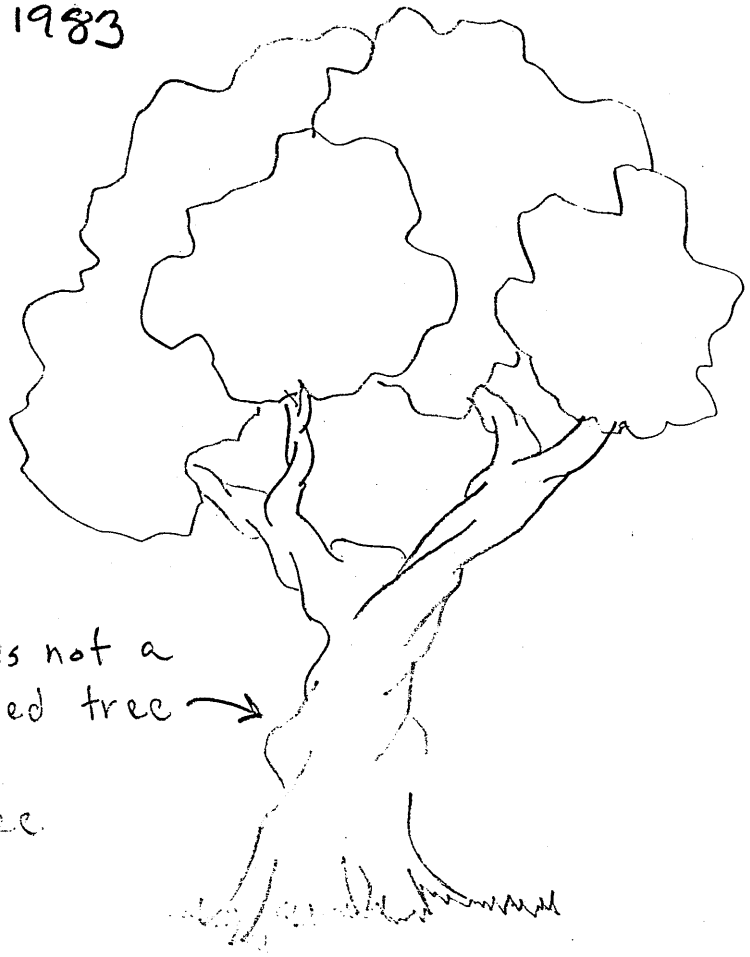
Brown Bagger by
D.W. Jones
May 5, 1983

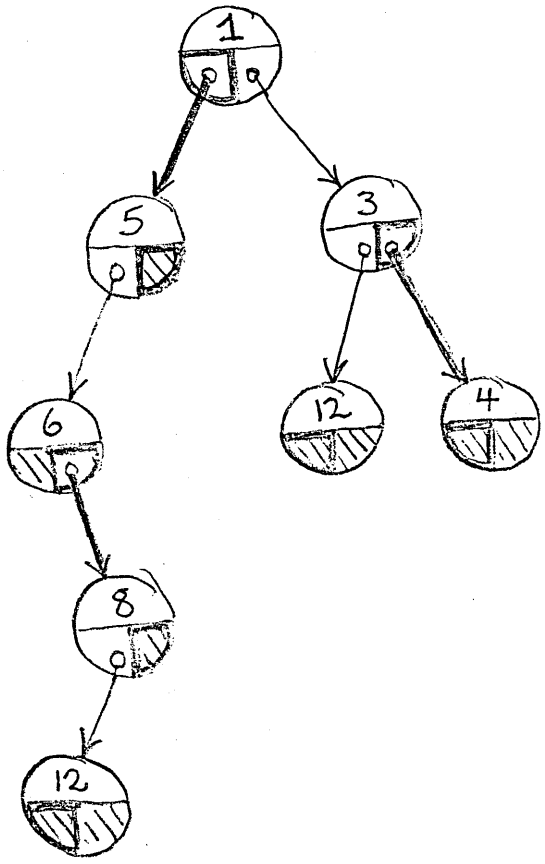
Twisted Trees

An implementation of priority queues



This is not a twisted tree →

This is a twisted tree.



A twisted tree is statically characterized by:

1) The heap invariant "the priority of each node is greater than the priority of that node's children" (higher priorities are represented by lower numbers)
2) Each node has 2 children, one of which is distinguished.

Twisted trees are dynamically characterized by an insertion rule and a deletion rule used to implement the enqueue and dequeue operations on priority queues:

1) Insertion:

The operation of inserting a new node in the tree involves a search down some path from the root to a leaf for the place that the new node can be inserted without disturbing the heap invariant. All nodes below that point will be demoted to make room for the new node.

Rule:

a) The path followed will be that which is made up only of distinguished children.

b) As each node on the path is visited, the roles of the distinguished and non-distinguished children are reversed. (ie: the node is twisted)

Insertion Examples

Starting with the (non vegitative) tree on page 1,

insert 

insert 

2) Deletion:

The operation of deleting a node from the root of the tree involves promoting the root of one of the subtrees to replace the deleted node. This recursive process ends up following a path from the root to some leaf.

Rule:

a) the highest priority child is promoted (any other choice would end up violating the heap invariant)

b) Along the promotion path, the distinguished child of each promoted node will be

i) the child below it on the promotion path. (bad)

ii) the child below it off of the promotion path. (seems good)

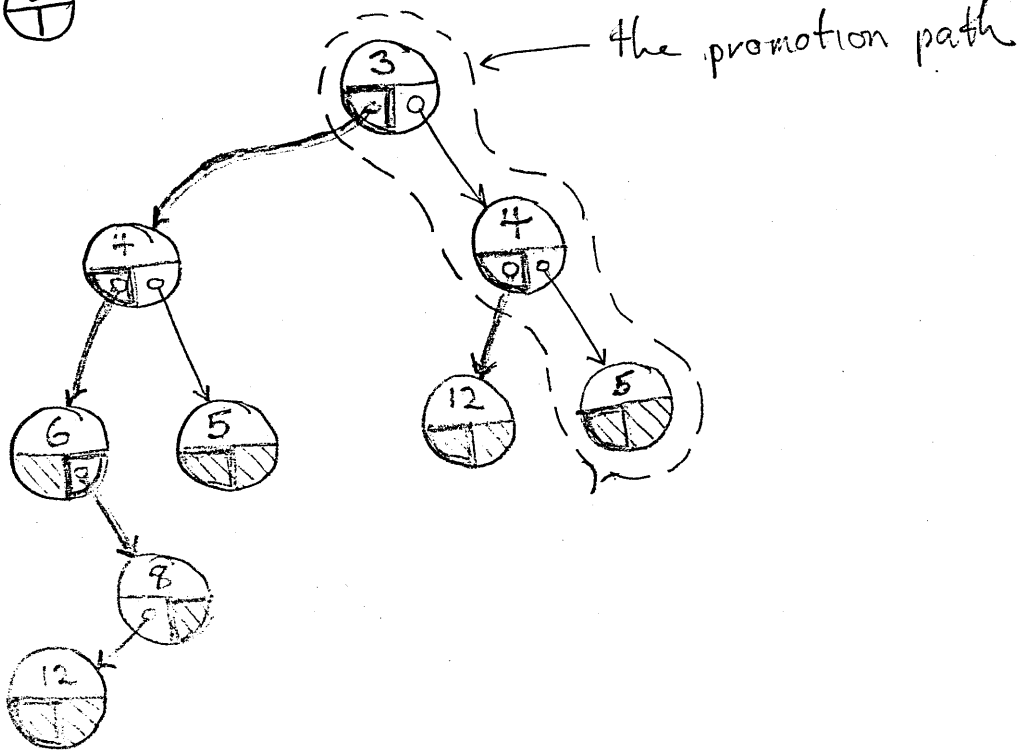
iii) determined by which child of the promoted node used to be distinguished. (not investigated — code is complex)

iv) determined by which child of the node the promoted node replaced used to be distinguished. (not investigated — code is complex)

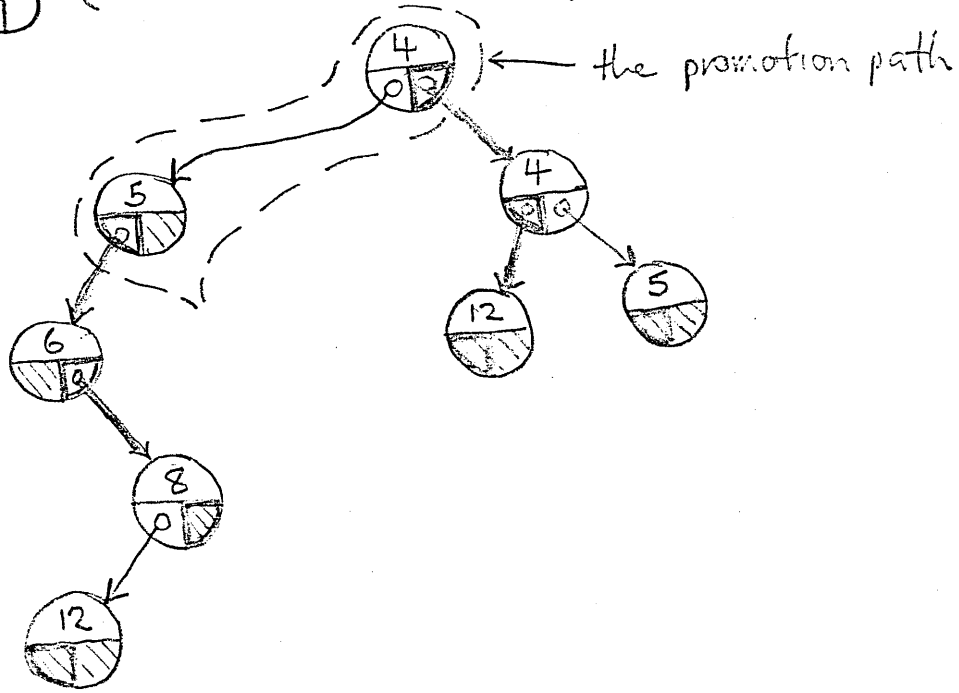
Deletion Examples (using alternative ii)

Starting with the tree at the bottom of page 3

delete (1)



delete (3) (break ties to the left)



Efficient implementation:

```
TYPE noderef = ↑node;
     node = RECORD
              left, right : noderef;
              priority : real;
            END
```

where the left son is the distinguished son, and the rotation operation is accomplished by exchanging the left and right sons.

CONJECTURE

The tree will usually stay balanced so the average performance will be $O(\log N)$ for insertion and deletion in a queue of length N

EMPIRICAL RESULTS

AVERAGE INSERTION TIME + DELETION TIME IS

$$2.6 \times 10^{-4} + 7.66 \times 10^{-5} \times \log_2(N)$$

(George Singer's measurement, my Pascal code on the Prime)

Problem:

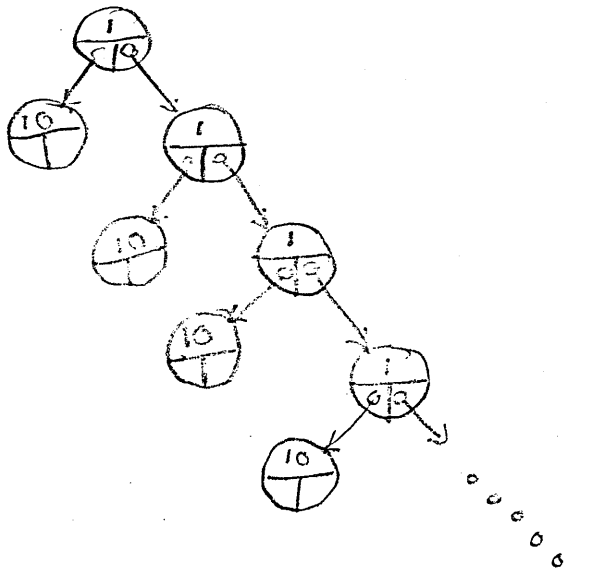
What is the worst case over a long sequence of insertions and deletions?

for a single insertion or deletion, the worst case is clearly O(N). Just construct a very unbalanced tree and let the insertion or deletion path follow the longest path.

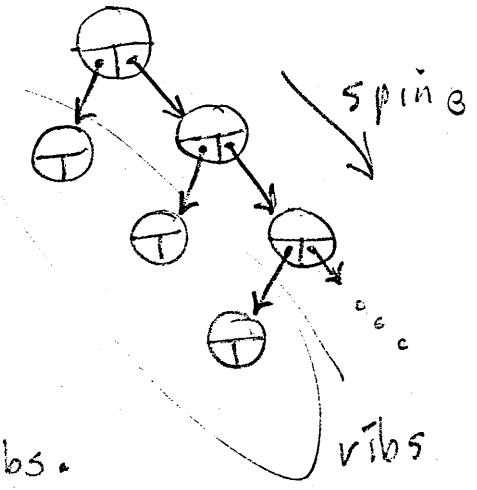
for deletion variant i (page 2), when an insertion follows a deletion, the demotion path will always follow the previous promotion path, so the following tree will have a worst case O(N) over a long sequence of insertions and deletions:

delete, then insert  then delete, then insert  ...

forever.



for deletion variant ii (page 2) try to construct a worst case:



after doing an insert down the spine ↘, all pointers will point into the ribs.

an insertion must be done into the ribs (or a deletion from them) to get an other insertion down the spine!

similar for deletion.

As a results if the structure is to remain constant in size, an equal number of insertions and deletions must be made in the ribs and spine at each level:

the average cost at level k will be,

$$cost(K) = \frac{\text{rib cost} + (1 + cost(K+1))}{2}$$

solution

cost(K) approaches ribcost + 1

as K approaches infinity —— ugh!