On the Pending Event Set and Binary Tournaments

Mauricio Marín
PRG, Computing Laboratory, Oxford University
E-mail: mmarin@comlab.ox.ac.uk

1 Introduction

Modern heaps are among the fastest general purpose priority queue (PQ) realisations, and they are usually used to implement the pending event set in discrete-event simulations. Optimal structures like the skew heap, are an efficient alternative to special purpose structures such as the calendar queue and the lazy queue. Conversely to special purpose queues, heaps exhibit the valuable attributes of simplicity and independence of the event-time distribution. Binary tournaments are the genesis of the many heaps known at present. In this paper we study the performance of the very first tournament based complete binary tree. We focus on discrete-event simulation and our results show that this unknown predecessor of heaps can be a more efficient alternative to the fastest pending event set implementations reported in the literature.

We also extend the idea of binary tournaments to a $(2, L)$-tournament structure which exhibits the property of delaying the processing of events with larger timestamps whilst it keeps similar theoretical performance bounds to the native $(2, 1)$-structure or CBT. This property can be certainly useful in systems where many pending events are expected to be deleted or rescheduled during the simulation.

2 Tournament trees

The CBT and our realisation of it as a PQ is based on the idea of binary tournaments. Each item stored in the PQ consists of a priority value and an identifier. We associate each leaf of the CBT with one item, and use the internal nodes to maintain a continuous binary tournament among the items. A match, at internal node d, consists of determining the item with higher priority (lesser numerical value) between the two children of d and writing the identifier of the winner in d. The tournament is made up of a set of matches played in each internal node located in each path from the leaves to the root. See figure 1.a. Every time we change the priority associated with a leaf l, the tournament is updated by performing matches along the unique path between l and the root of the tree. See figure 1.b. We call this last operation *update-cbt*. The operations *extract-min*, *delete* and *insert* are implemented using *update-cbt* as the basic primitive.

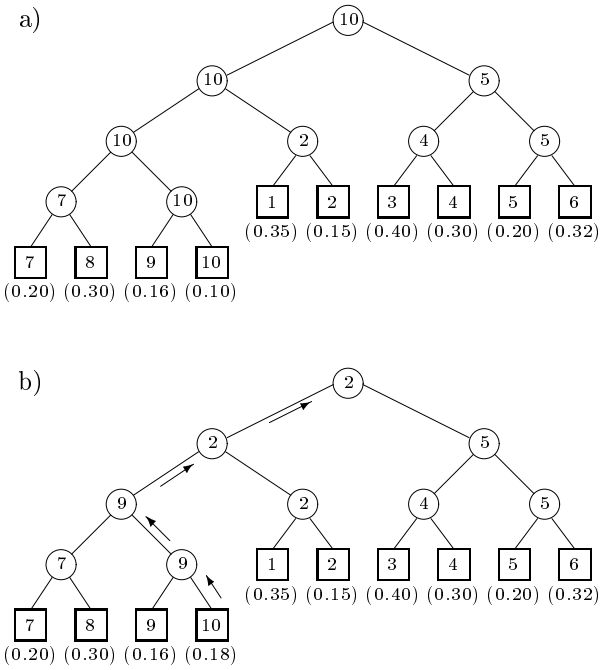

Figure 1: a) CBT for $N = 10$ (priority values in parentheses). b) CBT updated after changing the priority for item 10 to 0.18.

A PQ with N items is implemented using: (i) an array CBT$[1 \ldots 2N-1]$ of integers to maintain results of matches among items, (ii) an array Prio$[1..N]$ of priority values, and (iii) an array Leaf$[1..N]$ of integers to map between items and leaves. A node at position p in the array CBT has its children at positions $2p$ and $2p+1$. The parent of a node at p is at position $\lfloor \frac{p}{2} \rfloor$. All internal nodes are stored between positions 1 and $N-1$ of the CBT. The highest priority in the PQ is given by Prio[CBT[1]], its identifier is i=CBT[1], and its associated leaf is at position l=Leaf[i] of the CBT. [Note that it is not necessary to explicitly maintain the leaves of the tree in the array CBT since by using simple integer arithmetic on the array Prio we can calculate the priority associated with a given leaf]. Finally, to enable a dynamic reusing of item identifiers in the PQ, the array Leaf is also used to maintain a

single linked list of available item identifiers.

Deletions in the CBT are performed by removing the child with lower priority between the children of the parent of the rightmost leaf, and exchanging it with the target leaf to be deleted. For example, if we delete the item 4 in figure 1.b, then (i) the *father* of the leaf with item 10 becomes a leaf holding item 9 (no call to update-cbt is needed for this new leaf), (ii) the item 10 is moved to the leaf holding item 4, and (iii) the CBT is updated starting at this leaf (thus the right child of the root becomes CBT[3]=10).

On the other hand, insertions are performed by appending a new rightmost leaf and updating the CBT. This is done by expanding in two leaves the first leaf of the tree. Following the above example, if we now insert the priority 0.05 in the CBT, then (i) the leaf numerated with 9 becomes a new internal node of the tree with child leaves numerated 9 and 4 (we reuse identifier 4 to hold the priority 0.05), and (ii) the CBT is updated starting at position Leaf[4] (now the root of the CBT becomes CBT[1]=4).

The cost of each step of the *extract-min*, *delete* and *insert* operations is constant except by the cost of the *update-cbt* operation. In the following we analyse this cost, which in the worst case is obviously $O(\log N)$.

2.1 Average cost

The average cost of each *update-cbt* that reaches the root of the CBT can be calculated for $N > 2$ as follows. Some leaves are at the lowest level (level k) and the rest are one level up (level $k-1$). The cost of updating the tree from a leaf to the root is: (i) $\lfloor \log_2 N \rfloor + 1$ from level k and (ii) $\lfloor \log_2 N \rfloor$ 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 we obtain,

$$\ell(N) = \lfloor \log_2 N \rfloor + 2 - \frac{1}{N} 2^{\lfloor \log_2 N \rfloor + 1} .$$

However only for *extract-min* it is necessary to update the root of the tree. For *insert* the cost of *update-cbt* is constant. Assume $N = 2^k$. Starting from the leaves and going towards the root of the CBT we have that at the lowest level (i.e., the leaves) a total of N elements make 1 comparison, and from these N elements a total of $N/2$ are the winners. In the next level towards the root a total of $N/2$ elements make 1 comparison and the winners are now $N/4$, and so on. The average has the form

$$\ell^*(N) = \sum_{i=0}^{\log_2 N} \frac{1}{2^i} \;\leq\; 2 .$$

On the other hand, the variance can be calculated considering that a total of $N/2$ elements make just 1 comparison, $N/4$ make a total of 2 comparisons, and so on. This gives us

$$\sum_{i=1}^{\log_2 N} \frac{(\ell^* - i)^2}{2^i} \;\leq\; 2 .$$

Note that the constant factors involved are quite small since in the abstract sense each update step in the CBT takes just one comparison and one assignment, namely no additional work such as swaps in the implicit heap [1] or exchange of children in the skew heap [7] is made in each update step. Intuitively the cost of the *delete* operation is also $O(1)$ since it is equivalent to an insertion in the leaf associated with the item i to be deleted. However, the tournament as to be updated at the very least till the last internal node towards the root reached by item i. This effect is hard to estimate.

3 Empirical results

Using the Hold Model [5] we compared the running times of different PQs implementations with the CBT described in this paper. Basically the Hold Model is a random event generator which consists of inserting N items in the PQ and then performing a repetitive loop where in each cycle, called a hold operation, an *extract-min* followed by an *insert* is performed. The priority value (event timestamp) of each new item inserted in the PQ is computed as $k + X$ where k is the priority of the item most recently dequeued by *extract-min*, and X is a random priority increment sampled from a probability distribution. For X we used the exponential and triangular distributions [5]. We also included in this model a *delete* followed by an *insert*, sequence that occurs with a predetermined probability P_D after each sequence *extract-min/insert*.

Since each item deletion is immediately followed by an insertion, we implemented two versions of the CBT: With (CBT.b) and without (CBT.a) explicit deletion of items. Explicit item deletion (CBT.b) is the general method above described. In the second case (CBT.a), the operation *extract-min* retrieves the priority associated with a leaf l=CBT[1] and the same leaf l is used to host the next item to be inserted. Only one call to *update-cbt*(l) is performed in this case. The *delete* operation performs no computation and similar to the sequence *extract-min/insert* we assign the same leaf to the new item.

Empirical results for $P_D = 0.0$ and $P_D = 0.9$ are shown in figure 2. Note that the results are shown as the ratio between a PQ and the CBT without item deletions (CBT.a). These results show that CBT.b achieves better performance than heaps and, in some cases (triangular distribution), very fast special purpose queues like the calendar queue. In the

range $10^2 \leq N \leq 10^4$, CBT.b is clearly more efficient than heaps (which accounts for differing programming), and this better efficiency increases as N increases in most cases (i.e., the initial overhead involved in leaf management is amortised as N increases by a low cost tree updating). CBT.a achieves competitive performance with respect to the calendar queue for a wide range of values of $10 \leq N \leq 10^4$. We also observed that the better performance ratio of the calendar queue for exponential distribution remained fairly stable in about 0.7 for $1 \cdot 10^4 \leq N \leq 2 \cdot 10^4$. Although the comparison between CBT.a and the calendar queue is "biased", these results are useful to see the best possible performance for the CBT. This suggest the search for a CBT realisation with performance in the range [CBT.a, CBT.b]; hopefully with long periods of sequences of operations with performance near to CBT.a. An attempt in this direction is made in the following section.

4 $(2, L)$-Tournaments

To deal efficiently with large sequences of PQ operations where N is expected to remain fairly constant for long periods, and/or situations where rescheduling (and/or cancellation) of events are highly likely to occur, we can extend the basic idea of binary tournaments to a $(2, L)$-tournament structure as follows.

In each leaf of the CBT we maintain a set of $L = O(\log N)$ items, one of which is elected as the representative to participate in the continuous binary tournament maintained in the internal nodes. This election is made by doing $L - 1$ comparisons among the items stored in the leaf in order to determine the item with the highest local priority. We call this item *local minimum*. The internal nodes of the CBT are used to determine the item with the highest priority among these N/L local minima. The items stored in the leaves are maintained in unsorted double linked lists (one per leaf) so that local insertions and deletions take $O(1)$ time.

The average cost of an *update-cbt* up to the root of the CBT is now $\ell(N/L) \approx \log_2 N - \log_2 L$ whereas other calls to *update-cbt* should cost about $\ell^*(N/L) = O(1)$ (though similar effect to the above mentioned for deletions in the CBT.b must be considered in this cost). In the following discussion we are going to assume keys evenly distributed among the N/L lists and hold model with exponential distribution.

During an insertion, with probability $1/L$ the new item i coming to reside in the target leaf t has the highest local priority in the unsorted linked list associated with t (only one comparison is needed to know this since we can maintain the most recently elected item as the head of the list), and consequently the average cost of the insertion is at most $I = 1 + \ell/L = O(1)$ since it might be necessary to execute *update-cbt*, which in the worst case attain the root of the CBT. The worst case for an insertion is still $O(\ell)$. On the other hand, with probability $O(1/L)$ the deletion of an arbitrary item i produces a search for a new list representative and a call to *update-cbt*. Then the average cost for a deletion is at most $D = 1+\ell/L-2/L = O(1)$ with worst case $O(\ell + L)$. Finally, *extract-min* implies searching for a new representative in the list associated with Leaf[CBT[1]] and executing *update-cbt*, with a total cost $E = L - 2 + \ell = O(\log N)$ in the average and worst case.

Note that the assumptions made in the previous analysis are conservative since for insertions and deletions the probability of "hitting" a local minimum should be in practice smaller than $1/L$ given the increasing nature ($k \leq k + X$) of the new priority values being inserted in the queue, and deletions of items farther in the simulated time. In addition, for insertions and deletions we should expect more *update-cbt* operations with cost $\ell^* = O(1)$ than *update-cbt*s with cost $\ell = O(\log N)$.

To keep the theoretical average cost per operation on the $(2, L)$-structure similar to the one on the CBT, we can set $L = \lceil \frac{1}{2} \log_2 N \rceil$. Thus D and I are $O(1)$ whereas an evaluation of $E/\ell(N)$ with $E = L - 2 + \ell(N/L)$ for N ranging from 2^2 to 2^{20} gives $E/\ell(N) < 1.3$. Note that the average cost of *delete* on the $(2, L)$-structure is lower than the one on the CBT so this operation can amortise the ratio $E/\ell(N) < 1.3$. For example, using the sequence $H = 0.5\,(1 - P_D)\,E + 0.5\,P_D\,D + 0.5\,I(k + X)$ with exponential distribution for X, $10 \leq N \leq 10^4$, and $P_D = \{0.0, 0.25, 0.5, 0.75\}$ we have observed empirically $0.69 < H_{(2,L)}/H_{cbt} < 1.18$.

But some additional numbers are useful to see a key property of the $(2, L)$-structure. For example, when $N = 2^{10}$ and $L = 5$, we have that 4/5 of the items are maintained in what we can call the *overflow* area (i.e., the union of all the lists excluding local minima). When $N = 2^{15}$ we have 7/8 of the items in overflow area, and so on. Maintaining items in overflow area at low cost has clear advantages in simulations where many events are expected to be either deleted or rescheduled (or combinations of both) before taking place. The benefits are particularly evident when this occurs to events with larger timestamps. In these cases, the $(2, L)$-structure can be seen as a lazy structure that delays the processing of the events less likely to occur whilst it focuses on the more likely ones (i.e., the local minima). Thus the unsorted linked lists of size L are used as a first low cost buffering for a large number of items that could get their priorities changed or are simply deleted before being selected as the representatives of their lists. Note that we avoid any probability distribution dependence coming from the linked lists by distributing at random the items

in these lists; independently of their priority values (timestamps) as discussed below.

As we showed in previous section, the CBT is particularly efficient when no item deletions are performed in the tree and the number of items in it is maintained constant (CBT.a). For example, the operation *extract-min* retrieves the priority associated with a leaf l=CBT[1] and the same leaf l is used to host the next key to be inserted. Thus every two PQ operations, only *one* binary tournament is performed. This performance can be approached, for example, as follows.

In the $(2, L)$-structure we can double or halve the number n of leaves (lists) when N attain some threshold value. In this case, no item deletions are performed in the tree since we go up or down one level in the tree in each resize operation. For example, assuming a CBT with $n = 2^k$ leaves, and restricting L to be in $2 \leq L \leq 8$ we can double n as soon as N becomes equal to $8\,n = 2^{k+3}$, or halve n when N becomes equal to $2\,n = 2^{k+1}$. After a resize, the queue has $L = 4$ with n either 2^{k+1} or 2^{k-1}. A resize operation then occurs at least every $O(N)$ PQ operations, and the cost of each resize is $O(N)$ because no calls to *update-cbt* are needed for this process. Thus the amortized cost of resize operations is $O(1)$.

The items can be distributed randomly with a *hashing* function such as $h(k) = k \bmod n$ where k is an item identifier, n the current number of leaves, and $h(k)$ stands for the insertion list L_i with $i = h(k)$. Let us assume that the lists are maintained with a *local-min*(i) operation which determines the item with the highest priority in a list L_i and sets it as the head of L_i.

We can save calls to *local-min*/*update-cbt* by using a queue Z of pending *local-min*(i)/*update-cbt*(i) operations. In Z we enqueue list indentifiers i which can be positive or negative: $+i$ stands for *local-min*(i)+*update-cbt*(i) whereas $-i$ means *update-cbt*(i) only. The queue Z does not allow duplicated values, but any attempt to enqueue a value $+i$ which is already in Z as $-i$, makes it "positive" $+i$, whereas enqueuing $-i$ with $-i$ already in Z does not change $-i$. As we enumerate from 1 to n the leaves of the CBT, a simple array of flags (e.g. two bits per leaf) is enough to decide in $O(1)$ time whether or not a PQ operation has already enqueued a list identifier $\pm i$ in Z (these flags are reset as the requests enqueued in Z are served).

The use of Z is as follows. During an insertion of an item m if $+i$ is not already in Z, then we compare its priority value $m.x$ with the priority $L_i.x$ stored in the head of L_i with $i = h(m.k)$. If $m.x < L_i.x$ then we set m to be the head of L_i and enqueue $-i$ in Z. Otherwise we store m in the tail of L_i. On the other hand, the deletion of the head of a list L_i produces the enqueuing of $+i$ in Z. Finally, before extracting the item with the highest priority in the $(2, L)$-structure it is necessary to execute all the pending *local-min*(i)/*update-cbt*(i) enqueued in Z.

In general this organisation achieves efficiency near to CBT.a's in simulations where the number of pending events N is between 2^{k+1} and 2^{k+3} during long periods of the simulated time. Empirical results with the hold model show that the average overhead of the $(2, L)$-structure is about 20% of the total cost of CBT.a.

5 Conclusions

We have described an old idea of binary tournaments, embedded in a complete binary tree (CBT), along with algorithms to implement priority queues upon it. Our results show that this idea can be effectively used in the efficient implementation of the pending event set (PES) associated with discrete-event simulations. The structure keeps the heaps' attributes of simplicity and efficient behaviour under demanding simulations.

We have observed empirically that the CBT achieves better performance than heaps in the domain of discrete-event simulation. The average performance bounds of the CBT confirm its suitability for this task: $O(\log N)$ for *extract-min* with very low constant factors which can be amortised by dominating $O(1)$ operations like *insert* and *delete*. Other operations such as *find-min* and *increase-priority-value* are also $O(1)$.

The $(2, L)$-tournament structure proposed in this paper can be useful to amortise the cost of large sequences of operations since it provides an efficient way of dealing with an area of overflow in the queue. Implementations of this idea are multiple; we have presented just one instance.

Although we do not claim that the CBT is the best choice for all classes of simulations, in particular the CBT does not guarantee by itself the stability of priority values (i.e., identical priority values retrieved in "first come first serve" order from the queue), we believe this data structure represents a good alternative for many complex systems since it keeps the heaps' attributes of simplicity and independence of the event-time distribution whereas it achieves better performance than modern heaps and most of the specially devised PES implementations. Note that the basic form of CBT described in this paper was early presented in [6] as a first approximation to the description of the implicit heap [1]. However, to the best of our knowledge, this CBT has not deserved attention as a useful data structure in the literature on PQs and PES implementations. Note that the CBT can be implemented either as an static structure embebed in an linear array (like the presented in this paper) or as a dynamic structure with pointers to children and father

like most of the heap-ordered priority queues [5].

References

[1] J. Bentley. "Thanks, heaps". *Comm. ACM*, 28(3):245–250, 1985.

[2] R. Brown. "Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem". *Comm. ACM*, 31(10):1220–1227, Oct. 1988.

[3] M.L. Fredman, R. Sedgewick, D.D. Sleator, and R.E. Tarjan. "The pairing heap: A new form of self-adjusting heap". *Algorithmica*, 1:111–129, Mar. 1986.

[4] J.O. Henriksen. "An improved event list algorithm". In *1977 Winter Simulation Conference*, pages 547–557. IEEE Piscataway, N.J., 1977.

[5] D.W. Jones. "An empirical comparison of priority–queue and event–set implementations". *Comm. ACM*, 29(4):300–311, 1986.

[6] D. E. Knuth. *"The Art of Computer Programming, Vol. 3,* Sorting and Searching*"*. Addison-Wesley, Reading, Mass., 1973.

[7] D.D. Sleator and R.E. Tarjan. "Self adjusting heaps". *SIAM J. Comput.*, 15(1):52–69, Feb. 1986.

[8] J.T. Stasko and J.S. Vitter. "Pairing heaps: Experiments and analysis". *Comm. ACM*, 30(3):234–249, 1987.

[9] R.E. Tarjan and D.D. Sleator. "Self-adjusting binary search trees". *Journal of ACM*, 32(3):652–686, July 1985.

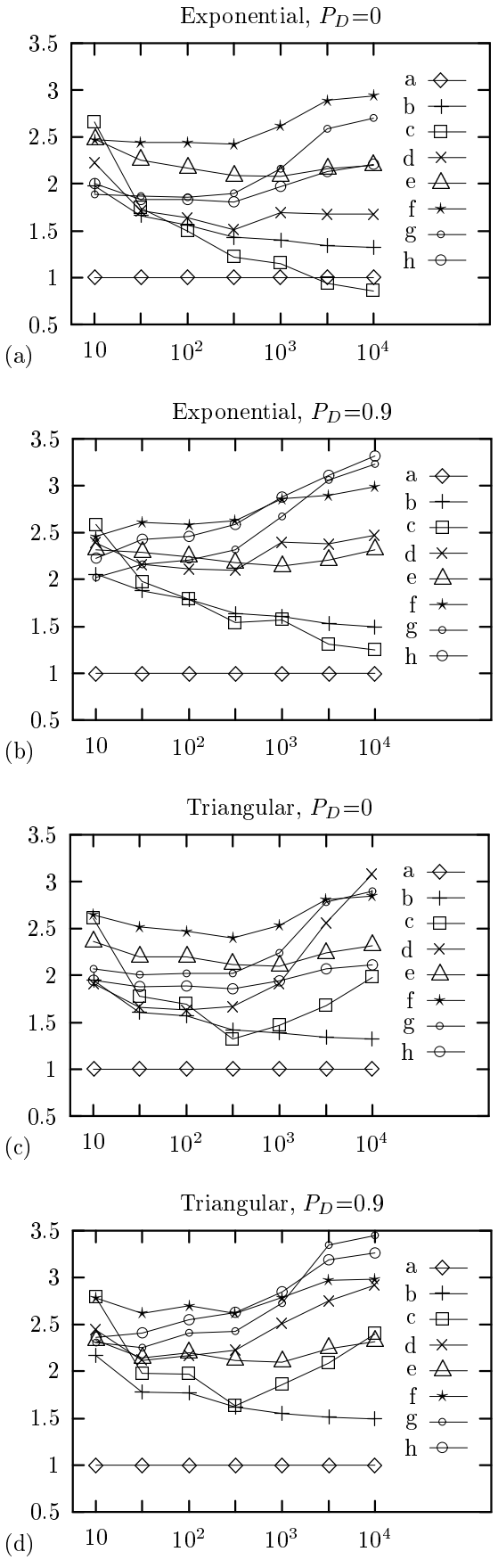

Figure 2: Total running time for 10^6 hold operations executed in an IBM/SP2 computer (using one dedicated processor and a micro-second resolution clock). Each curve shows the ratio (T_{PQ}/T_{CBT}) between the running time with a priority queue (PQ) and the running time with the complete binary tree without node deletions (CBT.a). In the graphics each PQ is identified by a letter: (a) Complete binary tree without item deletions, (b) Complete binary tree with item deletions, (c) Calendar queue [2], (d) Henriksen's queue [4], (e) Implicit heap [1], (f) Pairing heap with twopass variant [3, 8], (g) Skew heap with top-down variant [7], and (h) Splay tree with bottom-up splaying [9].