Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 12

Dr Nick Hayward

Video - Algorithms and Data Structures

The Joy of Data - Packet Switching

The Joy of Data - Packet Switching - UP TO 41:40

Source - BBC - The Joy of Data - YouTube

hash tables - prevent duplicate entries - part 1

- key consideration for working with hash tables
  - *prevention of duplicate entries for data*
- e.g. consider initial scenario for user accounts and registration
  - *new user submits preferred username*
    - username is checked against existing records for user accounts
  - *if username already exists*
    - return user to registration page & try again...
  - *otherwise*
    - allow user to continue registration
- sounds like an easy process
  - *quickly creates a large dataset of user accounts, names, &c.*
- each time a new user submits a registration request
  - *app has to scan large, growing list of users to check for existing usernames*

hash tables - prevent duplicate entries - part 2

- better option using *hash table*
  - *create new table to keep track of users and associated usernames*

```
user_accounts = dict()
```

- then check if username already exists in table

```
user = user_accounts.get("daisy")
```

Algorithms and Data Structures

hash tables - prevent duplicate entries - part 3

- return data for queried username

```python
# create hash table for address book
user_accounts = dict()

# perform check for passed username
def check_users(name):
    if user_accounts.get(name):
        print("try again - username '" + name + "' already exists...")
    else:
        user_accounts[name] = "active"
        print("user account created...")

# check user accounts
check_users("daisy")
check_users("emma")
check_users("daisy")
```

- if we store such records in a list of users
  - *queries become very slow as number of users increases...*
  - *i.e. need to run a simple search over entire list*
- checking for duplicate entries in a hash table is very fast
  - *well-suited for this type of usage*

Algorithms and Data Structures

hash tables - caches - part 1

- another common use case for hash tables is *caching* with applications
- consider a web application
  - *regularly receives multiple requests for pages, data, and media*
  - *requests from both authenticated users and anonymous users*
- e.g. consider a standard usage pattern
  - *user submits request to web application - sent to defined host server*
  - *server processes request - returns data and updated page for web application*
  - *user views and interacts with page...*
- a standard, abstracted pattern for such usage
  - *provides data and page for user*
- may also find many users submit same requests for data and pages
  - *e.g. latest weather, news, photos...*
- requests may take a few seconds, perhaps even minutes, to process and return
- common usage scenario for website caching
  - *i.e. remembering processed data for submitted queries and requests*
  - *saves repetitive requests and recalculations of data*

Algorithms and Data Structures

hash tables - caches - part 2

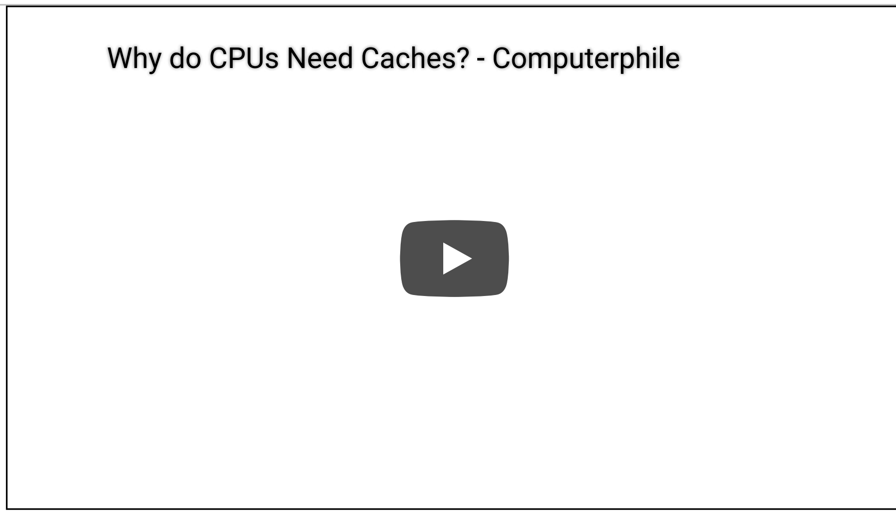
- similar pattern for authenticated users and anonymous users
  - *logged-in user may require personalised, tailored data and pages*
  - *calculated and returned by the server*
- anonymous users will see same page structure and data
  - *i.e. web application receives repetitive requests for data and pages*
  - *e.g. user's registration and login page*
- help lessen server usage
  - *server remembers such pages for anonymous users*
  - *sends same page...*
- *caching* of pages and data has two notable advantages
  - *requested web page for application returned faster*
    - removes need for repetitive requests and calculations
  - *server and web application has less work to do...*

hash tables - caches - part 3

- data may be cached in a hash table
  - *i.e. define mapping of URLs from web app's pages to associated page data*
- as user visits and requests various pages and data
  - *web app checks for cached versions of page in hash table*
  - *if page exists - server sends cached copy for request to user*
- hash tables are particularly useful for the following
  - *modeling relationships*
  - *filtering duplicate entries*
  - *caching data*

Video - Algorithms and Data Structures

cache and systems



What's a cache for? - UP TO 4:08

Source - What's a cache for? - YouTube

hash tables - collisions - intro

- better understand relative merits and performance of hash tables
  - *need to consider collisions*
- might strive for an ideal solution
  - *i.e. hash function always maps different keys to different slots in array*
  - *n.b. not always possible*
- for many hash functions, simply not possible to achieve

hash tables - collisions - example

- consider initial example
  - *simple hash function assigns data in array alphabetically*
- for single items of each letter
- this function will work fine
  - *i.e. assign a single title to a given letter*
  - *maintains fast performance*
- if we start adding further titles per letter
  - *encounter issue of collision*
  - *i.e. multiple keys assigned same index in array*
- if we continue with current assignment of index per letter
  - *overwrite previous titles with new title*
  - *ie. query may work but return value will be incorrect*
- need to consider a solution for such collisions

Algorithms and Data Structures

hash tables - collisions - linked list

- simplest solution for this issue of collisions
  - *use a linked list with hash table*
- e.g. if multiple keys are mapped to same slot in hash table
  - *create a linked list at that position*
- i.e. *d* may store multiple records in hash table
  - *using a linked list as a value in the array*
- a working solution for smaller linked lists of records
  - *not a fast solution for larger hash tables*
  - *still restricted by slower search of linked list for chosen letter*

hash tables - collisions - considerations

- *collision* demonstrates importance of chosen *hash function*
  - *crucial for performance and maintenance of hash table*
- good hash function will map keys evenly across hash table
- good hash function will create fewer collisions within hash table

hash tables - performance

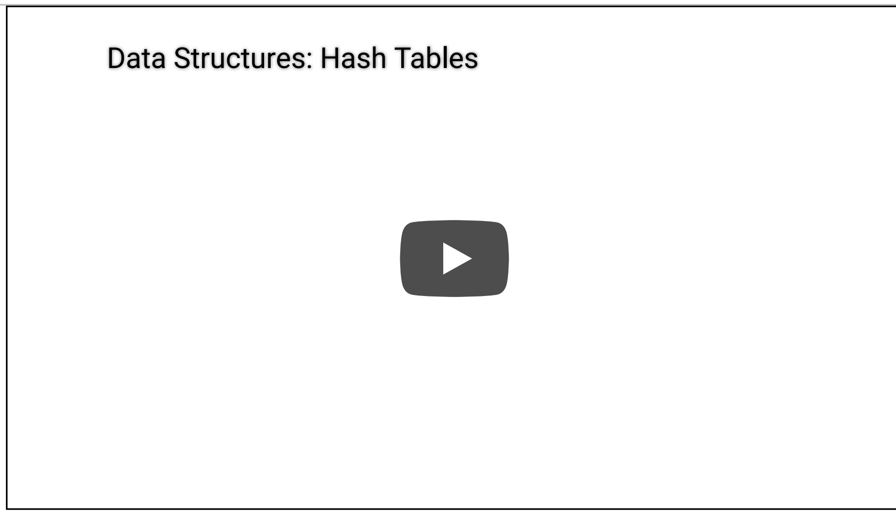
- *bookshop* example demonstrated
  - *need to query data instantly*
  - *i.e. at least as far as possible...*
- a real benefit of *hash tables* is their performance
- e.g. summary of *hash table* performance

operation	average case	worst case
search	O(1)	O(n)
insert	O(1)	O(n)
delete	O(1)	O(n)

- for average cases
  - *hash table is* `O(1)`*, constant time*
  - *n.b. does not mean instant time*
  - *means performance time will stay same regardless of hash table size*

Video - Algorithms and Data Structures

hash tables - part 4



Hash tables - performance - Java - UP TO 6:08

Source - Hash tables - performance - YouTube

hash tables - performance - average case comparison

- quick comparison
  - *simple search will take `O(n)`, linear time*
  - *binary search takes `O(log n)`, log time*
- compared such functionality on graphs
  - *may see a flat horizontal line for a hash table*
- why is graph for a *hash table* a flat line?
  - *representative of underlying nature of query relative to a hash table*
- i.e. regardless of size of hash table
  - *e.g. one element or ten million*
  - *able to retrieve element in same amount of time*
- same as querying a known array
  - *also takes constant time for indexed queries*
- for *average case*
  - *hash tables are very fast...*

Algorithms and Data Structures

hash tables - performance - worst case comparison

- compare *worst case* performance
  - *hash table takes `O(n)`, linear time for everything*
  - *very slow for applications &c.*
- useful to compare this performance
  - *e.g. against arrays and linked lists*

operation	hash table (avg case)	hash table (worst case)	arrays	linked list
search	O(1)	O(n)	O(1)	O(n)
insert	O(1)	O(n)	O(n)	O(1)
delete	O(1)	O(n)	O(n)	O(1)

- average case for hash tables
  - *Hash tables are as fast as arrays at searching*
  - *i.e. getting an indexed value*
  - *also as fast as linked lists for insertion and deletion*
- worst case may raise concerns with *hash tables*
- for worst case
  - *hash table is slow at each of these operations*
- need to ensure we do not hit worst case performance for hash table
  - *common option for reducing this possibility is to avoid collisions*
- help with collision avoidance
  - *low load factor*
  - *good hash function*

Algorithms and Data Structures

hash tables - load factor - part 1

- hash table's *load factor* is straightforward to consider and calculate
  - *i.e consider the following*

```
number of items in hash table / total number of slots
```

- may use array for storage of a hash table
- allows us to easily check number array usage...

hash tables - load factor - part 2

- e.g. consider basic hash table

```
-----------------------------------
|  3 |     |     |  7 |     |     |
-----------------------------------
```

- this hash table has a load factor of `2/6`
- following hash table has a load of `1/3`

```
-------------------
|    |  9  |     |
-------------------
```

- *load factor* measures usage and capacity of current hash table *hash tables - load factor usage - part 1*

- why is this inherently useful or important?
- e.g. if we have 100 or 200 elements
  - *need to store in a hash table*
  - *need to know if that table can efficiently handle data*
- e.g. if table has one hundred slots,
  - *load factor will be 1*
- if data increases to 200
  - *load factor will double to 2*
  - *i.e. each element will not get unique slot in table*
- load factor greater than 1
  - *poor usage for most cases*
  - *i.e. more elements than space in table*
- as load factor continues to grow
  - *need to add more slots to hash table*

hash tables - load factor usage - part 2

- as hash table is reaching capacity load
  - *need to consider a resize*
- depending on programming language used for hash table
  - *may need to create a larger array for table*
  - *good heuristic for increase is to double array size*
  - *e.g. double size to* `200`
- then re-insert existing elements into new hash table using hash function
- new hash table has an improved load factor
  - *i.e.* `100/200` *or* `0.5`
  - *lower load factor reduces number of collisions in table*
  - *table should also perform better*
- good heuristic for resizing a hash table
  - *when load factor is above* `0.7`
- resizing may incur a cost in time and performance
  - *resizing is expensive*
  - *need to ensure we do not resize a hash table on a regular basis*
- even with resizes - hash tables still average `O(1)`

hash tables - hash function

- load factor is important consideration for usage and management of a hash table
  - *not possible without a good hash function*
- i.e. a good hash function should try to evenly distribute values in underlying array
- a poor hash function will create groups of values
  - *thereby producing many collisions in hash table*
- may never need to write a hash function from scratch
  - *a good example to consider is **SHA function***

Video - Algorithms and Data Structures

hash tables - a custom hash function

Hash tables - custom hash function - UP TO 9:24

Source - Hash tables - custom hash function - YouTube

hash tables - hash function - SHA function - part 1

- as we use a hash table we need a good hash function
  - *determine where to assign a data element in an array*
  - *i.e. help work out even distribution to optimise load factor*
  - *try to avoid collisions as much as possible*
- able to perform constant-time lookups for hash table
  - *i.e. using a good hash function*
- good hash function
  - *app may quickly check value of key*
  - *i.e. returns index of array to check in* `O(1)` *time*
- *secure hash algorithm* (SHA) function
  - *example of good hash function*
  - *adapt for a hash table*
- e.g. pass a string such as `hello` to *SHA* and return a hash

```
'hello' -> 4dg54ab...
```

- *SHA* is a hash function
  - *generates a hash (a short string)*
  - *SHA will generate a different hash for every string*

hash tables - hash function - SHA function - part 2

- common usage may check and validate files
  - *e.g. file sharing, project usage &c.*
  - *particularly useful for very large files*
- e.g. two users may need to check and verify they're using the same file
  - *even though they may have separate copies.*
- SHA is used to calculate hash
  - *each user may then check their file against the hash*
- SHA is also useful for verification of passwords
  - *SHA used to compare strings without revealing original string content*
- e.g. a database may store generated SHA hash
  - *instead of original password string*
- to check and use these passwords
  - *hash input string*
  - *then check hash against saved hash in database*
  - *i.e. only comparing hashes, not original string passwords*
- another benefit of this use of SHA
  - *hash is one way*
  - *may get hash, but not original string from hash*

Video - Algorithms and Data Structures

hash algorithms and security - summary of hash function...

Hash algorithms and security - UP TO 3:35

Source - Hash algorithms and security - YouTube

Algorithms and Data Structures

hash tables - hash function - SHA function - locality insensitve

- another useful and important feature of SHA usage
  - *its lack of locality sensitive hashing*
- e.g consider the following string

```
daisy -> hu9m362g...
```

- if we modify string by a single character
  - *then generate the hash*
  - *SHA will return a new, different hash...*

```
daily -> h4dg96hj...
```

- clear benefit of this approach
  - *can't compare hashes to check for reverse engineering the hash*
  - *i.e. hashes can't be compared to iteratively return original string*

Algorithms and Data Structures

hash tables - hash function - SHA function - locality sensitive

- may be instances where we actually need such *locality sensitive* hash functionality
  - *may consider Simhash*
- modify a string and then generate a hash using *Simhash*
  - *Simhas generates hash that is only a slight update to previous hash*
- benefit is use for comparison of hashes
  - *e.g. determine proximity of two strings*
- for certain use cases, this can be particularly useful
- e.g. collation of texts, web crawlers &c. may use this approach
  - *check various online sources, then use Simhash to identify duplicates*
- editors, teachers, and anyone who wants to check various textual sources
  - *may use Simhash for this collation...*
- verification of copyrighted material is another sample use for *Simhash*

hash tables - hash function - SHA function - SHA family

- SHA is a group of algorithms we may use for hashing values
- e.g.
  - *SHA-0*
  - *SHA-1*
  - *SHA-2*
  - *SHA-3*
- if we need to use SHA to hash passwords &c.
  - *commonly use SHA-2 or SHA-3*
- further details are available at the following URL
  - *SHA algorithms - Wikipedia*

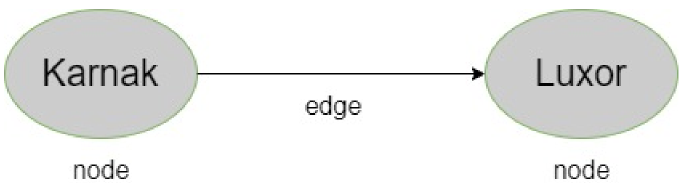
Video - Algorithms and Data Structures

SHA - Secure Hashing Algorithm

SHA: Secure Hashing Algorithm - UP TO 8:38

Source - SHA: Secure Hashing Algorithm - YouTube *graphs - intro*

- graph data structure in computer science
  - *a way to model a given set of connections*
- commonly use a *graph* to model patterns and connections for a given problem
- e.g. connections may infer relationships within data
- graph includes *nodes* and *edges*
  - *help us define such connections*
- e.g. we have two nodes with a single edge



Graph Nodes and Edge

- each node may be connected to many other nodes in the graph
  - *commonly referenced as neighbour nodes*

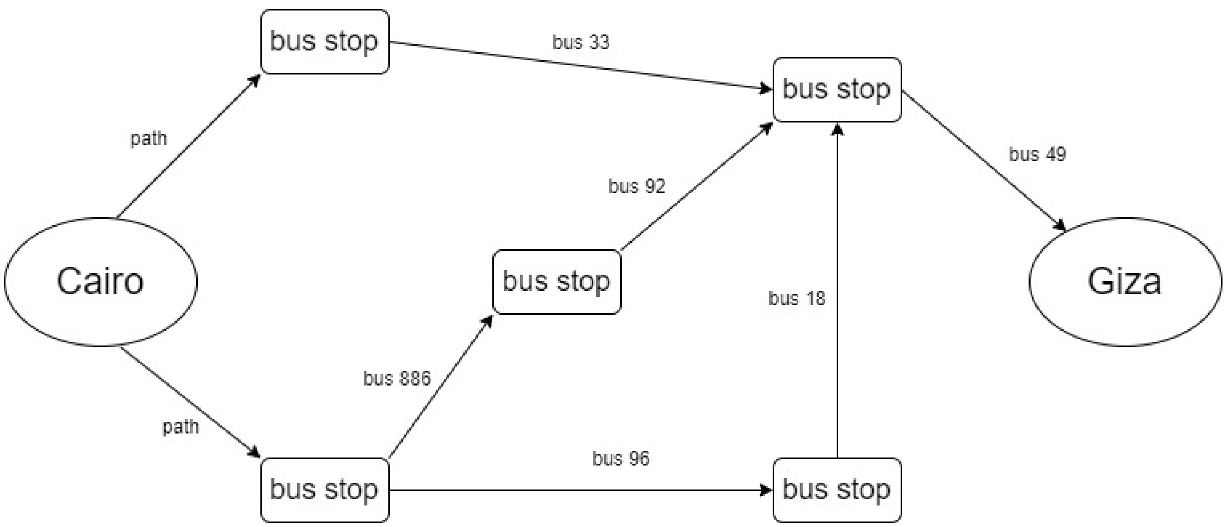
graphs - sample use case

- common use-case for describing conceptual use of graphs
  - *consider travel options and routes between various locations*
- e.g. consider traveling around Egypt to visit historical sites
  - *might need to travel from centre of Cairo to Giza*
  - *i.e. to view pyramids, Sphinx...*
- may use a bus to travel from centre of Cairo to Giza plateau
  - *need to optimise route with minimum number of possible connections*
- i.e. may have numerous options for available bus routes
  - *optimal choice allows us to find path with fewest steps*
- first step to solve this problem is to define it as a *graph*...

Image - Graphs

sample use case

- e.g. consider the following routes



Graph Routes

Video - Algorithms and Data Structures

graphs - Java - part 1

Intro to Graphs - UP TO 0:34

Source - Graphs - Java - YouTube

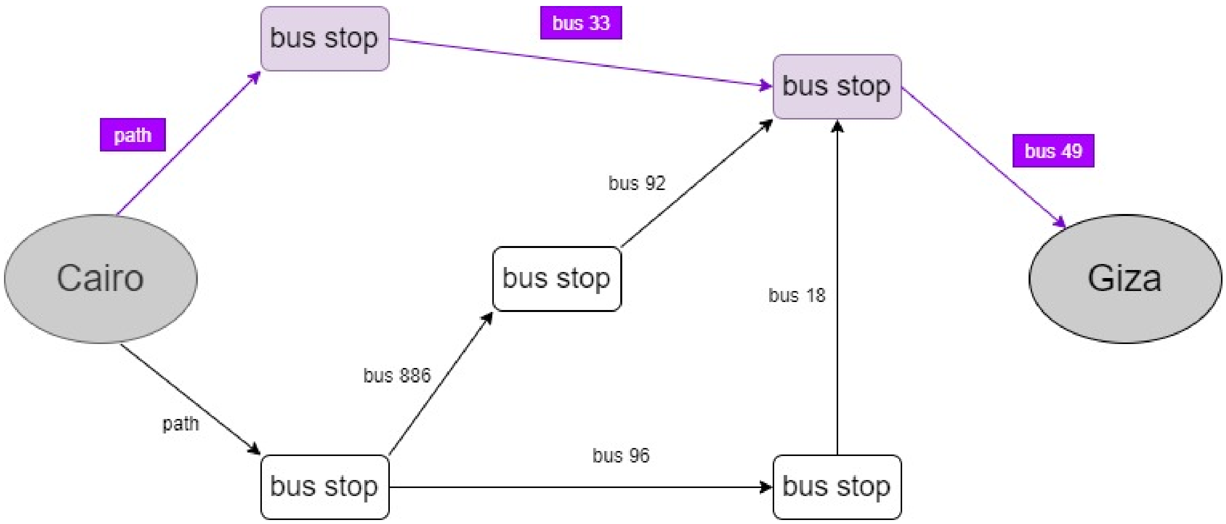
graphs - optimal path - part 1

- need to define an algorithm to find optimal path
  - *i.e. to travel from Cairo to Giza*
- begin by checking if we can take a single *step*
  - *to get from Cairo to Giza*
  - *obviously, this option is not available for current routes*
- then try two steps
  - *again, we can clearly see this is not possible*
- if we try three steps we can travel from Cairo to Giza…

Image - Graphs

optimal path

- e.g. shortest route for graph



Graph Routes - shortest

Video - Algorithms and Data Structures

graphs - Java - part 2

Graphs - Java - Depth-first Search - UP TO 2:20

Source - Graphs - Java - YouTube

Algorithms and Data Structures

graphs - optimal path - part 2

- need to take path to first bus stop
- then take *bus 33* to next bus stop
- then travel on *bus 49* to final destination, *Giza*
- takes us *three* steps to travel from centre of Cairo to Giza
- other possible routes using various combinations of buses
  - *longer than optimal route with three steps*
- problem is formally known as *shortest path problem*
- may use a *breadth-first search* algorithm
  - *use to initially consider and solve this type of problem*

graphs - breadth-first search - intro

- Breadth-first search is an algorithm we may use to query a *graph* data structure
- i.e. use this search algorithm to check for a couple of initial queries
- e.g.
  - *can we find a path from one node to another - does a path exist?*
    - e.g. from node ‘Cairo’ to node ‘Giza’
  - *what’s the shortest path between two nodes*
    - e.g. shortest path from ‘Cairo’ to ‘Giza’
- use *breadth-first* to determine shortest path for previous use case
  - *i.e. from ‘Cairo’ to ‘Giza’*

Video - Algorithms and Data Structures

graphs - Java - part 3

Graphs - Java - Breadth-first Search - UP TO 2:58

Source - Graphs - Java - YouTube

graphs - breadth-first search - does a path exist? - part 1

- may use *breadth-first* search to check for a given node in a defined graph
- e.g. we might begin with an initial list of family members
  - *use this list to check for a family member*
  - *e.g. who has visited a specific location in Egypt, perhaps 'Giza' or 'Karnak'*
- seems like a straightforward initial search
  - *begin by defining a list of current family members*
  - *use list to start our search*
- as we check each family member in list
  - *check whether they have visited a in Egypt...*

graphs - breadth-first search - does a path exist? - part 2

- initial search shows
  - *no family members who have visited site of Karnak*
- instead of closing search
  - *expand list to search through their family members...*
- search records of each family member
  - *also add all of their family members to the list*
- now able to search all of our family members
  - *plus a growing network of additional, connected family members*
- if a given family member has not visited Karnak
  - *add their family members and continue search*
- with this particular algorithm
  - *search entire network*
  - *until we find someone who has visited Karnak*
- i.e. checking if a path exists in graph
  - *someone who has visited Karnak*

Algorithms and Data Structures

graphs - breadth-first search - find the shortest path - part 1

- as we search our list
  - *may find multiple family members that have visited Karnak*
- which family member is closest?
- i.e. what is the shortest path between nodes
- can we find closest visitor to Karnak
- if we consider list of family members
  - *initial family members are defined as a first-degree connection*
  - *their family members are second-degree connections*
  - *and so on...*
- for search performance
  - *prefer a first-degree connection*
  - *then second-degree*
  - *until we find shortest path to a match*

Algorithms and Data Structures

graphs - breadth-first search - find the shortest path - part 2

- need to search all *first-degree* connections
  - *before we check second-degree*
  - *then continue to broaden search*
- search pattern is *breadth-first* search
- search algorithm will continue to radiate out
  - *radiates from a defined starting point*
- begin with first-degree connections
  - *then radiate out to second-degree*
  - *then third-degree*
  - *and so on*
- continue to check each level of connections
  - *until we find nearest match for given search query*

graphs - breadth-first search - precedence

- if we consider this radiated search of connections
  - *also see how nodes may be checked as they're added to search list*
- search nodes for first-degree connections before any second-degree connections
- *breadth-first* may be used to find a path from one node to another
  - *and the shortest path as well...*
- possible because we define a search with a precedence of insertion
- i.e. search nodes in same order they were added

graphs - breadth-first search - queues

- to help search with an order of precendence
  - *use a queue data structure*
  - *ensures check of nodes in order added*
- as with a stack
  - *may not access random elements in a queue*
- particularly useful as it enforces two operations we may use
  - *enqueue*
  - *dequeue*
- if we *enqueue* node `A`, then node `B`
  - *node `A` will be dequeued before node `B`*
- queue data structure follows a pattern of *first in, first out*
  - *FIFO*
- use this type of data structure to query list of family members
  - *and their connections as well*
  - *query using breadth-first search*

graphs - Java - part 4

Graphs - Java - Breadth-first Search - UP TO 3:54

Source - Graphs - Java - YouTube

Algorithms and Data Structures

graphs - implement a graph - part 1

- initially consider options for implementing a graph with Python
- graph is a series of nodes with various connections to neighbouring nodes
- e.g. represent a relationship such as

```
cairo -> giza
```

- implement this type of relationship in code
  - *e.g. consider a hash table*
- *hash table* allows us to map a *key* to a *value*
- in current example
  - *need to match a node to all of its neighbours*

graphs - implement a graph - part 2

- initially implement this structure in Python

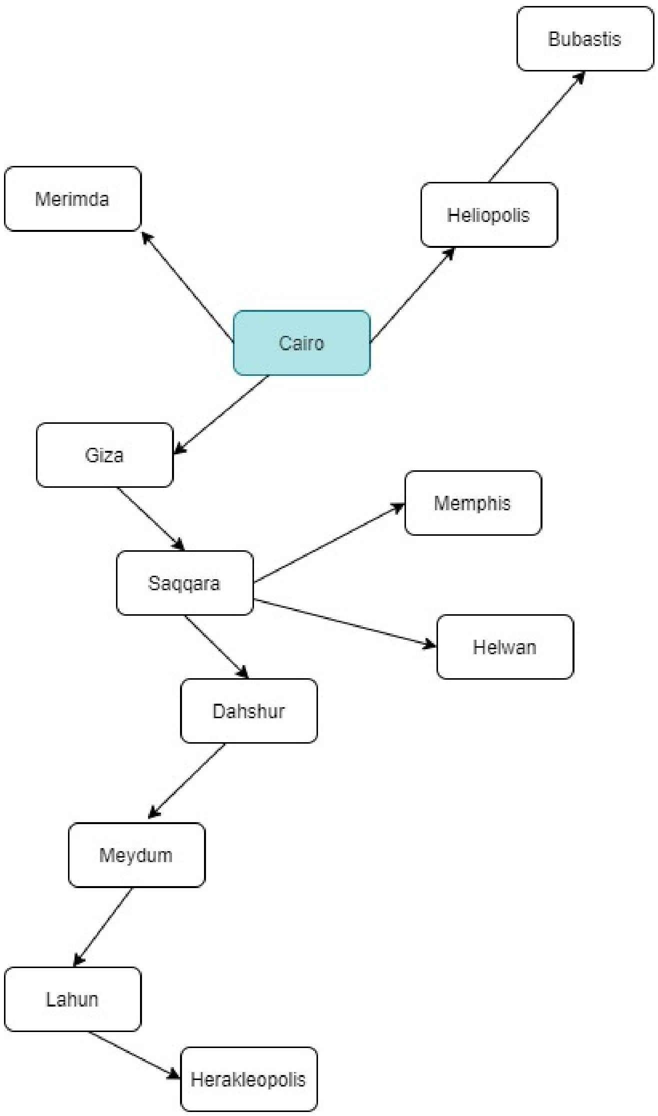
```
graph = {}
graph["cairo"] = ["giza", "merimda", "heliopolis"]
```

- *map* defined neighbouring nodes to an array for node `cairo`
- all we need for our graph in Python is a representation of its nodes and edges

Image - Graphs

implement a graph - part 1

- e.g. if we consider a larger graph



Graph - Sites in Lower Egypt

graphs - implement a graph - part 3

- implement this graph using Python

```
graph = {}
graph["cairo"] = ["giza", "merimda", "heliopolis"]
graph["heliopolis"] = ["bubastis"]
graph["giza"] = ["saqqara"]
graph["saqqara"] = ["memphis", "dahshur", "helwan"]
graph["dahshur"] = ["meydum"]
graph["meydum"] = ["lahun"]
graph["lahun"] = ["herakleopolis"]
graph["merimda"] = []
graph["bubastis"] = []
graph["memphis"] = []
graph["helwan"] = []
graph["herakleopolis"] = []
```

- simple example of graphs in Python - https://www.python.org/doc/essays/graphs/
  - *n.b. not optimal but shows graph creation...*

Algorithms and Data Structures

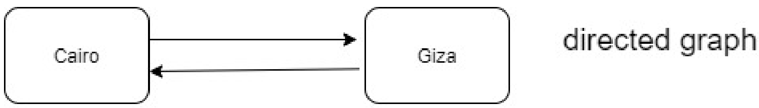
graphs - implement a graph - part 4

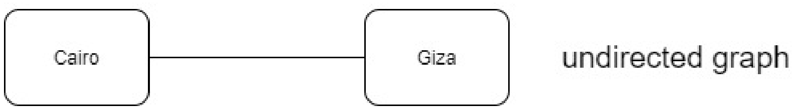
- compare diagram of graph and coded example with Python
  - *may consider whether insert order matters*
- if we consider underlying data structure - a hash table
  - *don't need to worry about order of insertion for defined key/value pairs*
- also, some nodes do not have any defined neighbours in this graph
- current example is known as a *directed graph*
  - *reflects one-way relationships for nodes and neighbours*
- in the current example
  - *Saqqara is neighbour of Giza*
  - *but Giza is not a neighbour of Saqqara*
- shown in diagram as a single directed arrow
- *undirected graph*, by contrast, defines both nodes as neighbours
  - *does not use directed arrows in example diagrams*

Image - Graphs

implement a graph - part 2

- represent such connections in both a directed and undirected graph
- e.g.





Directed & Undirected graph

- directed graph - both nodes are represented as neighbours
- undirected graph - default usage, both nodes are neighbours

Resources

various

- Python patterns - implementing graphs
- SHA algorithms - Wikipedia

videos

- Graphs - Java - YouTube
- Hash algorithms and security - YouTube
- Hash tables - Java - YouTube
- Hash tables - real-world usage - YouTube
- Joy of Data - YouTube
- SHA: Secure Hashing Algorithm - YouTube
- TED - What is the Internet? - YouTube
- What’s a cache for? - YouTube
- What is DNS? - YouTube