

The next-event simulation models for the single-server service node and the simple inventory system from the previous two sections have such short event lists (two events for the single-server service node and three events for the simple inventory system) that their management does not require any special consideration. There are next-event simulations, however, that may have hundreds or even thousands of events on their event list simultaneously, and the efficient management of this list is crucial. The material in this section is based on a tutorial by Henriksen (1983) and Chapter 5 of Fishman (2001).

Although the discussion in this section is limited to managing event lists, practically all of the discussion applies equally well, for example, to the management of jobs in a single-server service node. A FIFO or LIFO queue discipline results in a trivial management of the jobs in the queue: jobs arriving when the server is busy are simply added to the tail (FIFO) or head (LIFO) of the queue. A “shortest processing time first” queue discipline (which is commonly advocated for minimizing the wait time in job shops), on the other hand, requires special data structures and algorithms to efficiently insert jobs into the queue and delete jobs from the queue.

5.3.1 INTRODUCTION

An event list is the data structure that contains a list of the events that are scheduled to occur in the future, along with any ancillary information associated with these events. The list is traditionally sorted by the scheduled time of occurrence, but, as indicated in the first example in this section, this is not a requirement. The event list is also known as the calendar, future events chain, sequencing set, future event set, etc. The elements that comprise an event list are known as future events, events, event notices, transactions, records, etc. We will use the term *event notice* to describe these elements.

Why is efficient management of the event notices on the event list so important that it warrants an entire section? Many next-event simulation models expend more CPU time on managing the event list than on any other aspect (e.g., random number generation, random variate generation, processing events, miscellaneous arithmetic operations, printing reports) of the simulation.

Next-event simulations that require event list management can be broken into four categories according to these two boolean classifications:

- There is either a *fixed* maximum or a *variable* maximum number of event notices on the event list. There are clear advantages to having the maximum number of events fixed in terms of memory allocation. All of the simulations seen thus far have had a fixed maximum number of event notices.
- The event list management technique is either being devised for one specific model or is being developed for a general-purpose simulation language. If the focus is on a single model, then the scheduling aspects of that model can be exploited for efficiency. An event list management technique designed for a general-purpose language must be robust in the sense that it performs reasonably well for a variety of simulation models.

There are two critical operations in the management of the event notices that comprise the event list. The first is the insertion, or *enqueue* operation, where an event notice is placed on the event list. This operation is also referred to as “scheduling” the event. The second is the deletion, or *dequeue* operation, where an event notice is removed from the event list. A deletion operation is performed to process the event (the more common case) or because a previously scheduled event needs to be canceled for some reason (the rare case). Insertion and deletion may occur at a prescribed position in the event list, or a search based on some criteria may need to be initiated first in order to determine the appropriate position. We will use the term *event list management scheme* to refer to the data structures and associated algorithms corresponding to one particular technique of handling event list insertions and deletions.

Of minor importance is a *change* operation, where a search for an existing event notice is followed by a change in some aspect of the event notice, such as changing its scheduled time of occurrence. Similarly, an *examine* operation searches for an existing event notice in order to examine its contents. A *count* operation is used to determine the number of event notices in the list. Due to their relative rarity in discrete-event simulation modeling and their similarity to insertion and deletion in principle, we will henceforth ignore the change, examine, and count operations and focus solely on the insertion and deletion operations.

5.3.2 EVENT LIST MANAGEMENT CRITERIA

Three criteria that can be used to assess the effectiveness of the data structures and algorithms for an event list management scheme are:

- **Speed.** The data structure and associated algorithms for inserting and deleting event notices should execute in minimal CPU time. Critical to achieving fast execution times is the efficient searching of the event list. The balance between sophisticated data structures and algorithms for searching must be weighed against the associated extraneous overhead calculations (e.g., maintaining pointers for a list or heap operations) that they require. An effective general-purpose algorithm typically bounds the number of event notices searched for inserting or deleting.
- **Robustness.** Efficient event list management should perform well for a wide range of scheduling scenarios. This is a much easier criteria to achieve if the characteristics of one particular model can be exploited by the analyst. The designer of an event list management scheme for a general-purpose language does not have this advantage.
- **Adaptability.** An effective event list management scheme should be able to adapt its searching time to account for both the length of the event list and the distribution of new events that are being scheduled. It is also advantageous for an event list management scheme to be “parameter-free” in the sense that the user should not be required to specify parameters that optimize the performance of the scheme. A “black box” approach to managing an event list is particularly appropriate for a general-purpose simulation language since users have a variety of sophistication levels.

Given these criteria, how do we know whether our event list management scheme is effective? If this is for a single model, the only way to answer this question is by running several different schemes on the same model to see which executes the fastest. It is typically not possible to prove that one particular scheme is superior to all other possible schemes, since a more clever analyst may exploit more of the structure associated with a specific model. It is, however, possible to show that one scheme dominates another in terms of execution time by making several runs with different seeds and comparing execution times.

Testing event list management schemes for a general-purpose language is much more difficult. In order to compare event list management schemes, one must consider a representative test-bed of diverse simulation models on which to test various event-list management schemes. Average and worst-case performance in terms of the CPU time tests the speed, robustness, and adaptability of various event list management schemes.

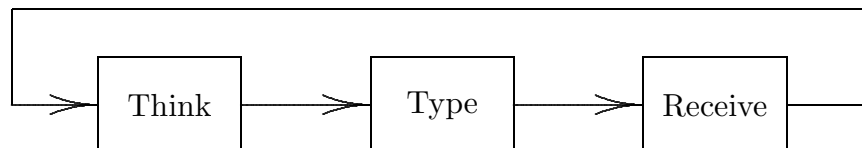
5.3.3 EXAMPLE

We consider the computer timesharing model presented in Henriksen (1983) to discuss event list management schemes. In our usual manner, we start with the conceptual model before moving to the specification and computational models.

Conceptual Model

Consider a user at a computer timesharing system who endlessly cycles from (1) thinking, to (2) typing in a command, to (3) receiving the output from the command. The user does not take any breaks, never tires, and the workstation never fails. A system diagram that depicts this behavior is given in Figure 5.3.1.

Figure 5.3.1.
Timeshare
system
diagram.



This is the first time we have encountered what is known as a “closed” system. The “open” systems considered previously are the queuing models (jobs arrive, are processed, then depart) and inventory models (inventory arrives, is purchased, then departs). In the timesharing model, there is no arrival or departure. The three-step activity loops endlessly.

The times to perform the three operations, measured in seconds, are:

- The time to think requires $Uniform(0, 10)$ seconds.
- There are an $Equilikely(5, 15)$ number of keystrokes involved in typing in a command, and each keystroke requires $Uniform(0.15, 0.35)$ seconds.
- The output from the command contains an $Equilikely(50, 300)$ number of characters, each requiring (the ancient rate of) $1/120$ second to display at the workstation.

If we were to simulate just one user, this would be a rather trivial model since there is only one event on the event list: the completion of the next activity (thinking, typing a keystroke, receiving an output character). To make event list management an issue, assume that the computer timesharing system consists of n users at n terminals, each asynchronously cycling through the three-step process of thinking, typing, and receiving.

Many critiques of this model would be valid. Do all of the users really all think and type at the same rate? Does the distribution of thinking time really “cut-off” at ten seconds as the *Uniform*(0, 10) thinking time implies? Are all of the *Uniform* and *Equilikely* distributions appropriate? Why doesn’t the receive rate degrade when several users receive output simultaneously? Why doesn’t the receiving portion of the system bog down as n increases? Because the purpose of this model is to illustrate the management of the event list, we will forgo discussion about the reasonableness and accuracy of the model. The important topic of developing “input” models that accurately mimic the system of interest is addressed in Chapter 9.

Back-of-an-Envelope Calculations

Since all of the distributions in this model are either *Uniform* or *Equilikely*, it is worthwhile to do some preliminary calculations which may provide insight into model behavior prior to the simulation. The mean of a *Uniform* or *Equilikely* random variable is, not surprisingly, the average of their two parameters. Thus the expected length of each cycle for each user is:

$$\begin{aligned} \left(\frac{0+10}{2}\right) + \left(\frac{5+15}{2}\right) \left(\frac{0.15+0.35}{2}\right) + \left(\frac{50+300}{2}\right) \left(\frac{1}{120}\right) &= 5 + 10 \cdot 0.25 + 175 \cdot \frac{1}{120} \\ &\cong 5 + 2.5 + 1.4583 \\ &= 8.9583 \end{aligned}$$

seconds. Thus if one were to observe a user at a random instant in time of a system in steady state, the probabilities that the user will be thinking, typing, and receiving are

$$\frac{5}{8.9583} \cong 0.56, \quad \frac{2.5}{8.9583} \cong 0.28, \quad \text{and} \quad \frac{1.4583}{8.9583} \cong 0.16.$$

These fractions apply to individual users, as well as the population of n users. At any particular point in simulated time, we could expect to see about 56% of the users thinking, 28% typing, and 16% receiving output.

Although it is clear from this analysis that the largest portion of a user’s *simulated time* is spent thinking and the smallest portion of a user’s *simulated time* is spent receiving, the opposite is true of the *number* of scheduled events. Each cycle has exactly one thinking event, an average of ten keystrokes, and an average of 175 characters received. Thus each cycle averages $1 + 10 + 175 = 186$ events. The expected fractions of events associated with thinking, typing a keystroke, and receiving an output character during a cycle are

$$\frac{1}{186} \cong 0.005, \quad \frac{10}{186} \cong 0.054, \quad \text{and} \quad \frac{175}{186} \cong 0.941.$$

The vast majority of the events scheduled during the simulation will be receiving a character. This observation will influence the probability distribution of the event times associated with event notices on the event list. This distribution can be exploited when designing an event list management scheme.

Specification Model

There are three events that comprise the activity of a user on the timesharing system:

- (1) complete thinking time;
- (2) complete a keystroke;
- (3) complete the display of a character.

For the second two events, some ancillary information must be stored as well: the number of keystrokes in the command and the number of characters returned from the command. For all three events, an integer (1 for thinking, 2 for typing and 3 for receiving) is stored to denote the event type. Ancillary information of this type is often called an “attribute” in general-purpose simulation languages.

As with most next-event simulation models, the processing of each event triggers the scheduling of future events. For this particular model, we are fortunate that each event triggers just one future event to schedule: the next activity for the user whose event is presently being processed.

Several data structures are capable of storing the event notices for this model. Two likely candidates are an array and a linked list. For simplicity, an array will be used to store the event notices. We can use an array here because we know in advance that there will always be n events on the event list, one event notice for the next activity for each user. We begin with this very simplistic (and grossly inefficient) data structure for the event list. We will subsequently refine this data structure, and eventually outline more sophisticated schemes. We will store the times of the events in an array of length n in an order associated with the n users, and make a linear search of all elements of the array whenever the next event to be processed needs to be found. Thus the deletion operation requires searching all n elements of the event list to find the event with the smallest event time, while the insertion operation requires no searching since the next event notice for a particular user simply overwrites the current event notice.

However, there is ancillary information that must be carried with each event notice. Instead of an array of n times, the event list for this model should be organized as an array of n event structures. Each event structure consists of three fields: **time**, **type**, and **info**. The **time** field in the i th event structure stores the time of the event for the i th user ($i = 0, 1, \dots, n - 1$). The **type** field stores the event type (1, 2, or 3). The **info** field stores the ancillary information associated with a keystroke or display of a character event: the number of keystrokes remaining (for a Type 2 event) or the number of characters remaining in the output (for a Type 3 event). The **info** field is not used for a thinking (Type 1) event since no ancillary information is needed for thinking.

The initialization phase of the next-event simulation model schedules a complete thinking time event (Type 1 event) for each of the n users on the system. The choice of beginning with this event and applying the choice to all users is arbitrary*. After the initialization, the event list for a system with $n = 5$ users might look like the one presented in Figure 5.3.2. Each of the five $Uniform(0, 10)$ completion of thinking event times is placed in the `time` field of the corresponding event structure. The value in the `time` field of the third structure is the smallest, indicating that the third user will be the first to stop thinking and begin typing at time 1.305. All five of these events are completion of thinking events, as indicated by the `type` field in each event structure.

time	9.803	3.507	1.305	2.155	8.243
type	1	1	1	1	1
info					
	0	1	2	3	4

Figure 5.3.2.
Initial
event list.

The remainder of the algorithm follows the standard next-event protocol — while the terminal condition has not been met: (1) scan the event list for the most imminent event, (2) update the simulation clock accordingly, (3) process the current event, and (4) schedule the subsequent event by placing the appropriate event notice on the event list.

As the initial event (end of thinking for the third user at time 1.305) is deleted from the event list and processed, a number of keystrokes [an *Equilikely*(5, 15) random variate which takes the value of 7 in this case — a slightly shorter than average command] and a time for the first keystroke [a $Uniform(0.15, 0.35)$ random variate which takes the value of 0.301 in this case — a slightly longer than average keystroke time] are generated. The `time` field in the third event structure is incremented by 0.301 to $1.305 + 0.301 = 1.606$, and the number of keystrokes in this command is stored in the corresponding `info` field. Subsequent keystrokes for the third user decrement the integer in the corresponding `info` field. The condition of the event list after the processing of the first event is shown in Figure 5.3.3.

time	9.803	3.507	1.606	2.155	8.243
type	1	1	2	1	1
info			7		
	0	1	2	3	4

Figure 5.3.3.
Updated
event list.

To complete the development of the specification model, we use the following notation:

- The simulation clock is t , which is measured in seconds.
- The simulation terminates when the next scheduled event is τ seconds or more.

The algorithm is straightforward, as presented in Algorithm 5.3.1.

* All users begin thinking simultaneously at time 0, but will behave more independently after a few cycles as the timesharing system “warms up.”

Algorithm 5.3.1 This algorithm is a next-event simulation of a think-type-receive time-sharing system with n concurrent users. The algorithm presumes the existence of four functions `GetThinkTime`, `GetKeystrokeTime`, `GetNumKeystrokes`, and `GetNumCharacters` that return the random time to think, time to enter a keystroke, number of keystrokes per command, and number of characters returned from a command respectively. The function `MinIndex` returns the index of the most imminent event notice.

```

t = 0.0;                               /* initialize system clock */
for (i = 0; i < n; i++) {               /* initialize event list */
    event[i].time = GetThinkTime();
    event[i].type = 1;
}
while (t <  $\tau$ ) {                       /* check for terminal condition */
    j = MinIndex(event.time);           /* find index of imminent event */
    t = event[j].time;                  /* update system clock */
    if (event[j].type == 1) {           /* process completion of thinking */
        event[j].time = t + GetKeystrokeTime();
        event[j].type = 2;
        event[j].info = GetNumKeystrokes();
    }
    else if (event[j].type == 2) { /* process completion of keystroke */
        event[j].info--; /* decrement number of keystrokes remaining */
        if (event[j].info > 0) /* if more keystrokes remain */
            event[j].time = t + GetKeystrokeTime();
        else { /* else last keystroke */
            event[j].time = t + 1.0 / 120.0;
            event[j].type = 3;
            event[j].info = GetNumCharacters();
        }
    }
    else if (event[j].type == 3) { /* process complete character rcvd */
        event[j].info--; /* decrement number of characters remaining */
        if (event[j].info > 0) /* if more characters remain */
            event[j].time = t + 1.0 / 120.0;
        else { /* else last character */
            event[j].time = t + GetThinkTime();
            event[j].type = 1;
        }
    }
}
}
}

```

Program ttr

The think-type-receive specification model has been implemented in program `ttr`, which prints the total number of events scheduled and the average number of event notices searched for each deletion. For each value of n in the table below, the simulation was run three times with seeds 123456789, 987654321, and 555555555 for $\tau = 100$ seconds, and the averages of the three runs are reported.

number of users n	expected number of events scheduled	average number of events scheduled	average number of event notices searched
5	10 381	9 902	5
10	20 763	20 678	10
50	103 814	101 669	50
100	207 628	201 949	100

The column headed “expected number of events scheduled” is determined as follows. Since the average length of each cycle is 8.9583 seconds, each user will go through an average of

$$\frac{100}{8.9583} \cong 11.16$$

cycles in $\tau = 100$ seconds. Since the average number of events per cycle is 186, we expect to see

$$(11.16) \cdot (186) \cdot n = 2076.3n$$

total events scheduled during the simulation. These values are reported in the second column of the table. The averages in the table from the three simulations are slightly lower than the expected values due to our arbitrary decision to begin each cycle thinking, the longest event. In terms of event list management, each deletion event (required to find the next event notice) requires an exhaustive search of the `time` field in all n event structures, so the average number of event notices searched for each deletion is simply n . The simplistic event list management scheme used here sets the stage for more sophisticated schemes.

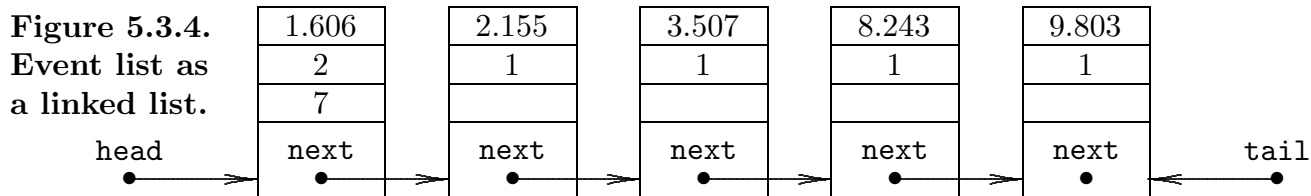
5.3.4 AN IMPROVED EVENT LIST MANAGEMENT SCHEME

Our decision in the previous example to store event times unordered is a departure from the traditional convention in simulation languages, which is to order the event notices on the event list in ascending order of event times, i.e., the event list is maintained in chronological order. If we now switch to an ordered event list, a deletion requires no searching and an insertion requires a search — just the opposite situation from the previous event list management scheme. This will be a wash time-wise for the think-type-receive model, since there is deletion for every insertion during the simulation. Every deletion associated with the scheduling of an event pulls the first event notice from the head of the list. This section is focused, therefore, on efficient algorithms for inserting event notices into the event list.

There is good and bad news associated with the move to an event list that is ordered by ascending event time. The good news is that the entire event list need not necessarily be searched exhaustively every time an insertion operation is conducted. The bad news, however, is that arrays are no longer a natural choice for the data structure due to the overhead associated with shuffling event notices down the array when an event notice is placed at the beginning or middle of the list. A singly- or doubly-linked list is a preferred data structure due to its ability to easily insert items in the middle of the list. The overhead of maintaining pointers, however, dilutes the benefit of moving to an event list that is ordered by ascending event time. Also, direct access to the array is lost and time-consuming element-by-element searches through the linked list are required.

A secondary benefit associated with switching to linked lists is that the maximum size of the list need not be specified in advance. This is of no consequence in our think-type-receive model since there are always n events in the event list. In a general-purpose discrete-event simulation language, however, linked lists expand until memory is exhausted.

Example 5.3.1 For the think-type-receive model with $n = 5$, for example, a singly-linked list, linked from head (top) to tail (bottom), to store the elements of the event list corresponding to Figure 5.3.3 is shown in Figure 5.3.4. The three values stored on each event notice are the event time, event type, and ancillary information (seven keystrokes remaining in a command for the first element in the list). The event notices are ordered by event time. A deletion now involves no search, but a search is required for each insertion.



One question remains before implementing the new data structure and algorithm for the search. Should the list be searched from head to tail (top to bottom for a list with forward pointers) or tail to head (bottom to top for a list with backward pointers)? We begin by searching from tail to head and check our efficiency gains over the naive event management scheme presented in the previous subsection. The table below shows that the average number of events scheduled is identical to the previous event management scheme (as expected due to the use of identical seeds), and improvements in the average number of searches per insertion improvements range from 18.8% ($n = 100$) to 23.4% ($n = 5$).

number of users n	average number of events scheduled	average number of event notices searched
5	9 902	3.83
10	20 678	8.11
50	101 669	40.55
100	201 949	81.19

These results are certainly not stunning. The improvement in search time is slight. What went wrong? The problem here is that we ignored our earlier back-of-an envelope calculations. These calculations indicated that 94.1% of the events in the simulation would be the receipt of a character, which has a very short inter-event time. Thus we should have searched the event list from head to tail since these short events are much more likely to be inserted at or near the top of the list. We re-programmed the search to go from head to tail, and the results are given in the table below.

number of users n	average number of events scheduled	average number of event notices searched
5	9 902	1.72
10	20 678	2.73
50	101 669	10.45
100	201 949	19.81

Confirming our calculations, the forward search performs far better than the backward search.* In this case the savings in terms of the number of searches required over the exhaustive search ranges from 66% (for $n = 5$) to 80% (for $n = 100$).

The think-type-receive model with $n = 100$ highlights our emphasis on efficient event list management techniques. Even with the best of the three event list management schemes employed so far (forward linear search of a singly-linked list, averaging 19.81 searches per insertion), more time is spent on event list management than the rest of the simulation operations (e.g., random number generation, random variate generation, processing events) combined!

5.3.5 MORE ADVANCED EVENT LIST MANAGEMENT SCHEMES

The think-type-receive model represents the simplest possible case for event list management. First, the number of event notices on the event list remains constant throughout the simulation. Second, the fact that there are frequent short events (e.g., receiving a character) can be exploited in order to minimize the search time for an insertion using a forward search.

* An interesting verification of the forward and backward searches can be made in this case since separate streams of random numbers assures an identical sequencing of events. For an identical event list of size n , the sum of the number of forward searches and the number of backward searches equals n for an insertion at the top or bottom of the list. For an insertion in the middle, however, the sum equals $n + 1$ for identical lists. Therefore, the sum of the rightmost columns of the last two tables will always lie between n and $n + 1$, which it does in this case. The sums are 5.55, 10.84, 51.00, and 101.00 for $n = 5, 10, 50,$ and 100 . The sum tends to $n + 1$ for large n since it is more likely to have an insertion in the middle of the list as n grows. Stated another way, when n is large it is a near certainty that several users will be simultaneously receiving characters when an event notice is placed on the event list, meaning that the event is unlikely to be placed at the front of the list.

We now proceed to a discussion of the general case where (1) the number of event notices in the event list varies throughout the simulation, (2) the maximum length of the event list is not known in advance, and (3) the structure of the simulation model is unknown so it cannot be exploited for optimizing an event list management scheme.

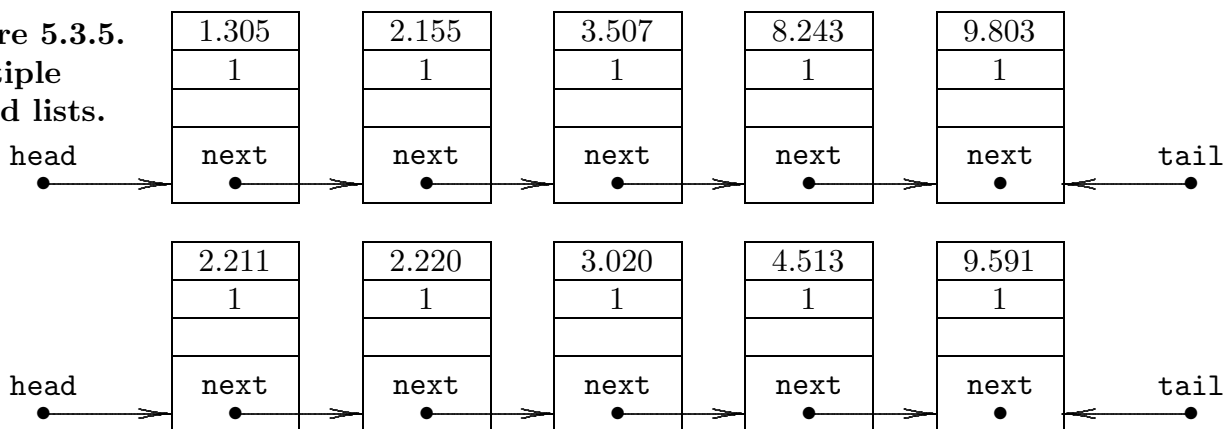
In order to reduce the scope of the discussion, assume that a memory allocation mechanism exists so that a memory location occupied by a deleted event notice may be immediately occupied by an event notice that is subsequently inserted into the event list. When memory space is released as soon as it becomes available in this fashion, the simulation will fail due to lack of memory only when an insertion is made to an event list that exhausts the space allocated to the event list. Many general-purpose languages effectively place all entities (e.g., event notices in the event list, jobs waiting in a queue) in the simulation in a single partitioned list in order to use memory in the most efficient manner. Data structures and algorithms associated with the allocation and de-allocation of memory are detailed in Chapter 5 of Fishman (2001).

The next four subsections briefly outline event list management schemes commonly used to efficiently insert event notices into an event list and delete event notices from an event list in a general setting: multiple linked lists, binary search trees, heaps, and hybrid schemes.

Multiple Linked Lists

One approach to reducing search time associated with insertions and deletions is to maintain multiple linear lists, each sorted by event time. Let k denote the number of such lists and n denote the number of event notices in all lists at one particular point in simulated time. Figure 5.3.5 shows $k = 2$ equal-length, singly-linked lists for $n = 10$ initial think times in the think-type-receive model. An insertion can be made into either list. If the list sizes were not equal, choosing the shortest list minimizes the search time. The time savings associated with the insertion operation is offset by (1) the overhead associated with maintaining the multiple lists, and (2) a less efficient deletion operation. A deletion now requires a search of the top (head) event notices of the k lists, and the event notice with the smallest event time is deleted.

Figure 5.3.5.
Multiple
linked lists.



We close the discussion of multiple event lists with three issues that are important for minimizing CPU time for insertions and deletions:

- The decision of whether to fix the number of lists k throughout the simulation or allow it to vary depends on many factors, including the largest value of n throughout the simulation, the distribution of the position of insertions in the event list, and how widely n varies throughout the simulation.
- If the number of lists k is allowed to vary throughout the simulation, the modeler must determine appropriate thresholds for n where lists are split (as n increases) and combined (as n decreases).
- The CPU time associated with inserting an event notice, deleting an event notice, combining lists, and splitting lists as functions of n and k should drive the optimization of this event list management scheme.

The next two data structures, binary trees and heaps, are well-known data structures. Rather than developing the data structures and associated operations from scratch, we refer the reader to Carrano and Prichard (2002) for basic definitions, examples, and applications. Our discussion of these two data structures here will be rather general in nature.

Binary Trees

We limit our discussion of trees to *binary trees*. A binary tree consists of n nodes connected by edges in a hierarchical fashion such that a *parent* node lies above and is linked to at most two *child* nodes. The parent-child relationship generalizes to the *ancestor-descendant* relationship in an analogous fashion to a family tree. A *subtree* in a binary tree consists of a node, along with all of the associated descendants. The top node in a binary tree is the only node in the tree without a parent, and is called the *root*. A node with no children is called a *leaf*. The *height* of a binary tree is the number of nodes on the longest path from root to leaf. The *level* of a node is 1 if it is the root or 1 greater than the level of its parent if it is not the root. A binary tree of height h is *full* if all nodes at a level less than h have two children each. Full trees have $n = 2^h - 1$ nodes. A binary tree of height h is *complete* if it is full down to level $h - 1$ and level h is filled from left to right. A full binary tree of height $h = 3$ with $n = 7$ nodes and a complete binary tree of height $h = 4$ with $n = 12$ nodes are displayed in Figure 5.3.6.

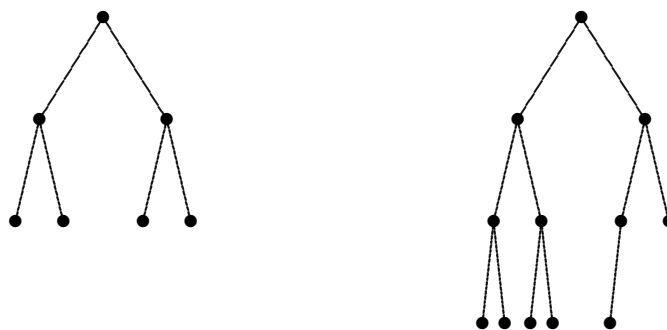
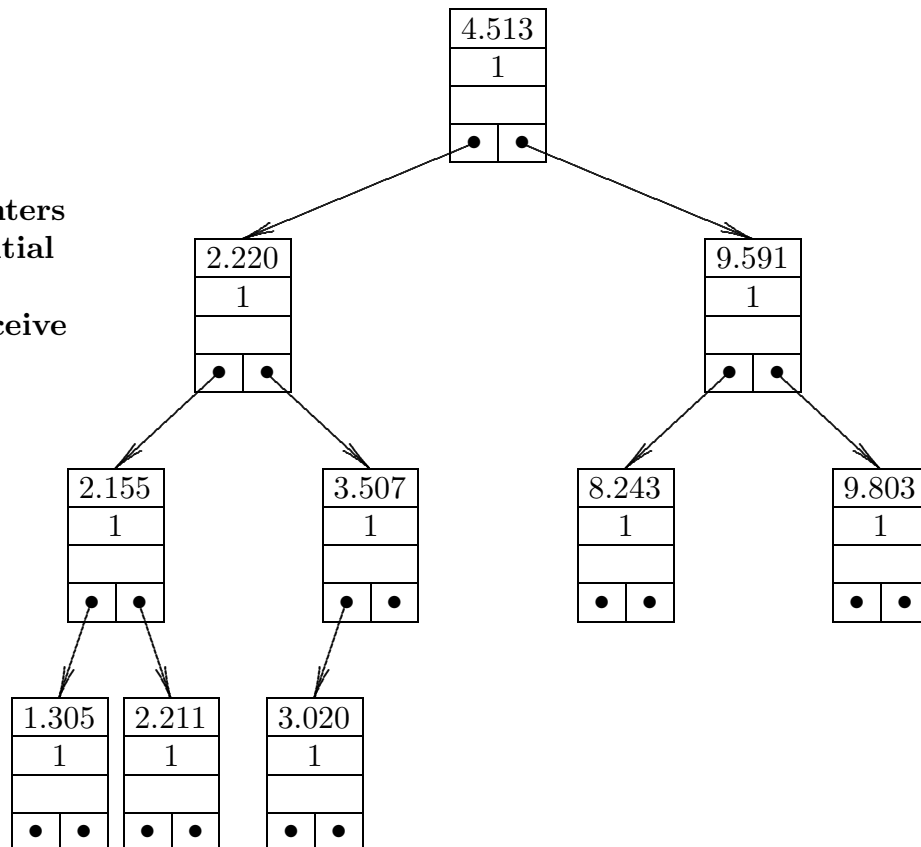


Figure 5.3.6.
Full and
complete
binary
trees.

Nodes are often associated with a numeric value. In our setting, a node corresponds to an event notice and the numeric value associated with the node is the event time. A *binary search tree* is a binary tree where the value associated with any node is greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree. The “or equal to” portions of the previous sentence have been added to the standard definition of a binary search tree to allow for the possibility of equal event times.

Example 5.3.2 There are many ways to implement a binary tree. Figure 5.3.7 shows a *pointer-based* complete binary tree representation of the ten initial events (from Figure 5.3.5) for a think-type-receive model with $n = 10$ users, where each user begins with a $Uniform(0, 10)$ thinking activity. Each event notice has three fields (event time, event type, and event information) and each parent points to its child or children, if any. Every event time is greater than or equal to every event time in its left subtree and less than or equal to every event time in its right subtree. Although there are many binary search tree configurations that could contain these particular event notices, the placement of the event notices in the *complete* binary search tree in Figure 5.3.7 is unique.

Figure 5.3.7.
Binary search tree with pointers for the ten initial events in the think-type-receive model.



One advantage to binary search trees for storing event notices is that the “leftmost” leaf in the tree will always be the most imminent event. This makes a deletion operation fast, although it may require reconfiguring the tree after the deletion.

Insertions are faster than a linear search of a list due to the decreased number of comparisons necessary to find the appropriate insertion position. The key decision that remains for the scheme is whether the binary tree will be maintained as a complete tree (involving extra overhead associated with insertions and deletions) or allowed to evolve without the requirement that the tree is complete (which may result in an “imbalanced” tree whose height increases over time, requiring more comparisons for insertions). *Splay trees*, which require frequent rotations to maintain balance, have also performed well.

Heaps

A heap is another data structure for storing event notices in order to minimize insertion and deletion times. In our setting, a heap is a complete binary tree with the following properties: (1) the event time associated with the root is less than or equal to the event time associated with each of its children, and (2) the root has heaps as subtrees. Figure 5.3.8 shows a heap associated with the first ten events in the think-type-receive model. This heap is not unique.* An obvious advantage to the heap data structure is that the most imminent event is at the root, making deletions fast. The heap property must be maintained, however, whenever an insertion or deletion operation is performed.

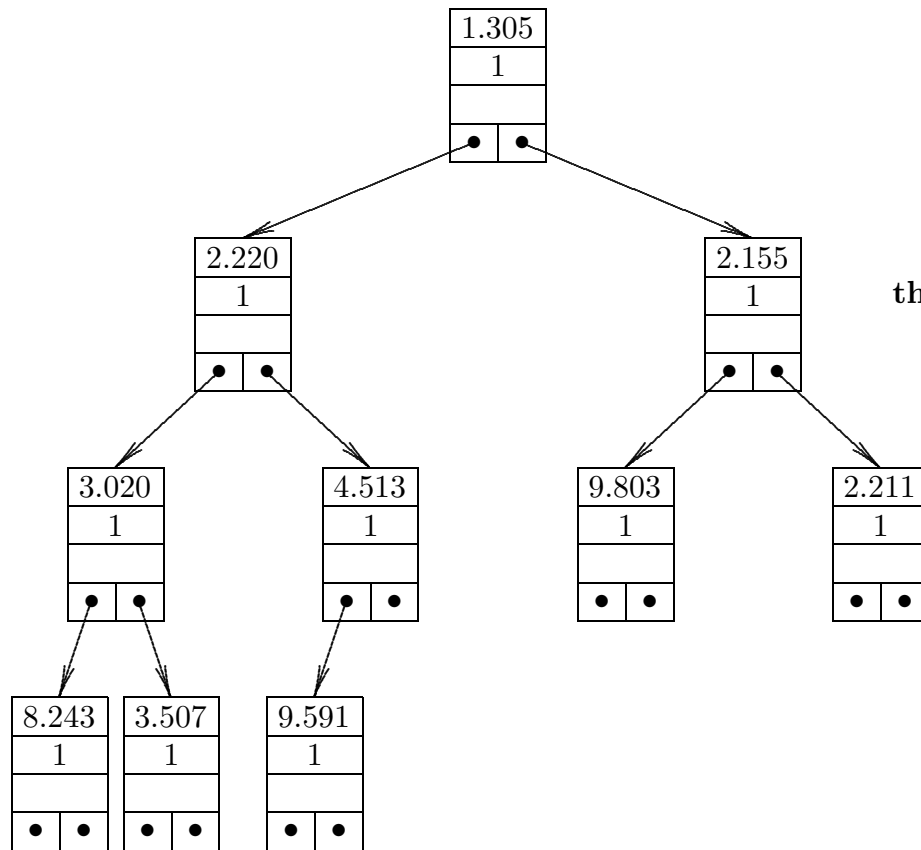


Figure 5.3.8.
Heap with
pointers for
the ten initial
events in the
think-type-
receive
model.

* If the event notices associated with times 2.220 and 2.211 in Figure 5.3.8, for example, were interchanged, the heap property would be retained.

Hybrid Schemes

The ideal event list management scheme performs well regardless of the size of the event list. Jones (1986) concludes that when there are fewer than ten event notices on an event list, a singly-linked list is optimal due to the overhead associated with more sophisticated schemes. Thus to fully optimize an event list management scheme, it may be necessary to have thresholds, similar to those that switch a thermostat on and off, that switch from one set of data structures and algorithms to another based on the number of events on the list. It is important to avoid switching back and forth too often, however, since the switch typically requires overhead processing time as well.

If a heap, for example, is used when the event list is long and a singly-linked list is used when the event list is short, then appropriate thresholds should be determined that will switch from one scheme to the other. As an illustration, when the number of event notices shrinks to $n = 5$ (e.g., n decreases from 6 to 5), the heap is converted to a singly-linked list. Similarly, when the number of event notices grows to $n = 15$ (e.g., n increases from 14 to 15), the singly-linked list is converted to a heap.

Henriksen's algorithm (Henriksen, 1983) provides adaptability to short and long event lists without alternating data structures based on thresholds. Henriksen's algorithm uses a binary search tree and a doubly-linked list simultaneously. This algorithm has been implemented in several simulation languages, including GPSS, SLX, and SLAM. At the conceptual level, Henriksen's algorithm employs two data structures:

- The event list is maintained as a single, linear, doubly-linked list ordered by event times. The list is augmented by a dummy event notice on the left with a simulated time of $-\infty$ and a dummy event notice on the right with a simulated time of $+\infty$ to allow symmetry of treatment for all real event notices.
- A binary search tree with nodes associated with a *subset* of the event notices in the event list, has nodes with the format shown in Figure 5.3.9. Leaf nodes have zeros for left and right child pointers. The leftmost node in the tree has a zero for a pointer to the next lower time tree node. This binary search tree is degenerate at the beginning of the simulation (prior to scheduling initial events).

Figure 5.3.9.
Binary search
tree node
format.

Pointer to next lower time tree node
Pointer to left child tree node
Pointer to right child tree node
Event time
Pointer to the event notice

A three-node binary search tree associated with the ten initial events in the think-type-receive model is given in Figure 5.3.10. The binary search tree is given at the top of the figure, and the linear doubly-linked list containing the ten initial event notices (plus the dummy event notices at both ends of the list) is shown at the bottom of the figure.

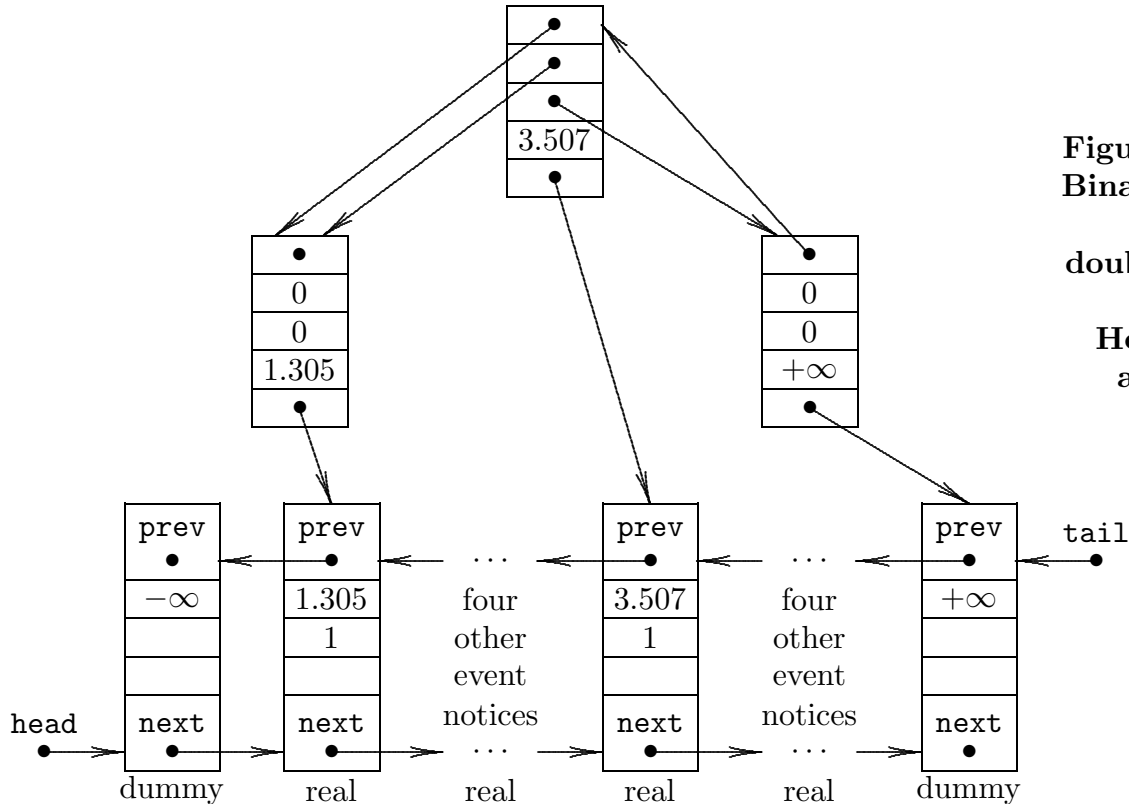


Figure 5.3.10.
Binary search
tree and
doubly-linked
list for
Henriksen's
algorithm.

A deletion is an $O(1)$ operation since the first real event notice in the event list is the most imminent event. To insert an event notice into the event list, the binary search tree is traversed in order to find the position of the event in the tree with the smallest event time greater than the event notice being inserted. A backward linear search of the doubly-linked event list is initiated at the event notice to the left of the one found in the binary search tree. This backward linear search continues until either:

- the appropriate insertion position is found in l or fewer searches (Henriksen recommends $l = 4$), in which case the new event notice is linked into the event list at the appropriate position, or
- the appropriate insertion position is not found in l or fewer searches, in which case a “pull” operation is attempted. The pull operation begins by examining the pointer to the event in the binary search tree with the *next lower time* relative to the one found previously. If this pointer is non-zero, its pointer is changed to the most recently examined event notice in the doubly-linked list, i.e., the l th event encountered during the search, and the search continues for another l event notices as before. If the pointer is zero, there are no earlier binary tree nodes that can be updated, so the algorithm adds a new level to the tree. The new level is initialized by setting its leftmost leaf to point to the dummy notice on the right (event time $+\infty$) and setting all other new leaves to point to the dummy notice on the left (event time $-\infty$). The binary search tree is again searched as before.

Henriksen's algorithm works quite well at reducing the average search time for an insertion. Its only drawback seems to be that the maximum search time can be quite long. Other hybrid event list management schemes may also have promise for reducing CPU times associated with insertions and deletions (e.g., Brown, 1988).

5.3.6 EXERCISES

Exercise 5.3.1 Modify program `ttr` so that the initial event for each user is the completion of the first character received as output, rather than the completion of thinking. Run the modified programs for $n = 5, 10, 50, 100$, and for initial seeds 123456789, 987654321, and 555555555. Compare the average number of events for the three simulation runs relative to the results in Section 5.3.3. Offer an explanation of why the observed average number of events goes up or down.

Exercise 5.3.2 Modify program `ttr` to include an event list that is sorted by event time and is stored in a linked list. Verify the results for a forward search given in Example 5.3.1.

Exercise 5.3.3 Assume that all events (thinking, typing a character, and receiving a character) in the think-type-receive model have deterministic durations of exactly 1/10 second. Write a paragraph describing an event list management scheme that requires *no* searching. Include the reason(s) that no searching is required.

Exercise 5.3.4 Assume that all events (thinking, typing a character, and receiving a character) in the think-type-receive model have *Uniform*(0.4, 0.6) second durations. If you use a doubly-linked list data structure to store the event list with events stored in chronological order, would it be wiser to begin an insertion operation with a search starting at the top (head) of the list or the bottom (tail) of the list? Justify your answer.

Exercise 5.3.5^a Assume that all events (thinking, typing a character, and receiving a character) in the think-type-receive model have *Exponential*(0.5) second durations. If you use a doubly-linked list data structure to store the event list with events stored in chronological order, would it be wiser to begin an insertion operation with a search starting at the top (head) of the list or the bottom (tail) of the list? Justify your answer.

Exercise 5.3.6 The *verification* process from Algorithm 1.1.1 involves checking whether a simulation model is working as expected. Program `ttr` prints the contents of the event list when the simulation reaches its terminal condition. What verification technique could be applied to this output to see if the program is executing as intended.

Exercise 5.3.7 The *verification* process from Algorithm 1.1.1 involves checking whether a simulation model is working as expected. Give a verification technique for comparing the think-type-receive model with (a) an unsorted event list with an exhaustive search for a deletion, and (b) an event list which is sorted by event time with a backward search for an insertion.