Week 1: Data structures and data formats

Algorithmic Data Science

2022-23

UNIVERSITY OF SUSSEX

Dr Adam Barrett

- What is an algorithm?
- Why do we need to consider algorithms carefully when doing data science?

Try this….

1. set *input* := your first name
2. set *sum* := 0
3. for each *letter* in *input* do the following:
   1. set *code* := position of *letter* in the alphabet
   2. set *sum* := *sum* + *code*
4. output *sum*

NOTE THAT THIS IS NOT PYTHON CODE! It is *pseudo-code* – a set of instructions written in a clear way independent of any programming language conventions.

What is an algorithm?

- Its a method, a recipe or a set of steps for doing something.
- Its a **well-defined procedure** that takes a value or set of values as **input** and produces some value or set of values as **output.**
- An algorithm is more than an interface which specifies functionality or mappings from inputs to output

Algorithms and interfaces

- One function in an interface may be implemented by a variety of different algorithms
- How else could we have achieved the “letter_sum” of a name?
- Consider the problem of long division. An interface would specify the required inputs and outputs i.e., the mathematical function, but not how to achieve the output in practice.
- What algorithms for division are there?

e.g.

$850 \div 16$

16, 32, 48, . . ., 848
1, 2, 3, ..., 53

850>16x10, but 850<16x100
So do
$\frac{850}{10} \div 16$ then multiply by 10

Why do we need to consider algorithms carefully in data science?

- We want to process large volumes of data....
- ... as quickly and cheaply as possible!
- How many seconds, minutes, hours or days will it take to process the data?
- How much of the data do we need in main memory at the same time?
- How much disk storage do we need? How fast can we access it?
- How can we speed processing up? Will adding another ‘node’ mean processing speeds up or slows down?

Module key information

- 11 week core module: 2 hour lecture + 2 hour lab for 10 weeks

Module convenor and main point of contact

Dr Adam Barrett,
Lecturer in Machine Learning and Data Science
adam.barrett@sussex.ac.uk

Office Hours: Mon 11-12
Weds 11-12

Please email me to book office hour slots

- All the info you need is on **Canvas**. Please ask questions, and post interesting module-related material you come across on Canvas.

- For student-to-student chat, **Discord**: https://discord.gg/bbr636z2z7

Labs

- Lab exercises are generally based on the lecture from the previous week.
- (Partial) solutions posted on Canvas on Fridays.
- Tip: Use One Drive to synch files between devices.

Assessments

- Two one-hour multiple choice quizzes, worth 10% each. Time windows for taking them:
  - Thurs 27th Oct – Fri 28th Oct (covering lectures 1-4).
  - Tue 29th Nov – Weds 30th Nov (covering lectures 5-9).

- Coursework worth 80%, submit report plus code, due Fri 9th Dec, 4pm.
  - Brief will be released middle of week 8.

Learning objectives

1. apply knowledge of standard data structures to the formulation and decomposition of big data
2. evaluate choice of computing model and data representation based on estimation and measurement of impact on space and time complexity and communication performance
3. understand the fundamental issues and challenges of developing parallel distributed algorithms for big data
4. apply appropriate methods to store and retrieve structured big data

Topics

Week	Who	Topic
1	Barrett	Data structures and data formats
2	Barrett	Algorithmic complexity. Sorting.
3	Barrett	Matrices: Manipulation and computation
4	Barrett	Similarity analysis
5	Rosas	Processes and concurrency
6	Rosas	Distributed computation
7	Barrett	Map/reduce
8	Barrett	Clustering, graphs/networks
9	Barrett	Graphs/networks, PageRank algorithm
10	Barrett	Databases
11		*independent study*

Data Types

- Programming languages typically support a number of atomic data **types:**-
  - integer
  - floating point number
  - string
  - character
  - boolean
- a single data item can be stored in a **variable**:

```
name = "Adam Barrett"
```

- python allows dynamic typing. Types of variables do not have to be declared and can change:

```
name = 25
```

Data structures

- A data structure is a **collection** of data items stored in memory **plus** a number of **operations** for manipulating that collection

- Specifies how the data is organised (at least conceptually) **and** how it should be accessed

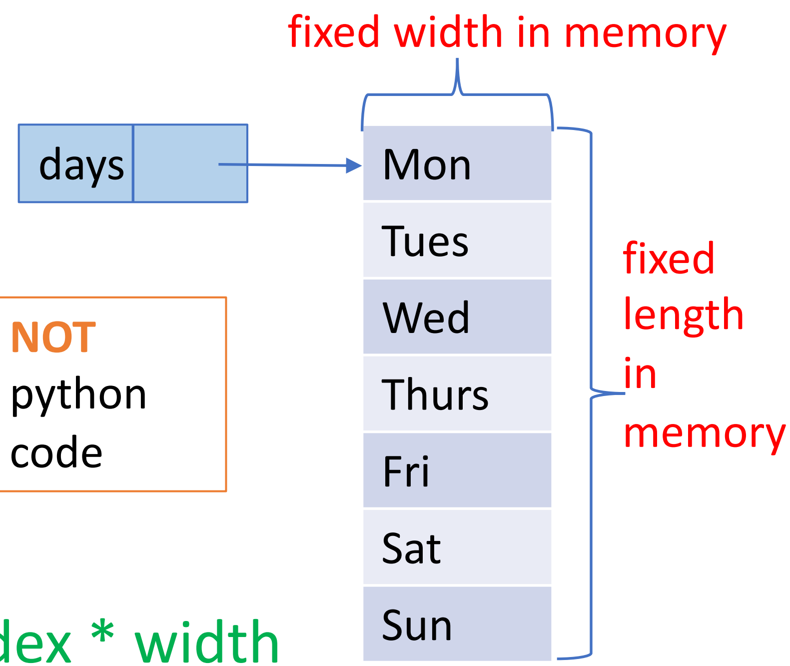
Static arrays

- The **array** is probably the most fundamental of data structures:

```
char[7][5] days = {"Mon",
"Tues", "Wed", "Thurs",
"Fri", "Sat", "Sun"};
printf("%s",days[2]);

> Wed
```

NOT python code



- fixed type and fixed length
- elements accessed via index:-
  - memory location = start + index * width
- indispensable in C and C++
- In Python: **numpy arrays**; **tuples**.

Linked lists

- **dynamic**, grow and change over time

- do not store pointers or indices to every item
- store a link or pointer from each item to the next
- perfect when data is going to be accessed sequentially
- easy to add/append items
- easy to concatenate two lists
- expensive to access i^{th} item
  - worst case, follow all n links
  - $O(n)$

Python Lists

code	output	description
planet_list=[]		create a new empty list and store it as planet_list
planet_list=[“Mercury”, “Venus”]		create a new list with n items and store as planet_list
planet_list.append(“Mars”)		add an item on to the end of a list
more_planets=[“Earth”,”Jupiter”] bigger_list = planet_list + more_planets		concatenate two lists
print (len(bigger_list))	5	print the length of a list
for p in more_planets: print (p)	Earth Jupiter	iterate over a list and do something with each member
print (bigger_list[2])	Mars	index (or splice) into a list
print (len(bigger_list[2:]))	3	splice a list

Are python lists arrays or linked lists?

- Python lists are (normally) implemented via C routines
- Underlying representation is an array
- Variable length is achieved by over-allocation
- Re-allocate if the array becomes full

- This means fixed cost random access time
  - i.e., time to access i^{th} element is independent of the length of the list n
  - $O(1)$
- Good list functionality (append, concatenate. . .)

Maps / Dictionaries

- Not all data is sequential
- Conceptually, a **map** or **dictionary** is a collection of (**key-> value**) pairs.
- Might be a set of attributes for an object e.g.,

  adammap = {“name”: “adam”, “occupation”: “lecturer”}
- Or a given attribute for a set of objects e.g.,

  occupations ={“adam”: “lecturer”, “bob”:”porter”}
- Values might be other data structures e.g., lists and dictionaries

  employees = {“bob”:{“name”: “bob”, “occupation”: “porter”,
  “age”: 49, “children”:[“fred”, “nancy”]},
  “sue”:{“name”: “sue”, “occupation”: “lawyer”,
  “age”: 34, “children”:[]}

Python dictionaries (dicts)

Code	description
emps={}	new empty dict called emps
emp={“name”: “sue”, “occ”: “lecturer”}	new dict with 2 keys called emp
print (emp[“name”])	lookup name in emp output: sue
emp[“occ”]=“reader”	update occupation in emp
emp[“age”]=42	create new field in emp
print (emp.keys())	print all of the keys in emp output: [“occ”, “age”, “name”]
emps[emp[“name”]]=emp	store emp in emps with key = name

for (name,record) in emps.items():
    if record[“age”] > 40:
        print (name)
	print names of all emps where age > 40
	output: sue

Lookup in a dictionary

- how do you implement a dictionary?
- a list of key-value pairs?

key	value
adam	lecturer
bob	porter
sue	lawyer
. . .	. . .

- lookup would be really slow
- might have to check every key before you find the right one
- *O(n)*

Thought experiment

- We have n folders of information about students (n = 12, n = 10, 000?)
- Each folder is labelled with the student's name
- We have n pigeon-holes to store the folders in
- How best do we arrange the folders for fast storage and fast access?

Hash tables

- Store the element with key *k* at slot *h(k)* where *h* is a hash function which maps *k* to a number in the range *{0,...,n-1}*
- e.g.: `h(k) = letter_sum(k) mod n`
- In an ideal world, access time is independent of *n*, i.e., *O(1)*, and is just dependent on time to compute hash function

Collisions

- a hash collision occurs when 2 (or more) keys map to the same slot
- minimise collisions by using a good hash function
- resolve collisions using techniques like chaining or open-addressing

Hash functions

- a good hash function satisfies the assumption of simple uniform hashing: *each key is equally likely to hash to any of the n slots*
- interpret keys as natural numbers
  - e.g., the name "ted" could be interpreted as the triple (116, 101, 100) by looking up the characters in the [ASCII character set]
  - express sequence using radix-128 notation

  ted = $116 * 128^2 + 101 * 128 + 100 = 1913572$
- map into *n* slots using e.g., modular division
  - good values for *n* are primes not too close to exact powers of 2

Optional maths exercise:

Consider a hash function in which h(k) = k mod m, where m= $2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if string *x* can be derived from *y* by permuting its characters, then *x* and *y* hash to the same value.

Collision resolution

- **Chaining** resolves collisions by putting all elements that hash to the same slot in a linked list
  - analogous to putting folders that collide in same slot and searching through them at access time
- **Open addressing** successively probes the hash table until a match or an empty slot is found
  - Hash function takes the probe number as second input.

    E.g.

    - Linear probing: $h(k,i)= (h_1(k) + i) \bmod n$

    - Double hashing: $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod n$

Data formats and data structures

- Here is an extract from the open exoplanet catalogue dataset, which is stored in a csv file

PlanetIdentifier	TypeFlag	PlanetaryMassJpt	RadiusJpt	PeriodDays
KOI-1843.03	0	0.0014	0.054	0.1768913
KOI-1843.01	0		0.114	4.194525
KOI-1843.02	0		0.071	6.356006
Kepler-9 b	0	0.25	0.84	19.22418
Kepler-9 c	0	0.17	0.82	39.03106
Kepler-9 d	0	0.022	0.147	1.592851
GJ 160.2 b	0	0.0321		5.2354
Kepler-566 b	0		0.192	18.42794624

- What Python data structure(s) would you most naturally use when you import a csv file like this?

What Python data structure would you use to represent a csv file?

- List
- Dictionary
- List of lists
- Dictionary of dictionaries
- List of dictionaries
- Dictionary of lists
- Something else

csv.reader ≈ list of lists

```
import csv
def readfile(filename):
    with open(filename) as instream:
        csvreader=csv.reader(instream)
        lines=[]
        for line in csvreader:
            lines.append(line)
    return lines

lines=readfile(filename)
print ("{} : {}".format(lines[0][0], lines[0][3]))
print ("{} : {}".format(lines[3][0], lines[3][3]))
```

```
PlanetIdentifier : RadiusJpt
KOI-1843.02 : 0.071
```

Representing json

- Here is an extract from items.json file scraped from kaggle in DSRM:


```
[
{"filename": ["perSpindle01.csv", "perSpindle02.csv",
"perSpindle03.csv", "perSpindle4_6.csv", "perTheta.csv"],
"link": "/jbouv27/eeg", "title": "EEG Analysis", "desc":
"Sleep Pattern detection", "popularity": 14, "size": 35387},
{"filename": ["cleanedmrdata.csv"], "link": "/wpncrh/marginal-
revolution-blog-post-data", "title": "Marginal Revolution Blog
Post Data", "desc": "Author Name, Post Title, Word count,
Comment Count, Date, Category Tag", "popularity": 10, "size":
6529129},
```


- What Python data structure(s) would you use to represent a json file like this?

What Python data structure would you use to represent a json file?

- List
- Dictionary
- List of lists
- Dictionary of dictionaries
- List of dictionaries
- Dictionary of lists
- Something else

json ≈ list or dict of dicts

```
import json
def readjsonfile(filename,key='title'):
    with open(filename,'r') as instream:
        data=json.load(instream)

    datadict={}
    for item in data:
        #what if key does not exist
        #for item or is repeated?
        datadict[item[key]]=item
    return datadict

jsonfile="items.json"
key='title'
data=readjsonfile(jsonfile,key)
a='Pokemon with stats'
b='popularity'
print("{} of {} is {}".format
      (b,a,data['Pokemon with stats']['popularity']))
```

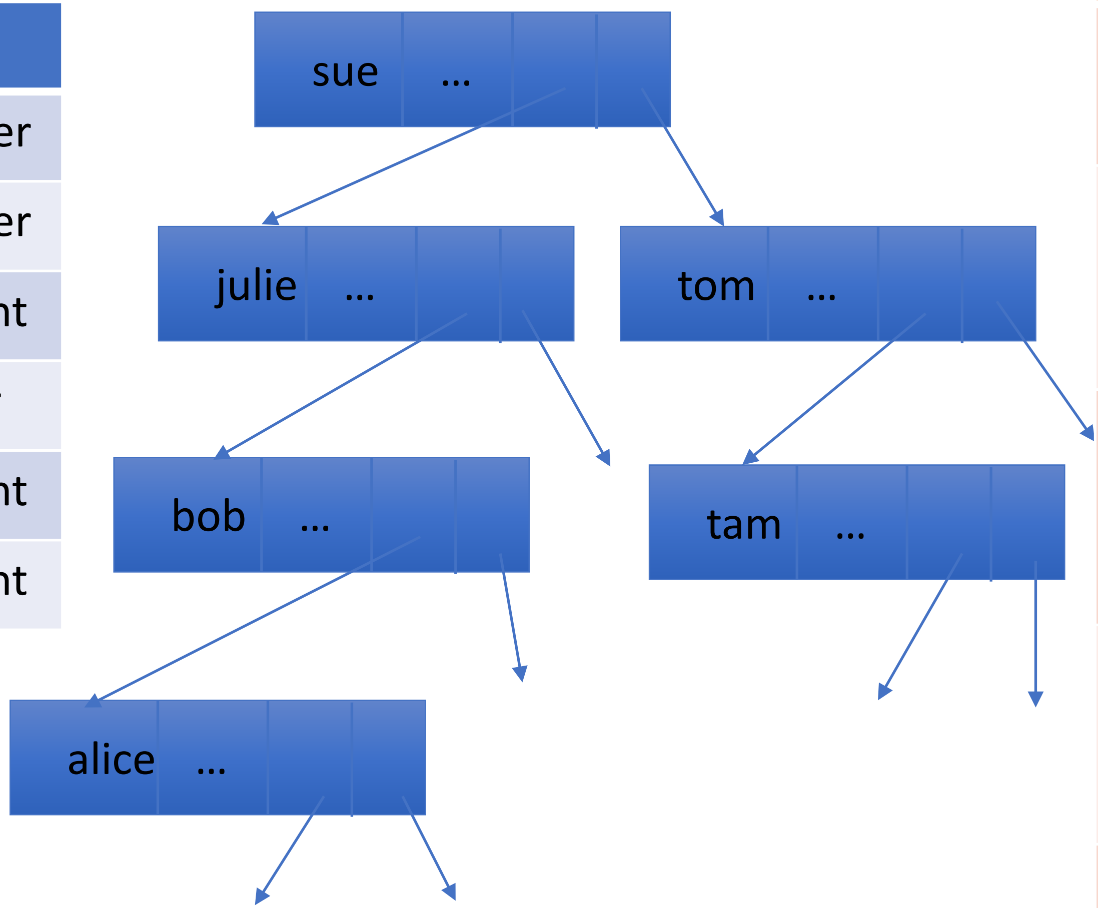
```
popularity of Pokemon with stats is 2310
```

Which format is more compact, csv or json?

Binary Search Trees

occupations = { }

key	value
sue	lecturer
julie	lecturer
tom	student
bob	porter
tam	student
alice	student



height	nodes	capacity
0	1	1
1	2	3
2	4	7
3	8	15
h	2^h	$2^{h+1}-1$

Binary Search Trees

- balanced tree -> number of nodes at height h is 2^h
- capacity n = $\sum_{i=0}^{h} 2^i$ = 2^{h+1} - 1
- query runs in $O(\log_2 n)$ time
- unbalanced tree will result if the keys are pre-sorted -> linked list and query runs in O(n) time
- useful data structure when there is a strict order over the keys but keys will be presented in random order
- can be used to implement dictionaries

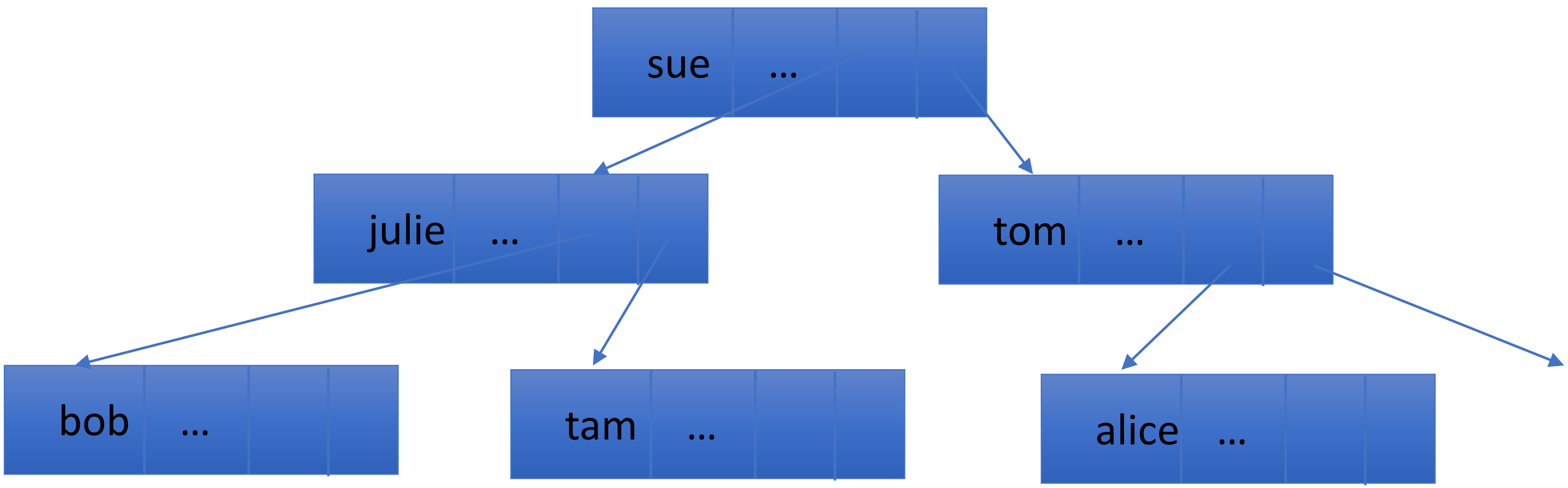
Example

- Suppose a librarian wants to know how many books in their library have an author that starts with A.

- Would it be quicker if the dictionary were stored in a hash table or a binary search tree?

Heaps (to be covered next week)

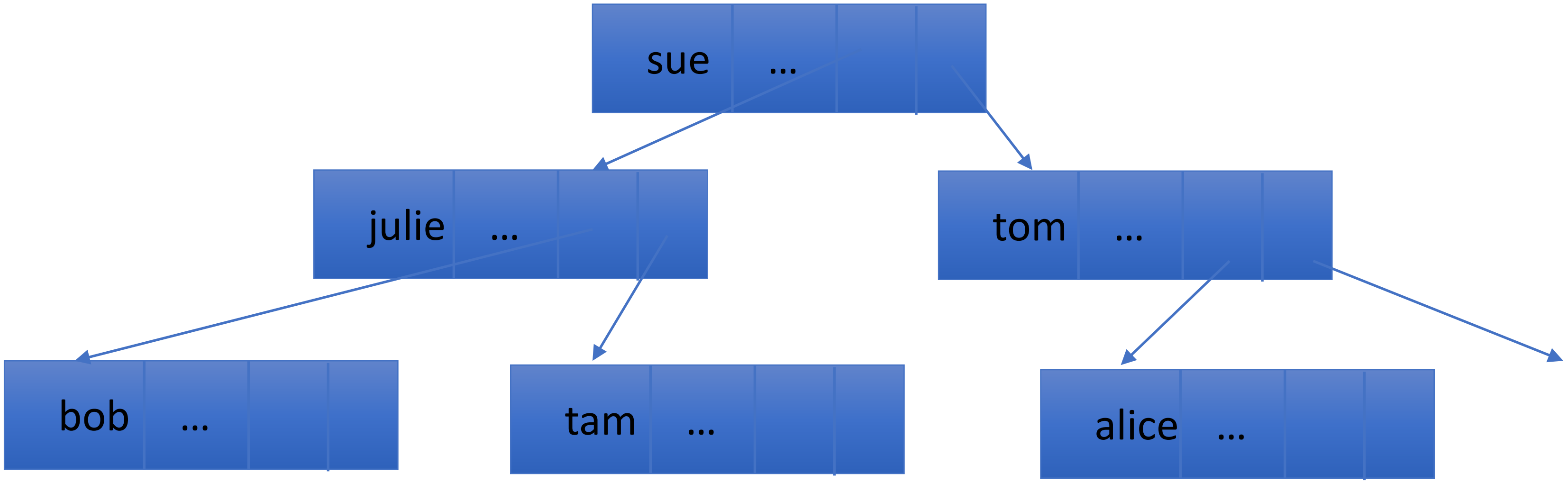
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- This is a complete binary tree. Does it satisfy the heap property?

Heaps

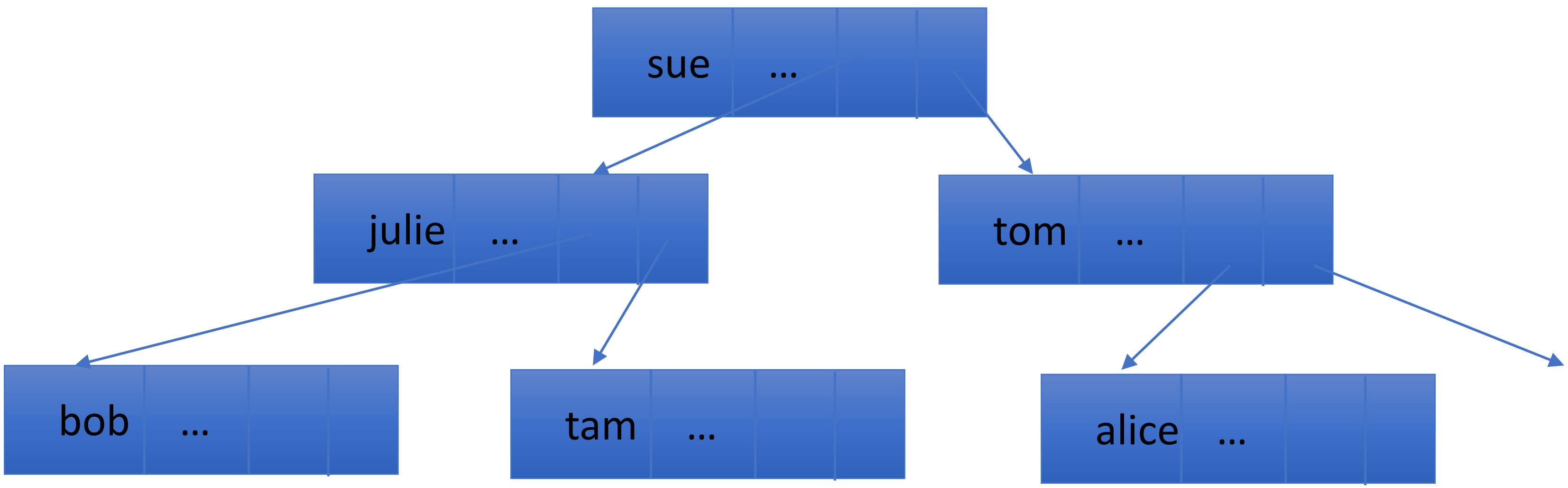
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- Run heapify on all non-leaf nodes in a bottom-up fashion i.e.
  - heapify([tom, julie, sue])

Heaps

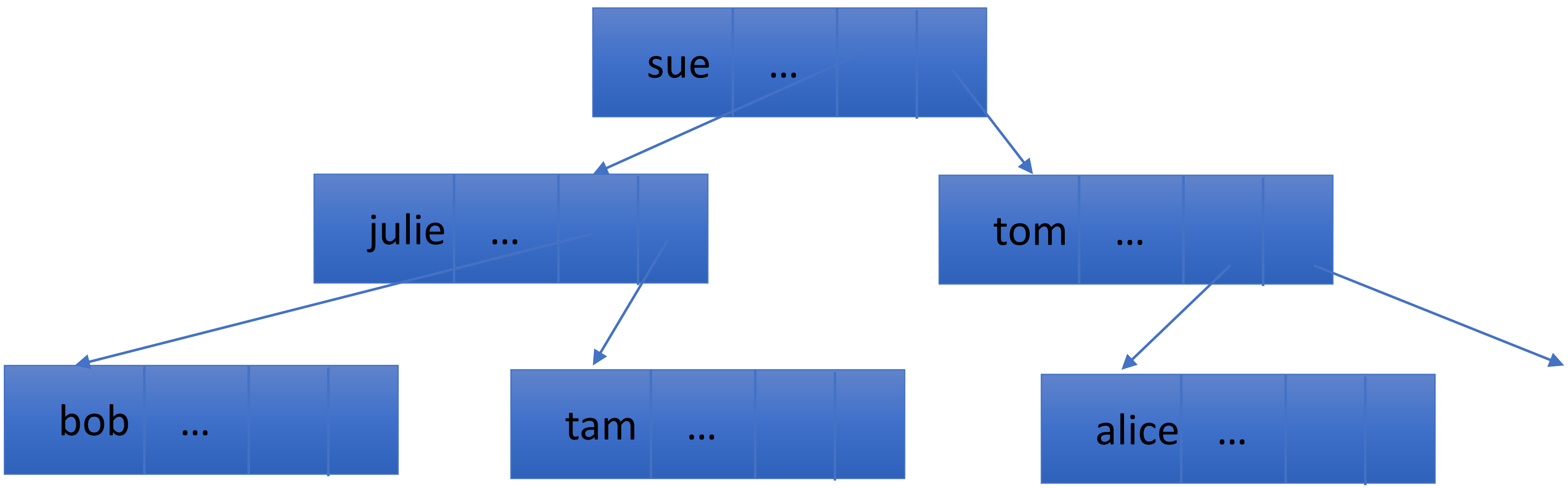
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(tom) :- tom > alice so heap property satisfied.

Heaps

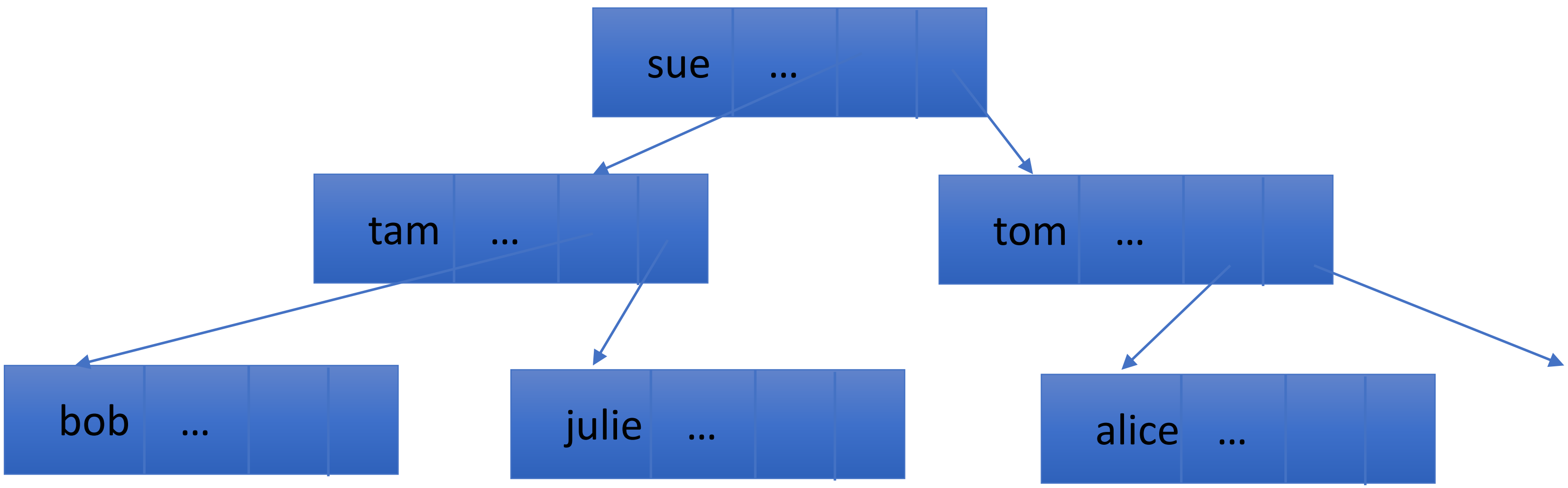
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(julie)
  1. **tam > julie so swap these nodes**
  2. heapify(julie)

Heaps

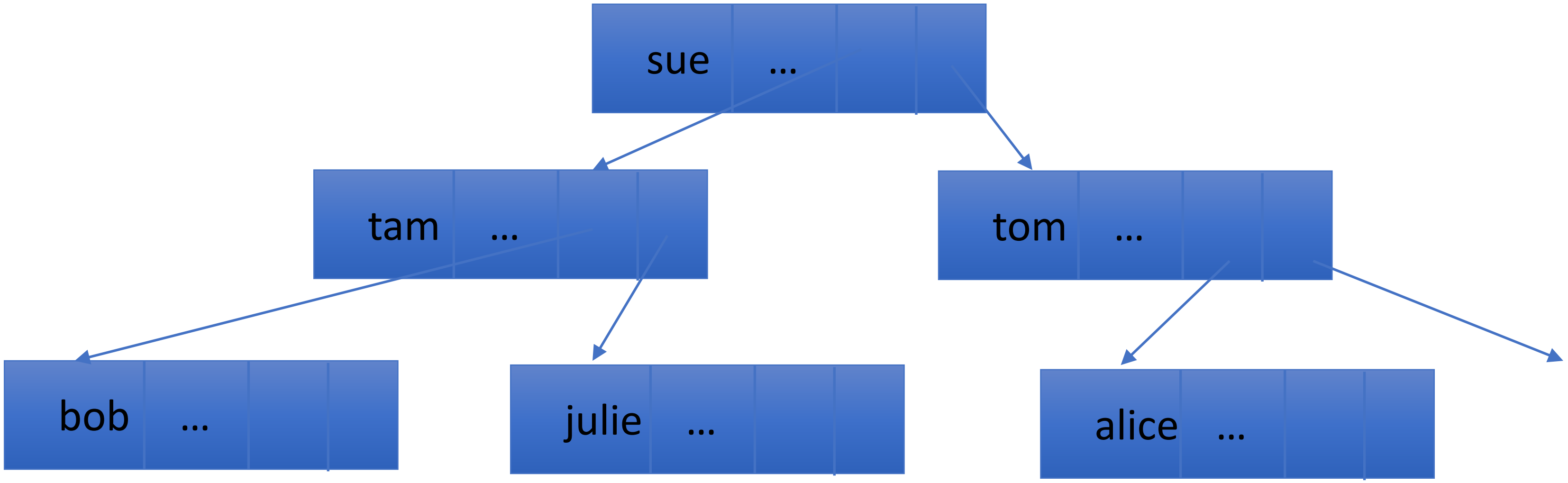
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(julie)
  1. tam > julie so swap these nodes
  2. **heapify(julie)**

Heaps

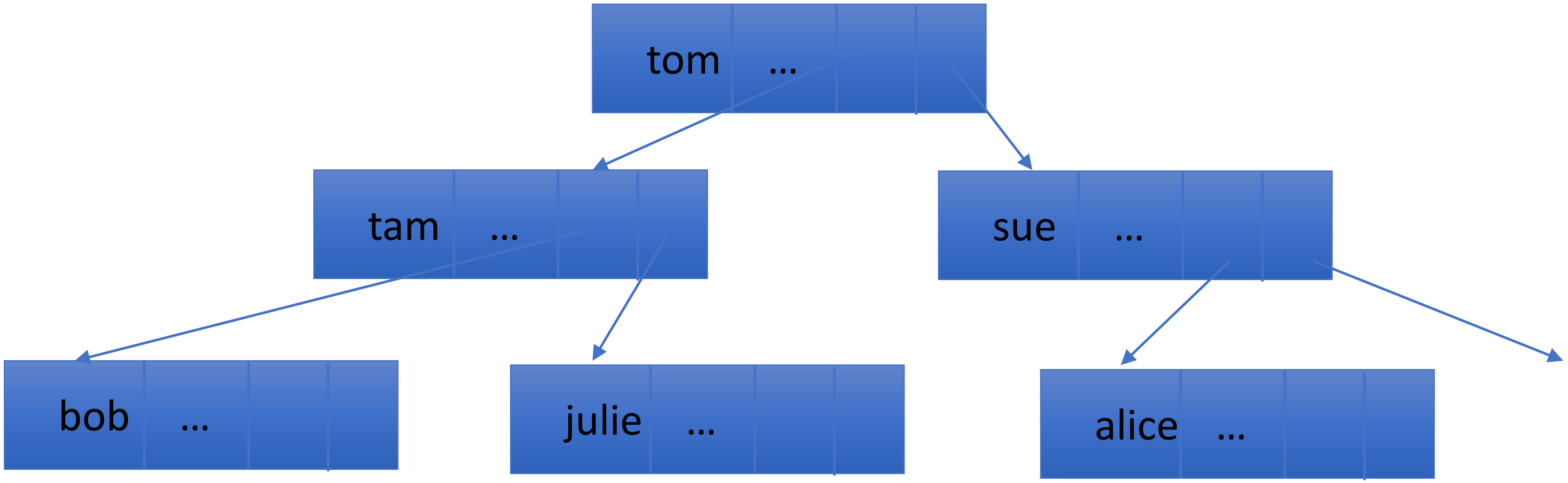
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :-
    1. **tom > sue & tam so swap sue and tom**
    2. heapify(sue)

Heaps

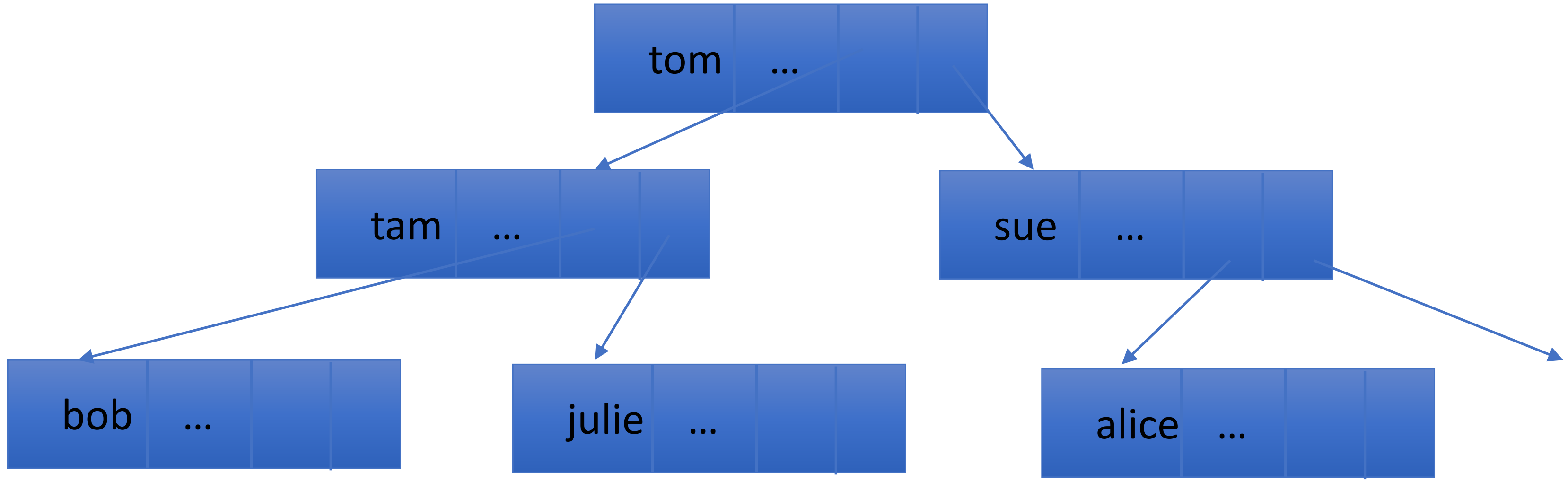
- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :-
  1. tom > sue & tam so swap sue and tom
  2. **heapify(sue)**

Heaps

- Heaps are **complete binary trees** which satisfy the **heap property**: Value[parent(i)] ≥ Value[i]



- heapify(sue) :- sue > alice so heap property satisfied

Why are heaps useful?

- Can be used for sorting (the Heapsort algorithm)
  - Remove maximum element at root of heap
  - Take last element, place it at the root and then heapify
  - Repeat
- Efficient implementation of a ‘priority queue’
- O(n log n) to build a heap

Summary

- We have talked about what an algorithm is and why data scientists should care about algorithms. We have introduced a number of important data structures including arrays, linked lists, python lists, dictionaries, hash tables, binary search trees and heaps.