Future Event List

Priority Queue Data Structures

Richard M. Fujimoto
Professor

Computational Science and Engineering Division
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0765, USA

http://www.cc.gatech.edu/~fujimoto/

Priority Queue: Operations

- Required Operations
  - Insert (FEL, ev, ts); /* aka enqueue */
    - Add event ev with timestamp ts to event list FEL
  - Event = Delete (FEL); /* aka dequeue */
    - Remove smallest time stamp event and return a pointer to it
  - DeleteArbitrary (FEL, ev);
    - Unschedule an event
    - Delete an arbitrary event ev (not necessarily one with smallest timestamp)
- Constraints for a general simulation engine
  - Maximum number of events required at one time unknown
    - Memory allocation issues
  - Amount of computation per event might be small
    - event list computation time may dominate overall execution time
  - Sequence of Insert and Delete operations unknown
  - Distribution of timestamps on successive Insert operations unknown

Data Structures

- Linked lists
  - Single linked list
  - Double linked list
- Tree Based
  - Heap
  - Splay tree, …
- Hybrid
  - Henricksen's algorithm
- Hashing schemes
  - Calendar queue
  - Ladder queue

Performance Comparison

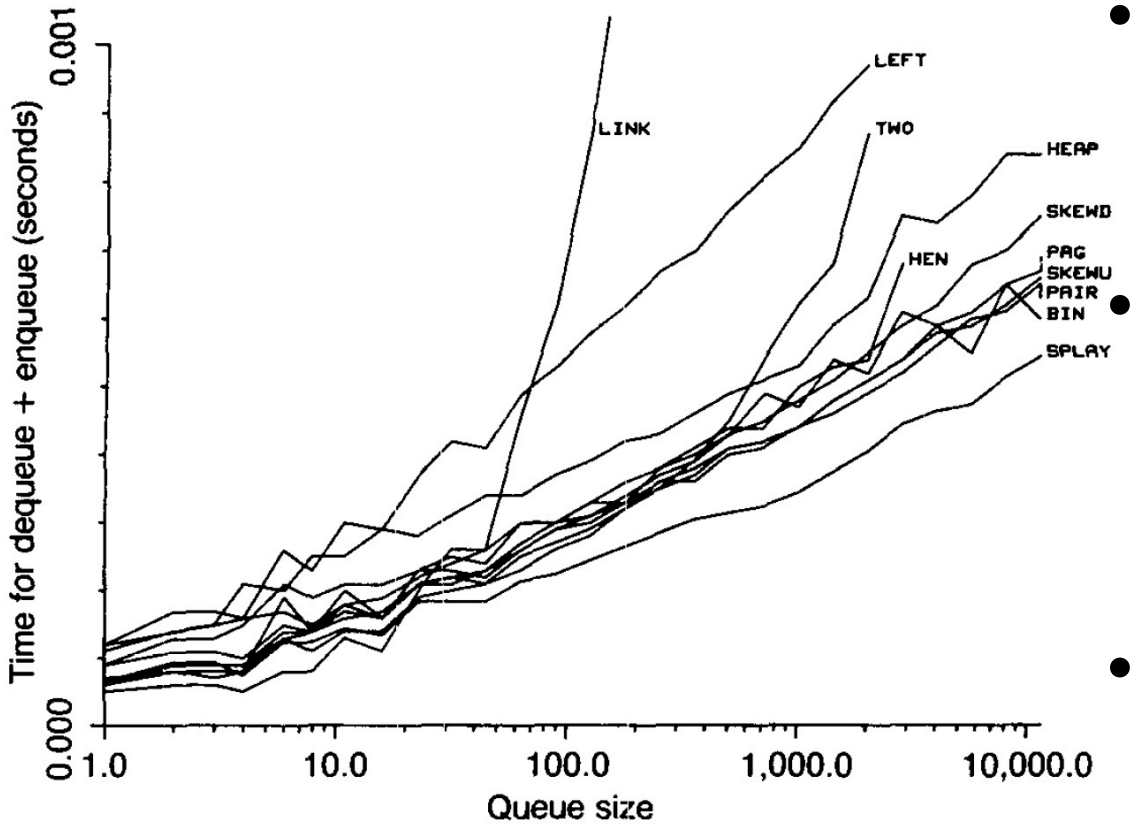


FIGURE 12. VAX 11/780 Data for the Exponential Distribution

- Hold model: fixed number of elements; repeat (dequeue, enqueue) operations
- Comparison of different data structures using exponential timestamp increment (see paper for other distributions)
- Empirical measurement of time for (dequeue +enqueue) operation for different queue sizes

D. Jones, "An Empirical Comparison of Priority-Queue and Event Set Implementations", Communications of the ACM, 29, 4 (April 1986), 300-311.

Conclusions [Jones]

TABLE II. Summary of Conclusions

Priority-queue implementation	Code size[a]	Performance		Relative speed[b]	Comments
		Average	Worst		
Linked list	47	$O(n)$	$O(n)$	11	Best for $n < 10$
Implicit heap	72	$O(\log n)$	$O(\log n)$	8	
Leftist tree	79	$O(\log n)$	$O(\log n)$	9–10	
Two list	104	$O(n^{0.5})$	$O(n)$	9–10	Good for $n < 200$
Henriksen's	68	$O(n^{0.5})$	$O(n^{0.5})$[c]	1–7	Stable
Binomial queue	188	$O(\log n)$	$O(\log n)$	1–7	
Pagoda	110	$O(\log n)$	$O(n)$	4–8	Delete in $O(\log n)$
Skew heap, top down	56	$O(\log n)$	$O(\log n)$[c]	5–7	
Skew heap, bottom up	103	$O(\log n)$	$O(\log n)$[c]	4–6	Delete in $O(\log n)$
Splay tree	119	$O(\log n)$	$O(\log n)$[c]	1–3	Stable
Pairing heap	84	$O(\log n)$	$O(\log n)$[c]	3–6	Promote in $O(1)$

[a] The total lines of Pascal code for initqueue, emptyqueue, enqueue, and dequeue.
[b] 1 is fastest; 11 is slowest: The rankings are based on Figures 12–14.
[c] An amortized bound; single operations may take $O(n)$ time!

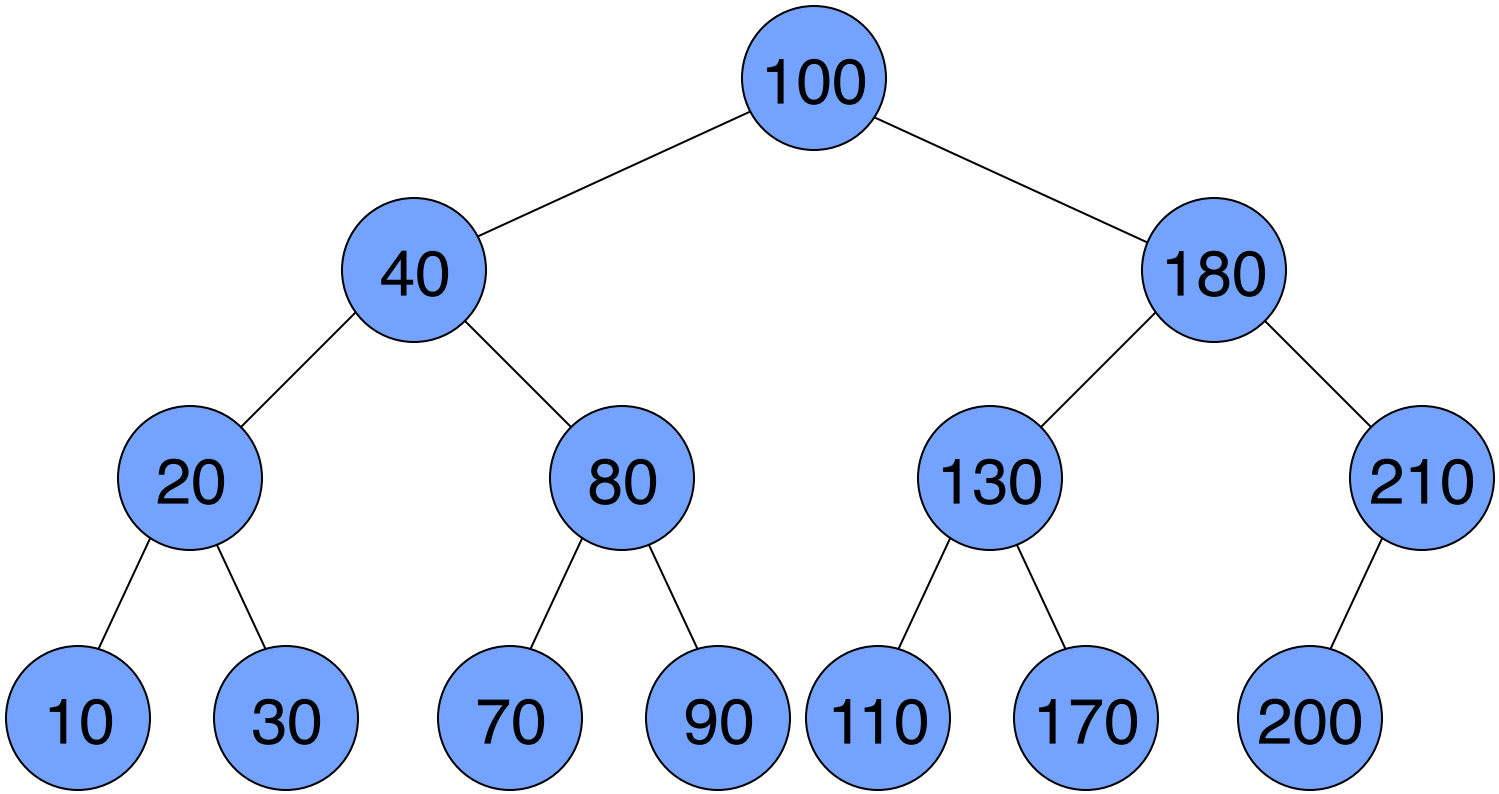
- Linear list fastest n<10, but very bad if n>50
- Splay trees
  - Stable: items with same priority treated in FIFO order
  - Supports delete arbitrary operation
  - Reasonably good performance across different distributions
- Results tend to be architecture dependent (caches)

Splay Trees

D. Sleator, R. Tarjan, "Self-Adjusting Binary Search Trees", Journal of the ACM, 32, 3 (July 1985), 652-686.

- Binary search tree; for any node (timestamp t)
  - Left subtree nodes have timestamp < t
  - Right subtree nodes have timestamp ≥ t
  - Contrast with the "heap property"
- Operations
  - Delete: where is smallest item?
  - Insert: how does insertion work?

Binary Search Tree Operations



- Delete: remove leftmost item (leaf, or has one child)
- Insert: traverse tree downward, insert at proper location
- Repeated insert/delete operations unbalance tree
- Performance $O(n^{0.5})$ [Kingston, 1985]

Splay Trees (cont.)

- Repeated insert/delete operations (e.g., repeated deletes) leaves unbalanced tree
- Could carefully rebalance after each operation (AVL tree)
- Splay tree
  - Heuristic to blindly restructure tree using rotation operations
  - Does not guarantee tree will be rebalanced; worst case time per operation is O(n), but…
  - *Amortized* complexity (average over many operations) $O(\log_2 n)$

Splaying Operation

- Splay(x): travel up tree from x to root, moving x to root
- Insert(x): insert x at leaf, splay(x)
- Delete: remove node, splay at parent of removed node
- Apply splaying operations in traversal up tree

  Visit x; p(x) = parent of x; three cases:

  (a) p(x) root

  (b) x and p(x) both left or both right children

  (c) x right child and p(x) left child or vice versa

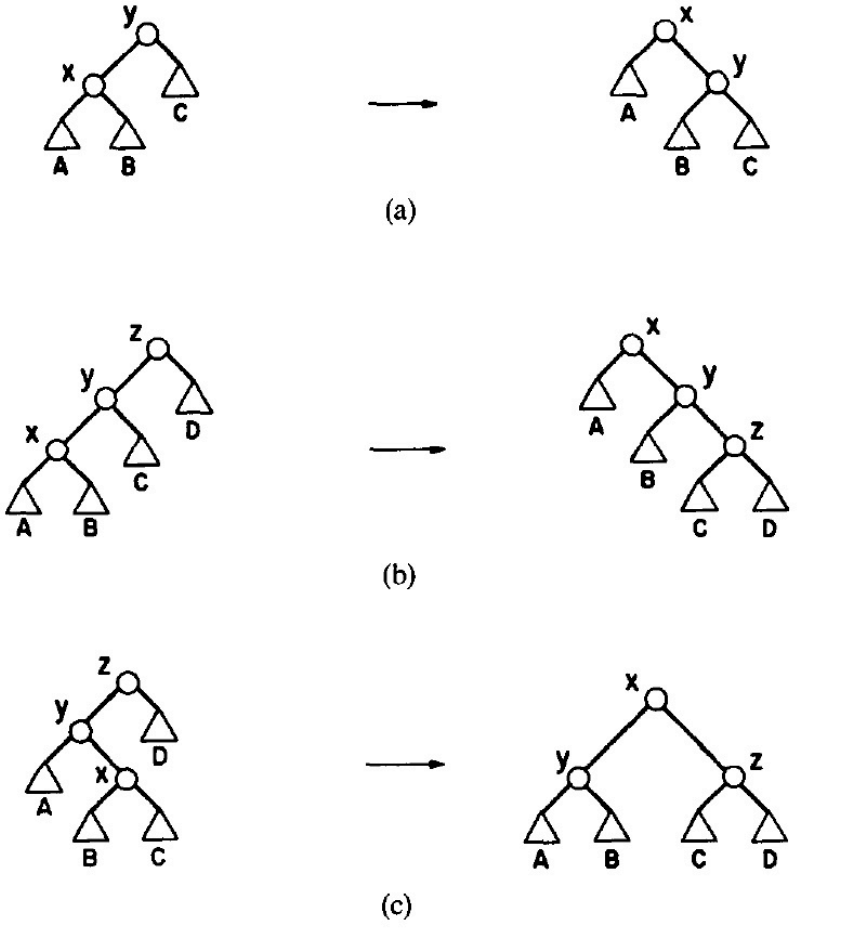


FIG. 3. A splaying step. The node accessed is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig–zig: two single rotations. (c) Zig–zag: double rotation.

Calendar Queues

R. Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," Communications of the ACM, 31, 10 (October 1988), 1220-1227.

- Analogous to desk calendar
  - Array of "buckets" denoting a time interval (e.g., 365 buckets, 1 day/bucket, spanning a year)
  - Each bucket covers a fixed interval of time (day)
  - Each bucket points to sorted linear list of events scheduled for that day
  - Events scheduled for next year simply placed in corresponding bucket (day)
  - Keep pointer to bucket with smallest timestamp event (today)

Calendar Queue Example

	Bucket 0:	16.2			/* 12–12.5 */
	Bucket 1:	16.6			/* 12.5–13 */
	Bucket 2:				/* 13–13.5 */
	Bucket 3:	17.8			/* 13.5–14 */
	Bucket 4:				/* 14–14.5 */
→	Bucket 5:	14.5	14.7	14.8	/* 14.5–15 */
	Bucket 6:	15.2	15.3	19.1	/* 15–15.5 */
	Bucket 7:	15.9			/* 15.5–16 */

The current year begins at 12.0 and ends at 16.0. Bucket 5 is the current date. Note that events scheduled in buckets 0 through 4 are scheduled for next year and are in the range 16–20. 0.5 is the length of a day. The numbers on the right are the time ranges for each day in the current year. Note that the event at 19.1 in bucket 6 is scheduled for next year.

FIGURE 1. Eight Day Calendar Queue

Calendar Queue Operations

(preliminary version)

- Insert
  - Map timestamp to appropriate bucket
  - Perform linear list insertion into list
- Delete
  - Starting from current (today) bucket
    - If first event in current year, remove it
    - Else, go to next bucket; if last bucket, advance to next year

- Potential problems
  - Long linear list
  - Need to scan many buckets on delete operation

Problems and Solutions

- Wrong number of buckets
  - If #events >> #buckets
    - Long linear list, long insert time
  - If #events << #buckets
    - Long delete time (scan many empty buckets)
  - Solution: resize number of buckets
    - If #events > 2 * # buckets, double # buckets
    - If #events < 0.5 * #buckets, halve # buckets
    - Resizing requires copying events; expensive
- Poor choice of length for a "day"
  - Ideally, about one event per day
  - If day too long, many events in "today" bucket (long insert time)
  - If day too short, many events in later years (long delete time)
  - Solution: Sample some events to estimate average time between events and set day length to that value; perform at each resize operation

Problems and Solutions (cont.)

- Bimodal distribution of events (many "near future" events and many "far future" events)
  - After removing "near future" events, must cycle through all buckets repeatedly to reach far future events
  - Solution: If scan through all buckets w/o success, search for smallest timestamp event by scanning all buckets

Performance Measurements

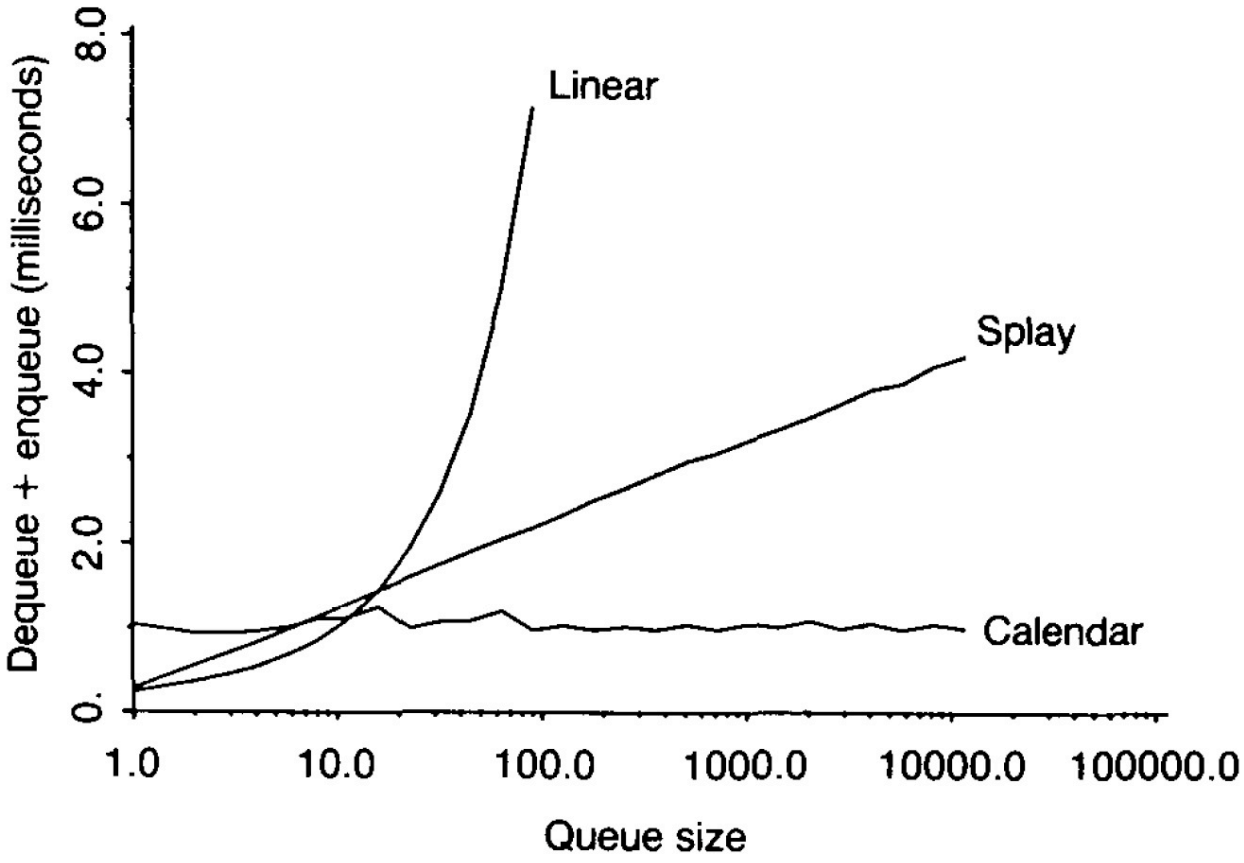


FIGURE 3. Comparison of Hold Times on Texas Instruments PC

- Hold model, exponential distribution

Calendar Queues

- "Well behaved" (i.e., hold like) behavior results in excellent performance for test cases
  - Often significantly faster the O (log n) data structures
- In practice, extremely poor performance sometimes observed, due to excessive resizing operations or many items mapping to the same bucket (e.g., see [Rönngren, Ayani, 1997])
- More recent refinements (e.g., Ladder Queue) offer variations on original calendar queue idea

Calendar Queue Critique

- Calendar queue performs well when number of elements in the queue (n) and mean timestamp increment (μ) constant [Ronngren and Ayani 1997]
- May yield O(n) performance if μ varies, even if n remains constant
  - Many events in small number of buckets (long insertion time)
  - Many empty buckets (long deletion time)
  - Attributed to size-based trigger to resize queue
- Large changes in n can cause poor performance
  - May trigger resize operation when not necessary if size fluctuates by factor of two
  - Resize operation expensive

Calendar Queue Critique (cont.)

- Sampling heuristic for CQ parameters (number of buckets and bucket width) ineffective for skewed event distributions
  - CQ heuristic to set bucket width to average time between events fails for skewed distributions
  - High variance in inter-event time leads to inappropriate settings (e.g., small bucket size leading to many empty buckets that must be skipped on deletion operation)
- Sorting events on enqueue operation
  - Makes enqueue operation expensive for long sublists
  - Sorting effort wasted if resize operation is triggered

Ladder Queue Data Structure

- Top
  - Unsorted list of “far future” events
  - Maintain: MaxTS (largest timestamp), MinTS (smallest timestamp) and NTop (number of events) in list
  - TopStart: all events in Top have timestamp at least this large
- Middle
  - Multi-level form of calendar queue
- Bottom
  - Sorted list of “near future” events
  - Nbot: Number of events in list

W. T. Tang, R.S.M. Goh, L-J. Thng, “Ladder Queue: An O(1) Priority Queue Structure for Large-Scale Discrete Event Simulation,” ACM Transactions on Modeling and Computer Simulation, 15, 3 (July 2005), 175-204.

Ladder Queue

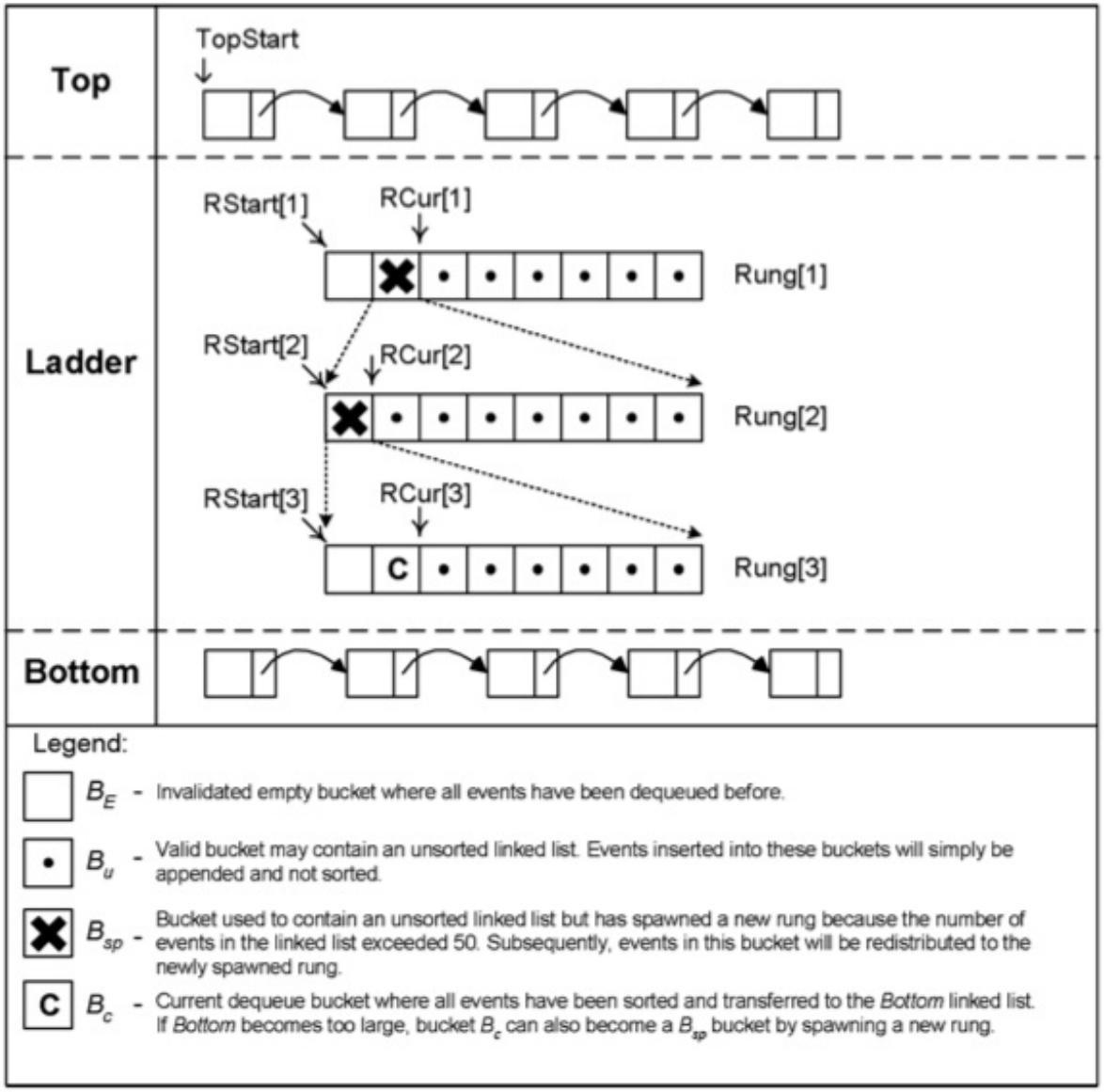


Fig. 1. Basic structure of Ladder Queue.

- Top
  - Unsorted list of "far future" events
- Ladder (middle)
  - Several "rungs" of buckets (rung like a CQ)
  - Each bucket holds unsorted events
  - If a rung has a bucket with too many events (event timestamps not uniformly distributed over bucket), "spawn" new rung
- Bottom
  - Sorted list of "near future" events
  - Dequeue events here

Ladder (middle tier)

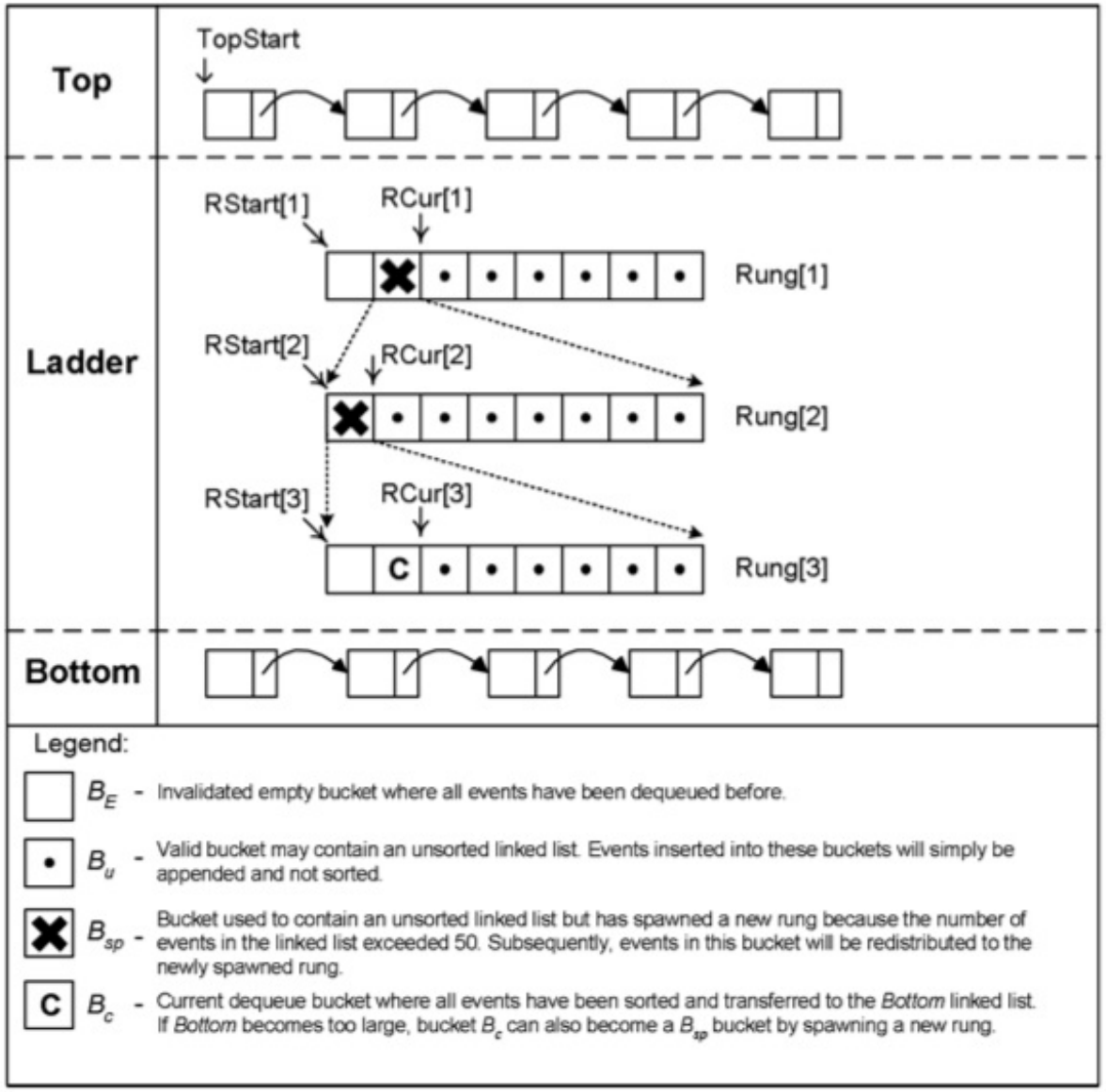


Fig. 1. Basic structure of Ladder Queue.

Spawning

- If a bucket has "too many" events, form a new calendar queue (rung) containing events within that bucket
- Repeat the above step for the new calendar queue (rung)
- Process of creating a new rung called "spawning"

Ladder: Main Ideas

- Each rung of ladder has a different bucket size, accommodating different timestamp distributions
- Delay sorting events until small timestamped events about to be dequeued
  - Insertion to Ladder and Top simply append to list; only Bottom list is sorted
  - Avoids sorting list, then discarding work due to resize operation
- Resize operation from an insertion only occurs in Bottom
  - Size of Bottom limited to threshold, limiting cost of resize, independent of number of events in the queue
- Theoretical complexity is O(1), amortized over multiple operations

LQ Operation: Epoch

- Initially, all events reside in the Top
- Epoch starts with first Dequeue operation
- Move events from top to form a single rung of ladder (middle section)
- If first non-empty list of events in ladder has fewer events than threshold
  - Then: sort them and move to Bottom
  - Else: spawn; move list of events into a new rung (form a new CQ with its own bucket size parameters based on events in the new CQ), and repeat this step
- Dequeue first event from Bottom
- Subsequent enqueue/dequeue operations eventually result in all events in Top

LQ: Spawn Example

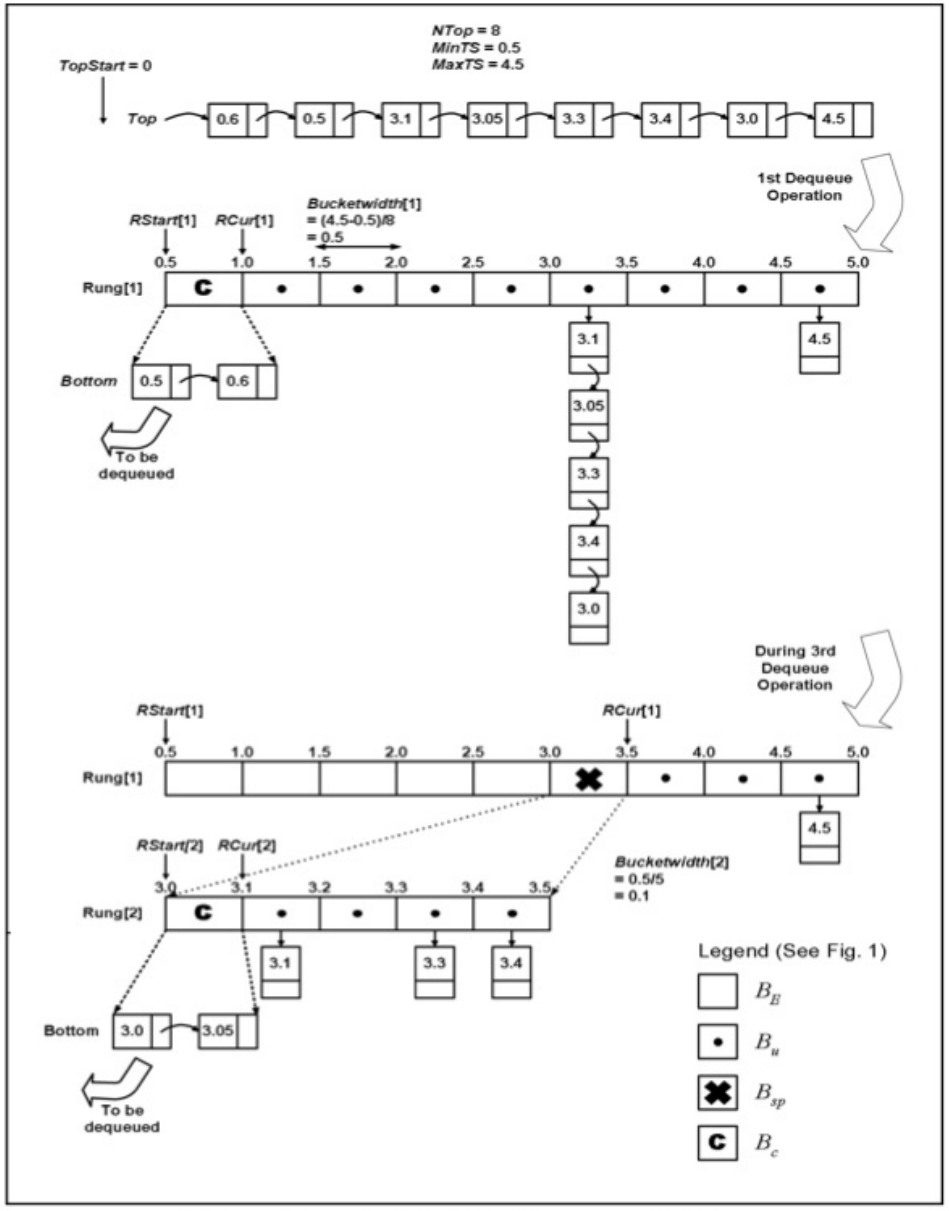


Fig. 2. An example illustrating the dequeue algorithm.

- Initially, all events in Top
- Dequeue:
  - Move Top events to ladder
  - Move first two events in ladder to Bottom
  - Return first event from Bottom (ts=0.5)
- Dequeue:
  - Return event from Bottom (ts=0.6)
- Dequeue:
  - List too long (length exceeds threshold)
  - Triggers spawn operation
  - Move first two events to Bottom
  - Return first event (ts=3.0)

Enqueue Operation

- Determine which part (Bottom, ladder, Top) will receive event based on the event timestamp
  - Top: Unsorted insertion
  - Ladder: Find rung, then insert (like CalQ insert)
  - Bottom: Sorted insertion; if Bottom contains too many events (= threshold)
    - Spawn a new rung in Ladder
    - Populate new rung with events from Bottom
    - First list in new rung becomes the new Bottom

Summary

- Priority queue performance can be important for some very large simulations (hundreds of thousands to millions of events in FEL)
- Variety of implementations exist
  - Linear list generally fastest for small event lists
  - O (log n) data structures (e.g., heap, splay tree) give predictable performance, though often not the fastest available approach
  - Calendar Queue data structures offer best performance (constant time insert/delete) for many applications, but can yield surprisingly poor performance in some cases
  - Ladder Queue addresses unstable property of calendar queue