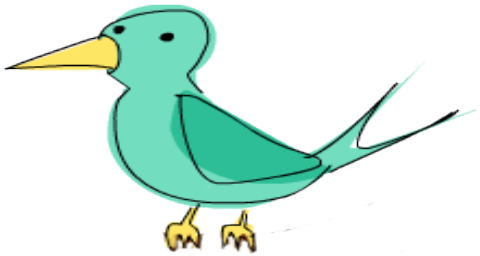
Esoteric Data Structures

Moses Mugisha
CTO, Sparkplug
@mossplix
October 2014

Talk Structure



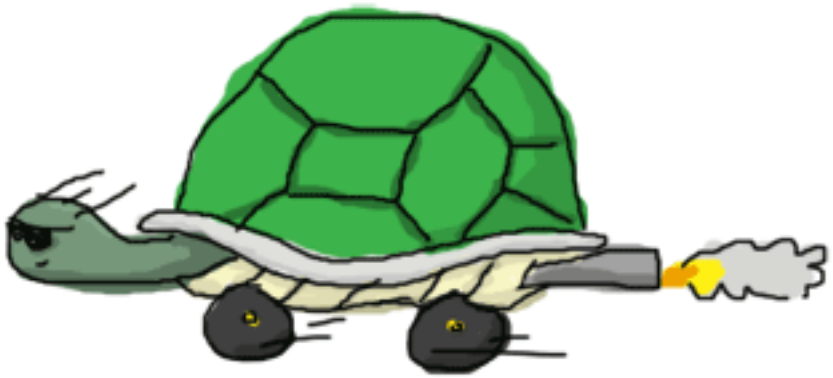
We shall start with simple structures and build on to complex structures

Simple	Complex
Arrays	Hash Table
Bit Vectors	Bloom filter
Heaps and Stacks	Trie
Linked lists	Sorted Map
	Skip Lists

`Talk Materials https://github.com/mossplix/geeknight`

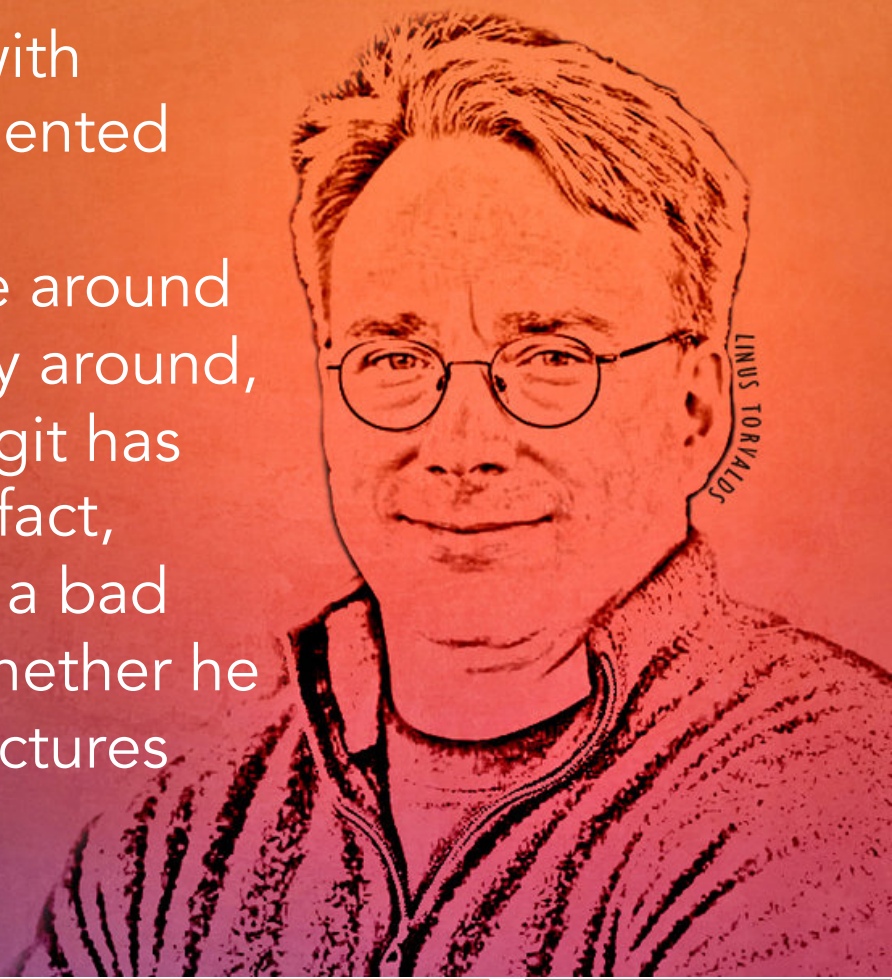
Why Data Structures Matter

The Best Code Can't Make Up For Poor Data Structures



Why Data structures

Git actually has a simple design, with stable and reasonably well-documented data structures. In fact, I'm a huge proponent of designing your code around the data, rather than the other way around, and I think it's one of the reasons git has been fairly successful […] I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.



Why Data structures

- The efficiency of an algorithm depends on the data structures used in an algorithm
- A wise choice of the structure can greatly reduce execution time and memory usage
- With the right data structures , the algorithms will almost always be self-evident and it will be very easy to reason about programs
- Programming == Data + Operations on Data (Algorithms)

Arrays

Low Level Arrays

- In computers, bits of information are grouped into bytes
- To keep track of the bytes , computers use memory addresses
- In general programming languages keep track of the association between an identifier and the memory address in which the associated value is stored
- Related variables can be stored one after another in a contiguous portion of the computer's memory …. This representation is called an Array

Python Arrays

- Python Lists/Sequence types are dynamic arrays
- Lets do simple amortized analysis of list storage

```
import sys
data=[]
for i in range(20):
    print "%d : %d"%(i,sys.getsizeof(data))
    t1=time()
    data.append(None)
    t2=time()
    print (t2-t1)
```

- Python uses 16 bits to store Unicode characters

Python Lists

- Lists are always big

```
>>> sys.getsizeof('a’)
38
>>> sys.getsizeof([1])
80
>>> sys.getsizeof(u'a')
52
```

- Python lists are 64-bits arrays for memory addresses for the items
- This means they take up 5x more storage than normal arrays

Compact Arrays

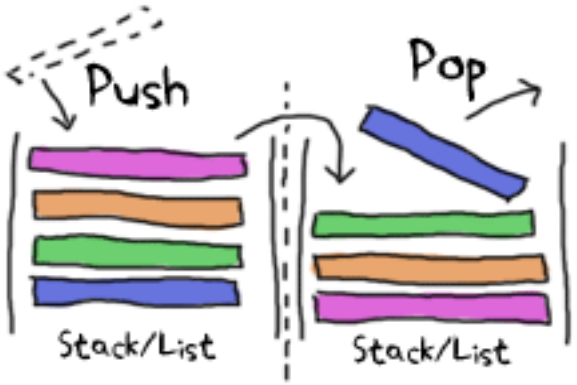
- Use ctypes module

```
>>> import  ctypes
>>> array=ctype.c_int*5
>>> evens=array(2,4,6,8,10)
>>> sys.getsizeof(evens)
>>> sys.getsizeof([2,4,6,8,10])
```

Stacks

Stack

- Last In First Out
- Python lists can act as stacks
- List ADT methods: push, pop, is_empty, top
- Let's do a simple application



Simple Stack

Simple Python implementation

```
class Stack:
    def __init__(self):
        self.data = []
    def len(self):
        return len(self.data)

    def is_empty(self):
        return len(self.data) == 0

    def push(self, e):
        self.data.append(e)

    def top(self):
        if self.is_empty():
            raise Empty("Stack is empty")
        return self.data[-1]

    def pop(self):
        if self.is_empty():
            raise Empty("Stack is empty")
        return self.data.pop()
```

Parenthesis Matching

```
def is_balanced(expr):
    left = "({["
    right = ")}]"
    stack = Stack()
    for e in expr:
        if e in left:
            stack.push(e)
        elif e in right:
            if stack.is_empty():
                return False
            if right.index(e) != left.index(stack.pop()):
                return False

    return stack.is_empty()
```

Queues & Dequeues

Queues

- FIFO
- ADT methods enqueue,dequeue,is_empty,first

- Enqueue => add to the end of queue
- Dequeue => pop item from front of queue

Dequeues

- Doubly ended queues

Python collections module contains their implementations

Dequeue Example

```
>> From collections import deque
>> def tail(filename, n=10):
        'Return the last n lines of a file'
        return deque(open(filename), n)
>> tail(“american-english”)
```

Bit vector/Bit array

- Bit Vector is meant to condense bit values (or booleans) into an array so that no space is wasted.
- Why not just create a Boolean array?

```
>>> bit_vector=(ctypes.c_bool*20)()
```

- Most compilers use a larger data type, such as an integer, in place of a Boolean

Bit Fields

- More like bit vectors but with more than 2 states
- Comes handy when sending data between applications over the network

Asymptotic Analysis

Before we jump to complex structures, we need to analyze the efficiency of data structures

Two factors …

- Time complexity (time taken to insert , delete or find an element)
- Space complexity (storage needs of a data structure)

Time Complexity

For example given linear search of a list

- Worst-case: An upper bound on the running time for any input of given size (n comparisons)
- Average-case: Average time of all inputs of a given size (N/2 comparisons)
- Best-case: The lower bound on the running time(1 comparison)

In asymptotic analysis we are interested in the worst case scenario

BIG-O

- The Big-O of an algorithm is a function that roughly estimates how the algorithm scales when it is used on different sized datasets
- Typical Complexities

Complexity	**Notation**	**Description**
Constant	O(1)	Constant number of operations not depending on input size e.g n=1000000 -> 1-2 operations
Logarithmic	O(log n)	Number of operations proportional to log n e.g n=1000000000 -> 30 operations
Linear	O(n)	Number of operations proportional to input data e.g n=10000 -> 5000 operations

BIG-O continued

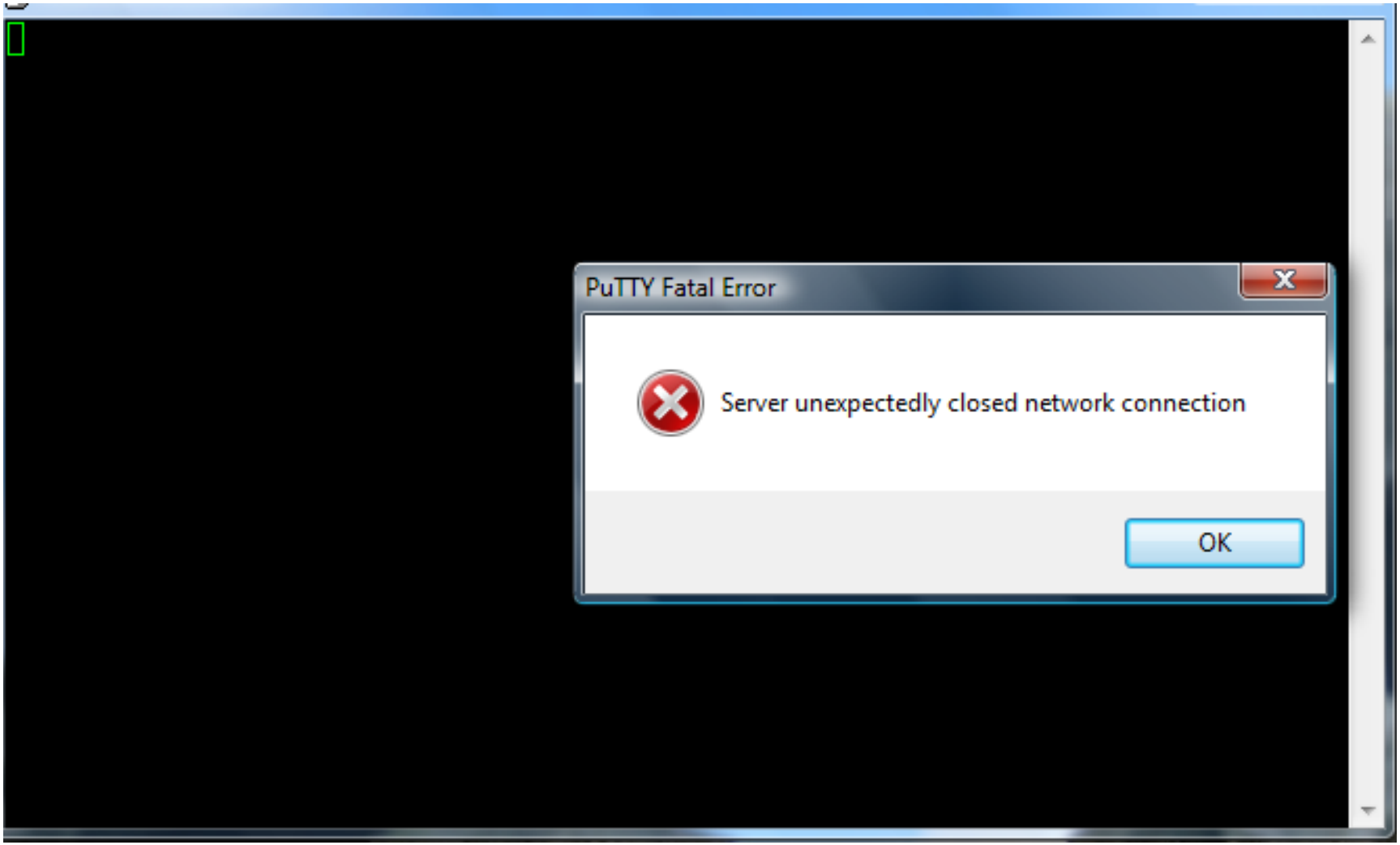
.

Complexity	Notation	Description
Quadratic	O(n^2)	number of operations proportional to the size of input data e.g n=500 -> 250000 operations
cubic	O(n^3)	**Number of operations proportional to the cube of the size of the input data, e.g. n = 200 → 8 000 000 operations**
exponential	**O(2^n), O(k^n), O(n!)**	**Exponential number of operations, fast growing, e.g. n = 20 → 1 048 576 operations**

Time Complexity and Speed

Complexity	10	20	50	100	1 000	10 000	100 000
O(1)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n)	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
O(n*log(n))	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s
$O(n^2)$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	2 s	3-4 min
$O(n^3)$	< 1 s	< 1 s	< 1 s	< 1 s	20 s	5 hours	231 days
$O(2^n)$	< 1 s	< 1 s	260 days	hangs	hangs	hangs	hangs
O(n!)	< 1 s	hangs	hangs	hangs	hangs	hangs	hangs
$O(n^n)$	3-4 min	hangs	hangs	hangs	hangs	hangs	hangs

Time Complexity and Speed



Data structure efficiency

Data Structure	Add	Find	Delete	Get-by-index
Array	O(n)	O(n)	O(n)	O(1)
Linked list	O(1)	O(n)	O(n)	O(n)
Dynamic array	O(1)	O(n)	O(n)	O(1)
Stack	O(1)	-	O(1)	-
Queue	O(1)	-	O(1)	-

Memory Hierarchies

Memory Level	Size	Response Time
CPU Registers	≈ 100B	≈ 0.5ns
L1 Cache	≈ 64KB	≈ 1ns
L2 Cache	≈ 1MB	≈ 10ns
Main Memory	≈ 2GB	≈ 150ns
Hard Disk	≈ 1TB	≈ 10ms

Caching

Hash Table

Given the following data

Name	Id
Joan	23456
Carol	34
Mary	464674748

- What if we want to store above data in a data structure, so that we can easily access the names by their id
- We could use an array , ids as indices and names as their value
- We would end up using an array of size 464674748 storing only 3 elements
- Alternatively we could use linked lists , though with many elements, it would be terribly inefficient (O(n) lookup)

Hash Table Continued

We need to

- Quickly store sparse key-based data in a reasonable amount of space
- Quickly determine if a certain key is within the table
- The important word here is *quickly !!*.
- Most easiest way of placing a key into a hash table is to modulo the id by the size of the table.
- We will use an array as our underlying data structure
- The array size will be the prime number after the number of items. e.g. for 3 items we use 5
- e.g. to place Mary in the array, 464674748%5 = 3
- We would access Mary using Mary = table[id % 3];
- Modulo key is a hash function

Collisions

- Consider inserting these two girls with these ids into the ten-cell table: 143,674 and 645,395.
- You can't. Because both numbers modulo down to 3, they should both be placed into the same cell, but a cell can only hold one item! This is called a *collision*. The only ways to resolve a collision are to use a better *hashing* function or modify the table in a way that makes collisions okay.
- Hash table efficiency

Add	**Find**	**Delete**	**Get-by-index**
O(1)	O(1)	O(1)	-

Bloom Filter

A Bloom filter is a probabilistic data structure for checking if an item is most likely in a dataset or not by checking a few bits

- Bloom filters may give false positives but never false negatives
- A bloom filter is essentially a huge bit array
- We shall need a hash function … in this case lets use a murmur hash

```
>> from bitarray import bitarray
>> import mmh3
>> bit_array=bitarray(10)
>> bit_array.setall(0)
>> b1 = mmh3.hash(”mary", 41) % 10
>> b2 =  mmh3.hash(“joan", 41) % 10
>> bit_array[b1]=1
>> bit_array[b2]=1
```

Bloom filter continued

```
from bitarray import bitarray
import mmh3

class BloomFilter:

    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)

    def add(self, string):
        for seed in xrange(self.hash_count):
            result = mmh3.hash(string, seed) % self.size
            self.bit_array[result] = 1

    def lookup(self, string):
        for seed in xrange(self.hash_count):
            result = mmh3.hash(string, seed) % self.size
            if self.bit_array[result] == 0:
                return False
        return True
```

Bloom Filter

Let us test our bloom filter

```
>> bf = BloomFilter(500000, 7)

>> lines = open(”american-english").read().splitlines()
>>for line in lines:
    bf.add(line)

>> bf.lookup(”cat")
>>> True
>> bf.lookup(”yahoo")
>>> False
>>> sys.getsizeof(bf)
```

Trie

- A trie, or *prefix tree*, is a data structure that stores strings by decomposing them into characters. The characters form a path through a tree until such time as a child node of the tree represents a string uniquely or all characters of the string have been used.

Trie continued

```
class Trie(object):
    def __init__(self):
        self.children = {}
        self.flag = False # Flag to represent that a word ends at this node

    def add(self, char):
        self.children[char] = Trie()

    def insert(self, word):
        node = self
        for char in word:
            if char not in node.children:
                node.add(char)
            node = node.children[char]
        node.flag = True

    def contains(self, word):
        node = self
        for char in word:
```

Trie continued

```
        if char not in node.children:
                return False
            node = node.children[char]
        return node.flag

    def all_suffixes(self, prefix):
        results = set()
        if self.flag:
            results.add(prefix)
        if not self.children: return results
        return reduce(lambda a, b: a | b, [node.all_suffixes(prefix + char) for
(char, node) in self.children.items()]) | results

    def autocomplete(self, prefix):
        node = self
        for char in prefix:
            if char not in node.children:
                return set()
            node = node.children[char]
        return list(node.all_suffixes(prefix))
```

Trie .. Simple application

- Auto complete dictionary terms

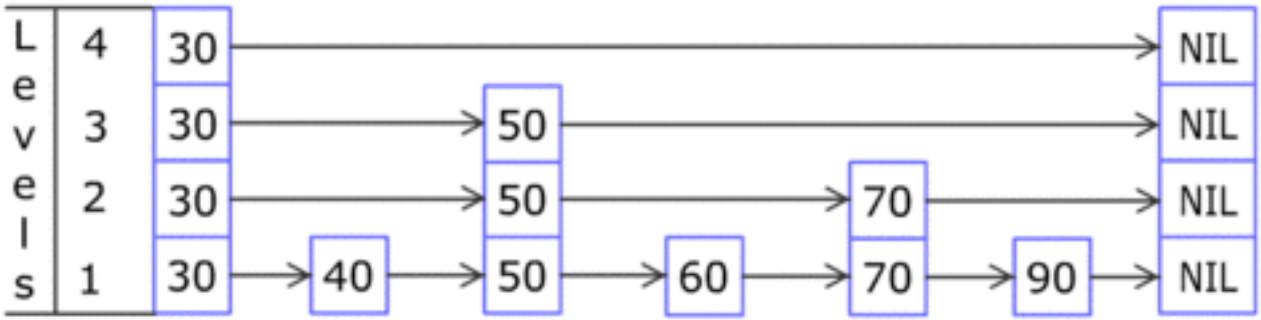
```
>>trie=Trie()
>> lines = open(”american-english").read().splitlines()
>> for line in lines:
        trie.insert(line)
>> trie.autocomplete(mo)
```

Sorted Map

- ADT allows a user to look up the value associated with a given key, but all the keys are sorted
- Say you are storing events that have occurred with their timestamps
- And you want to store the time stamps as keys and the event as a value
- However, the map ADT does not provide any way to get a list of all events ordered by the time at which they occur, or to search for which event occurred closest to a particular time
- Some applications => Flight Databases ,Maxima sets

Skip Lists

- The data structure for realizing a sorted map is a skiplist
- Skip lists make random choices in arranging the entries in such a way that search and update times are O(1ogn) on average, where n is the number of entries in the dictionary
- The main advantage of using randomization in data structure and algorithm design is that the structures and methods that result are usually simple and efficient



Skip Lists - Continued

- New element should certainly be added to bottommost level

(Bottommost list contains all elements)

- Which other lists should it be added to?
- With what probability should it go to the next level?

Sorted Map – Simple application

- Many implementations out there. Python doesn't't include it in its collections
- Lets use https://github.com/malthe/skipdict

Keep track of high scores

```
>> from skipdict import SkipDict
 >> highscores= SkipDict(maxlevel=4)
>> highscores[’Moses'] = 100
>> highscores[’Raymond'] = 95
>> highscores[‘Voldmort’]= 110
>> highscores[‘orochimaru’]=105
>> highscores.keys(max=100)[0]
Ray
>> highscores.keys(min=100)
Moses
```


Q&A