香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

DSAA 2043 - Design and Analysis of Algorithms

- Lecture Time: Thursday 4:30PM - 7:20PM, 02-SEP-2024 - 6-DEC-2024
- Lab Session: Tuesday 5:00PM - 5:50PM, 02-SEP-2024 - 6-DEC-2024
- Room: W1-101
- Instructor: Jing Tang, jingtang@hkust-gz.edu.cn or jingtang@ust.hk
- TAs
  - Xiaolong Chen, xchen738@connect.hkust-gz.edu.cn
  - Han Linghu, hlinghu866@connect.hkust-gz.edu.cn
  - Yifan Song, ysong853@connect.hkust-gz.edu.cn
  - Yihong Luo, yluocg@connect.ust.hk
  - Siya Qiu, sqiual@connect.ust.hk

Introduction

Contents (Tentative)

- Preliminaries (L1 – L2)
  - Python and basic data structures (Array, List, Stack, Queue)
  - Asymptotic complexity
- Sorting (L2 – L5)
  - Insertion sort, bubble sort
  - Merge sort, quick sort (Divide and Conquer paradigm)
  - Advanced data structures for sorti eaps, Hashing)

Mid-term!

- Graph Algorithms (L6 – L8)
  - BFS, DFS, SCC and Topological Sorting
  - Shortest Path Algorithms
  - Network Flow
- More algorithmic paradigms (L9 – L11)
  - Dynamic Programming
  - Greedy and minimum spanning tree
- Advanced topics (L12)
  - Complexity Analysis
  - Greedy for Approximation

References

Textbooks:

- Introduction to Algorithms. Cormen, Leiserson, Rivest, and Stein
- Algorithm Design. Kleinberg and Tardos
- The Algorithm Design Manual. Steven Skiena

Courses:

- MIT 6.006 - Introduction to Algorithms
- Stanford CS161 - Design and Analysis of Algorithms

Assessment (Tentative)

- Pre-class participation and class participation
- Labs (25%): work on lab exercises and submit by the deadline (each week)
- Project (20%): a large-scale programming exercise
- Mid-term exam (25%): closed book, written
- Final exam (30%): closed book, written

Pay attention to the academic integrity rules of this course

  - Plagiarism is zero tolerant

Recommended Learning Style

Goal: Encourage deeper and more diverse understanding via questioning, discussion, and teaching

- Prepare for the lecture
- Class participation. Questions and discussion welcome and to be rewarded
- Lab exercises
- Review contents timely (using, e.g., Feynman's method) Ask questions!

Tips:

- Learning by doing
- Good time management skills
- Ask (yourself/others) "why" and don't settle with one answer (even if it comes from an authority)

Advanced Python Functionalities

Python Learned Before

- Basics of the language
  - Control flow
- Basic datatypes:
  - Int, float, bool
  - List, dict, set
- Modules:
  - Importing and executing
  - Commonly used functions

Next Topics

- Function definition
- Positional and keyword arguments of functions
- Functions as objects
- Higher-order functions
- Namespaces and Scopes
- Object Oriented programming in Python
- Inheritance
- Iterators and generators

Most slides here are just for your references. You probably do not need to use them at the beginner's stage.

Functions in Python - Essentials

- Functions are first-class objects
- All functions return some value (possibly **None**)
- Function call creates a new namespace
- Parameters are passed by object reference
- Functions can have optional keyword arguments
- Functions can take a variable number of args and kwargs
- Higher-order functions are supported

Function Definition (1)

- Positional/keyword/default parameters

```
def sum(n,m):
    """ adds two values """
    return n+m

>>> sum(3,4)
7
>>> sum(m=5,n=3) # keyword parameters
8

#————————————————————————————————————

def sum(n,m=5): # default parameter
    """ adds two values, or increments by 5 """
    return n+m

>>> sum(3)
8
```

Function Definition (2)

- Arbitrary number of parameters (varargs)

```
def print_args(*items): # arguments are put in a tuple
    print(type(items))
    return items
>>> print_args(1,"hello",4.5)
<class 'tuple'>
(1, 'hello', 4.5)

def print_kwargs(**items): # args are put in a dict
    print(type(items))
    return items
```

Functions Are Objects

- As everything in Python, also functions are object, of **class function**

```
def echo(arg): return arg
type(echo)            # <class 'function'>
hex(id(echo))         # 0x1003c2bf8
print(echo)           # <function echo at 0x1003c2bf8>
foo = echo
hex(id(foo))          # '0x1003c2bf8'
print(foo)            # <function echo at 0x1003c2bf8>
isinstance(echo, object)        # => True
```

Function Documentation

- The comment after the function header is bound to the **__doc__** special attribute

```
def my_function():
    """Summary line: do nothing, but document it.
    Description: No, really, it doesn't do anything.
    """
    pass


print(my_function.__doc__)
# Summary line: Do nothing, but document it.
#
#     Description: No, really, it doesn't do anything.
```

Higher-Order Functions

- Functions can be passed as argument and returned as result
- Main combinators (**map, filter**) predefined: allow standard functional programming style in Python
- Heavy use of iterators, which support laziness
- Lambdas supported for use with combinators
- Lambda arguments: expression
  - The body can only be a single expression

Map

```
>>> print(map.__doc__)   % documentation
map(func, *iterables) --> map object
Make an iterator that computes the function using
arguments from each of the iterables.  Stops when the
shortest iterable is exhausted.
```

```
>>> map(lambda x:x+1, range(4))       % lazyness: returns
<map object at 0x10195b278>           % an iterator
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]          % map of a binary function
>>> z = 5             % variable capture
>>> list(map(lambda x : x+z, range(4)))
[5, 6, 7, 8]
```

Map and List Comprehension

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- **List comprehension** can replace uses of **map**

```
>>> list(map(lambda x:x+1, range(4)))
[1, 2, 3, 4]
>>> [x+1 for x in range(4)]
[1, 2, 3, 4]
>>> list(map(lambda x, y : x+y, range(4), range(10)))
[0, 2, 4, 6]    % map of a binary function
>>> [x+y for x in range(4) for y in range(10)] % multiple `for`
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5,...   % NO!
>>> [x+y for (x,y) in zip(range(4),range(10))] % OK
[0, 2, 4, 6]
>>> print(zip.__doc__)
zip(iter1 [,iter2 [...]]) --> zip object
Return a zip object whose .__next__() method returns a tuple where
the i-th element comes from the i-th iterable argument. The
.__next__() method continues until the shortest iterable in the
argument sequence is exhausted and then it raises StopIteration.
```

Filter (and List Comprehension)


```
>>> print(filter.__doc__)       % documentation filter(function
or None, iterable) --> filter object
Return an iterator yielding those items of iterable for which
function(item) is true. If function is None,
return the items that are true.
```

```
>>> filter(lambda   x : x % 2 == 0,[1,2,3,4,5,6])
<filter object at    0x102288a58>     % lazyness
>>> list(_)                            % '_' is the last value
 [2,  4,  6]
 >>>  [x for x in [1,2,3,4,5,6] if x % 2 == 0]
 [2,  4,  6] % same using list comprehension
% How to say "false" in Python
>>> list(filter(None,
        [1,0,-1,"","Hello",None,[],[1],(),True,False]))
[1, -1, 'Hello', [1], True]
```

More Modules for Functional Programming

- **functools**: Higher-order functions and operations on callable objects, including:
  - reduce(*function*, *iterable*[, *initializer*])
- **itertools**: Functions creating ***iterators*** for efficient looping. Inspired by constructs from APL, Haskell, and SML.
  - count(10) --> 10 11 12 13 14 ...
  - cycle('ABCD') --> A BCDA BCD ...
  - repeat(10, 3) --> 10 10 10
  - takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
  - accumulate([1,2,3,4,5]) --> 1 3 6 10 15

Decorators

- A **decorator** is any callable Python object that is used to modify a **function**, **method** or **class** definition
- A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition
- (Function) Decorators exploit Python **higher-order features**:
  - Passing functions as argument
  - Nested definition of functions
  - Returning function
- Widely used in Python (system) programming
- Support several features of meta-programming

Basic Idea: Wrapping a Function


```python
def my_decorator(func):          # function as argument
    def wrapper(): # defines an inner function
        print("Something happens before the function.")
        func() # that calls the parameter
        print("Something happens after the function.")
    return wrapper # returns the inner function
```

```python
def say_hello():    # a sample function
    print("Hello!")
# 'say_hello' is bound to the result of my_decorator
say_hello = my_decorator(say_hello) # function as arg
>>> say_hello()    # the wrapper is called
Something happens before the function.
Hello!
Something happens after the function.
```

Syntactic Sugar: The “Pie" Syntax

```
def my_decorator(func):          # function as argument
    def wrapper(): # defines an inner function
        ... # as before
    return wrapper # returns the inner function
```

```
def say_hello():            ## HEAVY! 'say_hello' typed 3x
    print("Hello!")
say_hello = my_decorator(say_hello)
```

- Alternative, equivalent syntax

```
@my_decorator
def say_hello():
    print("Hello!")
```

Another decorator: do_twice


```
def do_twice(func):
    def wrapper_do_twice():
        func()          # the wrapper calls the
        func()          #       argument twice
    return wrapper_do_twice
```

```
@do_twice                  # decorate the following # a
def say_hello():           sample function
    print("Hello!")
>>> say_hello()
                           # the wrapper is called
Hello!
Hello!
```

```
@do_twice                 #  does not work with parameters!!
def echo(str):            #  a function with one parameter
    print(str)
>>> echo("Hi...")         #  the wrapper is called
TypErr: wrapper_do_twice() takes 0 pos args but 1 was given
>>> echo()
TypErr: echo() missing 1 required positional argument: 'str'
```

do_twice for Functions with Parameters

- Decorators for functions with parameters can be defined exploiting ***args** and ****kwargs**

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
```

```
@do_twice
def say_hello():
    print("Hello!")
>>> say_hello()
Hello!
Hello!
```

```
@do_twice
def echo(str):
    print(str)
>>> echo("Hi... ")
Hi...
Hi...
```

General Structure of a Decorator

- Besides passing arguments, the wrapper also forwards the **result** of the decorated function
- Supports introspection redefining **_name_** and **_doc_**

```
import functools
def decorator(func):
    @functools.wraps(func)     #supports introspection
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

Example: Measuring Running Time

```
import functools, time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])

print(waste_some_time.__name__) # prints 'waste_some_time'
print(waste_some_time.__doc__)         #
```

Other Uses of Decorators

- Debugging: prints argument list and result of calls to decorated function
- Registering plugins: adds a reference to the decorated function, without changing it
- In a web application, can wrap some code to check that the user is logged in
- @staticmethod and @classmethod make a function invocable on the class name or on an object of the class
- More: decorators can be nested, can have arguments, can be defined as classes…

Example: Caching Return Values

```
import functools
from decorators import count_calls

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs): cache_key =
        args + tuple(kwargs.items()) if cache_key
        not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls    # decorator that counts the invocations
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

OOP in Python

- Typical ingredients of the Object Oriented Paradigm:

Encapsulation: dividing the code into a public **interface**, and a private **implementation** of that interface;

Inheritance: the ability to create **subclasses** that contain specializations of their parent classes.

Polymorphism: The ability to override methods of a Class by extending it with a subclass (inheritance) with a more specific implementation (**inclusion polymorphism**)

From https://docs.python.org/3/tutorial/classes.html:

- *"Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation."*

Defining a Class (Object)

- A class is a blueprint for a new data type with specific internal ***attributes*** (like a struct in C) and internal functions (***methods***).
- To declare a class in Python the syntax is the following:

```
class className:
    <statement-1>
   …
   <statement-n>
```

- **statements** are assignments or function definitions
- A new namespace is created, where all names introduced in the statements will go
- When the class definition is left, a ***class object*** is created, bound to **className**, on which two operations are defined: ***attribute reference*** and ***class instantiation***
- *Attribute reference* allows to access the names in the namespace in the usual way

Example: Attribute Reference on a Class Object

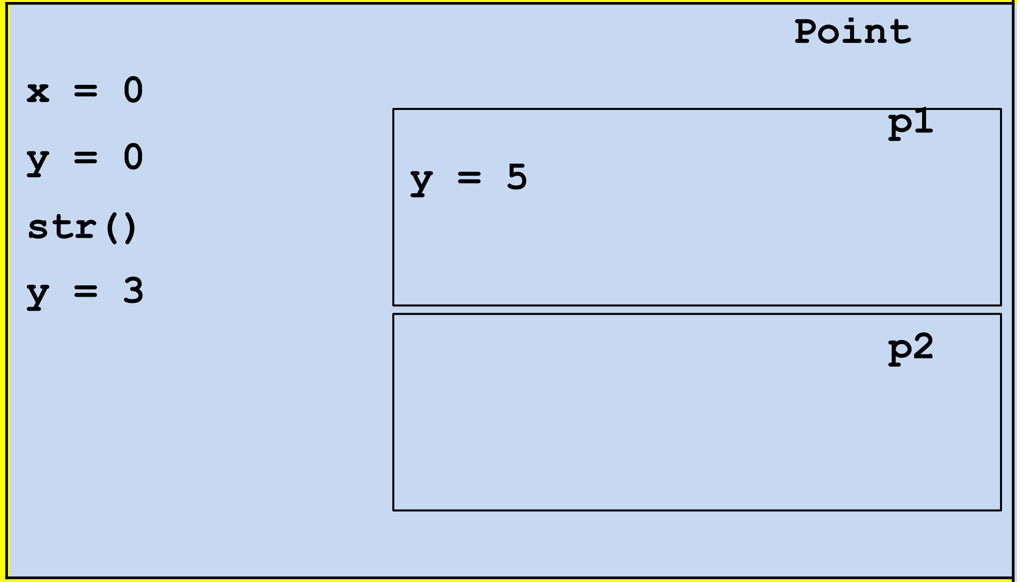

```
class Point:
  x = 0
  y = 0
  def str(): # no closure: needs qualified names to refer to x and y
      return  "x = " + (str) (Point.x) + ", y = " + (str) (Point.y)
#________
import ...
>>> Point.x
0
>>> Point.y = 3
>>> Point.z = 5 # adding new name
>>> Point.z
5
>>> def add(m,n):
      return m+n
>>> Point.sum = add # adding new function
>>> Point.sum(3,4)
7
```

```
                Point
x = 0
y = 0
str()
y = 3
z = 5
sum = add(m,n)
```

Creating a Class Instance

- A **class instance** introduces a **new namespace *nested in the class namespace***: by visibility rules all names of the class are visible
- If no *constructor* is present, the syntax of class instantiation is **className():** the new namespace is empty

```
class Point:
  x = 0
  y = 0
  def str():
      return   "x = "  + str(Point.x) +  ", y = " + str(Point.y)
#--------
>>> p1 = Point()
>>> p2 = Point()
>>> p1.y
0
>>> Point.y = 3
>>> print(p1.y, p2.y)
3 3
>>> p1.y = 5
>>> print(p1.y, p2.y)
5 3
```



Instance Methods

- A class can define a set of ***instance methods***, which are just functions:

```
def methodname(self, parameter1, ..., parametern):
         statements
```

- The first argument, usually called **self**, represents the ***implicit parameter*** (**this** in Java)
- A method ***must*** access the object's attributes through the **self** reference (eg. **self.x**) and the class attributes using **className.<attrName>** (or **self.__class__.<attrName>**)
- The first parameter must not be passed when the method is called. It is bound to the target object. Syntax:

```
obj.methodname(arg1, ..., argn):
```

- But it can be passed explicitly. Alternative syntax:

```
className.methodname(obj, arg1, ..., argn):
```

"Instance Methods"

- Any function ***with at least one parameter*** defined in a class can be invoked on an instance of the class with the dot notation.

```
class Foo
    def fun(par-0, par-1, ..., par-n):
          statements
#____
>>>obj = Foo()
>>>obj.fun(arg-1,...,arg-n) # is
syntactic sugar for
>>>obj.__class__.fun(obj,arg-1,...,arg-n)
```

- Since the instance **obj** is bound to the first parameter, **par-0** is usually called **self**.
- A name **x** defined in the (namespace of the) instance is accessed as **par-0.x** (i.e., usually **self.x**)
- A name **x** defined in the class is accessedas **className.x** (or **self.__class__.x**)

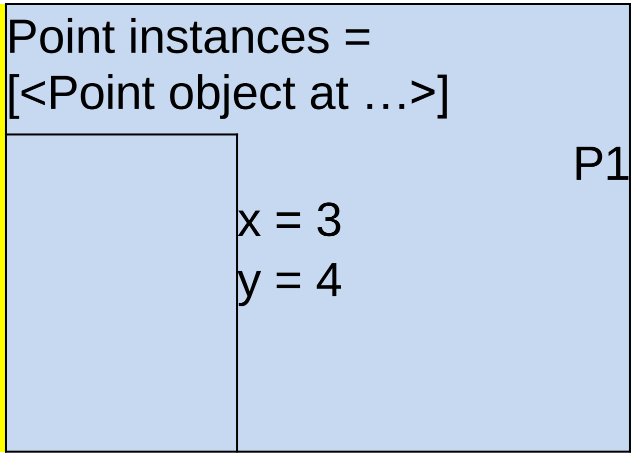
Constructors

- A constructor is a **special instance method** with name ____init______. Syntax:

```
def __init__(self, parameter_1, ..., parameter_n):
        statements
```

- Invocation: **obj = class Name(arg $_1$, ..., arg$_n$)**
- The first parameter **self** is bound to the new object.
- **statements** typically initialize (thus create) "instance variables", i.e. names in the new object namespace.
- Note: at most ONE constructor (**no overloading in Python**!)

```
class Point:
   instances = []
   def __init__(self, x, y):
      self.x = x
      self.y = y
      Point.instances.append(self)
#--------
>>> p1 = Point(3,4)
```



String Representation

- It is often useful to have a textual representation of an object with the values of its attributes. This is possible with the following instance method:

```
def __str__(self) :
        return <string>
```

- This is equivalent to Java's **toString** (converts object to a string) and it is invoked automatically when **str** or **print** is called.

Special Methods

- **Method overloading**: you can define special instance methods so that Python's built-in operators can be used with your class.

Binary Operators

Operator	Class Method
-	_sub_(self, other)
+	_add_(self, other)
*	_mul_(self, other)
/	_truediv_(self, other)

Operator	Class Method
==	_eq_(self, other)
!=	_ne_(self, other)
<	_lt_(self, other)
>	_gt_(self, other)
<=	_le_(self, other)
>=	_ge_(self, other)

Unary Operators

-	_neg_(self)
+	_pos_(self)

- Analogous to C++ overloading mechanism:

Pros: very compact syntax

Cons: may be more difficult to read if not used with care

(Multiple) Inheritance, in One Slide

- A class can be defined as a *derived class*

```
class derived(baseClass):
    statements
    statements
```

- No need of additional mechanisms: the namespace of **derived** is nested in the namespace of **baseClass**, and uses it as the next non-local scope to resolve names
- All instance methods are automatically virtual: lookup starts from the instance (namespace) where they are invoked
- Python supports **multiple inheritance**

```
class derived(base1,..., basen):
    statements
    statements
```

- **Method resolution order** (**MRO**) determines how to resolve a method (or an attribute) during multiple inherence
- Python 3: depth first, left-to-right order. C.f., https://www.geeksforgeeks.org/method-resolution-order-in-python-inheritance/

Encapsulation (and "Name Mangling")

- **Private** instance variables (not accessible except from inside an object)
  - **don't exist in Python.**

- **Convention:** a **name prefixed with underscore** (e.g. **_spam**) is treated as ***non-public part of the API*** (function, method or data member). It should be considered an implementation detail and subject to change without notice.

Name mangling

- Sometimes class-private members are needed to avoid clashes with names defined by subclasses. Limited support for such a mechanism, called *name mangling.*

- Any **name with at least two leading underscores and at most one trailing underscore** like e.g. _**spam** is textually replaced with **_class_spam**, where **class** is the current class name.

Example for Name Mangling

- Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = [] self.
        update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update # private copy of update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update() #
        but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Static Methods and Class Methods

- **Static methods** are simple functions defined in a class with no **self** argument, preceded by the **@staticmethod** decorator
- They are defined inside a class but they cannot access instance attributes and methods
- They can be called through both the class and any instance of that class
- **Benefits of static methods**: they allow subclasses to customize the static methods with inheritance. Classes can inherit static methods without redefining them.
- Class methods are similar to static methods but they have a first parameter which is the class name.
- Definition must be preceded by the **@classmethod** decorator
- Can be invoked on the class or on an instance.

Iterators

- An **iterator** is an object which allows a programmer to traverse through all the elements of a collection (**iterable** object), regardless of its specific implementation. In Python they are used implicitly by the **FOR** loop construct.
- Python iterator objects required to support two methods:
- **_iter_** returns the iterator object itself. This is used in **FOR** and **IN** statements.
- The **next** method returns the next value from the iterator. If there is no more items to return then it should raise a **StopIteration** exception.
- Remember that an iterator object can be used only once. It means after it raises
- **StopIteration** once, it will keep raising the same exception.
- Example:

```
for element in [1, 2, 3]:
        print(element)
```

```
>>> list = [1,2,3]
>>> it = iter(list)
>>> it
<listiterator object at 0x00A1DB50>
>>> it.next() 1
>>> it.next() 2
>>> it.next() 3
>>> it.next() -> raises StopIteration
```

Generators and Coroutines

- **Generators** are a simple and powerful tool for creating iterators.
- They are written like **regular functions** but use the **yield** statement whenever they want to return data.
- Each time the **next()** is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed).
- ***Anything that can be done with generators can also be done with class based iterators (not vice-versa).***
- What makes generators so compact is that the **_iter_()** and **next()** methods are created automatically.
- Another key feature is that the local variables and execution state are **automatically saved** between calls.

Generators (2)

- In addition to automatic method creation and saving program state, when generators terminate, they automatically raise **StopIteration**.
- In combination, these features make it easy to create iterators with no more effort than writing a regular function.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

#————————————————

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Typing in Python

- Dynamic, strong duck typing
- Code can be annotated with types

```
def greetings(name: str) -> str:
    return 'Hello' + name.
```

- Module **typing** provides runtime support for type hints
- Type hints can be checked statically by external tools, like **mypy**
- They are ignored by CPython

Duck Typing

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- "If it walks like a duck, and it quacks like a duck, then it must be a duck."
- The type or the class of an object is less important than the methods it defines. When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute.

```
>>> class TheHobbit:
...     def __len__(self):
...         return 95022
...
...
>>> the_hobbit = TheHobbit()

>>> the_hobbit
<__main__.TheHobbit object at 0x108deeef0>

>>> len(the_hobbit)
95022
```

Dynamic Adding Methods


```python
class User:
    pass

# Add instance attributes dynamically
jane = User()
jane.name = "Jane Doe"
jane.job = "Data Engineer"
jane.__dict__  # {'name': 'Jane Doe', 'job': 'Data Engineer'}
```

```
{'name': 'Jane Doe', 'job': 'Data Engineer'}
```

```python
# Add methods dynamically
def __init__(self, name, job):
    self.name = name
    self.job = job

User.__init__ = __init__
User.__dict__  # mappingproxy({'__init__': <function __init__ at 0x1036ccae0>})
```

```
mappingproxy({'__module__': '__main__',
              '__dict__': <attribute '__dict__' of 'User' objects>,
              '__weakref__': <attribute '__weakref__' of 'User' objects>,
              '__doc__': None,
              '__init__': <function __main__.__init__(self, name, job)>})
```

```python
linda = User("Linda Smith", "Team Lead")
linda.__dict__  # {'name': 'Linda Smith', 'job': 'Team Lead'}
```

```
{'name': 'Linda Smith', 'job': 'Team Lead'}
```

Miscellaneous

- Overloading: forbidden, but not necessary
- Overriding: ok, thanks to namespaces
- Generics: type hints support generics

Criticisms to Python: Syntax of Tuples


```
>>> type((1,2,3))
<class 'tuple'>
>>> type(())
<class 'tuple'>
>>> type((1))
<class 'int'>
>>> type((1,))
<class 'tuple'>
```

- Tuples are made by the commas, not by ( )
- With the exception of the empty tuple…

Criticisms to Python: Indentation

- Lack of brackets makes the syntax "weaker" than in other languages: accidental changes of indentation may change the semantics, leaving the program syntactically correct.

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
        foo(x - 1)
```

```
def foo(x):
    if x == 0:
        bar()
        baz()
    else:
        qux(x)
    foo(x - 1)
```

- Mixed use of tabs and blanks may cause bugs almost impossible to detect

Criticisms to Python: Indentation

- Lack of brackets makes it harder to refactor the code or insert new one
- "When I want to refactor a bulk of code in Python, I need to be very careful. Because if lost, I'm not sure what I'm editing belongs to which part of the code. Python depends on indentation, so if I have mistakenly removed some indentation, I totally have no idea whether the correct code should belong to that **if** clause or this **while** clause."
- Will Python change in the future?

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>>
```

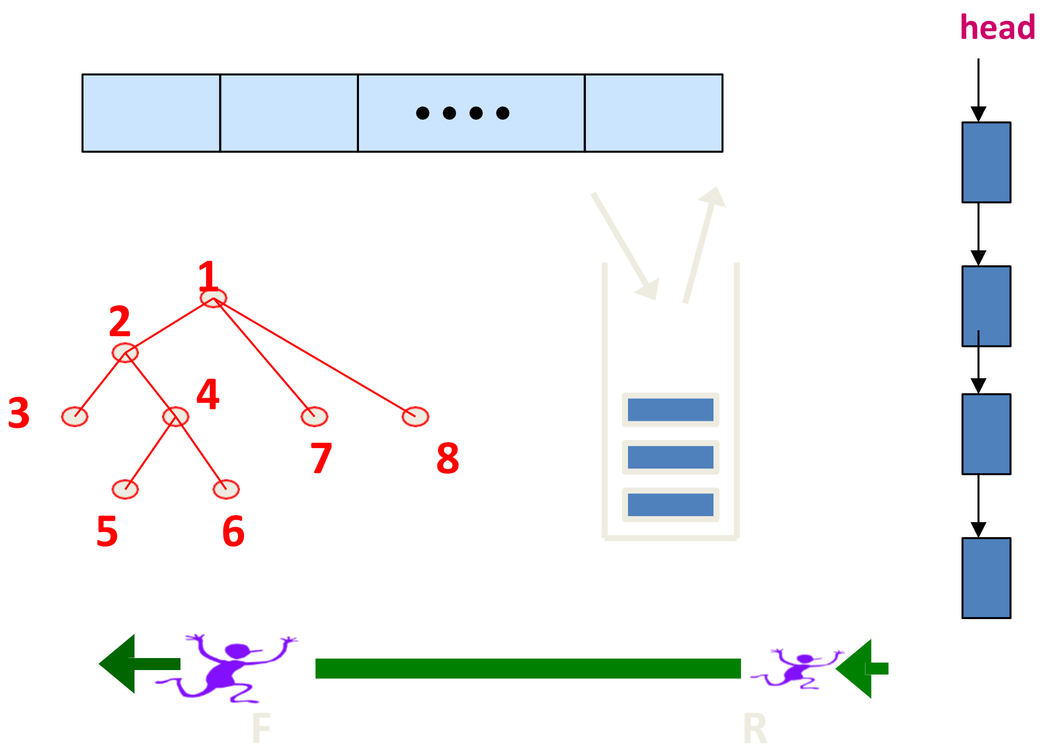
Builtin & Libraries

- The Python ecosystem is extremely rich and in fast evolution
- For available functions, classes and modules browse:
  - **Builtin Functions**
    - https://docs.python.org/3.8/library/functions.html
  - **Standard library**
    - https://docs.python.org/3.8/tutorial/stdlib.html
- There are dozens of other libraries, mainly for scientific computing, machine learning, computational biology, data manipulation and analysis, natural language processing, statistics, symbolic computation, etc.

Basic Data Structures

Elementary Data "Structures"

- Arrays
- Lists
- Stacks
- Queues
- Trees

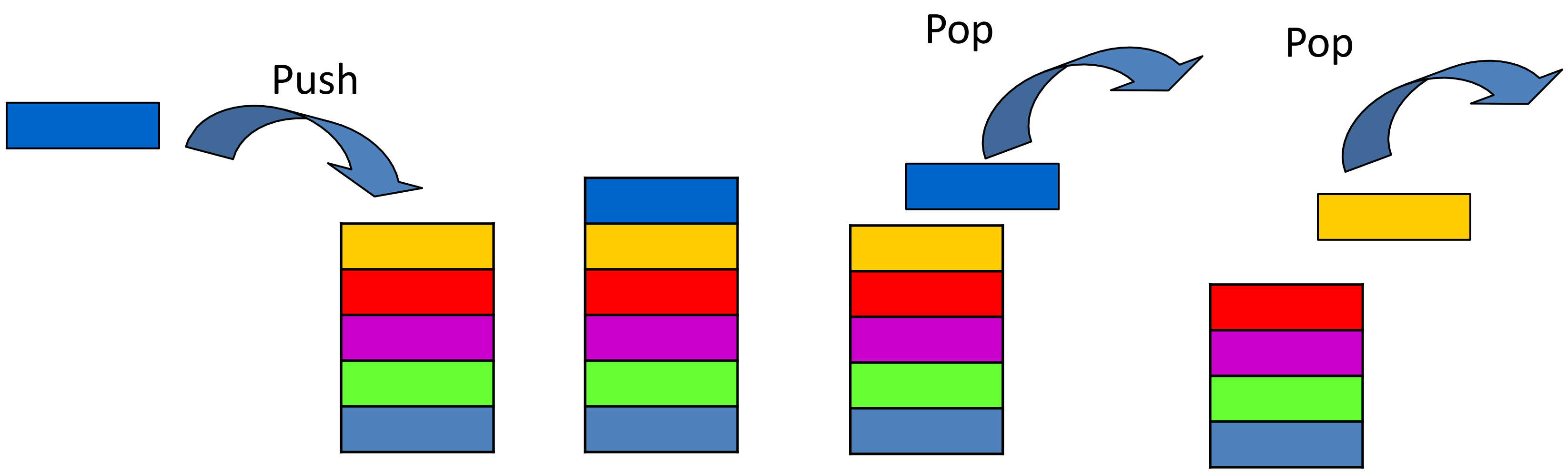


In some languages these are basic data types – in others they need to be implemented

Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

– LIFO – Last in, First out



What is This Good for ?

- Page-visited history in a Web browser

What is This Good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor

What is This Good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another

Abstract Data Type

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

A mathematical definition of objects, with operations defined on them

Examples

- Basic Types
  - integer, real (floating point), boolean (0,1), character
- Arrays
  - A[0..99] : integer array

	0	1	2	3	4	5	6	7					99
A	2	1	3	3	2	9	9	6	…				10

  - A[0..99] : array of images

0 1 2 99

ADT: Array

A mapping from an index set, such as {0,1,2,…,n}, into a cell type

Objects: set of cells

Operations:

- **create(A,n)**
- **put(A,v,i)** or **A[i] = v**
- **value(A,i)**

Also the "general" definition for functions

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure

- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

ADT for Stock Trade

- The data stored are buy/sell orders
- The operations supported are
  - order buy(stock, shares)
  - order sell(stock, shares)
  - void cancel(order)
- Error conditions:
  - Buy/sell a nonexistent stock
  - Cancel a nonexistent order

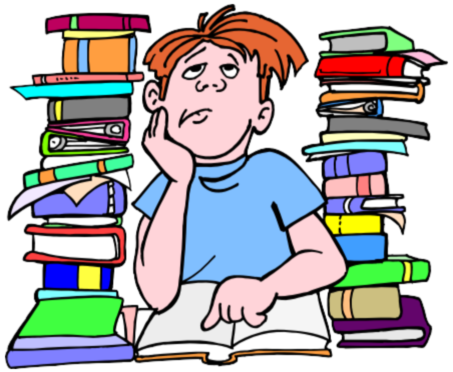
Stack ADT

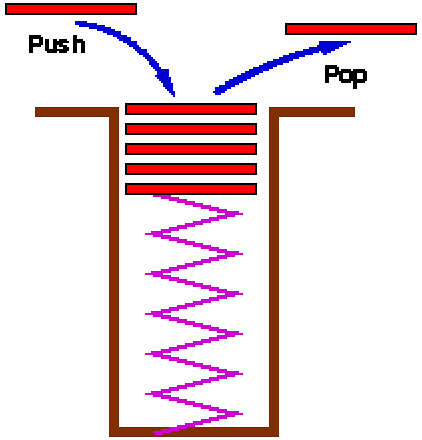
Objects:

A finite sequence of nodes

Operations:

- Create
- Push: Insert element at top
- Top: Return top element
- Pop: Remove and return top element
- IsEmpty: test for emptyness





Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Exercise: Stacks

- Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)

Array-based Stack

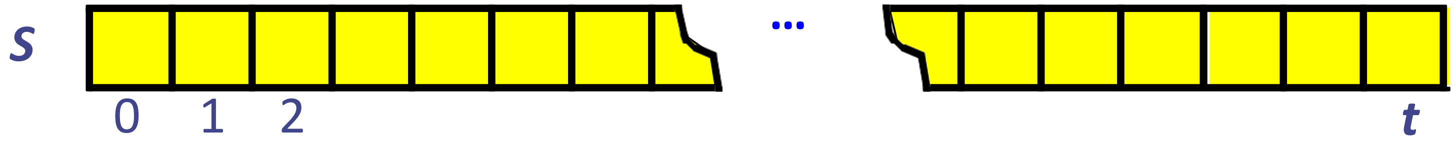
- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
  return t + 1

Algorithm pop()
  if empty() then
    throw EmptyStackException
   else
    t = t - 1
    return S[t + 1]
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
  if t = S.length - 1 then
    throw FullStackException
   else
    t = t + 1
    S[t] = o
```



Performance and Limitations

(array-based implementation of stack ADT)

- Performance
  - Let ***n*** be the number of elements in the stack
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)
- Limitations
  - The maximum size of the stack must be defined *a priori*, and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

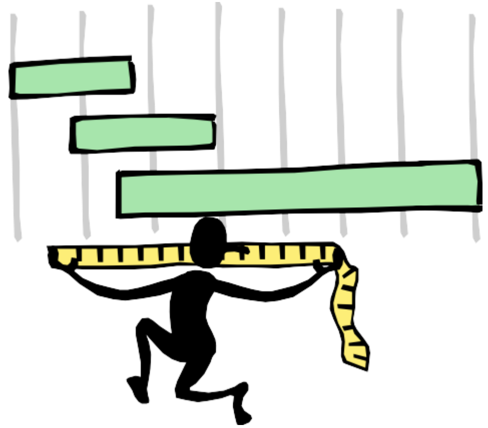
Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant ***c***
  - doubling strategy: double the size

```
Algorithm push(o)
  if t = S.length - 1
then
    A = new array of
          size …
    for i = 0 to t do
       A[i] = S[i]
       S = A
  t = t + 1
  S[t] = o
```

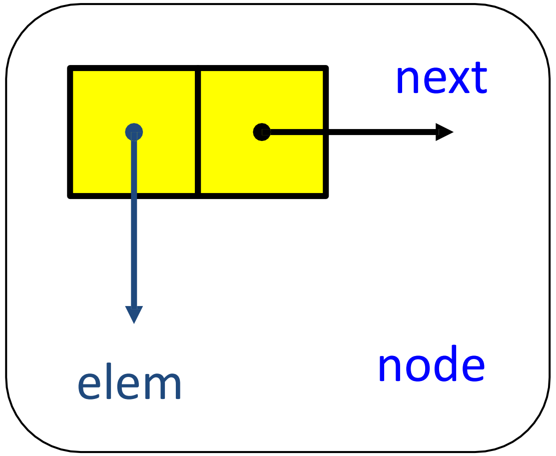
Comparison of the Strategies

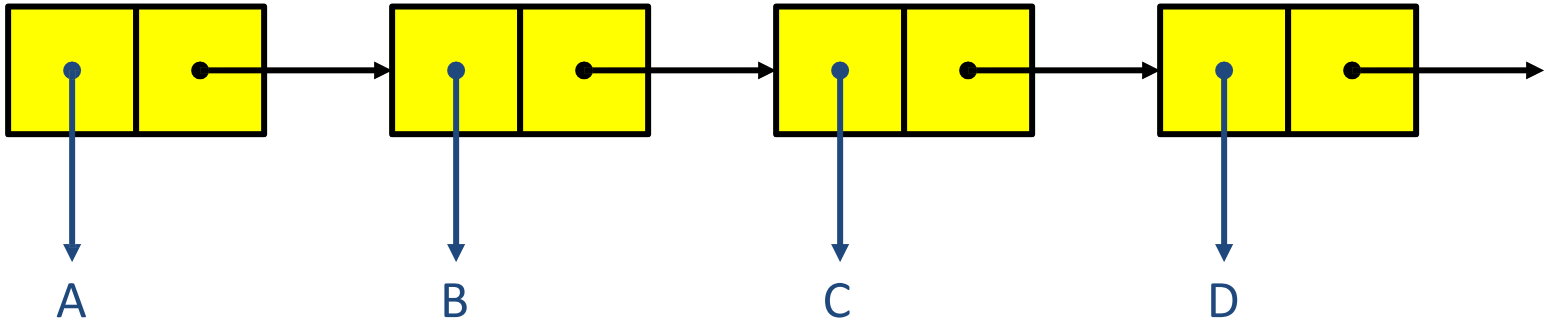
- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized time** of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$



Singly Linked List

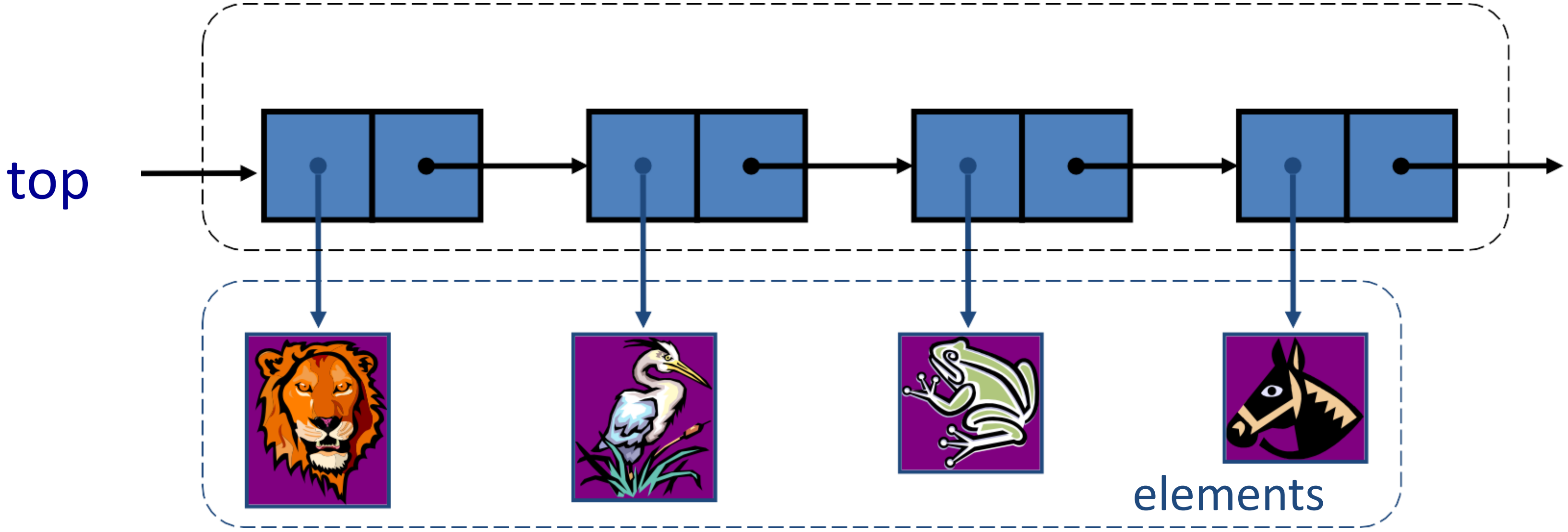
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is ***O***(***n***) and each operation of the Stack ADT takes ***O***(1) time

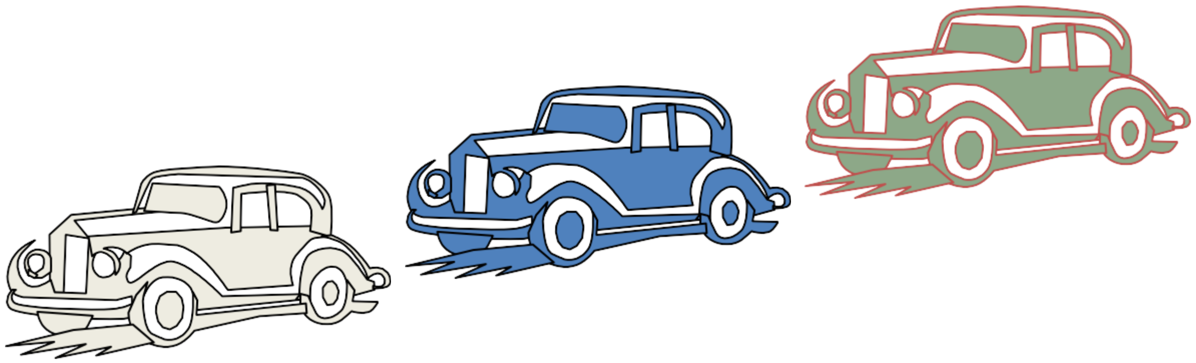


Stack Summary

- Stack Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
Pop()	O(1)	O(1)	O(1)
Push(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
Top()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

Queues



Outline and Reading

- The Queue ADT
- Implementation with a circular array
  - Growable array-based queue
- List-based queue

The Queue ADT

- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored

- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object o): inserts element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue

Exercise: Queues

- Describe the output of the following series of queue operations
  - enqueue(8)
  - enqueue(3)
  - dequeue()
  - enqueue(2)
  - enqueue(5)
  - dequeue()
  - dequeue()
  - enqueue(9)
  - enqueue(1)

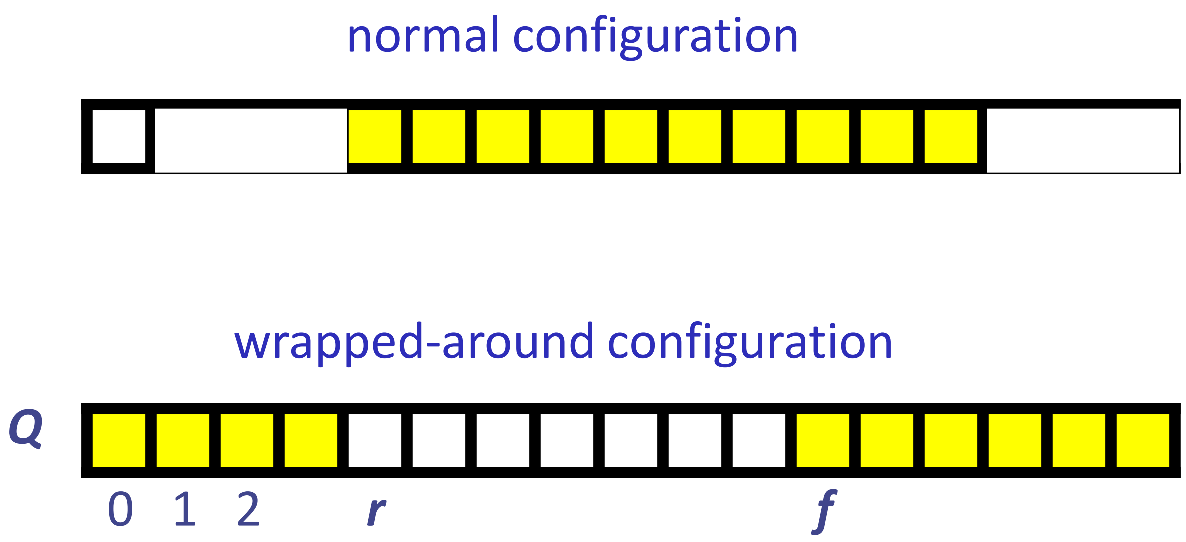
Applications of Queues

- Direct applications
  - Waiting lines
  - Access to shared resources (e.g., printer)

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

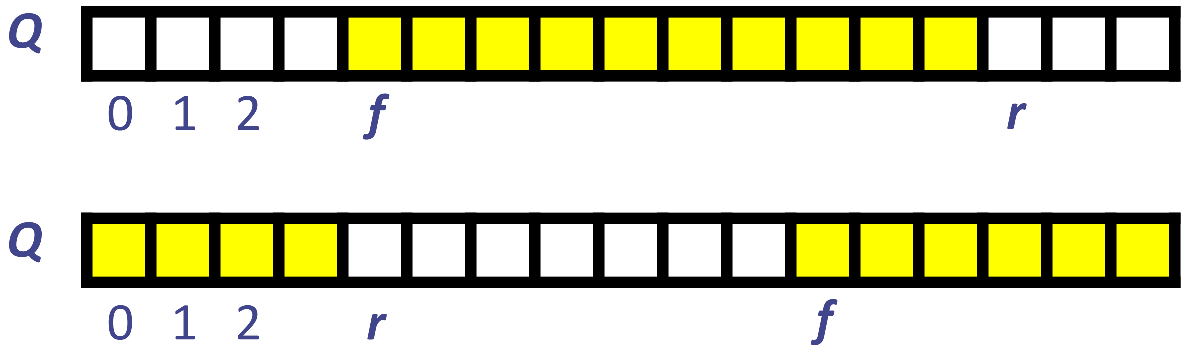
- Use an array of size ***N*** in a circular fashion
- Two variables keep track of the front and rear
  - ***f*** index of the front element
  - ***r*** index immediately past the rear element
- Array location ***r*** is kept empty



Queue Operations

- We use the modulo operator (remainder of division)

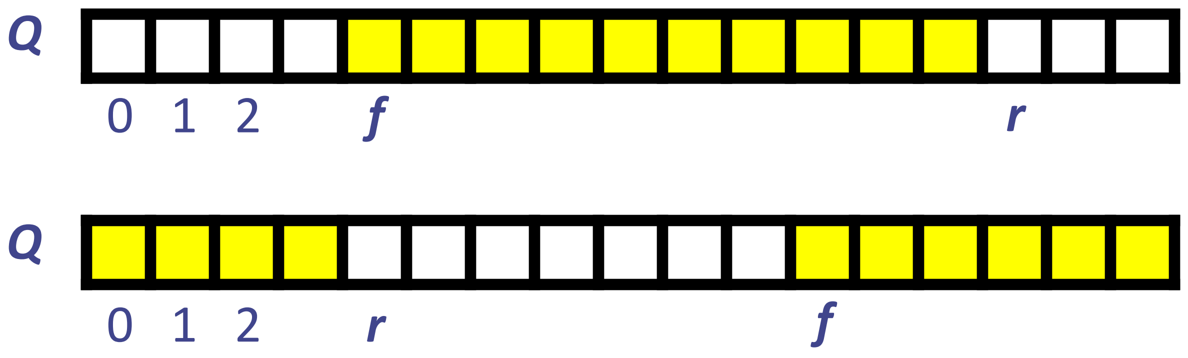
```
Algorithm size()
  return (N + r − f) mod N

Algorithm isEmpty()
  return (f = r)
```



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

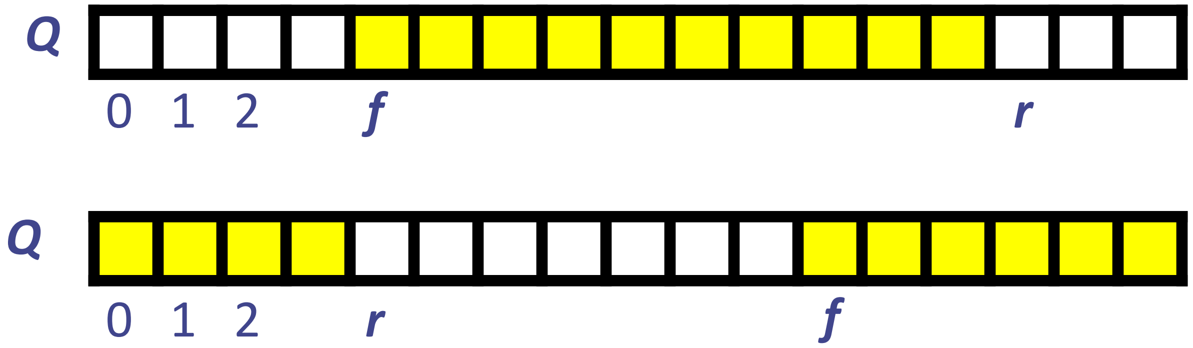
```
Algorithm enqueue(o)
  if size() = N - 1 then
    throw FullQueueException
   else
    Q[r] = o
    r = (r + 1) mod N
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
   else
    o = Q[f]
    f = (f + 1) mod N
    return o
```



Performance and Limitations

(array-based implementation of queue ADT)

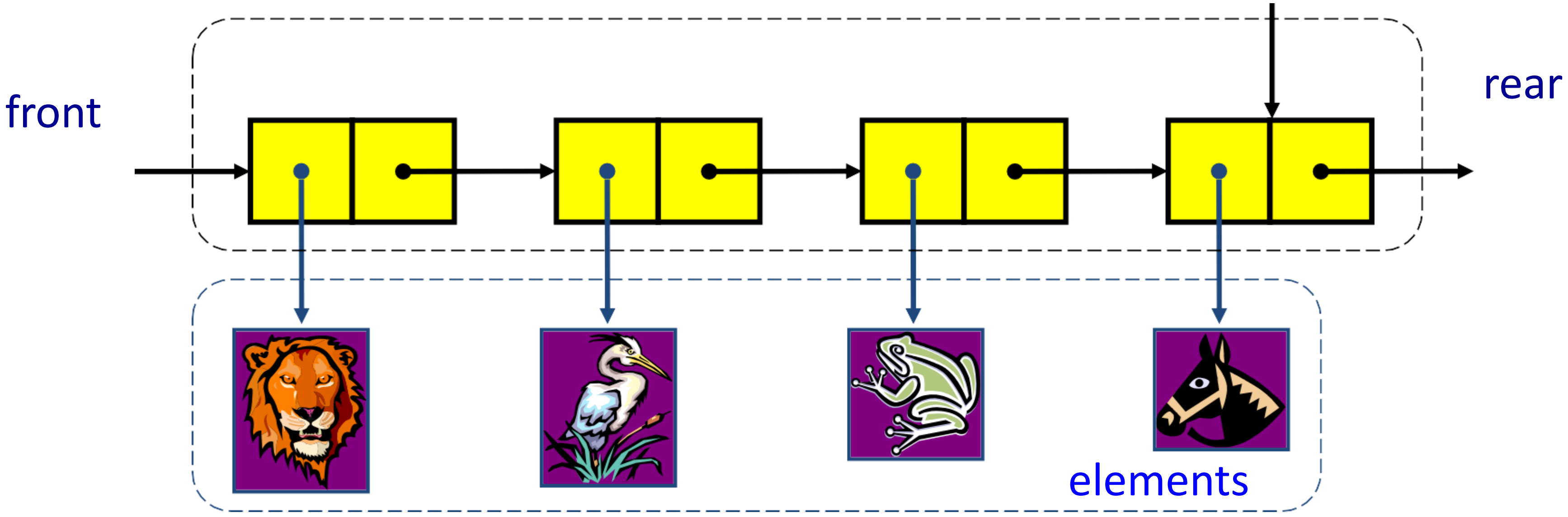
- Performance
  - Let ***n*** be the number of elements in the queue
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)
- Limitations
  - The maximum size of the queue must be defined *a priori*, and cannot be changed
  - Trying to enqueue an element into a full queue causes an implementation-specific exception

Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the head of the list
  - The rear element is stored at the tail of the list
- The space used is $\boldsymbol{O}(\boldsymbol{n})$ and each operation of the Queue ADT takes $\boldsymbol{O}(1)$ time
- NOTE: we do not have the limitation of the array based implementation on the size of the stack because the size of the linked list is not fixed, i.e., the queue is NEVER full



Queue Summary

- Queue Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
dequeue()	O(1)	O(1)	O(1)
enqueue(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
front()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

The Double-Ended Queue ADT

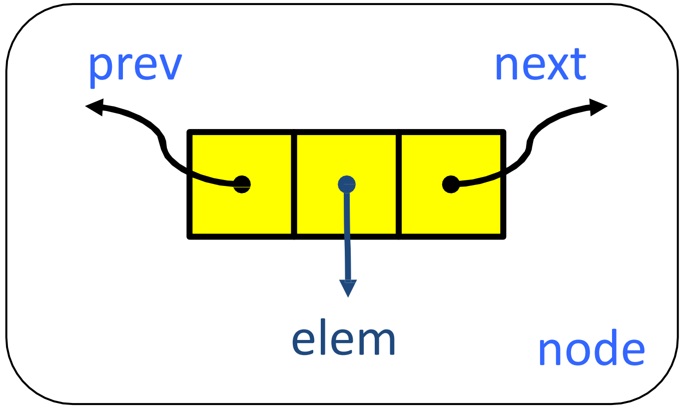
香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

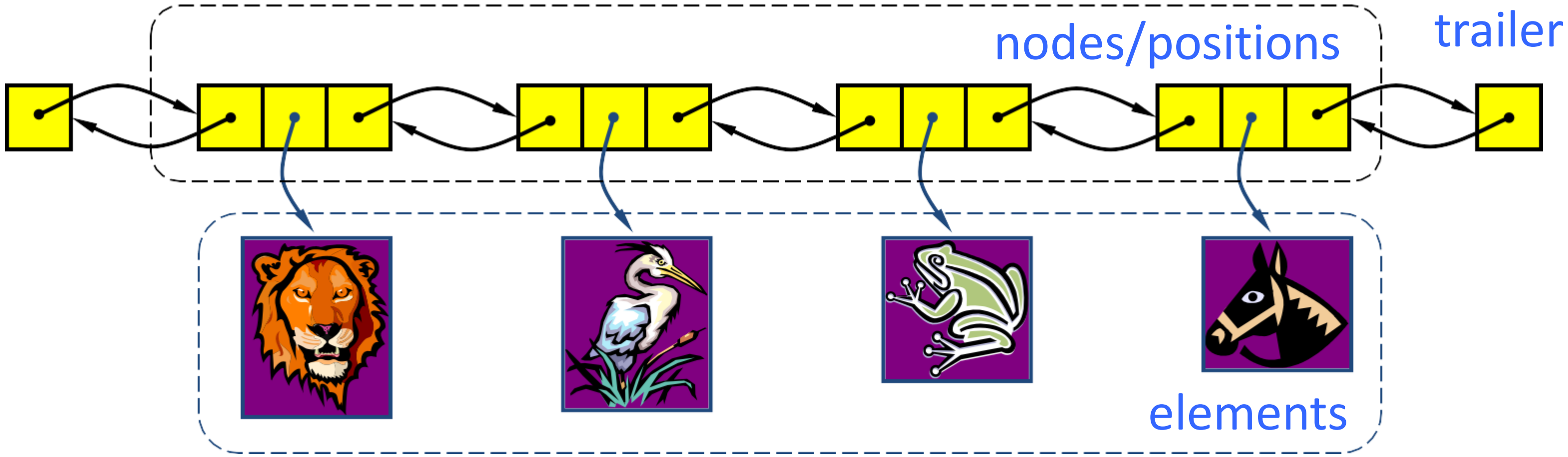
- The Double-Ended Queue, or Deque,
  - ADT stores arbitrary objects.

  (Pronounced ‘deck’)
- Richer than stack or queue ADTs. Supports insertions and deletions at both the front and the end.
- Main deque operations:
  - insertFirst(object o): inserts element o at the beginning of the deque
  - insertLast(object o): inserts element o at the end of the deque
  - RemoveFirst(): removes and returns the element at the front of the queue
  - RemoveLast(): removes and returns the element at the end of the queue
- Auxiliary queue operations:
  - first(): returns the element at the front without removing it
  - last(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyDequeException

Doubly Linked List

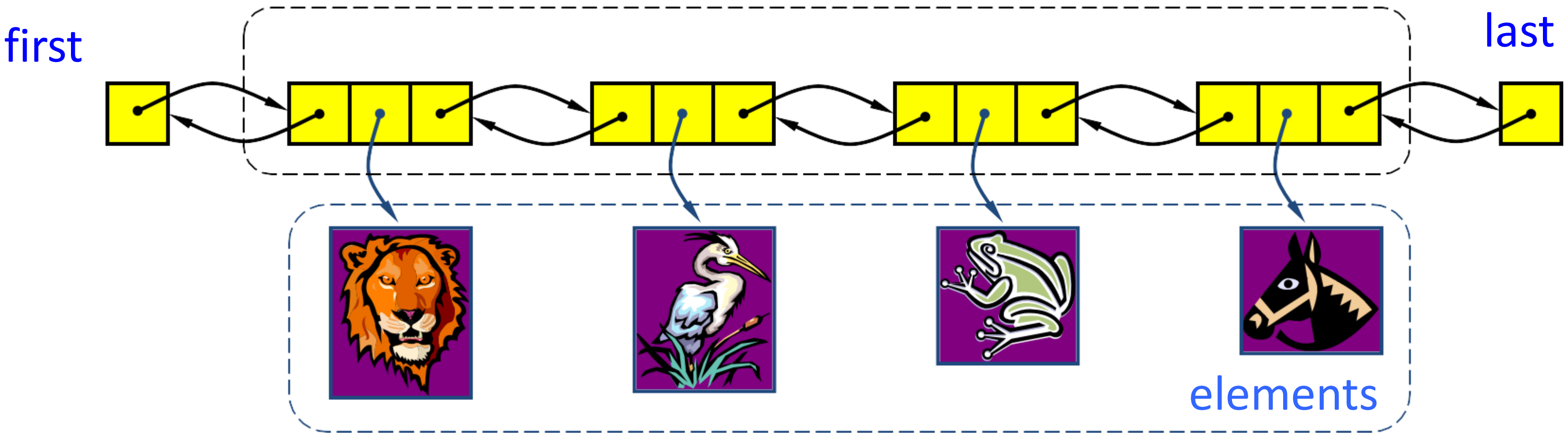
- A doubly linked list provides a natural implementation of the Deque ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





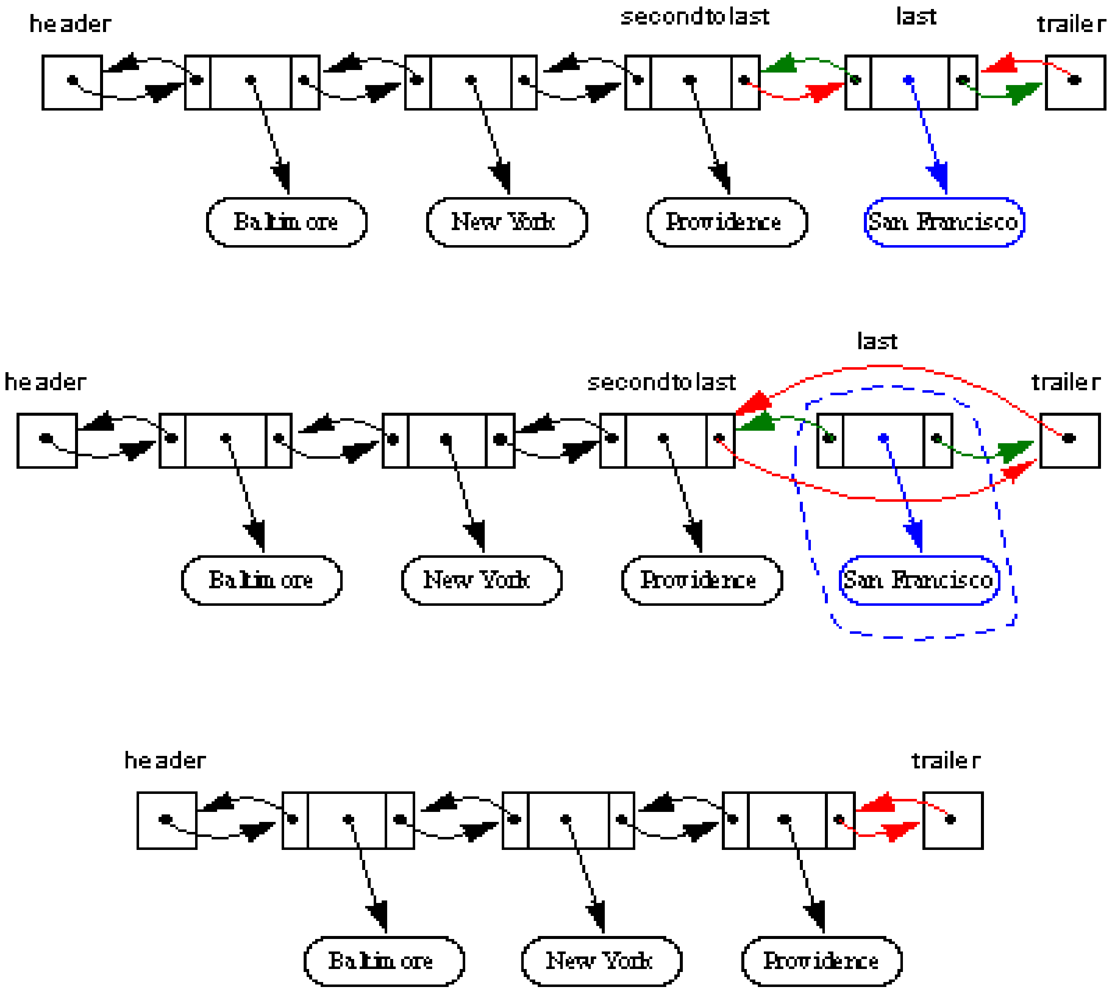
Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $\boldsymbol{O}(\boldsymbol{n})$ and each operation of the Deque ADT takes $\boldsymbol{O}(1)$ time



Implementing Deques with Doubly Linked Lists

Here's a visualization of the code for removeLast().



Performance and Limitations

(doubly linked list implementation of deque ADT)

- Performance
  - Let ***n*** be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - NOTE: we do not have the limitation of the array based implementation on the size of the stack because the size of the linked list is not fixed, i.e., the deque is NEVER full

Deque Summary

- Deque Operation Complexity for Different Implementations

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked	List Doubly-Linked
removeFirst(), removeLast()	O(1)	O(1)	O(n) for one at list tail, O(1) for other	O(1)
insertFirst(o), InsertLast(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)	O(1)
first(), last	O(1)	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)	O(1)

Implementing Stacks and Queues with Deques

Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

The End