Tournament sort

Tournament sort is a sorting algorithm. It improves upon the naive selection sort by using a priority queue to find the next element in the sort. In the naive selection sort, it takes O(n) operations to select the next element of n elements; in a tournament sort, it takes O(log n) operations (after building the initial tournament in O(n)). Tournament sort is a variation of heapsort.

Tournament sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	O(n log n)
Average performance	O(n log n)

Contents

Common application

Tournament replacement selection sorts are used to gather the initial runs for external sorting algorithms. Conceptually, an external file is read and its elements are pushed into the priority queue until the queue is full. Then the minimum element is pulled from the queue and written as part of the first run. The next input element is read and pushed into the queue, and the min is selected again and added to the run. There's a small trick that if the new element being pushed into the queue is less than the last element added to the run, then the element's sort value is increased so it will be part of the next run. On average, a run will be 100% longer than the capacity of the priority queue.[1]

Tournament sorts may also be used in N-way merges.

The tournament

The name comes from its similarity to a single-elimination tournament where there are many players (or teams) that play in two-sided matches. Each match compares the players, and the winning player is promoted to play at match at the next level up. The hierarchy continues until the final match determines the ultimate winner. The tournament determines the best player, but the player who was beaten in the final match may not be the second best – he may be inferior to other players the winner bested.

Implementation

The following is an implementation of tournament sort in Haskell, based on Scheme code by Stepanov and Kershenbaum.[2]

```haskell
import Data.Tree


-- | Adapted from `TOURNAMENT-SORT!` in the Stepanov and Kershenbaum report.
tournamentSort :: Ord t
       => [t]  -- ^ Input: an unsorted list
       -> [t]  -- ^ Result: sorted version of the input
tournamentSort alist
        = go (pure<$>alist) -- first, wrap each element as a single-tree forest
 where go [] = []
       go trees = (rootLabel winner) : (go (subForest winner))
        where winner = playTournament trees


-- | Adapted from `TOURNAMENT!` in the Stepanov and Kershenbaum report
playTournament :: Ord t
         => Forest t -- ^ Input forest
         -> Tree t   -- ^ The last promoted tree in the input
playTournament [tree] = tree
playTournament trees = playTournament (playRound trees [])


-- | Adapted from `TOURNAMENT-ROUND!` in the Stepanov and Kershenbaum report
playRound :: Ord t
       => Forest t -- ^ A forest of trees that have not yet competed in round
       -> Forest t -- ^ A forest of trees that have won in round
       -> Forest t -- ^ Output: a forest containing promoted versions
                   --   of the trees that won their games
playRound [] done = done
playRound [tree] done = tree:done
playRound (tree0:tree1:trees) done = playRound trees (winner:done)
 where winner = playGame tree0 tree1


-- | Adapted from `TOURNAMENT-PLAY!` in the Stepanov and Kershenbaum report
playGame :: Ord t
         => Tree t  -- ^ Input: ...
         -> Tree t  -- ^ ... two trees
         -> Tree t  -- ^ Result: `promote winner loser`, where `winner` is
                    --   the tree with the *lesser* root of the two inputs
playGame tree1 tree2
    | rootLabel tree1 <= rootLabel tree2  = promote tree1 tree2
    | otherwise                           = promote tree2 tree1


-- | Adapted from `GRAB!` in the Stepanov and Kershenbaum report
promote :: Tree t  -- ^ The `winner`
        -> Tree t  -- ^ The `loser`
        -> Tree t  -- ^ Result: a tree whose root is the root of `winner`
                   --   and whose children are:
                   --   * `loser`,
                   --   * all the children of `winner`
promote winner loser = Node {
    rootLabel = rootLabel winner,
    subForest = loser : subForest winner}


main :: IO ()
main = print $ tournamentSort testList
 where testList = [-0.202669, 0.969870, 0.142410, -0.685051, 0.487489, -0.339971, 0.832568,
0.00510796, -0.822352, 0.350187, -0.477273, 0.695266]
```

References

1. Donald Knuth, The Art of Computer Programming, Sorting and Searching, Volume 3, 1973. The "snowplow" argument. p. 254
2. Stepanov, Alexander and Aaron Kershenbaum. *Using Tournament Trees to Sort*, Brooklyn: Center for Advanced Technology in Telecommunications, 1986.

- Kershenbaum et al 1988, "Higher Order Imperative Programming" (http://stepanovpapers.com/hop.pdf)