CS 584/410 Spring 2011 HW3 key

3A a. The "payment" argument can be extended to the requeue case. Give each element \$4 whenever it enters the queue, either by an enqueue or a requeue. Then every element can ~~b~~ pay \$1 for every stack op. it is involved in. For an enqueued element, that is at most a push + pop on SI and ~~a~~ push + pop on SO, as before. For a requeued element, it is at most a push and pop on SO. We give out no more than \$4n for n operations, and that suffices to pay for all the stack ops, so we are still $O(n)$.

b. Consider the following sequence of n operations

enqueue(1)
enqueue(2)
⋮
enqueue($\frac{n}{2}$)
dequeue
unqueue
dequeue
unqueue
⋮
dequeue
unqueue

$n/4$ pairs

Each dequeue or unqueue ~~removes~~ moves all remaining pairs ~~from~~ between SI and SO. So we have a sequence of

$$\frac{n}{2} + \left(\frac{n}{2}-1\right) + \left(\frac{n}{2}-2\right) + \dots + 1$$

stack ops. This is approximately ~~$O(n^8)$~~ $n^2/8$ operations, which is not in $O(n)$.

3B. The data can simply be kept in a linear list (such as an array). Insert adds an element to the end of the list, which is $O(1)$. Delete-Top-Half ~~to~~ runs linear median finding to get the middle element and delete everything greater or equal to that element. It ~~that~~ takes $O(k)$ time if the list currently has k elements.

Example:

If S is

4 8 12 3 1 9 15 20

Then Insert(6, S) results in

4 8 12 3 1 9 15 20 6

Delete-Top-Half(S)

Finds Median(S) = 8

then makes a pass to remove everything ≥ 8 to get

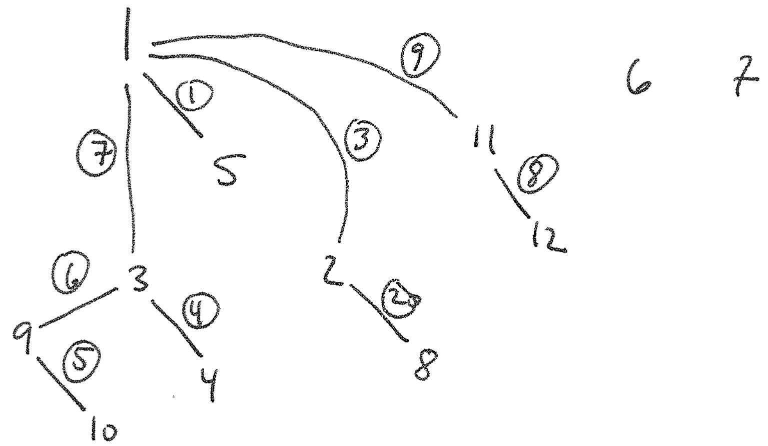
4 3 1 6

Complexity: While one Delete-Top-Half can be as bad as $O(m)$, that case can't happen repeatedly.

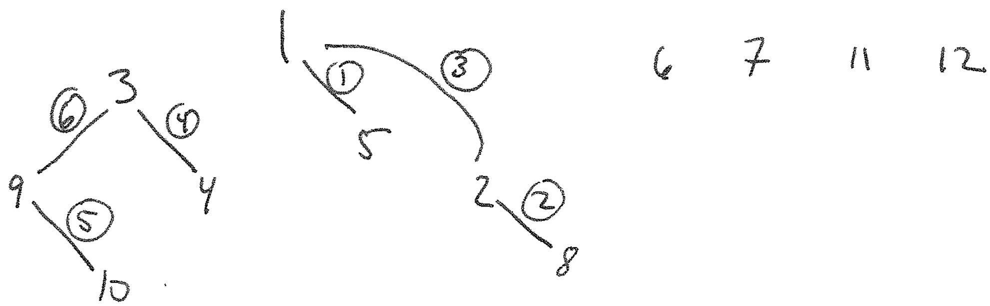
A payment argument: Give each element \$3 initially. Charge \$1 for the insert. So every element in the list has \$2 left. Whenever we do a ~~linear~~ Delete-Top-Half, we charge each element \$1. Thus if there are k elements at that point, we collect \$k to ~~h~~ pay for the $O(k)$ cost. Also, we take the remaining $\$\lceil \frac{k}{2} \rceil$ from the deleted elements and give them to the ~~new~~ remaining elements. So they now have \$2 again. Thus, every element in the list always has \$2 after any operation.

Since we gave out at most \$3m for m operations, and that pays for all the costs, those operations take $O(m)$ time.

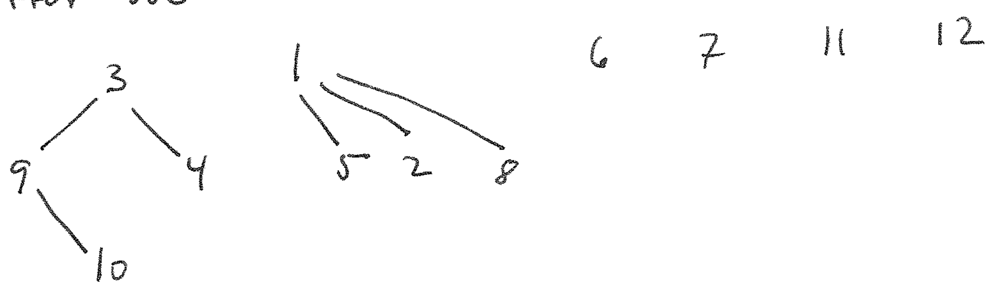
3C. The tree on the next page is the result w/o path compression. The circled #'s are the order that edges are created.



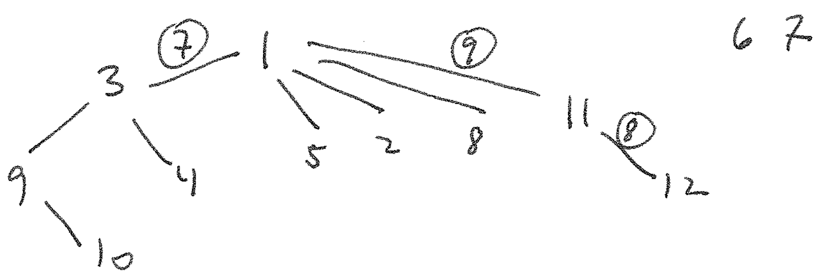
With path compression, just before Find(8) we have



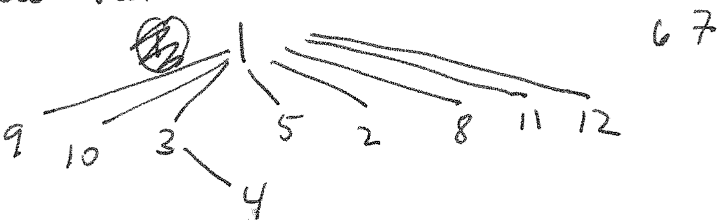
After we have



After the next 3 unions we have



After the last 2 finds we have



3D. Basically, we can perform a DFS-like process, except if we encounter a node for a second (or subsequent) time, we keep going. If we we ever run out of edges, and we aren't at the target node, we know some ~~node~~ node has in-edges > out-edges, and there is no Eulerian Circuit. [Note that if some node has out-edges > in-edges, it implies there is another node with ~~out~~ in-edges > out-edges, since the total in-edges and out-edges must be the same. So we only need the one check.]

The one complication is that we may get back to the start node, but with edges elsewhere unused. ~~Thus a~~ (See the example.) Thus as we return from the recursion, we see if there are further edges to explore, and don't output the node until just before returning. [The path will come out in reverse order.]

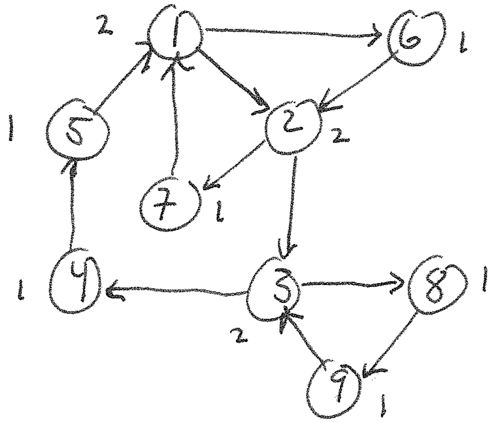
1. Calculate $IN(v)$ = in degree for every node v. Do this by traversing all the ~~adjan~~ adacency lists and incrementing $IN(w)$ ~~too~~ for each w we find on any list. If $IN(v) = 0$ for any node v, emit "fail" (can't reach v)
2. Pick ~~and~~ an initial ~~node~~ node u and do Euler(u).
3. Scan the nodes, and emit fail if there remains a node u where $IN(u) > 0$ (graph is not connected)

n = # of nodes
e = # of edges

```
Euler (v)
   while ADJ(v) not empty
      remove w from ADJ(v)
      IN(v) ← IN(v) - 1
      Euler (w)
   if IN(v) > 0 then "fail" else emit v
```

Complexity: Step 1 scans all nodes and all edges, so $O(n+e)$. In step 2, Euler is invoked at most once per edge, so $O(e)$. Step 3 scans the nodes, so $O(n)$. Thus $O(n+e) = O(\max(n,e))$ over-all.

Example:



small numbers are IN(v)

Here is the call sequence, starting at 1. Little number is IN(v) on entry.

- EULER(1) 2
  - EULER(2) 2
    - EULER(3) 2
      - EULER(4) 1
        - EULER(5) 1
          - EULER(1) 1
            - EULER(6) 1
              - EULER(2) 1
                - EULER(7) 1
                  - EULER(1) 0
      - EULER(8) 1
        - EULER(9) 1
          - EULER(3) 0

Emitting the nodes in order that the calls return gives us the path (in reverse order)

$1 \leftarrow 7 \leftarrow 2 \leftarrow 6 \leftarrow 1 \leftarrow 5 \leftarrow 4 \leftarrow 3$
$\leftarrow 9 \leftarrow 8 \leftarrow 3 \leftarrow 2 \leftarrow 1$

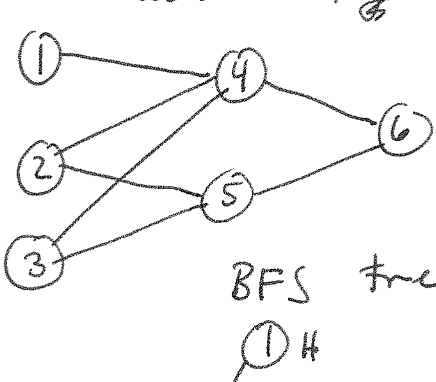
[To match our representation for paths, we should remove the final 1]

3E. The basis for this algorithm is Breadth First Search. We label nodes H (superhero) and V (supervillain) when first encountered. The label is always the opposite the label of the node on the other end of the edge by which we encounter the node. If we find an edge with the same label at both ends we fail. Otherwise, we have the desired labeling. Q is the breadth-first queue of nodes

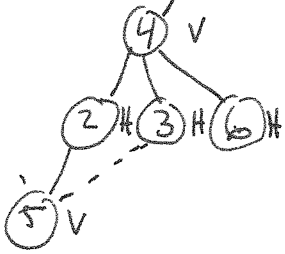
```
While there is an unlabeled node v
    label(v) ← H
    enqueue (v, Q)
    while non-empty (Q)
        w ← dequeue (Q)
        for each edge {w, u}
            if unlabeled(u) then
                label(u) ← opposite (label(w))
                enqueue (u, Q)
            else if label(u) = label(w) then "fail"
```

The complexity is that of BFS, which is $O(n+e)$

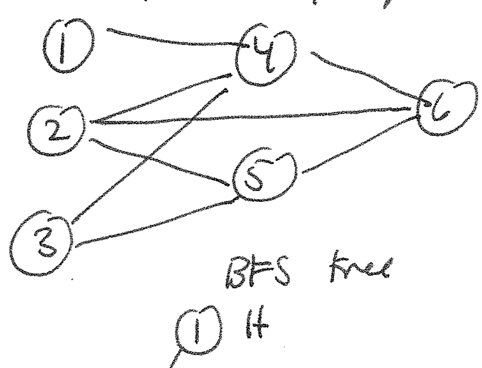
Success example



BFS tree



Failure example



BFS tree