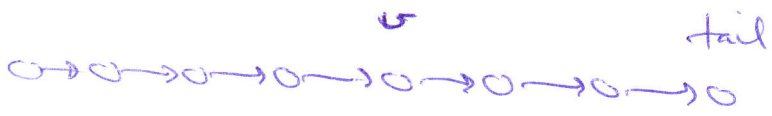
List Ranking

For singly linked list, let rank(v) be # of edges from v to tail of the list:



rank(v) = 3

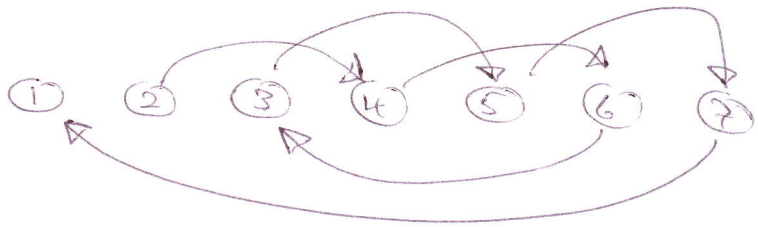
List ranking problem:

Input: The edges of a singly linked list (in any order), with each edge of the form:
(Node-ID, Successor Node-ID)

Plus the ID's of head and tail node.

Output: list of (node-ID, rank) pairs

Example:



Rank 0 6 3 5 2 4 1

Input: (5,7), (2,4), ... Output: (1, 0), (2, 6), (3, 3), ...

assuming node IDs are 1, 2, 3, ..., N,

Internally, the problem is rather easily solved in $O(N)$ time. [How?]

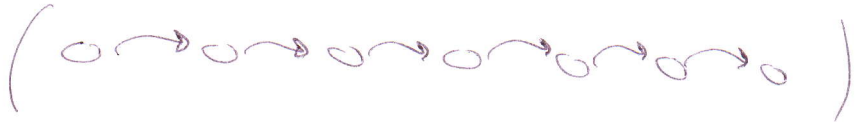
Externally, this simple algorithm takes $O(N)$ I/O's ($\gg$ Sort(N), normally).

Today: $O(\text{Sort}(N))$ external algorithm

Note: By sorting edges of list by source-ID [i.e., by first component] and sorting output by node-ID [also first component] we can in a synchronous scan transfer the ranks to the edges.

We can then sort the edges by rank, resulting in the edges being laid out in "list order":

(2, 4, 6) (4, 6, 5), (6, 3, 4), (3, 5, 3), (5, 7, 2), (7, 1, 1)



So list ranking gives a way of "traversing a list" (in any layout in memory) efficiently, that is, in $O(Sort(N))$ I/O's.

Idea 1: solve a more general problem:

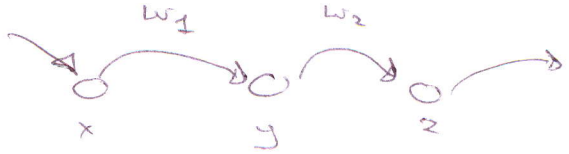
- edges have weights
- rank(v) = sum of weights on path to tail from v.

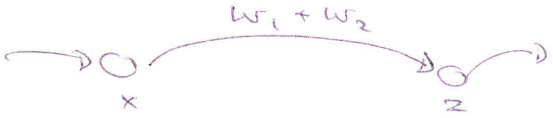
Idea 2: a) Find independent set I of nodes, i.e. $\forall x, y \in I$: (x, y) is not an edge (in other words, x, y are not neighbors in list). We require head, tail $\notin I$.

b) Remove I, solve recursively on smaller instance, then insert I again.

Details of b)

"Bridge out" of a node y : [Remove]



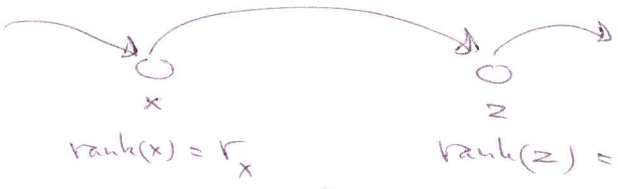


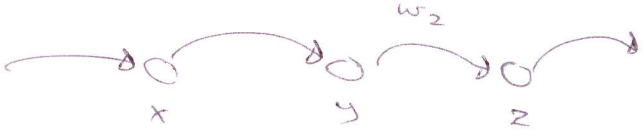
Note : y ≠ head, tail is necessary

Note : weighted rank of any remaining node is not changed

Note : We still have a list.

Having solved the smaller instance recursively [Stopping the recursion when |list| is $O(1)$], we can generate the correct rank for y ("bridge in" node y) :





rank(y) = $w_2 + r_z$

So from a solution to the smaller instance (without y), we can generate the solution to instance with y.

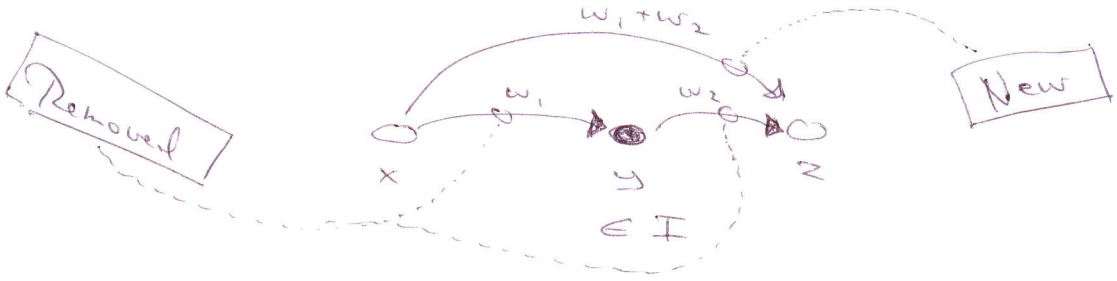
A crucial observation is that since I is an independent set, the bridge-out and bridge-in operations on nodes $y \in I$ do not overlap (they touch different edges).

So they can be done <u>simultaneously</u> in one batch without compromising correctness.

Here is how to do it in $O(\text{Sort } N)$ I/O's:

i) Create independent set I and:
   - A = edges sorted by source-ID
   - B = copy of edges sorted by destination-ID
   - C = I sorted by node-ID

ii) In a synchronous scan [generalized merge] of A, B, and C, create
   - D = new edges from the bridge-outs
   - E = list of edges removed by —"—



F = list of second edge $[(y, z, w_2)$ above$]$ of all pairs of removed edges.

iii) In a synchronous scan of A and E (sorted by source-ID), create $G = A \setminus E$ (i.e. list of all edges not removed)

iv) Recurse on the new list $D \cup G$, while storing F for later.

v) Recursion returns list $H = (z, \text{rank}(z)), (x, \text{rank}(x)), \ldots$ of ranks [of the list $D \cup G$]. Update this to list for A during a synchronous scan of H sorted on node-ID and F sorted on destination-ID: from $(z, \text{rank}(z))$ and (y, z, w_2) create and add $(y, \text{rank}(y))$

$\uparrow$

$\text{rank}(z) + w_2$ All new

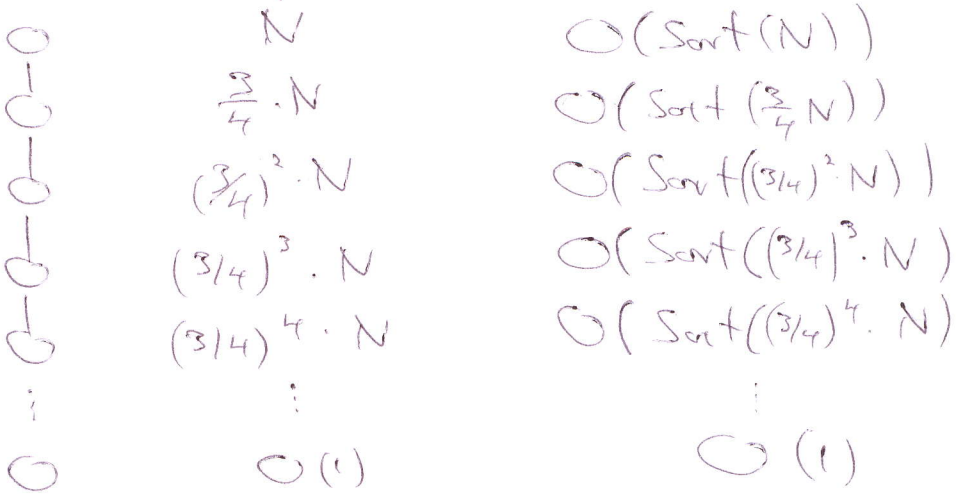
Besides finding I,

all actions above [not counting the recursion] are in total $O(1)$ sorts and scans, so takes $O(\text{Sort}(N))$ I/O's.

We still need to discuss how to find an independent set I (a new set for each level of the recursion).

Claim For $N \geq 24$, we can in $O(Sort(N))$ I/O's find an independent node-set I not containing head or tail, whose size $|I|$ is at least $N/4$.

From the claim, and the analysis above, the recursion tree of the algorithm is:



So total cost is bounded by

$$O\left(\sum_{i=0}^{\infty} Sort((3/4)^i \cdot N)\right) = O\left(\sum_{i=0}^{\infty} \frac{(3/4)^i \cdot N}{B} \cdot \log_{3/M}\left(\frac{(3/4)^i \cdot N}{M}\right)\right)$$

$$= O\left(\sum_{i=0}^{\infty} \frac{(3/4)^i \cdot N}{B} \cdot \log_{3/M}\left(\frac{N}{M}\right)\right) =$$

$$O\left(\frac{N}{B}\cdot \log_{B/M}(N/M)\cdot \underbrace{\sum_{i=0}^{\infty}(3/4)^i}_{\frac{1}{1-3/4}=\frac{1}{1/4}=4}\right)$$

$$= O(\text{Sort}(N)).$$

The total space used by the algorithm (the lists F saved for each level of the recursion) sums to N, since any given node y is only stored once in a F list. So this space is N tuples, or $O(N/B)$ blocks.

What remains is just to prove the claim, i.e. give the details of a):

Idea 3: create a 3-coloring of the nodes, i.e. color each node with one of three colors [here chosen to be Red, Blue, and Green] in such a way that for no edge, its two endpoints have the same color.

Given this, we choose I to be the

set of nodes having been colored with the most occuring color among the three colors (with head and tail removed, if necessary).

This means that $|I| \geq N/3 - 2$.

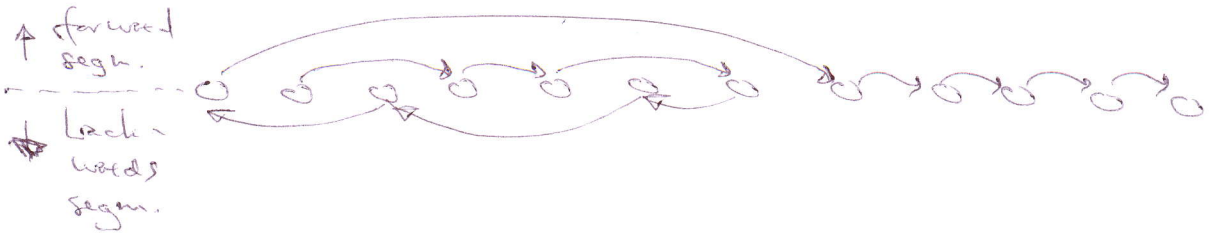
Since $N/3 - 2 \geq N/4$ for $N \geq 24$, this will give the claim.

To find 3-coloring:

Call an edge (x, y) forward if $y > x$

backward if $y < x$

Split the two types of edges into two files (in a scan). This gives a set of forward segments (subpaths) and backwards segments:



The goal is to color each forward sub path

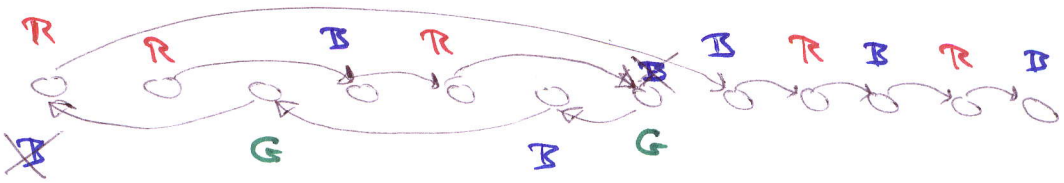
Red, Blue, Red, Blue, Red, ...

head of subpath

and each backward subpath

Green, Blue, Green, Blue, Green, Blue, ... (or vice versa)

Note that some nodes are both tail in a forward subpath and head in a backwards subpath! Their final color is the one they received as head.



It is not too hard to check that this is a 3-coloring.

To color the forward segments (backwards analogous):

```
Sort edges by source-ID
While scanning the edges:
    IF  Source-ID ≠ PQ.FindMin()   // Head is found
        Output(Source-ID, Red)
        PQ.Insert(Destination-ID, Blue)
    ELSE
        (Source-ID, color) = PQ.DeleteMin()
        PQ.Insert(Destination-ID, opposite color)
        Output(Source-ID, color)
(Source-ID, color) = PQ.DeleteMin()      } just to color last node
Output(Source-ID, color)                 }
```

Here, PQ is an I/O-efficient priority queue

It is easy to see that the following invariant is maintained:

PQ contains exactly the destination-IDs of the forward edges crossing the scan point, and their destination-ID is annotated with the correct color.

From this, correctness is clear.

Since operations on a PQ can be done in amortized $O(\frac{1}{B}\log_{M/B}(N/M))$ I/O's, the total I/O's for both forward and backward coloring is $O(Sort(N))$.

□