Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 6

Basic data structures: arrays and lists

Algorithms and Data Structures

So far we looked at algorithms for different kind of problems.

All our algorithms work with data of some sort, where the data is put in specific **data structures**.

Data structures are crucial as they help us design algorithms that are more concise and efficient.

In particular, we used the following basic data structures:

- arrays (e.g. searching and sorting algorithms)
- strings (e.g. finding length of longest palindrome)

In this lecture we will look closer at arrays and how we can expand them to become more usable.

The problem of structure in data structures

A data structure stores data

- it is relatively easy to simply store things
- what is tricky is to do this in a fast/reliable/useful way

This is where structure becomes important!

As a visual analogy, think of data as the contents in a soup.

We need structure to be able to fish the data from the soup in a useful way

pic from: thehappyfoodie.co.uk

Arrays

Arrays are the most common data type for storing data:

- an array is a sequence of elements of fixed length
- the elements of an array can be read and modified by using their index
- but the length of an array is fixed and cannot be changed

E.g. here is an array (call it `a`) of integers:

2	3	4	5	42

In this module we assume that indexing inside an array starts from 0 and ends at “length of the array minus 1”. So:

- the length of `a` is 5
- the elements of `a` are: `a[0]`,`a[1]`,`a[2]`,`a[3]`,`a[4]`
- we can read the first element of `a` e.g. by: `first = a[0]`
- we can modify the first element of `a` e.g. by: `a[0] = 24`

Arrays: pros and cons

Some pros of using arrays:

- easy to understand and use
- expressive enough for a wide range of tasks
- typically very fast to read/write (low-level, close to underlying hardware)
- few functionalities so not many things can go wrong

Some cons of arrays – too few functionalities:

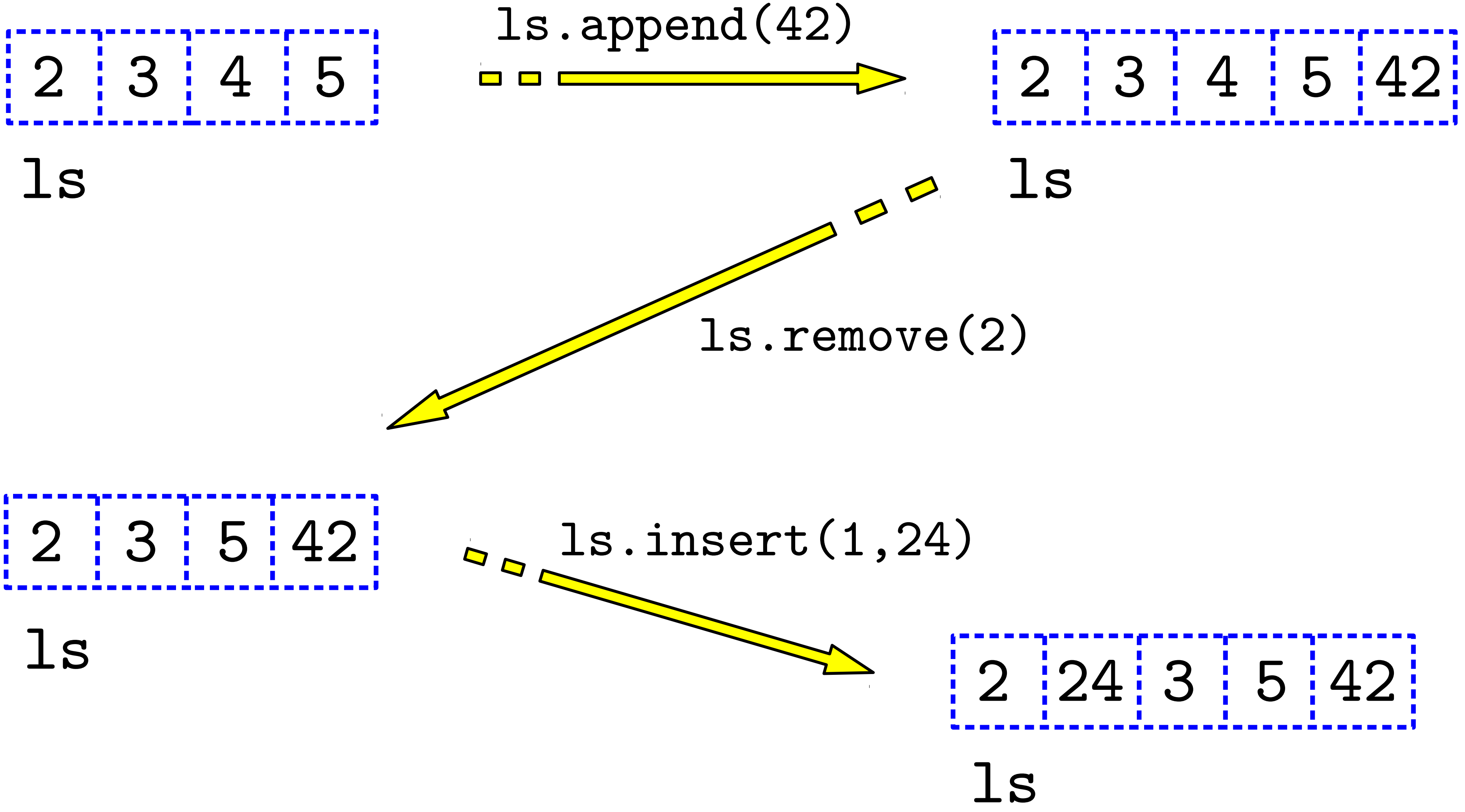
- fixed size, e.g. cannot enlarge an array
- sometimes we need more elaborate data structures than simply storing data in a row (i.e. the whole point of having data structures)

Lists

Another standard data structure is that of a **list**:

- the elements of a list are stored in a sequence
- we can read/write the elements of a list (not necessarily via indexes)
- we can add new elements at any position inside a list, or remove existing ones, changing the length of the list

Example list use



Lists we will look at

In fact, lists are a family of data structures, of which we will look at two representatives:

- **array lists:** an “extension” of arrays which also behave like lists
- **linked lists:** a new data structure based on chaining of elements

In the rest of this lecture we will look at array lists and how to implement them using arrays.

Array lists in Java

Array lists in Java are, well, ArrayLists:

```
ArrayList ls = new ArrayList<Integer>();
for (int i=2; i<6; i++)
   ls.add(i);
System.Out.println("initial list: " + ls);
ls.add(42);
ls.remove(2);
ls.add(1,24);
System.Out.println("final list: " + ls);
```

Array lists in Python

In the first lecture, we mentioned that Python arrays are actually *lists*, i.e. they have extra functionalities. These are in fact array lists.

```
ls = []
for i in range(2,6):
   ls.append(i)
print("initial list: " + str(ls))
ls.append(42)
ls.pop(2)
ls.insert(1,24)
print("final list: " + str(ls))
```

Functionalities of array lists

We require that array lists have the following functions:

- **create**: a function for creating a new array list of length 0
- **get**: for reading the `i`-th element of an array list
- **set**: for writing an element `e` into `i`-th element of an array list
- **length**: for retrieving the current length of an array list
- **append**: for appending an element `e` at the end of an array list, increasing its length by 1
- **insert**: for inserting an element `e` in the `i`-th position of an array list and moving all subsequent elements one position to the right, increasing its length by 1
- **remove**: for removing the element in the `i`-th position of an array list, returning it and moving all subsequent elements one position to the left, reducing its length by 1

actual array list implementations have (many) more functions, but for us the above 7 will do

Abstract data types

What we described so far are the functionalities of array lists, i.e. the functions available for accessing and modifying them.

Such a description is called an **Abstract Data Type (ADT)**.

ADTs and Data Structures are different sides of the same coin:

- an ADT gives us an outside view of a data object – tells us *how to use it*
- a Data Structure gives us the inside view of the object – shows us *how the object really works*

Now, our next question is how to implement the array lists ADT.

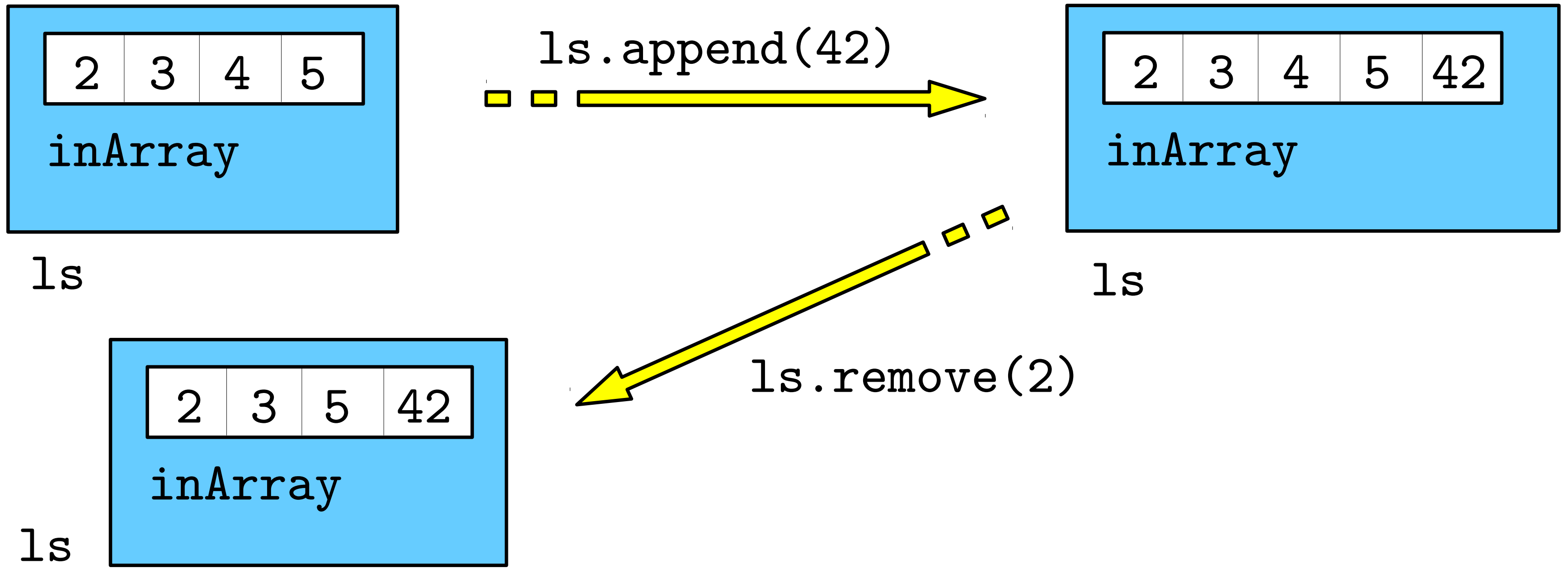
This basically boils down to:

how to build arrays that change size?

First (naive) attempt: a simple array

How to build arrays that change in size? Here is a first attempt:

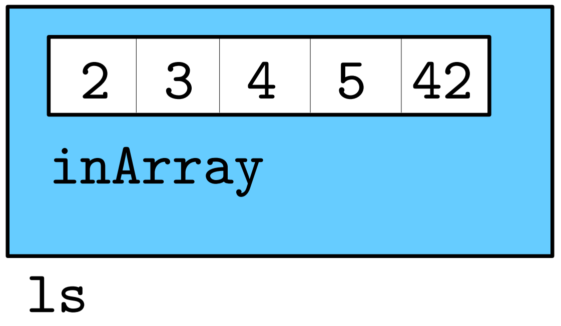
- An array list is simply an object containing an internal array
- We perform any read/write operations directly on the internal array
- To perform an operation that changes the length of the array, we make a copy of the array of the right size, perform the required operations on it, and finally make the new array our internal array



Class of naive array lists

Data structures are naturally organised as objects.

For example, a naive array list like:



is neatly implemented as an object of a class which has:

- methods `create`, `write`, `read`, etc.
- an internal variable `inArray` of type array

Let us call this class `NaiveArrayList`.

Objects in Python

A class in Python is defined as follows:

```
class BestPythonClassEver:
    classVar1 = "some class variable"
    ...
    def myFunction1(self, ...):
    ...
```

Each class function has always `self` as first argument. The `self` argument refers to the object the function is called on (like `this` in Java).

One of the class functions is the constructor, which is called `__init__`

In each class, there can be two kinds of variables:

- instance variables (one for each object), which are accessed via object-dot-name notation, e.g. `self.instanceVar`
- class variables (common to all objects), accessed directly via their name, e.g. `classVar`

Implementation of NaiveArrayList class

A (partial) implementation of `NaiveArrayList` could look as follows.

```
class NaiveArrayList:

    def __init__(self):
        self.inArray = []

    def get(self, i):
        return self.inArray[i]

    def set(self, e, i):
        self.inArray[i] = e

    def length(self):
        return len(self.inArray)

    def append(self, e):
        newArray = [0 for i in range(len(self.inArray)+1)]
        for i in range(len(self.inArray)):
            newArray[i] = self.inArray[i]
        newArray[self.len(inArray)] = e
        self.inArray = newArray
```

First attempt – inefficient

Naive array lists are inefficient:

- to make a copy of an array we need go through all its elements
- so, if the array list has length n, then the functions for appending, inserting and removing all have complexity $O(n)$
- this is not good if we intend to make many of these operations

We need a better idea:

- we can have a large internal array that contains the array list and some free space
- so, each addition does not need to create a new internal array, just use some of its unused space
- when the unused space gets used up, then create a much bigger internal array to replace the current one

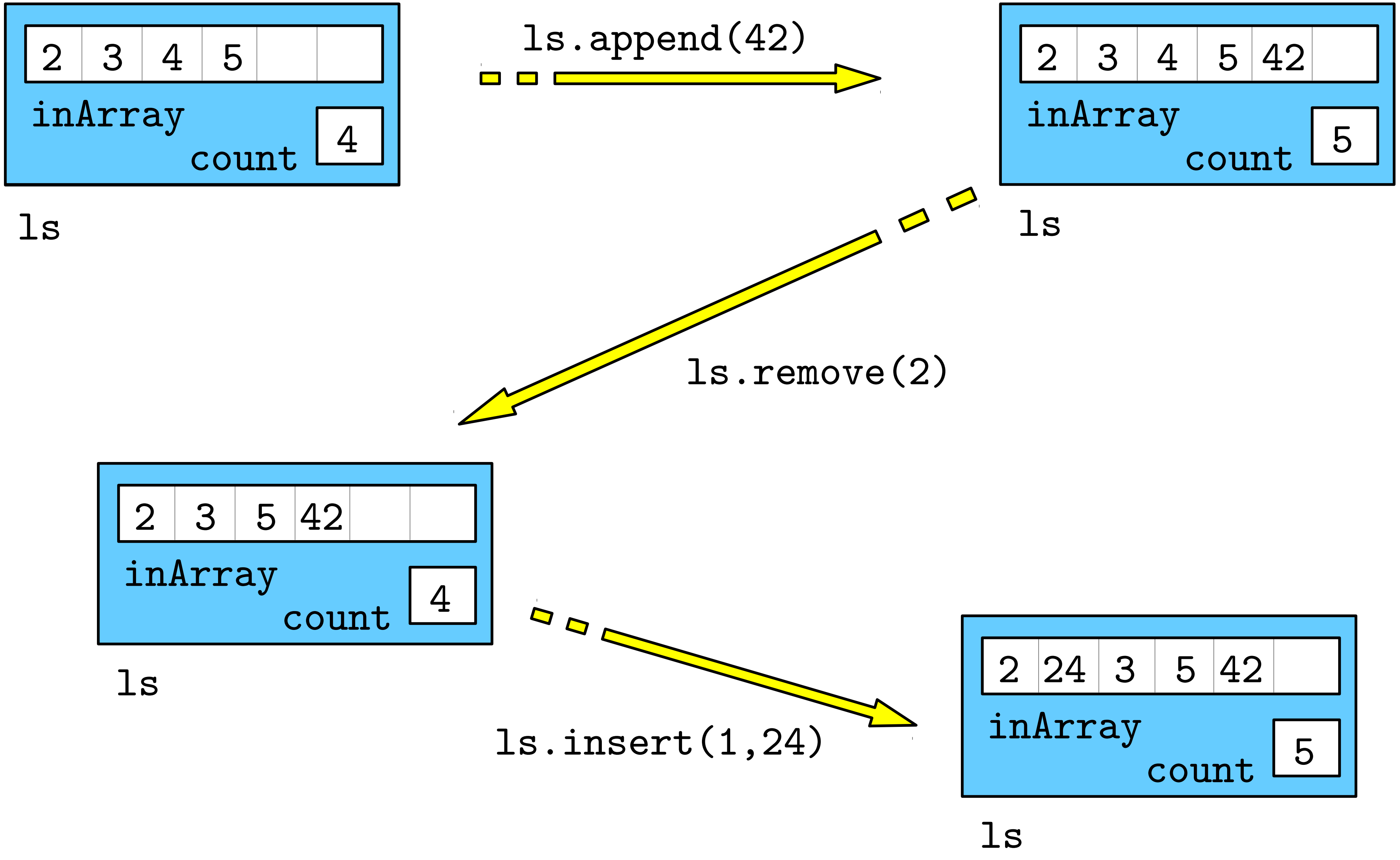
Second attempt: array and count

How to build arrays that change in size?

Second attempt:

- An array list is an object containing an internal array and a counter that remembers the used part of the internal array
- The internal array will typically have size larger than the count, so that it can accommodate addition of new elements
- We perform any read/write operations directly on the internal array
- To perform an operation that changes the size, we do not create a new internal array but, rather, work on the existing one and change the count value
- We need to resize if the count reaches the length of the internal array (the size of the internal array is sometimes called *capacity*)

Array and count



Array and count implementation

```
class ArrayList:

    def __init__(self):
        self.inArray = [0 for i in range(10)]    # capacity set to 10 for start
        self.count = 0

    def get(self, i):
        return self.inArray[i]

    def set(self, i, e):
        self.inArray[i] = e

    def length(self):
        return self.count

    def append(self, e):
        self.inArray[self.count] = e
        self.count += 1
        if len(self.inArray) == self.count:
            self._resizeUp()     # resize array if reached capacity
```

Array and count functions

```
def insert(self, i, e):
    for j in range(self.count,i,-1):
        self.inArray[j] = self.inArray[j-1]
    self.inArray[i] = e
    self.count += 1
    if len(self.inArray) == self.count:
        self._resizeUp()   # resize array if reached capacity

def remove(self, i):
    self.count -= 1
    val = self.inArray[i]
    for j in range(i,self.count):
        self.inArray[j] = self.inArray[j+1]
    return val

def _resizeUp(self):
    newArray = [0 for i in range(2*len(self.inArray))]
    for j in range(len(self.inArray)):
        newArray[j] = self.inArray[j]
    self.inArray = newArray
```

Array and count efficiency

Array and count is the standard implementation of array lists:

- typically, the internal array is also resized down when the count becomes too small (e.g. half than the capacity)
- if our array list has length n, then `count` is n and the functions `append`, `insert` and `remove` have complexity $O(n)$ in the worst case:
  - if `append` or `insert` triggers a resize then we will need to copy the full array, which takes n steps
  - if we `remove` the first element of the array then we need to move all remaining elements one position to the left, i.e. n-1 steps
- however, the worst case does not happen too often statistically – e.g. in most cases `append` needs just one step so is $\Theta(1)$

  In fact, each resize doubles the capacity of the array list, so we need to wait at least another n `append`'s before resizing again

Abstract Data Types vs Data Structures

The distinction between abstract data types (ADTs) and data structures is a subtle one and we will not be insisting on it a lot.

But it is an important distinction!

As a means of analogy, think of (filter) coffee machines:

- the ADT of a coffee machine is:

  `putCoffee,  putWater,  getSingleCoffee, …`

- the data structure is the actual build of the machine:
  - a heated water container connected with a coffee filter
  - a heating mechanism for the water
  - a pipe letting the water vapour condense and reach the filter
  - …

Using array lists

Array lists are extensions of arrays, so our array algorithms directly apply to array lists, we just need to adapt notation. Which means:

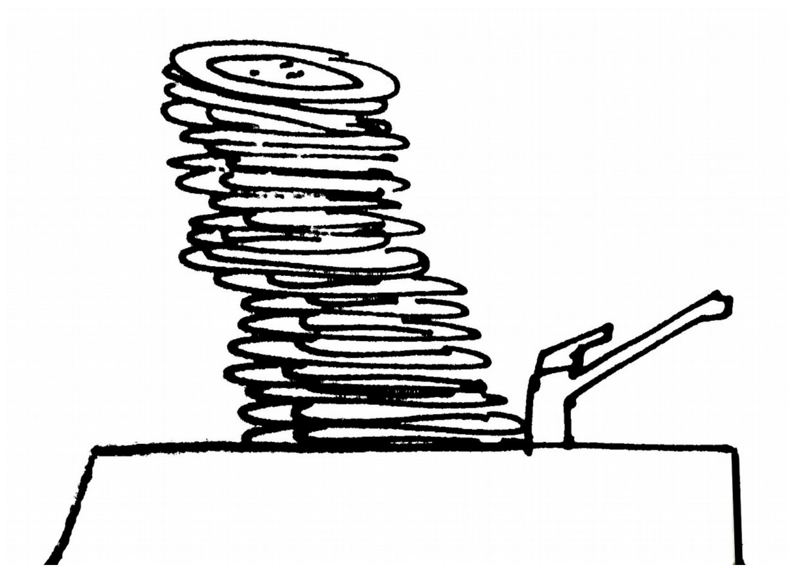
- reading is done via `A.read(i)` instead of `A[i]`,
- `len(A)` is replaced by `A.length()`, etc.

Also, some algorithms become simpler with array lists. For example:

```
def quicksort(ls):    # ls is an ArrayList
    pivot = ls.get(0)
    smaller = ArrayList()
    greater = ArrayList()
    for i in range(1,ls.length()-1):
        if ls.get(i) < pivot:
            smaller.append(ls.get(i))
        else:
            greater.append(ls.get(i))
    smaller = quicksort(smaller)
    greater = quicksort(greater)
    smaller.append(pivot)
    for i in range(0,greater.length()):
        smaller.append(greater.get(i))
    return smaller
```

Stacks and Queues

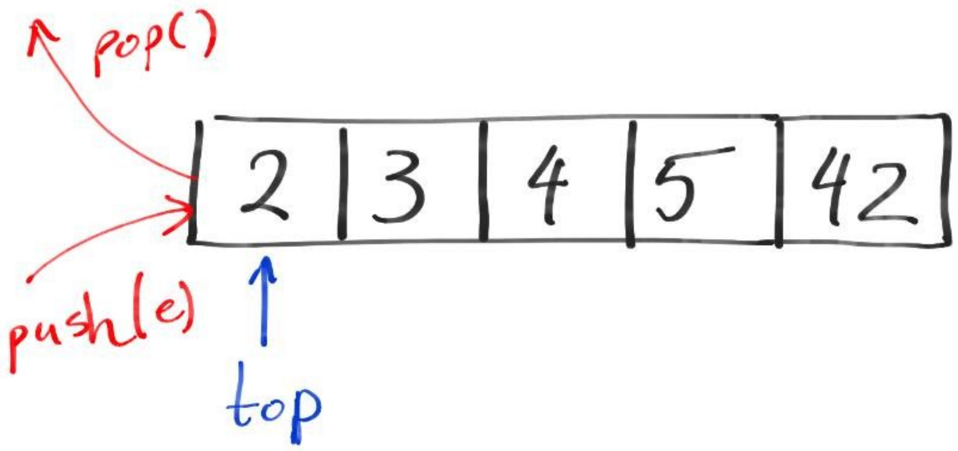
Two simpler array-like data structures with variable sizes:



stack

queue

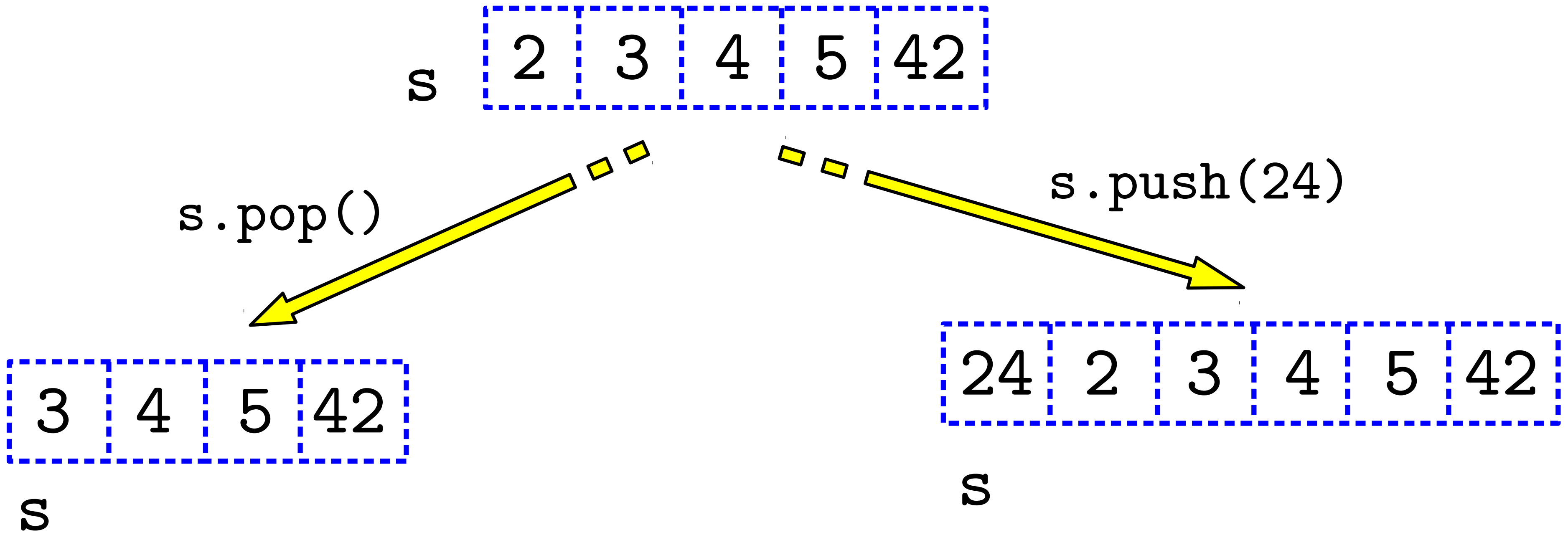
Stacks



LIFO

Last In

First Out

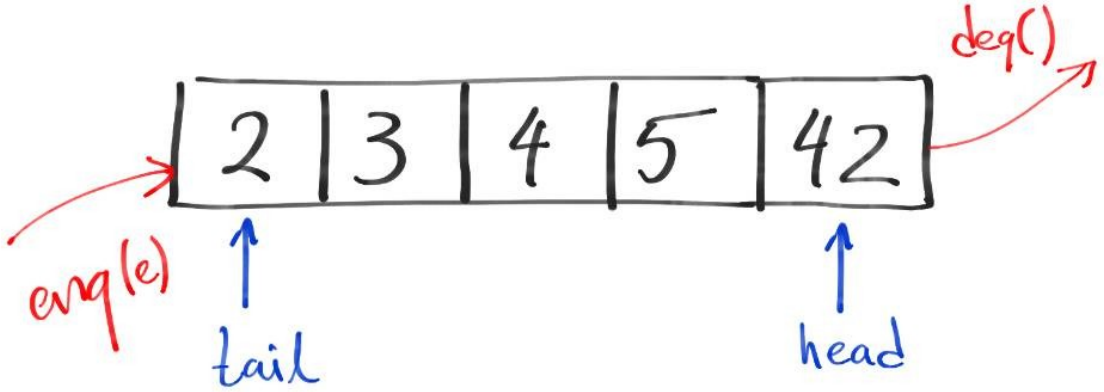


Stack ADT

We require that stacks have the following functions:

- **create**: a function for creating a new stack of size 0
- **size**: for retrieving the current size of a stack
- **push**: for adding an element at the top of a stack, increasing its size by 1
- **pop**: for removing the element at the top of the stack and returning it, decreasing its size by 1

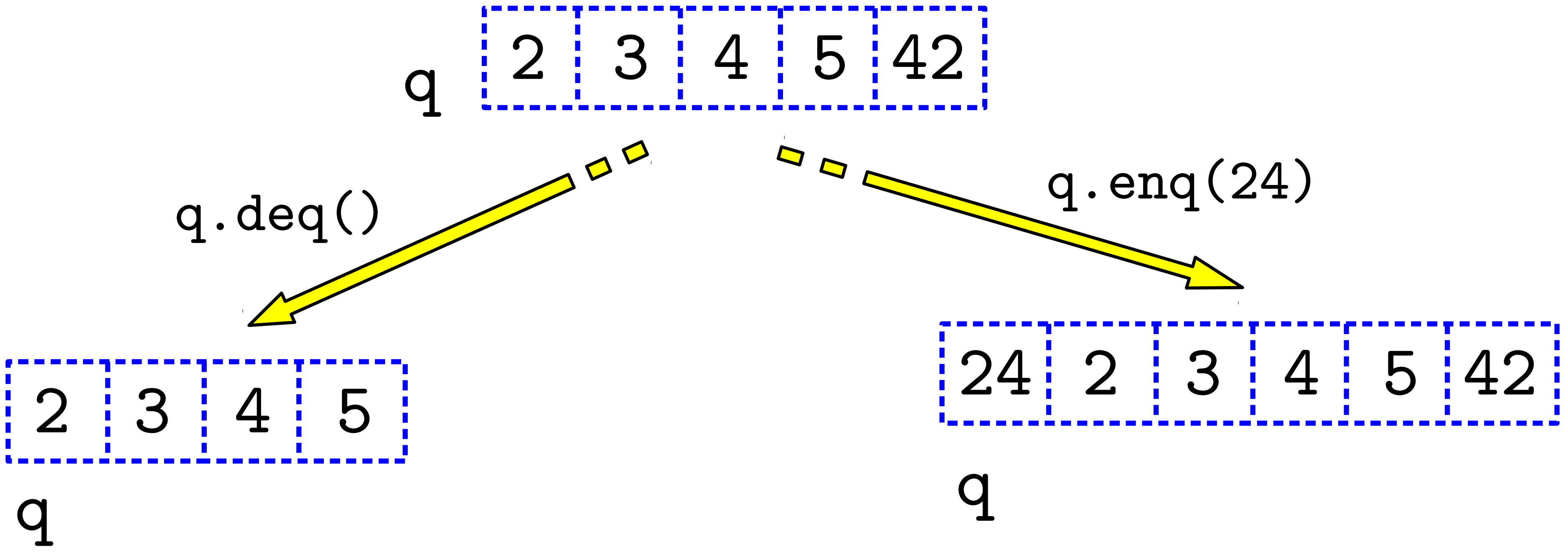
Queues



FIFO

First In

First Out



Queue ADT

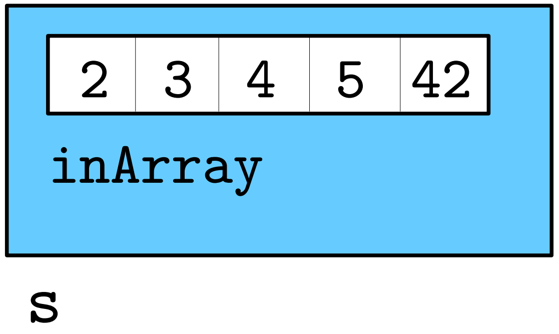
We require that queues have the following functions:

- **create**: a function for creating a new queue of size 0
- **size**: for retrieving the current size of a queue
- **enq**: for adding an element at the tail of a queue, increasing its size by 1
- **deq**: for removing the element at the head of the queue and returning it, decreasing its size by 1

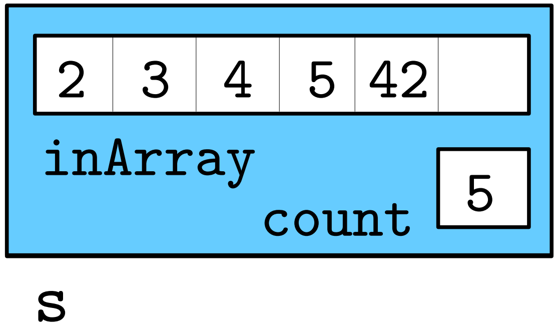
Stack implementation

Stacks are similar to array lists, only that we can only change them via push and pop.

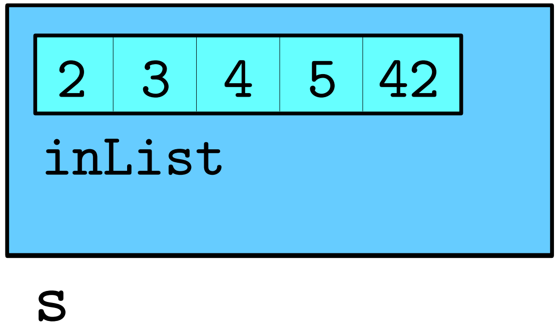
So, we can implement them as:



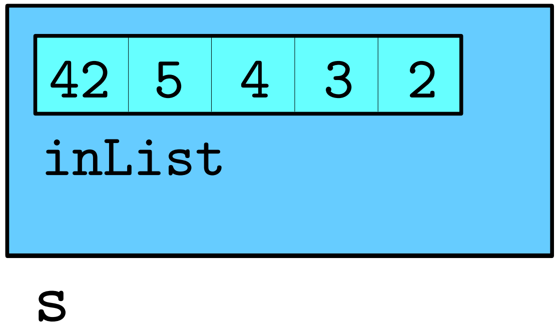
or



or



or, better still,



Stack implementations using ArrayLists

```
# more intuitive, less efficient

class Stack:

    def __init__(self):
        self.inList = ArrayList()

    def size(self):
        return self.inList.length()

    def push(self, e):
        self.inList.insert(0,e)

    def pop(self):
        return self.inList.remove(0)
```

```
# less intuitive, more efficient

class Stack:

    def __init__(self):
        self.inList = ArrayList()

    def size(self):
        return self.inList.length()

    def push(self, e):
        self.inList.append(e)

    def pop(self):
        top = self.inList.length()-1
        return self.inList.remove(top)
```

Efficiency of first implementation

In the first case, if the stack has size n, then the functions `push` and `pop` always have complexity $\Theta(n)$:

- `push` calls an `insert` at the beginning of the internal array list, which takes n steps (all elements need to be copied one position to the right).
  If the `insert` triggers a resizing up, then we need n+1 more steps
- `pop` calls a `remove` of the first element of the internal array list, which takes n-1 steps (all elements need to be copied one position to the left)

Efficiency of second implementation

In the second case, if the stack has size n, then the functions `push` and `pop` have lower complexities:

- `push` calls an `insert` at the end of the internal array list, which takes 1 step.

  If the `insert` triggers a resizing up, then we need n+1 more steps.

  So, the worst case is $\Theta(n)$ (due to resizing), but most often we get $\Theta(1)$

- `pop` calls a `remove` of the last element of the internal array list, which takes 1 step (simply reduce the count by 1)

Summary

We started looking at Data Structures: there are ways to structure data so that we can use them more efficiently

The simplest data structure is the array: it is simple but also has limitations

One major limitation of arrays is that their structure cannot be changed – we cannot change the length of an array

Lists overcome this limitation. In this case we looked at array lists and their implementation using array and count

We also saw two other simple array-like data structures: stacks and queues

Exercises

1. Let `ls` be the list `[2,3,4,5,42]`. Perform the following operations on `ls` in sequence:

   ```
   ls.insert(3,3)
   ls.remove(2)
   ls.append(23)
   print(ls.remove(3))
   ```

2. Write an ArrayList class function

   ```
   isEmpty(self)
   ```

   that returns `True` if the array list is empty, otherwise `False`

3. Write an ArrayList class function

   ```
   contains(self,e)
   ```

   that searches the array list and returns the first index with value `e`, or -1 if `e` is not in it

4. Let `q` be the list `[2,3,4,5,42]`. Perform the following operations on `q` in sequence:

   ```
   q.enq(3)
   q.deq(2)
   q.enq(23)
   print(q.deq())
   ```

5. Write an implementation of queues (using arrays or array lists). Recall that the ADT of queues has the following functions:
   - create: a function for creating a new queue of size 0
   - size: for retrieving the current size of a queue
   - enq: for adding an element at the tail of a queue, increasing its size by 1
   - deq: for removing the element at the head of the queue and returning it, decreasing its size by 1

6. We said that the length of an array is fixed. What does the following code do then?

   ```
   myArray = [0 for i in range(10)]
   myArray = [0 for i in range(20)]
   ```

7. Try to make your implementation of queues from Exercise 5 more efficient, so that enqueues and dequeues run in $\Theta(1)$ in most cases

Exercises – code

```
class ArrayList:

    def __init__(self):
        self.inArray = [0 for i in range(10)]
        self.count = 0

    def get(self, i):
        return self.inArray[i]

    def set(self, i, e):
        self.inArray[i] = e

    def length(self):
        return self.count

    def append(self, e):
        self.inArray[self.count] = e
        self.count += 1
        if len(self.inArray) == self.count:
            self._resizeUp()

    def insert(self, i, e):
        for j in range(self.count,i,-1):
            self.inArray[j] = self.inArray[j-1]
        self.inArray[i] = e
        self.count += 1
        if len(self.inArray) == self.count:
            self._resizeUp()

    def remove(self, i):
        self.count -= 1
        val = self.inArray[i]
        for j in range(i,self.count):
            self.inArray[j] = self.inArray[j+1]
        return val

    def _resizeUp(self):
        newArray = [0 for i in range(2*len(self.inArray))]
        for j in range(len(self.inArray)):
            newArray[j] = self.inArray[j]
        self.inArray = newArray


class Stack: # intuitive, not very efficient

    def __init__(self):
        self.inList = ArrayList()

    def size(self):
        return self.inList.length()

    def push(self, e):
        self.inList.insert(0,e)

    def pop(self):
        return self.inList.remove(0)
```

Exercises – solutions

1. Here they are:

```
ls = [2,3,4,5,42]
ls.insert(3,3) # [2,3,4,3,5,42]
ls.remove(2)   # [2,3,3,5,42]
ls.append(23)  # [2,3,3,5,42,23]
print(ls.remove(3)) # prints 5
```

2. Here is a possible solution:

```
def isEmpty(self):
    return self.count == 0
```

3. Here is a possible solution:

```
def contains(self,e):
    for i in range(self.count):
        if self.inArray[i] == e:
            return i
    return -1
```

4. Here they are:

```
q = [2,3,4,5,42]
q.enq(3)  # [3,2,3,4,5,42]
q.deq()   # [3,2,3,4,5]
q.enq(23) # [23,3,2,4,5]
print(q.deq()) # prints 5
```

5. There is a simple implementation that is very similar to the one for stacks:

```
class Queue:

    def __init__(self):
        self.inList = ArrayList()

    def size(self):
        self.inList.length()

    def deq(self):
        head = self.inList.length()-1
        return self.inList.remove(head)

    def enq(self,e):
        self.inList.insert(0,e)
```

6. We don’t change the array, we simply assign to the variable `myArray` a new array of different length