CS 261: Data Structures

Week 1: Introduction

Lecture 1a: Course overview

David Eppstein
University of California, Irvine

Spring Quarter, 2024

Who is running the course?

Instructor:

David Eppstein

Teaching assistant:

Shion Fukuzawa

Online resources

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

Online course discussions: Ed Discussion (Canvas)

Return graded exams: Gradescope (Canvas)

Confidential questions about your performance: Email us!

Course material

Lectures

In person only

Lecture slides will be linked on course web site

Weekly problem sets

Not turned in, not graded
Group work is encouraged

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?

Example

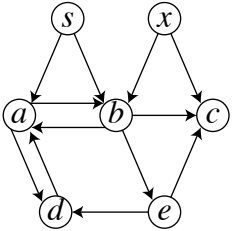
Let's look at a standard algorithm: depth-first search, to find the subset of vertices in a graph G that can be reached from a given starting vertex s. (In Python; later we'll use pseudocode.)

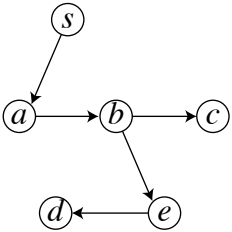
```
def DFS(s,G):
    visited = set()      # already-processed vertices

    def recurse(v):      # call for each vertex we find
        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
```

Depth-first reachability example





recurse(s)
  visited: $\{s\}$
  recurse(a)
    visited: $\{s, a\}$
    recurse(b)
      visited: $\{s, a, b\}$
      recurse(c)
        visited: $\{s, a, b, c\}$
      recurse(e)
        visited: $\{s, a, b, c, e\}$
        recurse(d)
          visited: $\{s, a, b, c, d, e\}$

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?

Stack API

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

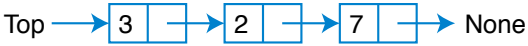
Implementation of stacks as linked lists

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):



Coding of stacks as linked lists

```
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
```

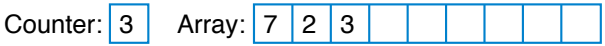
Implementation of stacks as arrays

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):



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

Styles of analysis

Worst-case analysis

- Consider each operation X (and its implementation) separately from each other
- Find a sequence P of previous operations that causes X to be as slow as possible
- Equivalently, find a possible state S of the data structure that causes X to be as slow as possible
- Worst case time for operation X
  $=$ time for X after sequence P
  $=$ time for X on state S

Averaging

Intuitions:

- The whole point of a data structure is to do a sequence of operations, not just a single thing
- The time for an operation may vary, rather than always being its worst case
- We usually care only about the total time of the overall algorithm, not how much time each individual operation took
- So we should average the time per operation over a whole sequence of operations, while still looking for the worst-case sequence, the one that makes the average as large as possible

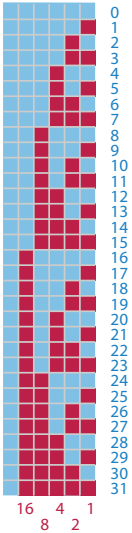
Example of averaging: Binary counter

When you add one to a binary number:

- The least significant bit that has value 0 changes to 1
- All lower-order bits change from 1 to 0

Example:

- $167 = 10100111_2$
- $168 = 10101000_2$



Example of averaging: Binary counter

Represent a binary number as an array C where $C[i]$ is the ith bit in the binary representation of the number (starting from $i = 0$ for the ones bit), so the number itself is $\sum_i 2^i C[i]$.

Initially all cells are zero.

Increment by flipping bits until finding the lowest-order 0 and flipping it to 1:

```
def increment():
    i = 0
    while C[i] == 1:
        C[i] = 0
        i += 1
    C[i] = 1
```

Averaged analysis of binary counter

Total time over a sequence of n increments $= O(\#$ bits flipped$)$

Cell $C[i]$ gets flipped $\lfloor n/2^i \rfloor$ times.

So total # flips $\leq \sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq 2n - 1$ (geometric series).

Average time per increment is $O\left(\frac{2n-1}{n}\right) = O(1)$.

In comparison, worst case is $\Theta(\log n)$
(e.g. when incrementing to the largest power of two $\leq n$)

Naive averaging is problematic!

It worked for this simple example, but...

- What sequences of operations are the slow ones?

  (It's easier to analyze single operations than to find worst-case sequences.)

- If you have more than one type of operation in a sequence, how much of the time is caused by each type of operation?

Amortization (next time) is a method for averaging that lets us handle both of these problems.