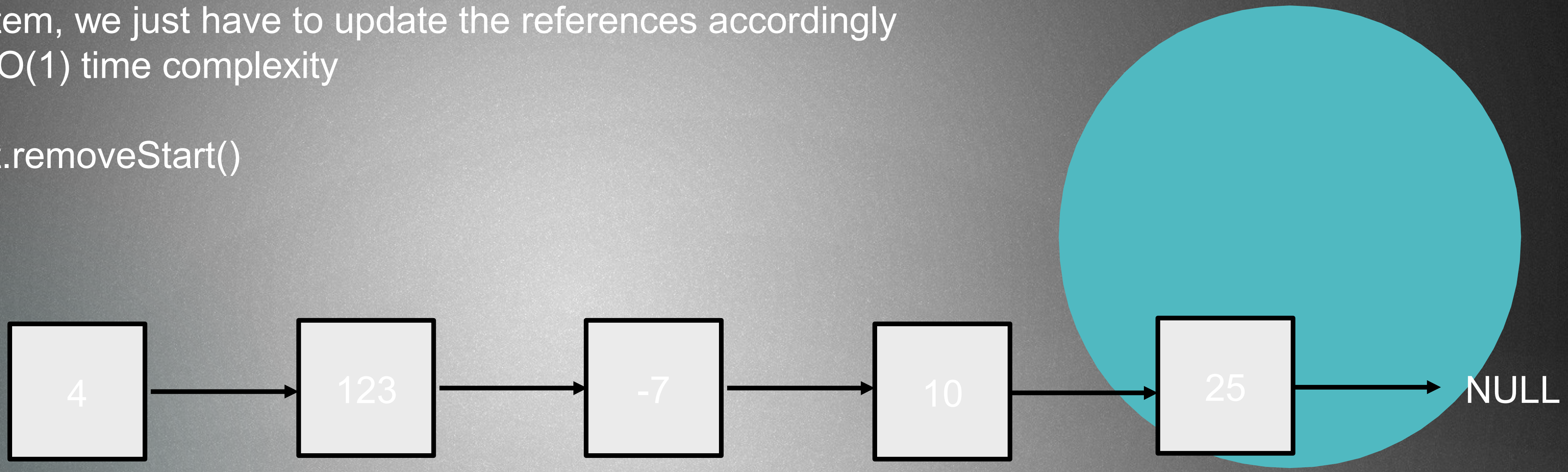


Linked list operations remove

Remove item at the beginning of the list is always very fast: we do not have to search the item, we just have to update the references accordingly

O(1) time complexity

linkedList.removeStart()

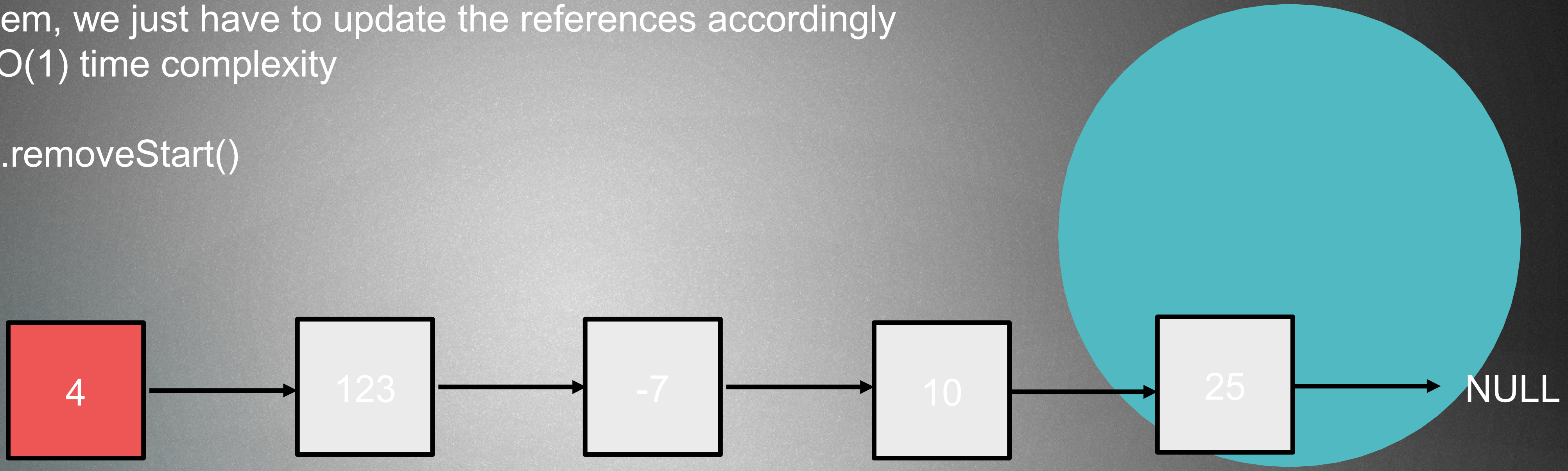


<u>Linked list operations</u> remove

Remove item at the beginning of the list is always very fast: we do not have to search the item, we just have to update the references accordingly

O(1) time complexity

linkedList.removeStart()

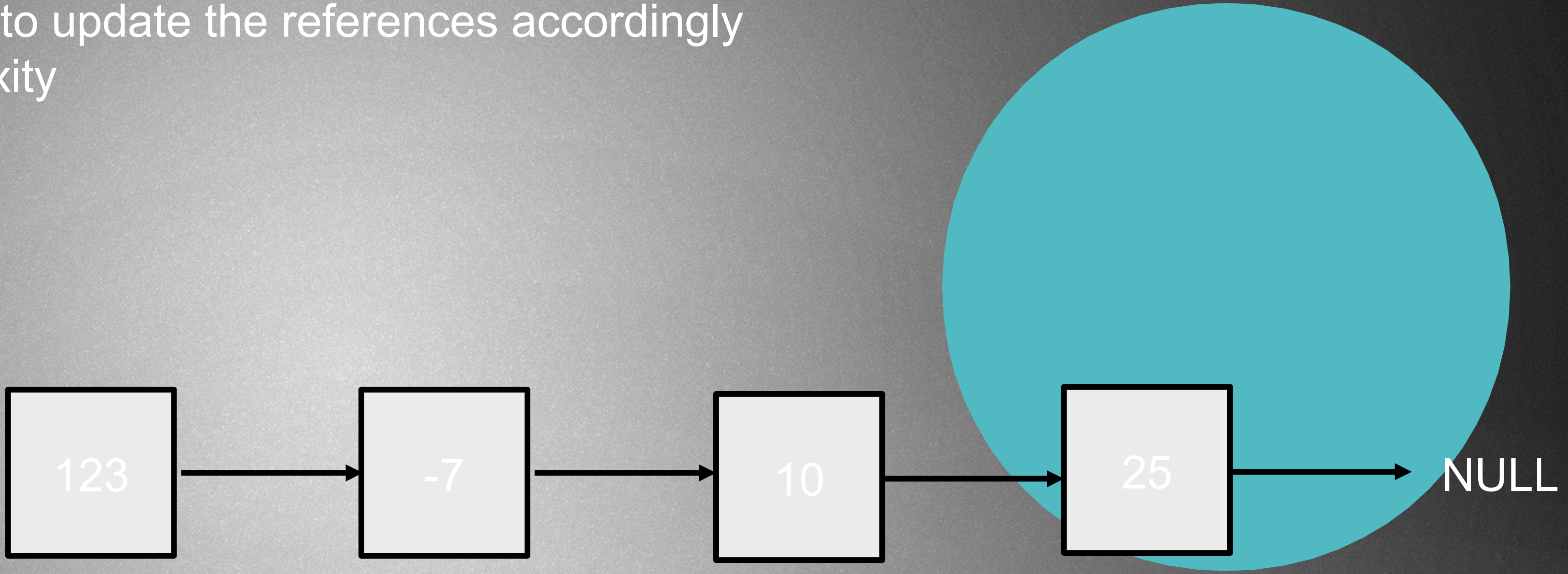


Linked list operations   remove

Remove item at the beginning of the list is always very fast: we do not have to search the item, we just have to update the references accordingly

O(1) time complexity

linkedList.removeStart()



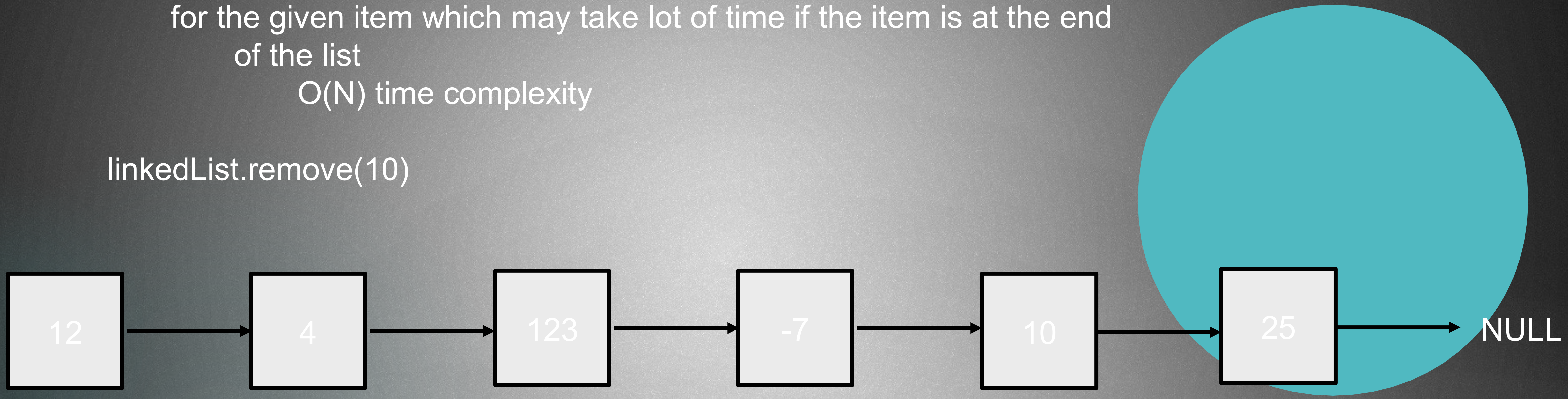
We just to remove it, very simple and fast operation !!!

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)

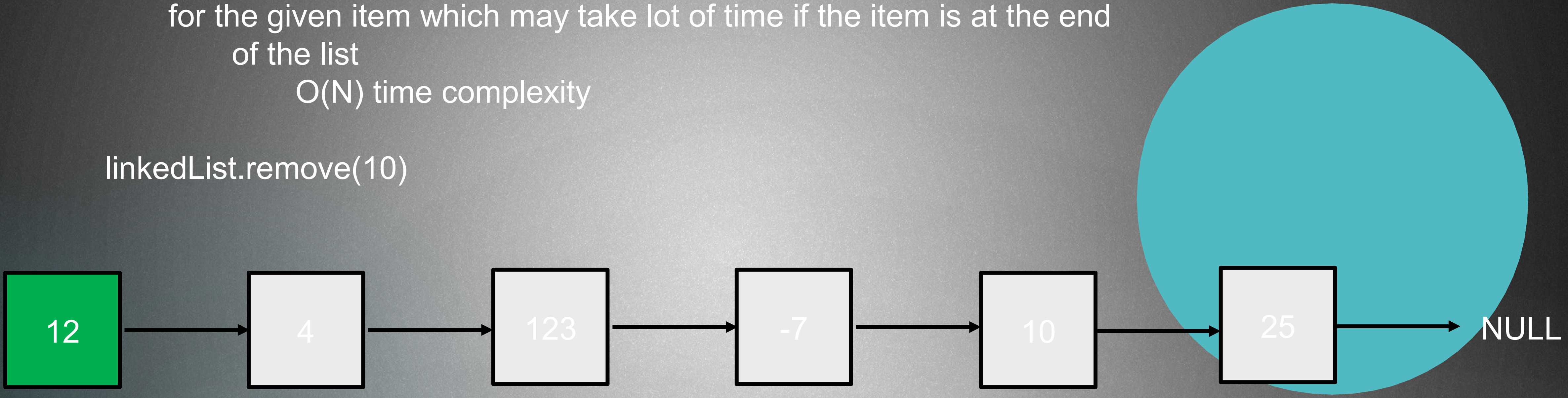


Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



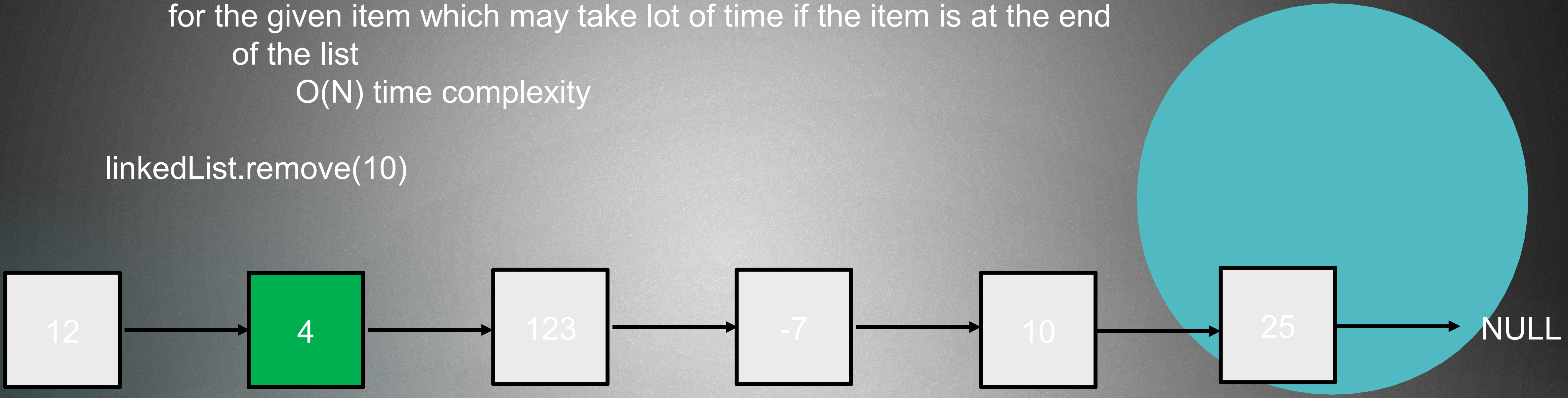
Is it the item we are looking for? NO, keep going

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



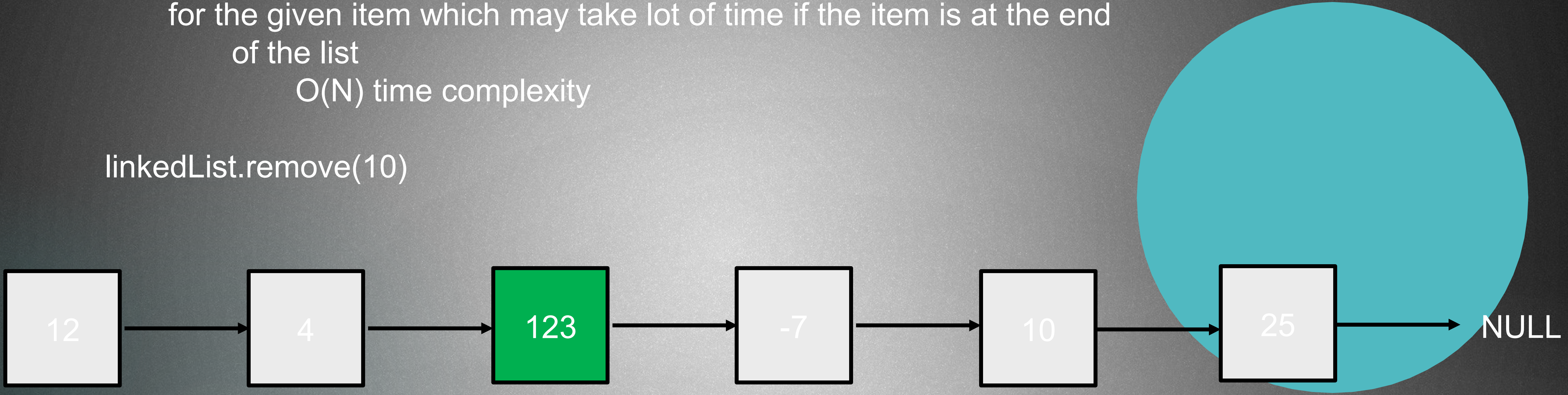
Is it the item we are looking for? NO, keep going

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



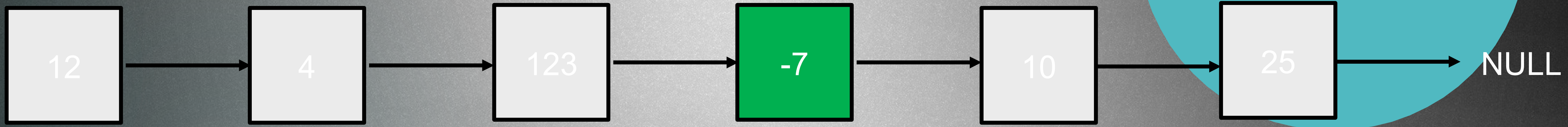
Is it the item we are looking for? NO, keep going

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



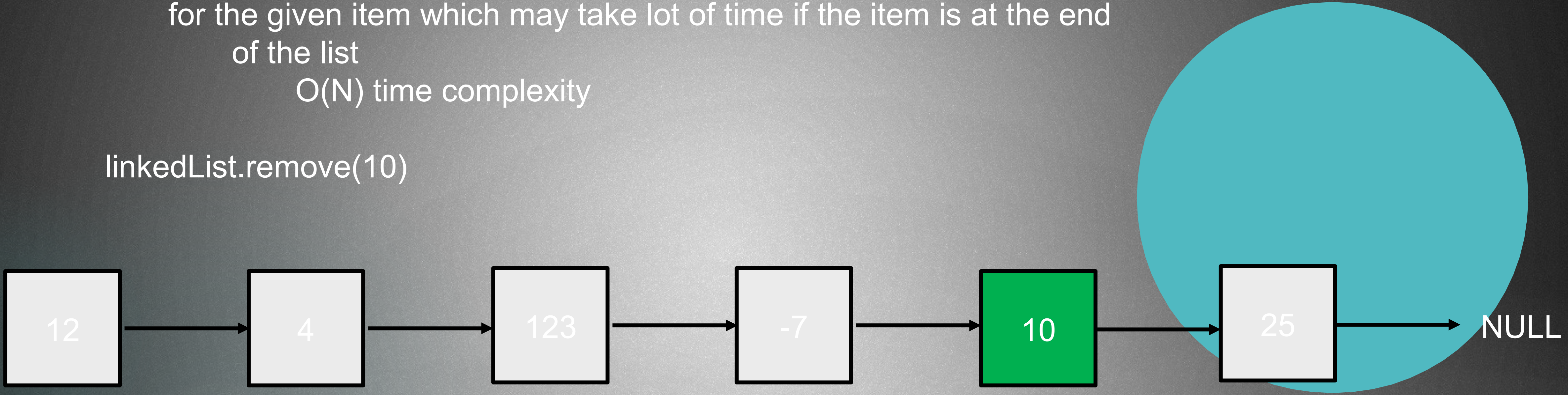
Is it the item we are looking for? NO, keep going

Linked list operations   remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



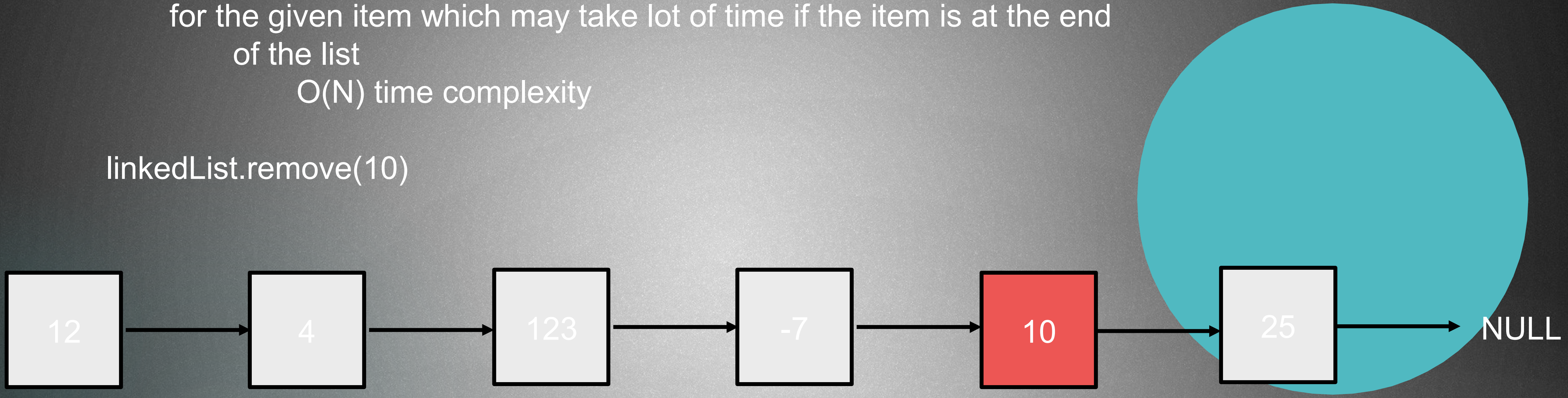
Is it the item we are looking for? YES

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



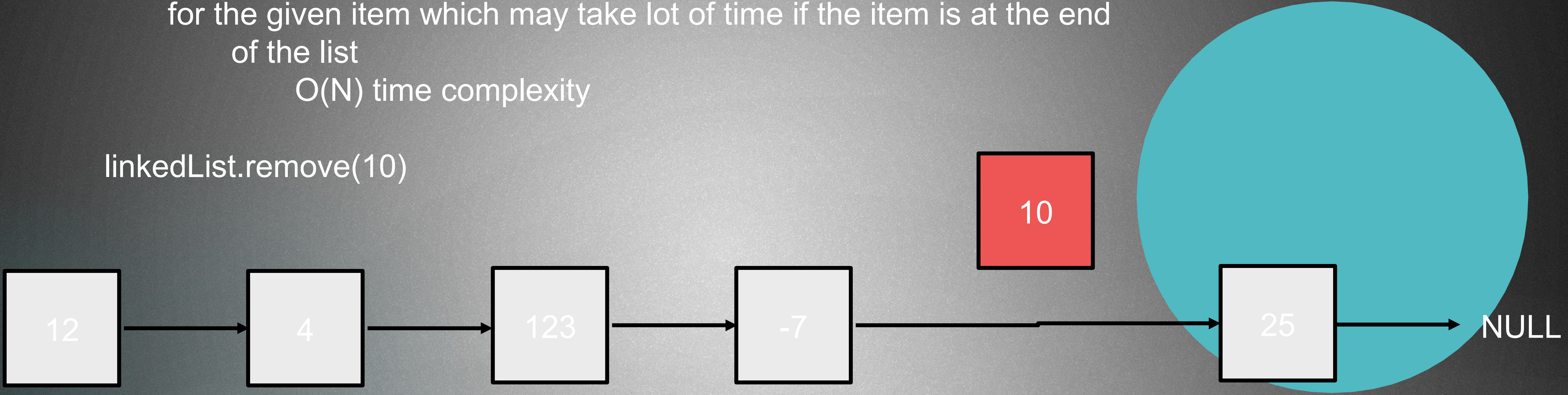
Is it the item we are looking for? YES, we just have to update the references: the previous node should point to the next node

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



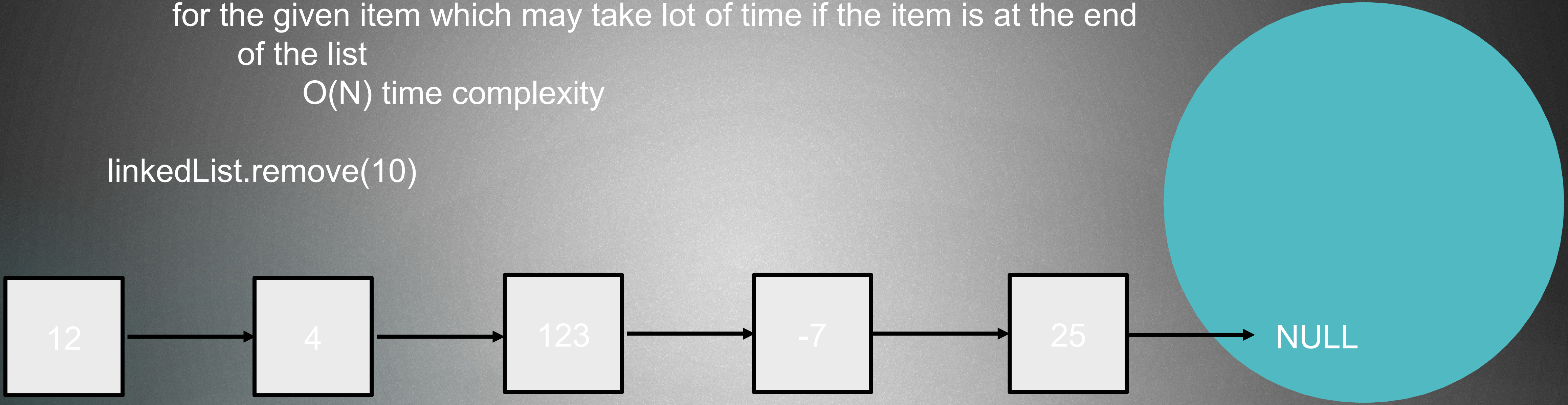
Is it the item we are looking for? YES, we just have to update the references: the previous node should point to the next node

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



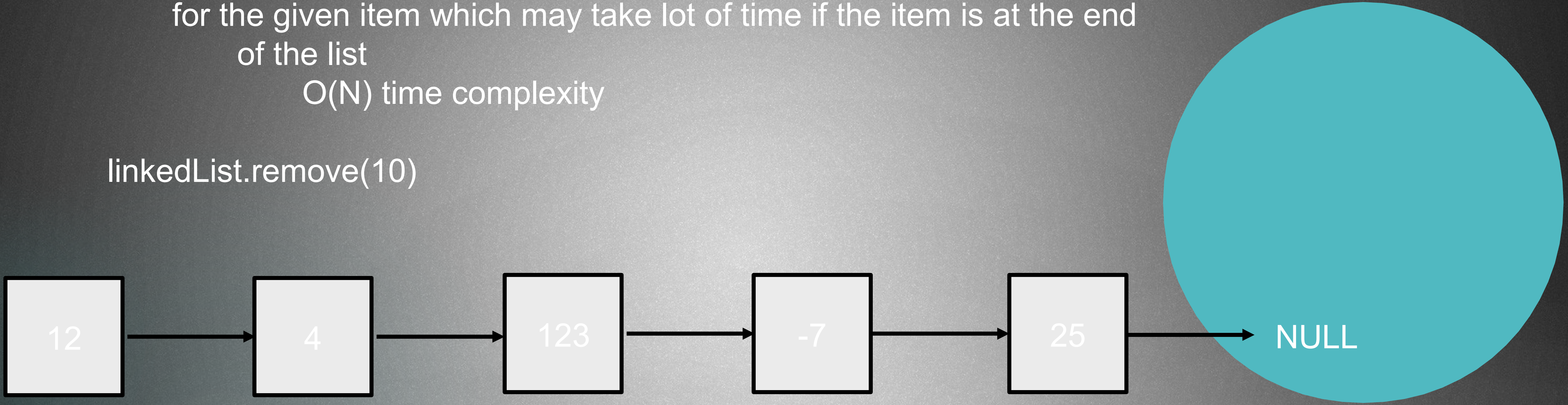
Is it the item we are looking for? YES, we just have to update the references: the previous node should point to the next node

Linked list operations remove

Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

linkedList.remove(10)



SO: we have managed to get rid of node with data 10, but we had to traverse the list ... O(N) time complexity

Linked list operations remove

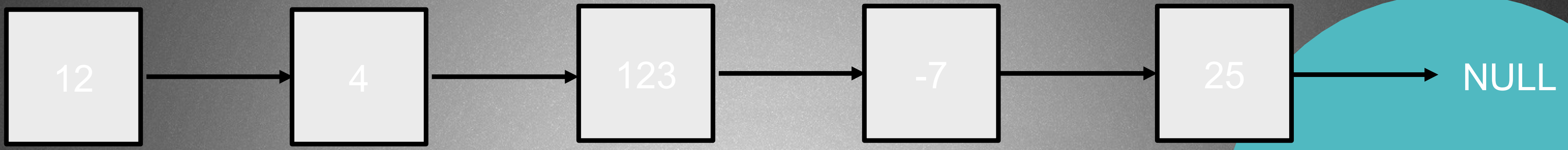
Remove item at a given point of the list is not always very fast: we have to search for the given item which may take lot of time if the item is at the end of the list

O(N) time complexity

Remove items at the beginning: O(1)

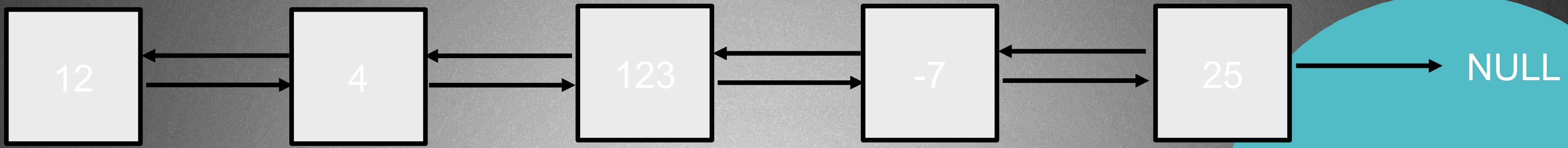
Remove items at given positions: O(N) in the main

Problems with linked lists



We can get from 4 to 25 because we just have to hop to the next nodes BUT we can not go from 25 to 4 because the references are in the opposite directions !!!

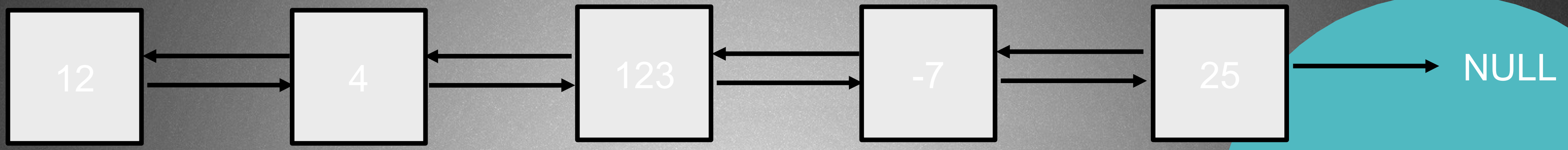
Problems with linked lists



We can get from 4 to 25 because we just have to hop to the next node BUT we can not go from 25 to 4 because the references are in the opposite directions !!!

Solution: doubly linked list → Node class has two references, one pointing to the next node, one pointing to the previous node

Problems with linked lists



We can get from 4 to 25 because we just have to hop to the next node
BUT we can not go from 25 to 4 because the references are in the
opposite directions !!!

Solution: doubly linked list → Node class has two references, one pointing to the next node, one pointing to the previous node

OK we can get from everywhere to everywhere BUT it is not so memory friendly, we have to store lots of references

BUT there is no need to track the previous node during traversal !!!

Linked lists VS arrays

1.) Search

- Search operation yields the same result for both data structure
- ArrayList search operation is pretty fast compared to the LinkedList search operation
- We can use random access with arrays: getItem(int index) which is O(1) time complexity
- LinkedList performance is O(N) time complexity
- So the conclusion: ArrayList is better for this operation
- Why?
- ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list
- On the other hand LinkedList requires the traversal through all the items for searching an element

2.) Deletion

- LinkedList remove operation takes O(1) time if we remove items from the beginning and usually this is the case
- ArrayList: removing first element (so at the beginning) takes O(N) time, removing the last item takes O(1) times
- But on average: we have to reconstruct the array when removing
- So the conculsion: LinkedList is better for this operation
- Why?
- LinkedList basically operates with pointers: removal only requires change in the pointer location which can be done very fast

3.) Memory management

- Arrays do not need any extra memory
- LinkedLists on the other hand do need extra memory because of the references / pointers
- So in this aspect: arrays are better, they are memory friendly !!!

	LinkedList	Arrays
Search	O(N)	O(1)
Insert at the start	O(1)	O(N)
Insert at the end	O(N)	O(1)
Waste space	O(N)	0

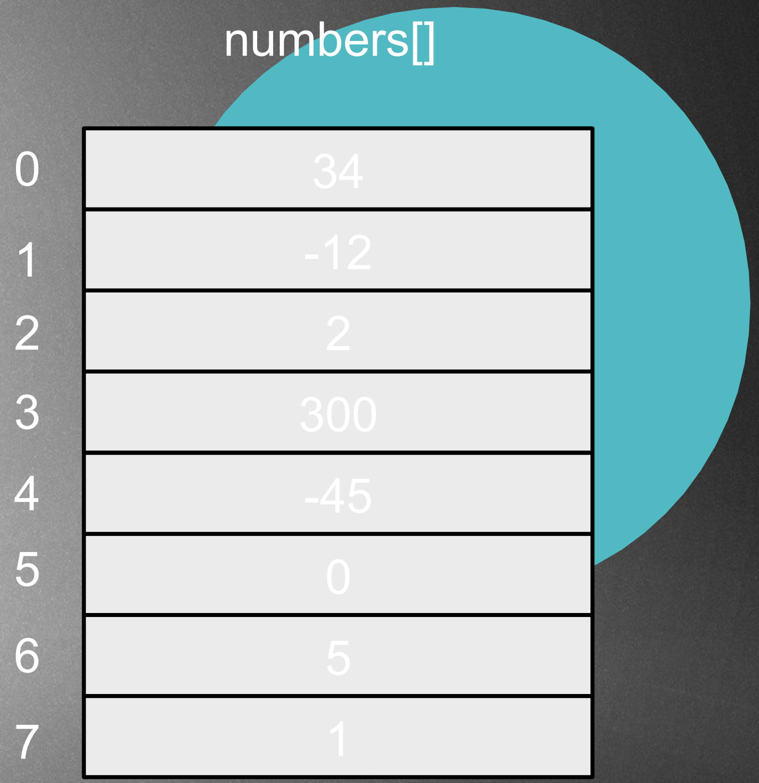
ARRAYS

ARRAY

Arrays: data structures

A collection of elements / values
each identified by an array index or key !!!

- index starts at zero
- because of the indexes: random access
  is possible



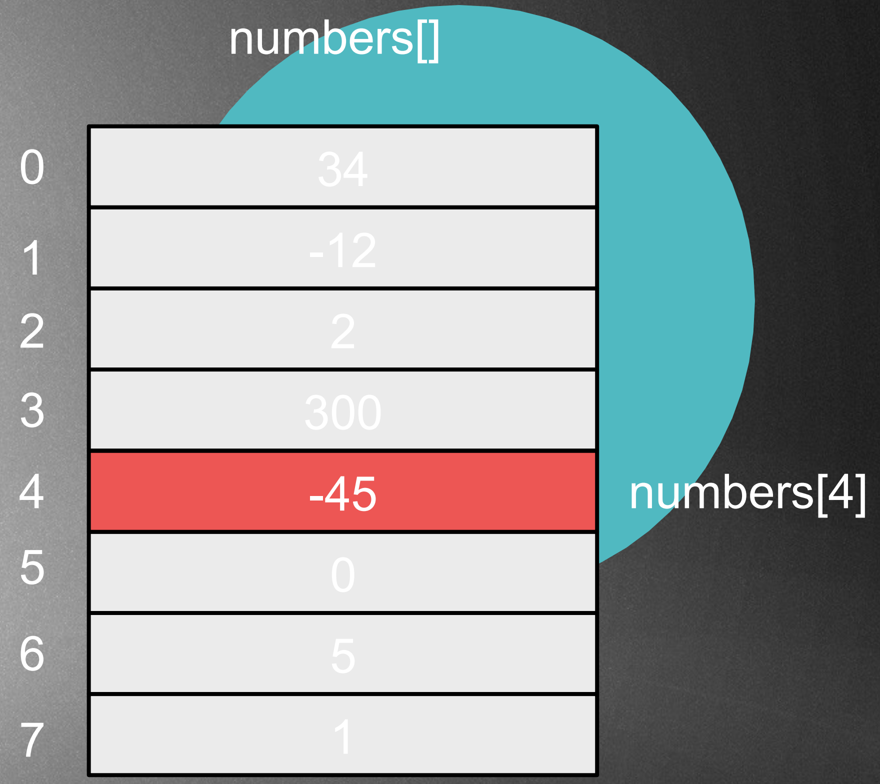
numbers[]

0	34
1	-12
2	2
3	300
4	-45
5	0
6	5
7	1

Arrays: data structure

A collection of elements / values
each identified by an array index or key !!!

- index starts at zero
- because of the indexes: random access
  is possible

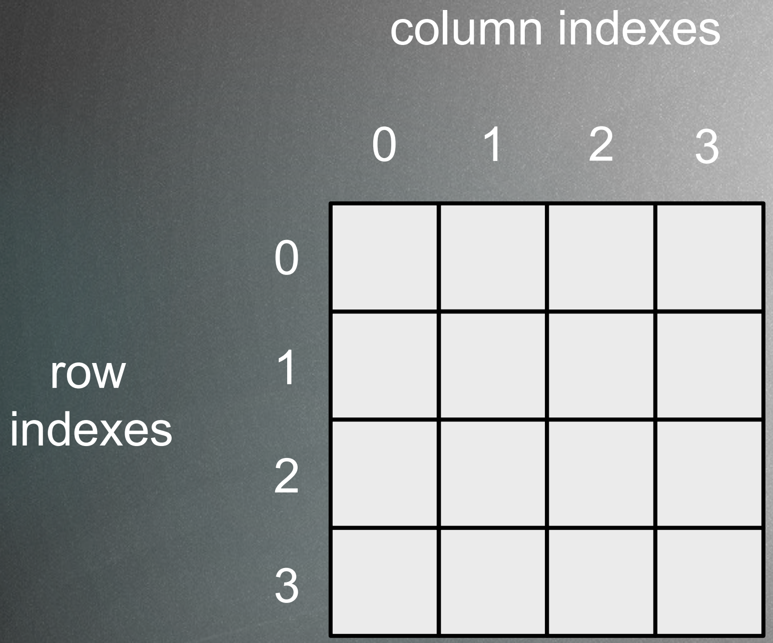


numbers[]

0	34
1	-12
2	2
3	300
4	-45
5	0
6	5
7	1

numbers[4]

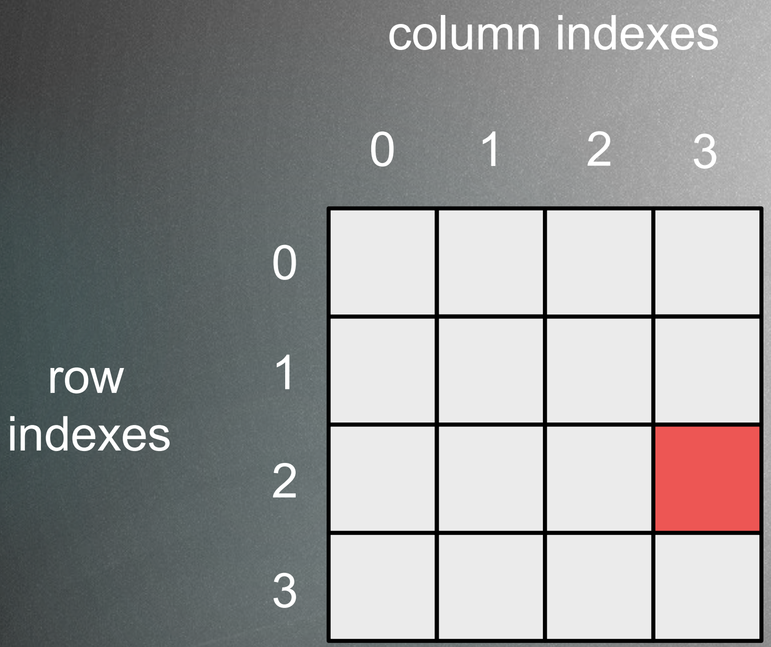
Multidimensional arrays: it can prove to be very important in mathematical related computations (matrixes)



numbers[][] two dimensional array

First paramter: row index
Second parameter: column index

Multidimensional arrays: it can prove to be very important in mathematical related computations (matrixes)



column indexes

0 1 2 3

row indexes

0 1 2 3

number[2][3]

numbers[][] two dimensional array

First paramter: row index
Second parameter: column index

Arrays

- Arrays are data structures in order to store items of the same type
- We use indices as keys !!!
- Arrays can have as many dimensions as we want: one or two dimensional arrays are quite popular
- For example: storing a matrix → two dimensional array
- Dynamic array: when the size of the array is changing dynamically
- Applications: lookup tables / hashtables, heaps

Advantages

- We can use random access because of the keys: getItem(int index) will return the value with the given key very fast // **O(1)**
- Very easy to implement and to use
- Very fast data structure
- We should use arrays in applications when we want to add items over and over again and we want to take items with given indexes!!! ~ it will be fast

Disadvantages

- We have to know the size of the array at compile-time: so it is not so dynamic data structure
- If it is full: we have to create a bigger array and have to copy the values one by one // reconstructing an array is **O(N)** operation
- It is not able to store items with different types

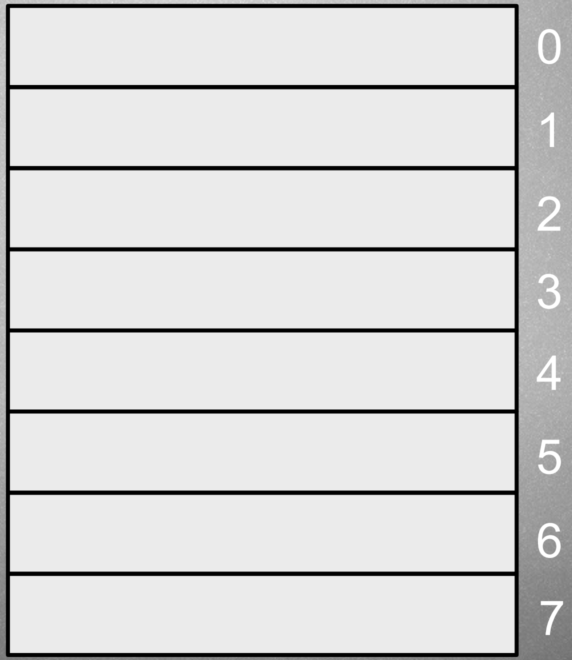
Arrays operation: add

We can keep adding values to the array as far as the array is not full

Arrays operation: add

We can keep adding values to the array as far as the array is not full

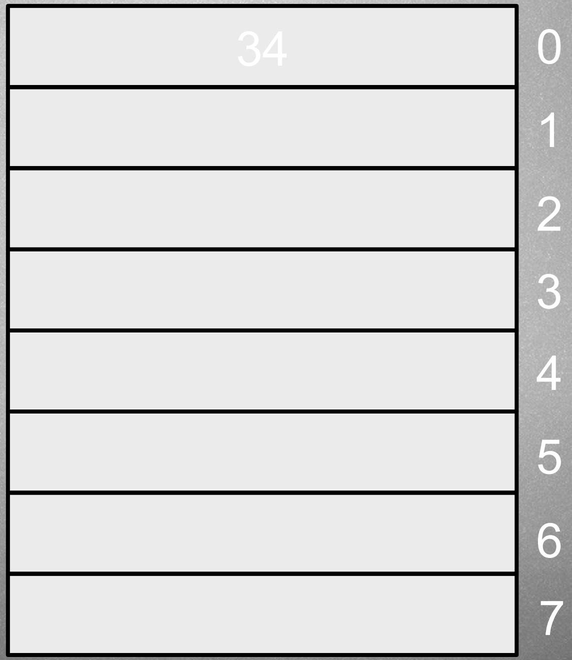
add(34)



Arrays operation: add

We can keep adding values to the array as far as the array is not full

add(34)

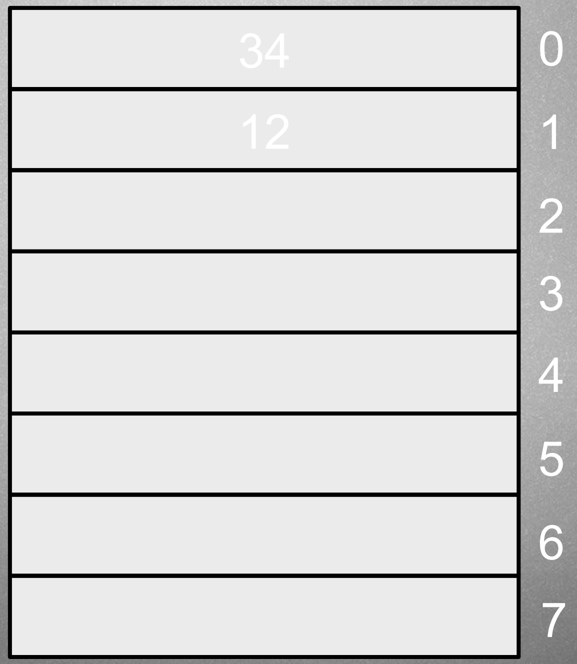


34	0
	1
	2
	3
	4
	5
	6
	7

<u>Arrays operation</u>: add

We can keep adding values to the array as far as the array is not full

add(12)

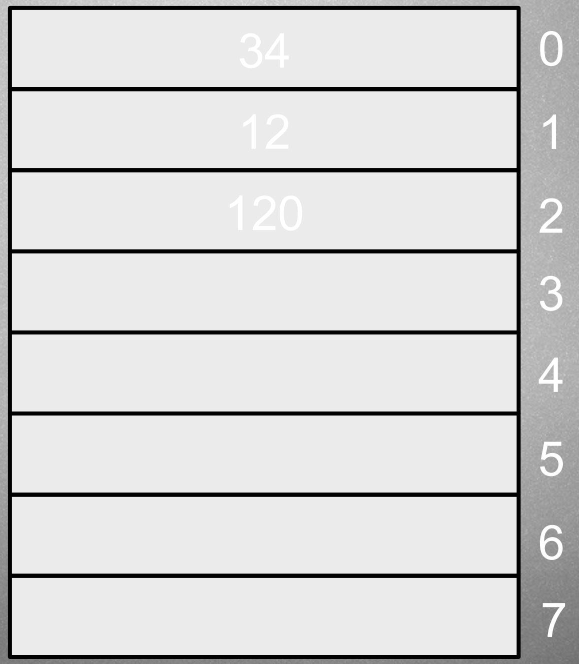


34	0
12	1
	2
	3
	4
	5
	6
	7

Arrays operation: add

We can keep adding values to the array as far as the array is not full

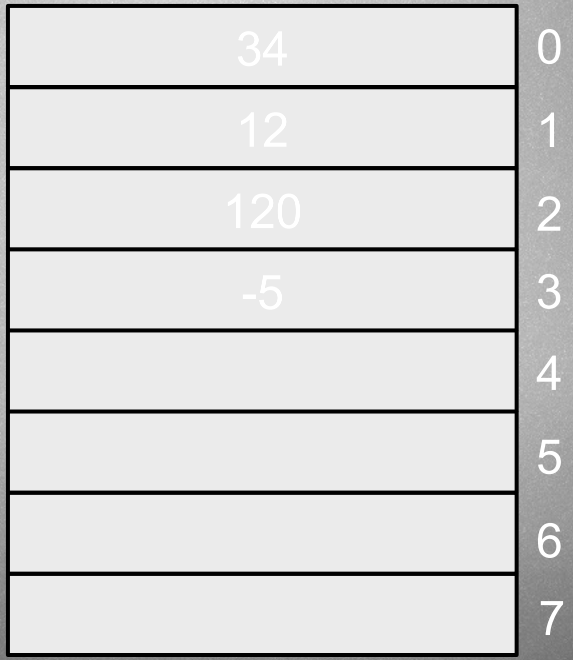
add(120)



Arrays operation: add

We can keep adding values to the array as far as the array is not full

add(-5)



34	0
12	1
120	2
-5	3
	4
	5
	6
	7

Arrays operation: add

We can keep adding values to the array as far as the array is not full

34	0
12	1
120	2
-5	3
	4
	5
	6
	7

So: when adding new values to the list, we just have to insert it with the next index → very fast **O(1)** operation

Arrays operation: insert item

We would like to insert a given value with a given index

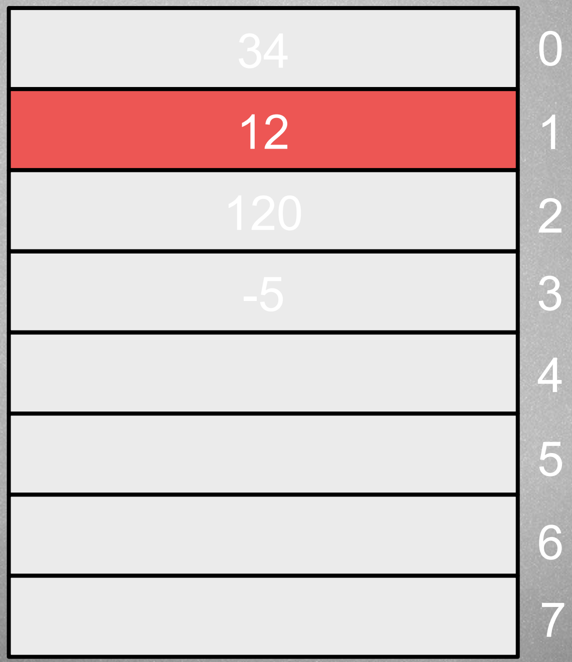
```
insert(23,1);
```

34	0
12	1
120	2
-5	3
	4
	5
	6
	7

Arrays operation: insert item

We would like to insert a given value with a given index

insert(23,1);



34 0

12 1

120 2

-5 3

4

5

6

7

<u>Arrays operation</u>: insert item

We would like to insert a given value with a given index

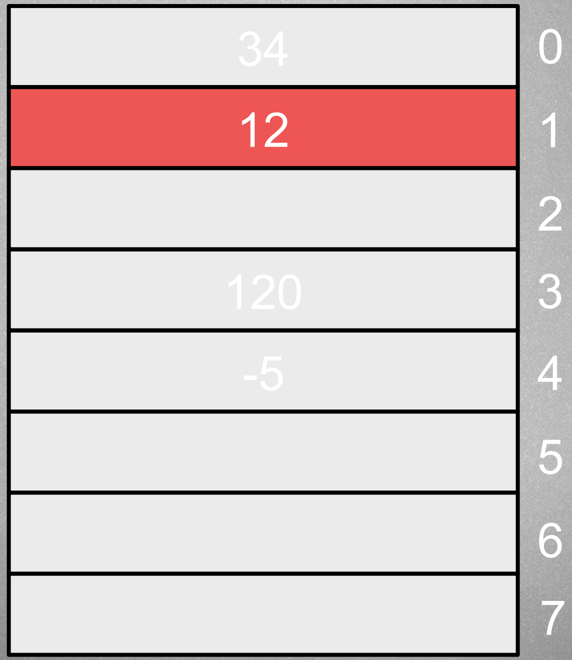
insert(23,1);

34	0
12	1
120	2
	3
-5	4
	5
	6
	7

Arrays operation: insert item

We would like to insert a given value with a given index

insert(23,1);

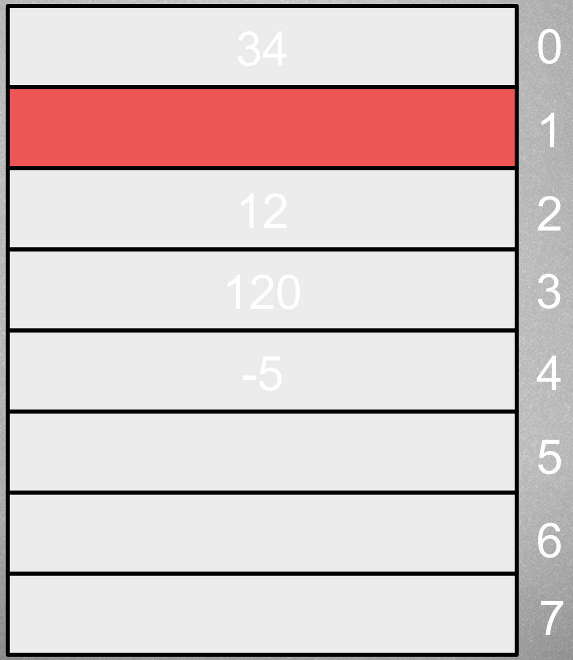


34	0
12	1
	2
120	3
-5	4
	5
	6
	7

Arrays operation: insert item

We would like to insert a given value with a given index

insert(23,1);

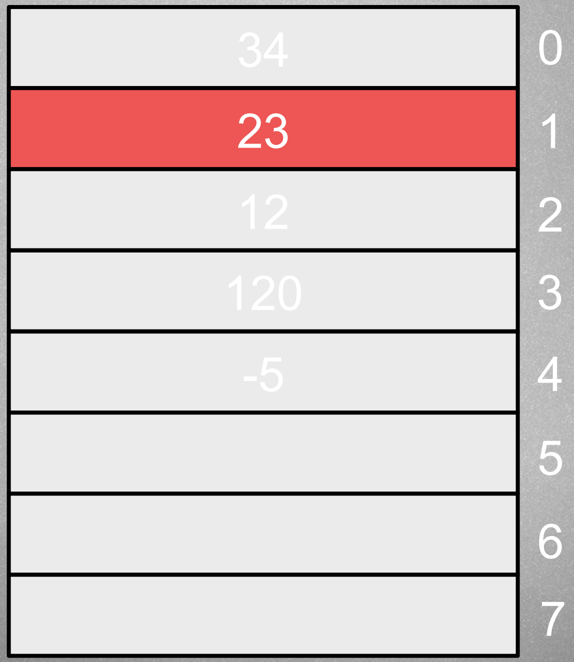


34	0
🟥	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: insert item

We would like to insert a given value with a given index

```
insert(23,1);
```

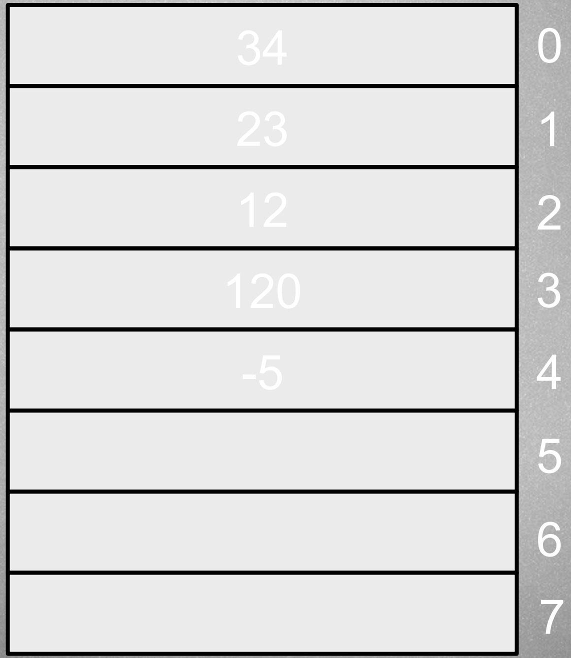


34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: insert item

We would like to insert a given value with a given index

insert(23,1);



34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

So: it is a bit more problematic, sometime we have to shift lots of values in order to be able to insert the new one !!! ~ **O(N)** time complexity

<u>Arrays operation</u>: insert item

We would like to insert a given value with a given index

Add new item:	**O(1)**
Insert item to a given index:	**O(N)**

Arrays operation: remove items

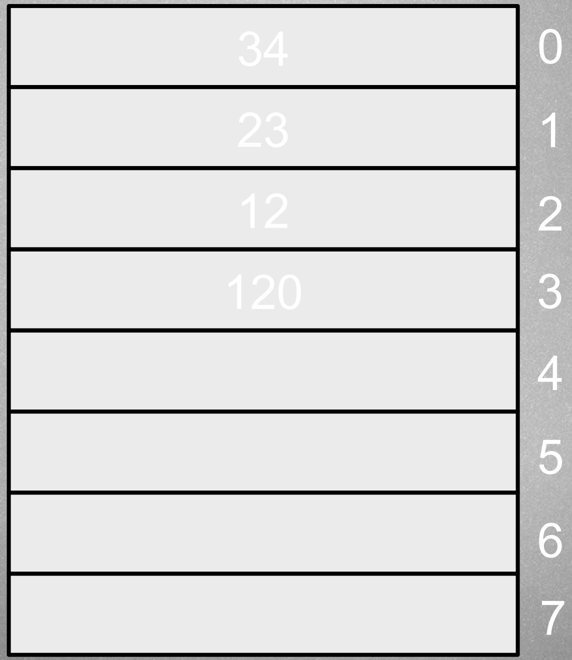
We would like to remove the last item, it is very simple, just remove it // **O(1)** time complexity

34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: remove items

We would like to remove the last item, it is very simple, just remove it // **O(1)** time complexity

removeLast();

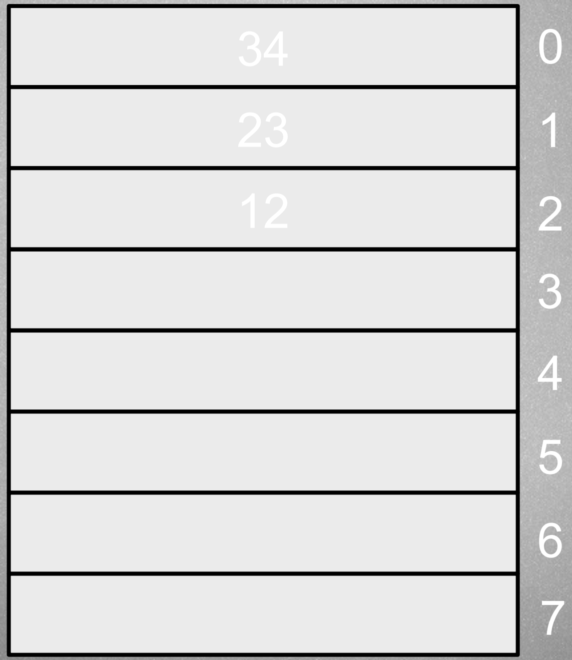


34	0
23	1
12	2
120	3
	4
	5
	6
	7

Arrays operation: remove items

We would like to remove the last item, it is very simple, just remove it // **O(1)** time complexity

removeLast();

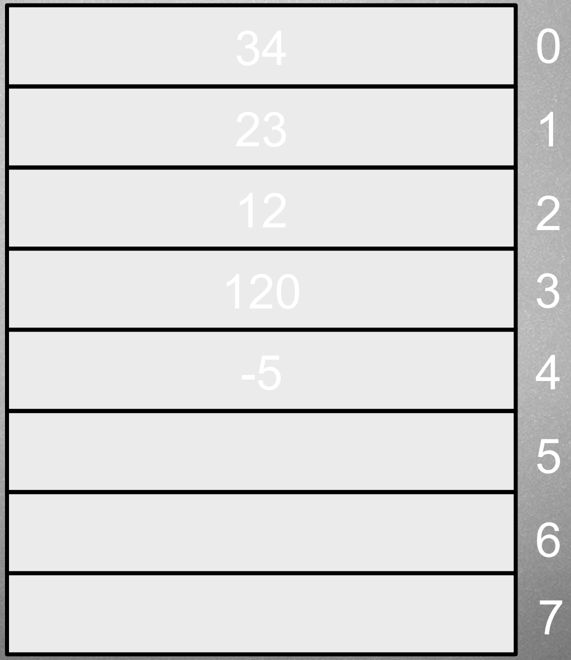


34	0
23	1
12	2
	3
	4
	5
	6
	7

Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity



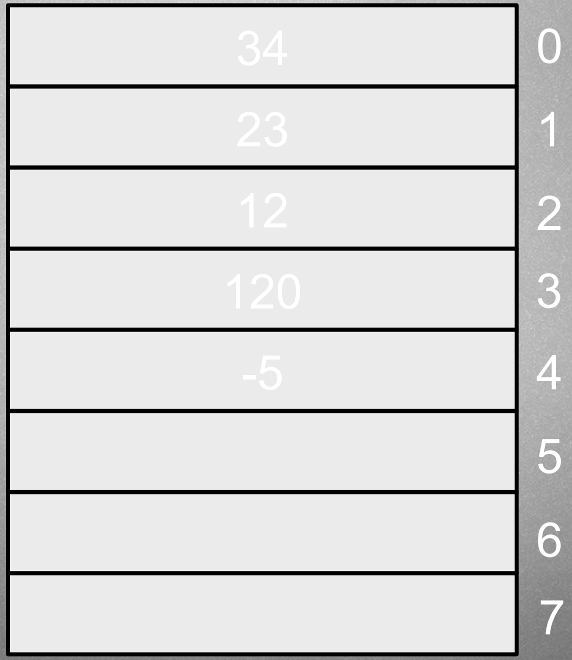
34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

<u>Arrays operation</u>: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity

remove(1);



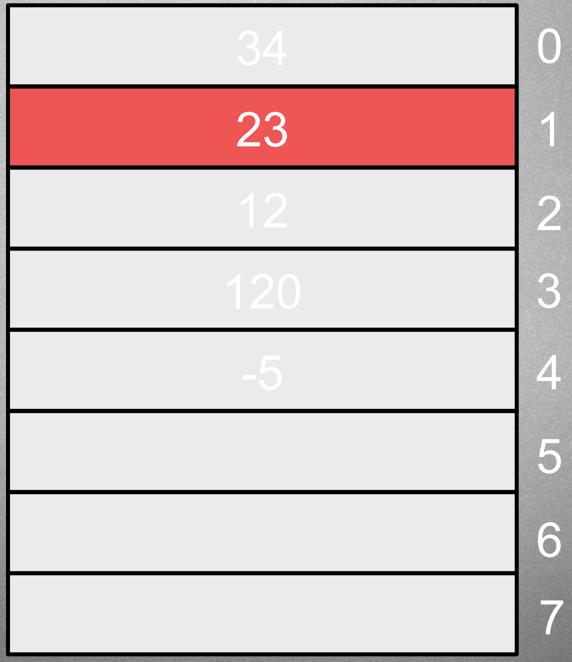
34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity

remove(1);



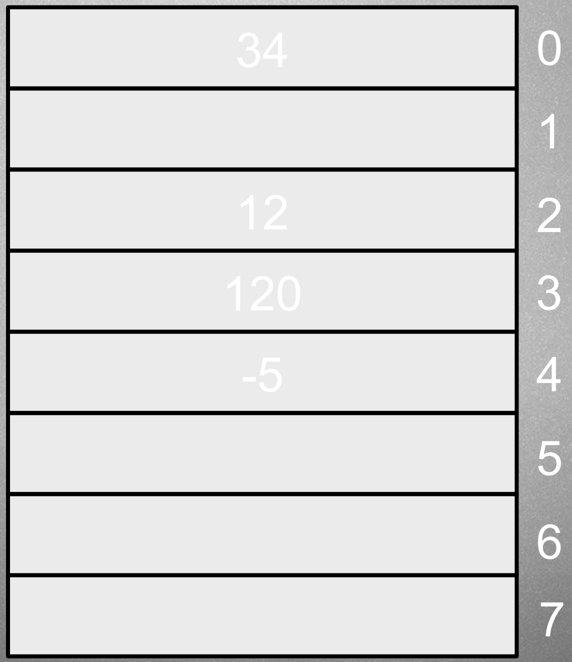
34	0
23	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity

remove(1);

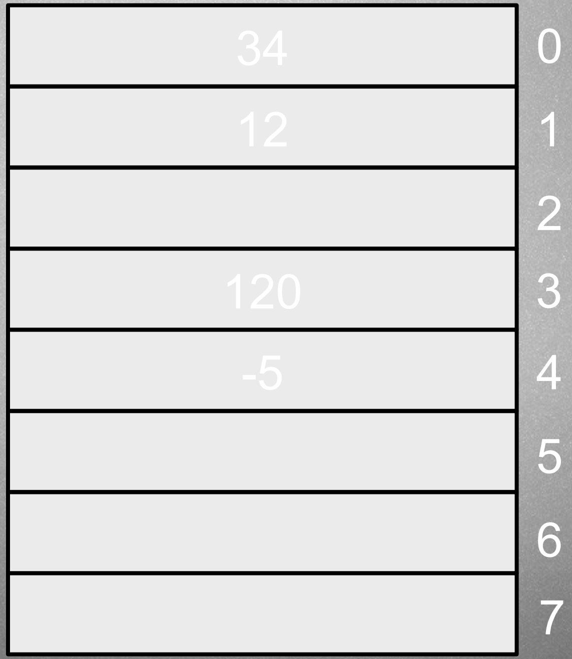


34	0
	1
12	2
120	3
-5	4
	5
	6
	7

Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items // **O(N)** time complexity

remove(1);



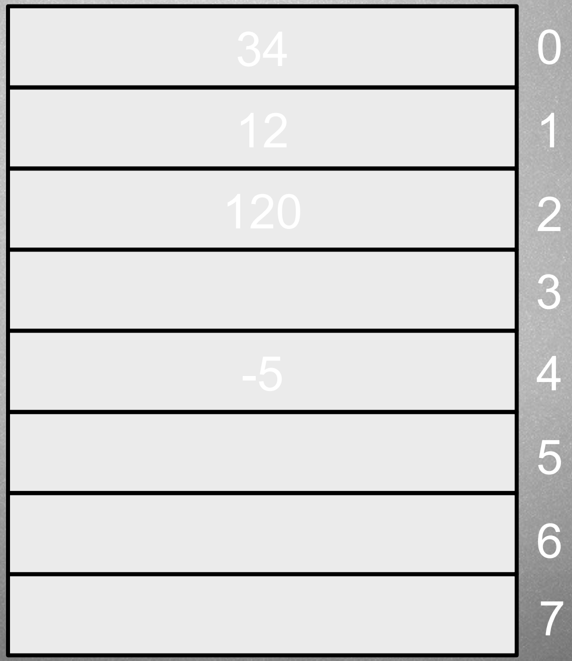
34	0
12	1
	2
120	3
-5	4
	5
	6
	7

<u>Arrays operation</u>: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity

remove(1);

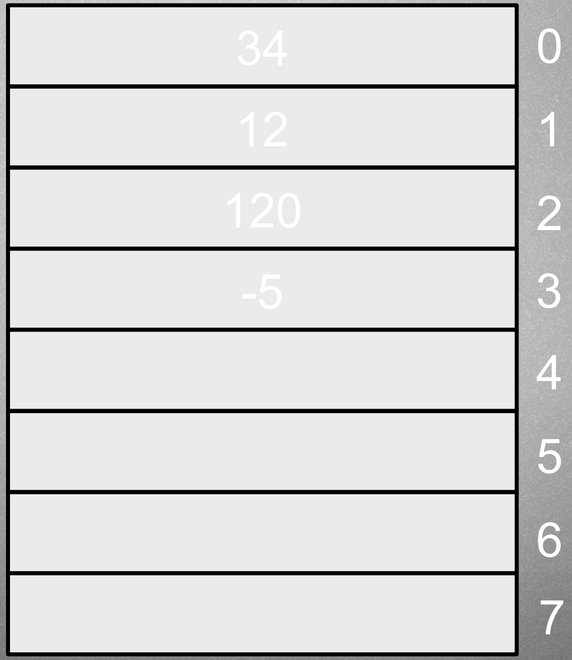


Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// **O(N)** time complexity

remove(1);



34	0
12	1
120	2
-5	3
	4
	5
	6
	7

So: overall complecity will be linear **O(N)**

Arrays operation: remove items with given index

We would like to remove a value with a given index, it is not that simple, we may have to shift items

// O(N) time complexity

Removing last item:	**O(1)**
Removing f.e. middle item:	**O(N)**