WEEK 1

WHY GRAPHS?

① Check if network is connected: example (Kevin Bacon number movie network)

② driving directions in Google Maps

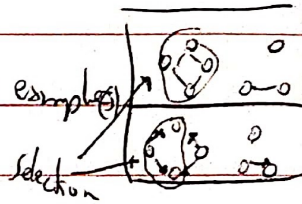
③ formulate a plan when only certain changes can be applied to a system

④ Compute the "pieces" (or "components") of a graph.

- clustering or structure of a certain graph.

GENERIC GRAPH SEARCH

Goals ①: find everything findable from a given start vertex: example



② Don't explore anything twice Goal: $O(m+n)$

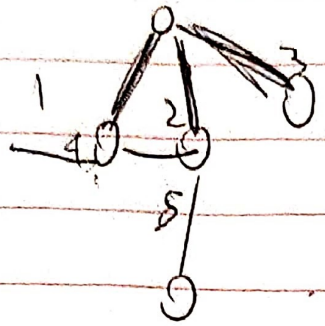
Generic algorithm (given graph G, vertex s)

- initially only s has been explored
- while possible:
  - choose edge (u,v) with u explored and v unexplored, (if none, halt)
  - mark v explored (explored.append(v))

Claim: at the end of the algorithm v explored $\Leftrightarrow$ G has a path from s to v

Note: how to choose between frontier edges?

BFS



etc, explore all nodes some of distance before exploring their ch

DFS plunges as deep as it can. Backtracks when necessary

BFS: O(m+n) time using a Queue (F.F.O)
to compute shortest paths
compute connected components of an undirected graph

DFS:
- Compute topological ordering of directed acyclic graph
- Compute "connected components" in directed graphs

Breadth First Search

THE CODE

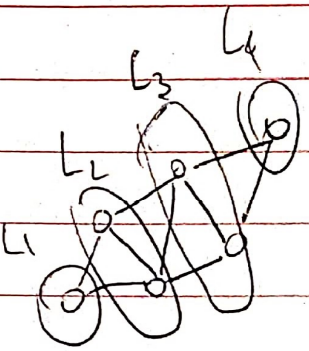
BFS (graph G, start vertex s)

explored = [ ] # all nodes initially unexplored

explored.append(s) #mark s as explored

Let Q = queue data structure (FIFO), initialized with s.

- while Q ≠ ∅:
  - remove the first node of Q, call it v
  - for each edge (v, w)
    - if w unexplored
      - mark w as explored
      - add w to Q



APPLICATION OF BFS: SHORTEST PATHS

Goal: compute dist(v), the fewest # of edges on a path from s to v

Extra code (to add to BFS-2 code):
- initialize $dist(v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$
- when considering edge (v, w):
  - if w unexplored, then set dist(w) = dist(v) + 1

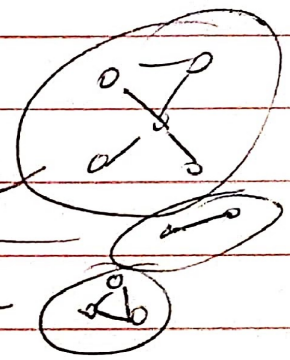
WEEK 1 ✓

APPLICATION OF BFS: UNDIRECTED CONNECTIVITY

Directed/undirected connectivities are very different.

Let $G = (V, E)$ be an undirected graph.

Connected components = the "pieces" of G



Formal definition:
Equivalence classes of the relation $v \sim w \Leftrightarrow \exists$ v-w path in G

Goal: compute all connected components

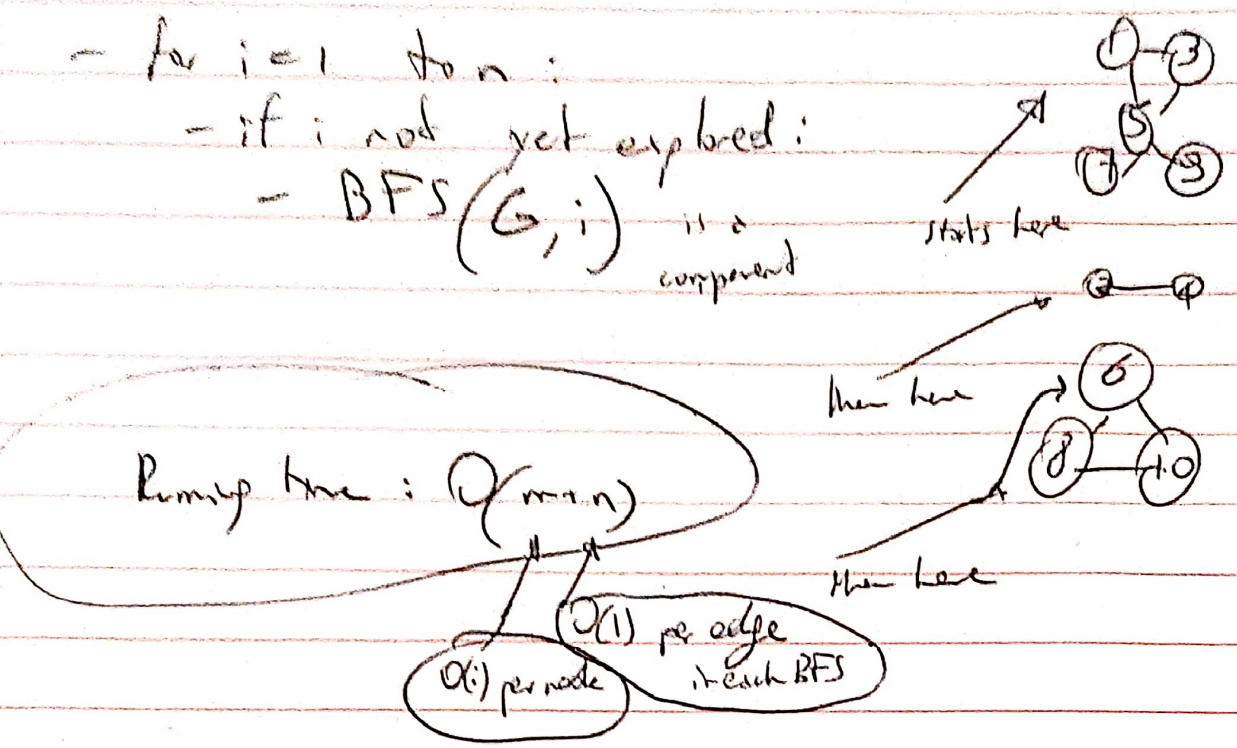
Why?:
- Check if network is disconnected.
- Display graph?
- clustering

CONNECTED COMPONENTS VIA BFS

To compute all components (undirected case):

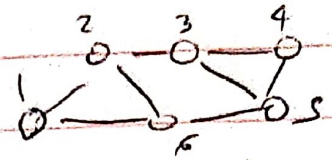
- all nodes unexplored

- for i = 1 to n:
  - if i not yet explored:
    - BFS(G, i) // i a component



APPLICATION OF DFS: UNDIRECTED CONNECTIVITY

DFS



Backtrack when necessary (by the last somewhere we've already been

- computes a topological ordering of a directed acyclic graph
- and strongly connected components of directed graphs.

Run time is $O(m+n)$

THE CODE

DFS, (LIFO) instead of (FIFO)

DFS(graph G, start vertex s):
  Lifo = []
- mark s as explored
- for every edge from Lifo (s, v):
  - if v unexplored, add to Lifo

Running Time is $O(ns + ms)$

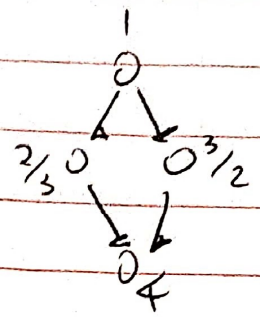
ns: # of nodes reachable from s

ms: # of edges reachable from s

FINALLY, WHAT THE HELL IS A TOPOLOGICAL ORDERING (of a directed acyclic graph)

Definition: A topological ordering of a directed graph G is a labelling F of G's nodes s.t.:

1. The $F(v)$'s are the set $\{1, 2, \ldots, |n|\}$
2. $\forall (v, w) \in G \Rightarrow F(v) < F(w)$



Motivation: Sequence tasks while respecting all precedence constraints

(COND1) DO: You need the graph to be acyclic !!!!!

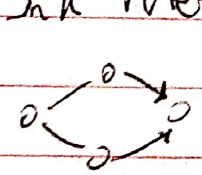
Or you're screwed too!

Theorem: (IF COND1) I can compute TOrd in linear time.

STRAIGHTFORWARD SOLUTION

Note: every directed acyclic graph has at least one sink vertex

(Duh, imagine 0 → 0 then induction)



To compute topological ordering:

The only candidates for last position are sink vertices

- let $f(v) = n$
- recurse on $G - \{v\}$

TOPOLOGICAL SORT via DFS (Slick).

M1

DFS-Loop (graph G)
mark all nodes unexplored
current_label = n (to keep track of ordering)

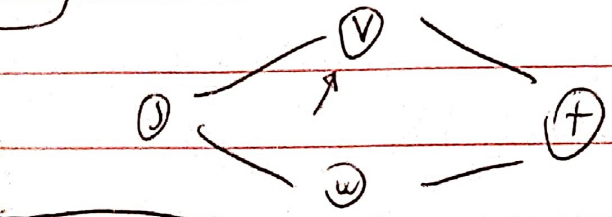
for each vertex $v \in G$
- if v not yet explored (in some previous DFS call)
  - DFS(G, v)

M2

DFS(graph G, start vertex s)
- mark s explored
- for every edge (s, v):
  - if v not yet explored
    - DFS(G, v)

Crucial bit: remember what you've explored already

M1)



v → t, sink so append t to s(t)
Back to v, sink so 3
Back to outer for loop, yay
Skip v because already explored!
Move to w

M2)

"	"
"	"
"	"
Check v	
:	

$O(n+m)$, correct.

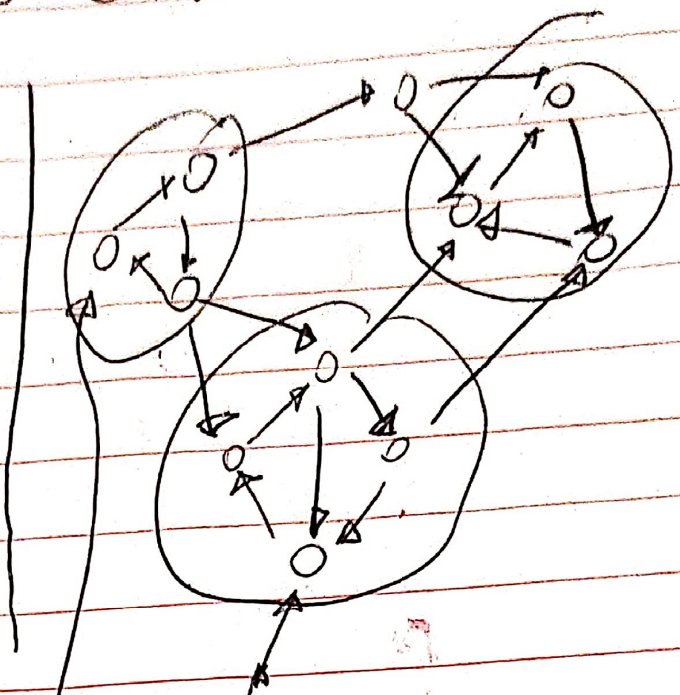
$a=1$

$b=1$

$d=$ calculate, $b^d=2$, case 1,

STRONGLY CONNECTED COMPONENTS in directed graphs

SCCs are cycles and nodes out of cycles



Why DFS?

If we start from ; won't just find SCC. Get a union of possible SCCs

If we start from , all graph!

Need to choose where we want to start.

KOSARAJU'S TWO PASS ALGORITHM

SCCs in $O(m+n)$ time

Algo:

① Reverse all arcs, let G^{rev} = G with all arcs reversed

② Run DFS on G^{rev} ← goal: compute "magical ordering" of nodes

③ Run DFS on G ← goal: discover the SCCs one by one

process nodes in decreasing order of finishing times from ②

DFS-Loop

DFS Loop (graph G)

global variable $t = 0$ (# of nodes finished processing so far)

[# of nodes processed so far]

global variable $s = NULL$ (current source vertex) [for leaders in 2nd pass]

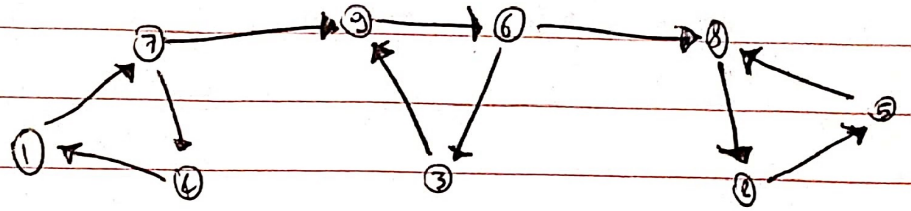
Assume nodes labeled with finishing times 1 to n

for $i = n$ down to 1:
  if i not yet explored:
    DFS (G, i)

DFS (Graph G, node i):

- mark i as explored
- set leader(i) = node s
- For each arc $(i, j) \in G$:
  - if j not yet explored:
    - DFS (G, j)
- $t++$
- set $f(i) = t$ (finishing time)

G^{rev}



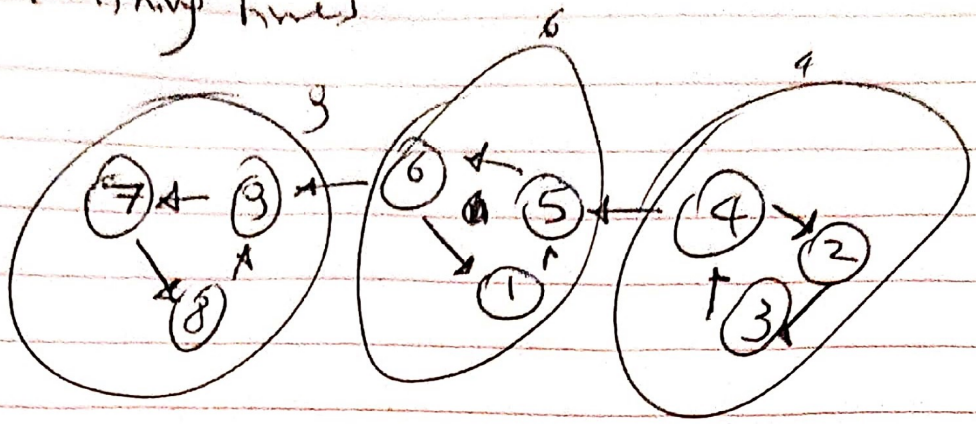
Starting from 9: 9, 6, 3, finish! $f(3) = 1$
Back to 6: 6, 8, 2, 5, finish! $f(5) = 2$
Back to 2: 2, finish! $f(2) = 3$
Back to 8: 8, finish! $f(8) = 4$
Back to 6: 6, finish! $f(6) = 5$
Back to 9: 9, finish! $f(9) = 6$

ALL of these have leader 9!

$f(n) = [7, 3, 1, 8, 2, 5, 9, 4, 6]$

Next [illegible] unexplored node: 7

All the DFS is executed, 9 nodes' names change to finishing times.



Run DFS again for leaders. top → bottom

from 9: 9, 7, 8, done!, leader 9

from ~~8~~, 7, 6: 6, 1, 5, done!, leader 6

from ~~8~~, 4: 4, 2, 3, done!, leader 4.

Running time: 2 × DFS = O(m+n)

WEEK 1 L8

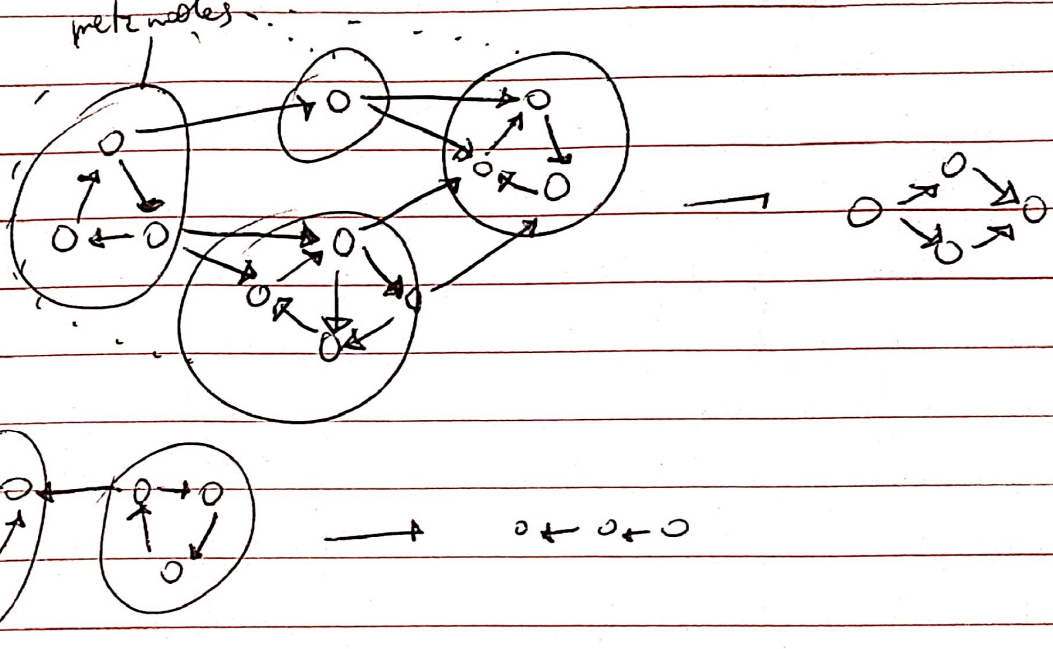
GENERAL ARGUMENT FOR LINEAR TIME

OBSERVATON

Claim: The SCCs of a directed graph induce an acyclic "meta-graph"

- meta ~~graph~~ nodes: the SCCs
- $\exists$ arc $C \to \hat{C} \Leftrightarrow \exists$ arc $(i,j) \in G$, with $i \in C$, $j \in \hat{C}$

Examples:



The meta-graph is always acyclic, any cycle of SCCs is just one SCC.

THE SCCs ~~ARE~~ the same IN G and G^{rev}

(and in the second pass of DFS in Kosaraju's algo too!)

Key lemma: | We'll prove this later

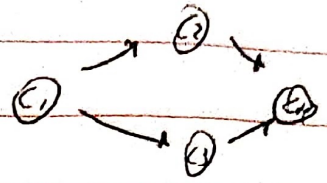
Consider the "sink" SCC in G

(cont)

...d pass of DFS (cop being somewhere in a sink SCC C^*

First call discovers C^* and nothing more.

After that, goes back up" to C_2 or C_3, etc etc.



Got it? Now let's prove the Lemma.

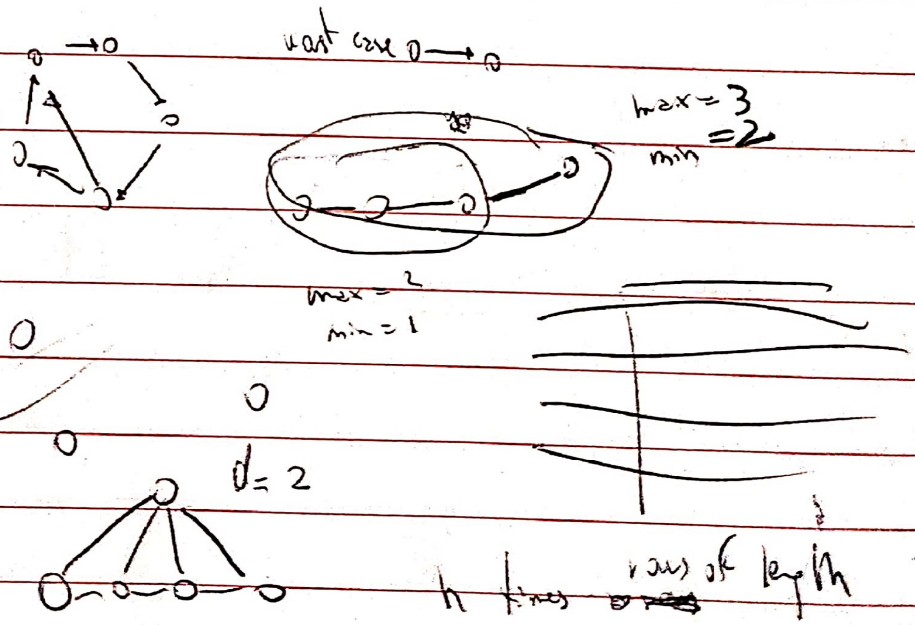
PROOF OF KEY LEMMA

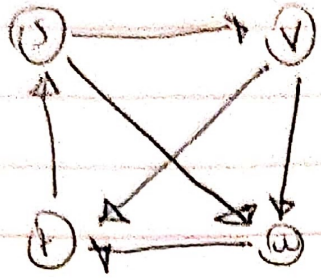
Case ($v \in C_1$): we'll see all of C_1 before C_2

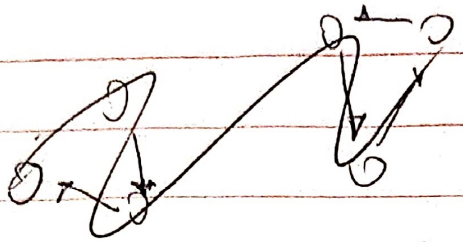
Case ($v \in C_2$): DFS (G^{rev}, v) won't finish until $C_1 \cup C_2$ completely explored

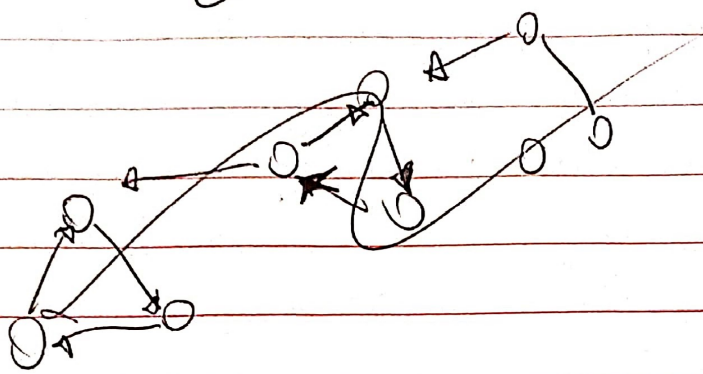
---

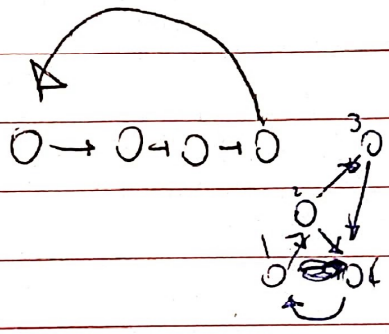
PROBABILITY



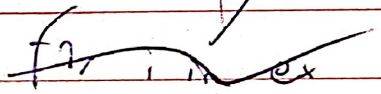








$$\left(1:[2],\ 2:[3,4],\ 3:[4],\ 4:[1]\right)$$



2

$$\left(2:\ \right)$$

Week 2 L1

Dijkstra's shortest path algorithm.

Input: directed graph $G_p = (V, E)$

- Each edge has non negative length
- source vertex s

Output: for each $v \in V$, compute

$L(v)$ = length of shortest s-v path

Assumptions:

1. [for convenience], consider strongly connected, $\exists$ s-v path for $\forall v$
2. $l_e \geq 0$, $\forall l_e \in E$ (edge length (for convenience))

WHY DO WE NEED ANOTHER SHORTEST PATH ALGO? CAN'T WE ALREADY USE BFS IN $O(n)$?

Yes. IF $l_e = 1$ for every edge.

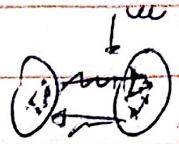
Question: why not replace each edge by multiple edges of length 1.

Stupid question

DIJKSTRA'S ALGORITHM

Initialize:

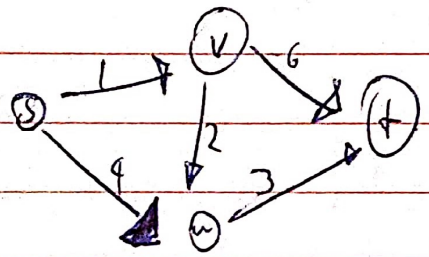
- $X = \{s\}$ #(vertices processed so far)
- $A[s] = 0$ #(computed shortest path distance)
- $B[s] =$ empty path (computed spot) // This is only for show not included in a real implementation



- while $X \neq V$:
  - among all edges $(v,w) \in E$ with $v \in X$, $w \in X$, pick the edge (v^*, w^*) that minimizes the following criterion:
    $A[v] + l_{vw}$ (Dijkstra's greedy criterion)
  - add w^* to X
  - set $A[w^*] = A[v^*] + l_{v^*w^*}$
  - set $B[w^*] = B[v^*] \cup [v^*, w^*]$?

WEEK 2 L2

EXAMPLE



First iteration: all edges leaving to outside X

Dijkstra's criterion: (s,v) has score of $0+1$, (s,w) $= 0+4$

So we go to s,v

Second iteration:

All edges leading to outside set X: (s,u), (v,w), (v,t)

Dijkstra's score for each: $(0+4)$, $(1+2)$, $(1+6)$

We then add w to X, etc etc ($A[w] = 3+2 = 5$)

Third iteration

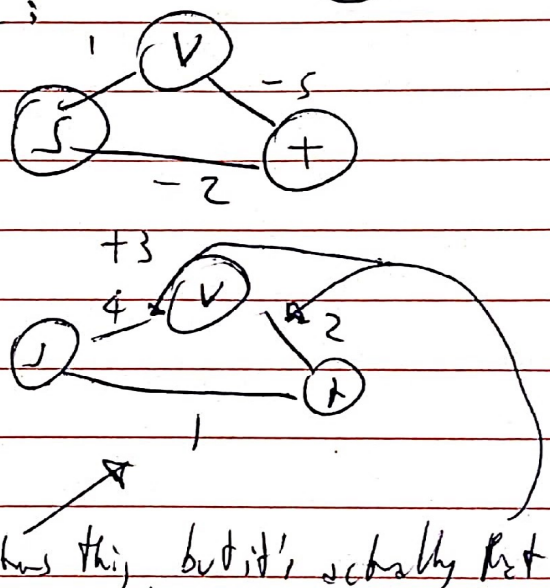
All edges: ~~to 23~~ (v,t), (w,t) greedy $(1+6)$ $(3+3)$

Bring in t,

$A(t) = 3+3 = 6$

~~Breaks~~ The while loop now breaks due to $X = V$

Can't reduce ~~negative~~ length to non negative by adding a large number do all lengths because:



relaxes this, but it's actually fine

WEEK 2 C3

WHY IS DIJKSTRA'S ALGO CORRECT

Proof by induction:

Base case: $A[1] = L[1] = 0$ correct ✓

Then on and on. Seems pretty simple.

Induction hypothesis: all previous iterations correct.

→ In current iteration, we set $B[v^*] = B[v] \cup (v^*, w$
computed length

DIJKSTRA AND RUNNING TIME

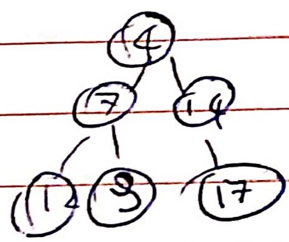
Naively appears to be $O(mn)$ (roughly)

HEAP OPERATIONS

Raison d'être of heap = perform Insert, Extract-min in $O(\log n)$ time

[rest of video assumes familiarity with heaps]

- Conceptually, a perfectly balanced binary tree
- heap property: at every node, key ≤ children's keys



- Extract-min by plucking off the root, bubble down
- Insert via bubbling down

Also: will need ability to delete from middle of heap.

Because you maintain heap as pbbt, height $\approx \log_2 n$

All operations run in $O(\log(n))$ time.

TWO INVARIANTS

elements in heap = vertices of V-X

#1

For $v \notin X$

key(v) = smallest Dijkstra's greedy

#2

knockout tournament.

two rounds: first each vertex runs a local round to find smallest edge from X pointing to it, put them in the heap

Then min(heap) finds best out of those local winners.

We'll be tracking the min on a sliding platter, kind of like #3

to maintain #2

that $\forall v \notin X$, key[v] = smallest greedy score of all edges (v,v), $v \in X$

- When w is extracted from heap (added to X):
  - for each edge $(w,v) \in E$:
    - if $v \in V - X$
      - delete from heap
      - recompute key[v] = min(key[v], A[w] + l_{wv}]
      - reinsert v into heap (maybe it slides down)

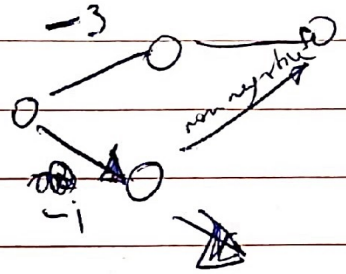
Running Time is $O(\log n)$ for each heap operation

- $(n-1)$ extract mins
- each edge (v,w) results in at most one Delete/insert combo if v added to X first.

So # of heap operations is $O(2m) = O(m)$

Problem

✓
✓
X
X



Can't go back, so yeah. Can add positive constant to all ~~path~~ edges

~~Dijkstra~~

Insert is usually $\log n$, but could be n

Will always terminate.

Opposite for priority queue

with negative sh# mixed results

Largest elem $O(n)$

Median some

Exp time.

The smallest $\log(n)$

Week 3 - 21

DATA STRUCTURES

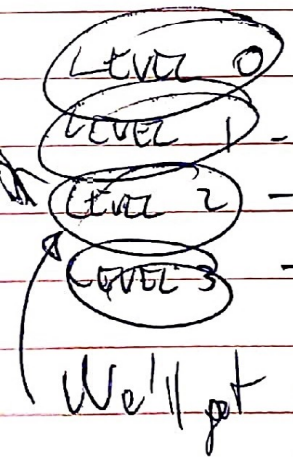
Point: organize data so it can be accessed quickly and usefully

Examples: lists, stacks, queues, heaps, search trees, hashtables, bloom filters, union find etc.

Why so many?
They all support different sets of operations with different times.
So they are suitable for different kinds of tasks.

Rule of Thumb: choose the "minimal" data structure that supports all the operations that you need.

TAKING IT TO THE NEXT LEVEL



- What's a data structure?
- Cocktail party knowledge, head of stuff, can't use in tech interviews
- "This problem calls for a heap"
- "I only use data structures I wrote myself"

Week 3 L2

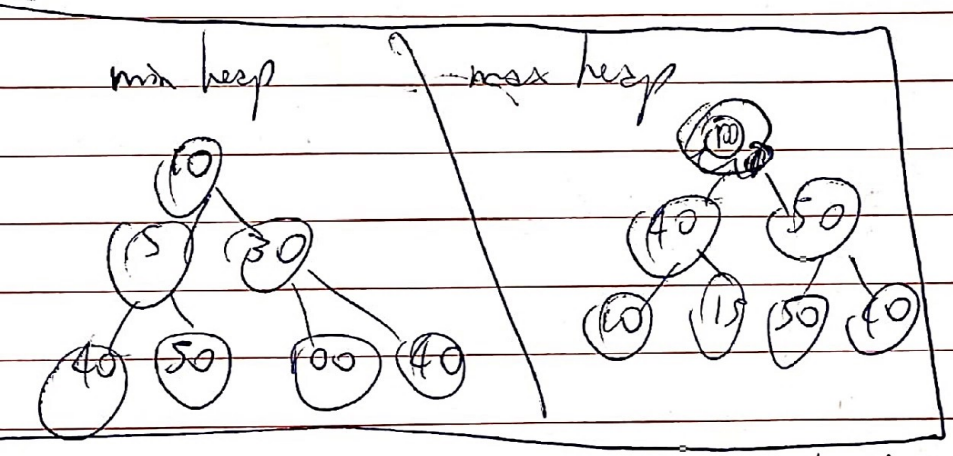
Heap HEAP: SUPPORTED OPERATIONS

- a container for objects that have keys
- employee records, network edges, events etc

INSERT: add object to a heap Runtime $O(\log n)$

EXTRACT-MIN (or MAX): remove an object with a minimum key value (ties broken arbitrarily) Runtime $O(\log n)$

quite good contents, 6/2



MAKEHEAPIFY: (n batched inserts in $O(n)$ time).

You don't need to create an empty heap and put n objects in. n times $\log n$ time needed for insertion $O(n \log n)$, We can do it in a slick $O(n)$ way

DELETE an arbitrary element from middle of a heap in $O(\log n)$

~~Coolest uses of heaps~~

CANONICAL USE OF HEAP

Canonical use: fast way to do repeated minimum computations.

Example: Selection Sort = $\Theta(n)$ linear scans, $\Theta(n)$ time on array of length n

Heap Sort:

① Insert all n array elements into a heap
② Extract-Min to pluck out elements in sorted order

Running time = $2n$ heap operations = $O(n \log n)$ time

Can we do better? NO. $n \log n$ is lower bound for any sort

ANOTHER APPLICATION: EVENT MANAGER.

"Priority queue" — synonym for a heap

Imagine videogame simulation:

- objects = event records, event-triggering events added with time stamp key
- Extract-min → next scheduled event.

APPLICATION: MEDIAN MAINTENANCE

I give you an increasing amount of numbers, one by one.
I want the median each time step:
Need to do it in $\log n$ (i) time

WEEK 3 L4

BALANCED SEARCH TREES

~~BA~~ Sorted Arrays: supported operations

3	6	10	11	17	23	32	36

OPERATIONS	RUNNING TIME
SEARCH (bin search)	$O(\log n)$
SELECT in statistic	$O(1)$
MIN / MAX	$O(1)$
PREDECESSOR / SUCCESSOR	$O(1)$
RANK (rank of 23 is 6)	$O(\log n)$
OUTPUT IN SORTED ORDER	$O(1)$ n times, $O(n)$
INSERTION	$O(n)$
DELETION	$O(n)$

We want insertion and deletion

That's a lot for insertion and deletion unless you barely ever do them

That's why we make:

BALANCED SEARCH TREES (BSTs)

Recipe of the: sorted array + fast (logarithmic) inserts and deletes

OPERATIONS	RUNNING TIME
SEARCH	$O(\log n)$
SELECT	$O(\log n)$
MIN / MAX	$O(\log n)$
PRED / SUCC	$O(\log n)$
RANK	$O(\log n)$
OUTPUT IN SORTED	$O(n)$

Up from $O(1)$

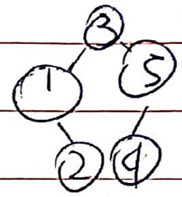
SAME FOR HEAP of MIN or MAX $O(1)$ elements

WEEK 3 LS

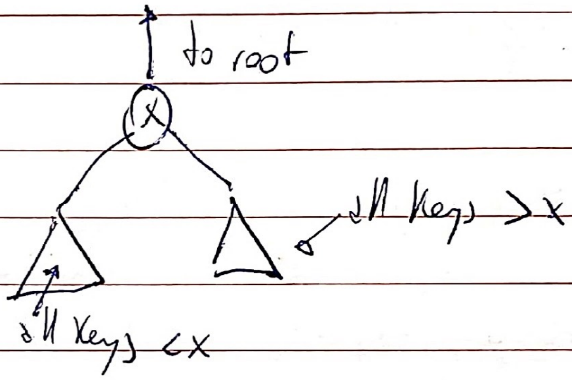
Raison d'être : like sorted array + list logarithmic

BSTs

- exactly one node per key



- most basic version:
each node has:
  - left child pointer
  - right child pointer
  - parent pointer

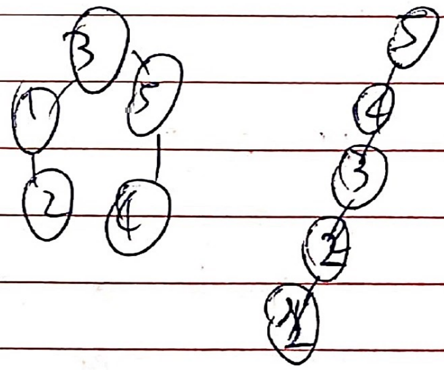


HEIGHT OF BST

best case, perfectly balanced; worst case: chain

Roughly $\log_2(n)$ usually, up to n

Many possible trees for a set of keys:



SEARCH FOR KEY IN TREE T

- start at the root
  - traverse left/right child pointers as needed
  - return node with key k or NULL as appropriate

TO INSERT

- search for k (unsuccessfully)
- put new node with key k at the end of pointer

Week 5

Search or insert operations in a binary tree take time at most Θ(height)

MIN, MAX, PRED AND SUCC

To compute min: start at root, you follow left pointer until the end Θ(height) again

PREDECESSOR OF KEY K

- easy case: if left subtree to K, non-empty, return K's max in left subtree
- otherwise: if left subtree empty, follow parent pointers until you get to a key less than K

PRINT KEYS IN INCREASING ORDER

- Let r = root of search tree with subtrees T_L and T_R
- recurse on T_L
- print r
- recurse on T_r

$O(1)$ time per call, n recursive calls. $O(n)$

DELETE A KEY FROM A SEARCH TREE

- SEARCH FOR KEY

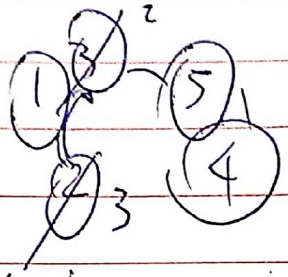
Then: three cases

(easy) no children. Just delete

(medium) splice out's K's node. Its one child takes its p

DIFFICULT CASE (K's node has 2 children)

- Compute K's predecessor L until you get to top, then swap top



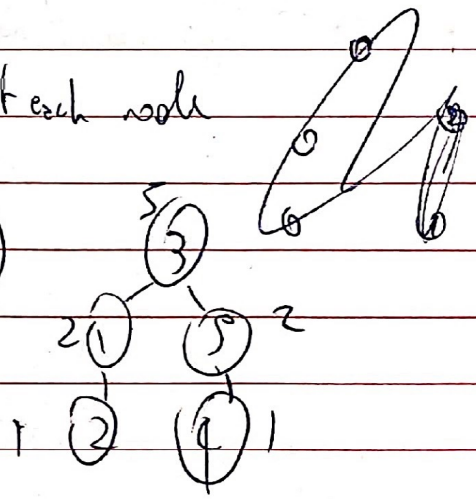
Then delete 3.

The running time $\Theta(\text{height})$

SELECT AND RANK

Idea: Store a little bit of extra info at each tree node.
about the tree itself

Example: store # of nodes in subtree rooted at each node



Note: if x has children y and z, $(\text{size}(y) + \text{size}(z) + 1)$

Also: Easy to keep sizes up to date in $\Theta(\text{height})$

TO SELECT:

- Start at root x, with children y and z
- let $a = \text{size}(y)$ [$= 0$ if x has no left child]
- if $a = i-1$ return x's key
- if $a \geq i$ recursively compute i^{th} on root y
- if $a < i$ recurse $(i-a-1)^{th}$ on z

WEEK 3 L7

Idea: Ensure height always $\log(n)$

Example: Red-Black Trees

RED BLACK INVARIANTS

1. Each node red or black
2. Root is black
3. No 2 reds in a row
4. Every root-NULL path has the same number of black nodes

~~EXAMPLE~~

NO CHAIN CAN BE AN RB TREE

HEIGHT GUARANTEE:

$$\text{height} \leq 2\log_2(n+1)$$

WEEK 4 L1

HASH TABLES

Purpose: insertible random access, through keys

Insert: add new record

Delete: delete existing record

Lookup: check for a particular record

} Using a key

ONLY THESE OPERATIONS, BUT GUARANTEED $O(1)$ TIME (if implemented well)

(1) Properly implemented
(2) Non pathological

APPLICATION: DE-DUPLICATION

Input: a "stream" of objects

Goal: Ignore duplicates

hashtable ~~Hash~~ lookup.
If not found, insert x into H

APPLICATION: 2-SUM PROBLEM

Input: Unsorted array of n integers. T, target sum t

Goal: Determine whether two numbers in Array x + y = t

Naive solution: $O(n^2)$ $(n(n-1))$

<2SUM PROB CNT'D

Even better: ① Insert elements of A in H. $O(n)$
② for each x, look in H for t-x $O(n)$

FURTHER OBVIOUS APPLICATIONS

- historical application: symbol tables in compilers
- blocking network traffic
- Search algorithms (eg game tree exploration), can use to avoid exploring more than once

WEEK 4 L2

HASH TABLE IMPLEMENTATION

High Level rules:

Setup: Universe U (generally, really big)

Goal: Maintain evolving set $S \subseteq U$ (of reasonable size)

Naive Solution ①: Array based solution indexed by n ($O(1)$ ops, $O(n)$ space)

Naive Solution ②: List based solution ($O(1)$ space, $O(S)$ lookup)

Solution:

① Pick n = # of "buckets" to store data in with $n \geq |S|$

② Choose a hash function $h: U \to \{0, 1, 2, \ldots, n-1\}$ (spits out position in array) (for simplicity we'll assume S doesn't vary too much)

③ Use Array A of length n, store x in $A[h(x)]$

What happens when there are collisions? $h(\text{"Bob"}) = h(\text{"Alice"})$

Think of birthdays! 23 people!

RESOLVING COLLISIONS:

Collision: $h(x) = h(y)$ with $x, y \in U$.

Solution #1: Separate chaining. Keep linked list in bucket, perform operations on list

Solution #2: Open Addressing (one object per bucket) h now specifies probe sequence

Double hashing: use buckets use hash hits themselves, takes lots of time!

Linear probing, or

WEEK 4 C3

HASH TABLES IMPLEMENTATION DETAILS
WHAT IS A GOOD HASH FUNCTION?

Note: in hash table with chaining, insert $O(1)$ most of the time

Performance depends on the choice of hash function!

Properties of a good hash function:

① Should spread data out well (good standard: completely random)
② Should be simple enough (not take too much time) (must be $O(1)$) to calculate

EXAMPLES OF BAD HASH FUNCTIONS

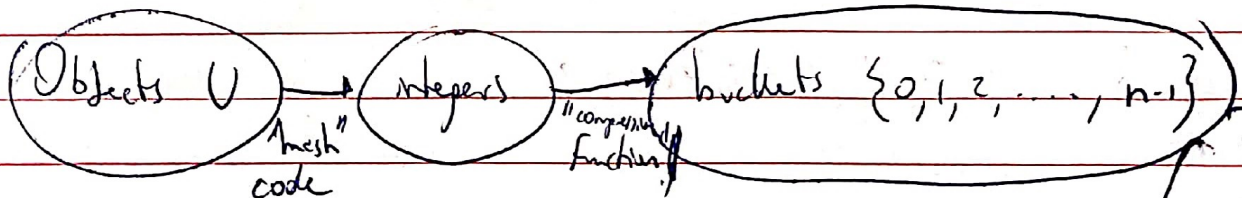
Ex: Keys = phone numbers $|U| = 10^{10}$

terrible: $h(x)$ = 1st 3 digits of x $|n| = 10^3$
mediocre: $h(x)$ = last 3 digits of x $|n| = 10^3$

Example: keys = memory locations (power of 2)

bad hash func = $h(x) = \text{mod } 1000\,(x)$ (all valid tickets are empty!)

QUICK AND DIRTY HASH FUNCTIONS



Very commonly used approach

- n should be
  - not too close to a power of 2
  - prime
  - not too close to a power of 10

Typical hash code naturally sends keys to integers

compression function eg mod(n) or mod n() (n = number of buckets)

① n should be

WEEK 9 L4

THE LOAD OF A HASHTABLE

Definition: The load factor of a hash table is:

$$\alpha = \frac{\#\text{ of objects in buckets}}{\#\text{ of buckets}}$$

Conditions for good performance:

1. $\alpha = O(1)$, else won't run in constant time. (NEED TO BE ABLE TO COMPUTE α in $O(1)$)
2. For open addressing, $\alpha < 1$ (must be)

For good performance, we need to control load

Also, THE PERFECT HASH FUNCTION DOES NOT EXIST. FOR EVERY $H(+)$ THERE IS A PATHOLOGICAL DATA SET

PATHOLOGICAL DATA IN THE REAL WORLD

Main Point: Can analyze several real-world systems by exploiting badly designed hash functions.

If the hash function is open-source and simple, anyone can read it and reverse engineer it.

SOLUTIONS

1. Use a cryptographic hash function which can't be reverse engineered
2. Use randomization: design a family H of hash functions and pick one at random

WEEK 4 L8

BLOOM FILTERS

Advancement on hashtables, save space, have errors

PROS	CONS
es space	① Can't store associated object ② No deletions ③ Small false positive probability

~~Original~~ APPLICATIONS

Original: early spell checkers

Genomics: List of forbidden passwords

Modern: Network routers
- limited memory, need to be super fast still

WHAT IS A BLOOM FILTER?

Ingredients:

① an array of n bits ; $\frac{n}{|S|}$ = # of bits per object in data set
② K hash functions $h_1, h_2, \ldots, h_K$ K = small constant
③ for $i = 1, 2, \ldots, K$, $A[h_i(x)] = 1$, (whether or not it was 1 already)

Lookup(x) return TRUE if $A[h_i(x)] = 1$ for every $i = 1, 2, \ldots, K$

There are false positives, but no false negatives and it saves

WEEK 4 L9

HEURISTIC ANALYSIS OF BLOOM FILTERS

There is a trade off between % of false positives and space saved.

After inserting dataset S, $1-\left(1-\frac{1}{n}\right)^{K|S|}$ probability for a given bit to have been set to 1

prob $\in 1-e^{\frac{-K}{b}}$ $b=\frac{n}{|S|}$, number of bits per object

This means the false positive prob $\emptyset$ is bounded by $\left(1-e^{-K/b}\right)^K$

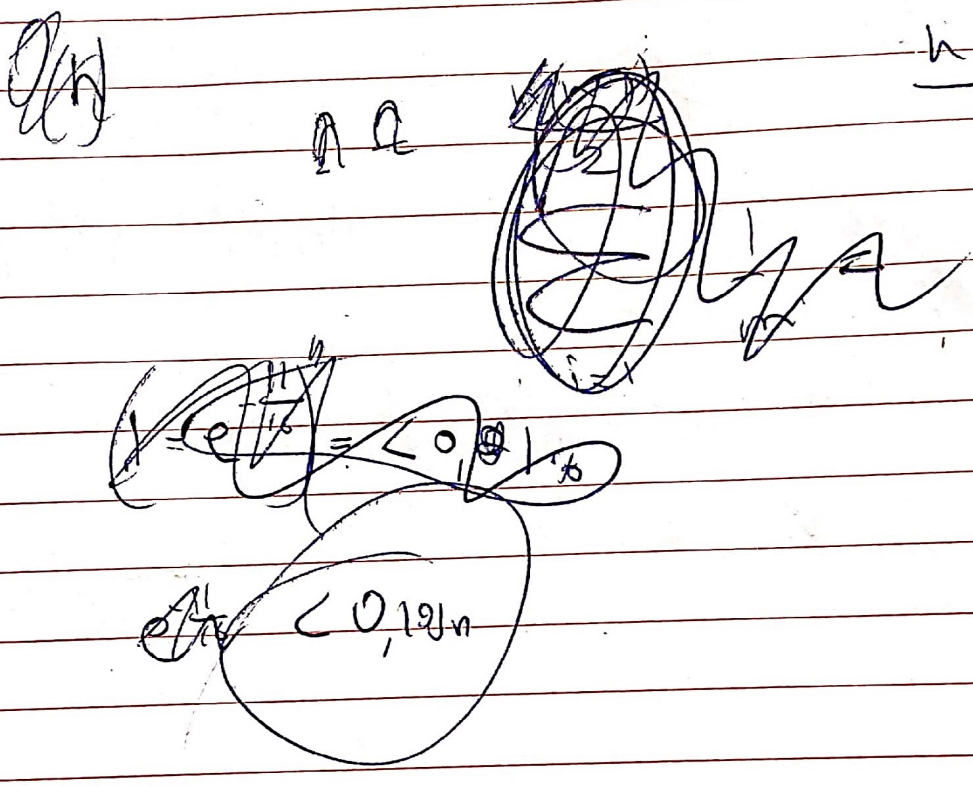
how to find K? For fixed b, the false positive probability is minimised by setting $K \simeq (\ln 2)\cdot b$

Example: $b=8$, $K=5$ or 6, error prob $\simeq 2\%$

Problem Sinking

h(n keys) → [illegible]

n/m ... like with the n tasks for m servers



$\frac{n(n-1)}{2} \cdot \frac{1}{m} =$