Reprinted from

Computer Physics Communications

Computer Physics Communications 92 (1995) 214–224

An empirical assessment of priority queues in event-driven molecular dynamics simulation

Mauricio Marín [a,1], Patricio Cordero [b,2]

[a] *Departamento de Computación, Universidad de Magallanes, Casilla 113-D, Punta Arenas, Chile*

[b] *Departamento de Física, Universidad de Chile, Casilla 487-3, Santiago, Chile*

Received 17 January 1995; revised 31 May 1995

ELSEVIER

Computer Physics Communications 92 (1995) 214–224

Computer Physics Communications

An empirical assessment of priority queues in event-driven molecular dynamics simulation

Mauricio Marín [a,1], Patricio Cordero [b,2]

[a] *Departamento de Computación, Universidad de Magallanes, Casilla 113-D, Punta Arenas, Chile*
[b] *Departamento de Física, Universidad de Chile, Casilla 487-3, Santiago, Chile*

Received 17 January 1995; revised 31 May 1995

Abstract

In the last decades a number of near optimal priority queues have been developed. Many of these priority queues are suitable for the efficient management of events generated during simulations of hard-particle systems. In this paper we compare the execution times of the fastest priority queues known today as well as some forms of binary search trees used as priority queues. We conclude that an unusual adaptation of a strictly balanced binary tree has the best performance for this class of simulations.

Keywords: Molecular dynamics; Algorithms; Hard particles; Priority queues

1. Introduction

Event-driven molecular dynamics [1] is the optimal choice with picewise constant interactions. In this case particles move free of each other except at discrete times when they suffer impulsive forces or *events*. The evolution of each particle between events follows Newton's equations of motion with whatever external field (e.g., gravity) may exist. The events take place whenever a particle hits one of the steps of its potential energy function. The simulation jumps analytically from one event to the next following the variable intervals that the dynamics of the system dictates. Henceforth it is an *event-driven* molecular dynamics. Application of these type of simulations can be found in [1–13] and efficient algorithms to simulate large systems are found in [14–18].

During initialization the general simulation algorithm consists of computing *event-times* among each pair of particles. These times together with the identifiers of the particles involved in each event are stored as event-tuples in a *future event list* (FEL). Next the event-tuples are removed one by one from the FEL (in increasing time order) and processed chronologically. Processing an event causes a change in the states of the particles involved in the current event and a computation of new events for these particles, i.e., new event-tuples that are then inserted in the FEL. This process is cyclically repeated until some end condition is reached. After

[1] E-mail: mmarin@ona.fi.umag.cl.
[2] E-mail: pcordero@cec.uchile.cl.

SSDI 0010-4655(95)00120-4

each event other events stored in the FEL are invalidated because of the change in the states of the particles presently involved.

The basic requirement for event management is a FEL that efficiently supports the following operations: NEXTEVENT() that *extracts* the event of smallest time from the set of events calculated in previous steps of the simulation, SCHEDULE(e) that *inserts* a new event e in the FEL, and CANCEL(i) that *deletes* all the events associated to particle i. These operations define the FEL as a *priority queue* (PQ) since NEXTEVENT only retrieves the event with higher priority, namely the one with smallest *event-time*.

Efficient solutions for the implementation of PQs have been proposed by several authors [19–25]. Most of these PQs have been empirically compared using different approaches and applications [26–30]. From this it has been concluded that no single implementation is the best for all cases and applications. For example, the tests reported in [30] show that the *pairing heap* [23] is the best structure for the minimum spanning tree problem. But the tests presented in [28] show that the *splay tree* [21] is the best structure under the empirical *hold model* with several probability distributions for the priorities.

The data structures used as PQs in hard-particle simulations [14–18] have been the *binary search tree* [14,16,24,25], the *implicit heap* [15,17,24,25] and the *complete binary tree* [18,24]. But besides the aforementioned data structures, there are several other PQs that have been developed during the last decades, such as the *skew heap* [22], the *binary priority queue* [25], the *leftist tree* [24,25], the *binomial queue* [20] and the *priority tree* [19,25], whose suitability for event-driven molecular dynamics simulations have not yet been analyzed.

It is important to note that many of these strategies have a similar near-optimal performance from the theoretical viewpoint. The selection of the best PQ for hard-particle simulations from these analysis is difficult. For example, some final-expressions for the performance are given as asymptotic bounds $O(\ )$ in terms of different concepts such as worst-case and amortized-time. But even in the case of strategies with the same type of bounds we must remember that the descriptor $O(\ )$ encloses coefficients and lower-order terms that can play a significant role when selecting the best PQ. Thereby, when dealing with competing strategies, we must finally resort to a well-designed experimental framework to determine the best PQ for the particular application at hand.

The results of this paper have been mainly obtained from simulating a two-dimensional gas of hard disks using the general strategy described in [18] but replacing in the FEL administrator ten different PQs to compare the relative performance of them. For our empirical tests we have selected the PQ implementations reported in the literature as the most efficient ones, together with some other PQs for which no tests have been reported. We conclude that our implementation of the *complete binary tree* [24] is the best PQ for hard particle simulations when the number of particles is either fixed or variable. The implementation for the *complete binary tree* that we propose in this work (see Appendix) is faster than the one proposed in [18].

The next sections are organized as follows: Section 2 gives details about the experiments performed, Section 3 describes the PQs tested and mentions the empirical results, Section 4 presents a theoretical analysis of the performance of the *complete binary tree*, and Section 5 analizes the results and gives our conclusions. An appendix provides a C-language implementation of the *complete binary tree* reported as the most efficient one in all our experiments.

2. Experimental design

The empirical results were obtained simulating a system of N hard disks moving inside a rectangular box using the general conditions defined in [18]. To compare the different PQs we have used the same program – with the algorithms proposed in [18] – but replacing in each case the PQ used in [18] by one of the queues described in Section 3. We show results for different area densities $\rho = 0.1, \ldots, 0.7$ for systems of N particles with $N = 10^1, \ldots, 10^4$. Each experiment was stopped after $100\,N$ disk-disk collisions.

To measure the execution times of each PQ the standard *clock*( ) function was used. More precision for this function was attained measuring only the *total execution time* of the simulation program in opposition to measure the time of each function used to administer the PQ. By choosing this alternative we have negleted the fact that the program has three major components which affect the overall running time. They are: (a) the basic cycle, where the event-times are calculated, (b) the cell administrator, that enables the calculations of events between neighboring particles, and (c) the PQ used to manage the events. Note that in [18] it was empirically determined that the weight of the PQ in the total running time is below 23% for a system with 2500 particles ranging through densities from 0.05 to 0.7. Then for our experiments it was expected that a significant reduction in the total running time could only be achieved by introducing a considerably more efficient PQ.

We have used the *clock*( ) function to measure the total running time of the simulation with a $t_1 = clock()$ instruction at the end of the initialization and executing $t_2 = clock()$ at the end of the simulation of $100\,N$ disk-disk collisions. Thus the total running time associated with each PQ was $t_2 - t_1$. Each experiment was repeated 10 times observing an error rate below 0.1% in all the computers used for executing the experiments. This error is defined as the ratio between the standard deviation and the average.

The programming style was the traditional one with minimal calls to subroutines and no recursive algorithms. The experiments were carried out in workstations DEC Alpha, DG Aviion and SUN 690, all with UNIX at minimal load (no other users and no background processing), and programs written in C language compiled with the gcc compiler at the maximal speed option O2. To avoid the effects of paging activity in the total running time a relatively small number of particles ($10 \rightarrow 10^4$) was used. To avoid calls to the operating system due to PQs that use dynamic memory allocation, all the memory required by these PQs was pre-allocated before the start of each experiment.

3. Priority queues and empirical results

The first PQ implementations that we compared were the following:

- **Complete Binary Tree 1 (CBT1)** in which each leaf has the label associated to one different particle and each internal node (recursively up to the root) has the particle label with smallest event-time of its two children [18,24]. No deletions are performed in the tree since every time a particle i changes its event-time, the path from the leaf i is updated up to the root. This is the implementation proposed in [18].
- **Complete Binary Tree 2 (CBT2)** similar to CBT1 but *with* node deletions. Deletions from the tree are performed by removing the rightmost leaf and exchanging it with the target leaf to be deleted. Then the tree is updated up to the root considering the above changes. Insertions are performed by appending a new rightmost leaf and updating the CBT.
- **Complete Binary Tree 3 (CBT3)** similar to CBT1 but avoiding updates up to the root of the tree. In this case, the event-time comparisons are stopped as soon as possible (compare with CBT1).
- **Complete Binary Tree 4 (CBT4)** similar to CBT3 but with node deletions like CBT2.
- **Implicit Heap 1** using the implementation proposed in [25] but with further code to reduce the number of swaps between nodes. No node deletions are performed in the tree since every time a node changes its event time this is moved up or down depending on its new value.
- **Implicit Heap 2** similar to *implicit heap* 1 but with node deletions.

Note that up to now we have basically described two PQs. The *complete binary tree* has not been considered in the literature as a useful data structure in practice until [18] (in [24] the *complete binary tree* is described only as a first approximation to the *implicit heap*.) However, the CBT3 was consistently more efficient than all other PQs in all experiments and computers that we used. Also note that we have implemented the *heap* and the *complete binary tree* with and without deletion of nodes in the tree. The reason to include node deletion was to compare the performance of the *implicit heap* and *complete binary tree* with other PQs for which it is impossible to avoid deletions. Even though we have tested some heap ordered structures for which deletions

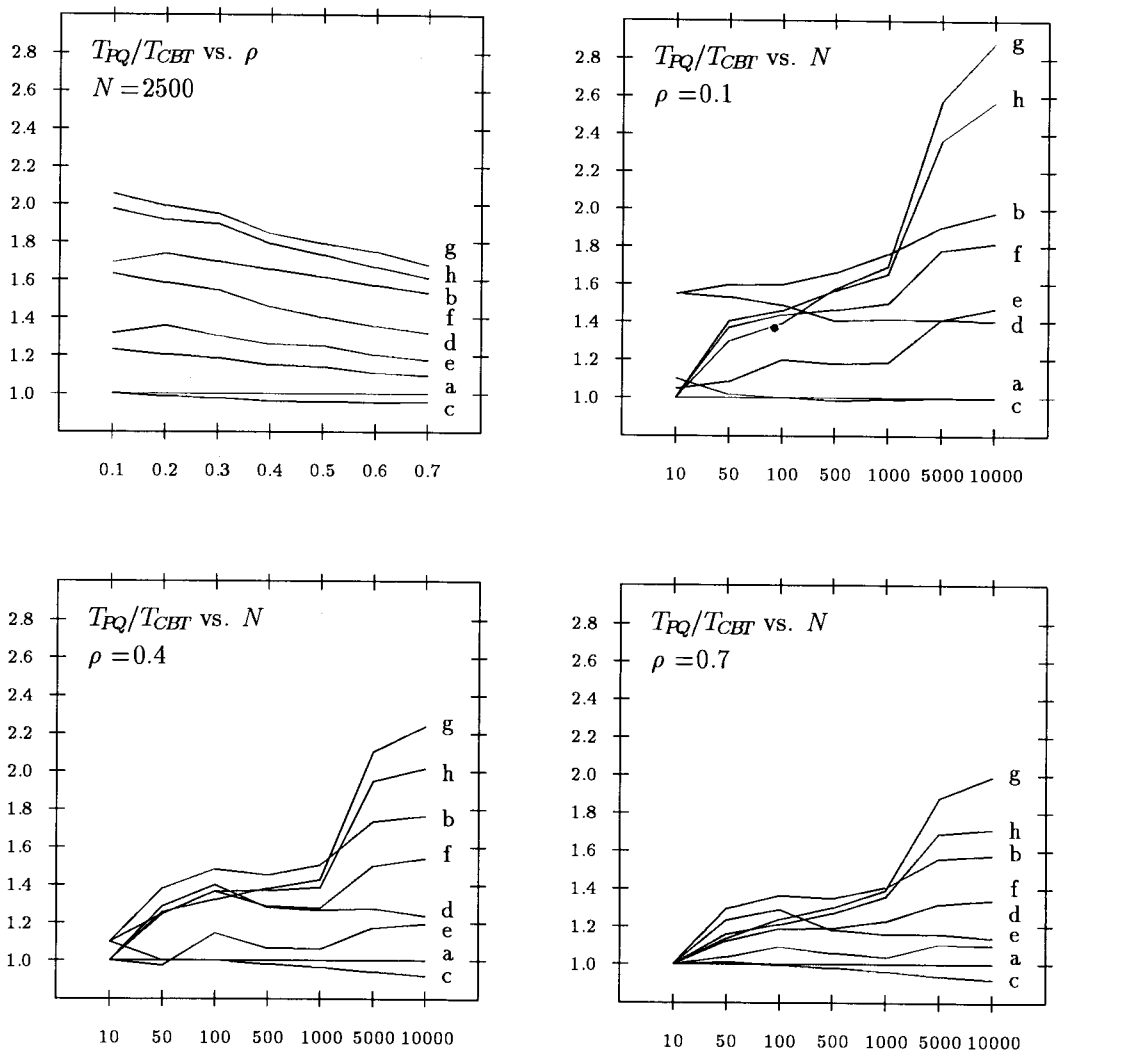

Fig. 1. Total running time for simulations executed in a DEC Alpha computer. Each curve shows the ratio (T_{PQ}/T_{CBT1}) between the running time with a priority queue PQ and the running time with the *complete binary tree* CBT1. Each priority queue PQ is identified by a letter: (a) CBT1, (b) CBT2, (c) CBT3, (d) CBT4, (e) Implicit heap 1, (f) Pairing heap, (g) Priority tree, (h) Binary PQ.

can be avoided, we have not considered such an alternative because the update algorithms (up/down shifting) become identical to the *implicit heap*. The other PQ implementations tested in this work were the following:

- **Binary Search Tree** [24,25].
- **Pairing Heap** with twopass variant [23].
- **Skew Heap** with top-down variant [22].
- **Splay Tree** with the bottom-up splaying [21].
- **Binary Priority Queue** [25].
- **Leftist Tree** [24,25].
- **Binomial Queue** [20,25].
- **Priority Tree** [19,25].

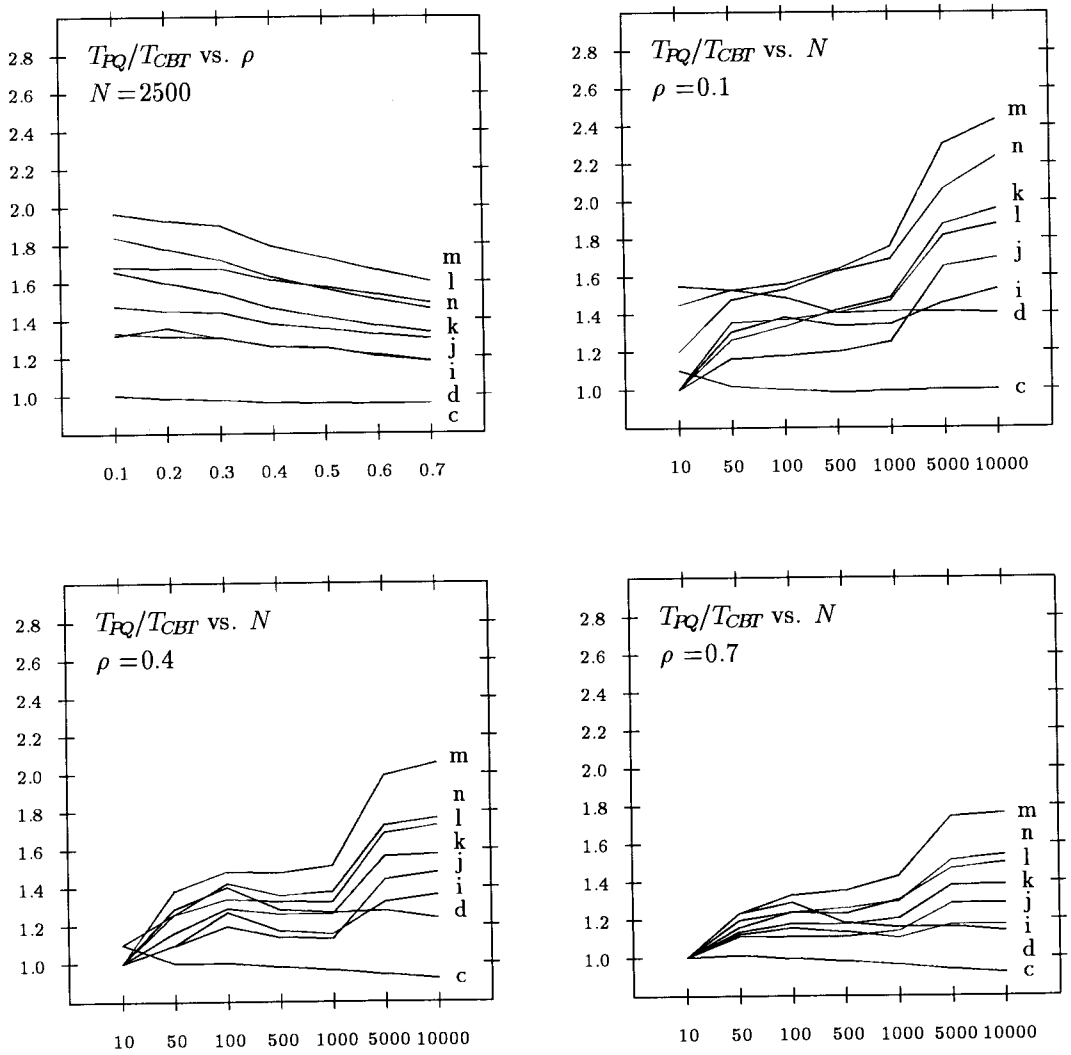

Fig. 2. Total running time for simulations executed in a DEC Alpha computer. Each curve shows the ratio (T_{PQ}/T_{CBT1}) between the running time with a priority queue PQ and the running time with the *complete binary tree* CBT1. Each priority queue PQ is identified by a letter: (c) CBT3, (d) CBT4, (i) Implicit heap 2, (j) Binary search tree, (k) Skew heap, (l) Splay tree, (m) Binomial queue, (n) Leftist tree.

To have a first glance at the performance of these PQs look at Figs. 1 and 2. They show the empirical results obtained in a DEC Alpha WorkStation. Note that all our results are presented as the ratio, T_{PQ}/T_{CBT1}, of the running time T_{PQ} associated to one of the PQs over the total running time obtained with the *complete binary tree* implementation proposed in [18] (T_{CBT1}). Similar results were obtained in the DG Aviion and SUN 690 computers, therefore we omit plotting those data. It should be noted, however, that the maximum values for T_{PQ}/T_{CBT1} were 2.54 in DG Aviion and 2.92 in SUN 690.

4. Analysis of the CBT

The theoretical analysis of the CBT is simple. From Fig. 3 we can observe that some leaf nodes are at the lowest level (level k) and the rest (of the leaf nodes) are one level up (level $k-1$). The cost – in event-time comparisons – of updating the tree from a leaf to the root is: (a) $\lfloor \log_2 N \rfloor + 1$ from level k and (b) $\lfloor \log_2 N \rfloor$ from level $k-1$. Furthermore, the number of leaves at level k is $2N - 2^{\lfloor \log_2 N \rfloor + 1}$ and at level $k-1$ is $2^{\lfloor \log_2 N \rfloor + 1} - N$. Taking the average, the cost of the CBT is

$$\lfloor \log_2 N \rfloor + 2 - \frac{1}{N} 2^{\lfloor \log_2 N \rfloor + 1}.$$

If N is an integer power of 2 then the optimal cost $\log_2 N$ is reached while the worst case comes when updating from level k upwards and it has cost of $\lfloor \log_2 N \rfloor + 1$ (event-time comparisons).

For each processed event it is always necessary to make at least one update up to the root of the CBT and the worst case, when processing an event, has a cost not larger than $2\lfloor \log_2 N \rfloor + 2$ (or $2 \log_2 N$ when N is a power of 2). Therefore the CBT performance as a PQ for hard particle simulations is close to $O(\log N)$ from its best to its worst cases.

The efficiency of some of the PQs tested in this work depends on the probability distribution of the time increment of the new inserted events. That is, if an event occurs at time t and n new events with times t_1, t_2, ..., t_n are inserted, the probability distribution of the differences $t_i - t$ ($i = 1...n$) may affect noticeably the performance of some PQs [27,28,31]. The reduction on performance of these PQs (which are different forms of binary trees) come from the fact that depending on the $t_i - t$ distribution the data structure tends to lose its balance. Experimental results in the hard-disk systems used in this work indicate that the probability distribution of the increments $t_i - t$ is very close to an exponential distribution. This leads to a best case behavior of the $t_i - t$ dependent PQs. However, the CBT has the nice property of being a strictly balanced binary tree so its logaritmic bounds are maintained no matter which is the system being simulated.

Notice that the $O(\log N)$ bounds for the CBT operations contrasts with better theoretical bounds attained by other PQs. For example, some of them have a few PQ operations with cost $O(1)$. But usually this better bounds are accomplished by using more complex algorithms that execute additional computer intructions to maintain a suitable data structure. Hence, depending on the application, these additional intructions can produce an overhead in the PQ running time that override the gain in performance obtained with better bounds.

For instance, in our experiments we have observed that the *implicit heap* 1 performed less event-time comparisons than the CBT1 (showing that the analytical *implicit heap* bounds are consistently better than the ones for the CBT), but this gain in performance was not noted in the running time. The reason is quite simple: the cost of each *swap* in the *implicit heap* is higher than the cost of each *match* in the CBT (in each internal CBT-node n a match consists of comparing the event-times of the children of the node n, and writing in n the identifier of the winner child, namely the son with lesser event-time, see Fig. 3.) In the CBT each match takes one event-time comparison and one assignment whereas with a careful implementation each swap in the *implicit heap* takes at least two event-time comparisons and two assignments.

5. Analysis of results and conclusions

We have presented empirical results that show the importance of selecting a convenient PQ implementation when performing event driven simulations of hard particle systems. We emphasize that in our experiments we have selected the fastest PQ implementations reported in the literature and obtained performances (total running time of the simulation) that differ among them by as much as a factor of three. This reinforces the use of experimental tests to determine the best PQ for hard-particle simulations.

An important conclusion from our experiments is that the PQ implementations based on the *complete binary tree* were the most efficient ones to simulate the hard-disk system. In particular, the fastest implementations in all of our experiments were those that reduce the event-time comparisons performed in the tree, namely CBT3 and CBT4 (it should be clear that CBT4 must be compared with strategies that perform node deletions.) In addition, the theoretical analysis of the previous section helps us to understand the efficiency of the CBT in this kind of simulations and encourages us to expect similar optimal behavior in other types of hard-particle systems and software/hardware architectures.

In Figs. 1 and 2 the curves with letters (c) and (d) represent the performance of CBT3 and CBT4 respectively. These curves show that for large N CBT3 and CBT4 perform better than the other PQ implementations. This better performance was observed both when there is node deletion, (d), and when there is not, (c).

Another important and decisive feature in favour of CBT3 and CBT4 is that they tend to be more efficient than other PQs as the number of particles N increases. This property of CBT3 and CBT4 makes them particularly convenient for the simulation of large systems. For the node deletion case, however, several other structures were more efficient than CBT4 for systems with $N < 1000$. On the other hand for $N > 100$ CBT3 was the absolute winner.

It is also interesting to observe that in the case of node deletion, CBT2 was by far less efficient than CBT4. This shows the effectiveness of reducing the number of event-time comparisons in the *complete binary tree*. The effect of this reduction is less evident for the case with no node deletion. Node deletion can be useful in systems where the particles are created and eliminated dynamically during the simulation. In this case we propose applying a mixed strategy using CBT3 algorithms during all the life-time of particles and using CBT4 algorithms only when it is necessary to remove particles from the system.

Therefore – based on empirical results obtained with different machines, the logarithmic bounds for the CBT operations, that are independent of the type of system being simulated, together with the low cost of each CBT match – we conclude that the best choice is the *complete binary tree* (CBT3 or CBT4) for performing event-driven simulations of hard particle systems with any of the approaches proposed in [14–18].

Finally we remark that in [18] it was empirically determined that the strategy of [18] performs better both in running time and computer memory used than the strategies proposed in [14,15]. The strategy proposed in [17] is similar to [15]. However, both strategies [15,17] use a simpler cell administrator than the one used in [14,18] because they always scan the complete neighborhood of the particles involved after each event that takes place. This implies that there is less programming work at the expense of a slower simulation.

Acknowledgements

This work has been partially funded by Chilean research projects FONDECYT 1931105 and 1950622, and U.Magallanes F1-O1IC-95.

Appendix. CBT3 and CBT4

Data structure

To obtain the event with lesser time in the PQ a *complete binary tree* (CBT) that performs a binary tournamen between all the event times is used. Each leaf has a particle number and each internal node (recursively up t the root) has the particle number with the smallest event-time of its two children. Therefore the root of th tree has the particle number with smallest event-time (see Fig. 3a). Every time a new event-time is compute for a particle i, the tournament is updated for all nodes in the path from the leaf labeled with i to the root (se Fig. 3b).

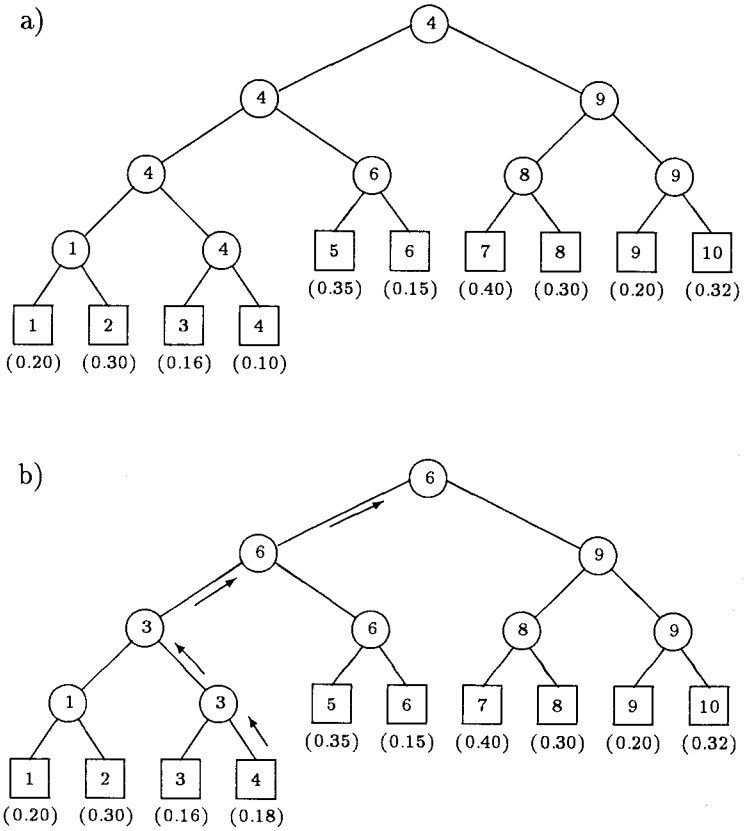

Fig. 3. (a) CBT for ten objects (event-time in parentheses). (b) CBT updated after changing the event-time for object 4 to 0.18.

The CBT is implemented using an array of $2N-1$ integers. A node at position n has its children at positions $2n$ and $2n+1$. The parent of node n is at position $\lfloor \frac{n}{2} \rfloor$ of the array. All internal nodes are stored between positions 1 and $N-1$.

Deletions from the CBT are performed by removing the rightmost leaf and exchanging it with the target leaf to be deleted. Then we have to update the CBT considering the above changes. Insertions are performed by appending a new rightmost leaf and updating the CBT.

C-language implementation

```
#define N  10000  /* Number of particles */
int CBT[N*2];     /* Complete Binary Tree implemented
                     in an array of 2*N integers */
EVENT *Min[N+1];  /* Array of pointers to the event with
                     minimal time for each particle */

void UpdateCBT(i)
   int i; /* particle number */
{
   int f, /* father */
```

```
        l, /* left child */
        r, /* right child */
        w; /* old winner */

    /* At least it is necessary to cover the old path of particle i  */
    for( f=CBT[i+N-1]/2; f>0; f=f/2 ) {
        if ( CBT[f]!=i ) break; /* jumps to the next "for" */
        l = CBT[f*2];
        r = CBT[f*2+1];
        if ( Min[l]->time < Min[r]->time )
            CBT[f] = l;
        else
            CBT[f] = r;
        }
    /* Now the event time comparisons are stopped as soon as possible  */
    for( ; f>0; f=f/2 ) {
        w = CBT[f];  /* old winner */
        l = CBT[f*2];
        r = CBT[f*2+1];
        if ( Min[l]->time < Min[r]->time )
            CBT[f] = l;
        else
            CBT[f] = r;
        if ( CBT[f] == w ) return; /* end of the event time comparisons */
        }
} /* End of UpdateCBT */


/******* Delete operation for the complete binary tree ******/

int Leaf[N+1]; /* when the number of particles is not constant it is
                  necessary to maintain an additional array to indicate
                  the leaf associated with each particle. In this case
                  the initialization "f=CBT[i+N-1]/2" of the first "for"
                  in UpdateCBT must be replaced with "f=Leaf[i]/2" . */

int NP;       /* current number of particles */

void Delete(i)
    int i;
{
    int l;
    void UpdateCBT();

    if ( NP<2 ) { CBT[1]=0; Leaf[0]=1; NP--; return; }

    l=NP*2-1; /* the last leaf */
```

```
    if (CBT[l-1]!=i) {
       Leaf[CBT[l-1]]=l/2;
       CBT[l/2]=CBT[l-1];
       UpdateCBT(CBT[l-1]);
       }
    else {
       Leaf[CBT[l]]=l/2;
       CBT[l/2]=CBT[l];
       UpdateCBT(CBT[l]);
       NP--;
       return;
       }
    if (CBT[l]!=i) {
       CBT[Leaf[i]]=CBT[l];
       Leaf[CBT[l]]=Leaf[i];
       UpdateCBT(CBT[l]);
       }
    NP--;
    return;
} /* End of Delete() */
/******* Insert operation for the complete binary tree ******/

void Insert(i)
     int i;
{
     int j;
     void UpdateCBT();

     if (NP==0) {CBT[1]=i; NP++; return;}

     j=CBT[NP];
     CBT[NP*2]=j;
     CBT[NP*2+1]=i;
     Leaf[j]=NP*2;
     Leaf[i]=NP*2+1;
     NP++;
     UpdateCBT(j);

} /* End of Insert() */
```

Note that there are several alternatives for the implementation of CBT3 and CBT4. Another implementation that takes into account the limits of some compilers can use two arrays of N integers instead of one array of $2N$ integers. It is also possible to implement the *complete binary tree* using dynamic memory allocation as with the *binary search tree*. In this case, each node has one integer to store the particle number, and three pointers to store the children and father of the node (other PQs using dynamic memory allocation require the same number of pointers).

References

[1] B.J. Alder and T.E. Wainright, J. Chem. Phys. 27 (1957) 1208.
[2] B.J. Alder and T.E. Wainright, J. Chem. Phys. 31 (1959) 459.
[3] M. Mareschal and E. Kestemont, Phys. Rev. A 30 (1984) 1158.
[4] E. Melburg, Phys. Fluids 29 (1986) 3107.
[5] M. Mareschal and E. Kestemont, J. Stat. Phys. 48 (1987) 1187.
[6] A. Puhl, M. Malek-Mansour and M. Mareschal, Phys. Rev. A 40 (1989) 1999.
[7] M. Mareschal, M. Malek-Mansour, A. Puhl and E. Kestemont, Phys. Rev. Lett. 41 (1988) 2550.
[8] D.C. Rapaport, Phys. Rev. Lett. 60 (1988) 2480.
[9] D.C. Rapaport, Phys. Rev. A 43 (1991) 7046; A 46 (1992) 1971.
[10] D. Risso and P. Cordero, in: Condensed Matter Theories, Vol. 7, eds. A.N. Pronto and J. Aliaga (Plenum, New York, 1991).
[11] D. Risso and P. Cordero, in: Instabilities and Nonequilibrium Structures, E. Tirapegui and W. Zeller, eds. (Kluwer, Dordrecht, 1992)
[12] J. Koplik, J.R. Banavar and J.F. Willemsem, Phys. Rev. Lett. 60 (1988) 1282.
[13] M. Alaoui and A. Santos, Phys. Fluids A 4 (1992) 1273.
[14] D.C. Rapaport, J. Comput. Phys. 34 (1980) 184.
[15] B.D. Lubachevsky, J. Comput. Phys. 94 (1991) 255.
[16] K. Shida and Y. Anzai, Comput. Phys. Commun. 69 (1992) 317.
[17] A. Krantz, Ph.D. thesis, University of Colorado, Departament of Computer Science, Boulder, CO 80309-0430 (1993).
[18] M. Marín, D. Risso and P. Cordero, J. Comput. Phys. 109 (1993) 306.
[19] A.T. Jonassen and O.J. Dahl, BIT 15 (4) (1975).
[20] M.R. Brown, SIAM J. Comput. 7 (3) (Aug. 1978).
[21] R.E. Tarjan and D.D. Sleator, J. ACM 32 (3) (July 1985) 652–686.
[22] D.D. Sleator and R.E. Tarjan, SIAM J. Comput. 15 (1) (Feb. 1986).
[23] J.T. Stasko and J.S. Vitter, Commun. ACM 30 (1987).
[24] D.E. Knuth, The Art of Computer Programming, Vol. 3, Sorting and Searching (Addison-Wesley, Reading, MA, 1973).
[25] G.H. Gonnet and R. Baeza-Yates, Handbook of Algorithms and Data Structures, 2nd Ed. (Addison-Wesley, Reading, MA, 1991).
[26] W.R. Franta and K. Maly, Commun. ACM 21 (10) (Oct. 1978).
[27] W. McCormack and R. Sargent, Commun. ACM 24 (1981) 801.
[28] D.W. Jones, Commun. ACM 29 (1986) 300.
[29] A.M. Liao, Algorithmica 7 (1992).
[30] B.M. E Moret and H.D. Shapiro, in: DIMACS Series in Discrete Mathematics and Theorical Computer Science, Vol.15 (June 1994)
[31] J.H. Kingston, Acta Informatica 11 (1) (April 1985).