Who is running the course?

Instructor:
    David Eppstein
eppstein@uci.edu

Teaching assistants:
    Hadi Khodabande
    khodabah@uci.edu
    Nitya Raju
    nraju1@uci.edu
Online resources

Course web site:  
https://www.ics.uci.edu/~eppstein/261/

Online course discussions: Ed Discussion (Canvas)

Turn in homework and return graded homeworks: Gradescope

Confidential questions about your performance: Email us!
Course material

Lectures

In person but also available online and recorded through Zoom/Canvas

Lecture slides will be linked on course web site

Weekly homeworks

Group work is permitted

Midterm and final exams

Short answer questions
Closed book, closed notes, no devices
See course web site for schedule
Expectations

What do I expect you to know already?

- Undergraduate-level data structures: how to implement standard and basic structures including stacks, queues, lists, heaps, and binary search trees. Some basic analysis of these structures.

- Undergraduate-level algorithms: how to describe algorithms in pseudocode, what $O$-notation means, divide-and-conquer algorithms and their analysis using recurrence equations, and some other standard algorithms such as depth-first search and Dijkstra’s algorithm.
What is a data structure?
Let’s look at a standard algorithm: **depth-first search**, to find the subset of vertices in a graph $G$ that can be reached from a given starting vertex $s$. (In Python; later we’ll use pseudocode.)

```python
def DFS(s, G):
    visited = set()  # already-processed vertices
    def recurse(v):
        visited.add(v)  # remember we’ve found it
        for w in G[v]:  # look for more in neighbors
            if w not in visited:
                recurse(w)
    recurse(s)
    return visited
```
What data structures does this algorithm use?

Even this very simple algorithm uses multiple structures:

- A set of visited vertices
- The input graph, organized as a dictionary whose keys are vertices and whose values are collections of neighbors
- The collections of neighbors for each vertex
- (Implicitly) a stack, keeping track of the sequence of subroutine calls and their local variables
So what is a data structure?

The information a program or algorithm needs to access and update as it runs

The layout of that information into words of memory on a computer

A catalog of methods by which the information is accessed and modified

Algorithms for performing those methods efficiently

Analysis of how much memory the structure uses and how much time its methods use
Levels of abstraction

**API**

What operations does the structure provide?

**Implementation**

How do we organize the data, and how do we perform its operations?

**Analysis**

How much space and time per operation does it use?
What operations does the structure provide?

Example: Stack

- `push(x)`: add item `x` to the stack
- `pop()`: remove and return most-recently-added item
- `top()`: return the item that `pop` would return, but without removing it
- `isempty()`: test whether stack has any remaining items
- `new()`: create a new empty stack object
Implementation of stacks as linked lists

How do we organize the data, and how do we perform its operations?

Stacks using singly-linked lists:

- Stack is a collection of nodes with value and pointer to next node; stack itself points to top node
- Empty stack = null pointer
- To push, make a new node and point to it
- To pop, find top value and change stack pointer to next item

For example after push(7), push(2), and push(3):

```
Top → 3 → 2 → 7 → None
```
class stack:
    def __init__(self):
        self._top = None

    def push(self, x):
        self._top = (x, self._top)

    def pop(self):
        popped = self._top[0]
        self._top = self._top[1]
        return popped

    def top(self):
        return self._top[0]

    def isempty(self):
        return self._top is None
Implementation of stacks as arrays

How do we organize the data, and how do we perform its operations?

Stacks using arrays:

- Stack is an array of cells holding values and a length counter
- Empty stack = array with counter value zero
- To push, increase counter and add value to array
- To pop, find top value and decrease counter

For example after push(7), push(2), and push(3):

Counter: [3]  Array: [7 2 3]
Analysis of stack implementations

How much space and time per operation does it use?

Stack example:

- Both implementations use $O(1)$ time per operation
- List implementation uses $O(1)$ space per element
- Array implementation uses space $= \max \#$ of elements
- Array is probably faster in practice (no overhead for node allocation/release; more predictable memory access pattern) but showing this involves experimentation, not theory