Data Structures and Algorithms

ADT and Linear Data Structures

- ADT and Algorithms
- List
- Stack and Queue
- Set

Abstract Data Types (ADT) and Algorithms

Abstract Data Types

- **Abstract data types (ADT)**
  - Mathematical models of data structures, specify
    - The types of data stored
    - Operations supported
    - Types of the operation's parameters

- Focus is on ***what*** and not on ***how***
  - Point of view of the user

- In Java an ADT is *expressed* by an **interface** (or **abstract class**)

Abstract Data Types

- Specify:
  - **What** data structure is used and
  - **What** each operation results
  - Does **not** specify **implementation** details

- The ADT is realized by a data structure
  - implemented in one or more classes
  - Classes specify how the operations are performed

Algorithms

- A ***finite*** sequence of instructions, each of which has a clear meaning, that can be performed with a ***finite*** amount of effort in a ***finite*** length of time

- Three required properties:
  1. Unambiguous (clear meaning)
  2. Executable (can be performed)
  3. Terminating (finite length of time)

Algorithms

An algorithm description contains:

- A set of **inputs** for which the algorithm is designed
- the **output** that results from the algorithm given the input provided

Summary

- ADTs describe conceptual models that can be applied in any programming language

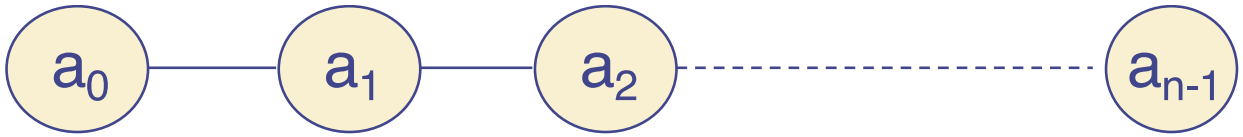
- Algorithms operate on ADT's to perform logical operations

List

- Sequences
- The List ADT
- Implementation
  - Array-based List
  - Linked List
- Computational Complexity

Sequence

- A Sequence $<a_0, a_2, \ldots, a_{n-1}>$ is a collection of zero or more *linearly ordered* elements of a given type



- We want to be able to insert, remove, find elements
  - Preserving an ordering

List ADT

- First data structure we see that allows to manipulate a sequence of elements

- Main operations:
  - insertFront/Back(e): inserts an element e
  - removeFront/Back (): removes an element
  - search(e): checks if an element is in the List
  - get(index): returns the element at position index
  - first/last: return the first/last element
  - isEmpty()/size()

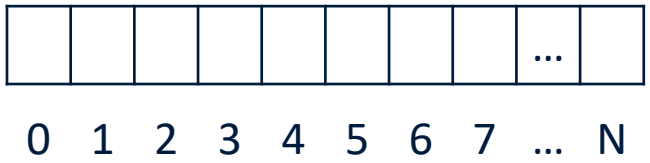
List ADT

```
public interface List<E> {
    void insertFront(E e);
    void insertBack(E e);
    E removeFront();
    E removeBack();
    boolean search(E e);
    E get(int index);
    E first();
    E last();
    boolean isEmpty();
}
```

List implementations

- Two main strategies

Array-based



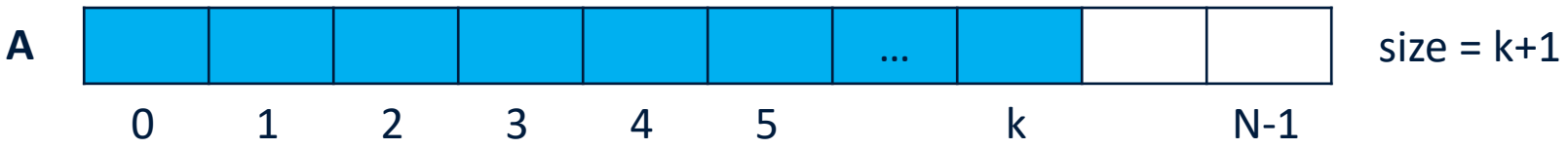
- Elements are stored in adjacent locations in memory
- Direct access if location is known
- The whole array must be allocated in memory

Linked-based

- Elements are stored in independent structures: Nodes
- The location of the first element is stored
- Each node contains the location of the next element
- We allocate memory space for nodes only when we need it

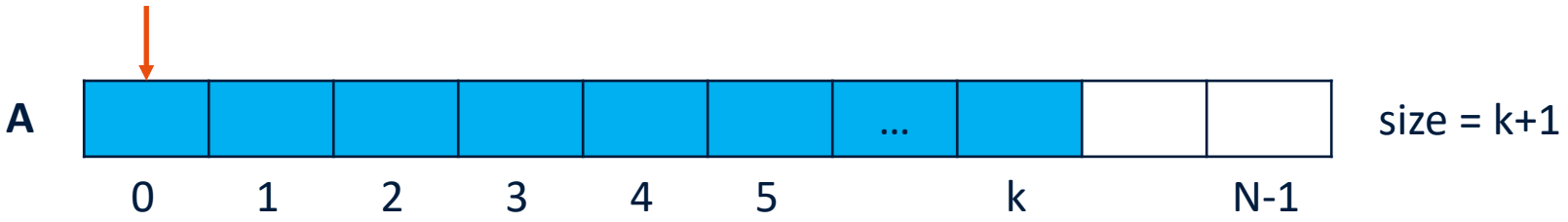
Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

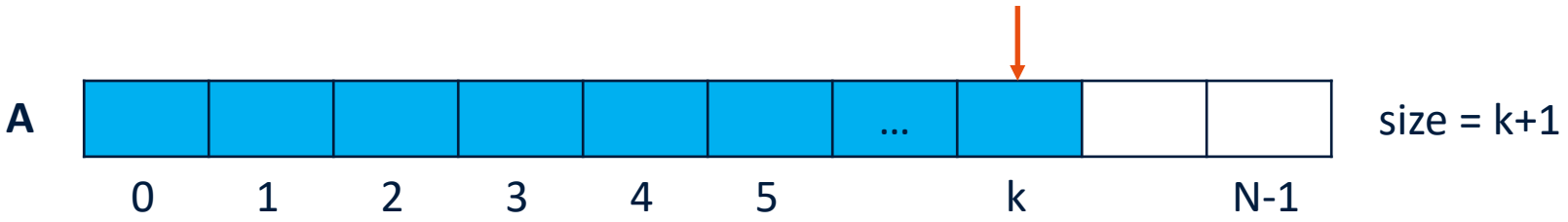


first() just return the elements at the position 0

- Complexity O(1) – constant time

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



last() just return the element at the position k=size-1

- Complexity O(1) – constant time

Array-based Lists

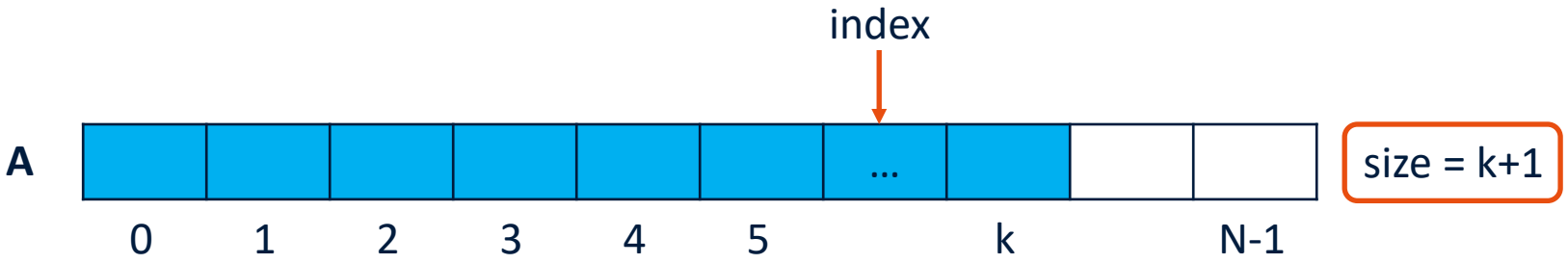
- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

isEmpty() checks if size is 0 or greater

- Complexity O(1) – constant time

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

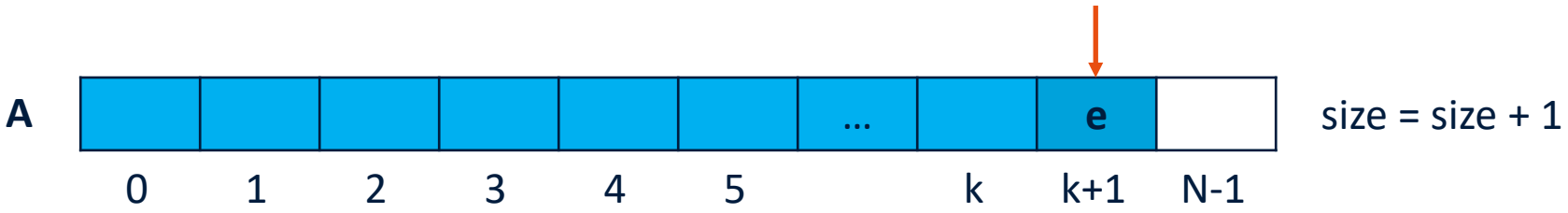


get(int index) return the elements at position indes

- Complexity O(1) – constant time

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

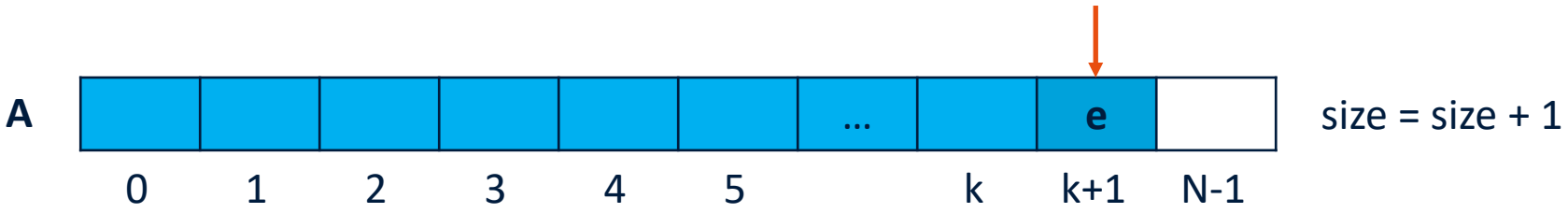


insertBack(E e):

- Insert e at position size and increments size
- Complexity O(1) – constant time

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

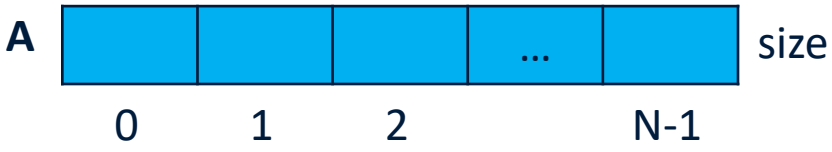


insertBack(E e):

- Insert e at position size a
- Complexity O(1) – consta

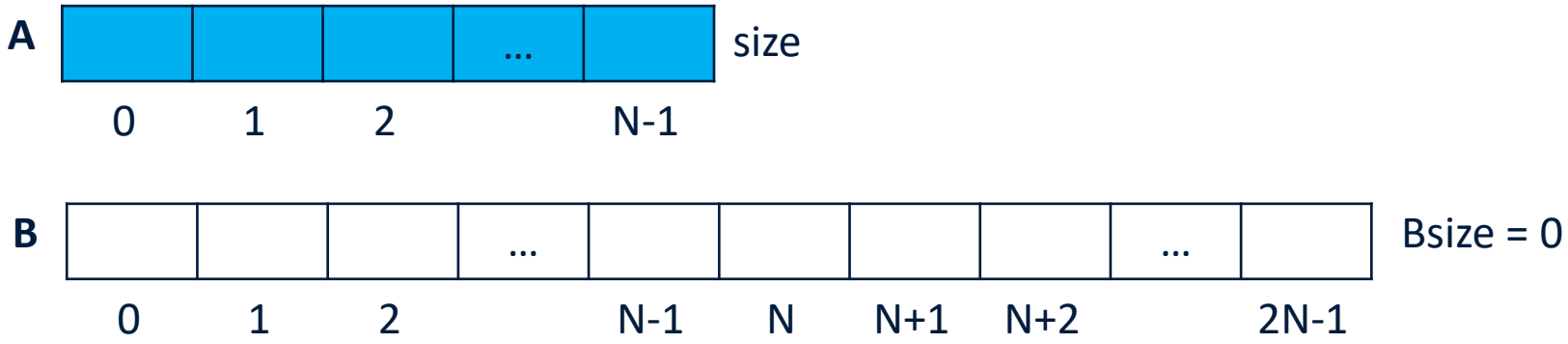
Resize Array

- If we the array is full, we cannot insert
- We check comparing ***N*** (total capacity) and ***size***
- If necessary, we need to increase the capacity



Resize Array

- If we the array is full, we cannot insert
- We check comparing ***N*** (total capacity) and ***size***
- If necessary, we need to increase the capacity

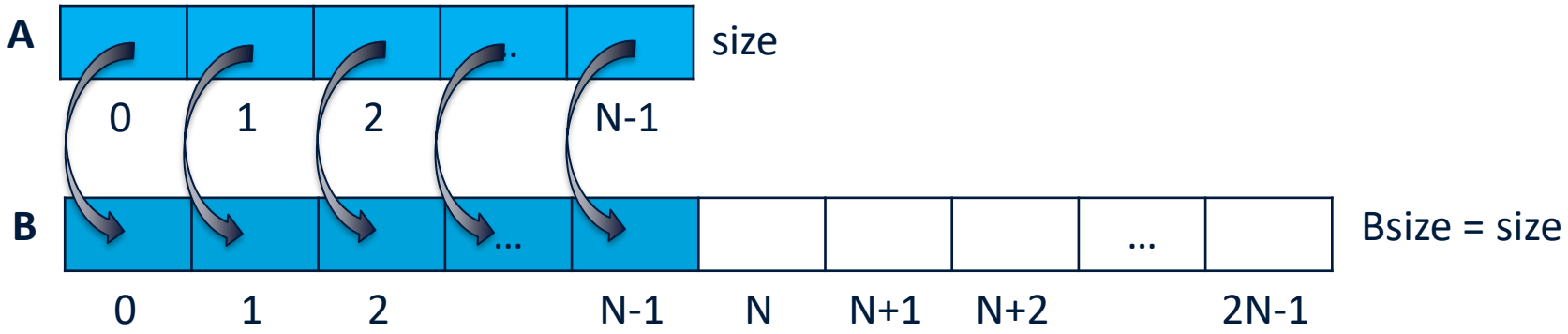


Resizing:

- We allocate a new array ***B*** with bigger size

Resize Array

- If we the array is full, we cannot insert
- We check comparing ***N*** (total capacity) and ***size***
- If necessary, we need to increase the capacity



Resizing:

- We allocate a new array ***B*** with bigger size
- We copy all elements from ***A*** to ***B***

Resize Array

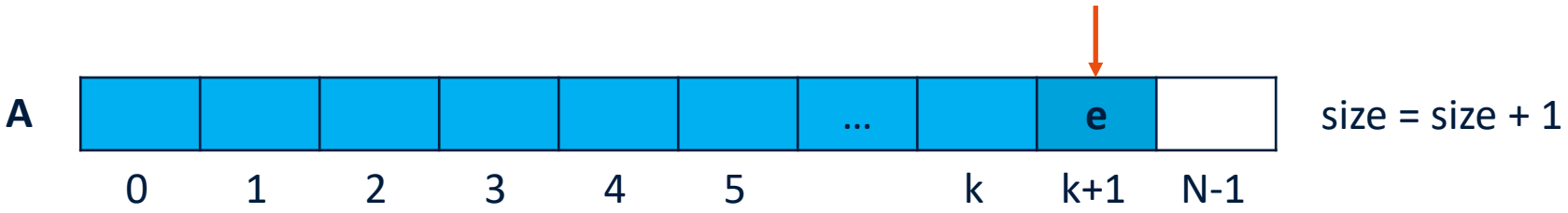
- If we the array is full, we cannot insert
- We check comparing ***N*** (total capacity) and ***size***
- If necessary, we need to increase the capacity

Resizing:

- We allocate a new array ***B*** with bigger size
- We copy all elements from ***A*** to ***B***
- We assign ***A*** = ***B***

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



insertBack(E e):

- Insert e at position size and increments size
- Complexity O(N) – linear time*

(*) in case we need to resize

Array-based Lists

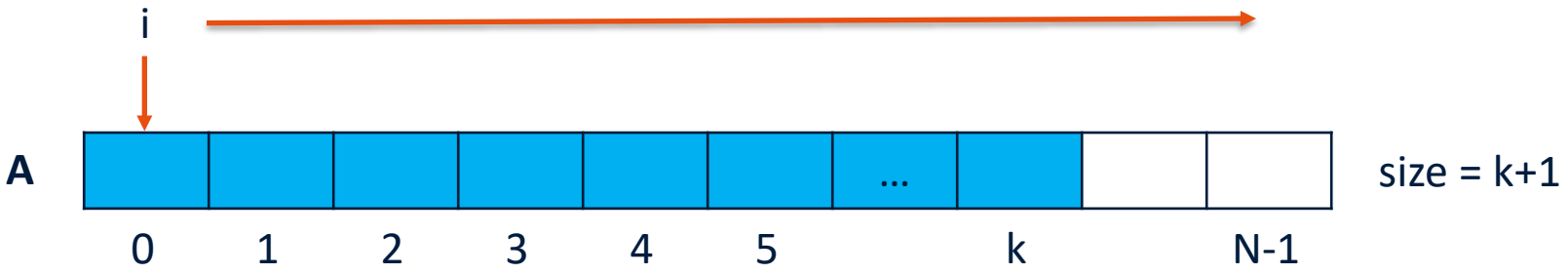
- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

removeBack():

- Remove at position size-1 and decrements size
- Complexity O(1) – constant time

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

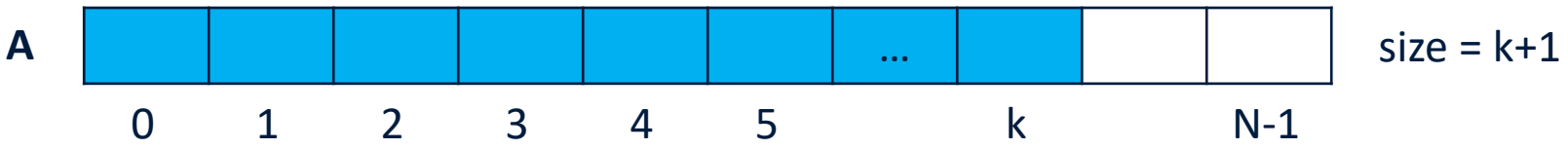


search(e)

- Iterates over the array until it finds e
- Complexity O(N) – linear time

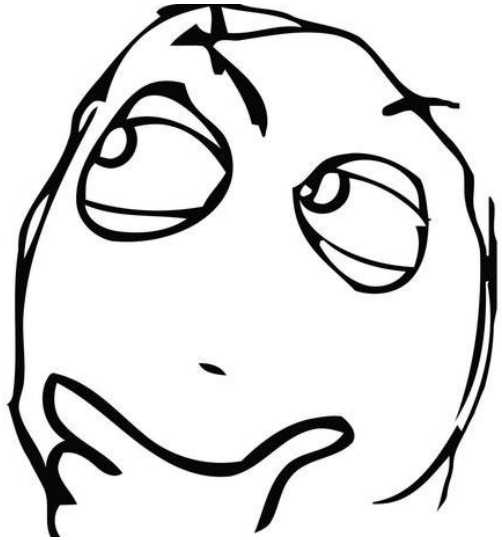
Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



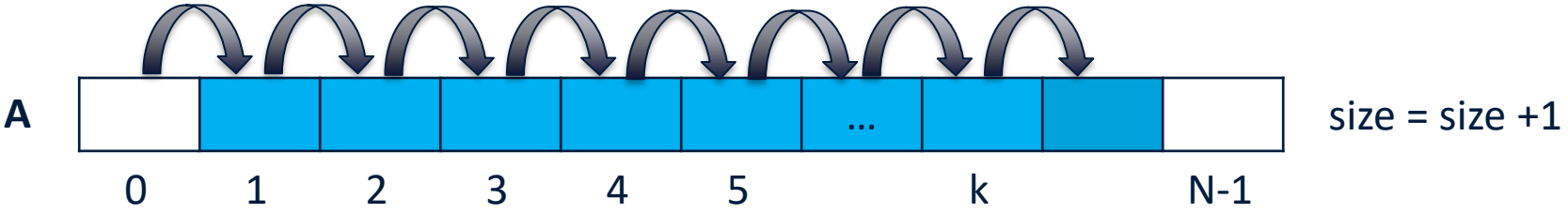
insertFront(e)

- We need to insert in position 0



Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

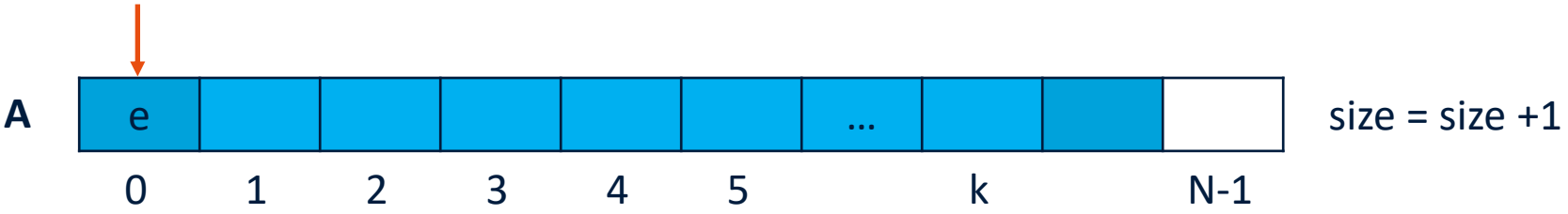


insertFront(e)

- We **shift right** all the elements

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



insertFront(e)

- We **shift right** all the elements
- Then, we insert e at position 0
- Complexity O(N) – linear time

Array-based Lists

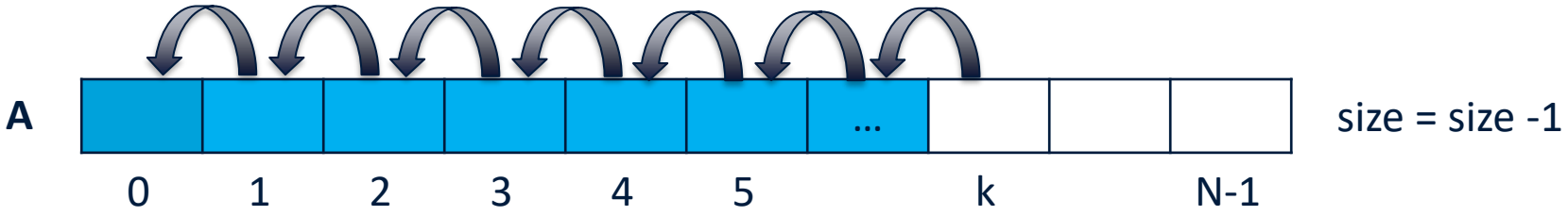
- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)

removeFront(e)

- We remove the element at position 0

Array-based Lists

- Use an array **A** of size ***N***
- A variable ***size*** keeps track of the size of the list (number of elements stored)



removeFront(e)

- We remove the element at position 0
- Then, we **shift left** all elements
- Complexity O(N) – linear time

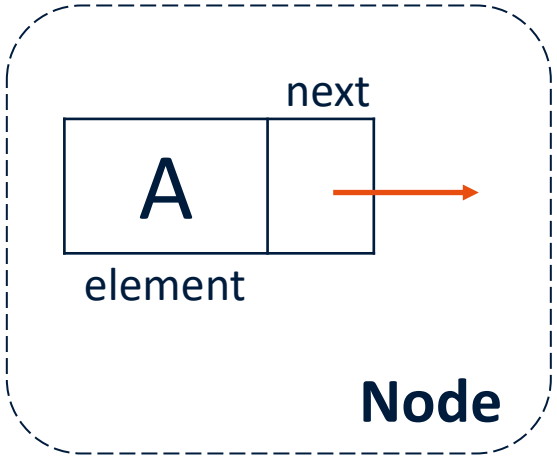
Linked Lists

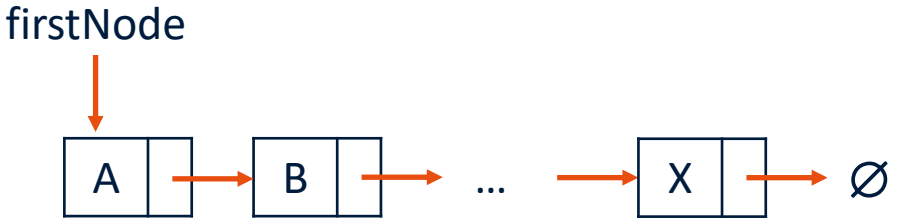
- Linked lists store elements in “nodes”
- Addressing by relative positions

- Two implementations of linked lists:
  - Linked List
  - Doubly-linked List

Linked Lists

- A linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node
- The linked list data structure maintains the reference of the first node





Node ADT

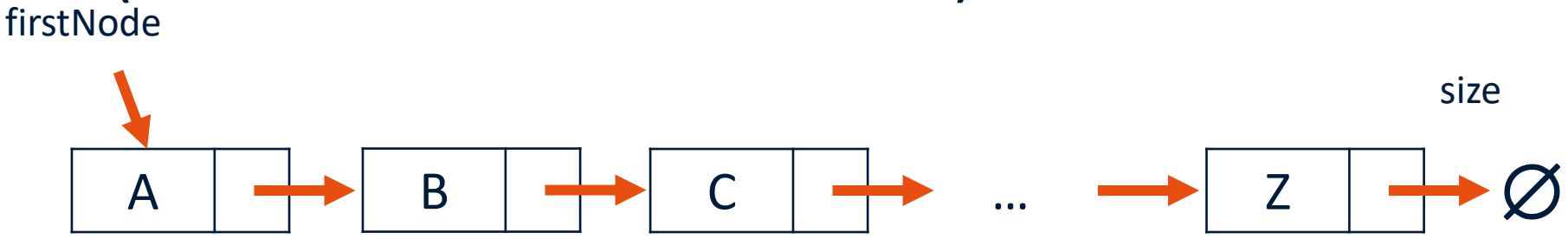
- A Node encapsulates the element and the reference of the next Node in the list

- The simplest way to define a node

```
public interface Node<E> {
    E getElement();
    Node<E> getNext();
}
```

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

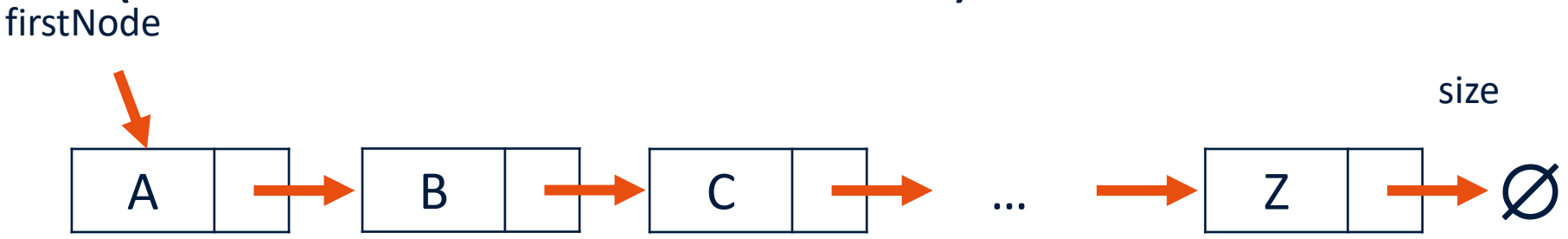


first() just return the element of the first Node

- Complexity O(1) – constant time

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

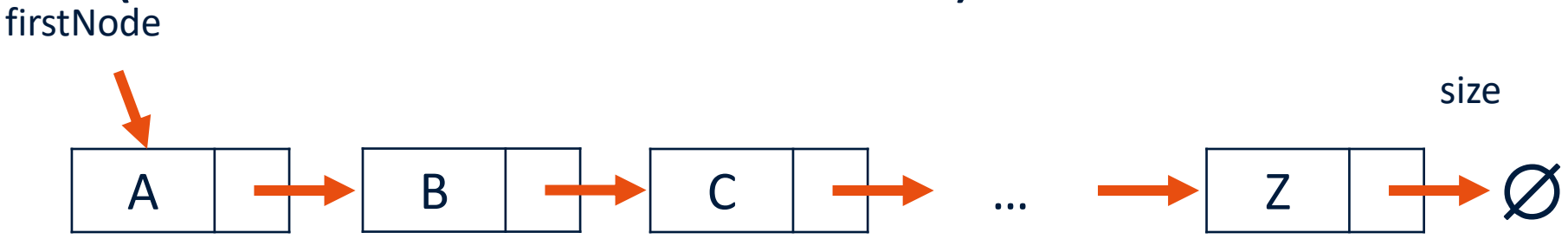


last() needs to iterate over the whole list to get the reference of the last

- Complexity O(N) – linear time
- O(1) if we keep also a reference for the last Node

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

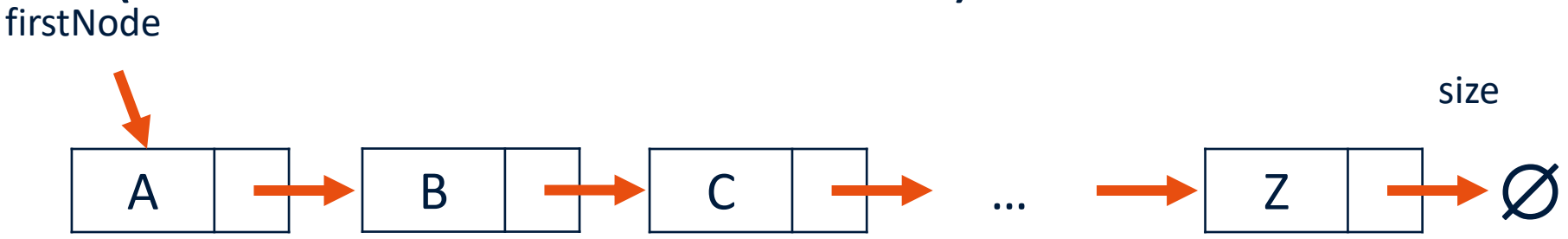


isEmpty() checks if size is 0 or greater

- Complexity O(1) – constant time

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

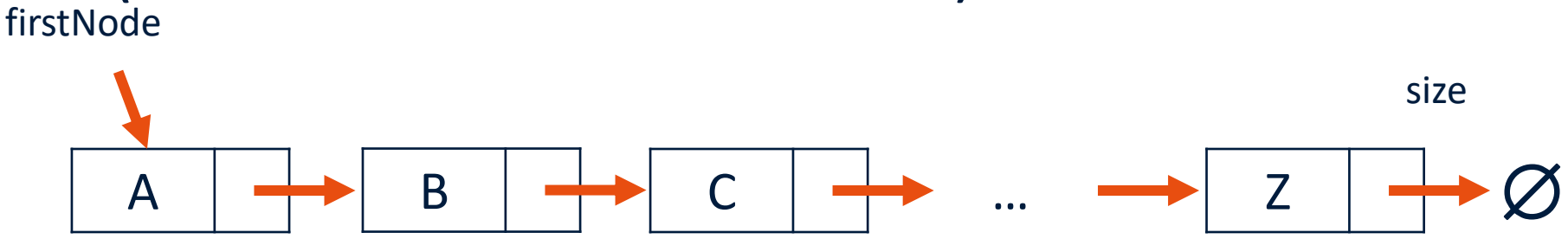


get(int index)

- needs to iterate over the whole list until it reaches the index-th element
- Complexity O(N) – linear time

Linked Lists

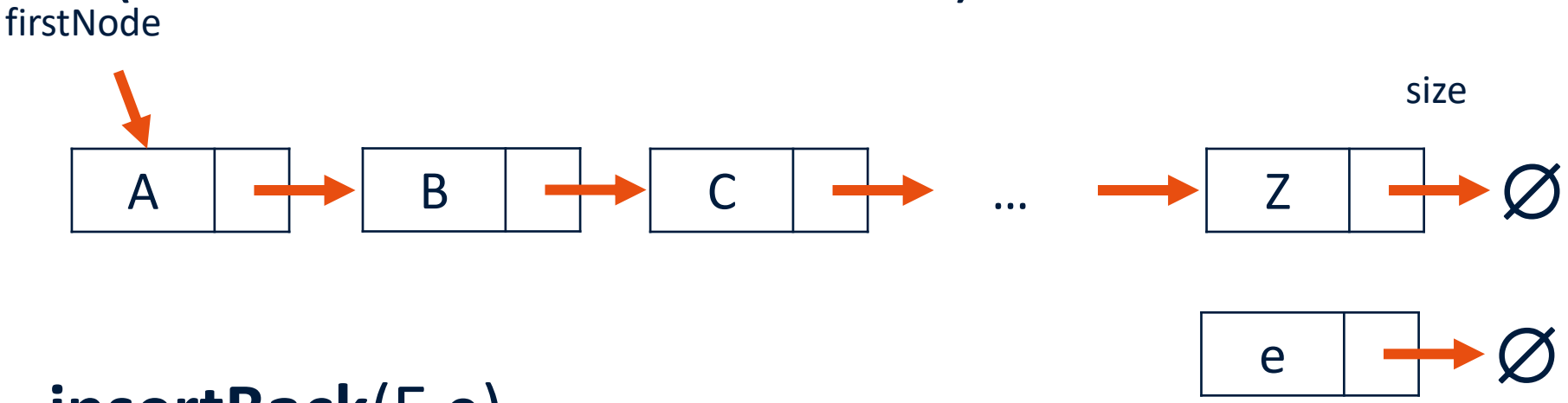
- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)



insertBack(E e)

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

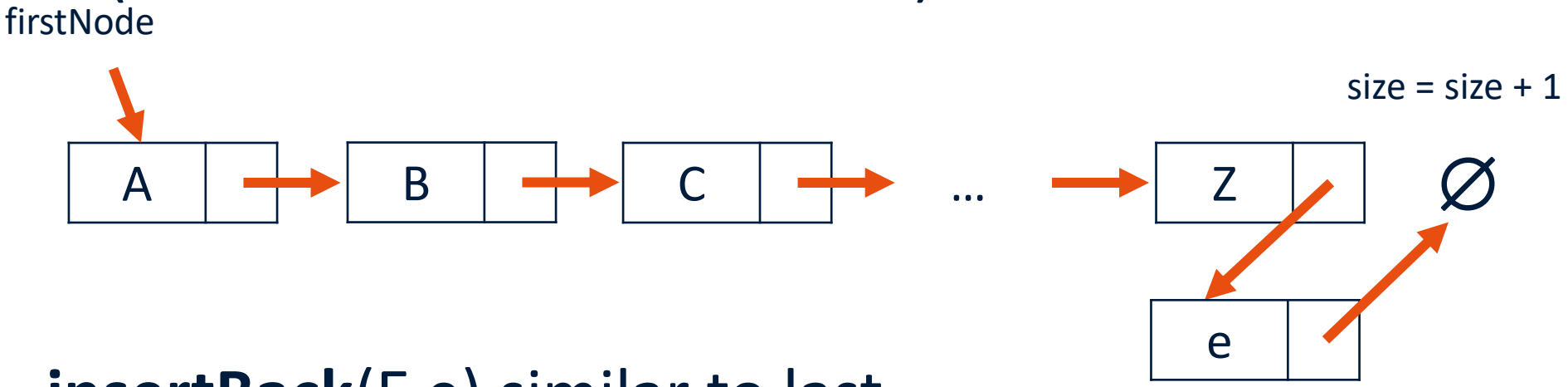


insertBack(E e)

- We create a Node having e as element

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

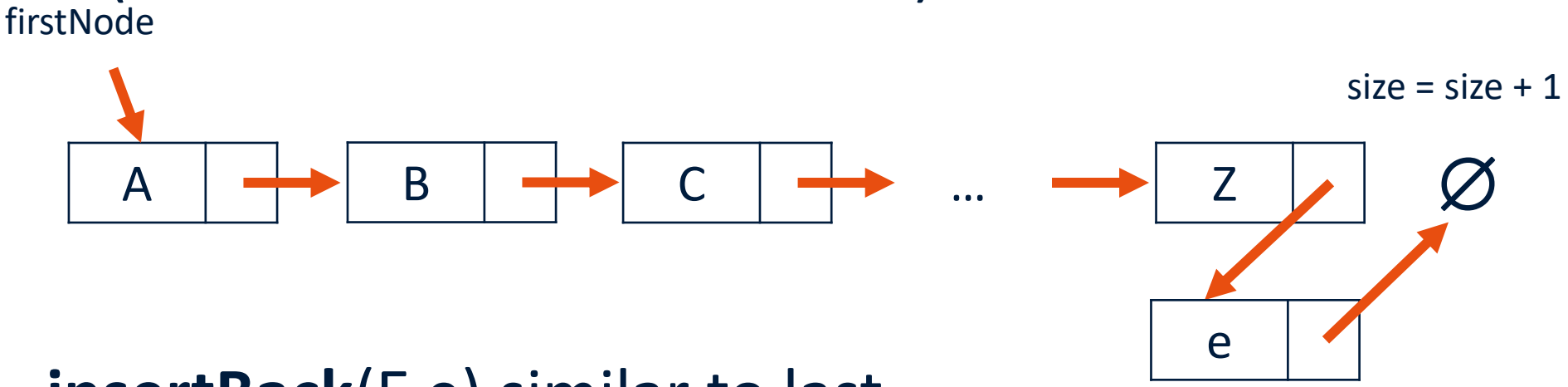


insertBack(E e) similar to last

- We create a Node having e as element
  - Then we iterate until the last element (Z)
  - We update references to put the new node at the end

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

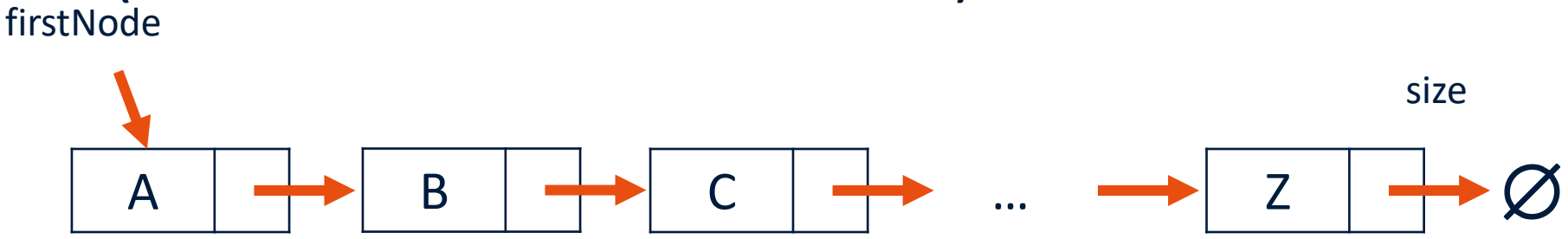


insertBack(E e) similar to last

- Complexity O(N) – linear time
- O(1) if we have the reference of the last element

Linked Lists

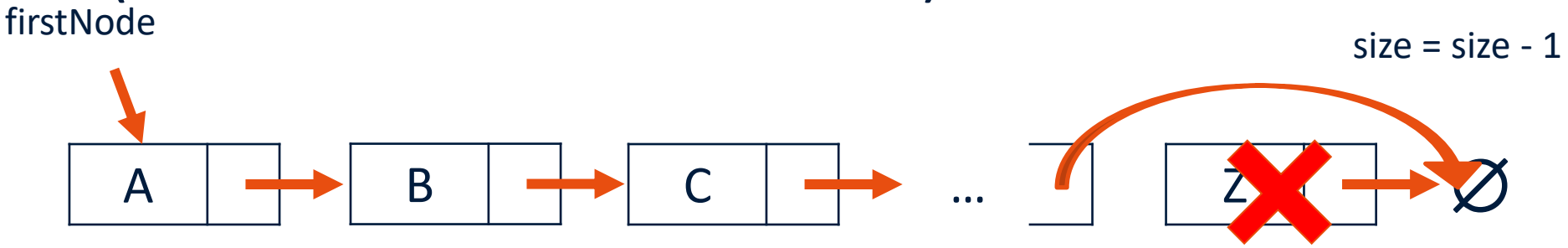
- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)



removeBack()

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

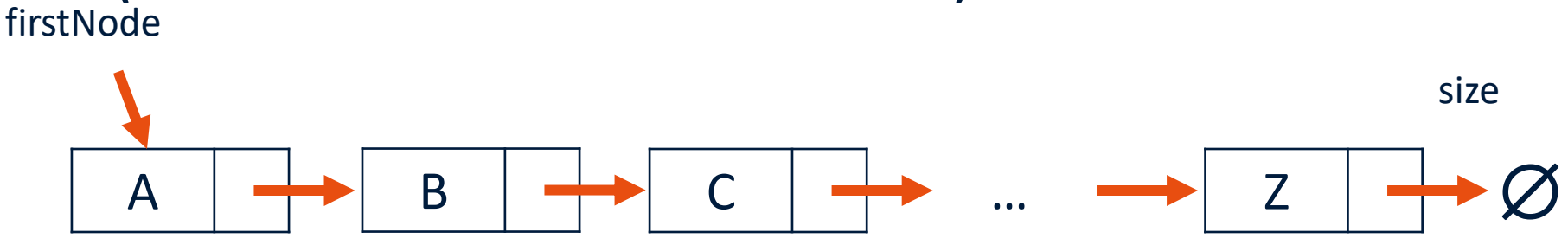


removeBack()

- needs to iterate over the list until the second-to-last element
  - Then update is ***next*** to ∅

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

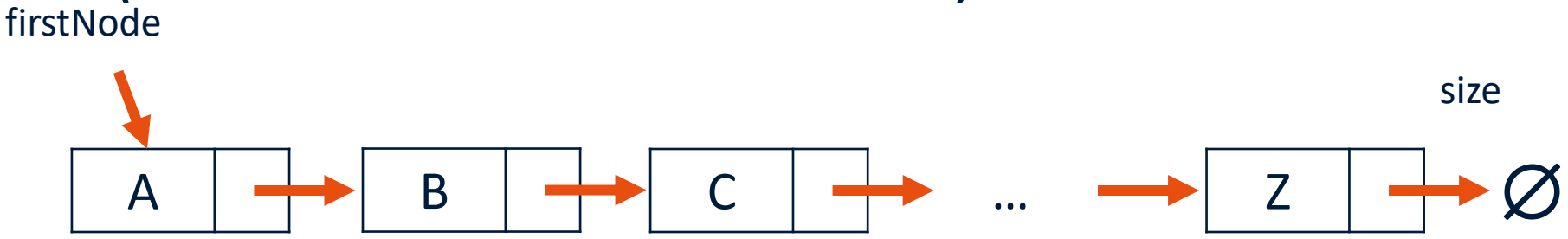


search(E e)

- Iterates over the array until it finds e
- Complexity O(N) – linear time

Linked Lists

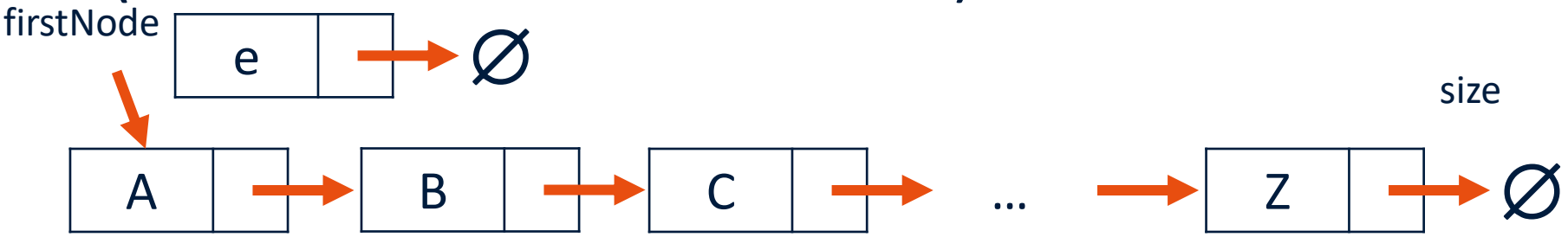
- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)



insertFront(E e)

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

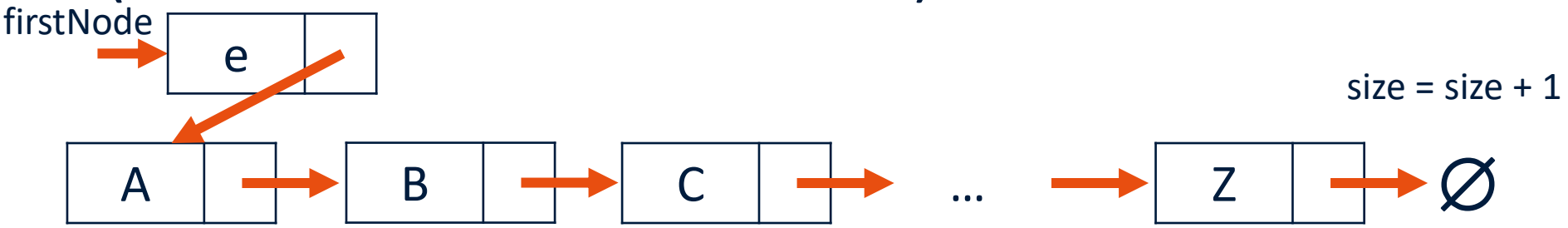


insertFront(E e)

- We create a Node having e as element

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

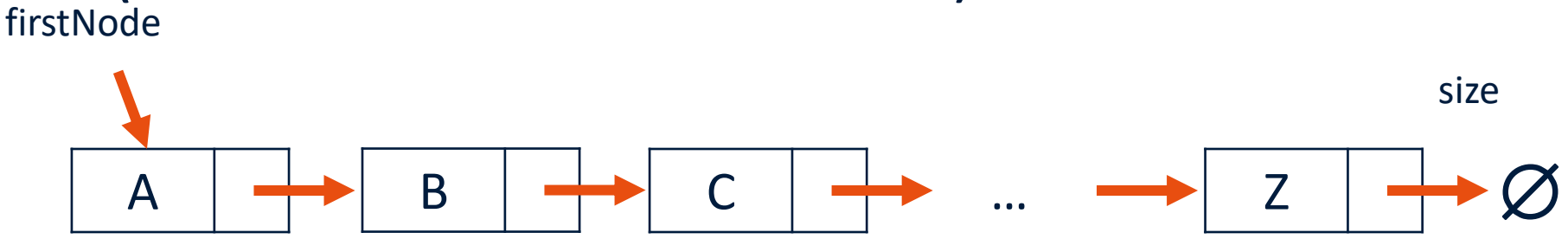


insertFront(E e)

- We create a Node having e as element
  - We update references for **firstNode** and its **next**
- Complexity O(1) – constant time

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)

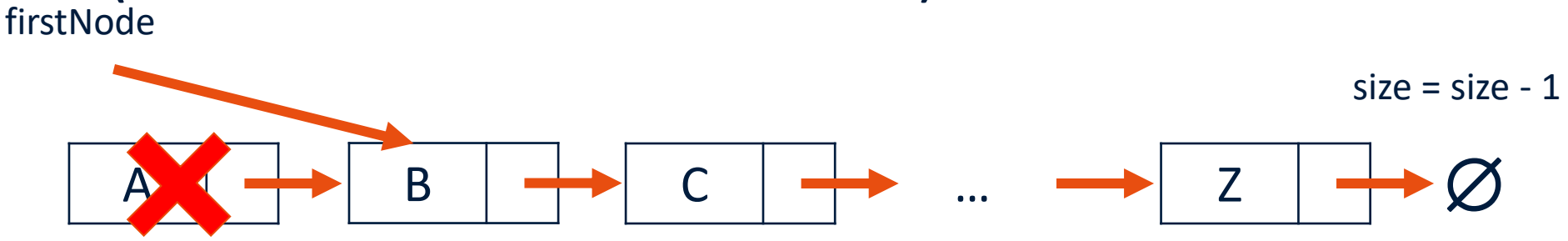


removeFront()

- Also, we only need to update the reference of the first element
- Complexity O(1) – constant time

Linked Lists

- Has a reference of the **firstNode**
- A variable ***size*** keeps track of the size of the list (number of elements stored)



removeFront()

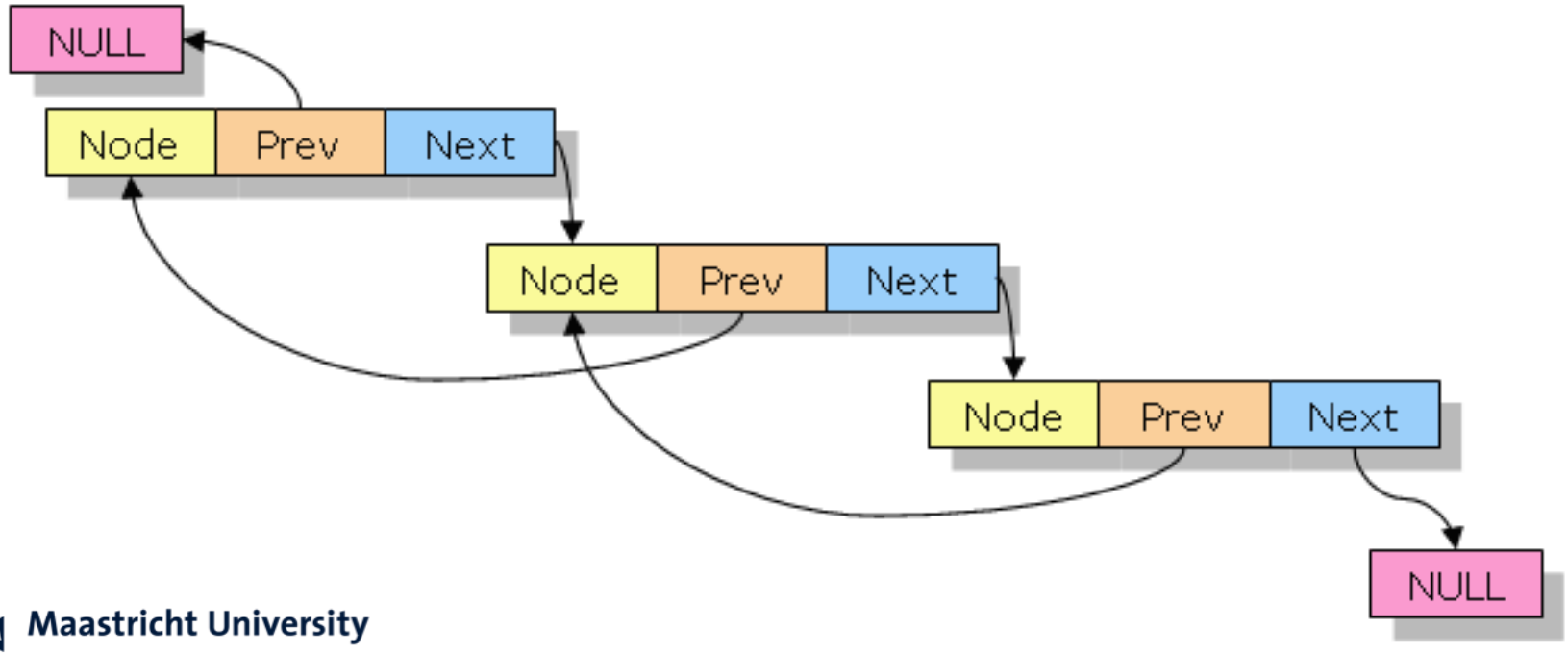
- Also, we only need to update the reference of the first element
- Complexity O(1) – constant time

Computational Time Complexity

Operation	Array-based List	Linked List
first()	O(1)	O(1)
last()	O(1)	O(1)*
isEmpty()	O(1)	O(1)
get(index)	O(1)	O(N)
search(e)	O(N)	O(N)
insertBack(e)	O(1)	O(N)
removeBack()	O(1)	O(N)
insertFront(e)	O(N)	O(1)
removeFront()	O(N)	O(1)

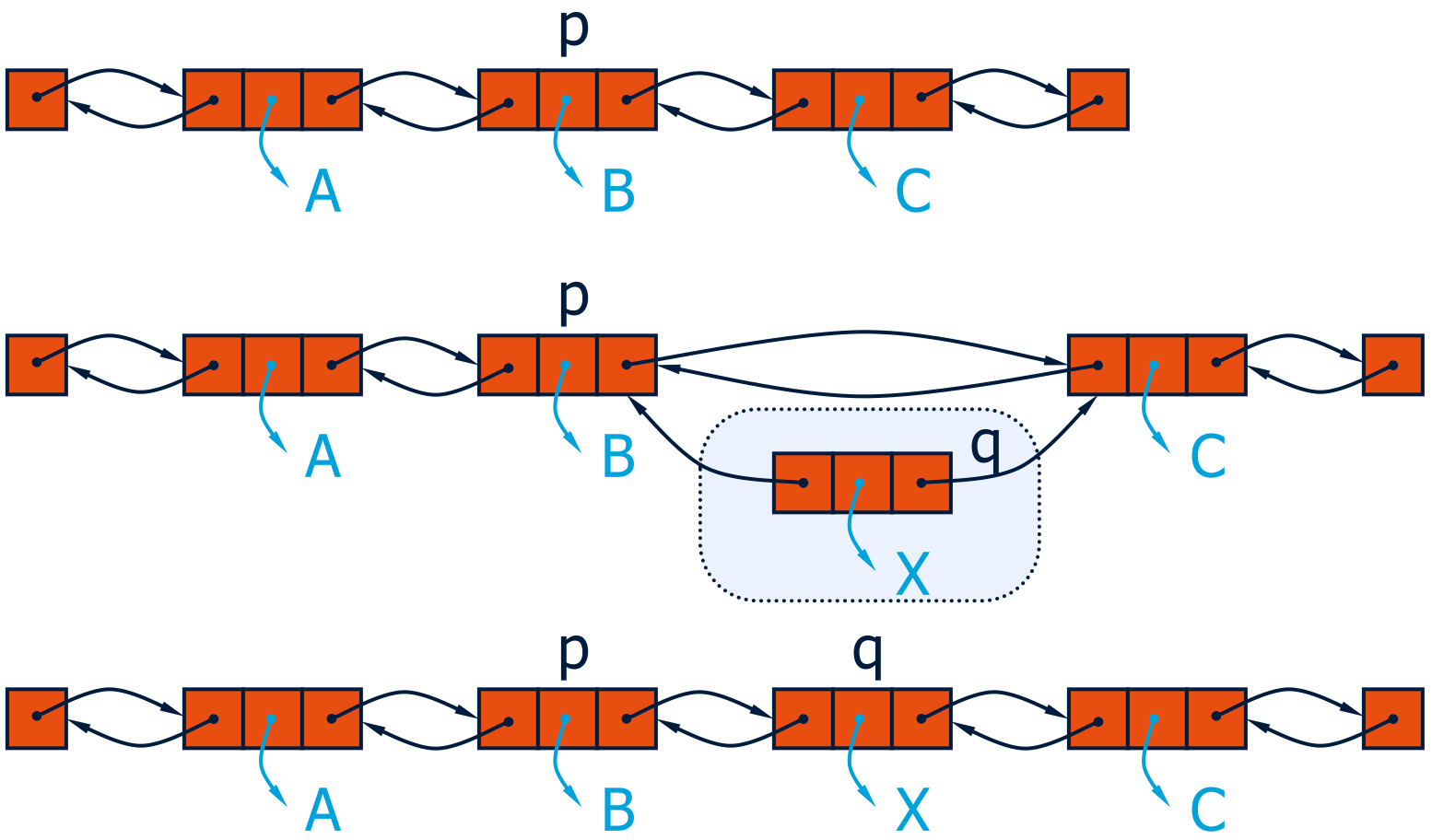
Doubly-Linked Lists

- Doubly-linked Lists Nodes store:
  - Element
  - Reference to next position
  - Reference to previous position



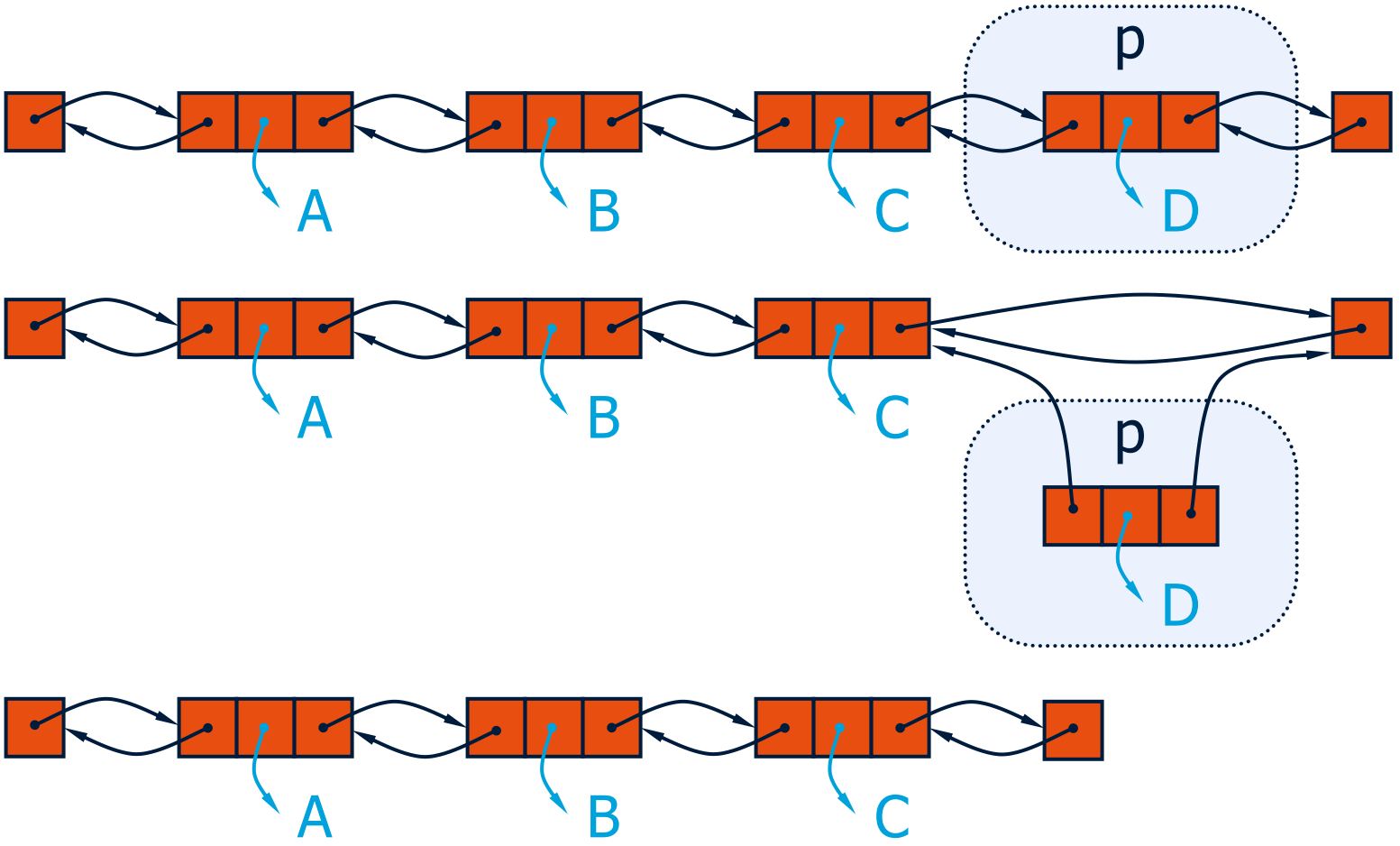
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly-linked list.

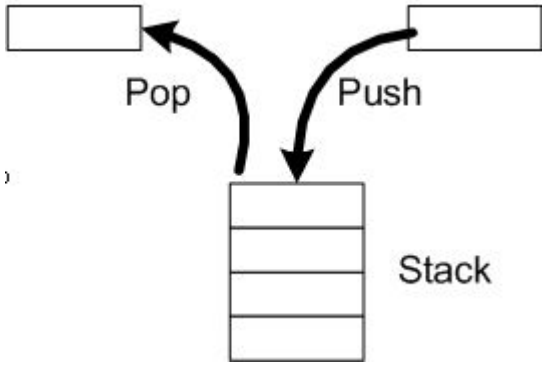


Stack and Queue

Stacks

- Last-in first-out (LIFO) data structure
  - Stack Overflow?

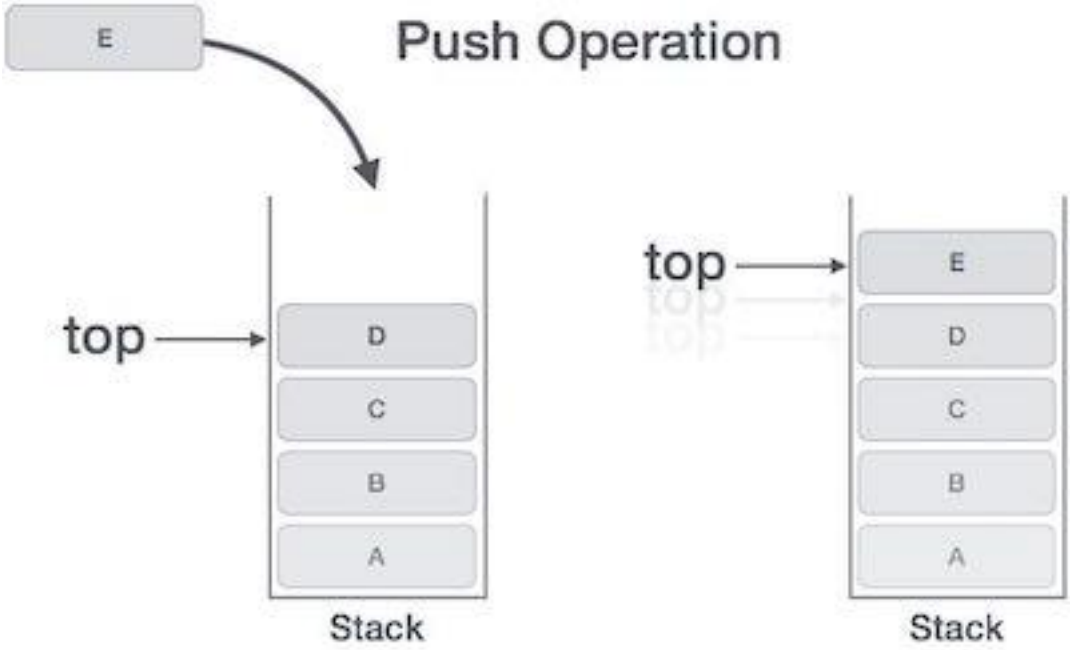
- Main operations:
  - push(e): insert an element
  - pop(): *remove and return* last inserted element



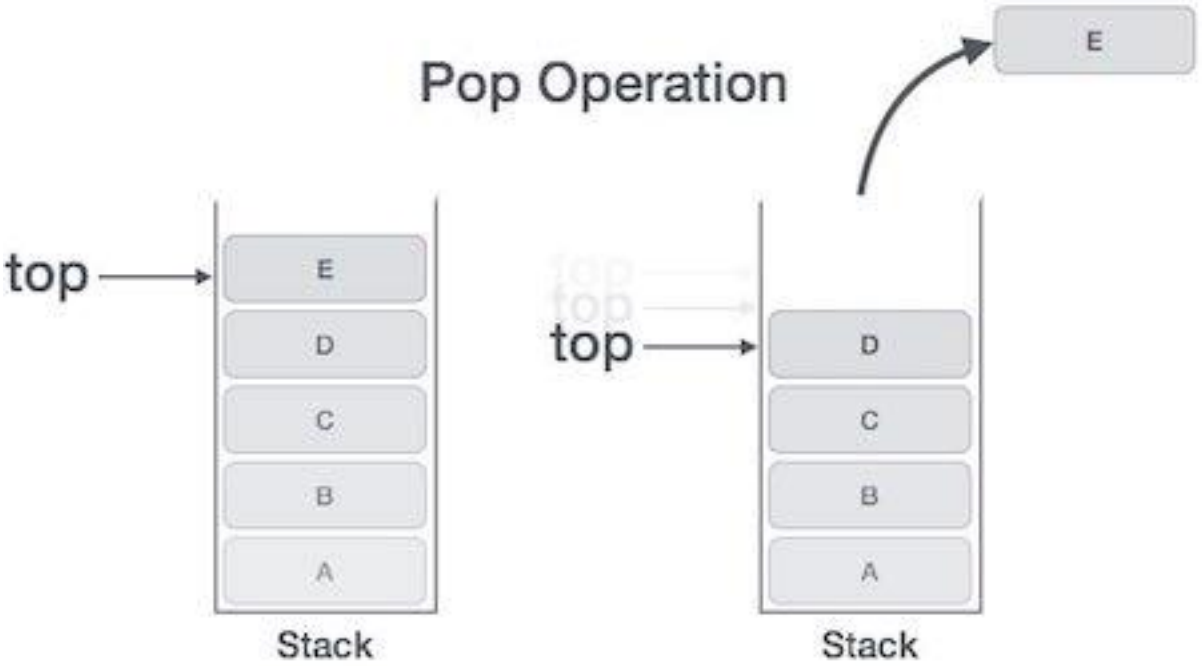
Stack ADT

```
public interface Stack<E> {
    E pop();
    void push(E);
    boolean isEmpty();
    E top();
    int size();
}
```

Stack Operations



Stack Operations



Stack Example

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

Stack Applications

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a program

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
    (This holds for ALL data structures)

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

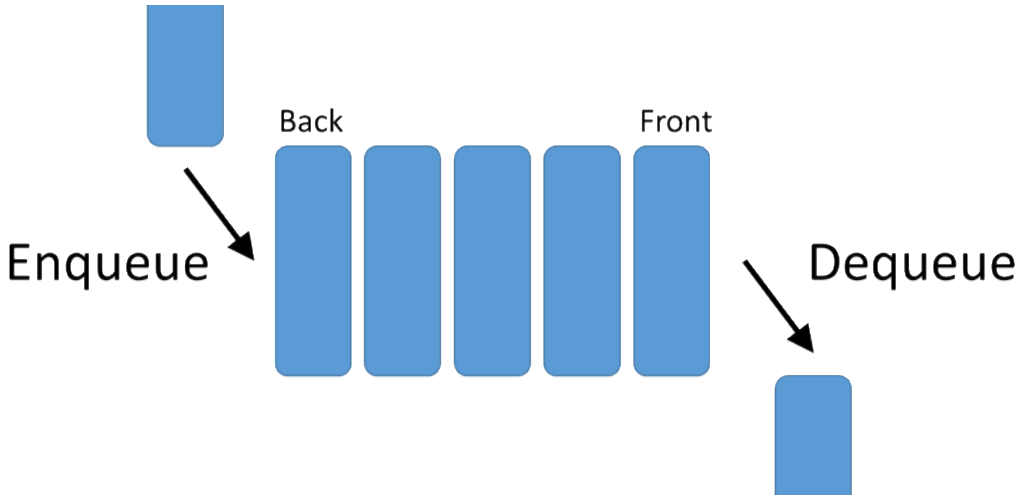
Array-based Stack

- The array storing the stack elements may become full
  - We may need to **resize**

Queues

- First-in first-out (FIFO) data structure
  - Insertions at the back
  - Removals at the front

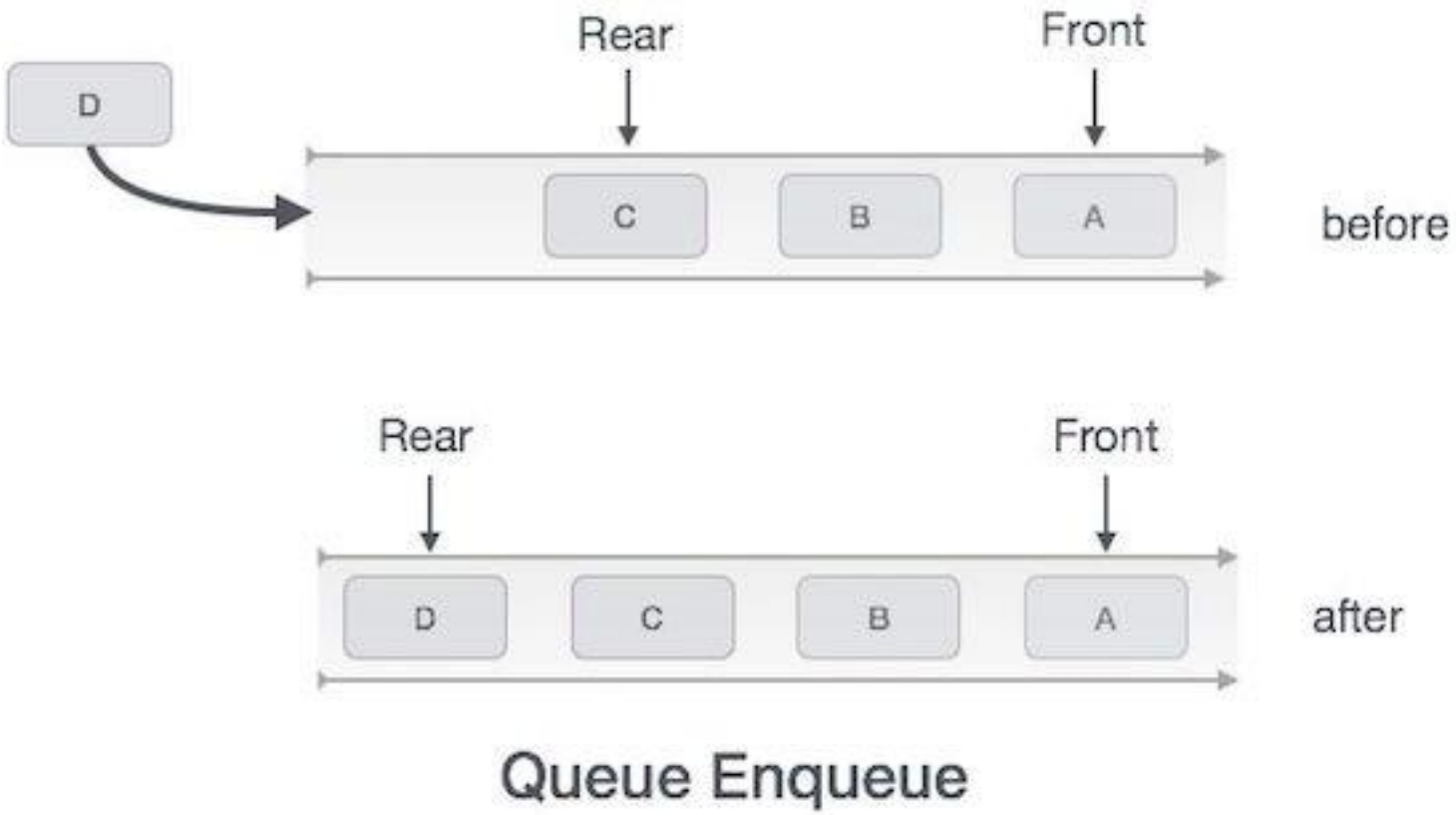
- Main operations:
  - enqueue(e): inserts e at end of queue
  - dequeue(): *removes and returns* element at front of queue



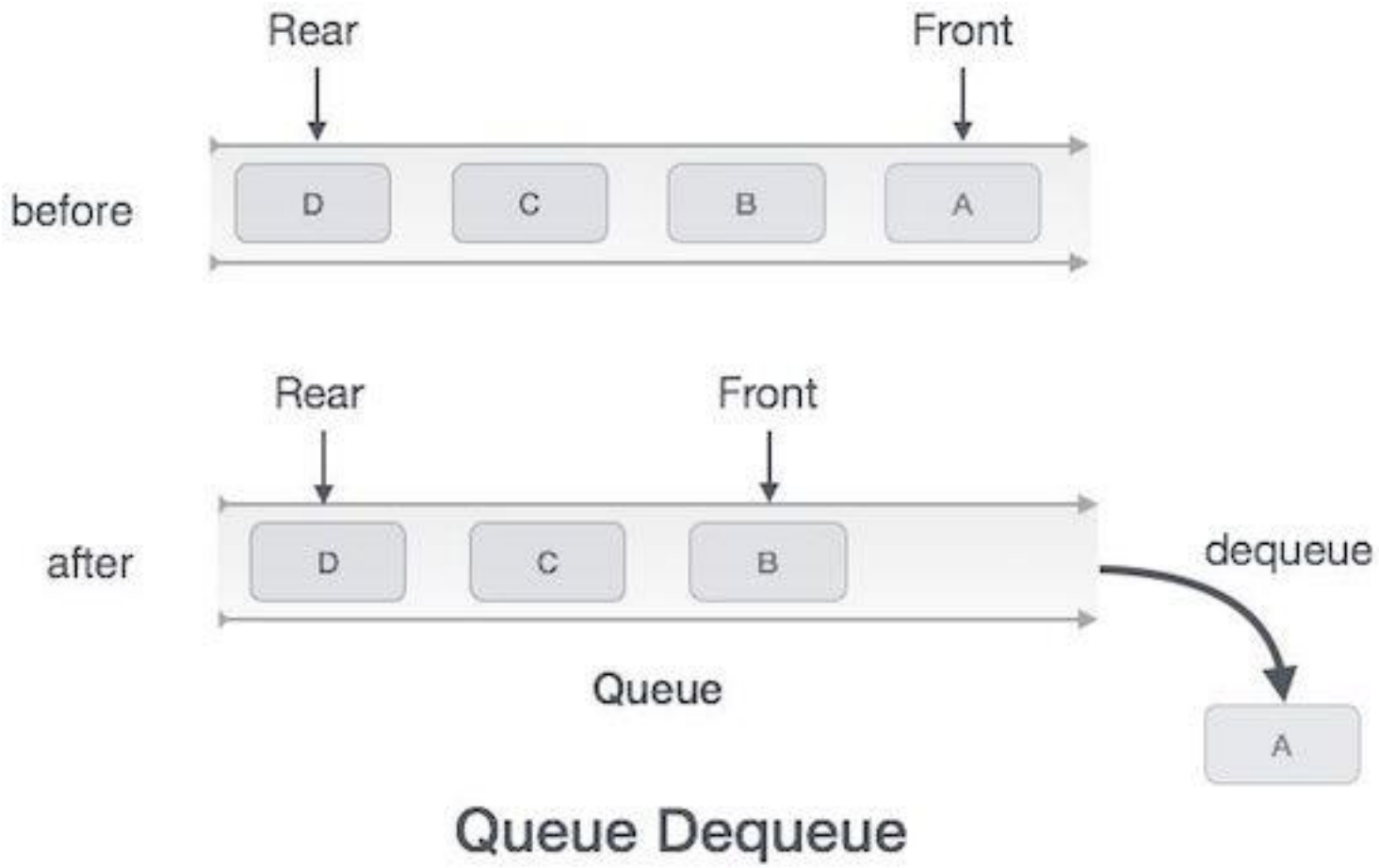
Queue ADT

```
public interface Queue<E> {
    E dequeue();
    void enqueue(E);
    boolean isEmpty();
    E first();
    int size();
}
```

Queue Operations



Queue Operations



Queue Dequeue

Queue Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

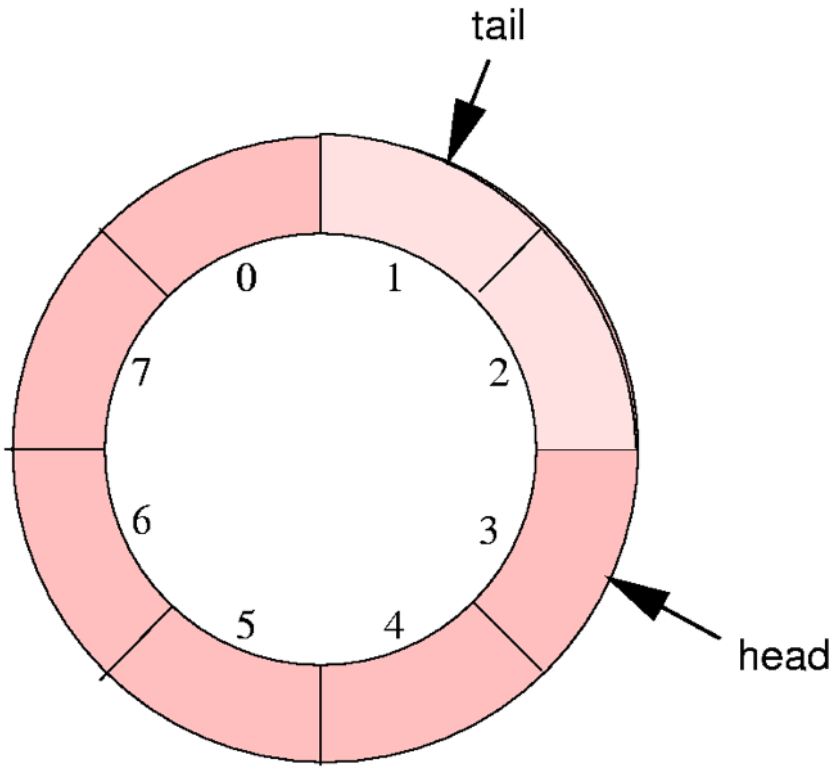
Queue Applications

- Direct applications
  - Access to shared resources (e.g., printer)
  - Multi-threaded programming

- An array-based implementation
  - Problem?

Array-based Queue

- Regular removal / insertion at front
  - Problem using “regular array”?
- Use a “circular array”
- Keep track of:
  - f: index of front element
  - n: number of elements

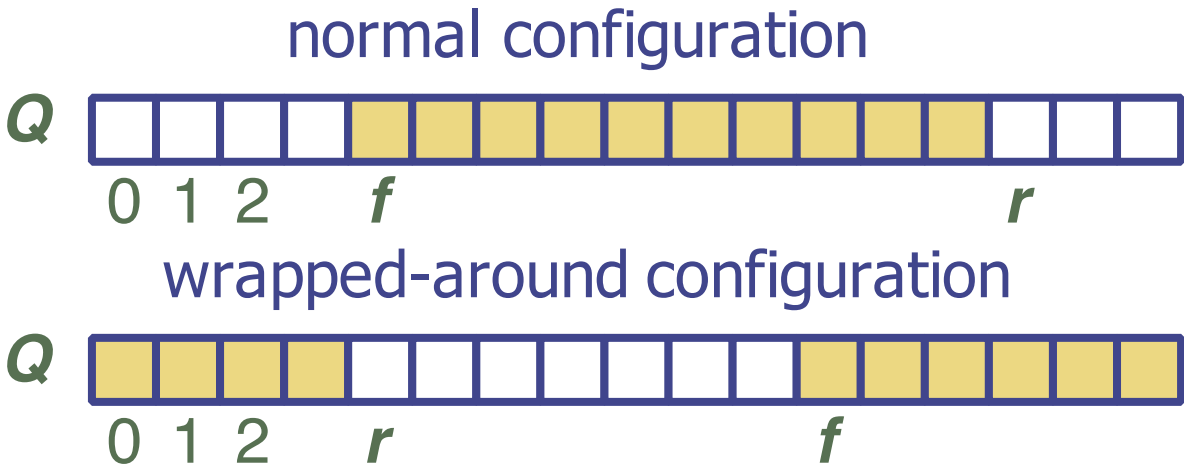


Array-based Queue

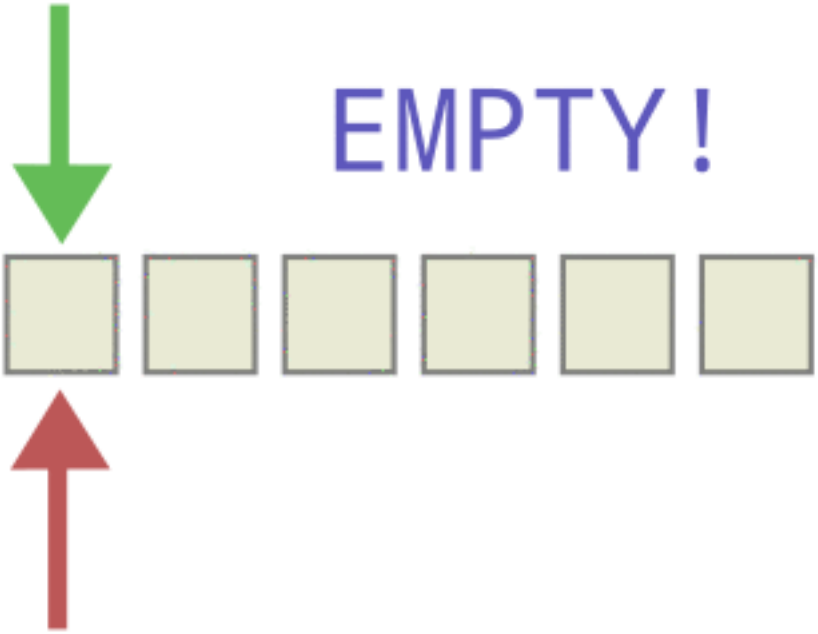
When the queue has fewer than N elements, array location:

$$r = (f + n) \bmod N$$

is the first empty slot past the rear of the queue



Array-Based Queue



Array-based Queue

```
public void enqueue(E value) {
    a[f + n % a.length] = value;
    n++;
}

public E dequeue() {
    n--;
    E value = a[f];
    f = (f + 1) % a.length;
    return value;
}
```

Queues and Stacks

Some key-questions:

- Could you implement them as Linked Lists?
- If so, what would be the benefit/downside?
- For which application would you prefer which implementation?

Set

Sets

- Store a collection of values
  - No ordering
  - No repetitions

- Main operations:
  - insert(e): inserts an element
  - delete(e): removes an element
  - search(e): checks if an element is in the Set

Set ADT

```
public interface Set<E> {
    void insert(E);
    void delete(E);
    boolean search(E);
    boolean isEmpty();
    int size();
}
```

Set Example

Method	Return Value	Set Content
Insert(3)	-	(3)
Insert(8)	-	(3, 8)
Search(6)	False	(3, 8)
Remove(3)	-	(8)
Insert(7)	-	(8, 7)
Insert(5)	-	(8, 7, 5)
Search(7)	True	(8, 7, 5)
Insert(4)	-	(8, 7, 5, 4)
Insert(7)	-	(8, 7, 5, 4)
Remove(8)	-	(7, 5, 4)
Search(8)	False	(7, 5, 4)

Array-based Set

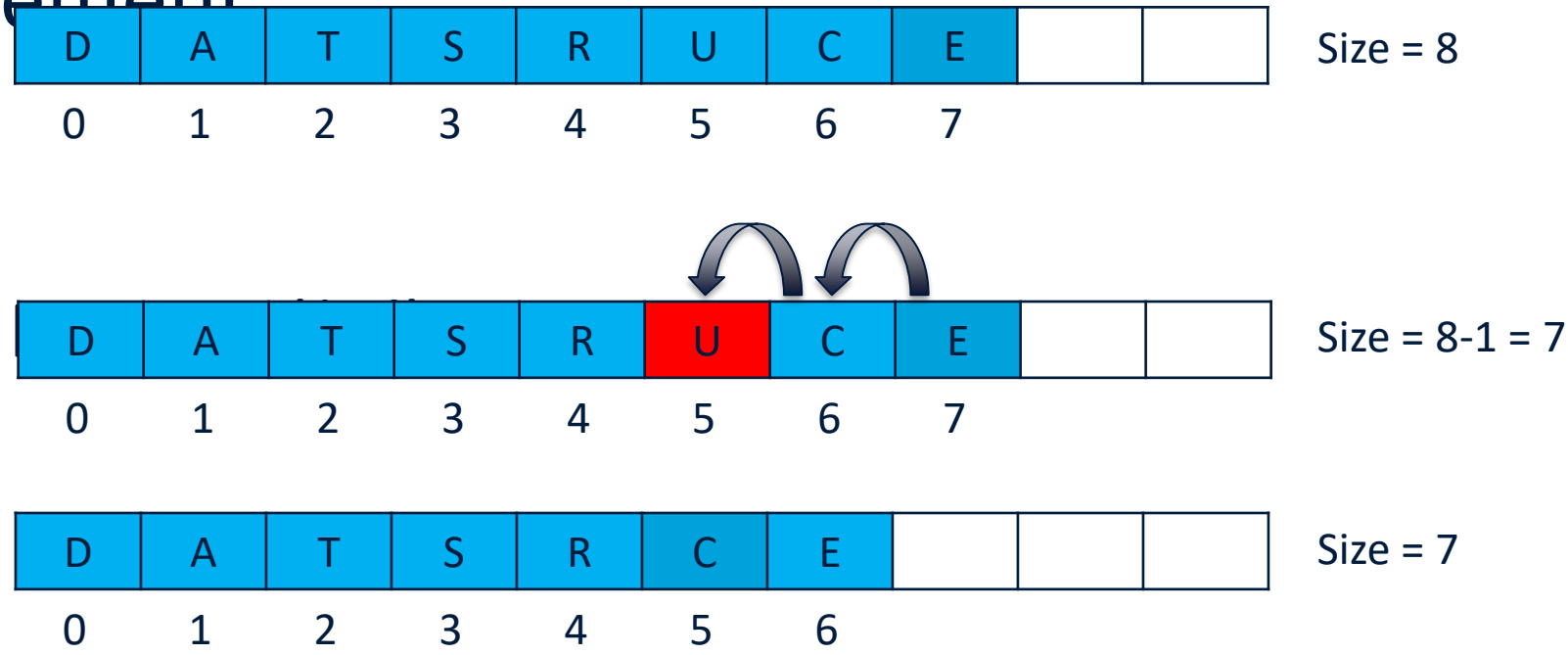
- Basic implementation
- We insert elements from left to right
  - Before inserting a new element, we check if it is present, if not we insert it in the
- A variable keeps track of the size of the Set

An Array-based Set

- The array storing the Set elements may become full
  - Again, we need to **increase the size**!

An Array-based Set

- When removing, we **shift left** all the elements on the right of the removed element



Set operations complexity

- In the Array-based Set, all the operations have a linear complexity O(n)
  - In all the cases we might need to iterate over the whole array
- Data structures and Algorithms we will introduce next weeks could be used to optimize some of the operations
  - Hash Maps
  - Ordering