

# 122com Data structures and types

David Croft

Coventry University

david.croft@coventry.ac.uk

2016

# Overview

- 1 Arrays
- 2 Linked lists
  - Array example
  - LL example
- 3 Data structures
- 4 Abstract data types
- 5 Queues
- 6 Stacks
- 7 Sets
- 8 Other
- 9 Trees
- 10 End

A series of objects all of the same size and type.

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

- Stored in contiguous blocks of memory.
- Python lists are functionally closest.
  - But are not arrays.
- Can't be resized.

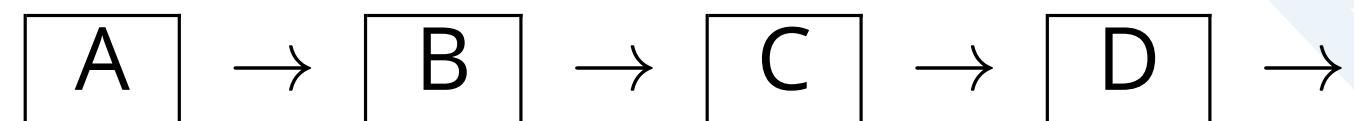
# Linked lists

C

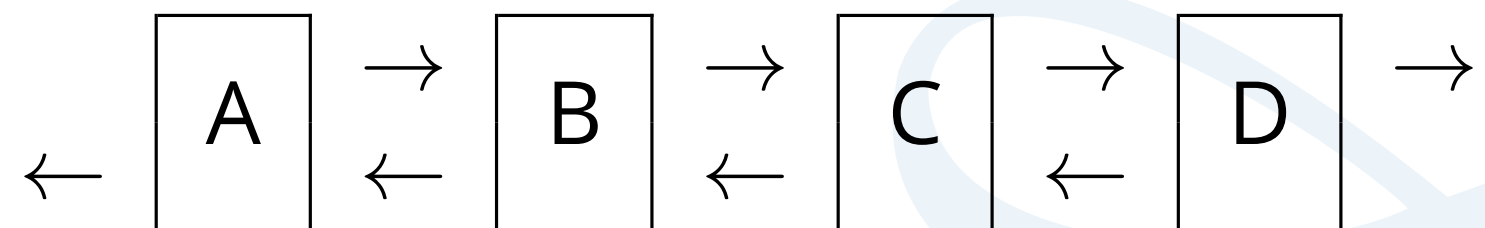
The challenger for array's crown.

- Series of nodes, each of which points to the next element.
  - And to the previous element if it's a doubly linked list.

Singularly linked



Doubly linked

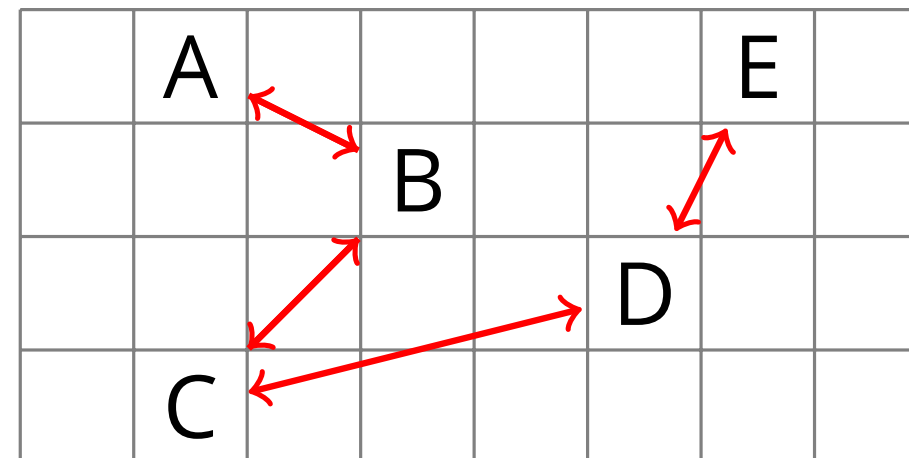


# Linked lists II

A

Not in contiguous memory.

- Each node is separate.
- Scattered.
- Dynamic memory (pointers!).



- Why would we use linked lists instead of arrays?
  - Can change size.
  - Can quickly insert and delete elements.

```
class Node:
    __prev = None
    __next = None
    value = None
```

```
class Node
{
private:
    Node *prev;
    Node *next;

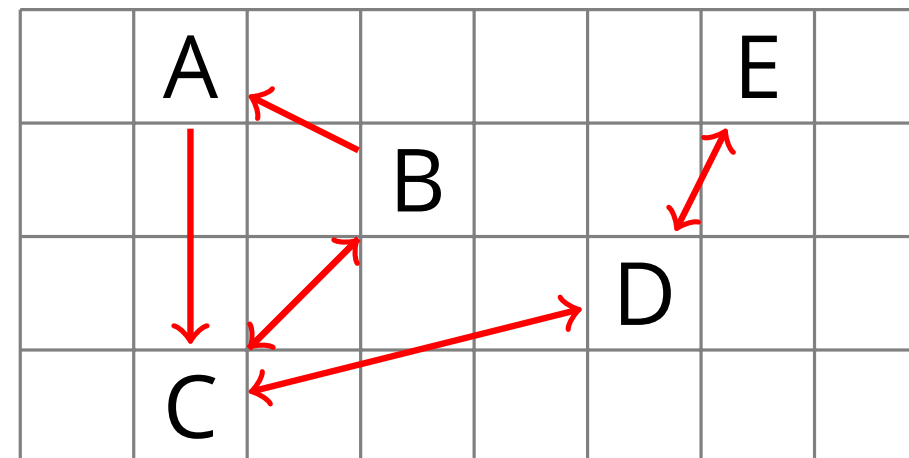
public:
    int value;
};
```

# Linked lists II

A

Not in contiguous memory.

- Each node is separate.
- Scattered.
- Dynamic memory (pointers!).



- Why would we use linked lists instead of arrays?
  - Can change size.
  - Can quickly insert and delete elements.

```
class Node:
    __prev = None
    __next = None
    value = None
```

```
class Node
{
private:
    Node *prev;
    Node *next;

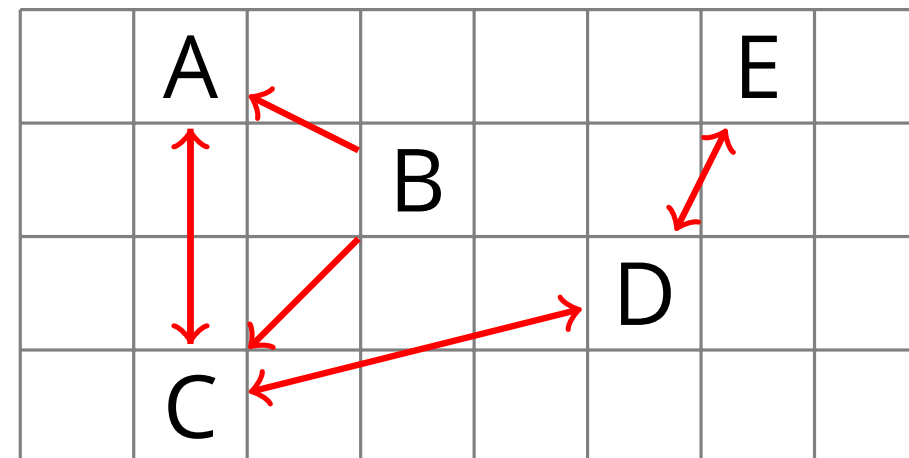
public:
    int value;
};
```

# Linked lists II

A

Not in contiguous memory.

- Each node is separate.
- Scattered.
- Dynamic memory (pointers!).



- Why would we use linked lists instead of arrays?
  - Can change size.
  - Can quickly insert and delete elements.

```
class Node:
    __prev = None
    __next = None
    value = None
```

```
class Node
{
private:
    Node *prev;
    Node *next;

public:
    int value;
};
```

# Removing array elements

C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A	B	C	D	E	

- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?



# Removing array elements

C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A		C	D	E	

- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?
  - 1 Remove element from the array.

# Removing array elements

C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A	C		D	E	

- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?
  - 1 Remove element from the array.
  - 2 Move next element to occupy the empty space.

# Removing array elements

C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A	C	D		E	

- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?
  - 1 Remove element from the array.
  - 2 Move next element to occupy the empty space.
  - 3 Repeat.

# Removing array elements

C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A	C	D	E		

- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?
  - 1 Remove element from the array.
  - 2 Move next element to occupy the empty space.
  - 3 Repeat.

# Removing array elements

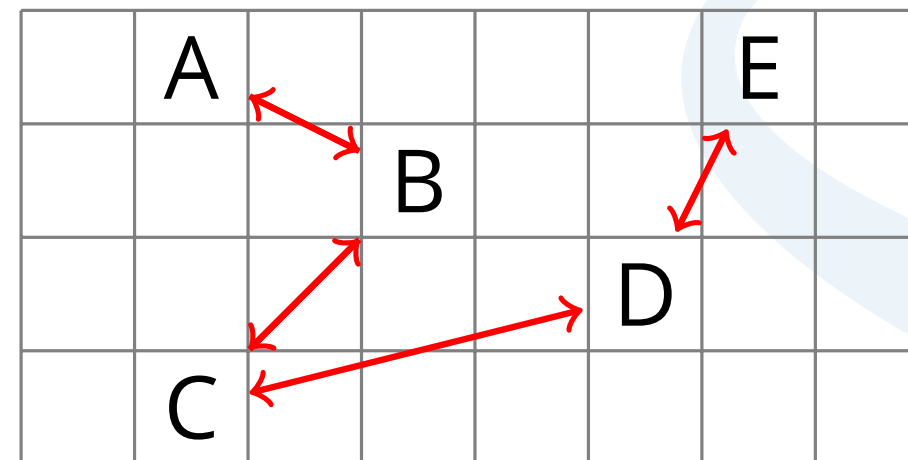
C

```
char array[] = {'A', 'B', 'C', 'D', 'E'};
```

		A	C	D	E		

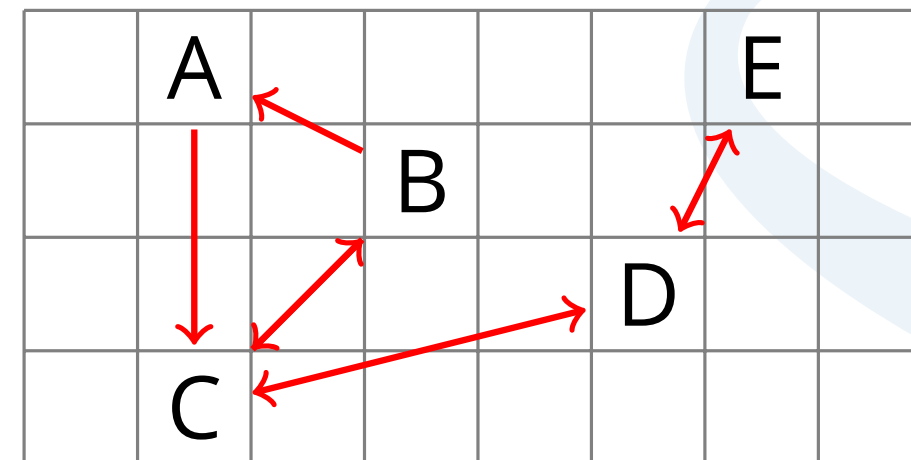
- Array in memory, multiple elements in a contiguous block.
- How do we remove elements from the middle?
  - 1 Remove element from the array.
  - 2 Move next element to occupy the empty space.
  - 3 Repeat.
- Is very slow with large arrays.

# Removing linked list elements



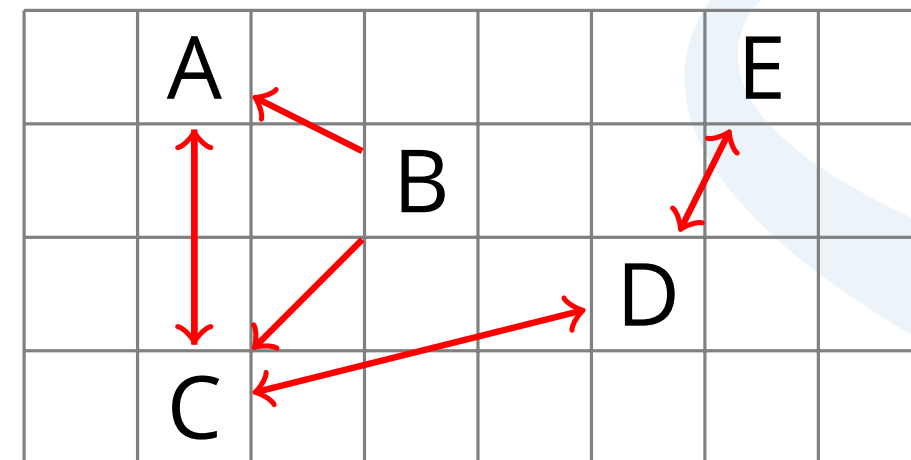
- Linked list, separate elements scattered in memory.
- Each pointing to the next/prev element.
- How do we remove elements?

# Removing linked list elements



- Linked list, separate elements scattered in memory.
- Each pointing to the next/prev element.
- How do we remove elements?
  - 1 Change pointers.

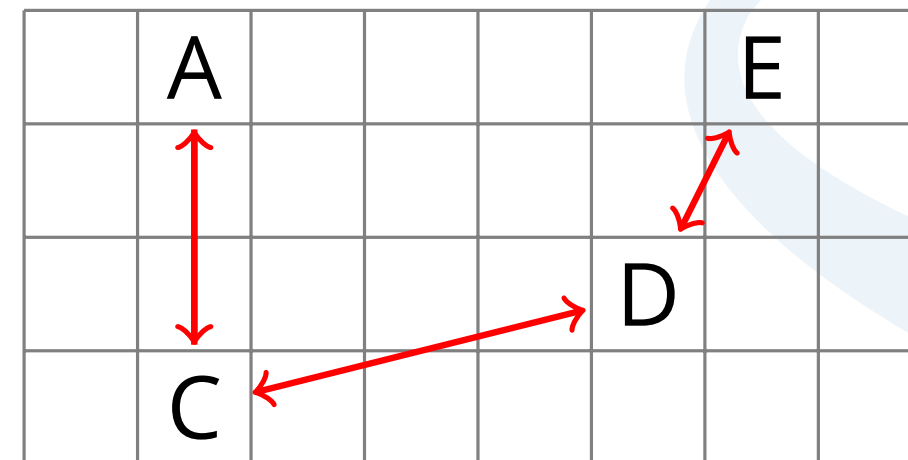
# Removing linked list elements



- Linked list, separate elements scattered in memory.
- Each pointing to the next/prev element.
- How do we remove elements?
  - 1 Change pointers.



# Removing linked list elements



- Linked list, separate elements scattered in memory.
- Each pointing to the next/prev element.
- How do we remove elements?
  - 1 Change pointers.
  - 2 Delete old element.

## Advantages

- Inserting and deleting elements is very fast.
  - $O(1)$ .
- No size limits, can keep adding new elements.
- Doesn't waste memory.

## Disadvantages

- Not indexed.
  - Can't ask for the 20<sup>th</sup> element etc.
  - Have to step through the list (slow).
- Needs more memory than an array to store the same number of elements.
  - Have to keep track of where the next/prev nodes are.

# Data structures

C

Arrays and linked lists are data structures.

- A specific way of storing data.
- Can see how the various elements of the structure are laid out in memory.
- Direct access to the underlying memory.

# Abstract data types

I

As we move to storing more complex information in our software we will start to encounter Abstract Data Types (ADTs).

- Software engineering principal.

# Abstract data types

I

As we move to storing more complex information in our software we will start to encounter ADTs.

- Software engineering principal.
- Keep what a data type can do...

# Abstract data types

I

As we move to storing more complex information in our software we will start to encounter ADTs.

- Software engineering principal.
- Keep what a data type can do...  
...and how it does it separate.

# Abstract data types

I

As we move to storing more complex information in our software we will start to encounter ADTs.

- Software engineering principal.
- Keep what a data type can do...  
...and how it does it separate.
- Unlike data structure ADTs only concerned with the interface.

# Abstract data types

I

As we move to storing more complex information in our software we will start to encounter ADTs.

- Software engineering principal.
- Keep what a data type can do...  
...and how it does it separate.
- Unlike data structure ADTs only concerned with the interface.
- Internals of ADTs can vary widely between implementations.



Imagine an ADT like a car.

- It has a set of supported operations, go faster, go slower, turn left, turn right.

Imagine an ADT like a car.

- It has a set of supported operations, go faster, go slower, turn left, turn right.
- Don't care how it achieves these.

Imagine an ADT like a car.

- It has a set of supported operations, go faster, go slower, turn left, turn right.
- Don't care how it achieves these.
- Don't care if, internally, it's using a combustion engine or an electric motor.

Imagine an ADT like a car.

- It has a set of supported operations, go faster, go slower, turn left, turn right.
- Don't care how it achieves these.
- Don't care if, internally, it's using a combustion engine or an electric motor.
- Only care about the result.

A First In First Out (FIFO) ADT.

- Ends of the queue are called the front and back.
- New elements added to back of queue only.
  - Pushing - push(value)
- Old elements removed from front of queue only.
  - Popping - pop()
- No cutting in.

## A FIFO ADT.

- Ends of the queue are called the front and back.
- New elements added to back of queue only.
  - Pushing - push(value)
- Old elements removed from front of queue only.
  - Popping - pop()
- No cutting in.
- Buffer to hold items for processing in the order in which they arrive.

## A FIFO ADT.

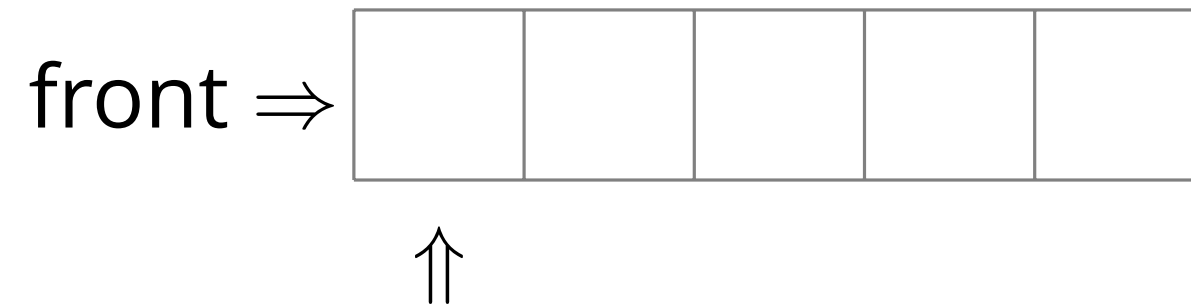
- Ends of the queue are called the front and back.
- New elements added to back of queue only.
  - Pushing - push(value)
- Old elements removed from front of queue only.
  - Popping - pop()
- No cutting in.
- Buffer to hold items for processing in the order in which they arrive.
- Which would be better for a queue? An array or a linked list?

## A FIFO ADT.

- Ends of the queue are called the front and back.
- New elements added to back of queue only.
  - Pushing - push(value)
- Old elements removed from front of queue only.
  - Popping - pop()
- No cutting in.
- Buffer to hold items for processing in the order in which they arrive.
- Which would be better for a queue? An array or a linked list?
  - Linked list.



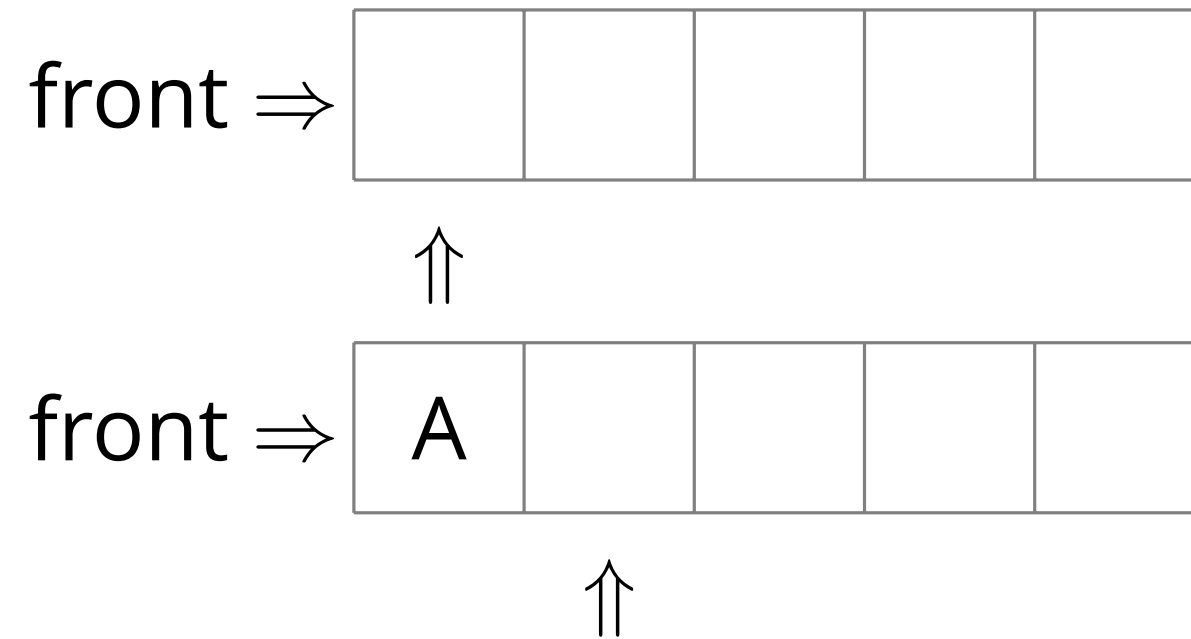
# Array as a queue.



- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.

# Array as a queue.

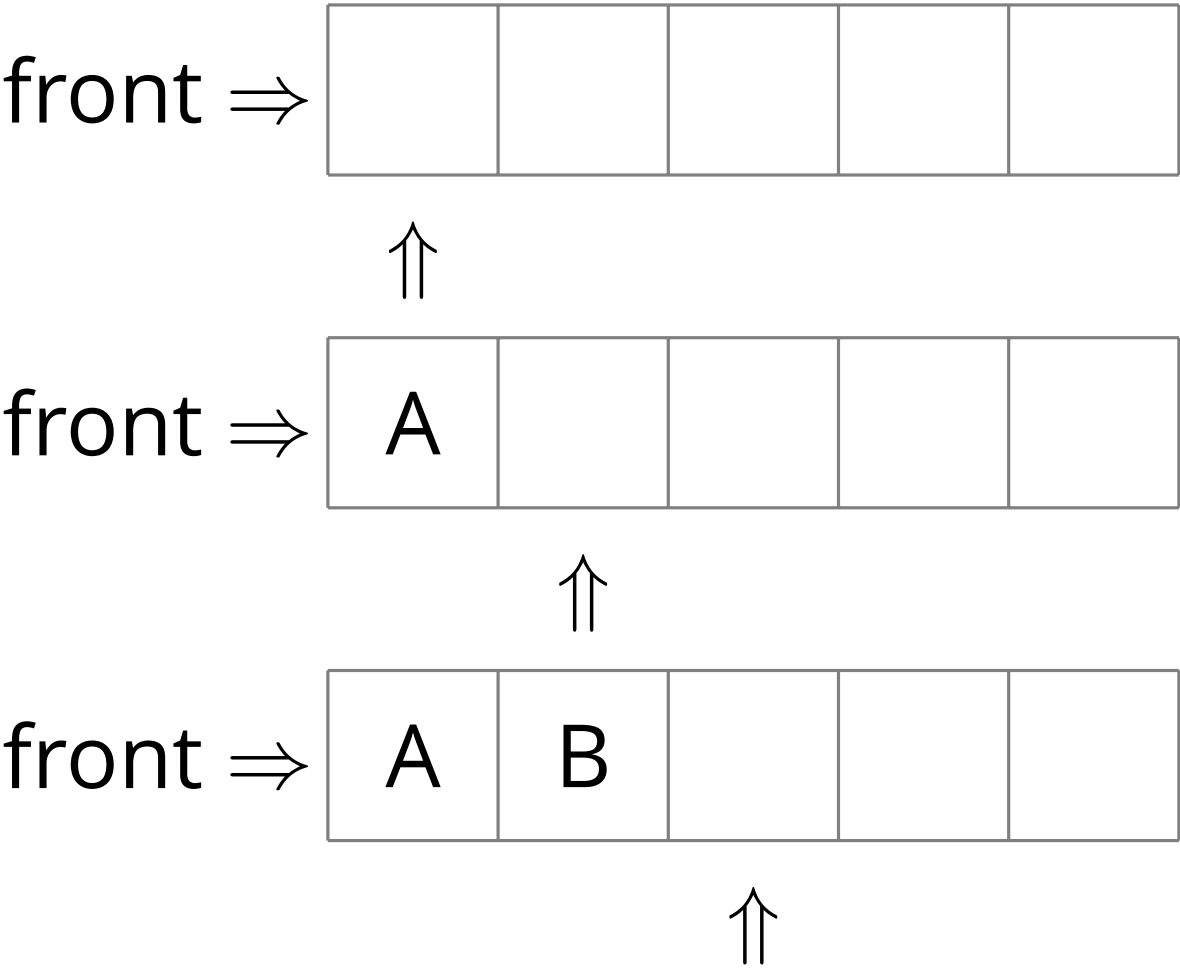
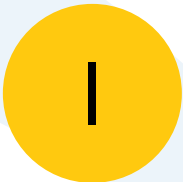
1



push(A)

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.

# Array as a queue.

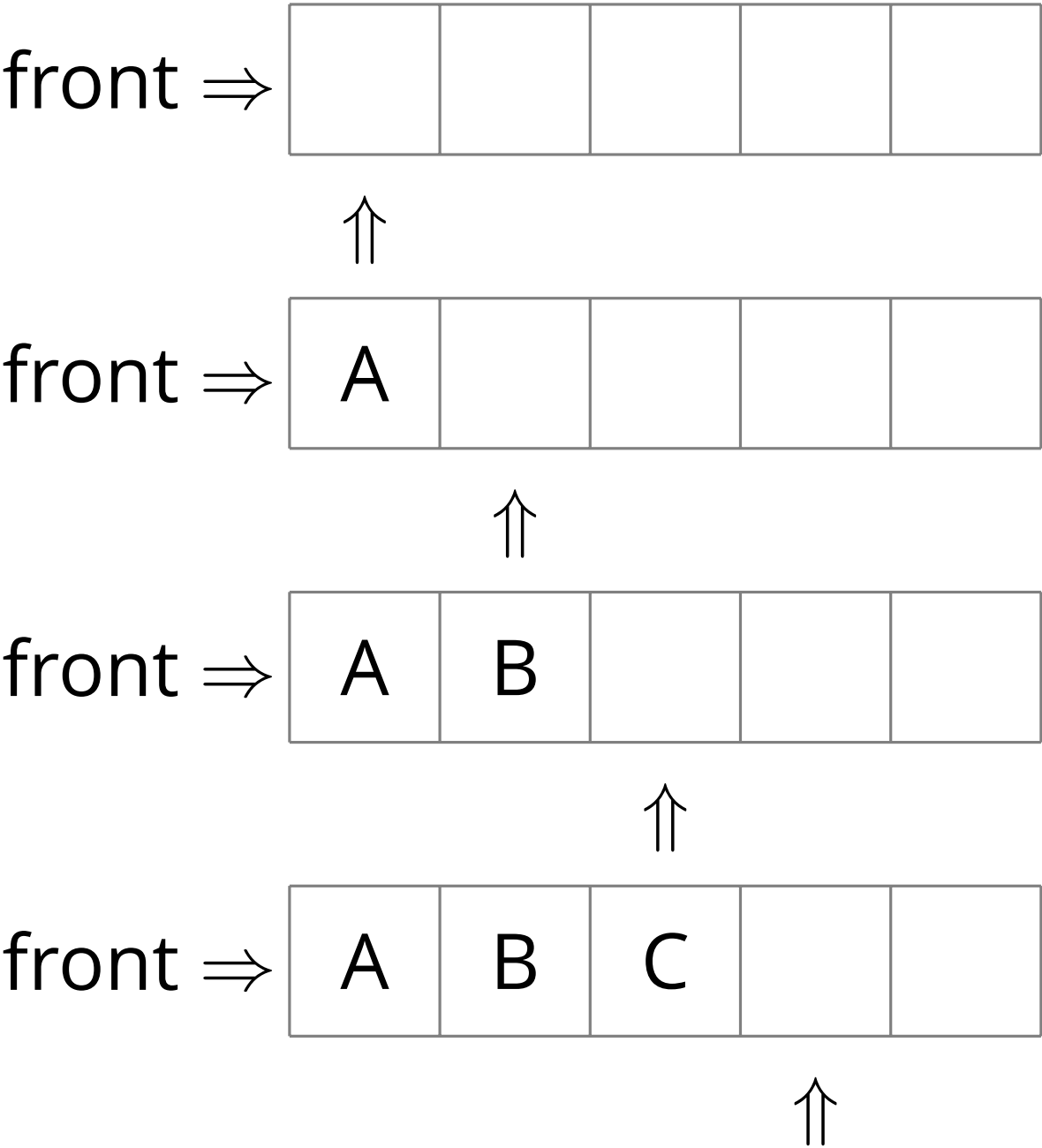
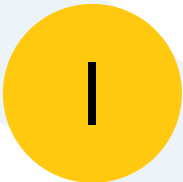


push(A)

push(B)

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.

# Array as a queue.



push(A)

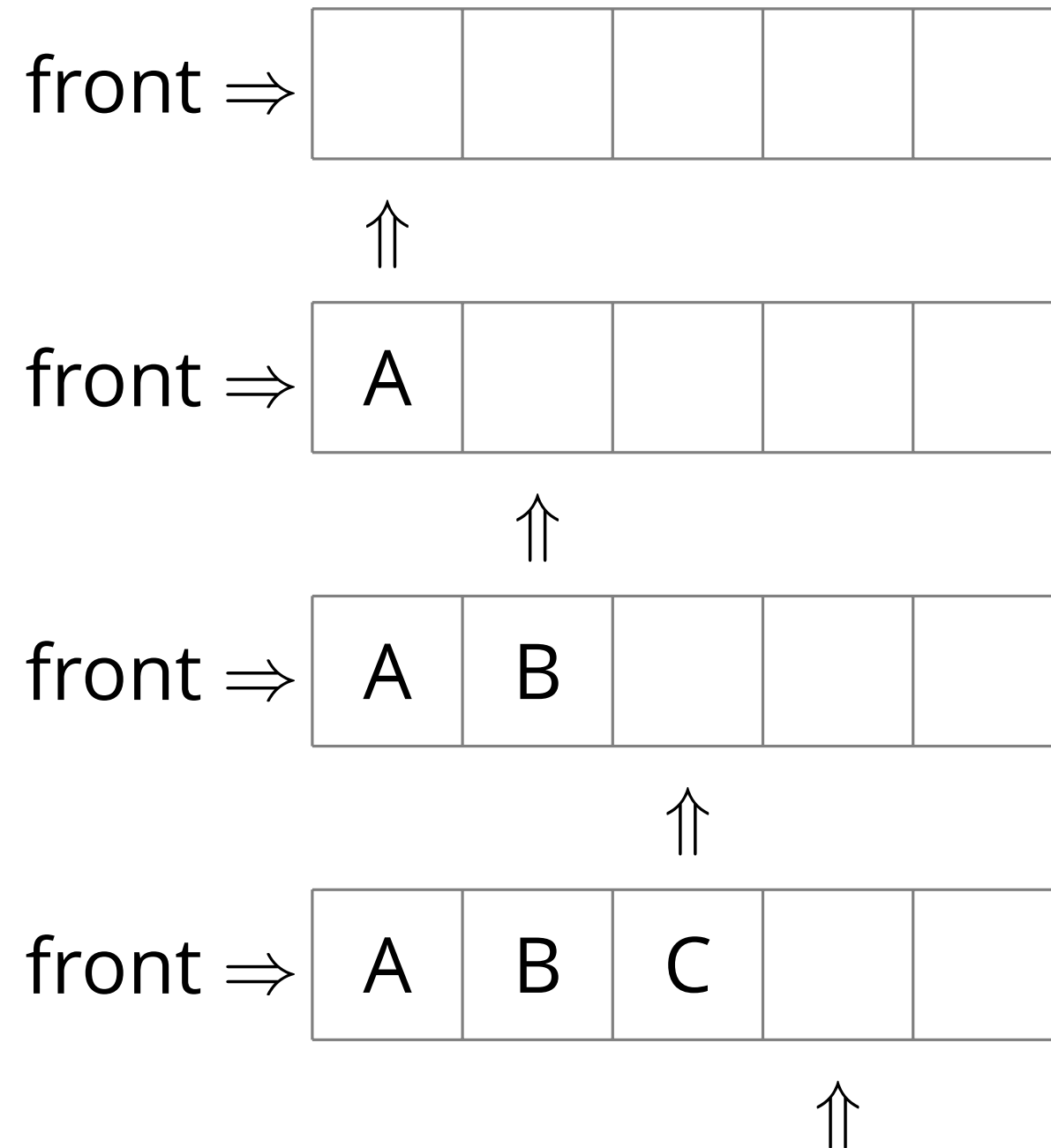
push(B)

push(C)

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.

# Array as a queue.

1



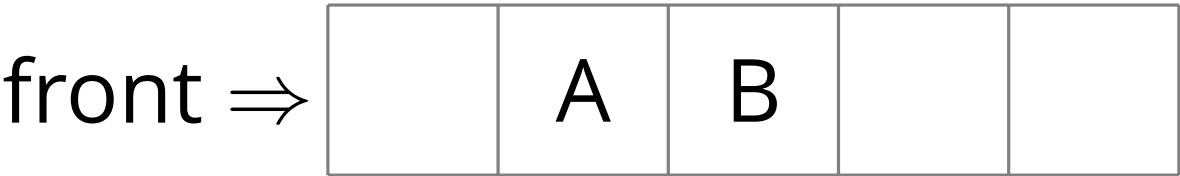
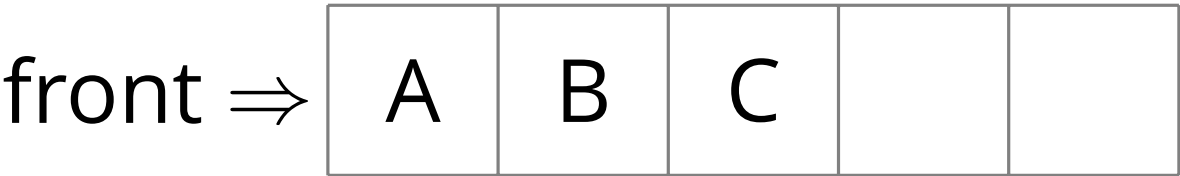
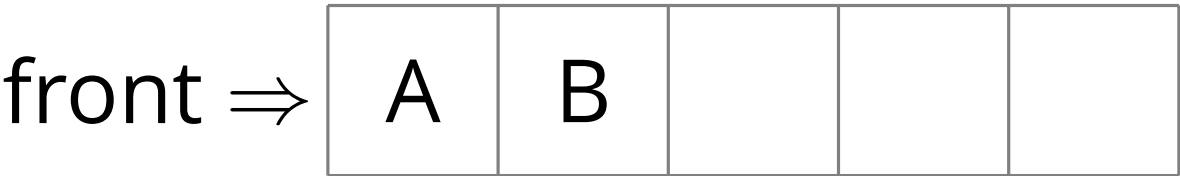
push(A)

push(B)

push(C)

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.
- What happens when we pop()?
  - Have to shuffle every element forward one space.
  - Inefficient.

# Array as a queue.



push(A)

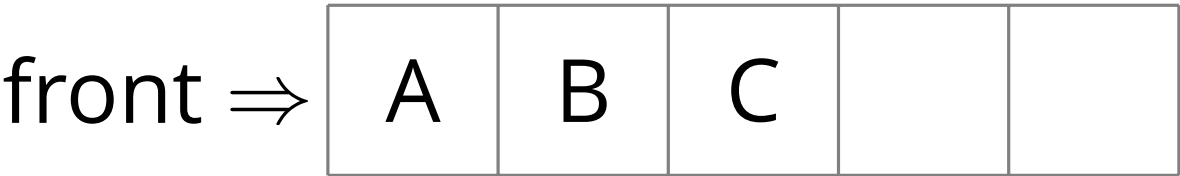
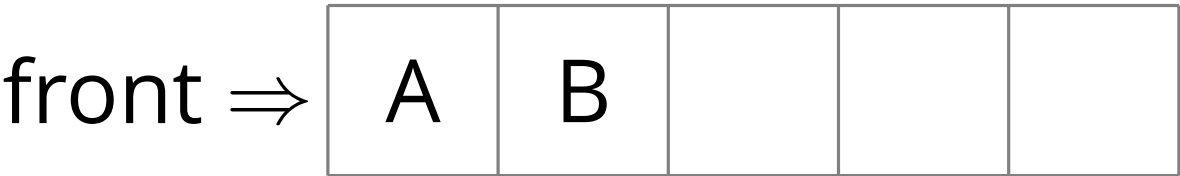
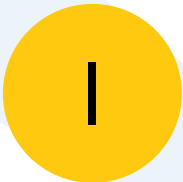
push(B)

push(C)

pop()

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.
- What happens when we pop()?
  - Have to shuffle every element forward one space.
  - Inefficient.

# Array as a queue.



push(A)

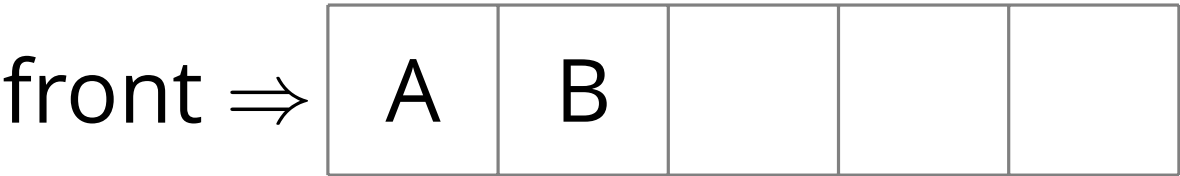
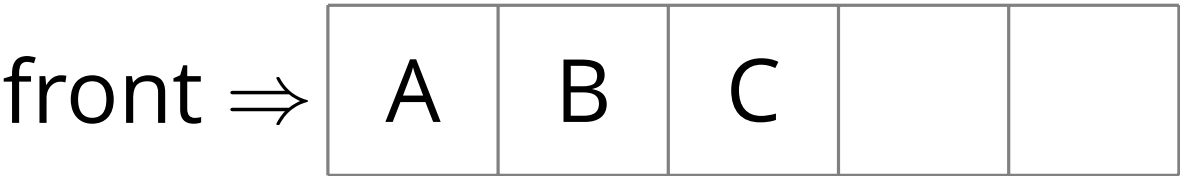
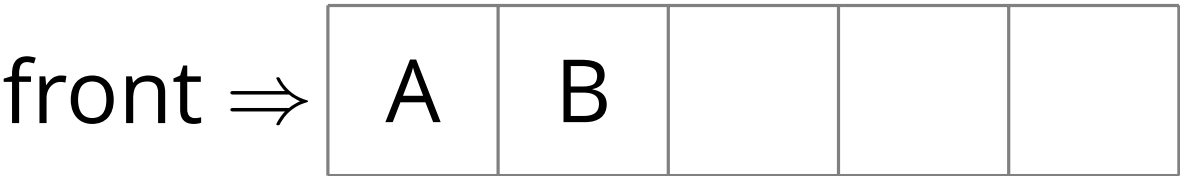
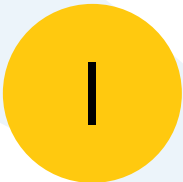
push(B)

push(C)

pop()

- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.
- What happens when we pop()?
  - Have to shuffle every element forward one space.
  - Inefficient.

# Array as a queue.



push(A)

push(B)

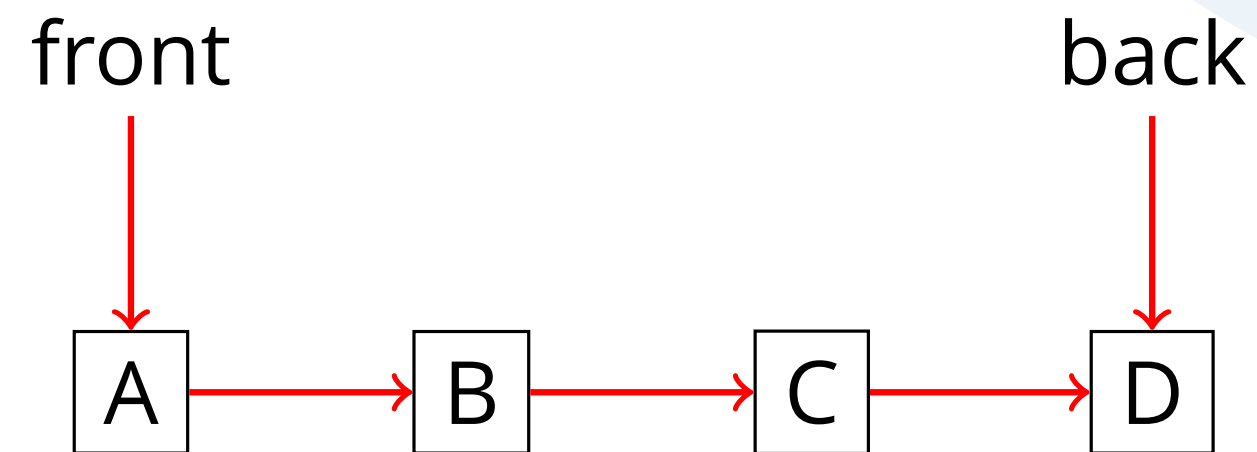
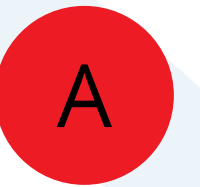
push(C)

pop()

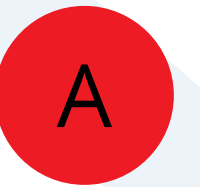
- Very similar to stacks.
  - Keep track of next free space.
  - Limited size.
- What happens when we pop()?
  - Have to shuffle every element forward one space.
  - Inefficient.



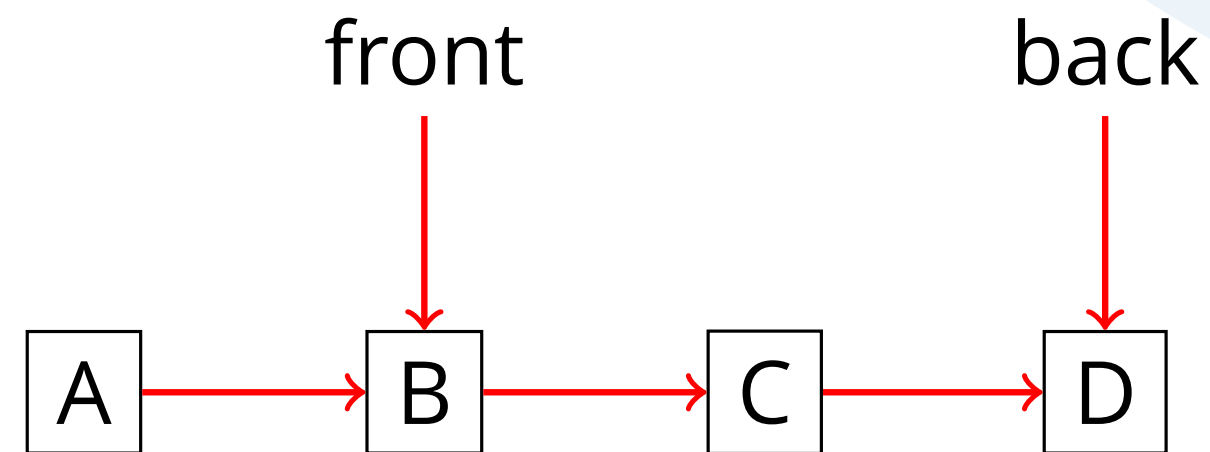
# Linked list as a queue.



# Linked list as a queue.



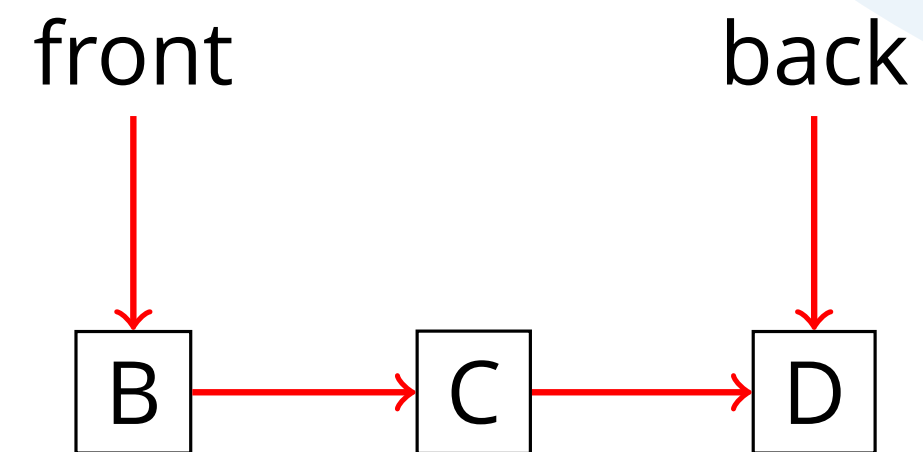
pop()



# Linked list as a queue.

A

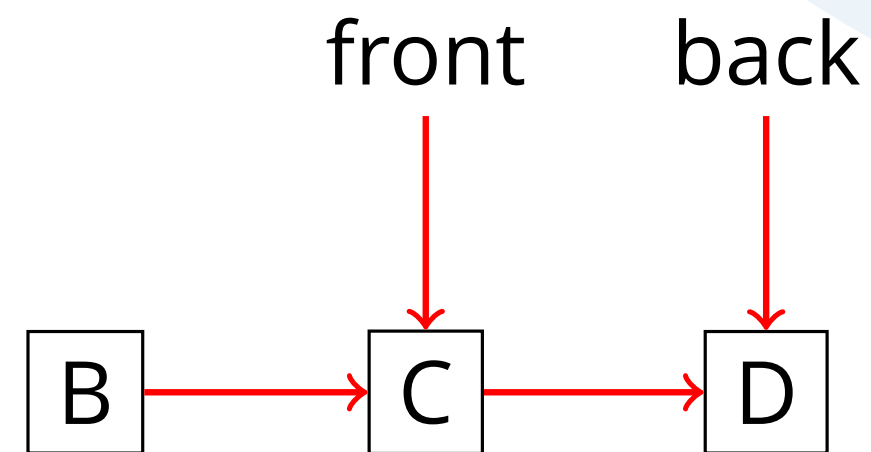
pop()



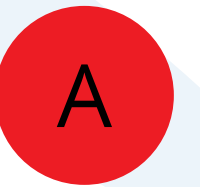
# Linked list as a queue.

A

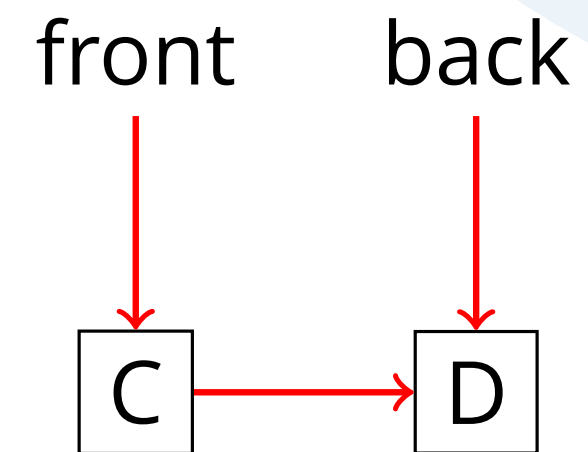
pop() , pop()



# Linked list as a queue.



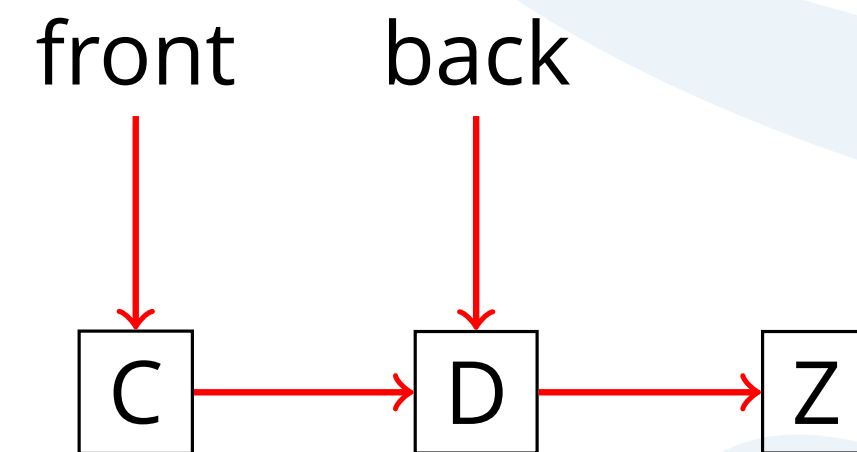
pop() , pop()



# Linked list as a queue.

A

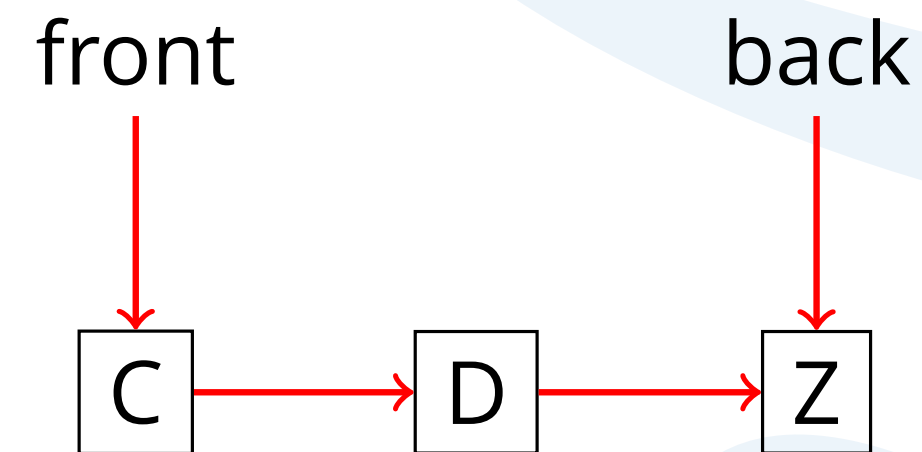
pop() , pop() , push(Z)

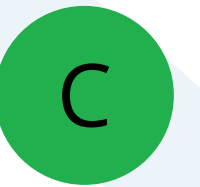


# Linked list as a queue.

A

pop() , pop() , push(Z)

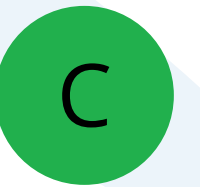




A First In Last Out (FILO) ADT.

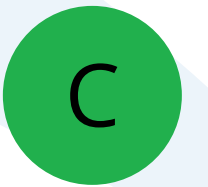
- Ends of the stack are called the top and bottom.
- New elements add to top of stack only.
  - Pushing - push(value)
- Old elements removed from top of stack only.
  - Popping - pop()
- No cutting in.





## A FILO ADT.

- Ends of the stack are called the top and bottom.
- New elements add to top of stack only.
  - Pushing - push(value)
- Old elements removed from top of stack only.
  - Popping - pop()
- No cutting in.
- Which would be better for a stack? An array or a linked list?



## A FILO ADT.

- Ends of the stack are called the top and bottom.
- New elements add to top of stack only.
  - Pushing - push(value)
- Old elements removed from top of stack only.
  - Popping - pop()
- No cutting in.
- Which would be better for a stack? An array or a linked list?
  - Doesn't matter performance wise.
  - Linked list if  $n$  is unknown.

# Array as a stack.

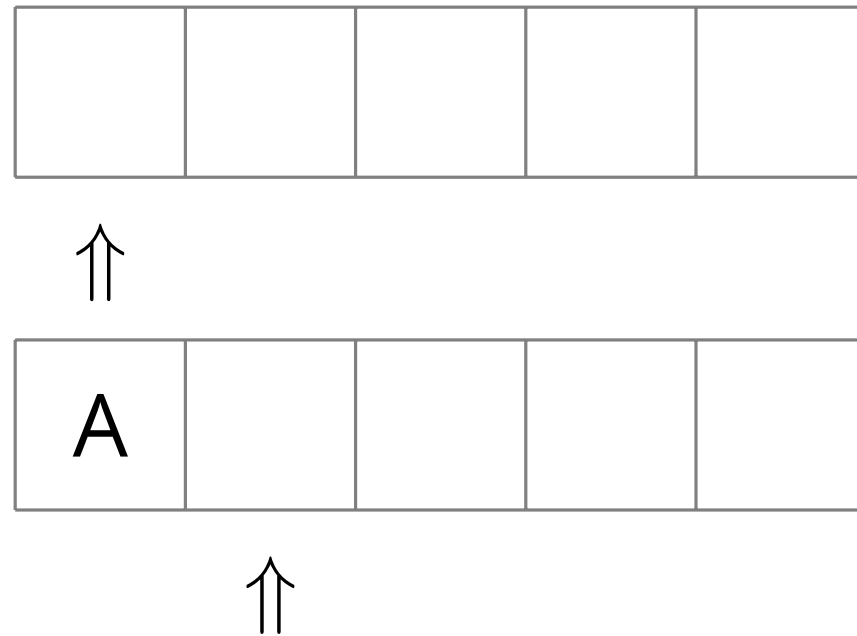
I



- Keep track of position of the next free space in the array.
- Arrays have a fixed size.
  - Can't hold more values than we have space for.

# Array as a stack.

1

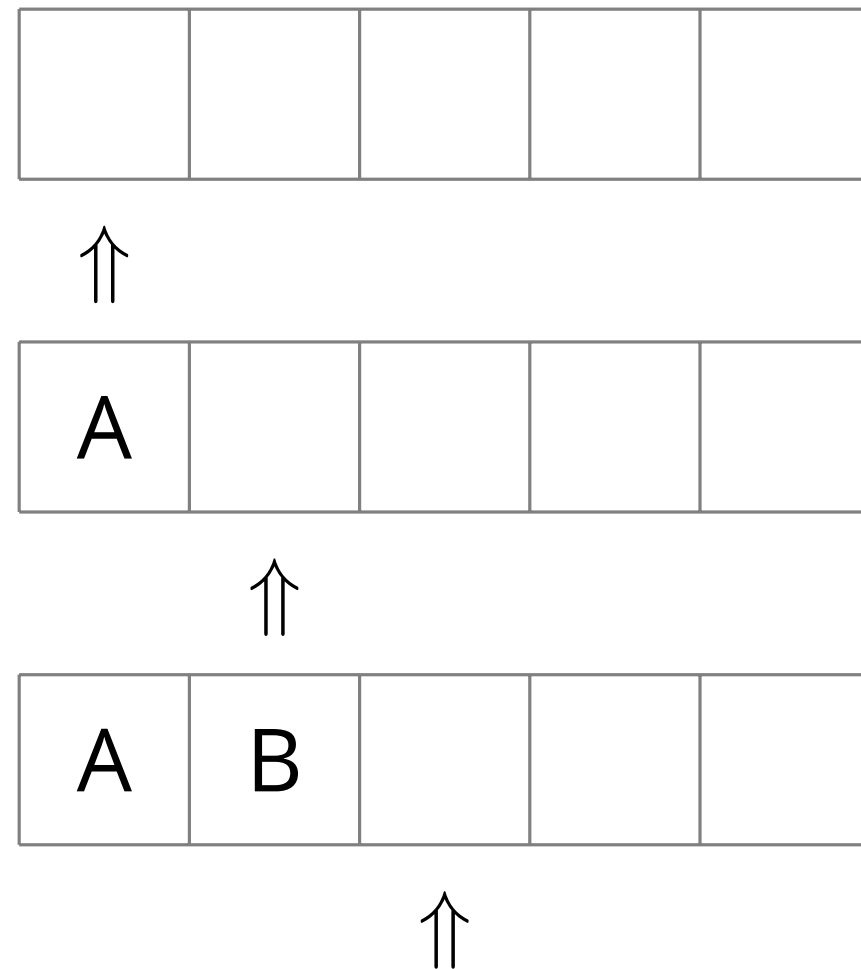


push(A)

- Keep track of position of the next free space in the array.
- Arrays have a fixed size.
  - Can't hold more values than we have space for.

# Array as a stack.

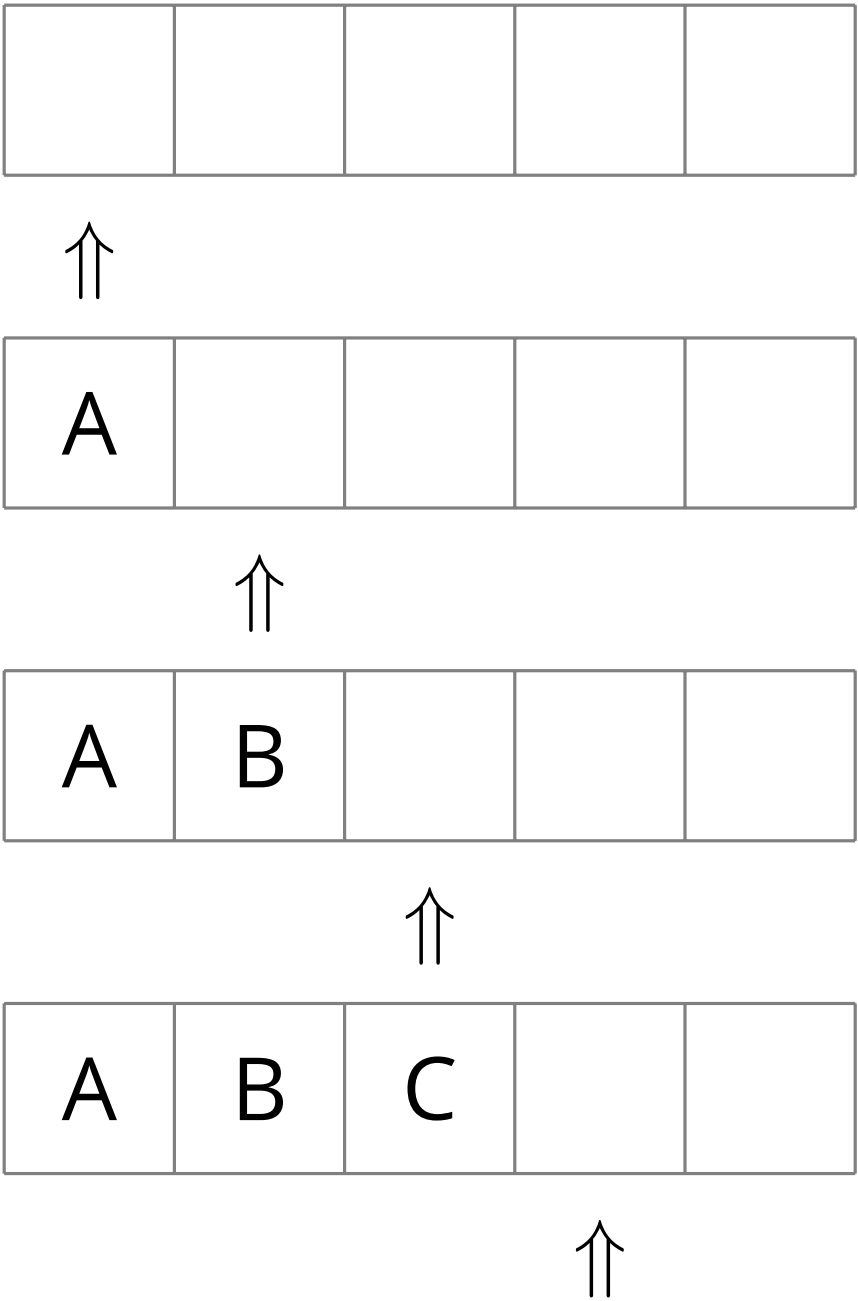
I



push(A)

push(B)

- Keep track of position of the next free space in the array.
- Arrays have a fixed size.
  - Can't hold more values than we have space for.



push(A)

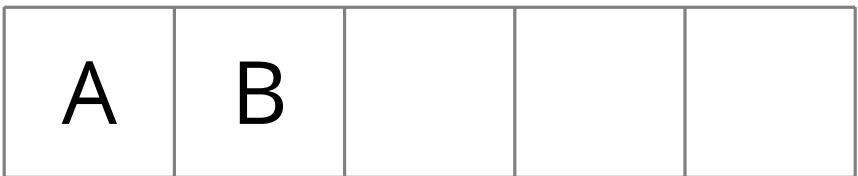
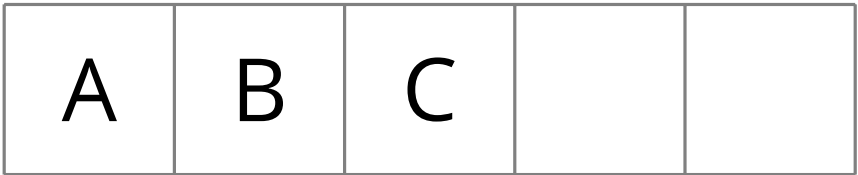
push(B)

push(C)

# Array as a stack.



- Keep track of position of the next free space in the array.
- Arrays have a fixed size.
  - Can't hold more values than we have space for.



push(A)

push(B)

push(C)

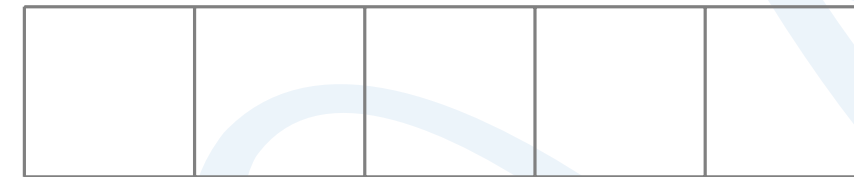
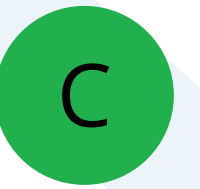
pop()

# Array as a stack.



- Keep track of position of the next free space in the array.
- Arrays have a fixed size.
  - Can't hold more values than we have space for.

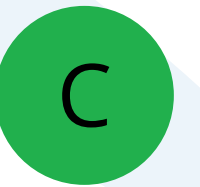
# Sets



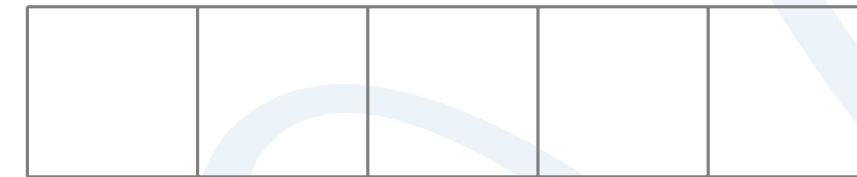
- An unordered ADT.
  - Items can be rearranged.
- Sets contain unique elements.
  - Can't contain duplicates.
- Can add items to a set.
- Can remove items from a set.
- Can see if an item is in a set.
- Can't get the  $n^{th}$  element.



# Sets

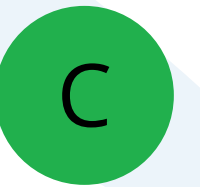


- An unordered ADT.
  - Items can be rearranged.
- Sets contain unique elements.
  - Can't contain duplicates.
- Can add items to a set.
- Can remove items from a set.
- Can see if an item is in a set.
- Can't get the  $n^{th}$  element.

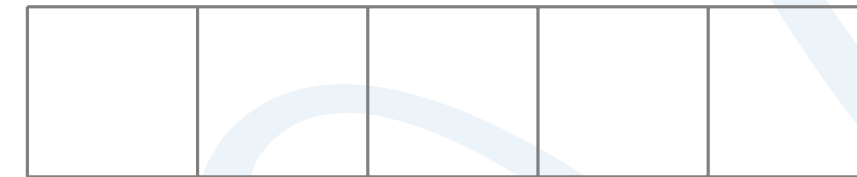


add(A)

# Sets



- An unordered ADT.
  - Items can be rearranged.
- Sets contain unique elements.
  - Can't contain duplicates.
- Can add items to a set.
- Can remove items from a set.
- Can see if an item is in a set.
- Can't get the  $n^{th}$  element.

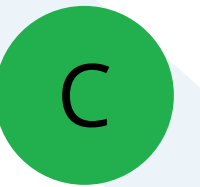


add(A)

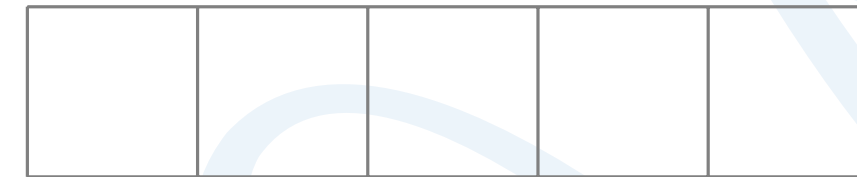


add(B)

# Sets



- An unordered ADT.
  - Items can be rearranged.
- Sets contain unique elements.
  - Can't contain duplicates.
- Can add items to a set.
- Can remove items from a set.
- Can see if an item is in a set.
- Can't get the  $n^{th}$  element.



add(A)

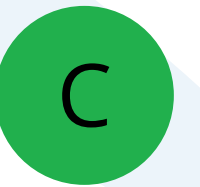


add(B)

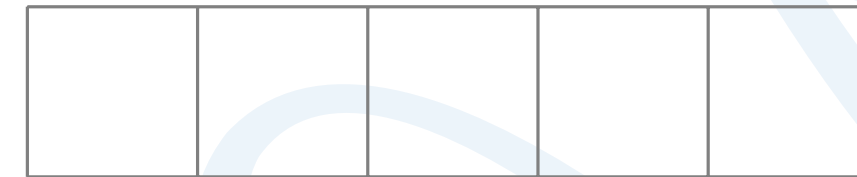


add(A)

# Sets



- An unordered ADT.
  - Items can be rearranged.
- Sets contain unique elements.
  - Can't contain duplicates.
- Can add items to a set.
- Can remove items from a set.
- Can see if an item is in a set.
- Can't get the  $n^{th}$  element.



add(A)



add(B)



add(A)



remove(A)

...and the others

C

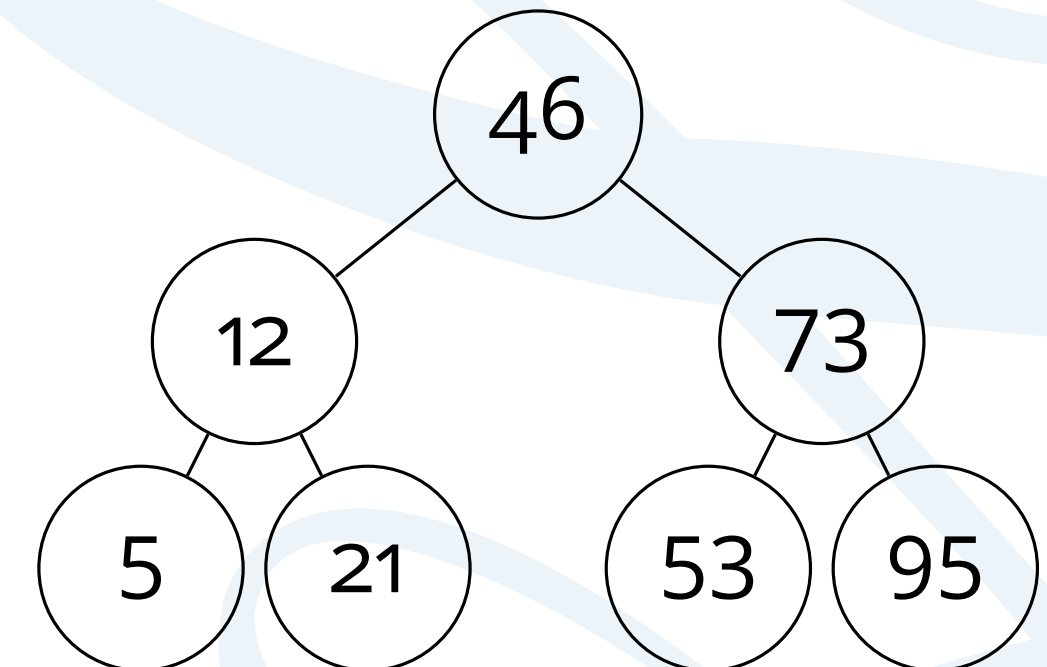
- Lots of other ADTs.
- Different names in different languages.
  
- Lists.
- Circular lists.
- Associative arrays.
  - Dictionaries/Maps.
- Double-ended queues.
- Trees.
- Graphs.

# Trees

A

## Variation on linked lists.

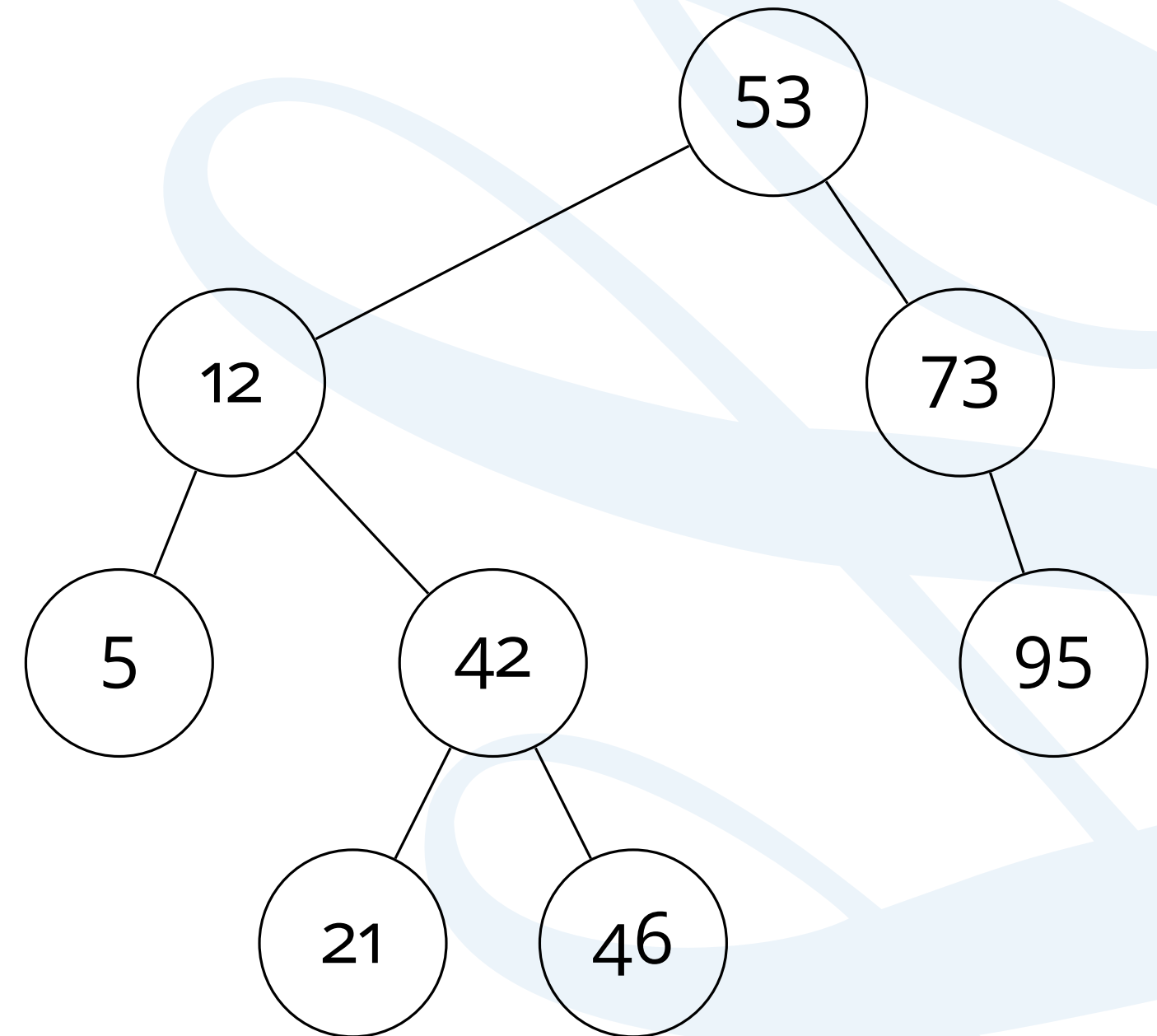
- Made of nodes and relationships.
- Root node at top.
- Each node can have  $\geq 0$  children.
- Binary search tree.
  - Very common type.
  - Ordered.
  - Max two children.
  - Binary searching.
  - Very good for sets.



# Balance

A

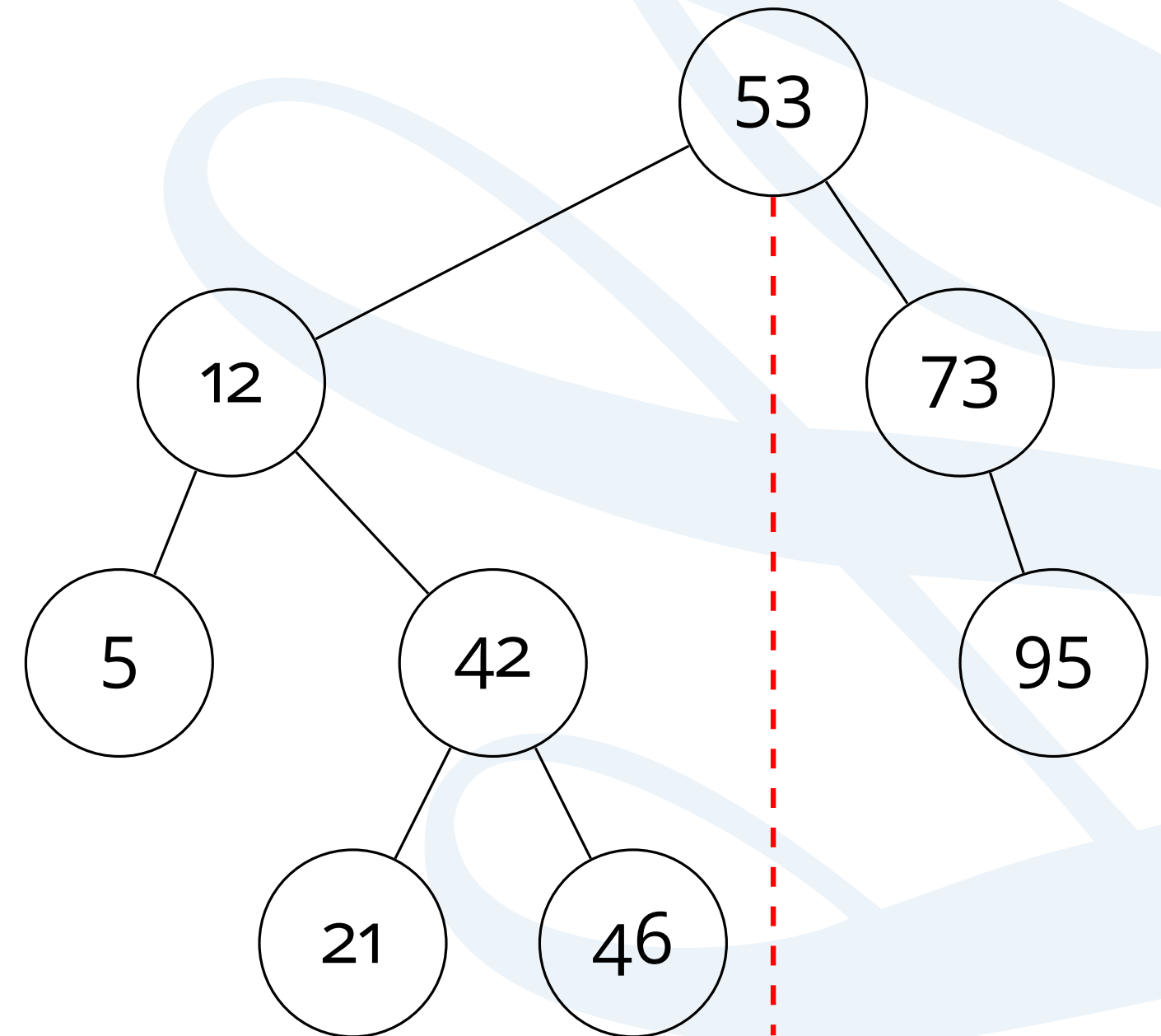
- Trees can be balanced or unbalanced.
- Not required for all trees.
- Going to be talking about BSTs from here on.
- Unbalanced because more than a one node difference between the two halves.



# Balance

A

- Trees can be balanced or unbalanced.
- Not required for all trees.
- Going to be talking about BSTs from here on.
- Unbalanced because more than a one node difference between the two halves.
  - For the whole tree...

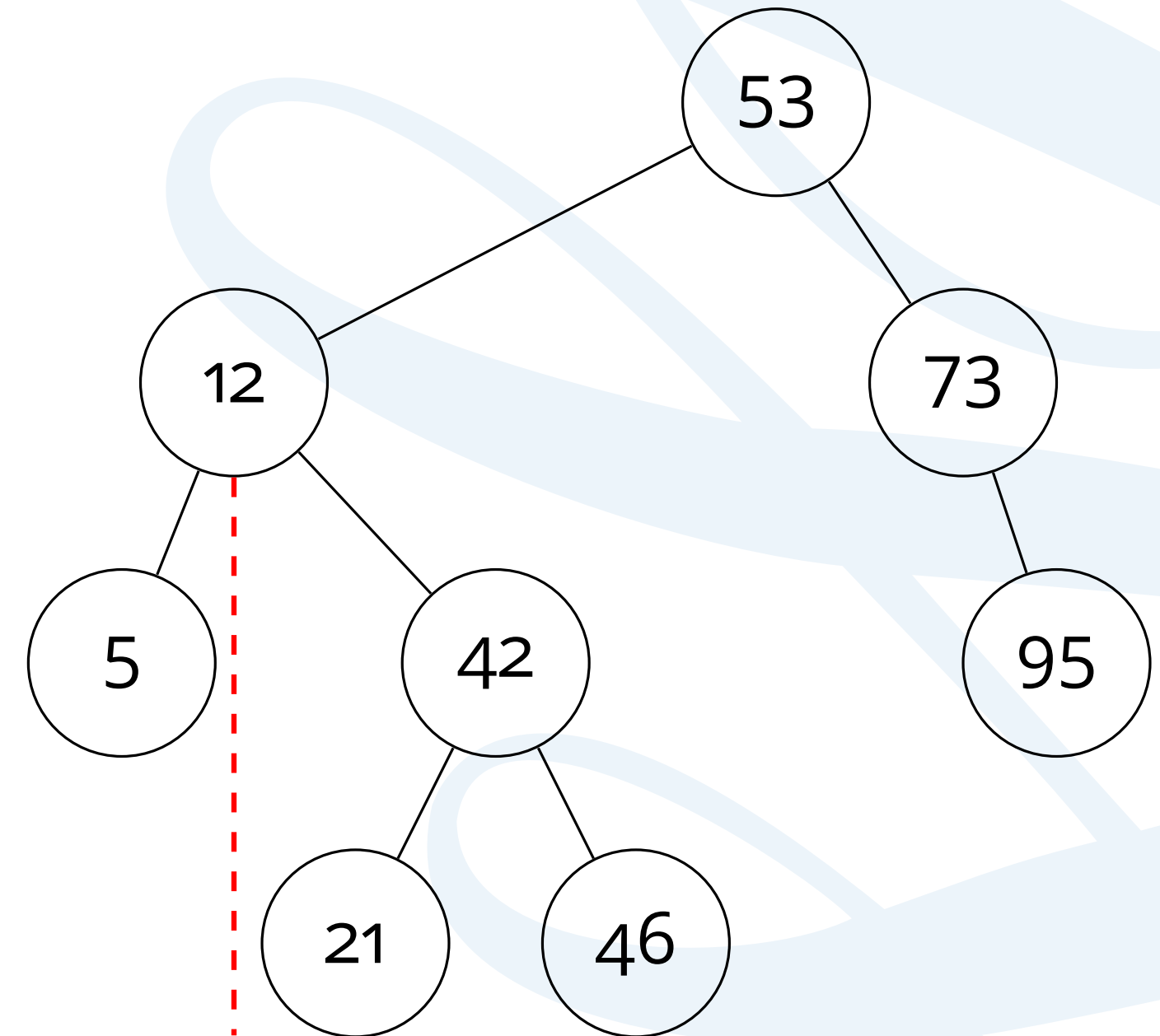




# Balance

A

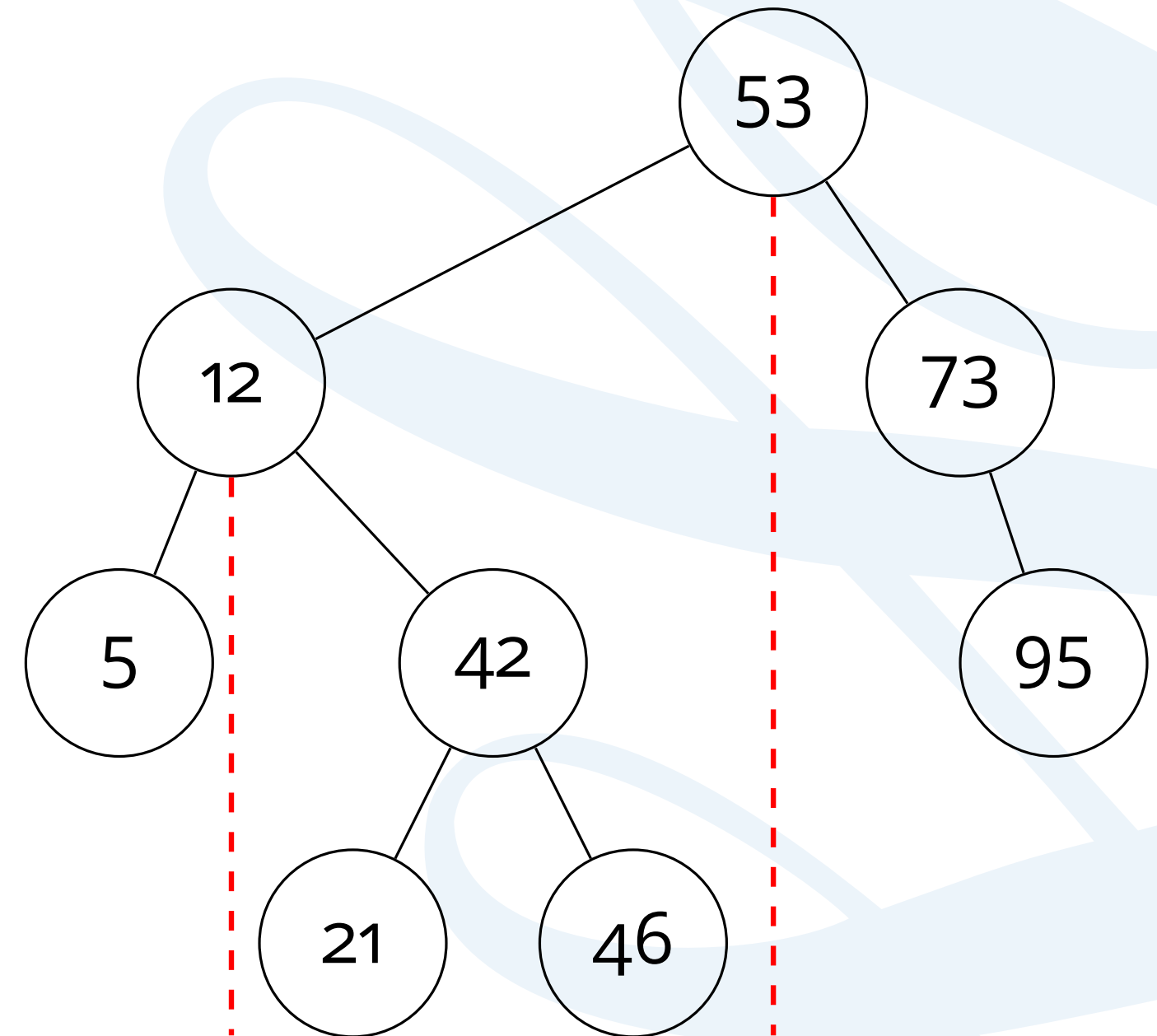
- Trees can be balanced or unbalanced.
- Not required for all trees.
- Going to be talking about BSTs from here on.
- Unbalanced because more than a one node difference between the two halves.
  - For the whole tree...
  - ...and one of the subtrees.



# Balance

A

- Trees can be balanced or unbalanced.
- Not required for all trees.
- Going to be talking about BSTs from here on.
- Unbalanced because more than a one node difference between the two halves.
  - For the whole tree...
  - ...and one of the subtrees.

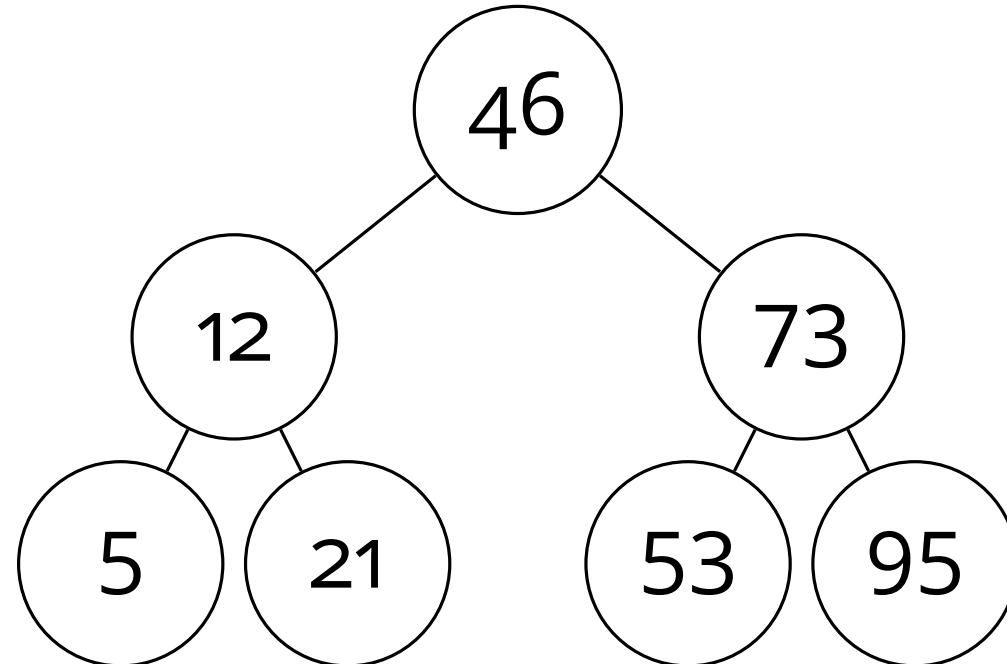


# Balance

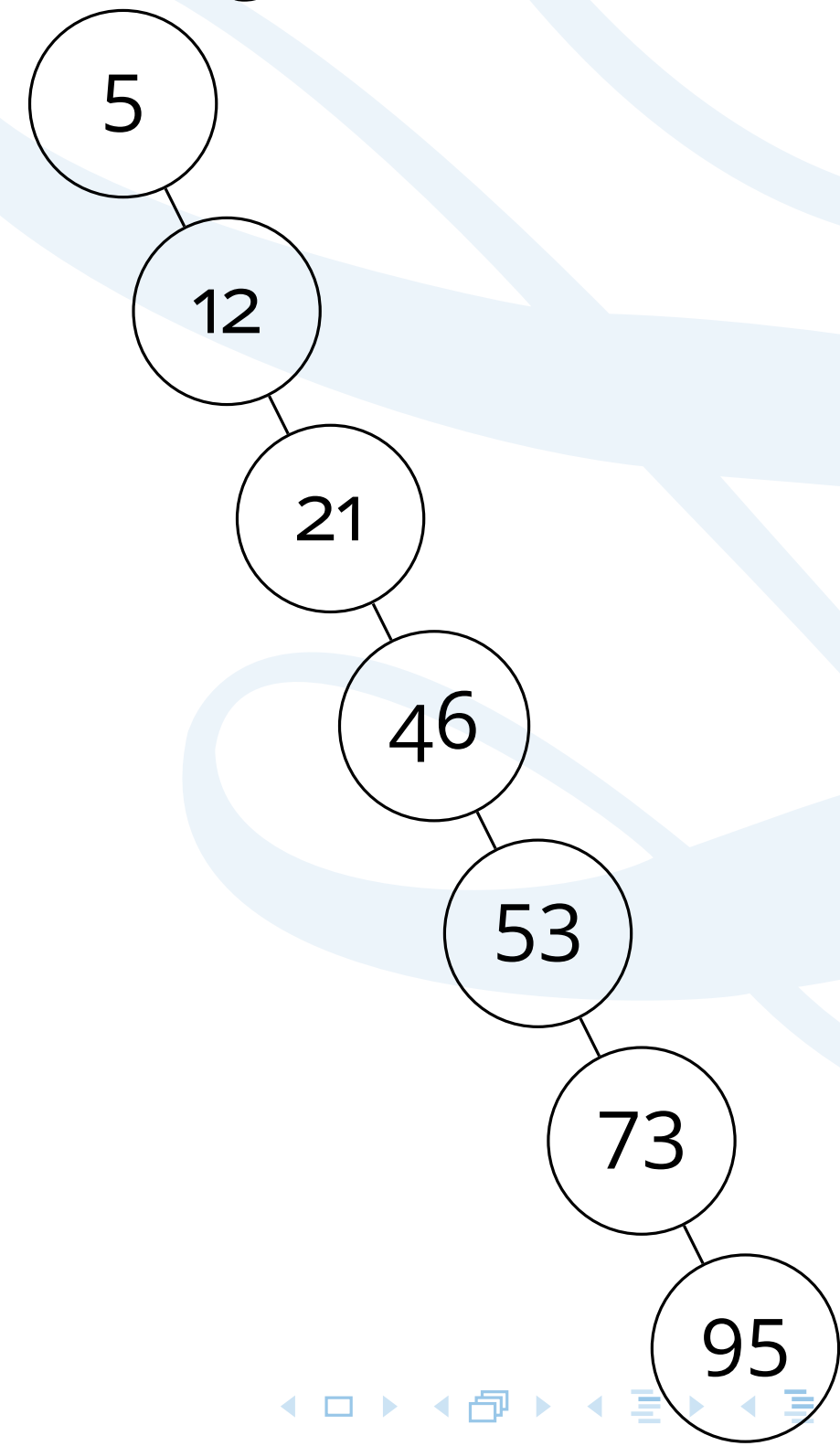
A

Important that you keep your BSTs balanced.

Perfect tree.



Degenerate tree.



Arrays

Linked lists

Array example

LL example

Data  
structures

Abstract data  
types

Queues

Stacks

Sets

Other

Trees

End

# Quiz

# Recap

- Arrays.
  - Advantages/disadvantages.
- Linked lists .
  - Advantages/disadvantages.
  - How to insert/delete.
- Difference between data structure and ADTs.
- Stack.
  - FILO.
  - Using an array as one.
  - Using a LL as one.

- Queue.
  - FIFO.
  - Using an array as one.
  - Using a LL as one.
- Sets.
  - No duplicates.
  - Unordered.
- Trees.
  - Balanced/unbalanced.

# The End