Revision

Why Are We Doing This?

The skill of being able to build large-scale systems that can handle large amounts of data efficiently is an essential skill for software developers.

- Software design patterns are the established ways of implementing solutions for common tasks, for efficiency, easy maintenance, and easy extension.

The main emphasis is on efficiency, as scaling up to real applications requires efficient code.

This Module

This module covers the foundational knowledge that marks us as computer scientists or professional software engineers. You can learn to program, but without this knowledge, there's something lacking when it comes to scaling things up.

We want to understand:

- How to implement each pattern.
- What the space requirements are.
- What the time complexity is.
- Which uses are appropriate for each different data structure or algorithm.

These are transferrable techniques that can be used for almost any programming language or hardware.

In many cases, what we see is not the best way to do it in Python, but is a fundamental technique which will work in multiple different languages.

Complexity

We measure complexity in terms of the number of steps, where a step is a basic operation expected to take constant time regardless of the input parameters.

This is useful because it isn't limited to a specific language or computer architecture.

We use Big-O notation.

Big-O Notation

This is a formal notation for expressing worst-case complexity in terms of known mathematical functions.

Sometimes we use it informally averaged over many operations.

Sometimes we use it for the *expected* average.

[missed a bit here]

Definition

- f(x) is O(g(x)) is and only if, for some constant k, there is another constant C so that for all values of x bigger than k, f(x) < Cg(x).

Useful Examples

```
1 + 2 + 3 + … + n = (0.5)(n)(n + 1)
```

This is O(n squared).

```
2**0 + 2**1 + … + 2**n = 2**(n + 1) - 1
```

A complete binary tree with n nodes has depth of:

```
floor(log n)
```

Array-Based Sequences

An array is a contiguous block of memory holding items of the same size.

Any item can be accessed using its index i by jumping to the memory address which is the starting address + i*(size of the items).

This is constant time access.

We can increase the size of an array by reserving new space for the larger array, and copying items across. * Doubling the array when needed means O(n) to build an array of n items, so O(1) to add each item *on average*.

Python Lists

Python lists are array-based. All data items in Python are objects, and Python stores references to those objects in the arrays – since a reference is just a memory address, these are of constant size, no matter what you're referring to.

To add a new item to a full Python list, Python increases the underlying size and internally marks the extra cells as available.

You need to know the complexity of adding items in different positions, searching for items, and removing items.

E.g. Python arrays don't like gaps, so if you remove something from the middle of a list, every item after it needs to be moved up to fill the gap, which is O(n).

Linked Lists

We store a collection of nodes in memory, wherever we want. Each node points to the value it contains, the next node in the chain, and the previous node

We use dummy head and tail nodes (with a value of `None`) to make operations easier.

[missed a bit]

Linked Trees

These are binary trees, so now each node points to its value, its left child, its right child, and its parent.

We've only looked at binary trees, but we will look at more general trees next year.

Tree Traversals and Array Representation

- Preorder:
    1. Parent
    2. Left child
    3. Right child
- Postorder:
    1. Left child
    2. Right child
    3. Parent
- Inorder:
    1. Left child
    2. Parent
    3. Right child
- Breadth-first:
    1. Root node
    2. Children of the root node
    3. Grandchildren of the root node
    4. etc

Binary Search Trees

A tree representation of a sorted list, which allows binary searches on linked structures.

For all nodes, all left descendants must have lower values, and all right descendants must have higher values.

To add a new value:

```
find where it should be
if it isn't there
    add a new node with value
```

The complexity of adding a new value is O(height of tree).

Removing a Value

```
if it is a leaf
    wipe the node
else if it is a semileaf
    join parent to child, then wipe
else if it is internal
    1. Find the biggest element less than our current node
    2. Move that value into our current node
    3. Then remove the node we have copied, by this
procedure
```

Note: This is how to change the sketch; the implementation details might be different.

AVL Trees

AVL trees are binary search trees with no node unbalanced. We define a node to be unbalanced if:

- the heights of its children differ by 1 or more
- the node is a semileaf, with a child of height of one or more

[insert rotation details here]

Searching for anything will be O(log n).

Abstract Data Types

We defined abstract data types for standard containers. Each ADT specifies operations you can do to the container. Each ADT can be implemented [missed the rest]

- Stack
- Queue
- Positional List
- [missed the rest]

Stack

- push: place an element onto the top of the stack
- pop: get the first element off the top of the stack (and remove it from the stack)
- top: report the top element form the stack (but leave it on the stack)
- length: report how many elements are in the stack
- is empty: report whether or not the stack is empty

[check]

Queue

[insert ADT]

Implement with a doubly-linked list: * `enqueue()`, `dequeue()`, and `front()`: O(1)

Array-Based Queue

Simple implementation is O(n) for dequeue. Avoiding list.pop() and using a head reference wastes space.

Instead, we maintain head and tail references, and wrap elements around the array. We use modular arithmetic for the cleanest coding.

Complexity: O(1)* for both enqueue and dequeue

Priority Queue

Unsorted lists: O(1) or O(1) *to add, O(n) to remove Sorted lists: O(1) or O(1)* for remove, O(n) to add AVL trees: O(log n) to add, remove and find

Binary Heap

Complete tree where each node has lower value than its children.

- Add a value in last place, then bubble up to restore property.
- Remove the top item, copy last into place, bubble down, always choosing the lowest value child to swap with.

Complexity: O(log n) to add and remove, but O(1) to find

Array Representation of Binary Heap

Each value from the binary heap tree appears in the cell corresponding to the breadth-first traversal.

I.e. the root node corresponds to index 0, its leftchild to 1, the root's rightchild to 2, and so on.

For node in cell i: * parent is `(i - 1) // 2` * leftchild is `2i + 1` * right child is `2i + 2`

Map (or Dictionary)

Hash Tables

An efficient representation of a map:

- hash function converts key to an integer
- compression converts integer to an array index

Bucket array stores a list of items in each cell.

Open addressing stores items in first free cell after index, based on a probing scheme. * We have to be careful when deleting items to maintain correctness – have to mark to say that something was in there, and to continue if searching.

If the table is resized appropriately, then has tables offer O(1) get, contains, set, and del in the expected case (though worst case is O(n)).

€€General Advice

- You need to know the ADTs
- Understand the array and linked storage methods
    - Learn how to be efficient with space.
    - Know the names of the different structures.
- Learn the tree-based methods for storing sorted lists and priority queues
    - Learn the add, remove, rotate, bubble procedures.
- Understand how hash tables work.
- Either learn complexities, or know how to work them out.