BSS Priority Queue

Leslie Murray
Universidad Nacional de Rosario,
Facultad de Ciencias Exactas, Ingeniería y Agrimensura,
Escuela de Ingeniería Electrónica
Rosario, Argentina, (2000)
leslie@eie.fceia.unr.edu.ar

and

Gerardo Rubino
IRISA/INRIA and ENST B,
Rennes, France, (35042-CEDEX)
Gerardo.Rubino@irisa.fr

and

María E. Urquhart
Universidad de la República, Facultad de Ingeniería,
Depto. Investigación Operativa, Instituto de Computación,
Montevideo, Uruguay, (11300)
urquhart@fing.edu.uy

Abstract

In Discrete Event Simulation the whole running time is mainly determined by the type of data structure intended to manage the pending event set. The Bounded Sequential Searching (BSS) Priority Queue is a pending event set implementation proposal for which empirical evidence of good performance is shown. BSS implementation is stable, its algorithms are easy to program, no resize operation is ever necessary and there's no sector devoted to the overflow. The construction mechanism and its resulting structure are as simple and clear as to let variations be attempted. Compared to the classical Implicit Heap under the Hold Model, a considerable difference in favor of BSS Priority Queue is experimentally shown. BSS main features are commented, the complexity of its associated algorithms are assessed and some implementation guidelines are given. Finally a list of open problems and current work is remarked.

Key Words: Discrete Event Simulation, Scheduling, Priority Queue, Sequential Searching, Data Structure.

1. Introduction

In Discrete Event Simulation (DES) events are represented by objects or notices capable to record, among other items, the time at which the associated event is to occur. After an event generation, due to the occurrence and evaluation of prior events, one or more notices may have to be inserted (scheduled) into a set from which they are to be removed later on. The Pending Event Set (PES) is the set of all generated but not yet evaluated events [11]. When the tasks associated to previously removed events are over, the simulator scans the PES in order to remove and evaluate the event with the least value of time. This mechanism performs a Priority Queue over the notices, where the time is the priority value [9]. The whole simulation running time is therefore strongly tied to the type of data structure used to hold the PES and, thereby, to the algorithm designed to manage it. A number of alternatives for this Priority Queue implementation has been proposed throughout the literature, none of them being considered the best [1, 2, 10, 12, 13, 14].

A sorted singly-linked list, where nodes are removed from one of the extremes and insertions are aided by Sequential Searching, performs this Priority Queue well enough, in spite of its simplicity (in fact, because of it). For models generating few events (fifty or less), this implementation is accepted to be the fastest. But for larger sized lists of events, the simplicity of its mechanisms becomes less important and solutions algorithmically smarter become faster, even though their initial overhead.

To improve singly-linked lists performance as a PES, one solution is to carry out Binary or Dichotomic Searching [6] rather than ordinary Sequential Searching. This makes necessary to address the notices inside the structure somehow, otherwise the only way to move from one to another is by going through every one in between. The set of positions that Binary Searching needs to know follows an iterative pattern, such as the center, and the center of the segments determined by splitting the list at the center, and so on. In ordinary implementations (by languages like C or Pascal) it is not hard to keep a pointer to the middle of the list. To keep two more pointers to the center of the segments determined in the list by the central pointer, demands more work. Attempting to raise the number of pointers up to a higher number leads to complex algorithms, and even more complex if the number of pointers is to be self-adjusted to the size of the list.

Binary Searching is a particular case of a more general method in which the list is firstly divided into several segments of the same size (two or more). Afterwards, the segment where the searched site is known to be, is divided again into as many segments as it was the whole list the first time. Proceeding this way, searching success is guaranteed after no more than $log_B N^{B-1}$ notices are checked ($log_2 N$ for Binary Searching), where N is the size of the list and B the number of segments in which the list or any of its subdivisions is split in every time. **FIGURE 1** depicts the Searching Complexity for this mechanism and shows that, algorithmically, the fastest option is the one in which $B=2$.

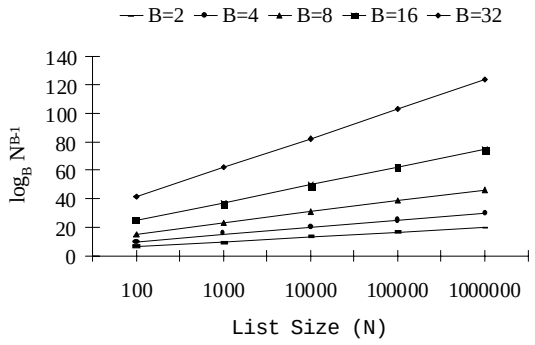


FIGURE 1. Searching Complexity

This report attempts to reach a good efficiency level with a list-based implementation, but for larger sized Queues. Our proposal searching method also attempts to successively divide the segment where the searched site is known to be, but the pattern it follows is different than the former. To carry out this process an auxiliary supporting substructure is added to the notices list. Section **2** explains a mechanism to build such a substructure. Sections **3** and **4** comments the methods main properties. Empirical results shown in Sections **5**, **6**, **7** and **8** prove that this implementation proposal satisfactorily works out events management in DES scheduling for a wide range of events Queue sizes. The algorithms complexity are assessed in Section **9**, some implementation guidelines are given in Section **10** and our conclusions are issued in Section **11**.

2. Bounded Sequential Searching

In a simple (Sequential Searching aided) sorted singly-linked list implementation of a Priority Queue the hardest work is done whenever new events need to be scheduled. When *N* notices are already queued, searching success for the insertion site can only be guaranteed after *N* nodes are checked. Let's set a bound to this process by stating that **"no search should have to traverse through more than *B* nodes"**, being *B* a fixed value. If a new insertion is required after the Queue has reached the size *B*, a supporting substructure, to avoid the traverse of the resulting *B+1* nodes in the next insertions, is added. The mechanism to build such a substructure begins by the insertion of an auxiliary node in a parallel auxiliary queue, placed (linked) above the events Queue (see **FIGURE 2**).

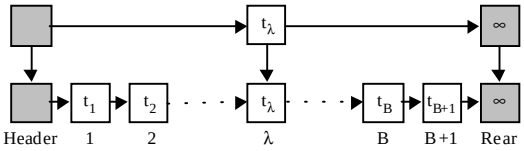


FIGURE 2. First Auxiliary Node Insertion

The auxiliary node points to the event located in position λ in the events Queue, being $\lambda=ceiling((B+1)/2)$, and copies to itself the time of the pointed event, t_λ. Thereafter, the entrance to start further searches is not the Header of the events Queue anymore, but the Header of the just added supporting queue instead. The next insertion will seek the site to place a new event by firstly seeking the corresponding site in the just created auxiliary queue. This determination is achieved by firstly comparing the time of the new event to t_λ, and eventually to the time of the auxiliary queue Rear, which is ∞. Afterwards, searching should continue in the events Queue, either from the Header or from the event pointed by the auxiliary node, according to the former comparisons. If no removal takes place, the process continues until further insertions make one of the two segments determined in the events Queue reach the size (span) *B+1*. Then a new auxiliary node is properly inserted in the corresponding place of the parallel queue, following the same mechanism than before. Thereafter searches still start from the Header of the new supporting auxiliary queue, but now up to three comparisons must be made in order to decide the event from which searching must continue in the lowest level Queue. The supporting auxiliary queue may then grow until its size becomes *B+1*. When this happens a new parallel supporting queue is added above the existing one and, consequently, the starting point for further searches is moved up to the Header of the lastly added queue.

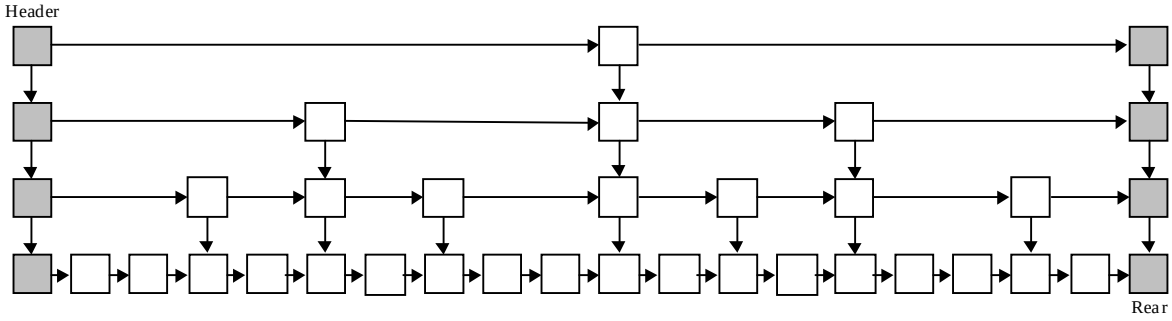


FIGURE 3. BSS Construction (B=2)

The supporting mechanism is made out of a number of parallel queues, each one of them in a different level, such as all the nodes of a certain level queue (including Header and Rear) point to some other in the level below, but not vice versa. Looking at the structure that is being built (see **FIGURE 3**), the prior statement can now be rewritten into **"no search will have to traverse through more than *B* nodes in any of the queues"**. The cost of Sequential Searching is Bounded by the value of *B+1* in every level of the resulting structure, because *B+1* is actually the maximum number of nodes that will have to be checked in every queue. Thereby the name of "Bounded Sequential Searching (BSS) Priority Queue" given to the structure.

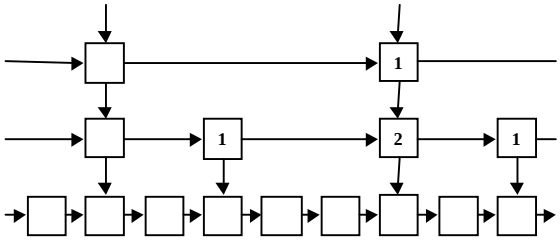
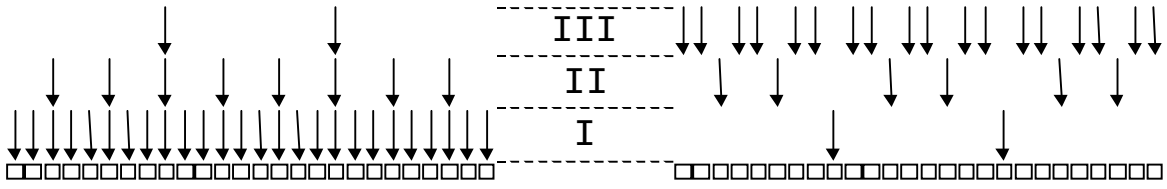


FIGURE 4. BSS Span Determination

As shown, the way the list or any of the resultant segments are divided, always forces *k+1* comparisons whenever segments of *k* nodes need to be traversed through. In real implementations the segment determination and the corresponding span indication is suggested to be done as shown in **FIGURE 4**.
Even under no explicit equivalence between nodes and links, BSS outline suggests the existence of an underlying m-ary tree [5]. The number of BSS nodes is higher, and some of the m-ary tree links corresponds to BSS paths. Nevertheless, both structures arrange nodes under similar patterns and perform searches similarly.

3. Searching Algorithms Comparison

The similarities and, in a sense, the equivalence between BSS and the former mechanism of successive subdivisions, is now approached in more detail. The scheme under which BSS set auxiliary nodes, and therefore determines the segments span, depends upon the time of the arriving notices, reason why even when bounded, all span values are, in general, different. In spite of being a "very" particular case, the snapshot of **FIGURE 5**(a) captures the evolution of some BSS Priority Queue, for *B=4*, and represents a useful example for our current purpose. Level I corresponds to the event notices queue, whose creation occurs in first term. Successively, levels II and III completes the structure as the number of events grows higher.



(a) BSS (b) Succ. Div.

FIGURE 5. Algorithms Comparison

On the other hand, **FIGURE 5**(b) outlines a pointers distribution aimed to accomplish insertions by successive subdivisions (by *3*) of a sorted list. Level I shows the first stage, where the queue is divided into three segments of the same size. The largest notice of the lowest two segments is then checked, being determined which one of the resultant three hold the site to allocate a new notice (in **FIGURE 5**(b) all segments are shown to be divided, but actually only the successful one needs to be). Segments are divided iteratively until no further division is possible, what in the present example occurs when the last three segments are composed by only one notice (level III).
Both mechanisms undertake the dividing process in opposite directions. BSS starts from the smallest and evolves up to the largest segment, while the other method does it from largest to smallest. Nevertheless there is an underlying relation between them, even numerical. In the example the BSS (particular) configuration, for *B=4*, provides the same searching potential as the method that keeps dividing the list by *3*. Furthermore, the method that successively divides the list by *k*, finds an equivalence in some BSS configuration for *B=2(k-1)*. But the most interesting result can be obtained from a visual review of **FIGURES 5** (a) and (b). In fact, ultimately, BSS construction proceeds, in any level, by inserting "*1* auxiliary node every *B-1* in the level below". In the example it does it like "*1* auxiliary node every *3* in the level below". As a consequence it finally gets to split the whole queue into *3* parts (in level III). So there is a tight connection between the counting of nodes to "build" segments in the lowest level and the number by which the whole queue is finally divided in the highest level. And this fact makes explicit the way both methods are related.
The size of the queue under analysis was selected on purpose, and the BSS structure only adjusts itself exactly to the shown pattern as a particular case. Anyway the example shows that in a sense, both models performs similarly and that, somehow, BSS methods behavior tends to the other.

4. BSS Features

As BSS was designed to perform DES scheduling, regardless of special cases, removes will always occur on the first node, the earliest. Only the presence of auxiliary nodes linked above the event to be removed will make a difference between this operation and ordinary sorted singly-linked list removals. The deletion of a whole "column of first nodes" only implies computational effort, but as it never generates segments larger than *B* in any of the queues, no logical process needs to be performed afterwards. Thus, being *s* the span of every segment, only for the first one in every level (the most to the left) holds the relation $0 \le s \le B$, while for all the rest, $(B/2) \le s \le B$.
If a long sequence of insertions is executed, being the time set to every event taken from a uniform distribution, the span of all generated segments will follow a distribution in which most of them will have the value *(B/2)*. The next, in decreasing proportion, will be *(B/2)+1*, *(B/2)+2*, ..., *B* (see **FIGURE 6**, experimentally obtained for the case *B=16*). But if, rather than taken from a uniform distribution, the times for the events to be inserted were generated by an algorithm like $t=t_{li}+r$, being t_{li} the time of the last inserted node, and *r* always the same positive value, all

spans will converge to *(B/2)*. None of these tested evolutions may ever occur in real applications, but they both give a clear idea about the way the construction of the structure actually progresses, and likewise of the relation between the average span of the resulting segments and the incoming events temporal distribution.

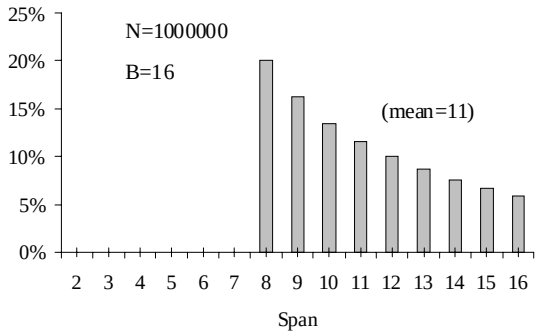


FIGURE 6. BSS Span Distribution

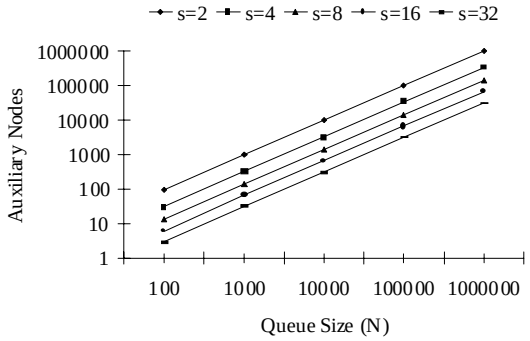


FIGURE 7. BSS Supporting Structure

FIGURE 7 shows the size of the whole supporting auxiliary structure, measured by the number of auxiliary nodes, under the hypothetical fact that all spans *s* were fixed at the same value.
If used for different purposes, in which deletions can take place on any of the nodes, BSS removals might have to be followed by an insertion to re-establish the relation between segments span *s* and the bound *B*, but only if the removed node was pointed by one or more auxiliary nodes. On the other hand, auxiliary nodes may have to be removed whenever internal removals make the number of them be more than enough. This problem would increase the running time, however it is not frequent in DES over ordinary models.

5. BSS Filling

In real simulations, insertions and removals never occur one absolutely isolated from the other. Nevertheless, to try a scheduler under tests where only insertions or only removals occur, but not both, might be enlightening and help to obtain further conclusions about its behavior.

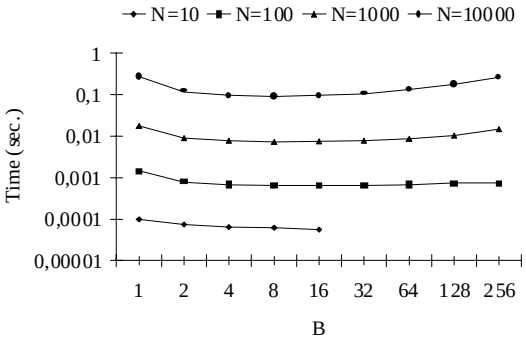


FIGURE 8. BSS Filling Time

If only insertions occur, for a fixed value of *B* the insertion time should grow logarithmically with the size of the Queue. This fact was outlined, by the analysis of a similar method in **FIGURE 1**, and will be approached in more detail in Section **9**. Some insertions may carry out additional insertions (in higher levels) while some other may not, varying the number of auxiliary insertions between zero and the number of auxiliary queues already in the structure plus one. This number depends on the span of the segments through which insertions may progress, but neither on the size, nor on any sort of threshold in the number of events already queued. This is a considerably important BSS feature, because no response "jumps" are ever expected.
Even when easy to manage (because of the low number of auxiliary queues and nodes) large values of *B* will make the BSS mechanism tend to disappear, and the whole implementation behave just as a sorted singly-linked list, increasing therefore the insertion time. Conversely, as *B* tends to *2*, the mechanism tends to behave quite like the Binary Searching [6], i.e. the fastest option (even when possible, *B=1* will be proved to lead to an impractical configuration), but an additional fact arises then, low values of *B* result in a high number of auxiliary queues and nodes, with their associated added computational effort. These are the reasons why the best performances are (experimentally) found to happen for values of *B* between *4* and *16*, for a wide range of Queue sizes (see **FIGURE 8**). The number of trials should be as large as to let all kind of insertions occur, in order to arrive to results statistically reliable.

6. BSS Empty

Concerning the number of removals to try in an "only remove" test, the criterion is now to make sure that columns of every existing height are to be deleted. For a fixed value of *N*, removals are faster as the value of *B* grows higher because the number of auxiliary queues, and therefore the density of auxiliary nodes, is inversely related to the value of *B*. As *B* tends to *N*, removing time tends to the values of the sorted singly-linked list implementation, with no auxiliary supporting structure above (see **FIGURE 9**).

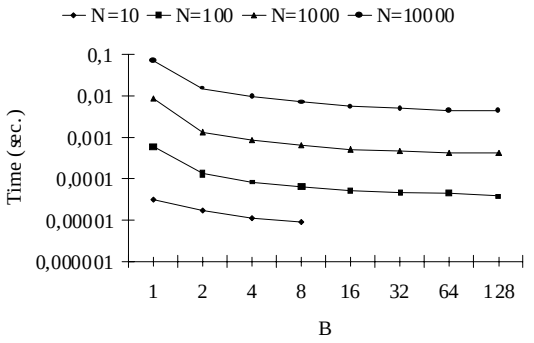


FIGURE 9. BSS Empty Time

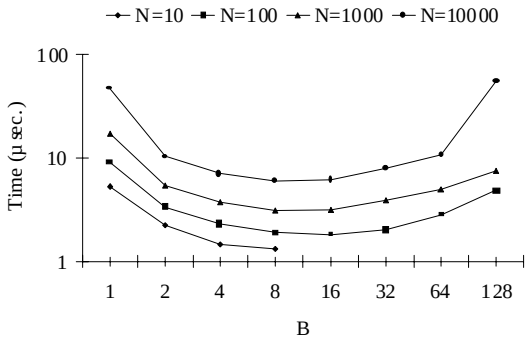


FIGURE 10. BSS Hold Time

7. BSS Hold

The Hold Model [4, 8] is the most accepted to test a DES scheduler. It performs "long" sequences of one removal followed by one insertion where: a) The scheduler must be sufficiently full and, b) The time set to the inserted event is obtained by addition of a random value to the time read in the just removed event. As a consequence the number of events is kept constant throughout the test.

The interpretation of the Hold tests can be aided by the "only remove" and "only insert" tests done so far. Actually, the main effects of both tests interact, making the Hold time grow for low values of *B*, because of the cost introduced by managing a high number of auxiliary nodes, and also grow for high values of *B*, because a low number of auxiliary nodes indeed result (algorithmically) in higher searching times. **FIGURE 10** shows these two facts for different values of *N*.

8. BSS vs. Other Priority Queue Implementations

Again under the Hold Model, BSS is now confronted to different implementations of DES scheduler. Linear is the simplest, performing only Sequential Searching over a sorted simply-linked list. It is the fastest as long a the number of events is kept very low but it deteriorates as the size of the Queue grows higher.

Implicit Heap is one of the oldest implementation based on a heap-ordered tree [6]. In the tested implementation nodes are allocated into an array and the traverse through the structure is carried out by simple calculation on the array index value. After insertions or removals the heap property is reestablished by swapping nodes, just like in the heapsort sorting algorithm. The value of *B* has been set to *8* for all BSS trials. Two different ranges of Queue sizes were tested. **FIGURES 11** and **12** show the results.

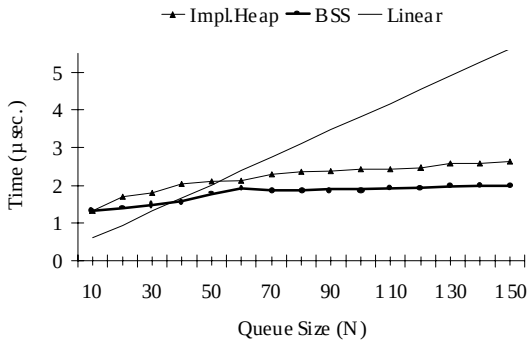


FIGURE 11. Hold Times (I)

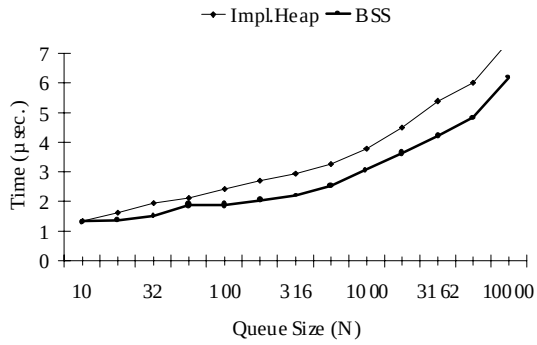


FIGURE 12. Hold Times (II)

All tests measurements reported in Sections **5**., **6**., **7** and **8**. were performed in a Pentium-MMX PC with 32 Mb of RAM and a 166 MHz clock, running the operating system Linux (Kernel v2.0.36). The programming language was C, and the compiler was gcc v2.7. Neither the "only insert", nor the "only remove" tests report the "single operation time", but the whole time taken to either completely fill or empty a BSS Priority Queue instead. On the other hand all the Hold operations reported (and the removal time tests of Section **9**) show the average "single operation time". Sequences of 10^5 "only insertions" and "only removals" were timed (using a random time for the notices). For the Hold operations the BSS Priority Queue was (randomly) filled up to the desired size, then a sequence of 10^6 Hold operations was performed to let it reach the steady state and finally another 10^6 operations were measured and averaged. The time that algorithms spends in tasks not directly related to insertions or removals has been subtracted. All random numbers used were taken from the Exponential distribution, obtained as *-log(drand48())*.

9. BSS Complexity

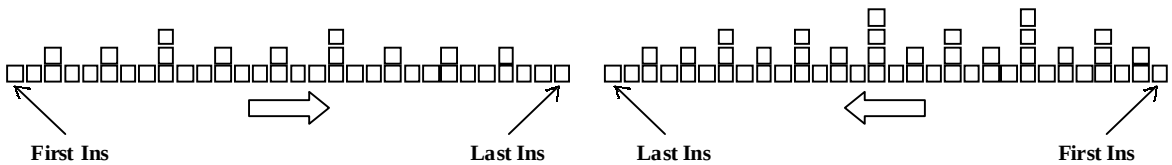
Let a BSS Priority Queue be formed by h queues, numbered bottom-up from *1* to h, each one of them containing N_i nodes, $i=1,2,3,...,h$, so that $N_1=N$ is the total number of event nodes. Accepting that all nodes, excepting the event ones and all the Headers, hold a span value s_{ij}, where i denotes the queue and j the position in the corresponding queue, the average span value is s:

$$s = \left(\sum_{i=2}^{h} \sum_{j=1}^{N_i+1} s_{ij} \right) / \sum_{i=2}^{h} (N_i + 1)$$

In order to estimate its height, BSS structure will be thought as to have the average span value, s, for all segments.
Let n_i be the number of nodes whose value of t might need to be compared to the one of the node to be inserted, in every level queue. So that, being N the total number of nodes in the events queue:

$$n_1 = N+1$$
$$n_2 = n_1/(s+1) = (N+1)/(s+1)$$
$$n_3 = n_2/(s+1) = n_1/(s+1)^2 = (N+1)/(s+1)^2$$
$$\vdots$$
$$n_h = n_{h-1}/(s+1) = n_1/(s+1)^{h-1} = (N+1)/(s+1)^{h-1}$$

The highest level queue may be thought as to hold as many nodes as any of the segments, i.e. s, and being the Rear also susceptible to be checked, $(s+1)$ should be accepted for n_h. It then follows that $(N+1) = (s+1)^h$ and therefore $h = \log_{s+1}(N+1)$, that for large values of N goes to $h = \log_{s+1} N$.
As there is no explicit bound on BSS height, to beware of the case $s \to 0$, in which the structure would grow unboundedly, the determination of h deserves further analysis.
For any given N, the maximum possible value of h can be reached if and only if all notices arrive in a proper sequence. To let such a sequence be effective, once the first $B+1$ notices have arrived, and the first subdivision has occurred, all the next incoming notices must fall into the same segment, until they fill it completely. Afterwards, the next incoming notices must also fall into the same segment, until it gets filled, and so on. The key of the process, is that not any segment is equally useful to allocate sets of consecutive incoming notices, it is necessary that the auxiliary nodes consecutively promoted to higher levels also keep falling (while there is still room for this) into the same segment. And the same criterion holds for all the following levels. If the arriving notices were scattered all over the existing segments, the sequence would not be effective in the sense of letting the structure grow up to its maximum possible height. Since removals only shorten the segments span, sequences only composed by insertions will be considered. Two possibilities for an effective sequence will be introduced, one of which will be shown to be the best.



(a) Increasing Time Sequence B=3 (b) Decreasing Time Sequence B=3

FIGURE 13. Most Effective Growing Sequences

The first one is composed by notices with always increasing time, so every new notice is appended to the corresponding queue, in every level. **FIGURE 13** (a) shows the case for $B=3$. Conversely, the second possibility is made out of notices with a decreasing value of time, being therefore successively added to the top of the

corresponding queue. For the case of *B=3*, the whole structure evolves as shown in **FIGURE 13** (b). The result is clear in favor of option (b), and the reason is that, as the segments are divided by the function *ceiling((B+1)/2)*, if *B* is odd, after every subdivision the left segment is always larger than the right one, and to keep inserting into the largest segment makes the limit of *B* be sooner overshooted. When *B* is even, to append notices or to add them to the top makes no difference. In a sense to always add notices to the top is a sort of idealization, because the value of time would rarely be indefinitely decreasing for real systems (it could only be so for certain periods). However, as the only purpose is to determine a bound for the value of *h*, this sequence is still worth considering. If segments were divided by function *floor()* rather than *ceiling()*, the most effective sequence would have been the one that appends notices with an increasing value of time.
The election of the "most effective" arriving sequence (the one that keeps adding to the top) now let us infer the number of notices in any level, as a function of the number in the level below. So that:

$N_{i+1} = (N_i - ((B/2)+1)) / ((B/2)+1)$ $\quad i=1,2,...,h-1$ if *B* is odd.
$N_{i+1} = (N_i - (B/2)) / ((B/2)+1)$ $\quad i=1,2,...,h-1$ if *B* is even.

Where / denotes the integer division. Let's have a look at them for some values of *B*:

$N_{i+1} = (N_i - 1)$ $\quad B=1$
$N_{i+1} = (N_i - 1) / 2$ $\quad B=2$
$N_{i+1} = (N_i - 2) / 2$ $\quad B=3$
$N_{i+1} = (N_i - 2) / 3$ $\quad B=4$

These functions, particularly the first one, help to make evident a critical case. If *B=1* the structure grows as shown in **FIGURE 14**, the number of notices in every auxiliary level is obtained by subtracting *1* to the number in the level below.

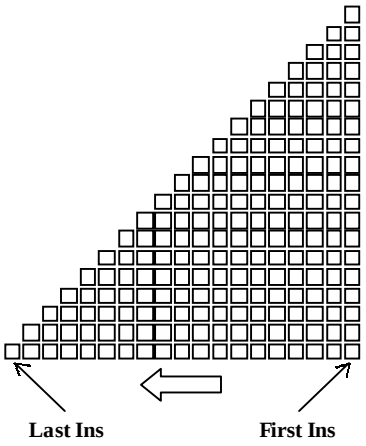


FIGURE 14. Decreasing Time Sequences B=1

The consequence is an ***O(N)*** searching cost. To approach an ***O(log N)*** complexity it is necessary that the size of every queue results from dividing the size of the queue underneath, by some factor greater than one, what for the current sequence is shown to happen for *B>1*. So an important recommendation arises in the sense of discarding the value of *B=1* whenever the worst case insertion cost is attempted to be kept better than ***O(N)*** (furthermore *1* was experimentally proved to be an inefficient value of *B*).

As *N* grows higher

$$(N_i - ((B/2)+1)) \approx (N_i - (B/2)) \approx N_i$$

The simplification only make sense for *B>1*, because for *B=1* it would turn $N_{i+1}=N_i-1$ into $N_{i+1}=N_i$, i.e. an unbounded growing rule. The simplification does not seem to be valid in higher levels either, because there N_i is in the order of *B*. However its application leads to a larger number of nodes in the corresponding level, and being the current purpose to determine a bound for the height of the structure, it can be accepted. Anyway, from now on *B* will be considered greater than *1*. Thereafter (i.e. upon the simplification),

$N_{i+1} = N_i / ((B/2)+1)$ $\quad i=1,2,...,h-1$ if *B* is either odd or even and *B>1*.

Then the structure grows like,

$$N_1 = N$$
$$N_2 = N_1 / ((B/2)+1) = N / ((B/2)+1)$$
$$N_3 = N_2 / ((B/2)+1) = N / ((B/2)+1)^2$$
$$\vdots$$
$$\vdots$$
$$N_H = N_{H-1} / ((B/2)+1) = N / ((B/2)+1)^{H-1}$$

To accept H as the highest occupied level of the structure is just like to accept that no element is allocated in level $H+1$. Then the value of H for which $H+1$ allocates at least one element is in fact a bound for the height of a BSS structure, filled up under the "most effective" sequence, with N nodes, and $B>1$.

$$N_{H+1} = N_H / ((B/2)+1) = N / ((B/2)+1)^H = 1 \quad \rightarrow \quad H = log_{((B/2)+1)}\, N$$

Comparing the average height h to the just determined bound H (i.e. $h=log_{s+1}\, N$ and $H=log_{((B/2)+1)}\, N$) an interesting observation comes up, in the sense of ratifying a fact already pointed in Section **4**: whenever $B>1$, the value of s will never be less than $B/2$. But the most important remark concerns the fact that, if $B>1$, the average and the bound of a BSS Priority Queue height, evolve logarithmically on the number of notices queued. **TABLE 1** compares experimental results to the analytical determinations of height. For every pair of N and B shown, h and s are determined running a program. The real value of h is an integer that moves by steps of 1 unit, what makes it hard to follow by continuous functions like $log_{s+1}\,(N+1)$ and $log_{((B/2)+1)}\, N$. Anyway the results obtained are satisfactorily adjusted to the real ones.

N	B	s	h	$log_{(s+1)}\,(N+1)$	$log_{((B/2)+1)}\, N$
1000	2	1.338	8	8.135	9.966
1000	4	2.678	6	5.305	6.288
1000	8	5.424	4	3.714	4.292
1000	16	10.564	3	2.822	3.144
10000	2	1.338	11	10.845	13.288
10000	4	2.677	7	7.074	8.384
10000	8	5.532	5	4.908	5.723
10000	16	11.217	4	3.680	4.192

TABLE 1. BSS Height Analysis

Throughout the analysis just carried out, the condition of $B<<N$ was implicitly accepted, because as $B \rightarrow N$, then $h \rightarrow 0$, and the structure is transformed into a sorted singly-linked list, with no BSS mechanism above.
After the height bound analysis, and under knowledge of its average value, the complexity of insertion is now undertaken. The process always starts by seeking the site to allocate the new node, for which a number of comparisons has to be done in every level. According to the BSS structure definition, searches will check a number of nodes between 1 and $B+1$ in every queue, just because all segments span a value between 0 and B. For a certain number of searches this number will average a value s'. Then:

$$C = s' \cdot h = s' \cdot log_{s+1}\,(N+1) \qquad 1 \leq s' \leq B+1$$

An $\boldsymbol{O}(log\ N)$ behavior is then proved for the average insertion cost. Best case occurs whenever the searching path has to make only one comparison per level, while the worst case corresponds to a path in which $B+1$ comparisons in every queue are necessary. The height of the structure is still h, independently of the path taken by searches. Best and worst case, out of a long sequence of insertions, will then be determined by $log_{s+1}\,(N+1)$ and $(B+1) \cdot log_{s+1}\,(N+1)$ respectively, what is to say that both are also $\boldsymbol{O}(log\ N)$.
The key to assess removals complexity is the height of the column of nodes to be withdrawn in each removal operation. These columns includes one event notice plus an additional number of auxiliary nodes, that may vary from 0 to $(h-1)$. Even though Rears are not removable nodes, in order to approach removal complexity analysis, to consider them as if they were, reports an important simplification. Rear columns perfectly look as the result of an insertion, and, to pretend that they are nodes to be removed only add one more of the most expensive removals. Under this assumption, a maximum of $n_1=(N+1)$ consecutive removals can be performed any time. Out of those, only (n_1-n_2) corresponds to columns of height 1, (n_2-n_3) to columns of height 2, and so on. The average height of the removed column will then be R:

$$R = [1\cdot(n_1-n_2)+2\cdot(n_2-n_3)+ \ldots +(h-1)\cdot(n_{h-1}-n_h)+h\cdot(n_h)] / n_1$$
$$R = [n_1+n_2+n_3+ \ldots + n_{h-1}+n_h] / n_1$$
$$R = [1+(s+1)^{-1}+(s+1)^{-2}+\ldots+(s+1)^{-(h-1)}+(s+1)^{-h}] = 1 + s^{-1} - (s+1)^{-h}$$

The value of *h* grows together with *N*, so that:

$$lim_{N\to\infty} R = lim_{h\to\infty} R = (s+1)/s$$

It then follows that the average removal time is ***O(1)***. So in a long sequence of removals, the average height of the column to be withdrawn does not depend upon the Queue size. This fact is shown in **FIGURE 15** for a wide range of queue sizes (a scope wider than in the rest of the experiments reported was tested just to show that as the size of the queue grows larger, removal cost does not depend on the queue size).
It's clear though that $R_{max} = h = log_{s+1}(N+1)$, meaning that, out of a long sequence of removals, worst case is ***O(log N)***.
In a real implementation a different amount of work has to be done to remove event notices nodes than to remove auxiliary nodes, because:

1) Auxiliary nodes to be removed have to be detected (by unavoidable comparisons in a bottom-up traverse of the column).
2) Auxiliary nodes ought to be deallocated, while event notices nodes are simply returned to the simulator.

This is the reason why the work to withdraw a column of nodes will not be proportional to its height. Let's pretend then that the cost to remove an auxiliary node is still *1* unit, and the corresponding cost to remove an event node is δ (accepted $\delta<1$). The initial expression of *R* will then be transformed into:

$$R = [\delta\cdot(n_1-n_2)+(\delta+1)\cdot(n_2-n_3)+ \ldots +(\delta+h-2)\cdot(n_{h-1}-n_h)+(\delta+h-1)\cdot(n_h)] / n_1$$

And consequently the final expression *(s+1)/s* transformed into *(δs+1)/s*, but clearly the character of ***O(1)*** will not be affected.

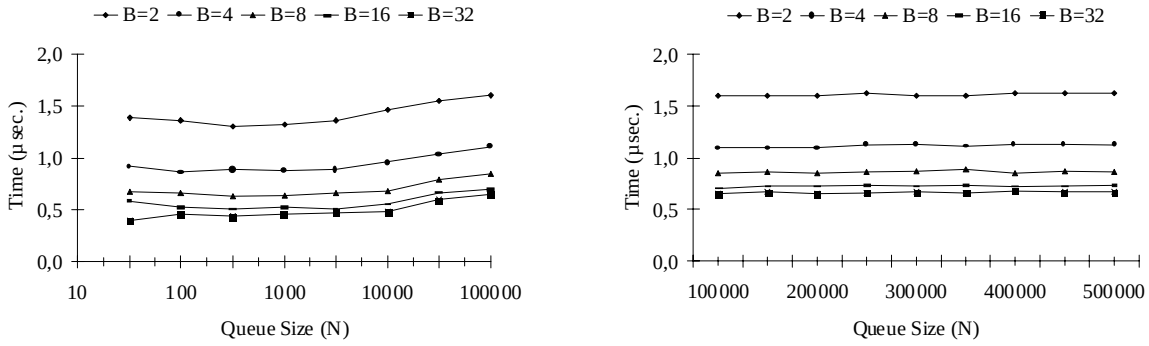


FIGURE 15. BSS Removal Single Time

10. BSS Implementation Guidelines

The BSS Priority Queue structure was developed in the C-programming-language. Some features of a program that supported the present report are commented in this Section.

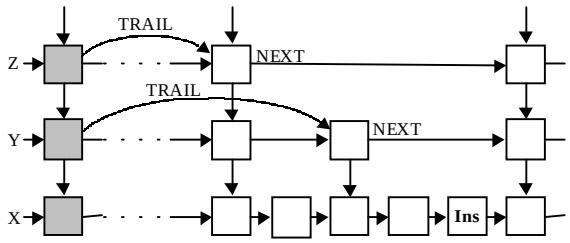


FIGURE 16. BSS Insertion Process

Most of the routines are extremely related to Sequential Searching and need no further analysis or comments. A process that is worth remarking, because it's critical to obtain a good efficiency level, is the insertion of auxiliary nodes and queues.

Whenever a new event insertion is required, searches are launched from the highest auxiliary header and a Sequential Searching is carried out in every level. While searching progresses, a pointer from every auxiliary header, called TRAIL, must be set to keep track of the last node whose time was not higher than the time of the node to be inserted. After the insertion process has reached the lowest level, the set of nodes pointed by TRAIL is composed by all the left extremes of the segments to which a new auxiliary node may have to eventually be added. A maximum of three level queues are involved in the insertion process every time. To clarify this, let's see **FIGURE 16** where the headers of the mentioned three queues are pointed by pointers X, Y and Z. Level X (the lowest) will always exist since, in the worst case, the events level does exist, empty at least. So, X is the level in which an insertion has actually occurred (either of an event notice, or an auxiliary node). Y and Z may or may not exist, and the role of each one of them is: for Y to eventually allocate a new auxiliary node and for Z to just modify the corresponding span in case the insertion of the auxiliary node would have occurred in Y. **FIGURE 17** shows an algorithm suitable to perform this process.

```
Y = X→UP;
if(Y){
    ++(Y→TRAIL→NEXT→SPAN);
    Z = Y→UP;
    while(Z){
        if(Y→TRAIL→NEXT→SPAN > B) add_XYZ;
        else return;
        X = X→UP;
        Y = Y→UP;
        Z = Z→UP;
    }
    if(Y→TRAIL→NEXT→SPAN > B) add_XY;
    else return;
    X = X→UP;
}
if(X→SIZE > B) add_X;
return;
```

FIGURE 17. BSS Insertion Algorithm

The program enters this code right after an event notice has been inserted in the lowest level Queue and, consequently, after all TRAIL pointers have been set. Then, it attempts to climb up through the Headers UP links, and does it until it can't go higher. Functions add_XYZ, add_XY and add_X (for which their arguments have been omitted) takes care of the mechanism of an auxiliary node insertion in level Y (also the creation of Y by add_X in case it would be necessary). The algorithm keeps checking the resulting SPAN of the segments that's been just updated in the prior iteration, and each one of the three functions performs the insertion under knowledge of the existing levels, so as add_XYZ knows that after the growth of X, a new auxiliary node will be placed in Y, existing Z above it. And so on.

11. Conclusions

As a good balance of efficiency and implementation simplicity, BSS Priority Queue can perfectly fit the needs of a DES general purpose scheduler. If searching comparisons are properly coded, the implementation is stable, i.e. events with the same time value are retrieved in FIFO order. The fact of having a well bounded worst case for sequences of insertions and removals (better than ***O****(N)*) makes BSS also suitable for real-time systems scheduling applications.

Even though it requires the set of *B* as an operative parameter, experimental results show that there is a wide range of Queue sizes for which *B* can be fixed to the same value, without losing efficiency. In a sense, this reminds of the well-known Henriksen's structure [4] and its "maximum sublist size" parameter, usually set to *4*.

BSS algorithms are easy to program. No resize operation is ever necessary and there's no sector devoted to the overflow [2]. The construction mechanism and its resulting structure are as simple and clear as to let variations be attempted.

The considerable difference in favor of BSS Priority Queue, compared to the classical Implicit Heap (**FIGURES 10**, **11**), has encouraged us to extend the scope of experiments and tests. Thus we are currently working on: 1) Comparison to other implementations under different events temporal distributions and larger sized Queues; 2) Implementation of other Priority Queue operations, like deletion and time modification of any notice already queued; 3) Construction using different types of node, attempting to distinguish and separate events and auxiliary zones, as much as possible (e.g., a specific type for auxiliary nodes and another, user selected, for the events); 4) Inclusion of a mechanism to adjust the value of *B*, in execution time, expecting to yield temporal distribution independence.

Acknowledgments

This work was developed as part of the "Programa de Desarrollo de las Ciencias Básicas" (PEDECIBA), UDELAR, Uruguay. Special thanks to Dr. Héctor Cancela for the time he spent supporting us with his accurate and helpful remarks.

References

[1] Blackstone J.H., Hogg G.L. and Phillips D.T., "A Two-List Synchronization Procedure for Discrete Event Simulation". *Comm. ACM*, 24 (12): 825-829 [December 1981].

[2] Brown, R., "Calendar Queues: A fast O(1) Priority Queue implementation for the simulation event set problem". *Comm. ACM*, 31 (10): 1220-1227 [October 1988].

[3] Jones D. W., "An Empirical Comparison of Priority-Queue and Event-Set Implementations". *Comm. ACM*, 29 (4): 300-311 [April 1986].

[4] Kingston J.H., "Analysis of Algorithms for the Simulation Event List", PhD Thesis. *Basser Dept. of Computer Science, University of Sidney, Australia* [July 1984].

[5] Knuth, D.E., "The Art of Computer Programming. Vol 1". *Addison-Wesley, Reading, Mass.* [1973].

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

[7] Marín M., "An Empirical Comparison of Priority Queue Algorithms", *Technical report PRG-TR-10-97, Oxford University* [1997].

[8] Mc Cormack W. and Sargent R., "Analysis of Future Event Set Algorithms for Discrete Event Simulation". *Comm. ACM*, 24 (12): 801-811 [December 1981].

[9] Mitrani I., "Simulation Techniques for Discrete Event Systems". *Cambridge University Press* [1982]

[10] Pugh W., "Skip Lists: A Probabilistic Alternative to Balanced Trees". *Comm. ACM*, 33 (6): 668-676 [June 1990].

[11] Röngrenn R. and Ayani R., "A Comparative Study of Parallel and Sequential Priority Queue Algorithms". *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, 7 (2): 157-209 [April 1997].

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

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

[14] Stasko J.T. and Vitter J.S., "Paring heaps: Experiments and Analysis". *Comm ACM*, 30 (3): 234-249 [March 1987].