CHAPTER 5
NEXT-EVENT SIMULATION

Sections

The three sections in this chapter all concern the *next-event* approach to discrete-event simulation. Section 5.1 defines the fundamental terminology used in next-event simulation such as system state, events, simulation clock, event scheduling, and event list (which is also known as the *calendar*), and provides an introduction to this fundamental approach as it applies to the simulation of a single-server service node with and without feedback. The algorithm associated with next-event simulation initializes the simulation clock (typically to time zero), event list (with an initial arrival, for example, in a queuing model), and system state to begin the simulation. The simulation model continues to (1) remove the next event from the event list, (2) update the simulation clock to the time of the next event, (3) process the event, and (4) schedule the time of occurrence of any future events spawned by the event, until some terminal condition is satisfied.

Section 5.2 provides further illustrations of this approach relative to the simulation of a simple inventory system with delivery lag and a multi-server service node. The multi-server service node provides an illustration of an event list which can have an arbitrarily large number of elements.

As the simulations in Sections 5.1 and 5.2 illustrate, an *event list* is an integral feature of the next-event approach. The data structures and algorithms that are used to manage the event list are crucial to the efficiency of a next-event simulation. Section 5.3 provides a sequence of examples associated with the management of an event list that begin with a naive and inefficient data structure and algorithm and iterate toward a more efficient scheme.

5.1 Next-Event Simulation

In this section we will present a general *next-event* approach to building discrete-event simulation models. From this chapter on, this next-event approach will be the basis for all the discrete-event simulation models developed in this book.

The motivation for considering the next-event approach to discrete-event simulation is provided by considering the relative complexity of the effort required to extend the discrete-event simulation models in Section 3.1 to accommodate the slightly more sophisticated corresponding models in Section 3.3. That is, at the computational level compare the simplicity of program `ssq2` in Section 3.1 with the increased complexity of the extension to `ssq2` that would be required to reproduce the results in Example 3.3.2. Yet the only increase in the complexity of the associated single-server service node model is the addition of immediate feedback. Similarly, compare the simplicity of program `sis2` in Section 3.1 with the increased complexity of the extension to `sis2` that would be required to reproduce the results in Example 3.3.4. Yet in this case the only increase in the complexity of the associated simple inventory system model is the addition of a delivery lag.

5.1.1 DEFINITIONS AND TERMINOLOGY

While programs `ssq2` and `sis2` and their corresponding extensions in Section 3.3 are valid and meaningful (albeit simple) discrete-event simulation programs, they do not adapt easily to increased model complexity and they do not generalize well to other systems. Based on these observations we see the need for a more general approach to discrete-event simulation that applies to queuing systems, inventory systems and a variety of other systems as well. This more general approach — next-event simulation — is based on some important definitions and terminology: (1) system state, (2) events, (3) simulation clock, (4) event scheduling, and (5) event list (calendar).

System State

Definition 5.1.1 The *state* of a system is a complete characterization of the system at an instance in time — a comprehensive "snapshot" in time. To the extent that the state of a system can be characterized by assigning values to variables, then *state variables* are what is used for this purpose.

To build a discrete-event simulation model using the next-event approach, the focus is on refining a description of the state of the system and its evolution in time. At the *conceptual* model level the state of a system exists only in the abstract as a collection of possible answers to the following questions: what are the state variables, how are they interrelated, and how do they evolve in time? At the *specification* level the state of the system exists as a collection of mathematical variables (the state variables) together with equations and logic describing how the state variables are interrelated and an algorithm for computing their interaction and evolution in time. At the *computational* level the state of the system exists as a collection of program variables that collectively characterize the system and are systematically updated as (simulated) time evolves.

Example 5.1.1 A natural way to describe the state of a single-server service node is to use the number of jobs in the service node as a state variable. As demonstrated later in this section, by refining this system state description we can construct a next-event simulation model for a single-server service node with or without immediate feedback.

Example 5.1.2 Similarly, a natural way to describe the state of a simple inventory system is to use the current inventory level and the amount of inventory on order (if any) as state variables. As demonstrated in the next section, by refining this system state description we can construct a next-event simulation model of a simple inventory system with or without delivery lag.

Events

Definition 5.1.2 An *event* is an occurrence that may change the state of the system. By definition, the state of the system can only change at an event time. Each event has an associated *event type*.

Example 5.1.3 For a single-server service node model with or without immediate feedback, there are two types of events: the *arrival* of a job and the *completion of service* for a job. These two types of occurrences are events because they have the potential to change the state of the system. An arrival will always increase the number in the service node by one; if there is no feedback, a completion of service will always decrease the number in the service node by one. When there is feedback, a completion *may* decrease the number in the service node by one. In this case there are two event types because the "arrival" event type and the "completion of service" event type are not the same.

Example 5.1.4 For a simple inventory system with delivery lag there are three event types: the occurrence of a *demand* instance, an *inventory review*, and the *arrival* of an inventory replenishment order. These are events because they have the potential to change the state of the system: a demand will decrease the inventory level by one, an inventory review may increase the amount of inventory on order, and the arrival of an inventory replenishment order will increase the inventory level and decrease the amount of inventory on order.

The *may* in Definition 5.1.2 is important; it is not necessary for an event to cause a change in the state of the system, as illustrated in the following four examples: (1) events can be scheduled that statistically *sample*, but do not change, the state of a system, (2) for a single-server service node with immediate feedback, a job's completion of service will only change the state of the system if the job is not fed back, (3) for a single-server service node, an event may be scheduled at a prescribed time (e.g., 5 PM) to cut off the stream of arriving jobs to the node, which will not change the state of the system, and (4) for a simple inventory system with delivery lag, an inventory review will only change the state of the system if an order is placed.

Simulation Clock

Because a discrete-event simulation model is dynamic, as the simulated system evolves it is necessary to keep track of the current value of simulated time. In the implementation phase of a next-event simulation, the natural way keep track of simulated time is with a floating point variable, which is typically named `t`, `time`, `tnow`, or `clock` in discrete-event simulation packages. The two examples that follow the definition of the simulation clock highlight the inability of the discrete-event simulation approach to easily generalize or embellish models. The next-event framework overcomes this limitation.

Definition 5.1.3 The variable that represents the current value of simulated time in a next-event simulation model is called the *simulation clock*.

Example 5.1.5 The discrete-event simulation model that program `ssq2` represents is heavily dependent on the job processing order imposed by the FIFO queue discipline. Therefore, it is difficult to extend the model to account for immediate feedback, or a finite service node capacity, or a priority queue discipline. In part, the reason for this difficulty is that there are effectively *two* simulation clocks with one coupled to the arrival events and the other coupled to the completion of service events. These two clocks are not synchronized and so it is difficult to reason about the temporal order of events if arrivals and completions of service are merged by feedback.

Example 5.1.6 The discrete-event simulation model that program `sis2` represents has only one type of event, inventory review, and events of this type occur deterministically at the beginning of each time interval. There is a simulation clock, but it is *integer-valued* and so is primitive at best. Because the simulation clock is integer-valued we are essentially forced to ignore the individual demand instances that occur within each time interval. Instead, all the demands per time interval are aggregated into one random variable. This aggregation makes for a computationally efficient discrete-event simulation program, but forces us in return to do some calculus to derive equations for the time-averaged holding and shortage levels. As outlined in Section 3.3, when there is a delivery lag the derivation of those equations is a significant task.

Event Scheduling

In a discrete-event simulation model it is necessary to use a *time-advance mechanism* to guarantee that events occur in the correct order — that is, to guarantee that the simulation clock never runs backward. The primary time-advance mechanism used in discrete-event simulation is known as *next-event* time advance; this mechanism is typically used in conjunction with *event scheduling*.

Definition 5.1.4 If event scheduling is used with a next-event time-advance mechanism as the basis for developing a discrete-event simulation model, the result is called a *next-event* simulation model.

To construct a next-event simulation model, three things must be done:

- construct a set of state variables that together provide a complete system description;
- identify the system event types;
- construct a collection of algorithms that define the state changes that will take place when each type of event occurs.

The model is constructed so as to cause the (simulated) system to evolve in (simulated) time by executing the events in increasing order of their scheduled time of occurrence. Time does not flow continuously; instead, the simulation clock is advanced discontinuously from event time to event time. At the computational level, the simulation clock is frozen during the execution of each state-change algorithm so that each change of state, no matter how computationally complex, occurs instantaneously relative to the simulation clock.

Event List

Definition 5.1.5 The data structure that represents the scheduled time of occurrence for the next possible event of each type is called the *event list* or *calendar*.

The event list is often, but not necessarily, represented as a priority queue sorted by the next scheduled time of occurrence for each event type.

5.1.2 NEXT-EVENT SIMULATION

Algorithm 5.1.1 A next-event simulation model consists of the following four steps:

- **Initialize.** The simulation clock is initialized (usually to zero) and, by looking ahead, the first time of occurrence of each possible event type is determined and scheduled, thereby initializing the event list.
- **Process current event.** The event list is scanned to determine the *most imminent* possible event, the simulation clock is then advanced to this event's scheduled time of occurrence, and the state of the system is updated to account for the occurrence of this event. This event is known as the "current" event.
- **Schedule new events.** New events (if any) that may be spawned by the current event are placed on the event list (typically in chronological order).
- **Terminate.** The process of advancing the simulation clock from one event time to the next continues until some terminal condition is satisfied. This terminal condition may be specified as a *pseudo*-event that only occurs once, at the end of the simulation, with the specification based on processing a fixed number of events, exceeding a fixed simulation clock time, or estimating an output measure to a prescribed precision.

The next-event simulation model initializes once at the beginning of a simulation replication, then alternates between the second step (processing the current event) and third step (scheduling subsequent events) until some terminate criteria is encountered.

Because event times are typically random, the simulation clock runs *asynchronously*. Moreover, because state changes *only* occur at event times, periods of system inactivity are ignored by advancing the simulation clock from event time to event time. Compared to the alternate, which is a *fixed-increment* time-advance mechanism, there is a clear computational efficiency advantage to this type of asynchronous next-event processing.*

In the remainder of this section, next-event simulation will be illustrated by constructing a next-event model of a single-server service node. Additional illustrations are provided in the next section by constructing next-event simulation models of a simple inventory system with delivery lag and a multi-server service node.

5.1.3 SINGLE-SERVER SERVICE NODE

The state variable $l(t)$ provides a *complete* characterization of the state of a single-server service node in the sense that

$$\begin{aligned} l(t) = 0 &\iff q(t) = 0 \quad \text{and} \quad x(t) = 0 \\ l(t) > 0 &\iff q(t) = l(t) - 1 \quad \text{and} \quad x(t) = 1 \end{aligned}$$

where $l(t)$, $q(t)$, and $x(t)$ represent the number in the node, in the queue, and in service respectively at time $t > 0$. In words, if the number in the service node is known, then the number in the queue and the status (idle or busy) of the server is also known. Given that the state of the system is characterized by $l(t)$, we then ask what events can cause $l(t)$ to change? The answer is that there are two such events: (1) an *arrival* in which case $l(t)$ is increased by 1; and (2) a *completion of service* in which case $l(t)$ is decreased by 1. Therefore, our conceptual model of a single-server service node consists of the state variable $l(t)$ and two associated event types, arrival and completion of service.

To turn this next-event conceptual model into a specification model, three additional assumptions must be made.

- The *initial state* $l(0)$ can have any non-negative integer value. It is common, however, to assume that $l(0) = 0$, often referred to as "empty and idle" in reference to the initial queue condition and server status, respectively. Therefore, the first event must be an arrival.

* Note that asynchronous next-event processing cannot be used if there is a need at the computational level for the simulation program to interact synchronously with some other process. For example, because of the need to interact with a person, so called "real time" or "person-in-the-loop" simulation programs must use a fixed-increment time-advance mechanism. In this case the underlying system model is usually based on a system of ordinary differential equations. In any case, fixed-increment time-advance simulation models are outside the scope of this book, but are included in some of the languages surveyed in Appendix A.

- Although the *terminal state* can also have any non-negative integer value, it is common to assume, as we will do, that the terminal state is also idle. Rather than specifying the number of jobs processed, our stopping criteria will be specified in terms of a time τ beyond which no new jobs can arrive. This assumption effectively "closes the door" at time τ but allows the system to continue operation until all jobs have been completely served. This would be the case, for instance, at an ice cream shop that closes at a particular hour, but allows remaining customers to be served. Therefore, the last event must be a completion of service.*
- Some mechanism must be used to denote an event as *impossible*. One way to do this is to structure the event list so that it contains *possible* events only. This is particularly desirable if the number of event types is large. As an alternate, if the number of event types is not large then the event list can be structured so that it contains both possible and impossible events — but with a numeric constant "∞" used for an event time to denote the impossibility of an event. For simplicity, this alternate event list structure is used in Algorithm 5.1.2.

To complete the development of a specification model, the following notation is used. The next-event specification model is then sufficiently simple that we can write Algorithm 5.1.2 directly.

- The simulation clock (current time) is t.
- The terminal ("close the door") time is τ.
- The next scheduled arrival time is t_a.
- The next scheduled service completion time is t_c.
- The number in the node (state variable) is l.

The genius and allure of both discrete-event and next-event simulation is apparent, for example, in the generation of arrival times in Algorithm 5.1.2. The naive approach of generating and storing all arrivals prior to the execution of the simulation is not necessary. Even if this naive approach were taken, the modeler would be beset by the dual problems of memory consumption and not knowing how many arrivals to schedule. Next-event simulation simply primes the pump by scheduling the first arrival in the initialization phase, then schedules each subsequent arrival while processing the current arrival. Meanwhile, service completions weave their way into the event list at the appropriate moments in order to provide the appropriate sequencing of arrivals and service completions.

At the end of this section we will discuss how to extend Algorithm 5.1.2 to account for several model extensions: immediate feedback, alternative queue disciplines, finite capacity, and random sampling.

---

* The simulation will terminate at $t = \tau$ only if $l(\tau) = 0$. If instead $l(\tau) > 0$ then the simulation will terminate at $t > \tau$ because additional time will be required to complete service on the jobs in the service node.

Algorithm 5.1.2 This algorithm is a next-event simulation of a FIFO single-server service node with infinite capacity. The service node begins and ends in an empty and idle state. The algorithm presumes the existence of two functions `GetArrival` and `GetService` that return a random value of arrival time and service time respectively.

```
l = 0;                                  /* initialize the system state */
t = 0.0;                                /* initialize the system clock */
t_a = GetArrival();                       /* initialize the event list */
t_c = ∞;                                  /* initialize the event list */
while ((t_a < τ) or (l > 0)) {          /* check for terminal condition */
  t = min(t_a, t_c);                              /* scan the event list */
  if (t == t_a) {                                  /* process an arrival */
    l++;
    t_a = GetArrival();
    if (t_a > τ)
      t_a = ∞;
    if (l == 1)
      t_c = t + GetService();
  }
  else {                              /* process a completion of service */
    l--;
    if (l > 0)
      t_c = t + GetService();
    else
      t_c = ∞;
  }
}
```

If the service node is to be an M/M/1 queue (exponential interarrival and service times with a single server) with arrival rate 1.0 and service rate 1.25, for example, the two t_a = `GetArrival()` statements can be replaced with t_a = t + `Exponential(1.0)` and the two t_a = `GetService()` statements can be replaced with t_c = t + `Exponential(0.8)`. The `GetArrival` and `GetService` functions can draw their values from a file (a "trace-driven" approach) or generate variates to model these stochastic elements of the service node.

Because there are just two event types, arrival and completion of service, the event list in Algorithm 5.1.2 contains at most two elements, t_a and t_c. Given that the event list is small and its size is bounded (by 2), there is no need for any special data structure to represent it. If the event list were larger, an array or structure would be a natural choice. The only drawback to storing the event list as t_a and t_c is the need to specify the arbitrary numeric constant "∞" to denote the impossibility of an event. In practice, ∞ can be any number that is *much* larger than the terminal time τ (100τ is used in program `ssq3`).

If the event list is large and its size is dynamic then a dynamic data structure is required with careful attention paid to its organization. This is necessary because the event list is scanned and updated each time an event occurs. Efficient algorithms for the insertion and deletion of events on the event list can impact the computational time required to execute the next-event simulation model. Henriksen (1983) indicates that for telecommunications system models, the choice between an efficient and inefficient event list processing algorithm can produce a five-fold difference in total processing time. Further discussion of data structures and algorithms associated with event lists is postponed to Section 5.3.

Program ssq3

Program `ssq3` is based on Algorithm 5.1.2. Note, in particular, the state variable `number` which represents $l(t)$, the number in the service node at time t, and the important time management structure `t` that contains:

- the event list `t.arrival` and `t.completion` (t_a and t_c from Algorithm 5.1.2);
- the simulation clock `t.current` (t from Algorithm 5.1.2);
- the next event time `t.next` ($\min(t_a, t_c)$ from Algorithm 5.1.2);
- the last arrival time `t.last`.

Event list management is trivial. The event type (arrival or a completion of service) of the next event is determined by the statement `t.next = Min(t.arrival, t.completion)`.

Note also that a statistics gathering structure `area` is used to calculate the time-averaged number in the node, queue, and service. These statistics are calculated exactly by accumulating time integrals via summation, which is valid because $l(\cdot)$, $q(\cdot)$, and $x(\cdot)$ are piecewise constant functions and only change value at an event time (see Section 4.1). The structure `area` contains:

- $\displaystyle\int_0^t l(s)\,ds$ evaluated as `area.node`;
- $\displaystyle\int_0^t q(s)\,ds$ evaluated as `area.queue`;
- $\displaystyle\int_0^t x(s)\,ds$ evaluated as `area.service`.

Program `ssq3` does not accumulate job-averaged statistics. Instead, the job-averaged statistics $\bar{w}$, $\bar{d}$, and $\bar{s}$ are computed from the time-averaged statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$ by using the equations in Theorem 1.2.1. The average interarrival time $\bar{r}$ is computed from the equation in Definition 1.2.4 by using the variable `t.last`. If it were not for the use of the assignment `t.arrival = INFINITY` to "close the door", $\bar{r}$ could be computed from the terminal value of `t.arrival`, thereby eliminating the need for `t.last`.

World Views and Synchronization

Programs `ssq2` and `ssq3` simulate exactly the same system. The programs work in different ways, however, with one clear consequence being that `ssq2` naturally produces *job*-averaged statistics and `ssq3` naturally produces *time*-averaged statistics. In the jargon of discrete-event simulation the two programs are said to be based upon different *world views*.* In particular, program `ssq2` is based upon a *process-interaction* world view and program `ssq3` is based upon an *event-scheduling* world view. Although other world views are sometimes advocated, process interaction and event-scheduling are the two most common. Of these two, event-scheduling is the discrete-event simulation world view of choice in this and all the remaining chapters.

Because programs `ssq2` and `ssq3` simulate exactly the same system, these programs should be able to produce exactly the same output statistics. Getting them to do so, however, requires that both programs process exactly the same stochastic source of arriving jobs and associated service requirements. Because the arrival times a_i and service times s_i are ultimately produced by calls to `Random`, some thought is required to provide this synchronization. That is, the random variates in program `ssq2` are always generated in the alternating order a_1, s_1, a_2, s_2, ... while the order in which these random variates are generated in `ssq3` cannot be known a priori. The best way to produce this synchronization is to use the library `rngs`, as is done is program `ssq3`. In Exercise 5.1.3, you are asked to modify program `ssq2` to use the library `rngs` and, in that way, verify that the two programs can produce exactly the same output.

5.1.4 MODEL EXTENSIONS

We close this section by discussing how to modify program `ssq3` to accommodate several important model extensions. For each of the four extensions you are encouraged to consider what would be required to extend program `ssq2` correspondingly.

Immediate Feedback

Given the function `GetFeedback` from Section 3.3, we can modify program `ssq3` to account for immediate feedback by just adding an `if` statement so that `index` and `number` are not changed if a feedback occurs following a completion of service, as illustrated.

```
else {                              /* process a completion of service */
  if (GetFeedback() == 0) {                 /* this statement is new */
    index++;
    number--;
  }
```

* A world view is the collection of concepts and views that guide the development of a simulation model. World views are also known as *conceptual frameworks*, *simulation strategies*, and *formalisms*.

For a job that is not fed back, the counter for the number of departed jobs (`index`) is incremented and the counter for the current number of jobs in the service node (`number`) is decremented. The simplicity of the immediate feedback modification is a compelling example of how well next-event simulation models accommodate model extensions.

Alternate Queue Disciplines

Program `ssq3` can be modified to simulate *any* queue discipline. To do so, it is necessary to add a dynamic queue data structure such as, for example a singly-linked list where each list node contains the arrival time and service time for a job in the queue, as illustrated.*

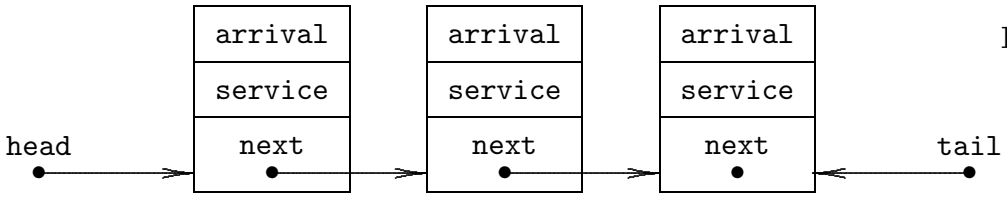


Figure 5.1.1. Queue data structure.

Two supporting queue functions, `Enqueue` and `Dequeue`, are also needed to insert and delete jobs from the queue, respectively. Then `ssq3` can be modified as follows:

- use `Enqueue` each time an arrival event occurs and the server is busy;
- use `Dequeue` each time a completion of service event occurs and the queue is not empty.

The details of this important modification are left as an exercise. This modification will result in a program that can be tested for correctness by using a FIFO queue discipline and reproducing results from program `ssq3`.

Note in particular that this modification can be combined with the immediate feedback modification illustrated previously. In this case, the `arrival` field in the linked list would hold the time of feedback for those jobs that are fed back. The resulting program would allow a priority queue discipline to be used for fed back jobs if (as is common) a priority assumption is appropriate.

Finite Service Node Capacity

Program `ssq3` can be modified to account for a finite capacity by defining a constant `CAPACITY` which represents the service node capacity (one more than the queue capacity) and declaring an integer variable `reject` which counts rejected jobs. Then all that is required is a modification to the "process an arrival" portion of the program, as illustrated.

* If this queue data structure is used, then service times are computed and stored at the time of arrival. In this way, each job's delay in the queue and wait in the service node can be computed at the time of entry into service, thereby eliminating the need to compute job-averaged statistics from time-averaged statistics.

```
if (t.current == t.arrival) {                /* process an arrival */
  if (number < CAPACITY) {
    number++;
    if (number == 1)
      t.completion = t.current + GetService();
  }
  else
    reject++;
  t.arrival     = GetArrival();
  if (t.arrival > STOP) {
    t.last      = t.current;
    t.arrival   = INFINITY;
  }
}
```

This code replaces the code in program `ssq3` for processing an arrival. As with the immediate feedback modification, again we see the simplicity of this modification is a compelling example of how well next-event simulation models accommodate model extensions.

Random Sampling

An important feature of program `ssq3` is that its structure facilitates direct *sampling* of the current number in the service node or queue. This is easily accomplished by adding a sampling time element, say `t.sample`, to the event list and constructing an associated algorithm to process the samples as they are acquired. Sampling times can then be scheduled *deterministically*, every δ time units, or *at random* by generating sampling times with an *Exponential*(δ) random variate inter-sample time. In either case, the details of this modification are left as an exercise.

5.1.5 EXERCISES

Exercise 5.1.1 Consider a next-event simulation model of a three-server service node with a single queue and three servers. (*a*) What variable(s) are appropriate to describe the system state? (*b*) Define appropriate events for the simulation. (*c*) What is the maximum length of the event list for the simulation? (Answer with and without considering the pseudo-event for termination of the replication.)

Exercise 5.1.2 Consider a next-event simulation model of three single-server service nodes in series. (*a*) What variable(s) are appropriate to describe the system state? (*b*) Define appropriate events for the simulation. (*c*) What is the maximum length of the event list for the simulation? (Answer with and without considering the pseudo-event for termination of the replication.)

Exercise 5.1.3 (*a*) Use the library `rngs` to verify that programs `ssq2` and `ssq3` can produce *exactly* the same results. (*b*) Comment on the value of this as a consistency check for both programs.

Exercise 5.1.4 Add a sampling capability to program `ssq3`. (*a*) With deterministic inter-sample time $\delta = 1.0$, sample the number in the service node and compare the average of these samples with the value of $\bar{l}$ computed by the program. (*b*) With average inter-sample time $\delta = 1.0$, sample *at random* the number in the service node and compare the average of these samples with the value of $\bar{l}$ computed by the program. (*c*) Comment.

Exercise 5.1.5 Modify program `ssq3` by adding a FIFO queue data structure. Verify that this modified program and `ssq3` produce *exactly* the same results.

Exercise 5.1.6[a] As a continuation of Exercise 5.1.5, simulate a single-server service node for which the server uses a shortest-job-first priority queue discipline based upon a knowledge of the service time for each job in the queue. (*a*) Generate a histogram of the wait in the node for the first 10000 jobs if interarrival times are *Exponential*(1.0) and service times are *Exponential*(0.8). (*b*) How does this histogram compare with the corresponding histogram generated when the queue discipline is FIFO? (*c*) Comment.

Exercise 5.1.7 Modify program `ssq3` to reproduce the feedback results obtained in Section 3.3.

Exercise 5.1.8 Modify program `ssq3` to account for a finite service node capacity. (*a*) Determine the proportion of rejected jobs for capacities of 1, 2, 3, 4, 5, and 6. (*b*) Repeat this experiment if the service time distribution is *Uniform*(1.0, 3.0). (*c*) Comment. (Use a large value of `STOP`.)

Exercise 5.1.9 (*a*) Construct a next-event simulation model of a single-server machine shop. (*b*) Compare your program with the program `ssms` and verify that, with a proper use of the library `rngs`, the two programs can produce *exactly* the same output. (*c*) Comment on the value of this as a consistency check for both programs.

Exercise 5.1.10[a] An M/M/1 queue can be characterized by the following system state change mechanism, where the system state is $l(t)$, the number of jobs in the node:

- The transition from state j to state $j + 1$ is exponential with rate $\lambda_1 > 0$ (the arrival rate) for $j = 0, 1, \ldots$.
- The transition from state j to state $j - 1$ is exponential with rate $\lambda_2 > 0$ (the service rate) for $j = 1, 2, \ldots$.

If the current state of the system is some positive integer j, what is the probability that the next transition will be to state $j + 1$?

As a continuation of the discussion in the previous section, in this section two next-event simulation models will be developed. The first is a next-event simulation model of a simple inventory system with delivery lag, the second is a next-event simulation model of a multi-server service node.

5.2.1 A SIMPLE INVENTORY SYSTEM WITH DELIVERY LAG

To develop a next-event simulation model of a simple inventory system with delivery lag, we make two changes relative to the model on which program `sis2` is based. The first change is consistent with the discussion of delivery lag in Section 3.3. The second change is new and provides a more realistic demand model.

- There is a lag between the time of inventory review and the delivery of any inventory replenishment order that is placed at the time of review. This delivery lag is assumed to be a *Uniform*$(0, 1)$ random variable, independent of the size of the order. Consistent with this assumption, the delivery lag cannot be longer than a unit time interval; consequently, any order placed at the beginning of a time interval will arrive by the end of the time interval, before the next inventory review.
- The demands per time interval are no longer aggregated into one random variable and assumed to occur at a constant rate during the time interval. Instead, individual demand instances are assumed to occur *at random* throughout the simulated period of operation with an average rate of λ demand instances per time interval. That is, each demand instance produces a demand for exactly one unit of inventory and the inter-demand time is an *Exponential*$(1/\lambda)$ random variable.

Figure 5.2.1 shows the first six time intervals of a typical inventory level time history $l(t)$.

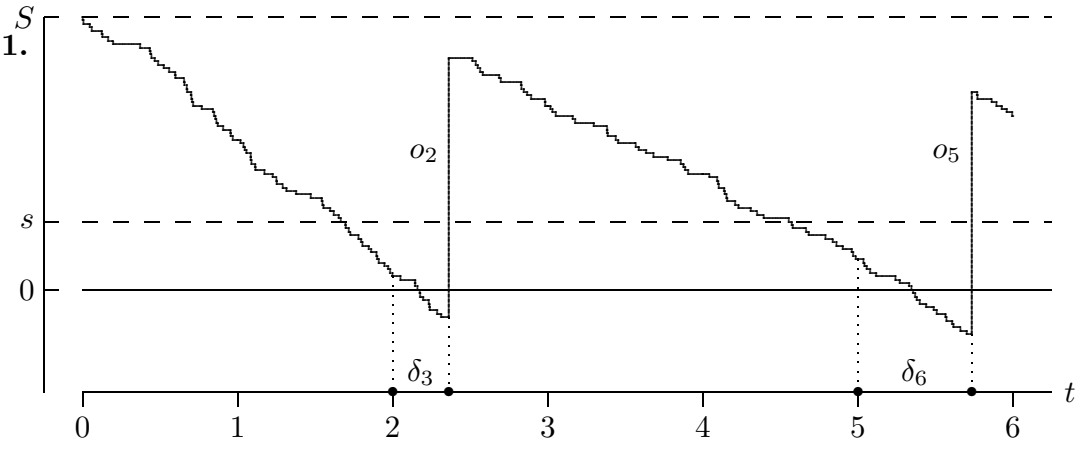


Figure 5.2.1. Inventory level history.

Each demand instance causes the inventory level to decrease by one. Inventory review for the i^{th} time interval occurs at $t = i - 1 = 0, 1, 2, \ldots$ with an inventory replenishment order in the amount $o_{i-1} = S - l(i-1)$ placed only if $l(i-1) < s$. Following a delivery lag δ_i, the subsequent arrival of this order causes the inventory to experience an increase of o_{i-1} at time $t = i - 1 + \delta_i$, as illustrated for $i = 3$ and $i = 6$ in Figure 5.2.1.

Recall that in program `sis2` the aggregate demand in each time interval is generated as an *Equilikely*(10, 50) random variate. Although the aggregate demand in each time interval can be any value between 10 and 50, within each interval there is nothing random about the occurrence of the individual demands — the inter-demand time is constant. Thus, for example, if the random variate aggregate demand in a particular interval is 25 then the inter-demand time throughout that interval is 0.04.

In contrast to the demand model in program `sis2`, it is more realistic to generate the inter-demand time as an *Exponential*$(1/\lambda)$ random variate. In this way the demand is modeled as an arrival process (e.g., customers arriving at random to buy a car) with λ as the arrival rate *per time interval*. Thus, for example, if we want to generate demands with an average of 30 per time interval then we would use $\lambda = 30$.

States

To develop a next-event simulation model of this system at the specification level, the following notation is used.

- The simulation clock (current time) is t and the terminal time is τ.
- At any time $t > 0$ the current inventory level is $l(t)$.
- At any time $t > 0$ the amount of inventory *on order* (if any) is $o(t)$.

In addition to $l(t)$, the new state variable $o(t)$ is necessary to keep track of an inventory replenishment order that, because of a delivery lag, has not yet arrived. Together, $l(t)$ and $o(t)$ provide a complete state description of a simple inventory system with delivery lag. Both $l(t)$ and $o(t)$ are integer-valued. Although t is real-valued, inventory reviews occur at integer values of t only. The terminal time τ corresponds to an inventory review time and so it is integer-valued.

We assume the initial state of the inventory system is $l(0) = S$, $o(0) = 0$. That is, the initial inventory level is S and the inventory replenishment order level is 0. Similarly, the terminal state is assumed to be $l(\tau) = S$, $o(\tau) = 0$ with the understanding that the ordering cost associated with increasing $l(t)$ to S at the end of the simulation (at $t = \tau$, with no delivery lag) should be included in the accumulated system statistics.

Events

Given that the state of the system is defined by $l(t)$ and $o(t)$, there are three types of events that can change the state of the system:

- a *demand* for an item at time t, in which case $l(t)$ will decrease by 1;
- an inventory *review* at (integer-valued) time t, in which case $o(t)$ will increase from 0 to $S - l(t)$ provided $l(t) < s$, else $o(t)$ will remain 0;
- an *arrival* of an inventory replenishment order at time t, in which case $l(t)$ will increase from its current level by $o(t)$ and then $o(t)$ will decrease to 0.

To complete the development of a specification model, the time variables t_d, t_r, and t_a are used to denote the next scheduled time for the three events inventory *demand*, inventory *review*, and inventory *arrival*, respectively. As in the previous section, ∞ is used to denote (schedule) an event that is not possible.

Algorithm 5.2.1 This algorithm is a next-event simulation of a simple inventory system with delivery lag. The algorithm presumes the existence of two functions `GetLag` and `GetDemand` that return a random value of delivery lag and the next demand time respectively.

```
l = S;                                  /* initialize inventory level */
o = 0;                                  /* initialize amount on order */
t = 0.0;                               /* initialize simulation clock */
t_d = GetDemand();                       /* initialize the event list */
t_r = t + 1.0;                           /* initialize the event list */
t_a = ∞;                                 /* initialize the event list */
while (t < τ) {
   t = min(t_d, t_r, t_a);                        /* scan the event list */
   if (t == t_d) {                     /* process an inventory demand */
     l--;
     t_d = GetDemand();
   }
   else if (t == t_r) {                /* process an inventory review */
     if (l < s) {
       o = S - l;
       δ = GetLag();
       t_a = t + δ;
     }
     t_r += 1.0;
   }
   else {                             /* process an inventory arrival */
     l += o;
     o = 0;
     t_a = ∞;
   }
}
```

Program sis3

Program `sis3` is an implementation of Algorithm 5.2.1. The event list consists of three elements `t.demand`, `t.review`, and `t.arrive` corresponding to t_d, t_r, and t_a respectively. These are elements of the structure `t`. Similarly, the two state variables `inventory` and `order` correspond to $l(t)$ and $o(t)$. Also, the time-integrated holding and shortage integrals are `sum.hold` and `sum.short`.

5.2.2 A MULTI-SERVER SERVICE NODE

As another example of next-event simulation we will now consider a *multi*-server service node. The extension of this next-event simulation model to account for immediate feedback, or finite service node capacity, or a priority queue discipline is left as an exercise. This example serves three objectives.

- A multi-server service node is one natural generalization of the single-server service node.
- A multi-server service node has considerable practical and theoretical importance.
- In a next-event simulation model of a multi-server service node, the size of the event list is dictated by the number of servers and, if this number is large, the data structure used to represent the event list is important.

Definition 5.2.1 A *multi-server service node* consists of a single queue, if any, and two or more servers operating *in parallel*. At any instant in time, the state of each server will be either *busy* or *idle* and the state of the queue will be either *empty* or *not empty*. If at least one server is idle, the queue must be empty. If the queue is not empty then all the servers must be busy.

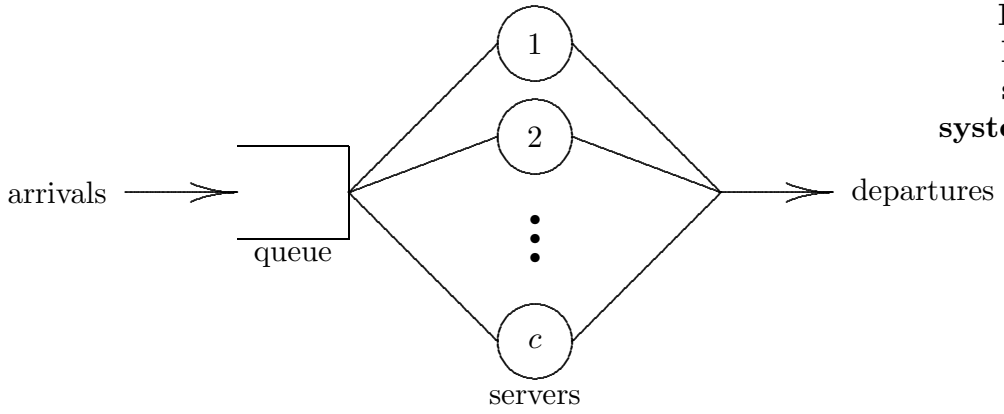


Figure 5.2.2. Multi-server service node system diagram.

Jobs arrive at the node, generally at random, seeking service. When service is provided, generally the time involved is also random. At the completion of service, jobs depart. The service node operates as follows. As each job arrives, if all servers are busy then the job enters the queue, else an available server is selected and the job enters service. As each job departs a server, if the queue is empty then the server becomes idle, else a job is selected from the queue to enter service at this server. Servers process jobs independently — they do not "team up" to process jobs more efficiently during periods of light traffic. This system configuration is popular, for example, at airport baggage check-in, banks, and roller coasters. Felt ropes or permanent dividers are often used to herd customers into queues. One advantage to this configuration is that it is impossible to get stuck behind a customer with an unusually long service time.

As in the single-server service node model, control of the queue is determined by the *queue discipline* — the algorithm used when a job is selected from the queue to enter service (see Section 1.2). The queue discipline is typically FIFO.

Server Selection Rule

Definition 5.2.2 A job may arrive to find two or more servers idle. In this case, the algorithm used to select an idle server is called the *server selection rule*.

There are several possible server selection rules. Of those listed below, the random, cyclic, and equity server selection rules are designed to achieve an equal utilization of all servers. With the other two server selection rules, typically some servers will be more heavily utilized than others.

- Random selection — select at random from the idle servers.
- Selection in order — select server 1 if idle, else select server 2 if idle, etc.
- Cyclic selection — select the first available server beginning with the successor (a circular search, if needed) of the last server engaged.
- Equity selection — select the server that has been idle longest *or* the idle server whose utilization is lowest.*
- Priority selection — choose the "best" idle server. This will require a specification from the modeler as how "best" is determined.

For the purposes of mathematical analysis, multi-server service nodes are frequently assumed to have *statistically identical, independent* servers. In this case, the server selection rule has no effect on the average performance of the service node. That is, although the utilization of the individual servers can be affected by the server selection rule, if the servers are statistically identical and independent, then the *net* utilization of the node is not affected by the server selection rule. Statistically identical servers are a convenient mathematical fiction; in a discrete-event simulation environment, if it is not appropriate then there is no need to assume that the service times are statistically identical.

States

In the queuing theory literature, the parallel servers in a multi-server service node are commonly called *service channels*. In the discussion that follows,

- the positive integer c will denote the number of servers (channels);
- the server index will be $s = 1, 2, \ldots, c$.

* There is an ambiguity in this server selection rule in that idle time can be measured from the most recent departure or from the beginning of the simulation. The modeler must specify which metric is appropriate.

As for a single-server node, the state variable $l(t)$ denotes the number of jobs in the service node at time t. For a multi-server node with distinct servers this single state variable does not provide a complete state description. If $l(t) \geq c$, then all servers are busy and $q(t) = l(t) - c$ jobs are in the queue. If $l(t) < c$, however, then for a complete state description we need to know which servers are busy and which are idle. Therefore, for $s = 1, 2, \ldots, c$ define

$$x_s(t) : \text{the number of jobs in service (0 or 1) by server } s \text{ at time } t,$$

or, equivalently, $x_s(t)$ is the state of server s at time t (with 0 denoting idle and 1 denoting busy). Finally, observe that

$$q(t) = l(t) - \sum_{s=1}^{c} x_s(t),$$

that is, the number of jobs in the queue at time t is the number of jobs in the service at time t minus the number of busy servers at time t.

Events

The $c+1$ state variables $l(t), x_1(t), x_2(t), \ldots, x_c(t)$ provide a complete state description of a multi-server service node. With a complete state description in hand we then ask what types of events can cause the state variables to change. The answer is that if the servers are distinct then there are $c + 1$ event types — either an arrival to the service node or completion of service by one of the c servers. If an *arrival* occurs at time t, then $l(t)$ is incremented by 1. Then, if $l(t) \leq c$ an idle server s is selected and the job enters service at server s (and the appropriate completion of service is scheduled), else all servers are busy and the job enters the queue. If a *completion of service* by server s occurs at time t then $l(t)$ is decremented by 1. Then, if $l(t) \geq c$ a job is selected from the queue to enter service at server s, else server s becomes idle.

The additional assumptions needed to complete the development of the next-event simulation model at the specification level are consistent with those made for the single-server model in the previous section.

- The initial state of the multi-server service node is empty and idle. Therefore, the first event must be an arrival.
- There is a terminal "close the door" time τ at which point the arrival process is turned off but the system continues operation until all jobs have been completed. Therefore, the terminal state of the multi-server node is empty and idle and the last event must be a completion of service.
- For simplicity, all servers are assumed to be independent and statistically identical. Moreover, equity selection is assumed to be the server selection rule.

All of these assumptions can be relaxed.

Event List

The event list for this next-event simulation model can be organized as an array of $c + 1$ event types indexed from 0 to c as illustrated below for the case $c = 4$.

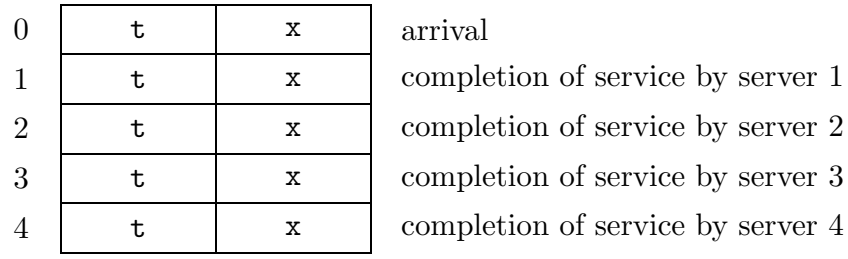


Figure 5.2.3. Event list data structure for multi-server service node.

The `t` field in each event structure is the scheduled time of next occurrence for that event; the `x` field is the current *activity status* of the event. The status field is used in this data structure as a superior alternative to the ∞ "impossibility flag" used in the model on which programs `ssq3` and `sis3` are based. For the 0^{th} event type, `x` denotes whether the arrival process is on (1) or off (0). For the other event types, `x` denotes whether the corresponding server is busy (1) or idle (0).

An array data structure is appropriate for the event list because the size of the event list cannot exceed $c + 1$. If c is large, however, it is preferable to use a variable-length data structure like, for example, a linked-list containing events sorted by time so that the next (most imminent) event is always at the head of the list. Moreover, in this case the event list should be partitioned into busy (`event[e].x = 1`) and idle (`event[e].x = 0`) sublists. This idea is discussed in more detail in the next section.

Program msq

Program `msq` is an implementation of the next-event multi-server service node simulation model we have just developed.

- The state variable $l(t)$ is `number`.
- The state variables $x_1(t), x_2(t), \ldots, x_c(t)$ are incorporated into the event list.
- The time-integrated statistic $\int_0^t l(\theta)\, d\theta$ is `area`.
- The array named `sum` contains structures that are used to record, for each server, the sum of service times and the number served.
- The function `NextEvent` is used to search the event list to determine the index `e` of the next event.
- The function `FindOne` is used to search the event list to determine the index `s` of the available server that has been idle longest (because an equity selection server selection rule is used).

5.2.3 EXERCISES

Exercise 5.2.1 Use program `ddh` in conjunction with program `sis3` to construct a discrete-data histogram of the total demand per time interval. (Use 10 000 time intervals.) (a) Compare the result with the corresponding histogram for program `sis2`. (b) Comment on the difference.

Exercise 5.2.2[a] (a) Modify program `sis3` to account for a $Uniform(0.5, 2.0)$ delivery lag. Assume that if an order is placed at time t and if $o(t) > 0$ then the amount ordered will be $S - l(t) - o(t)$. (b) Discuss why you think your program is correct.

Exercise 5.2.3 Modify program `sis3` so that the inventory review is no longer periodic but, instead, occurs after each demand instance. (This is transaction reporting — see Section 1.3.) Assume that when an order is placed, further review is stopped until the order arrives. This avoids the sequence of orders that otherwise would occur during the delivery lag. What impact does this modification have on the system statistics? Conjecture first, then simulate using `STOP` equal to 10 000.0 to estimate steady-state statistics.

Exercise 5.2.4 (a) Relative to program `msq`, provide a mathematical justification for the technique used to compute the average delay and the average number in the queue. (b) Does this technique require that the service node be idle at the beginning and end of the simulation for the computation of these statistics to be exact?

Exercise 5.2.5 (a) Implement a "selection in order" server selection rule for program `msq` and compute the statistics. (b) What impact does this have on the system performance statistics?

Exercise 5.2.6 Modify program `msq` so that the stopping criteria is based on "closing the door" after a fixed number of jobs have entered the service node.

Exercise 5.2.7 Modify program `msq` to allow for feedback with probability β. What statistics are produced if $\beta = 0.1$? (a) At what value of β does the multi-server service node saturate? (b) Provide a mathematical justification for why saturation occurs at this value of β.

Exercise 5.2.9 Modify program `msq` to allow for a finite capacity of r jobs in the node at one time. (a) Draw a histogram of the time between lost jobs at the node. (b) Comment on the shape of this histogram.

Exercise 5.2.10 Write a next-event simulation program that estimates the average time to complete the stochastic activity network given in Section 2.4. Compute the mean and variance of the time to complete the network.

5.3 Event List Management

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 a 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.

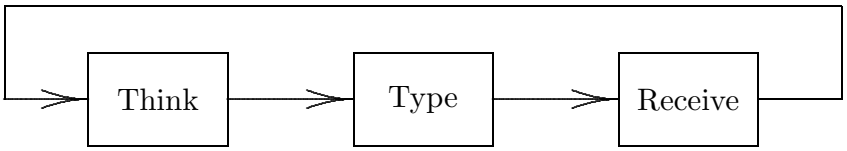
5.3.3 EXAMPLE

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 ith event structure stores the time of the event for the ith user ($i = 0, 1, \ldots, n-1$). 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.

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

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.

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

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 < τ) {                          /* 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.

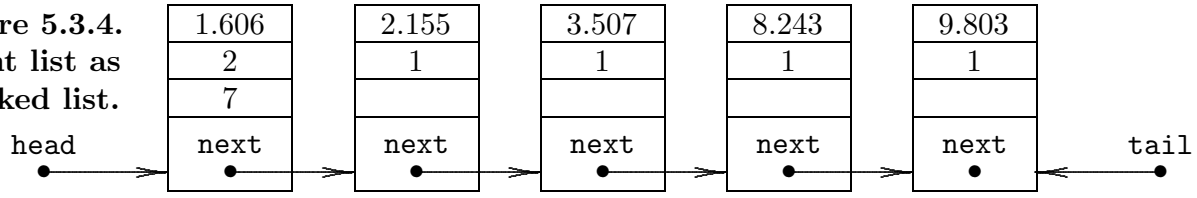


Figure 5.3.4. Event list as a linked list.

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.

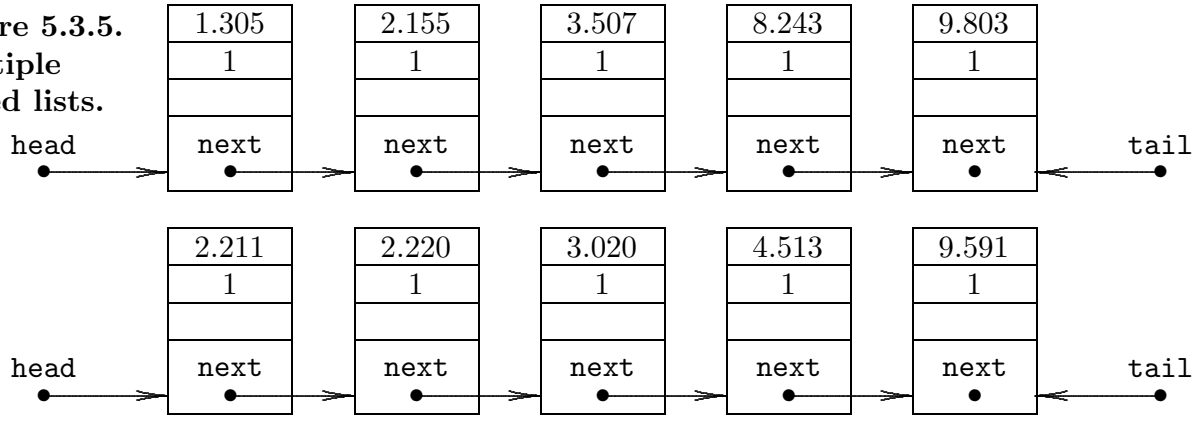


Figure 5.3.5. Multiple linked lists.

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.

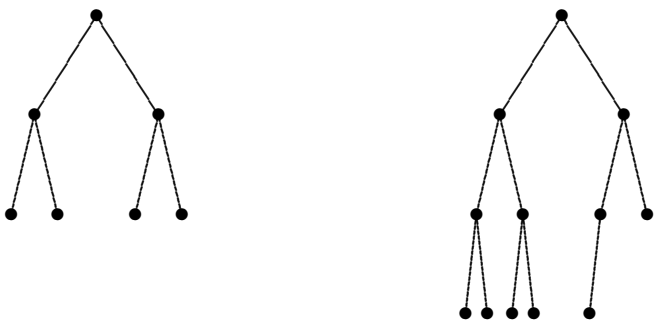


Figure 5.3.6. Full and complete binary trees.

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.

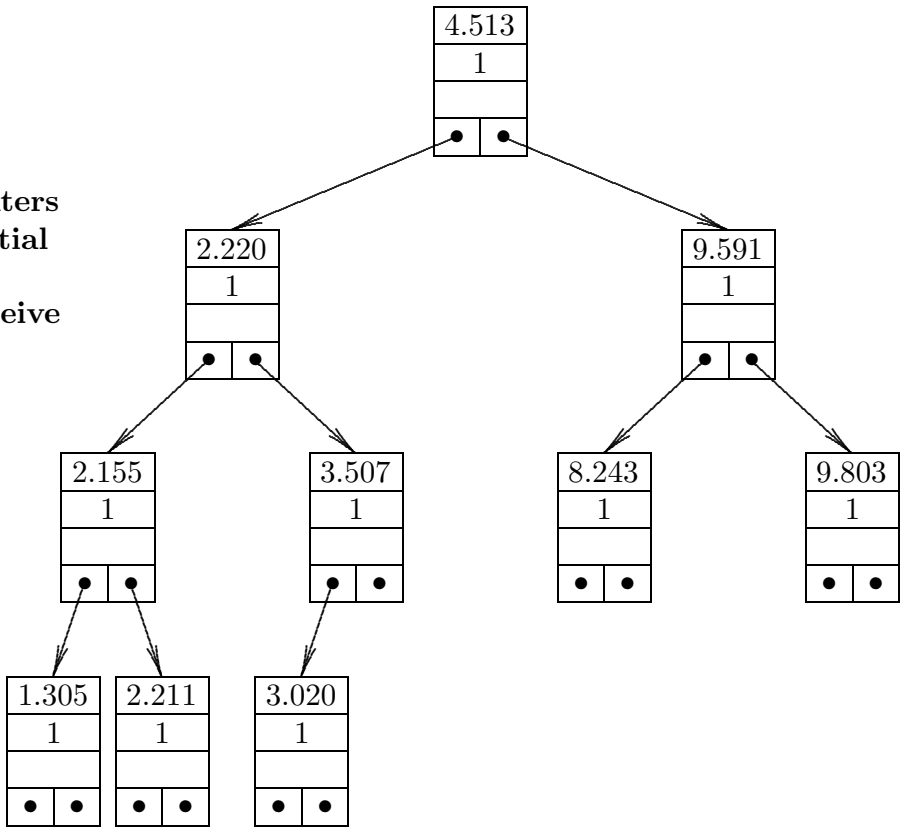


Figure 5.3.7. Binary search tree with pointers for the ten initial events in the think-type-receive model.

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.

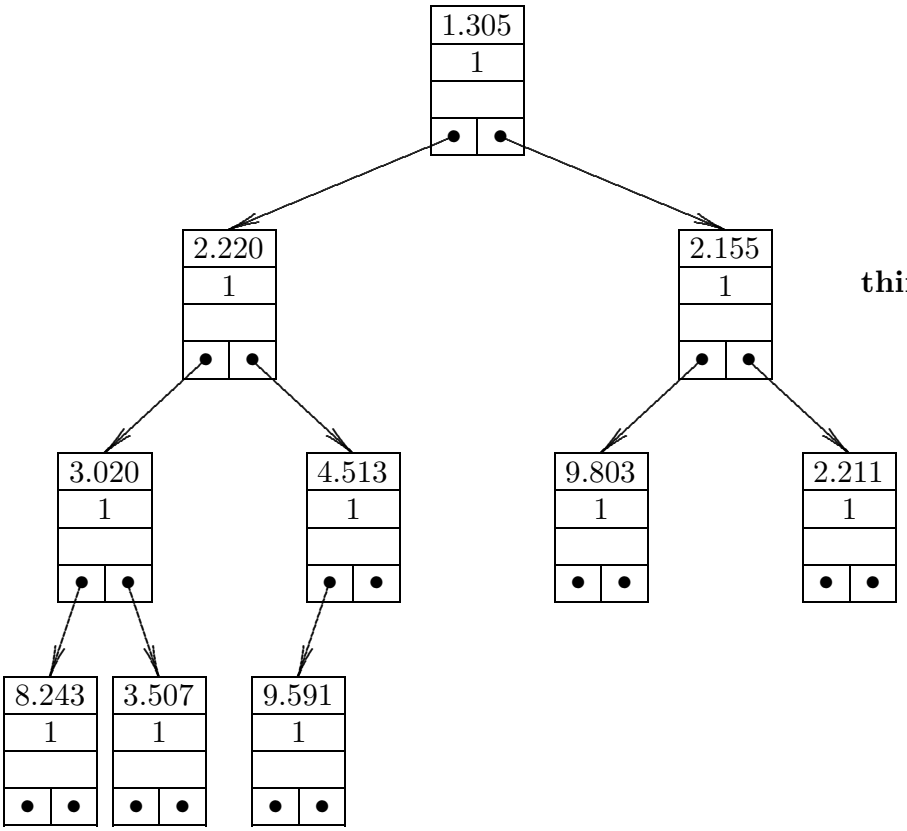


Figure 5.3.8. Heap with pointers for the ten initial events in the think-type-receive model.

* 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.

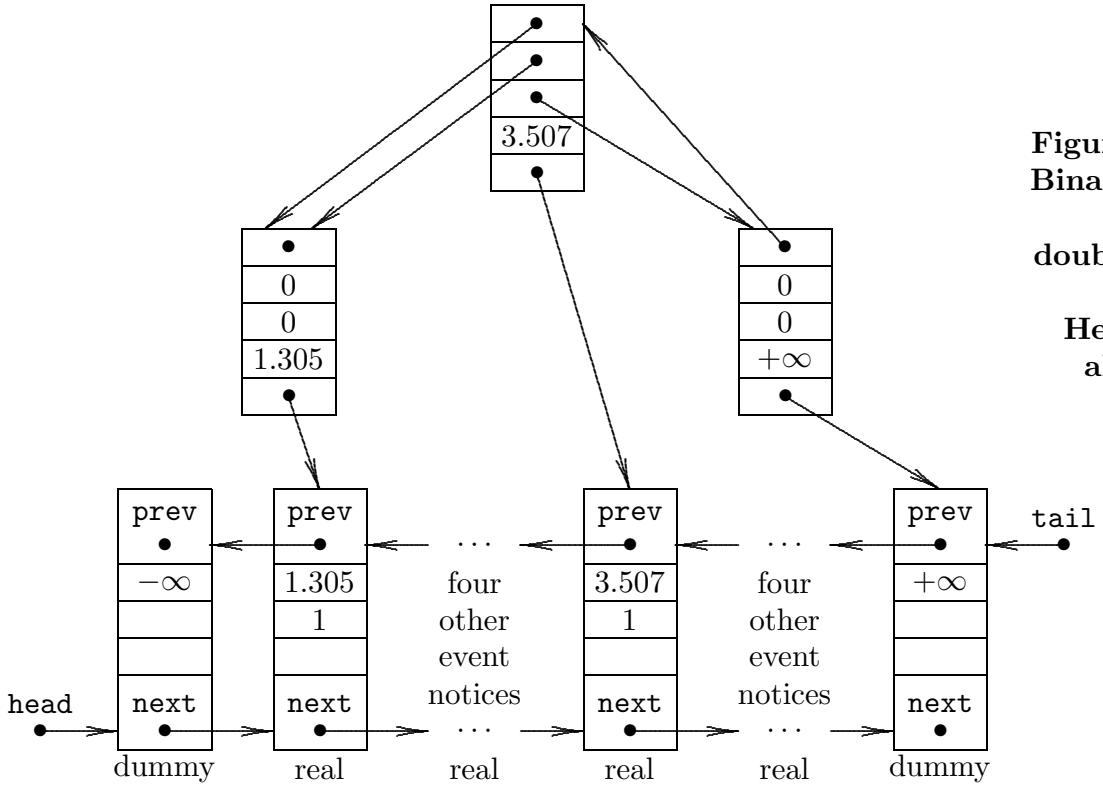


Figure 5.3.10. Binary search tree and doubly-linked list for Henriksen's algorithm.

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 lth 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.