

middle English: abstractus
draw away from

Latin: ab + trahere
from draw off

“ ”

Abstract Data Types

CSC148, INTRODUCTION TO COMPUTER SCIENCE

DIANE HORTON, JONATHAN CALVER, AND SADIA SHARMIN


Abstraction is critical

An interface presents us with an abstraction.

Client code can be written without knowing *anything* about the implementation.

- ✓ Reduces cognitive load for the programmer of client code.
 - Modern, complex software would be impossible otherwise.
- ✓ Implementation can change with no effect on client code.
 - We call this plug-out, plug-in compatibility.

ADTs take abstraction a step further

An Abstract Data Type such as  stack is

- beyond any particular implementation
- beyond even any particular programming language!

It is part of a common language used by programmers everywhere.

Abstract data types: a common language

Set

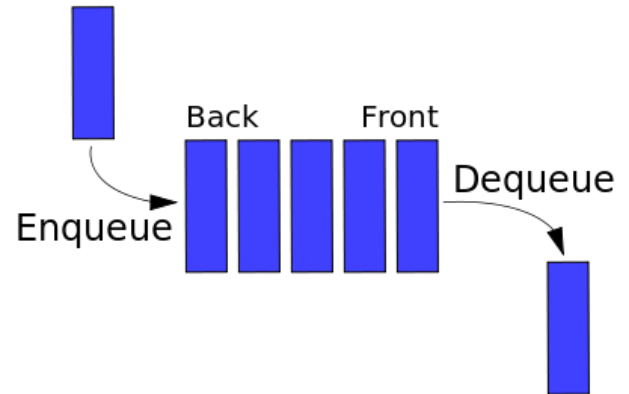
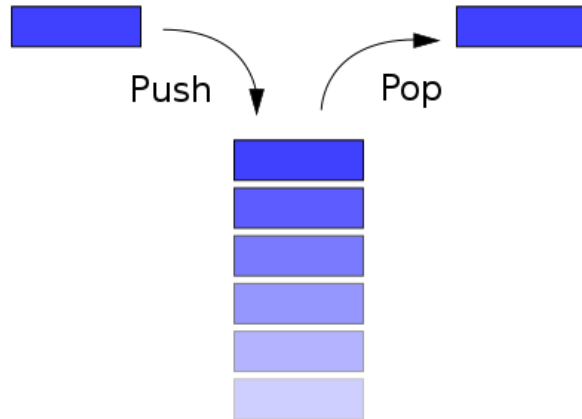
Multiset

List

✓ Map

Iterable

Stacks and Queues



Simplicity is powerful

In Python, frames for function calls form a **stack**.

In office hours, people join a **queue**.

Diversion: balancing parentheses



A string is *balanced* when...

Every (is followed by a matching). Nested is allowed.

Balanced


◦ () (1 2 (4 5)) (((4))) () () () () ()

Not balanced

◦ (1) (((bla () a () ah)

???

```
(a(s(d(sdf)safd(s(gdas)()))(qwerq  
wer))sadf)((werqew())df(asd(asdfa  
sdfasf))())sdfs(d(((sfsfsdad)sdf  
as)))(wer(wer(w)erwe(r)(()wqwqqew  
w)(()(d(fsdfadssdfs)((())dsfdfs  
a)dfsa(dass(fdasqwerwd(d(fas)dfas  
)dfas)))
```



Key ideas

Ignore all characters except (and).

Keep track of when you see a (, but forget about it when you've seen the matching).



Now design your function!



Evaluating efficiency of implementations

Given multiple implementations of the same interface, what are different ways we can compare them?



Consider another stack implementation



A timing experiment

A common technique used to gain evidence about the efficiency of some code is to run a **timing experiment** that simply runs the code and see how long it takes to run.

Such experiments often are repeated multiple times for different **sizes of data** (in our case, stack sizes).



Two fundamental questions

1. Why do Python lists behave this way?
2. How can we talk about running time more precisely, without relying on timing experiments?

Efficiency of Python lists

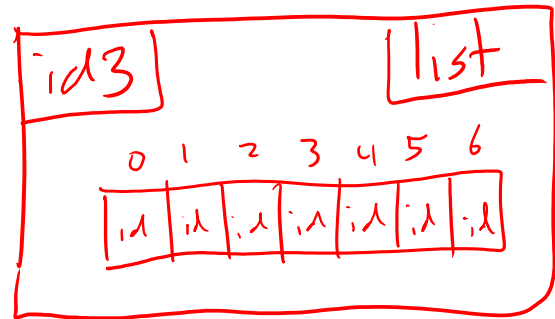
How are Python lists implemented, and what are the implications for the running time of list operations?



A Python list in memory

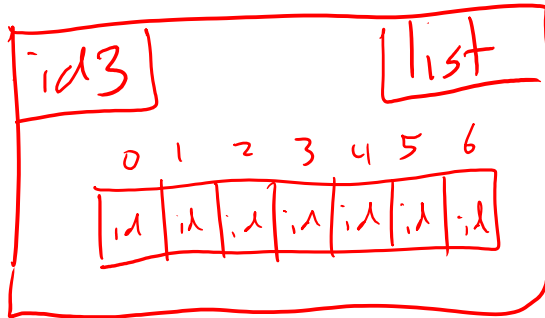
A Python list stores the ids of its elements in a **contiguous block of memory**.

This is an *array-based* implementation.



A Python list in memory

This
list:



is really stored more like this:

A Python list in memory

This makes list indexing take **constant time**:
the running time doesn't depend on the length of the list.

A Python list in memory

Insertions and deletions must preserve the contiguity of the element ids: elements must be *shifted over*.

Key thing to remember: the front of the list is fixed, while the back can “expand” to take up more or less space.



Communicating about running time

Goal: communicate how long a function/program takes to run as it is given larger and larger inputs.

In other words, we want to describe **running time as a function of input size**.



Communicating about running time

We want a way of talking about running time that doesn't depend on timing experiments or exact step counts.

What we care about is the *type of growth*:

- logarithmic, linear, quadratic, exponential, etc.

Big-Oh notation

$O(n)$, $O(n^2)$, $O(\log n)$, $O(2^n)$, ...

Homework: read section 3.4 of the Lecture Notes.

