CSE 326
Lecture 19

Spring Quarter 1998

Discrete Event Simulation

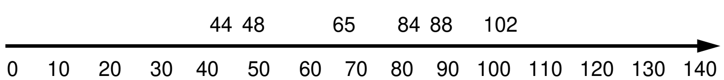
- Set of events E
- Each event e has an associated time t(e)
- Simulation loop

  Repeat

  - Find event e_0 with smallest associated time $t(e_0)$
  - Remove e_0
  - Process e_0
  - Insert any new events with associated times

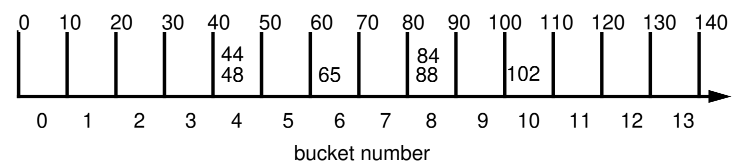
  until enough events simulated
- Priority Queue used to remove smallest and insert new events
- Static if exactly one new event is inserted per event simulated

Simulation Time



- e_0 is the current event which is supposed to occur at time $t(e_0)$
- When event e is generated its time t(e) is in the future, that is, $t(e) > t(e_0)$.
- Often the time $t(e) - t(e_0)$ can be modeled by a random variable of mean μ.

Infinite Bucket Calendar Queue

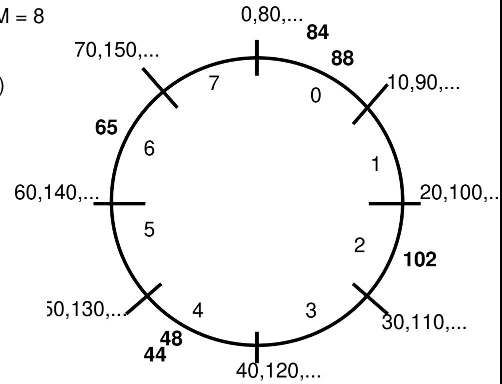


- Bucket width $\delta = 10$
- Number of Events N = 6
- Event e with time t(e) goes in bucket $t(e) / \delta$
- Calendar Queue Algorithm
  - if bucket empty go to next bucket
  - if bucket is not empty search for smallest in bucket to process

M Bucket Calendar Queue

- Number of Buckets M = 8
- Bucket Width δ = 10
- Event e with time t(e) goes in bucket

  t(e) / δ mod M



M Bucket Calendar Queue Data Structure

- Assume we have an array B[0..(M-1)] of pointers.
- A list entry is a node of the form

event node

	time: integer (possibly a real)
	event: pointer to what to do
	next: pointer to another event node

- The current time is indicated by the integer variable Now (simulation time)
- The parameter Delta is the bucket width which is an integer.

M Bucket Calendar Queue Insertion

- Insert(n) inserts the event node n into the calendar queue assuming that n.time > Now.
- Insert(n: event) {

```
k: local integer;
k := n.time/Delta mod M;
n.next := B[k];
B[k] := n }
```


M Bucket Calendar Queue Delete_min

- Delete_min returns the node n with minimum n.time assuming all nodes m in the calendar queue have m.time ≥ Now.
- Delete_min: event {

```
local integer k;
repeat
   k := Now/Delta mod M;
   n := min(B[k]);
   case
      n = null: Now := Now + Delta;
      n.time ≥ Now + Delta: Now := Now + Delta;
      n.time < Now + Delta: { Delete(n,B[k]), Return(n) }
   endcase
endrepeat }
```

What if Times are Reals?

- We should maintain an integer counter called INow that indicates the number of integral units of δ that have passed. This technique minimizes floating point error.
- Insertion code

  k := ⌊n.time/Delta⌋ mod M;
- Deletion code

```
repeat
   k := INow mod M;
   n := min(B[k]);
   case
       n = null: INow := INow + 1;
       n.time ≥ (INow + 1) * Delta: INow := INow + 1;
       n.time < (INow +  1) * Delta: { Delete(n,B[k]), Return(n) }
   endcase
endrepeat }
```


Evaluating the Calendar Queue

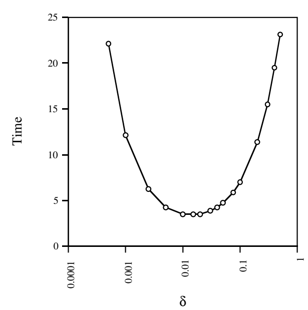
- We look at the static case where the number of events N is fixed.
- We will assume that the bucket width is δ and there are M buckets.
- We assume that $t(e) - t(e_0)$ is a random variable with mean μ.
- There are parameters b,c, and d such that
  - b is the cost per bucket visited
  - c is the cost per list entry visited
  - d is the cost per event processed.
- Cost per event is the important number.

$$\textit{Cost Per Event} = b \cdot j + c \cdot k + d$$

where j is the number of buckets visited and k is the number of list entries visited before finding the minimum.

Performance of the Infinite Bucket Calendar Queue

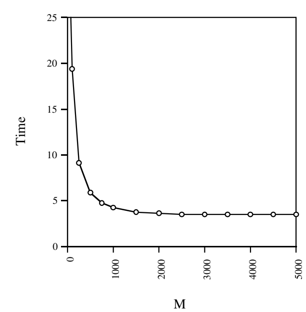
- Assume M is very large (infinite) then if δ is too small then too many buckets are visited and if δ is too large then too many list entries are visited in finding the minimum.



- b= c= d = μ = 1
- N = 100
- M is infinite

Performance of the M Bucket Calendar Queue

- Given a good δ for the infinite calendar Queue, if M is chosen too small then too many list entries are visited in finding the minimum, particularly list entries whose times are not within δ of the current time.



- b = c = d = μ = 1
- N = 1000
- δ is optimal for M infinite

Analysis of Calendar Queue

- Let $E(\delta)$ equal the expected time per event for the infinite bucket calendar queue. For $\delta = O(1/N)$

$$E(\delta) \cong \frac{cN}{2\mu} \cdot \delta + \frac{b\mu}{N} \cdot \frac{1}{\delta} + c + d$$

- Taking the derivative of E with respect to δ we get

$$\frac{d}{d\delta}E(\delta) \cong \frac{cN}{2\mu} - \frac{b\mu}{N} \cdot \frac{1}{\delta^2}$$

- Setting the derivative to zero we can solve for optimal δ.

$$\delta_{opt} \cong \sqrt{\frac{2b}{c}} \cdot \frac{\mu}{N}$$

Expected Time per Event

- Substituting into the original equations we obtain

$$E\left(\delta_{opt}\right) \cong c + \sqrt{2bc} + d$$

- Alpha DEC 3000/400 (133 MHz) with μ=1, N = 1,000, M = 10,000.

  b = .50 μs

  c = .64 μs

  d = 5.10 μs
- δ_{opt} = .00125 s which is close to μ/N
- Predicted by the theory: $E(\delta_{opt})$ = 6.54 μs per event
- Actual on Alpha: 6.28 μs per event (4% from predicted)

Selecting The Queue Size M

- There are two cases:

  1. t(e) - t(e_0) is bounded by β for some β.

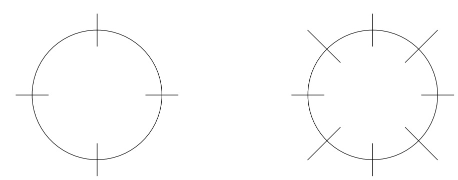
  2. t(e) - t(e_0) is unbounded.
- Bounded case. Choose M = β/δ + 1. This choice assures us the performance of M buckets is exactly the same as infinitely many buckets because there is no wrap-around.
- Unbounded case. Choose M = αN for a suitable α. For the exponential distribution α = 10 is more than satisfactory.

Dynamic Calendar Queue

- If N varies over time the calendar queue algorithm can be modified to keep good performance. We may need to change δ and M when performance is suffering.
- Two options.

  1. Stop everything. Move all events from one calendar queue to another.

  2. Keep going. Start up a new calendar queue which is empty. As events are traversed move events to the new calendar queue. In this case we have to check two calendar queues to do delete_min.



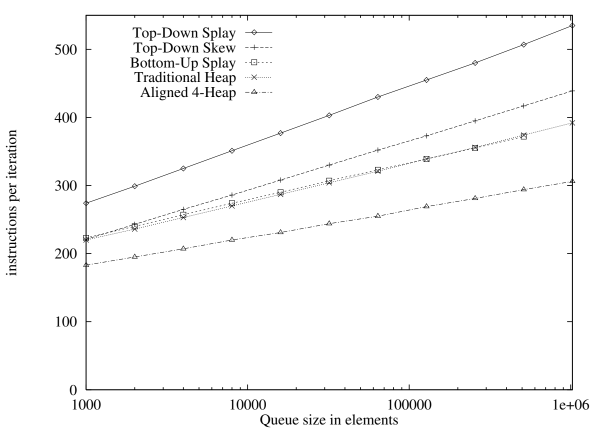
Selecting a Priority Queue

- Linked list: Best for very small priority queues. Sorted list might work best.
- Binary Heap: Simple and effective but does not support merge. Has worst case time O(log n) per operation.
- d-Heap: Slightly more complicated than binary heap and may be useful when time is critical. Choice of d=4 is probably best for cache memory systems.
- Leftist Trees: Simplest priority queue that supports merge and has worst case time O(log n) per operation.
- Skew Trees: Simplest priority queue that supports merge and has amortized time O(log n) per operation.
- Binomial queue: Special priority queue with O(1) insertion time, but O(log n) delete_min and merge time.
- Calendar Queue: In some cases, such as discrete event simulation, when there is knowledge of how the events are "jumping" in time, the calendar queue can achieve O(1) expected time per event. The calendar queue can be optimized for best performance.

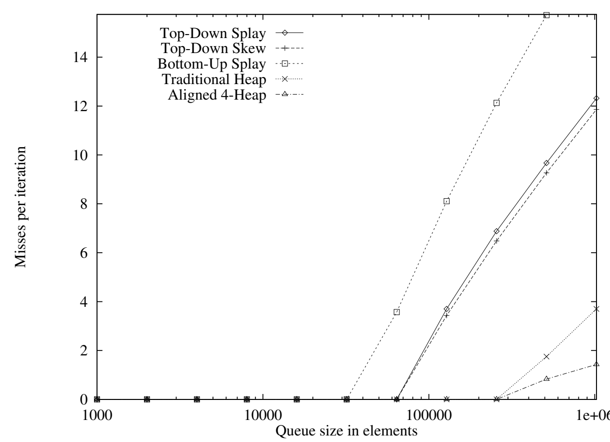
Comparison of Priority Queues

- Comparison of
  - Top down splay tree
  - bottom up splay tree
  - top down skew heap
  - traditional binary heap
  - 4-heap
- Event queue of a discrete event simulator
- DEC alpha 250 with 2 MB level 2 cache
- 8 byte keys, 4 keys per cache block for the 4-heap

Priority Queue Comparison Instruction Count



Priority Queue Comparison Cache Misses



Priority Queue Comparison
Execution Time