```
( Timer queue implementation               rjb 16:06 07/2186 )

( --------------  structure declaring words  ---------------- )

: <struc> CREATE , DOES> @ + ; ( 2nd generation defining word )

( 'struc' is used to create the defining word for a structure )
: struc CREATE ,    ( org struc <struc> ; creates a structure )
        DOES>   ( size <struc> <element> ; creates an element )
         DUP @ DUP >R ROT + SWAP !   ( update location counter )
         R> <struc> ;                ( make word for element )

( 'array' is used to create an array of structures )
: array CREATE ( #slots slot-size array <array> ; makes array )
         DUP , * ALLOT     ( save slot-size and allocate array )
         DOES>         ( subscript <array> -- &<array>element )
         DUP @ ROT * + WSIZE + ;  ( return pointer to the slot )

( 'org' is useful for doing the 4th version of a C 'union' )
: org   ( n org <strucname> ; re-initializes location counter )
         BL WORD FIND NOT ABORT" Undefined! " >BODY ! ;

( 'sizeof' is used for declaring structures of structures )
: sizeof ( sizeof <strucname> -- n ; gets the size of a struc )
         BL WORD FIND NOT ABORT" Undefined " >BODY @
         STATE @ IF [COMPILE] LITERAL THEN ; IMMEDIATE

(  ---------------- timer queue entry  ---------------- )

    0   struc   tq              ( a node in the timer queue )
         4       tq      key     ( the sort key: dispatch time )
         4       tq      dist      ( distance to nearest leaf )
         4       tq      left      ( pointer to left sub-tree )
         4       tq      rite     ( pointer to right sub-tree )
         4       tq      father    ( pointer to father of node )
         4       tq      word    ( word to execute at dispatch )
         4       tq      data      ( pointer to data for word )

( ---------------- timer queue implementation ---------------- )

( left! & rite! set the left and rite subtrees, respectively
  of a node, called the father.  The father pointer of the son
  node is updated to point to the pointer, left or rite, that
  points to the son, so that it may be cleared on an unque. )

: left! DUP 0<> IF OVER left OVER father !    ( Father Son -- )
         THEN SWAP left ! ;

: rite! DUP 0<> IF OVER rite OVER father !    ( Father Son -- )
         THEN SWAP rite ! ;

: go-rt DUP rite @ >R DUP ROT rite! R> ;    ( go dn rite side )

: dist@ DUP IF dist @ THEN ;        ( P -- P=nil ? 0 : dist @ )

( This is step M2: from Knuth p 619, sol'n to prob. 32, p 159 )
```

```
 57 : list-merge       ( P Q R -- P Q R D ; merge priques P & Q, )
 58     BEGIN  ( R is Roving pointer, D is Distance to near leaf )
 59       OVER 0= IF                          ( Q = nil ? )
 60          2 PICK dist@ EXIT THEN  ( yes, D = P->dist; done! )
 61       2 PICK 0= IF                        ( P = nil ? )
 62          ROT DROP OVER SWAP                 ( yes, P = Q )
 63          2 PICK dist@ EXIT THEN  ( yes, D = P->dist; done! )
 64       2 PICK key @   2 PICK key @       ( P->key < Q->key ? )
 65       < IF ROT go-rt ROT ROT ELSE      ( yes, P moves right )
 66           SWAP go-rt SWAP   THEN      ( no,  Q moves right )
 67    AGAIN ;       ( loop until one of the trees is eliminated )
 68
 69 ( This is steps M3: and M4: from Knuth p 619 )
 70 : fix-dist ( P Q R D -- P ; fixes up distance to nearest leaf )
 71   BEGIN
 72     OVER 0= IF 3DROP EXIT THEN      ( R = nil ?  yes, done! )
 73     ROT DROP OVER rite @ ROT ROT             ( Q = R->rite )
 74     OVER left @ dist@ OVER < IF       ( R->left->dist < D ? )
 75       DROP DUP left @ dist@ 1+     ( D = R->left->dist + 1 )
 76       OVER DUP left @ rite!           ( R->rite = R->left )
 77       OVER 4 PICK left!     ELSE            ( R->left = P )
 78       1+ OVER 4 PICK rite!  THEN       ( D++; R->rite = P )
 79     DUP 2 PICK dist !                    ( R->dist = D )
 80     >R ROT DROP SWAP DUP R>                ( P = R; R = Q )
 81   AGAIN ;                  ( spin down the right sub-tree )
 82
 83 ( tq-merge is Knuth's Algorithm M from p 619 )
 84 : tq-merge 0 list-merge fix-dist ;  ( P Q -- P ; merge 2 tq's )
 85
 86 ( the timer queue root pointer )          VARIABLE TQ 0 TQ !
 87
 88 ( TQ! updates the father pointer in the first node, & sets TQ )
 89 : TQ!                  ( tq -- ; sets TQ and father of tq )
 90       DUP TQ !                 ( set the timer queue head )
 91       DUP 0= IF DROP EXIT THEN         ( empty? yes, done! )
 92       father TQ SWAP ! ;       ( no, set father of top node )
 93
 94 : ?waiting DUP father @ @ = ;                 ( tq -- flag )
 95
 96 ( ------------ timer queue package entry points ------------- )
 97
 98 : tq-enque                      ( &tq -- ; enques to timer )
 99       0 OVER left ! 0 OVER rite !   ( nullify both sub-trees )
100       1 OVER dist !               ( distance to a leaf is 1 )
101       TQ @ tq-merge TQ! ;    ( merge new node with old queue )
102
103 : tq-deque                     ( -- &tq ; deques from timer )
104       TQ @ DUP 0= IF EXIT THEN  ( returns nil on empty queue )
105       DUP                          ( save head for answer )
106       DUP left @ SWAP rite @ tq-merg TQ! ;   ( merge remains )
107
108 : tq-unque                  &tq -- &tq ; removes from timer )
109       DUP ?waiting NOT IF EXIT THEN 0 OVER father @ !  ( cut )
110       DUP left @ OVER rit
```