CS 410/584, Algorithm Design & Analysis: Lecture Notes 3

Amortized Cost

- Not an algorithm design mechanism per se
- For analyzing cost of algorithms
  - especially for a sequence of operations
  - when worst case is not typical of average case

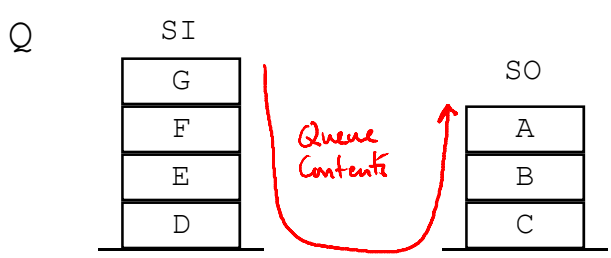
Example: Exercise 17.3-6

Implement a queue with two stacks

Why? less blocking on concurrent ops

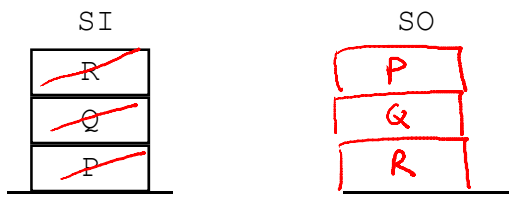
Queue Operations

```
enqueue(x, Q)
  push(x, SI)
dequeue(Q)
  return( top(SO) )
  pop(SO)
```



A Complication

```
dequeue(Q)
  if empty(SO) then
    while not empty(SI) do
      push(top(SI), SO)
      pop(SI)
  return(top(SO))
  pop(SO)
```



Complexity

- operation `enqueue` is always $O(1)$
- in a sequence of `n` queue ops, one `dequeue` could be as bad as $O(n)$
- but total cost of sequence isn't so bad as $O(n^2)$

Consider: each element has $4

$4n suffices
- pay $1 for pushing it on SI
- pay $2 for popping from SI and pushing on SO
- pay $1 for popping from SO.

Claim: can do a sequence of n enqueues + dequeues in $O(n)$ time

Union-Find

Assumptions

1. Disjoint union (no duplicate removal)
2. Elements 1 .. n
3. Only UNION + FIND (Look-up)
4. Name each set by an elt. in set

Naming by an element is not a limitation

Usually want to know if FIND(i) = FIND(j)

<u>Critical Assumption</u>: the number of ops is proportional to the number of elements

First Hack

Array `R[1..n]`

`R[i]` ≡ name of set that contains elt. i

Initially, `R[i] = i`

`Union(i,j)`: Scan `R`. For every element `k` with `R[k] = i`, set R[k]←j

i	1	2	3	4	5
R[i]	1	~~2~~ 4 3	3	~~4~~ 3	5

UNION(2,4) UNION(4,3)

Each union costs ~~possibly~~ O(n)

n unions in $O(n^2)$

n FINDs in O(n)

Improving Efficiency

More efficiency
- find just the elements of the set i
- always choose a smaller set to iterate through

`R[i]` as before

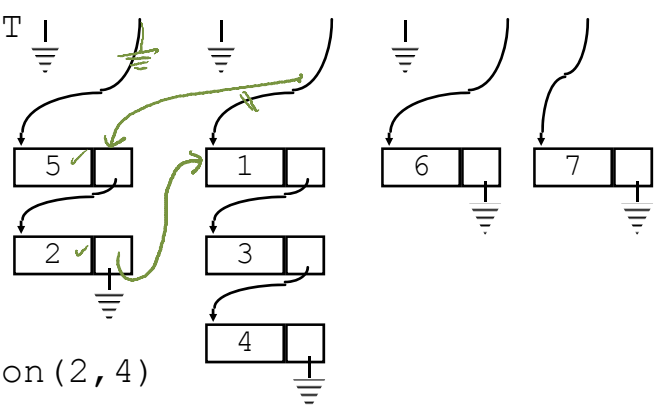
If `j` is a "name", then `LIST[j]` (array of linked lists) points to a linked list of elements in j's set

and `SIZE[j]` = # elts. in j's set

```
Union(i,j)   {assume i=R[i] , j=R[j] }
1. Assume SIZE[i] ≤ SIZE[j]
2. Traverse LIST[i] and change each element to set j
3. Add LIST[i] to LIST[j]
4. Adjust SIZE[j] ← SIZE[j] + SIZE[i]
   clear LIST, SIZE for i
```


Example

i	1	2	3	4	5	6	7
R[i]	4	~~2~~ 4	4	4	~~2~~ 4	6	7
SIZE	0	~~2~~ 0	0	~~3~~ 5	0	1	1
LIST							



Union(2,4)

Time Complexity

Theorem: Can execute `n-1 Union` operations in `O(n·log n)` time.

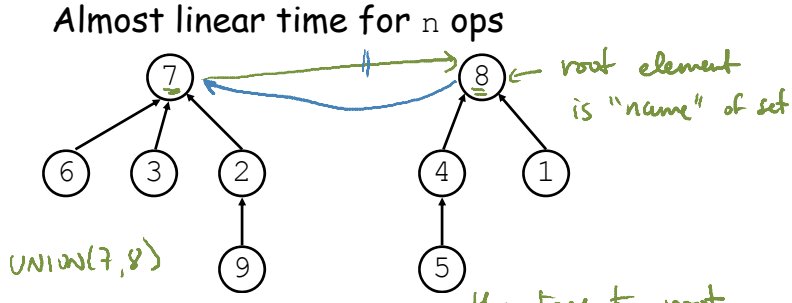
- Time for `Union` is proportional to the number k of elements moved
  size smaller set
- Charge the cost to the elements moved when they go from one set to another

- How many times may a single element be moved?
  Whenever it changes sets, it goes from a set of size k to a set of size ≥2k.
  Can only move lg n time

Tree-Structured Union-Find

Data structure: forest of trees, one per set

Almost linear time for `n` ops



UNION(7,8)

`Find(i)` - Traverse up the tree to root

`Union(i,j)` - Have i point to j

Balancing

Can get bad trees

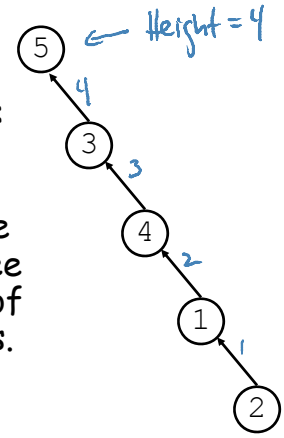
Keep trees somewhat balanced:
always merge smaller into larger

Lemma: If for every `Union`, the smaller tree is made a subtree of the larger, then any tree of height `h` has at least 2^h nodes.

longest path leaf → root

Proof: If `h = 0`, then the tree must be a single node.

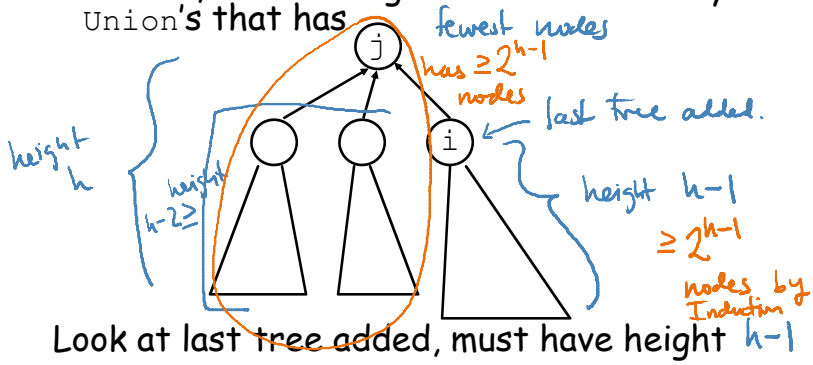
$2^0 = 1$ ✓



Suppose it's true for all heights $< h$

Proof, Continued

If `h > 0`, choose height `h` tree formed by `Union`'s that has fewest nodes



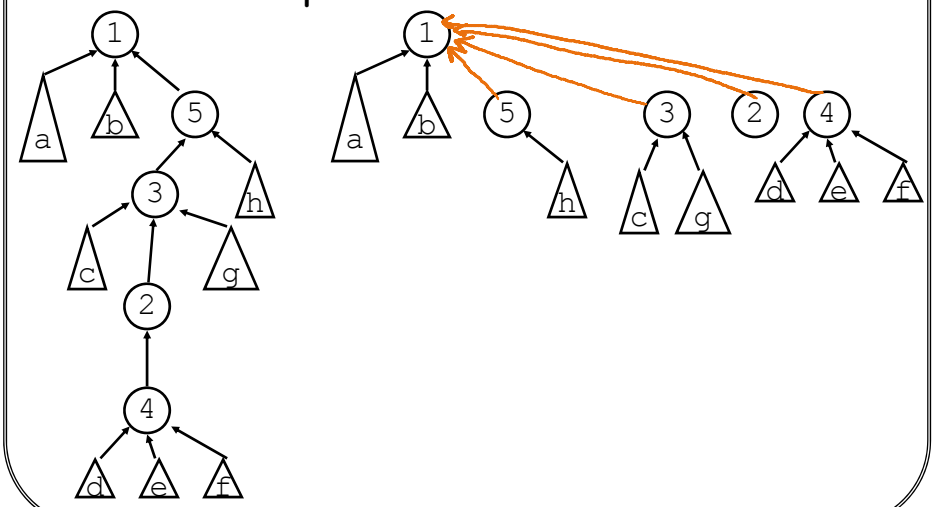
Look at last tree added, must have height $h-1$

Before the addition, T_j has height $\leq h-2$

Total nodes is $\geq 2^{h-1} + 2^{h-1} = 2^h$

Path Compression

Further improvement on `Find` time



Complexity

Need funny functions to prove complexity

`F(0)` = 1

`F(i)` = $2^{F(i-1)}$

i	0	1	2	3	4	5
F(i)	1	2	4	$2^4=16$	$2^{16}=65536$	2^{65536}

`G` is kind of an inverse

`G(n)` = the least `k` such that $F(k) \geq n$

$G(13) = 3$

In this universe, `G(n)` ≤ 5 for all practical `n`

Complexity 2

Show that a sequence σ of `c·n Union/Find`'s can be done in $O(n\,G(n))$

<u>Assume</u>: `Union` takes 1 "unit" of time

`Find` takes "units" proportional to the depth of the element found.

<u>Definition</u>: The `u-height` (union height) of an element relative to

1. Let σ′ be σ with FINDs removed (no path compression)
2. Construct a forest using σ′, call that F
3. `u-height(v)` is height of `v` in F

longest path (in edges) from a leaf to v

Complexity 3

Lemma: There are at most $\frac{n}{2^r}$ nodes of u-height r.

Proof: Node of `u-height = r` has at least 2^r descendents in forest `F`. (by last Theorem)

If `u-height(v) = u-height(w)`, sets of descendents are disjoint

There are at most $\frac{n}{2^r}$ distinct sets of 2^r elements, so at most that many nodes of u-height $n/2^r$

Corollary: no node has u-height greater than $\lg n$

Complexity 4

Lemma: If during execution of σ (not σ'), w is a proper descendent of v, then u-height(w) < u-height of v

Proof: w a proper descendent of v under σ implies
w is a proper descendent of v under σ'
by the UNION that made w a descendent of v in σ

Put nodes into <u>groups</u> by u-height

u-height = r goes in G(r)

u-height	0-1	2	3-4	5-16	17-65536	65537-2^{65536}	...
group	0	1	2	3	4	5	

Complexity 5

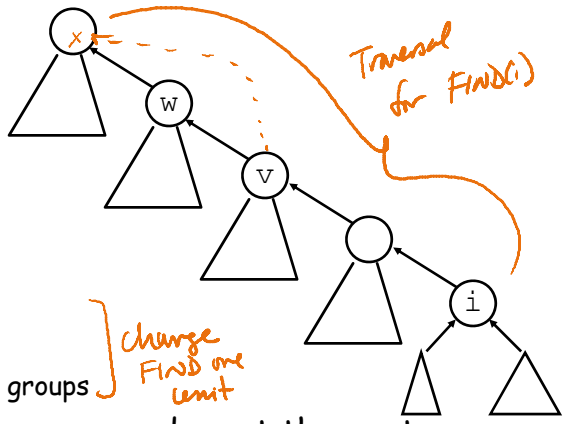
How to count cost of Find's in c·n operations.

Split cost of Find between
1. to the FIND itself
2. certain nodes used in doing the find

In the end have
sum over FIND
+
sum over nodes

Suppose Find(i) traverses k nodes
- How to split up k "units"?

Complexity 6



Consider each node `v` on path

1. If
   - `v` is root or
   - parent of v → `w` is root or
   - `v`, `w` in different groups

   charge FIND one unit

2. If `v`, `w` in same group and `w` not the root,

   charge v one unit.

Complexity 7

Along any upward path, `u-heights` increase

At most $G(n)$ different groups

so case 1 applies to at most $G(n)+1$ nodes per FIND

So a `Find` operation gets charged at most $O(G(n))$ units

Complexity 8

What happens when case 2 applies?

`v` gets a new parent `x`

and `u-height(x)` > `u-height(w)`

- How many times can `v` get a new parent before its parent is in a higher group, and case 1 applies?

At most $F(g) - F(g-1)$ times where g is u-height of v

Once case 1 applies to v, it always applies in the future.

Complexity 9

Charges to nodes:

Σ (max nodes in g)(max moves for node in g)

all groups g $N(g)$

$$N(g) \le \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r} \leftarrow \text{max nodes of u-height } r$$

$$\le \left(\frac{n}{2^{F(g-1)+1}}\right)\underbrace{\left[1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots\right]}_{\le 2}$$

$$\le \frac{2 \cdot n}{2^{F(g-1)+1}} = \frac{n}{2^{F(g-1)}} = n/F(g)$$

max moves for a node in a group $\le F(g)$

$$\le \sum_g n/F(g) \cdot F(g) = \sum_g n \in O(n\, G(n))$$

Graphs

An <u>undirected graph</u> G is a pair (N, E)

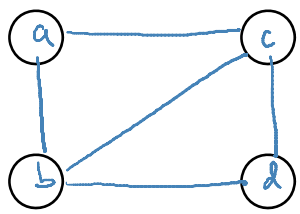
N finite set of nodes

E a collections of doubleton sets called edges

Example

N = {a, b, c, d}

E = {{a,b},{b,c},{a,c},{b,d},{c,d}}

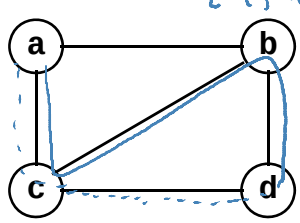


allowed ?

Paths

A <u>path</u> in graph G = (N,E) $n_i \in N$

A sequence $n_1, n_2, \ldots, n_k$ $k \geq 2$ of nodes such that $\{n_i, n_{i+1}\} \in E$ $1 \leq i < k$



a c b d ← simple

a c b d c a

A path is <u>simple</u> if no duplicate nodes

~~A cycle looks like a path~~

A path is a <u>cycle</u> if $\{n_k, n_1\} \in E$ and $k \geq 3$

can be viewed a – c – d – b

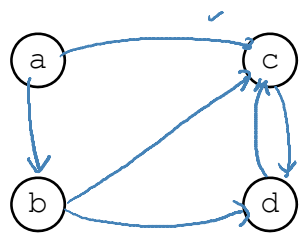
a c d not a cycle

Directed Graphs

In a directed graph (digraph) E is
collection of ordered pairs of nodes

Example

E = {(a,b), (b,c), (a,c), (b,d), (c,d), (d,c)}



(e,e)

a b c d path
c a b d not

Representation

$O(|N|^2)$ in a dense graph

Time and space complexity depends on how we capture the graph.

Edge list – works if there no isolated node

$O(|E|)$ {a,b}, {b,c}, {a,c}, {b,d}, {c,d}

Adjacency List

$O(|N|)$

a → b, c
b → a, c, d
c → a, b, d
d → b, c

Representation 2

Adjacency matrix

O(1)

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Undirected case – symmetric

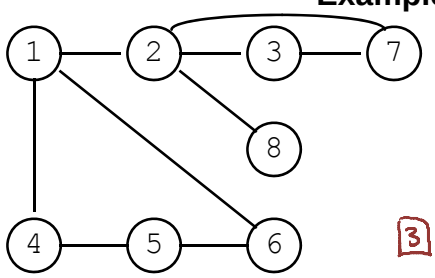
conventional to put 1's on diagonal for an undirected graph

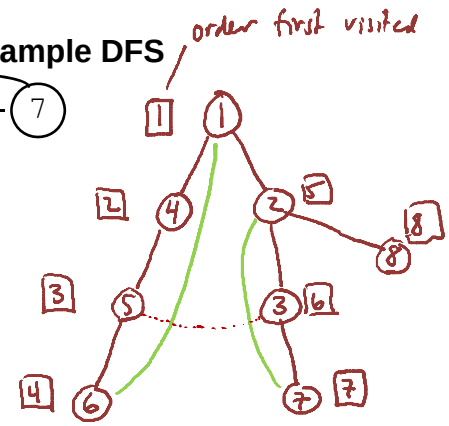
Depth-First Search–Undirected Graphs

A means to visit all nodes of an undirected graph

```
at node x
  pick edge {x,y}
  if y visited, pick another
   edge
  else recursively visit y
if no more edges, we're done
```

Example DFS





DFS Algorithm

```
DFS(v)
  mark v
  for each edge {v,w} do
    if w not marked then
      save ({v,w})
      DFS(w)
```

Once for each connected component

"saved" edges form the <u>depth-first spanning tree</u>

T = tree edges

B = E − T back edges

DFS number order in which nodes are initially visited

DFNum[v]

Time Complexity

Theorem: `DFS` requires `O(max(n,e))` steps on a graph with `n` nodes and `e` edges (given as an adjacency list).

Time spent looking for unmarked nodes (to start search from) is $O(n)$

`DFS(v)` is called once for each node.

time spent, apart from recursive calls, is proportional to # of nodes adjacent to v

$$\sum_{v \in N} (\#\text{nodes adjacent to } v) = 2e$$

Back Edges

If $\{v,w\} \in B$, then v is an ancestor of w in the spanning forest (or vice versa).

If v an ancestor of w in DFS tree, then `DFNum[v]` $<$ `DFNum[w]`.