

**EE/CpE 345**

**Modeling and Simulation**

**Fall 2003**

**Class 3**

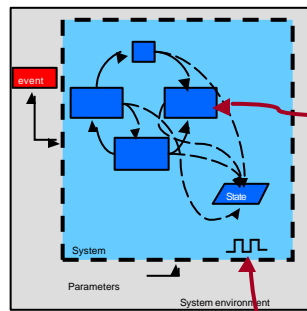
Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

54

## Today's topics

- Text – Chapter 3 – General Principles

## Concepts in Discrete-Event Simulation



- System:** set of objects, joined to accomplish some purpose
  - Model:** Abstraction of real system, defines relationships between entities, system parameters, system state, etc.
  - State of system:** collection of variables necessary to describe system at any time
  - Entity** object of interest in system
  - Attribute** property of an entity
  - Event:** Instantaneous occurrence that may be associated with change of system state
  - Activity** a predefined set of actions by system objects, usually in a specified time period
- 
- List:** a collection of associated entities, ordered in some logical fashion
  - Event notice:** Record of an event to occur at some present or future time, along with associated data
  - Event list:** List of event notices (Future Event List)
  - Delay:** Duration of time of unspecified indefinite length, not known until it ends
  - Clock:** A system variable representing simulated time

EE/CptE345: Modeling and Simulation  
Fall 2003

Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

56

Previously, we defined several standard terms used in discrete event simulation. This week, we will add a few more, specifically those used in the implementation of a simulation.

The **list** is a standard data structure used in computer science. In simulations, we can use this data structure to organize the data used in the simulation. The primary characteristic of the list is order, that is the fact that a list can convey sequence of data elements. In particular, in simulation, we can use a list to organize event notices.

Since the simulation is a discrete event simulation, it makes sense to organize the simulation around the events that drive it. As we discussed previously, a grocery store check-out, for instance, could be simulated considering the events of customer arrivals or service completion. So, the **event notice** is the record of an event – it could be the current event, in the present, or some yet to happen future event. Besides the time of the event, the event notice records data associated with the event, perhaps what type of event it is (arrival or departure) or which entity is involved in the event, e.g., which server.

The list of all event notices is the **event list**. In general, as the simulation runs, we will be interested in tasking the system to deal with events which have not happened yet, so the Future Event List will most often be the event list of interest. **[Discussion topic:** Why might the Future Event List be more interesting than the list of previous events?]

## Able and Baker, in our current context

- System state:
  - $L_Q(t)$  – the number of customers waiting to be served at time,  $t$
  - $L_A(t) = (0,1)$ , the idle/busy status of Able at time,  $t$
  - $L_B(t) = (0,1)$ , the idle/busy status of Baker at time,  $t$
- Entities:
  - Able and Baker are the only entities we tracked last time
- Activities:
  - Inter-arrival time
  - Able's service time
  - Baker's service time
- Delay
  - Customer waiting time until serviced by Able or Baker

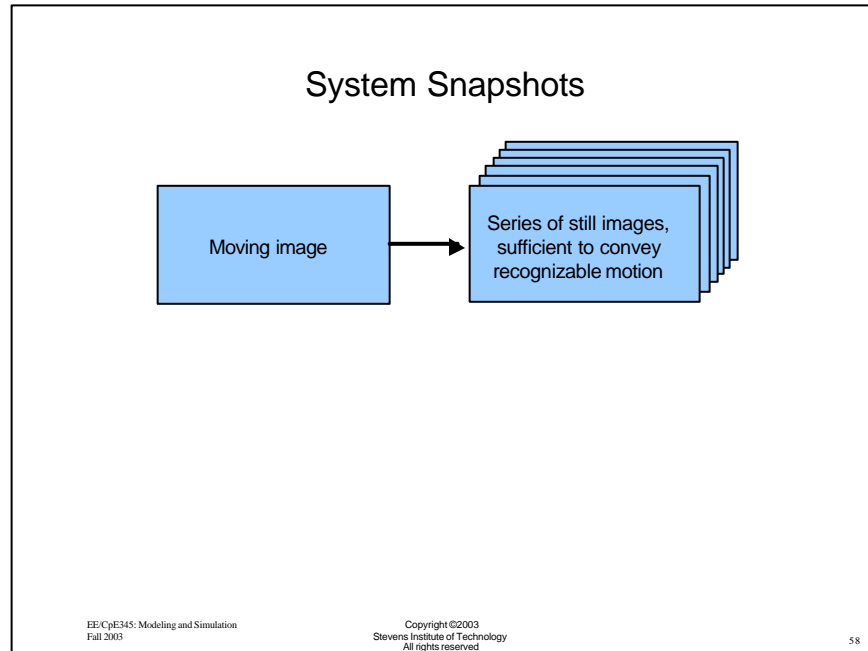
Referring back to the Able/Baker car hop simulation discussed last week, we can use that example to illustrate specific parameters of the simulation.

Remember that **System State** is the collection of variables needed to specify the condition of the system at any point of time. For the Able/Baker example, the only system state information we tracked was the number of waiting customers and the busy/idle condition of the two servers. So, we can define three variables as functions of time that can define the system state.  $L_{(sub)Q}$  is the number of customers waiting in the queue at any point of time.  $L_{(sub)Q}$  will be a nonnegative integer.  $L_{(sub)A}$  and  $L_{(sub)B}$  are variables that capture the condition of the two servers at any point of time. Since the servers could be Idle or Busy, we can use a binary  $\{0, 1\}$  to describe the {Idle, Busy} condition of the server.

We need to describe the **entities** in the system. Since we are not tracking individual customers, we don't have to define them as entities – all we care about customers is captured in the state variable that tells us the number waiting in the queue. On the other hand, since we are concerned about the {idle, busy} status of the servers, they would be the only entities in the system.

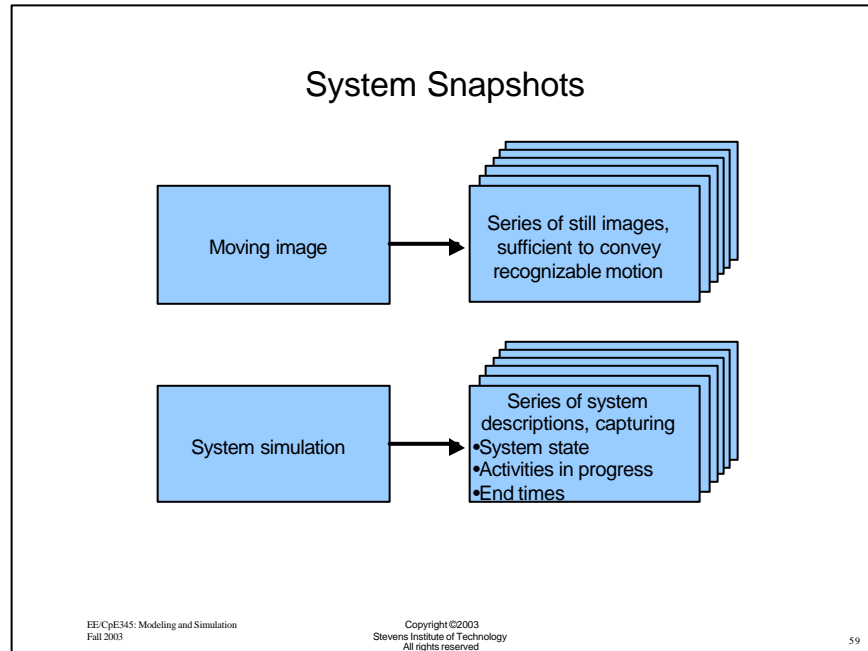
**Activities** for this simulation are customer arrivals, and service completion by the servers, so we must track the interarrival times and the server completion times.

Customers waiting in the queue are inconvenienced for the length of time that they must wait without being served, so this waiting time is a **delay** in the system that would generally need to be monitored.



The parameters described above are, in general, continually changing, so it is necessary to find a way to find a way to describe the system in static terms that allow it to be studied, while capturing the dynamic character of the system. For this we use **System Snapshots**.

The best way to understand system snapshots is by analogy. Consider a moving image, perhaps the real world as one might look at it through a window. When we want to record this image, it is not practical to capture the actual changing image. Instead, a video or film motion picture camera captures a series of still images of the scene. Although for each image, the scene is fixed, when we look from frame to frame, it is possible to see that the relative positions of objects have changed. If the rate of capture of fixed images is fast compared with the speed of movement of objects in the scene, and if the interval between each of the images is short compared to the response time of the human visual system, this series of fixed images gives the illusion of movement of the objects in the scene.



The parameters described above are, in general, continually changing, so it is necessary to find a way to describe the system in static terms that allow it to be studied, while capturing the dynamic character of the system. For this we use **System Snapshots**.

The best way to understand system snapshots is by analogy. Consider a moving image, perhaps the real world as one might look at it through a window. When we want to record this image, it is not practical to capture the actual changing image. Instead, a video or film motion picture camera captures a series of still images of the scene. Although for each image, the scene is fixed, when we look from frame to frame, it is possible to see that the relative positions of objects have changed. If the rate of capture of fixed images is fast compared with the speed of movement of objects in the scene, and if the interval between each of the images is short compared to the response time of the human visual system, this series of fixed images gives the illusion of movement of the objects in the scene.

Now, let's compare this motion picture image with records of the system descriptions used in a system simulation. By recording the system state, the activities in progress, and the end times of those activities, we have enough information to fully recreate the running simulation.

## Example system snapshot at time t

Clock	System State	Entities and Attributes	Set 1	Set 2	...	Future Event List (FEL)	Cumulative Statistics and Counters
t	(x,y,z,...)		Queue memberships			(3, t <sub>1</sub> ) (1, t <sub>2</sub> )	
t <sub>1</sub>	(x-1, y, z', ...)					(1, t <sub>2</sub> )	
...							

System state = (x, y, z, ...) means:

state variable 1 = x, (e.g., customers waiting = x)

state variable 2 = y, (e.g., Able is busy)

state variable 3 = z, ... (e.g., Baker is idle)

Future Event List = {(3,t<sub>1</sub>), (1,t<sub>2</sub>)} means:

A type 3 event will occur at t=t<sub>1</sub> (e.g., Baker service completion at t<sub>1</sub>)

A type 1 event will occur at t=t<sub>2</sub> (e.g., customer arrival at t<sub>2</sub>)

EE/CptE345: Modeling and Simulation  
Fall 2003

Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

60

Consider a representative system being simulated and let's look at the series of system snapshots: First, look at the **Clock** column. The clock is recording the current time while the simulation is running. Since this is a discrete event simulation, it is only at discrete points of time (t, t<sub>1</sub>, t<sub>2</sub>, ...) that we care about. When customer arrivals or service completions occur, we *must* record the time. In between, nothing externally visible is happening. Yes, customers may be planning on going to the server, but they haven't acted. The server may be in the midst of processing the last customer who arrived, but we see no external evidence of that server action. It is only when the customer actually arrives or the server completes that we consider events to have happened.

Next, look at the **system state** column. An event is the only thing that can change the system state, so we only need to record system state at the times events occur. If we have only arrival events and server completion events to consider, there are only a small number of changes to system state that can occur. A customer arrival will increase the number of customers in queue. It might also change a server state from idle to busy if there were previously no customers waiting. A service completion might change the server state from busy to idle or it might reduce the number of customers in the queue. Depending on the level of detail we want to capture, the **Entities and Attributes** and the **Set #i membership** columns may or may not be used. The former would track the objects we are watching in the system and their current condition while, if we were interested in following a particular customer through the system to see which server they were served by, etc., we could record that information under the queue membership. The most important field in this system snapshot is the **Future Event List**. On an ongoing basis, this column is used to do the bookkeeping for the simulation to be sure that upcoming events are properly scheduled. The FEL records a series of data elements, tracking the type of future event (e.g., an arrival, a departure) and when it is to occur. As several arrivals could happen before a departure (server completion) could happen, the FEL can grow to any arbitrary length.

Finally, we can capture information in the system snapshot to track the ever changing cumulative statistics (e.g., average queue length) as well as system counters (e.g., number of customers served)

## Event-scheduling/time-advance algorithm

Clock	System State	...	Future Event List	...
t	(5,1,6)		(3,t <sub>1</sub> ) (1,t <sub>2</sub> ) (1,t <sub>3</sub> ) ... (2,t <sub>4</sub> )	

1. Remove event notice for imminent event (at  $t=t_1$ )
2. Advance CLOCK to imminent event time
3. Execute imminent event  

Update system state, change entity attributes, set membership, as needed
4. Generate future events, if needed, and place them on FEL, ranked by time of occurrence
5. Update cumulative statistics and counters

Clock	System State	...	Future Event List	...
t <sub>1</sub>	(5,1,5)		(1,t <sub>2</sub> ) (4,t <sub>1</sub> *) (1,t <sub>3</sub> ) ... (2,t <sub>4</sub> )	

EE/CptE345: Modeling and Simulation  
Fall 2003

Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

61

Consider a series of system snapshots for a system being simulated. The event scheduling/time advance algorithm defines how the system changes as we move from one system snapshot to the next.

For this discussion, we will ignore many of the fields defined in the system snapshot and focus on three that are particularly important for following the system simulation: Clock time, system state and the future event list. Look at two views of the system snapshot at  $t$  and  $t_1$  (the time when the next event occurs). If the system is in a particular state at time  $t$ , here (5,1,6), with a particular set of future events listed in the future event list, we can describe what the future event list and system state must be at the next relevant point in time, here  $t_1$ . Note that the definition of system state in this example does not correspond to the definition on the previous slide. Here we might be capturing other state variable information about queue status and server state than was used for the Able/Baker simulation.

To move from  $t$  to the next time of interest, the algorithm follows a few specific steps: First, the first item on the FEL is the next event that is to occur - the so-called imminent event. Here, the FEL tells us that the next event will happen at  $t_1$ . To keep the simulation on track, this event is removed from the list and stored in a temporary location for execution/interpretation. The simulation variable that tracks time, the clock, is advanced to  $t_1$  to prepare for the event

Next, the imminent event is executed. This means that if it was a customer arrival, for instance, the new customer is put on the queue. If the event was a service completion, a customer is removed from the queue or the server is put into the idle state. In any case, the system state is changed and, if we are tracking this, the list of customers waiting is updated. If a customer arrival was the imminent event, we need to determine when the next customer will be arriving, based on the interarrival times. In this case, a future event is scheduled at the appropriate time and put on the FEL. If the imminent event was a service completion, then another customer will be served. Their service completion time must be determined and their departure scheduled and put on the FEL. In any case, the time ordering of the FEL must be maintained so that the first event on the list is always the next one scheduled.

Finally, we can now update system statistics, e.g., average waiting time. The scheduling/advance algorithm then restarts, preparing to execute the next event on the FEL.

## Actions occurring on the FEL

- Deleting the imminent event (e.g., servicing a customer in queue)
- Removing events that have already been scheduled (e.g., if customer can leave a queue)
- Adding future events as they get scheduled
- Maintaining time-ordering of events

Length and contents of FEL changes dynamically during simulation

How can you efficiently (in terms of storage and processing efficiency) maintain information about the FEL in a simulation?

Listed here are the actions that were performed on the FEL in the advance/scheduling algorithm. We must maintain a time ordering of events to allow simple processing of the simulation. If we had to search for the next event in the FEL, this would impede the efficiency of the simulation. Besides the generation of random numbers, the maintenance of the FEL is one of the most frequent activities that is performed in a system simulation.

For these reasons, the data structure that stores the FEL must define the ordering of data elements. It should allow efficient additions and deletions without requiring a great deal of overhead.

As we will see, an efficient data structure to perform the processing required for the FEL is something known as a linked list.

## Considerations in running an event-scheduling/time-advance simulation

- System snapshot at  $t=0$ 
  - Preset internal values for simulation – initial conditions
  - External controls – “exogenous” event – from the system environment
- Bootstrapping
  - Used for creating an external event stream, e.g., customer arrivals.
  - Events are placed on FEL by calculating offset from present simulation time.
- Simulation stopping conditions
  - Run for a preset time (e.g., 1000 seconds of simulator time)
  - Run for a preset number of events (e.g., 1000 customers have been served)
  - Run for a preset real-world time (e.g., until the conference paper is due to be sent)
  - Run until an internal event occurs (e.g., the receiver synchronizes)
  - Run until an external event occurs (e.g., user decides to modify experiment)

When we run a simulation using the scheduling/advance algorithm, there are several considerations we must address.

- (1) The initial state of the simulation must be established. We must create a system snapshot at  $t=0$ . We might start the simulation with servers idle and no users in the queue (e.g., like a grocery store opening at 8 am). Or, if we are simulating a continuously running systems (e.g., a 24 hour convenience store), we might choose to initialize the simulation with some expected average state – perhaps a number of customers waiting.
- (2) As we create events, we need to consider if we will be dealing with the system in absolute time or relative time. Since the statistics of customer arrivals are often defined in interarrival times, it is often most convenient to look at the next event in time relative to the current time. We use the concept of **bootstrapping** to define the system as a series of deltas from the current views, rather than in absolute terms. This also allows us to pick up the simulation at any point of time – we don’t need to initialize the simulation at zero initial conditions and run to a particular time, but can instead work from any given time forward.
- (3) How the simulation will be terminated needs to be addressed. If we are going to be measuring things like average queue length, there must be a time when we decide to stop the simulation to determine a value for the statistic. There are several methods of deciding when to stop the simulation. We might base it on time, events, or conditions. These might be internal to the simulation or external to the simulation. I have listed a few here that are commonly used.

## World Views

- System model is developed differently, based on "world view" employed:
  - Event scheduling
    - Events (the only things that can affect system state) are the prime focus of simulation
  - Process-interaction
    - A process is the life cycle of one entity in simulation
    - Entities:
      - request resources = *events* and
      - use them = *activities*
    - Well suited to object oriented languages (e.g., C++)
      - Entities and processes = objects
  - Activityscanning
    - Simulation runs in fixed time increments
    - Rule-based approach to decide if an activity can begin at any given point in simulated time
    - Events are quantized to the cycle time of the simulation

When one develops a simulation model, it is important to keep in mind how the system interaction with its environment will be viewed. This is known as the **world view**.

The previous slides have used a world view based on event scheduling. Since events (e.g., customer arrivals, service completions) are the only things that change system state, we can focus the simulation around how we deal with these events.

Another world view is to think of the system as a series of processes that interact with each other. Here, entities like customers present themselves to the system requesting services. We track the activities of the customer as they enter the system, wait and are served. Instead of abstractly looking at customers as the state of the queue, we can simulate more complex interactions of customers with the system and, perhaps, with each other. In this case, the simulation code would spawn multiple processes and track them as they run to completion through their life-cycle. Object-oriented programming languages like C++, Smalltalk, and Modula are well suited to these types of simulations.

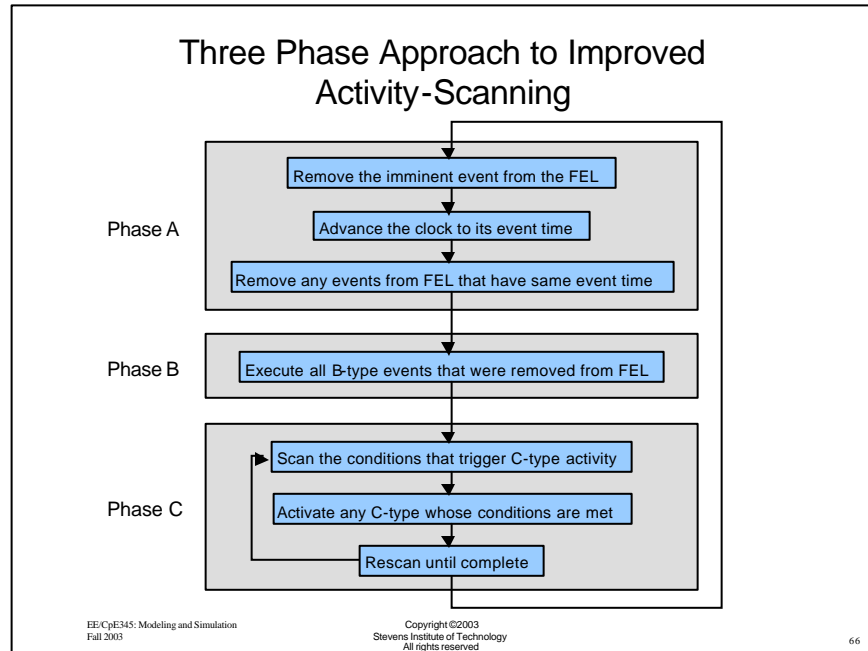
Another world view is to use activity-scanning. In this case, the simulation does not jump from one event time to another. Rather, it is continually looping, stepping through simulation time in small increments. During each time step, the simulation must determine which activities should be started in the next time interval.

## World Views - continued

- Activity-scanning costs considerable simulation overhead to decide if an activity can be started
- Hybrid activity-scanning/event-scheduling approach:
  - Two types of activities:
    - B activities:
      - Must occur.
      - Primary events (completion of activity) and unconditional activities
      - Can be scheduled in advance
    - C activities:
      - Conditional events or activities
      - Depend on conditions being true
      - Scan for them at the end of a time advance

As in all situations where there are choices to be made, picking one world view over another involves tradeoffs. Event scheduling is efficient in terms of processing the arrival and departure events, since the simulation jumps from one event to the next. However, there may be other continuously running background activities in the system that are not modeled properly by event scheduling. For instance, if we were modeling a cookie factory where dough is continuously turned into cookies, with quality control inspections and packaging happen as discrete events, there might be no convenient way to deal with a variable cookie creation rate. On the other hand, while we could run an activity scanning simulation with periodic updates of the number of cookies produced, there is a lot of overhead to determine if it is time to schedule an event.

One way to deal with this tradeoff is to use a hybrid scheme: activity scanning plus event scheduling. Here, activities are classified into two types: B activities are those that must occur. They might be so-called primary events – the completion of an activity like a customer being served, or they might be unconditional activities that must run during each interval. C activities are conditional events or activities. In the cookie factory example, the packaging of a box of cookies depends on (a) the production of enough cookies and (b) the acceptance of those cookies by the quality control inspector, who is rejecting misshapen cookies. Only when the conditions are met can the event of wrapping a box be scheduled.



The diagram above illustrates the three phase approach to an improved activity scanning algorithm. The steps are:

Phase A: In activity scanning mode, remove the imminent event, advance the clock to the event time, and remove any other events that are scheduled at the same time (or nearly the same time, within the quantization of the clock)

Phase B: Execute all the Type B activities – those which are unconditional and must be executed at their proper time (e.g., a service completion)

Phase C: Decide if the type C activities have their conditions met to be executed. If they have, execute them. If execution of one C type event changes the conditions that would allow another C event to be executed, keep iterating until all the C events that can be executed have been.

Repeat the A/B/C process

## List Processing

- Methods to handle lists of entities and the FEL
- Properties and operations on lists:
  - A list is a set of records with an ordering (ranking) operation, accessible in order
  - One item in a list is a "record" – used to store one entity or one event notice.
  - The head

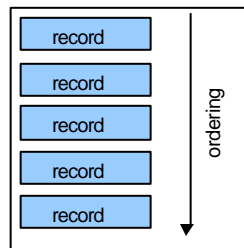
In the case of the simple event scheduling algorithm and the more complex hybrid algorithm, it is clear that there will be lots of bookkeeping to be done to run the simulation. We need to have an efficient way to manage these lists of events and other items, so we need to look at list processing.

There are a few properties of lists that we will consider, including the definition of a list. The most important consideration of the list is that there is an implicit ordering of items in the list. In the particular case of the future event list, time ordering is most important – the FEL will be accessed in order of the next event. Besides ordering, the next property of the list that we must look at is the storage of information. The individual records in a list each store one item, e.g., one event notice. The records in a list do not necessarily have to be simple data, like a single number, but can be richer data records. They might include multiple fields, for example besides the time of the next event, the FEL records would also store the type of event.

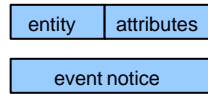
Finally, when we discuss list processing, we define the **head** of the list – the first element on the list and the next one to be processed.

## List processing

A generic list

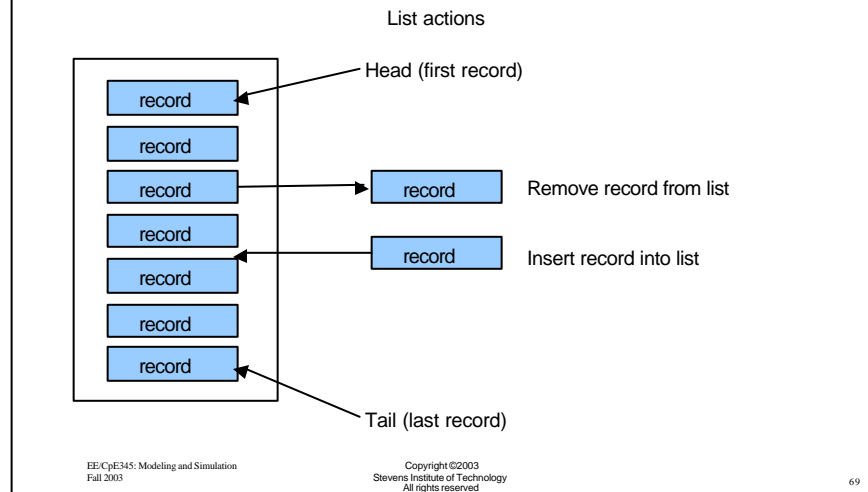


Sample list records for simulation



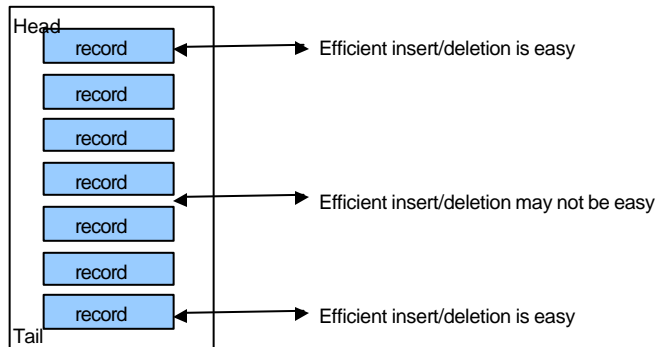
The slide above illustrates this information graphically. On the left, a generic list is shown, with a series of individual records with some implied ordering of those records. On the right is an illustration of what might be in one of the records: the name of an entity and its attributes, or the description of an event notice, generally consisting of time of the event and event description.

## List processing



For the generic list above, let's consider some of the things we might want to do with the list: We want to be able to easily access the first record (the imminent event) of the list. As future events need to be scheduled, we need to be able to insert a record anywhere into the list. If we allow customers to leave their queue or if there are other events that might need to be deleted in the middle of the list, we may require means to delete list records at arbitrary points in the list. All of these operations must be performed efficiently, especially if they are performed often during the simulation.

## List processing



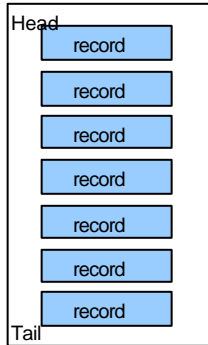
EE/CptE345: Modeling and Simulation  
Fall 2003

Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

70

Depending on how we implement a list data structure, we might find that some operations are harder or easier than others. In particular, adding or deleting items at the beginning or end of the list is generally relatively easy. On the other hand, it may or may not be efficient to do insertions and deletions in the middle of the list. We will be looking at data structure implementation that make the most important operations as easy as possible.

## List processing - data structures



- FORTRAN:
  - A list is an array. Successive records are in contiguous locations in memory
  - Shrinking/expanding lists may not make efficient use of memory
- C:
  - Use a structure for a linked list
  - Use malloc() for variable size structures
- C++
  - Use classes for lists
  - Allocate memory as needed
- LISP, SNOBOL, others
  - List processing is an inherent feature of language
- MatLab
  - wasn't designed for list processing, but structured data types simplify list processing
  - memory allocation is automatic

EE/CptE345: Modeling and Simulation  
Fall 2003

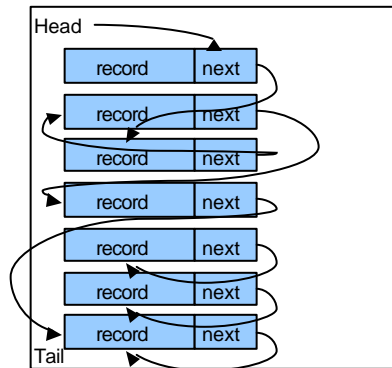
Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

71

Different programming languages have been implemented with different needs in mind. Languages like LISP (which stands for LIST Processing) and SNOBOL are particularly well suited for generic list processing, since these were design considerations for the language. FORTRAN, particularly the earlier versions, is not generally well suited for list processing. The variety of data structures that are built into the language are limited and lists are often implemented as arrays. With fixed storage used for the list, it is not easy to expand a list, adding elements, and deleting elements from a list may not free up memory.

Languages like C have general purpose data types like structures that can be used to easily implement lists. With dynamic run-time memory allocation, the data structure size can be changed as needed to accommodate larger or smaller lists.

## List processing - data structures, implementation with linked lists



Insertion of record M after record L, before N:  
Set M's "next" to record where L pointed  
Redirect "next" of record L to M

Deletion of record V, after U, before W:  
Change V to point to W, not U  
U's "next" is arbitrary. Set to "null", U,  
or any other useful value

Given R, finding Q, S, such that  $Q < R < S$   
may require search of all records up to S

The linked list data structure is often the best data structure to use for implementing a list that requires arbitrary additions and deletions. Unlike an array, which has a fixed order of data and no flexibility to move things around, the linked list stores with each data record a pointer to the "next" record. In this way, data can be added or deleted without actually moving data. If we need to delete the third element in the list, we can do so by simply skipping it. The second element in the list now points to the fourth, and it is as if the third element never existed. If we want to add an item to the list, let's say, right after the fifth, we can put the new item anywhere there is a free spot for it. The fifth item, instead of pointing to the sixth is now redirected to point to the new sixth element. The new element is set to point to the element that was the sixth element, and the list is still in order with minimal work.

We have to have a way to mark the last item on a list that is dynamically changing. If we just left the last item pointing to an arbitrary location, we would have no way to know it was the last, unless we separately maintained a count of the number of items in the list. An easier way to mark the end of the list, and the one that is used by standard convention, is to have the last item on the list point to itself. Normally, this would never make sense, but if we detect that an element is pointing to itself, we can use this to define the list end.

## List processing with structures in Matlab

```
FEL(1).event='arrival'  
FEL(1).time=5  
FEL(1).next=13  
  
FEL(13).event='arrival'  
FEL(13).time=8  
FEL(13).next=5  
  
FEL(5).event='departure'  
FEL(5).time=9  
FEL(5).next=5
```

Matlab was not designed with list data types being required, but the sample code above illustrates how a language, like Matlab, can use data structures to implement lists.

A data structure called FEL is an array data structure. This means that it has multiple elements, like an array, and each element has multiple parts, like a data structure. Here, the parenthetical number denotes the position in the array. The variable FEL has parts called “event,” “time,” and “next” to store the data needed for the list. So, the first three lines mean:

“the first item in the list is of event type ‘arrival.’ It is scheduled to occur at t=5. After this list item, the next list item in number 13”

In this way, we can follow the links through the list to see that the FEL described above is:

Arrival at t=5

Arrival at t=8

Departure at t=9

If we wanted to delete the t=8 arrival, only one change would be needed:

FEL(1).next = 5.

**[Discussion topic:** what is done with the “arrival” and “t=8” information that was stored in this data element? How do we ensure that it will not cause confusion later?]

## List processing enhancements

- A linked list must be searched from the beginning. The average search time to find where to put a record is on the order of  $N/2$  for an  $N$  record list.
- A doubly linked list includes “previous” record pointers, as well as “next” record pointers.
- Adding a “middle” pointer, besides the “head” and “tail” pointers allows faster traversal of a doubly linked list from the middle element. However, there is additional bookkeeping as records get added/deleted and the list center changes.
- “Hash tables” allow fast indexing into a list with minimal searching.

Speed of access to the elements of the linked list will be important since the list will be accessed at least once for each tick of the simulation clock. How can we make it more efficient? Let's consider some of the characteristics that might reduce efficiency and see how they can be improved.

If we need to insert a record somewhere into the middle of the list, we must find the right place to insert it. While the linked list is very efficient in terms of adding and deleting or even moving data without shuffling a lot of records, it is not especially efficient for searching. On average, if we want to insert a record at an arbitrary point, we would have to search through about half of the records in the list. In particular, if we knew that “t=23” is at a particular point in the list, how would we know where to insert “t=18”? We know it is before “t=23,” but we need to start from the beginning in a linked list, even if “t=23” were the last item in the list. A “doubly linked list” deals with this issue by adding a “previous” pointer to the “next” pointer of a linked list. This way it is easy to move in both ways through the list.

Some lists use a “middle pointer,” besides the required “head” and optional “tail” pointer to allow fast access to the center part of the list. But, if the additions and deletions are not balanced with respect to the “center,” this pointer will become useless unless it is updated.

Finally, if we need to jump to an arbitrary position in a large list, “hash tables” can be used. A hash table stores compressed data-dependent information. As an example, if we made a list of students in the class, rather than storing their entire first and last names, we could compress the names into a small number (say, 3 digit) by performing some arithmetic operation on the letters. For instance, we could add all the letters of the first and last name (represented as numbers between 1 and 26) together and then record the 3 least significant digits. The hash table would map this 3 digit number into the position in a list. To search for an arbitrary student, we would “hash” their name by applying the algorithm described and look for it in the hash table.

## Dynamic versus Static Memory-Management Tradeoffs for Simulation

- Static:
  - Allocate a fixed maximum amount of storage for simulation data
  - E.g., fixed array
- Dynamic
  - Request memory as needed for simulation data
  - E.g., malloc() or language that dynamically allocates from a “heap”

	Advantages	Disadvantages
Static memory allocation	<ul style="list-style-type: none"> <li>• Faster startup</li> <li>• More consistent simulation speed</li> </ul>	<ul style="list-style-type: none"> <li>• Predetermined maximum list sizes</li> <li>• May require extra time during simulation to reorder records</li> </ul>
Dynamic memory allocation	<ul style="list-style-type: none"> <li>• No fixed simulation size</li> <li>•</li> </ul>	<ul style="list-style-type: none"> <li>• Slower startups</li> <li>• Memory fragmentation may require slow “garbage collection”</li> </ul>

EE: CptE345: Modeling and Simulation  
Fall 2003

Copyright ©2003  
Stevens Institute of Technology  
All rights reserved

75

To wrap up this discussion of the design of a simulation, we must consider how we can trade the performance of the simulation against various implementations, particularly with regard to allocating memory for the simulation.

The simplest approach is to create arbitrary size arrays, for instance to store the largest FEL that we anticipate needing for the simulation. We would use a static memory allocation for the array and would use it as needed during the simulation. This technique has the advantage that the simulation will tend to start quickly and run at a consistent speed, since the memory allocation has been taken care of at the start. Unfortunately, it is often difficult to predict how large this arbitrary array will need to be. Consider this – before we start the simulation, we don’t really know how many customers will be in queue at any time during the simulation. If the queue grows too large, the FEL could grow beyond the limits of the fixed size array.

To counter this problem, we might use dynamic memory allocation. This has the advantage that there is no predetermined limit to the size of the simulation. However, the startup time will generally be slower as memory must be allocated to the simulation. During the simulation, as records are added and deleted, we might find that the simulation memory becomes fragmented – the same problem that occurs on the hard disk of a computer. To speed up the simulation, it might be necessary to defragment the memory used in the simulation. In some programming language environments, this is referred to as “garbage collection” where data elements that are no longer needed are discarded and active data is reordered to be in contiguous memory. Depending on the complexity of the data structure and the size of the memory used and available, this process can take a considerable amount of time.

## Homework 3

- Chapter 3, page 93, exercise 1.
- Continue your project started last week. Set objectives and define overall approach.