Indian Institute of Technology Hyderabad

Computer Science & Engineering Department CS5300

Parallel & Concurrent Programming

Comparative Study of Different Implementations of Lock Free FIFO Queues

Shagun (CS23MTECH14013)
Aishwarya Dash (CS23MTECH14001)
Akkinepally Phani Sahasra (CS23MTECH14008)

December 03, 2023

Circular Queue

Design of Circular Queue Class

We introduce a new approach of building circular queues that are lock-free, and support multiple producers and consumers using indirection and two queues. The design is ABA safe and live-lock free because we compare cycles and update indices accordingly.

Queue Data Structure :

Indirection :

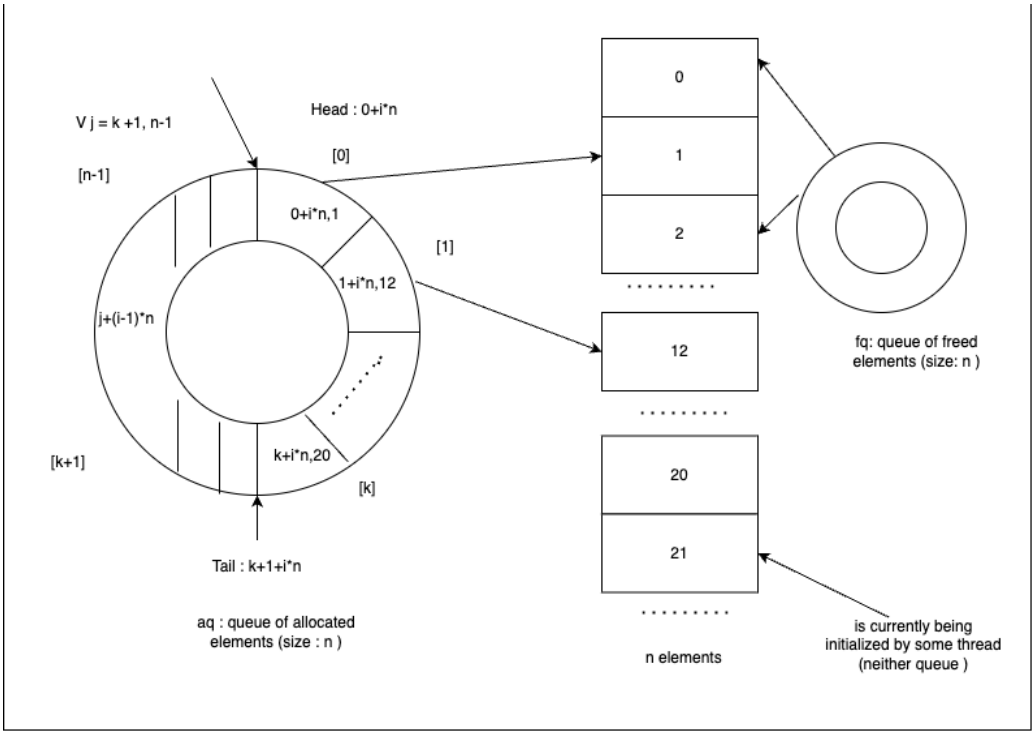
The queue doesn't store the data entries directly; instead, a queue entry records an index within the array of data. To simplify, we assume the data consists of integers.

Unallocated Entries Queue (fq) :

This queue is denoted by fq it keeps the indices to unallocated entries of the data array.

Allocated Entries Queue (aq) :

This queue keeps allocated indices which are to be consumed.



The entire data structure has the following properties :

- Both the queues have Head and Tail pointers and tail is incremented when new entries are enqueued, Head is incremented when new entry is dequeued.
- An entries array which will store the index and cycle of each entry in the queue.

- The total increments can be represented as i n + j where n is the capacity of the queue. Here i is called the cycle number and j is called index which denotes position in a circular array.
- Each queue has an n sized array called entries which will store the cycle and index of the elements in queue
- aq is initialized to be an empty queue. All entries are initialized to cycle 0 and Head = n , Tail = n
- fq is initialized to be a full queue. It will contain all the n entries. (We initialize it as empty queue and enqueue all the n indices)
- It has 2 methods enqueue() and dequeue().

Producer

A producer thread will dequeue an index from fq . If NULL / -1 is returned the queue is full o/w will return a valid index in which case we write data to the corresponding index and insert the index into aq.

Consumer

A consumer thread will dequeue index from aq . If NULL/-1 is returned the queue is empty o/w we get a valid index in which case we read the data from data array and insert the entry back into fq.

The algorithm below summarizes how different pointers fq and aq interact along with our actual data array to implement the circular queue.

```
// data:  an array of pointers
// aq is initialized empty
// fq is initialized full
1  bool enqueue_ptr(void * ptr)
2      int index = fq.dequeue();
3      if ( index = ∅ ) return False;        // Full
4      data[index] = ptr;
5      aq.enqueue(index);
6      return True;                          // Success
7  void * dequeue_ptr()
8      int index = aq.dequeue();
9      if ( index = ∅ ) return nullptr;      // Empty
10     ptr = data[index];
11     fq.enqueue(index);
12     return ptr;                           // Success
```

Enqueue Algorithm :

```
2  forall entry_t Ent ∈ Entries[n] do
3      Ent = { .Cycle: 0, .Index: 0 };
4  void enqueue(int index)
5      do
6          T = Load(&Tail);
7          j = Cache_Remap(T mod n);
8          Ent = Load(&Entries[j]);
9          if ( Cycle(Ent) = Cycle(T) )
10             CAS(&Tail, T, T + 1);     // Help to
11             goto 6;                   // move tail
12         if ( Cycle(Ent) + 1 ≠ Cycle(T) )
13             goto 6;        // T is already stale
14         New = { Cycle(T), index };
15     while !CAS(&Entries[j], Ent, New);
16     CAS(&Tail, T, T+1);      // Try to move tail
```

- Enqueue does not need to check if a queue is full. It is called only when an available entry out of n exists. (Only if entry can be dequeued from fq)
- New entry (Tail / N , value) is enqueued at index = (tail % N).

- **Claim :** Tail must be one cycle in front of the current entry.
- **Proof :** Tail is initialized to n and entry's cycles are initialized to 0. So the property is satisfied at the start (first round of enqueues).

- Now let's look at the second round of enqueues. We know that After every successful enqueue operation the tail is incremented by 1. So when we again come back at this point (2nd round) tail would have been incremented n times =) it would be 1 cycle ahead of the entry.

- If Tail's cycle number is same as entry's cycle number =) Some other thread has enqueued but has not advanced tail. So we increase tail for global progress. If both the conditions are satisfied we are reading an old value of tail (n enqueues happened).

- If the conditions are satisfied we do a Compare & Swap operation to update the entry at this index with a new entry (Cycle(Tail) , value). If the conditions are satisfied we do a Compare & Swap operation to update the entry at this index with a new entry (Cycle(Tail) , value).

Dequeue Algorithm:

```
17 int dequeue()
18     do
19         H = Load(&Head);
20         j = Cache_Remap(H mod n);
21         Ent = Load(&Entries[j]);
22         if ( Cycle(Ent) ≠ Cycle(H) )
23             if ( Cycle(Ent) + 1 = Cycle(H) )
24                 return ∅;        // Empty queue
25             goto 19;        // H is already stale
26     while !CAS(&Head, H, H+1);
27     return Index(Ent);
```

Dequeue is possible only if queue is not empty

Claim 1: Cycle number of current entry and Head should match.

Claim 2: The values of Hb and Tb before an item b is enqueued are the same. Using this above claim also holds.

The proofs for above claims are given by induction.

- If Head is one cycle ahead, then queue is empty in this case, so we return NULL
- In any other case we are reading an old value. So we again fetch the new value of Head

The Algorithm is Lock-Free

Proof :

The Circular Queue algorithm has 2 unbounded loops. First one in the enqueue() method and other in the dequeue method.

Case 1

- If CAS operation fails in the enqueue it again repeats the loops, =) the entry Entries[j] was updated by another thread who called the enqueue method. Dequeue method didn't modify the entries array.

=) That other thread was making progress and it succeeded in its enqueue operation.

Case 2

- If CAS operation fails in the dequeue it again repeats the loops, =) the Head pointer was updated by another thread who called dequeue method . Enqueue method didn't modify the head pointer.

=) That other thread was making progress and it succeeded in its dequeue operation.

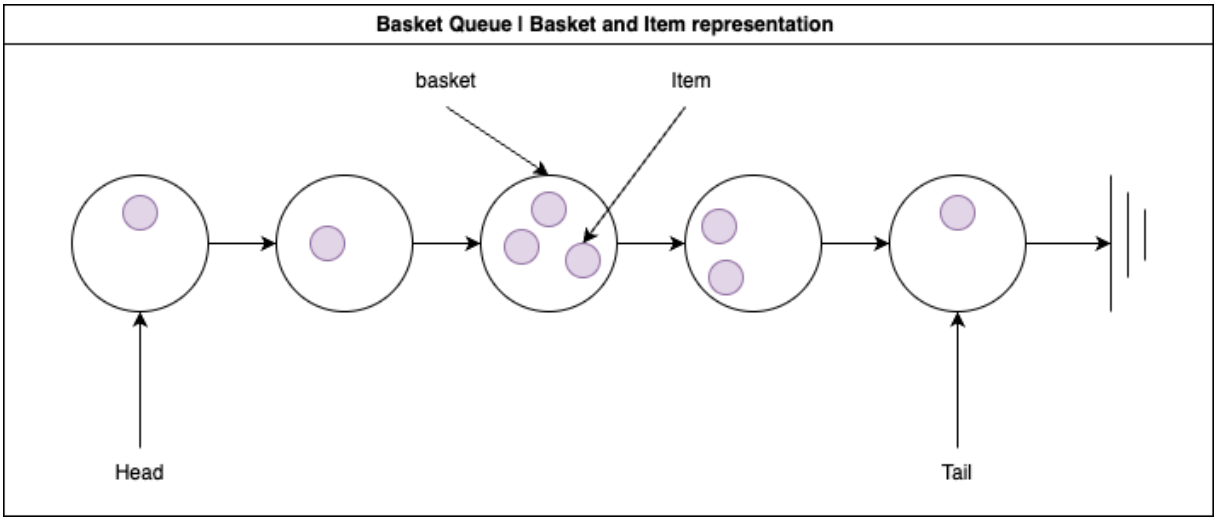
Basket Queue

Design of Queue

The Basket Queue represents a lock-free, linearizable, highly concurrent FIFO queue that diverges from the conventional fully ordered list by maintaining groups of items with mixed order. This queue facilitates the parallel execution of queue operations across various bins, enabling different queue operations to be concurrently performed in separate bins. Notably, the order of nodes within bins is not specified during the queuing process.

Linearizability allows us to change the order of concurrent operations. This motivates the idea of making the basket queue linearizable. Overlapping queue operations can be queued as a group (basket) without specifying the order of queued nodes. Nodes of the same bin can thus be removed from the queue in any order. Thus, the order of nodes discarded from the bin can be assigned as the order in which the nodes were queued in that bin.

It enables parallelism between different baskets by allowing threads to insert nodes into different baskets at the same time.



Basket queue's principles

- Time during which all nodes' enqueue operations overlap is time interval of that basket. This time interval is nothing but time when first enqueue operation among a group of concurrent enqueue operations is successfully completed. (Among threads executing CAS concurrently, first thread (winner) to perform CAS successfully)
- Order of baskets is decided by their respective time interval.
- Dequeue operations of nodes of a basket happen after the time interval where all enqueue operations of that basket have been overlapped.
- Dequeue operations of nodes of previous basket precede dequeue operations of nodes of next basket

Basket queue's properties defining FIFO order

- There is no need to specify nodes' order of a basket. (Order of nodes in which concurrent enqueue operations happen for a basket is assigned to be same as the order of nodes in which dequeue happens.)
- FIFO order is the nodes' order in different baskets.

Queue Initialization

```
struct pointer_t {
  <ptr, deleted, tag>: <node_t *, boolean, unsigned integer>
};

struct node_t {                                  struct queue_t {
  data_type value;                                 pointer_t tail
  pointer_t next;                                  pointer_t head
};                                               };

void init_queue(queue_t* q)
I01: node_t* nd = new_node()       # Allocate a new node
I02: nd->next = <null, 0, 0>       # next points to null with tag 0
I03: q->tail = <nd, 0, 0>;         # tail points to nd with tag 0
I04: q->head = <nd, 0, 0>;         # head points to nd with tag 0
```

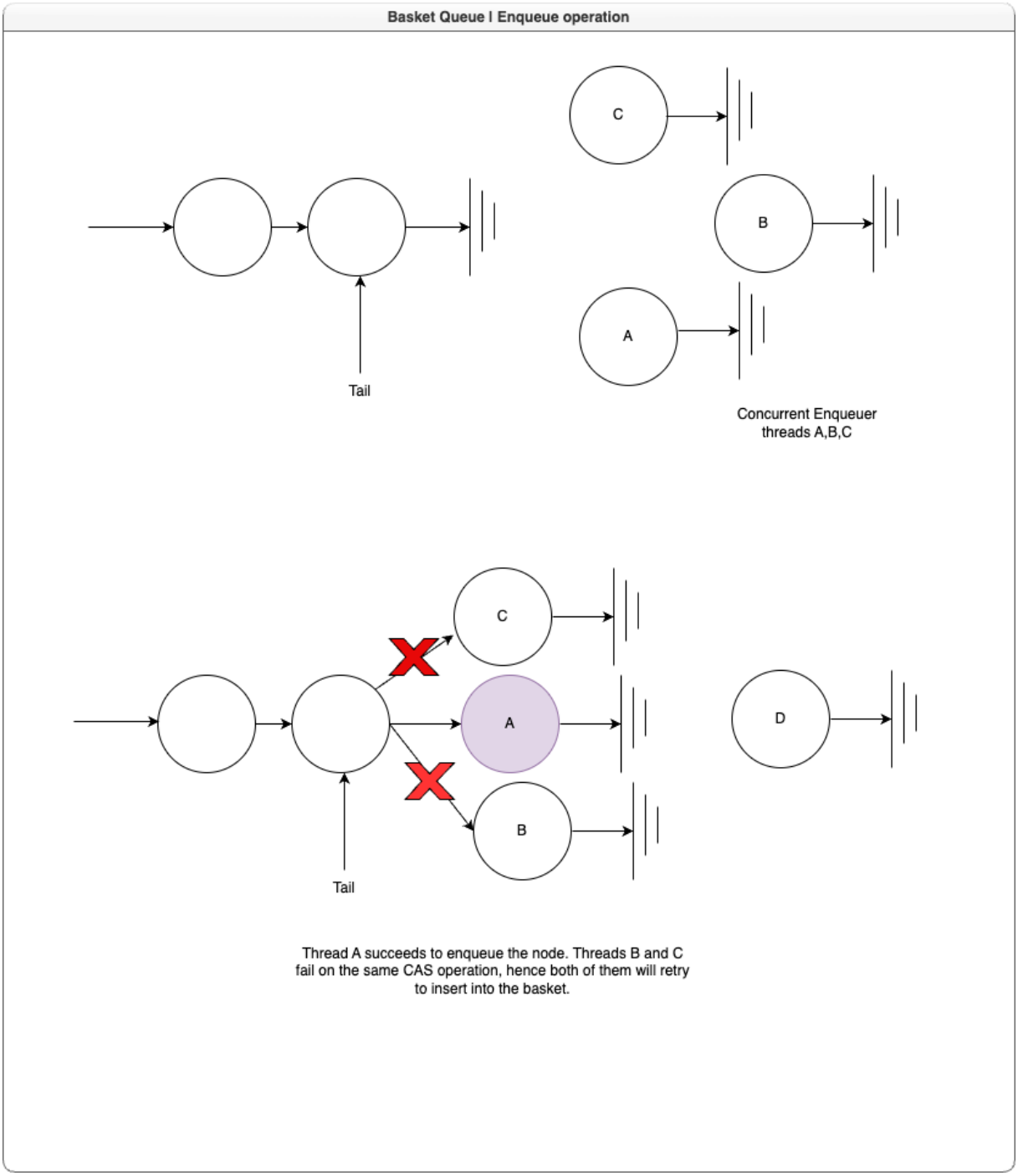
The basket queue is implemented as a clearly defined linked list of nodes, featuring head and tail pointers. The queue's head consistently points to dummy nodes, which may be succeeded by a series of marked (logically deleted) nodes. The queue's tail points to either the last basket of a node or the second-to-last basket. Two structures are introduced: pointer_t and node_t.

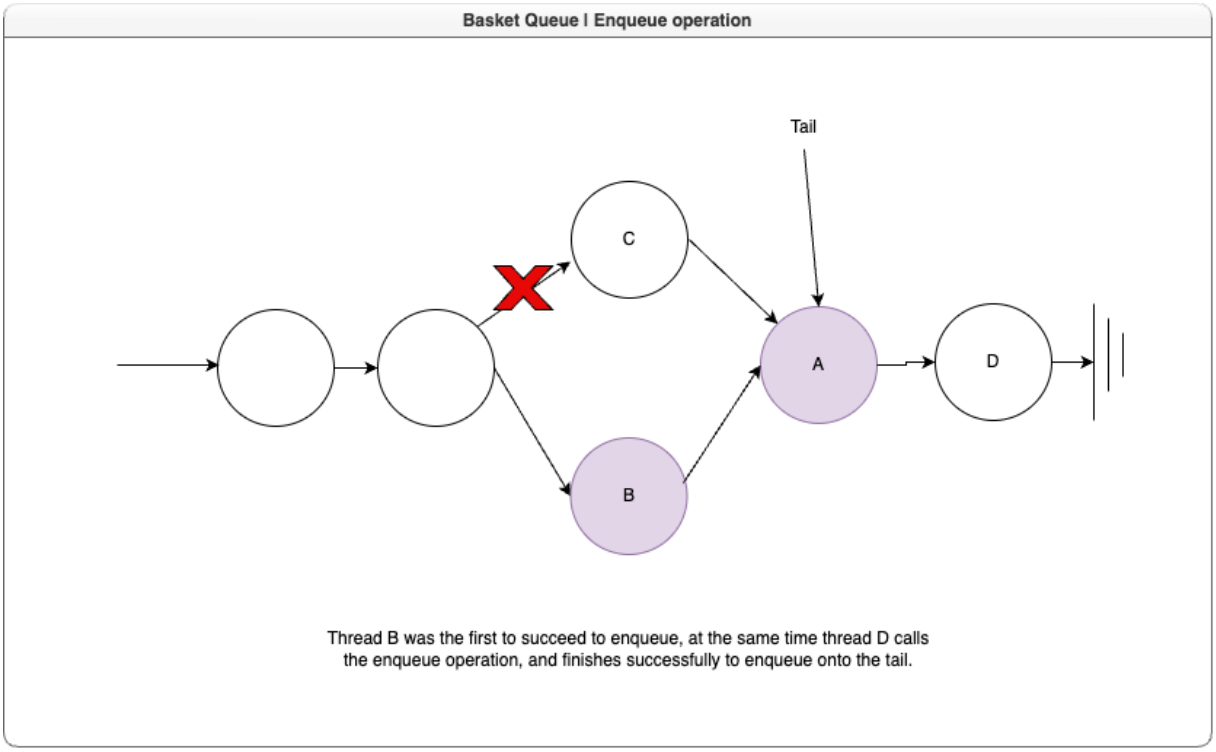
pointer_t has member fields ptr which stores the node_t pointer, boolean deleted which denotes whether node is logically deleted or not, unsigned integer tag which is used to solve ABA problem.
node_t has member fields value which stores the value stored in the node, atomic pointer_t next which stores the pointer of its next node.
Queue's head and tail are declared as atomic pointer_t and initialized both pointing to dummy node with deleted field 0 and tag field 0.

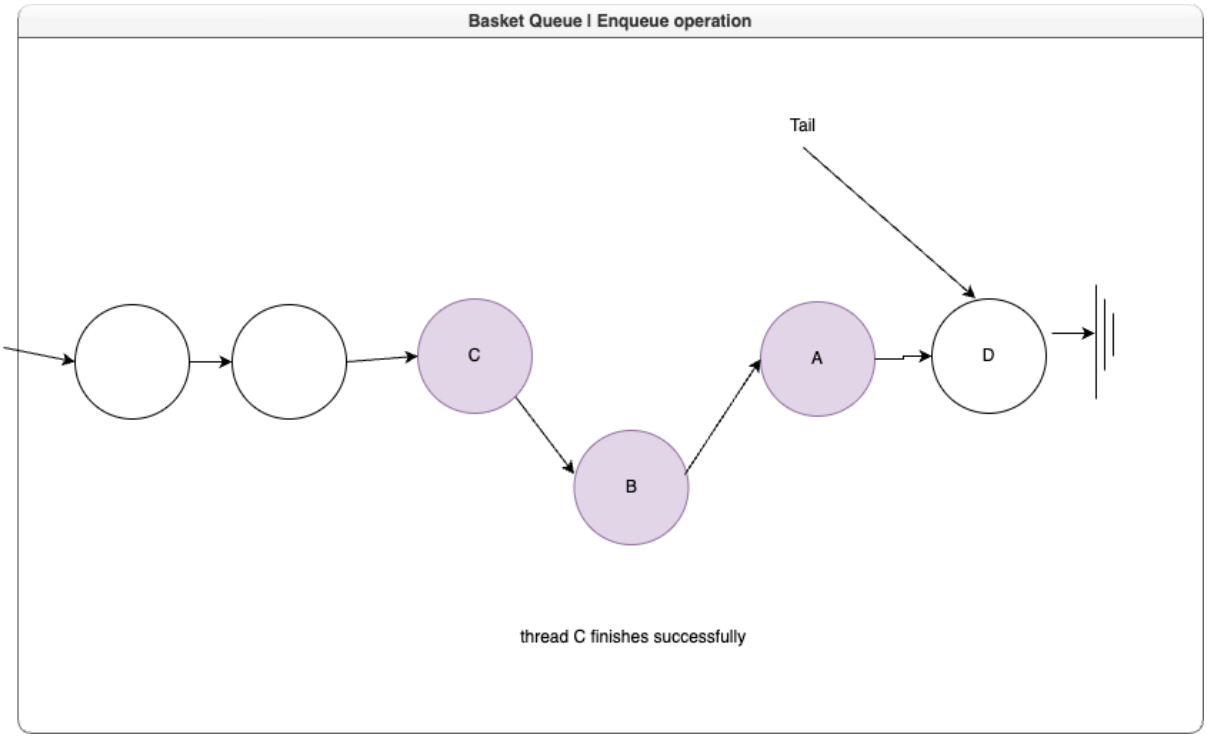
Enqueue algorithm

.

```
E01: nd = new_node()
E02: nd->value = val
E03: repeat:
E04:    tail = Q->tail
E05:    next = tail.ptr->next
E06:    if (tail == Q->tail)):
E07:       if (next.ptr == NULL):
E08:          nd->next = <NULL, 0, tail.tag+2>
E09:          if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E10:             CAS(&Q->tail, tail, <nd, 0, tail.tag+1>
E11:             return True
E12:          next = tail.ptr->next
E13:          while((next.tag==tail.tag+1) and (not next.deleted)):
E14:             backoff_scheme()
E15:             nd->next = next
E16:             if CAS(&tail.ptr->next, next, <nd, 0, tail.tag+1>):
E17:                return True
E18:             next = tail.ptr->next;
E19:       else:
E20:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
E21:             next = next.ptr->next;
E22:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1>)
```


Basket Queue I Enqueue operation
C
B
A
Tail
Concurrent Enqueuer
threads A,B,C
C
A
B
D
Tail
Thread A succeeds to enqueue the node. Threads B and C
fail on the same CAS operation, hence both of them will retry
to insert into the basket.





(E01-02) It is the initialization of a new node with a given value. (E04-05) First, enqueuer thread obtains a tail which is thread local from queue's tail pointer. next which is also thread local denotes the pointer of the tail's next node. (E06) It checks if the tail pointer and queue's tail pointer are the same or not. If they are not same, it means in the time interval thread assigns local tail as queue's tail and then checks if they are same or not, either other threads have enqueued successfully thereby changing queue's tail or other threads have changed queue's tail pointer to point to closest unmarked node. (E07) It checks if tail's next pointer is NULL or not. If it is NULL, it means the queue's tail is not lagging and it is pointing to the last node in the queue. (E08) If tail points to the last node, then the thread assigns the next pointer of the new node to point to a NULL node with deleted field 0 and tag field as 2+tail

field. (E09-E11) All the threads, who try the CAS operation will fall in the same bucket, since all these threads are concurrently enqueueing to the queue. Thus, the thread whose CAS operation of comparing atomically local tail's next to next and assigning the next pointer of the local tail to its new node succeeds will be the winner. That is, this thread will append its node to the queue's tail. (E10-E11) Winner thread does CAS on queue's tail to atomically check if its equal to local tail and assigns queue's tail to point to its new node and returns true, since it successfully appended its new node to queue. (E13-E18) Other threads who failed CAS operation on E09 further checks the nodes in queue by skipping logically deleted node and does CAS on first unmarked node (E16-E17) and if it succeeds, then it appends its new node before the local tail it has read initially. Thus all the nodes which failed CAS operation on E09 append their nodes in LIFO manner before the node inserted by the winner thread of CAS operation (see below figures). (E19-E22) If the queue's tail is lagging i.e it is not pointing to the last node, then the last node is searched and the queue tail is fixed. Thus fixing the queue's tail, thread retries its enqueue operation.

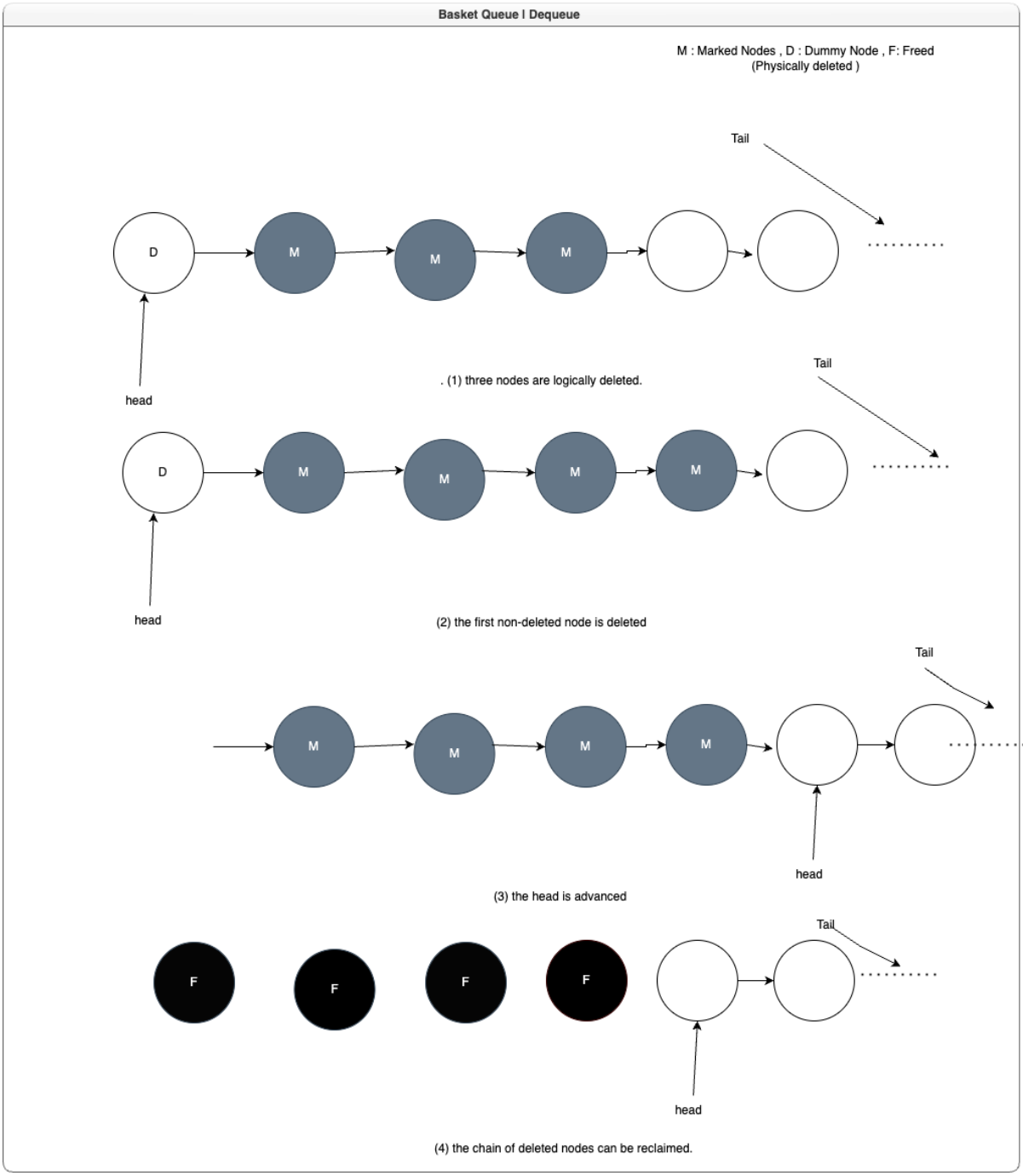
Method is lock-free since threads overlapping in the time interval of the winning thread of CAS operation are only finite. Thus, they will eventually complete their insertion (E13-E18). Threads which observe that their next node is not NULL (E19-E22) fixes the tail pointer of the queue and then retry their process of enqueue.

Dequeue algorithm

```
const MAX_HOPS = 3 # constant

data_type dequeue(queue_t* Q)

D01: repeat
D02:    head = Q->headf
D03:    tail = Q->tail
D04:    next = head.ptr->next
D05:    if (head == Q->head):
D06:       if (head.ptr == tail.ptr)
D07:          if (next.ptr == NULL):
D08:             return 'empty'
D09:          while ((next.ptr->next.ptr != NULL) and (Q->tail==tail)):
D10:             next = next.ptr->next;
D11:          CAS(&Q->tail, tail, <next.ptr, 0, tail.tag+1)
D12:       else:
D13:          iter = head
D14:          hops = 0
D15:          while ((next.deleted and iter.ptr != tail.ptr) and (Q->head==head)):
D16:             iter = next
D17:             next = iter.ptr->next
D18:             hops++
D19:          if (Q->head != head):
D20:             continue;
D21:          elif (iter.ptr == tail.ptr):
D22:             free_chain(Q, head, iter)
D23:          else:
D24:             value = next.ptr->value
D25:             if CAS(&iter.ptr->next, next, <next.ptr, 1, next.tag+1>):
D26:                if (hops >= MAX_HOPS):
D27:                   free_chain(Q, head, next)
D28:                return value
D29:             backoff-scheme()
```



MAX_HOPS is a constant declared as 3 which denotes the maximum number of logically deleted nodes in queue after which the logically deleted nodes will be physically deleted.
(D02-D04) Thread assigns local pointers head and tail as queue's head and tail respectively. It also assigns local pointer next as next pointer of its local head. (E05) It checks if head pointer and queue's head pointer are same or not. If they are not same, it means other threads have already deleted nodes and changed queue's head. Thus it retries. (E06-E11) If head and tail pointers are same that is they point to dummy node, then either (D07-D08) it means queue is empty, thereby returning empty or (D09-D11) queue's tail is lagging i.e. not pointing to last node of queue, thereby changing it to last node of queue and retries. (D12) Queue is not empty. (D13-D14) It assigns iter pointer pointing to local head pointer and initializes hop to 0. (D15-D18) It hops to next node until it finds first non logically-deleted node. (D19-D20) If it observes that at this point head is not same as queue's head, it means other threads have deleted nodes and changed queue's head pointer. Thus it retries. (D21-D22) If it finds that tail and iter are same i.e. it has reached end of list, it calls free_chain which physically deletes

logically deleted nodes between head and iter. (D23-D29) It retrieves value of next pointer. Note next pointer points to non marked node and iter points to previous node of next. It then checks if iter pointer's next is still next or not. If not, it means other threads have changed member fields of next node. If thread succeeds in performing CAS operation, then it marks the node by setting deleted field to true, thereby performing logical deletion. If it observes that more than MAS_HOPS nodes are logically deleted (marked), then it physically deletes those nodes thus reclaiming memory by calling free_chain.
Method is lock free since threads observing head and tail pointing to same node (D06-D11), either returns empty if queue is empty or fixes the tail node of queue. Else, all threads skip over all logically deleted threads and then do CAS (D25) to remove node pointed by next. All other threads who failed CAS (D25) retry their dequeue process.

Linearization points and Linearizability proof

Linearization point of enqueue of nodes inside a particular basket are set inside basket's shared time interval in the order of their respective dequeues i.e. linearization point of enqueues is decided only once items inside the baskets are dequeued.

Linearization point of dequeue returning a value (queue is not empty) is line D23 where the next pointer points to a non marked node (not logically deleted), thereafter the thread will mark it as logically deleted. Linearization point of dequeue returning empty (queue is empty) is line D04, where it reads the next pointer of head which is NULL.

Claim 1: Enqueue operations of same basket overlap in time

A new basket is created by the winner thread of CAS operation in (E09) of enqueue. Other threads who failed the CAS operation insert nodes in the same basket created by the winner thread. Thus the time interval in which these overlapping enqueue operations happen starts when the next pointer of the tail node was null and ends when winner threads successfully inserts its node to queue. Winner thread of CAS operation also overlaps the same interval too. Thus all enqueue operations of the same basket overlap in time.

Claim 2: Baskets are ordered according to order of their respective time intervals

A new basket is created when thread successfully executes CAS operation (E09) and appends nodes to the last node of the queue. Enqueue operations that failed the CAS operation (E09), retry their enqueue process to insert their nodes at the same list position they have read initially (in the same basket). Thus, the first node of the new basket is next to the last node of the previous basket and the last node of the basket is being inserted by the winner thread of CAS operation.

Claim 3: Linearization points of the dequeue operations of a basket come after the basket's shared time interval when all enqueue operations of same basket overlap

A node must be present in the queue for it to be dequeued. The first bin node is queued only after the winning thread succeeds in the CAS operation (E09) at the tail . Completion of a CAS operation marks the end of the shared time interval of the overlapping queue operations of that bin. Nodes can thus be marked (by dequeuers) only after the time interval of the bin has expired.

Claim 4: Bin nodes are dequeued before nodes of later bins, i.e. nodes of the previous bin are dequeued before nodes of the next bin.

Nodes are dequeued according to their sequential order in the queue (which in this case is logically divided into bins). Since once bin nodes are dequeued, i.e. logically removed, no further nodes are allowed to be added to the same bin.

Theorem: First In First Out queue is linearizable to a sequential First In First Out queue.

By Claim 2 and 4 above, we know the order in which basket's nodes are dequeued is same as order in which the nodes of these baskets are enqueued. By Claim 3, enqueue operation of a node precedes its dequeue operation. If we linearize the order of enqueue operations of a basket in the order of the nodes dequeued from that basket, then we can linearize the operations and the queue becomes linearizable.

Wait Free queue using Fast path and slow path methodology

We are trying to implement a wait free queue implementation using the method suggested by Kogan and Petrank's which is a fast-path-slow-path methodology to create our wait free queue.

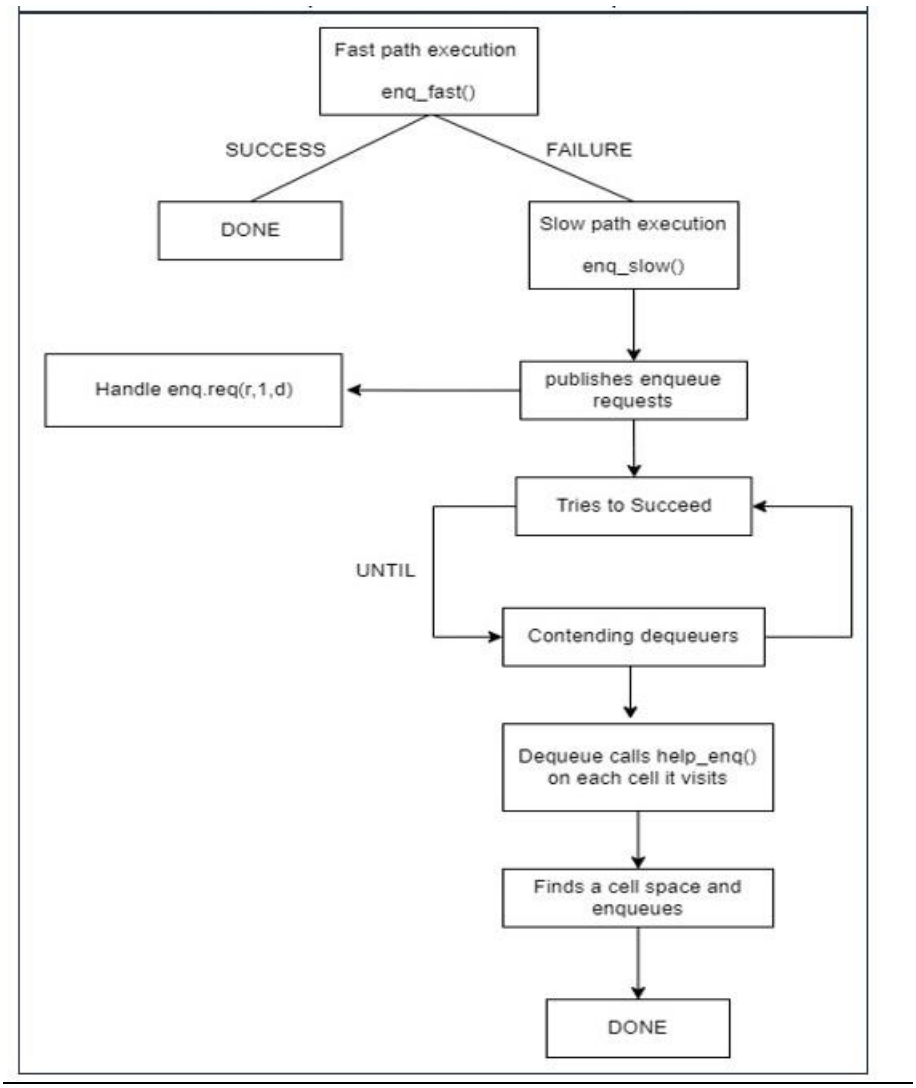
Our wait free queue uses operations like the regular enqueue and dequeue methods using the CAS and FAA operations as fast paths. We devised compatible, deliberate implementations of these operations to guarantee freedom from waiting. The enqueued operation initially endeavors to execute the swift path until it achieves success or encounters failures beyond a specified threshold. If required, it reverts to the deliberate slow path, ensuring eventual completion within a finite number of attempts. Subsequently, we outline the design of our high-level, wait-free queue implementation.

Ensure wait-freedom :-

1. **Enqueue**

Here, the whole aim is to make a particular operation wait-free by turning thecontenders' operations which compete i.e.there are the operations that run concurrently prevent this particular operation from succeeding, into helper operations, which help the operation to complete. Let us consider the case of an enqueueoperation on the fast-path,this operation may never complete its task because respective contending operations i.e. dequeues can mark every cell that they visit as not usable. To make them wait-free, a slow-path enqueue publishes an enqueue request. This request is made in Handle, which is the data structure that handles and stores the local state of a thread. Initially, the thread local states are linked in a ring; each dequeuer (contender)holds a pointer to the state of an enqueue peer. Whenever a dequeue operation changes a cell value to "bottom" marking it unusable, it helps its peer which is enqueue if the peer has a pending enqueue request in the handle. When that peer'srequest completes, the dequeue operation updates its peer pointer to the next enqueuer in the handle ring. Therefore, every time when an enqueue fails, its contending dequeuer which interrupts and block out enqueue, helps some enqueue request to complete. Eventually every pending enqueue request will succeed either by getting a location to enqueue itself or by getting help from the contending dequeuers. When all thecontending dequeuers become the helpers of pending enqueues, the enqueue will definitely succeed eventually.

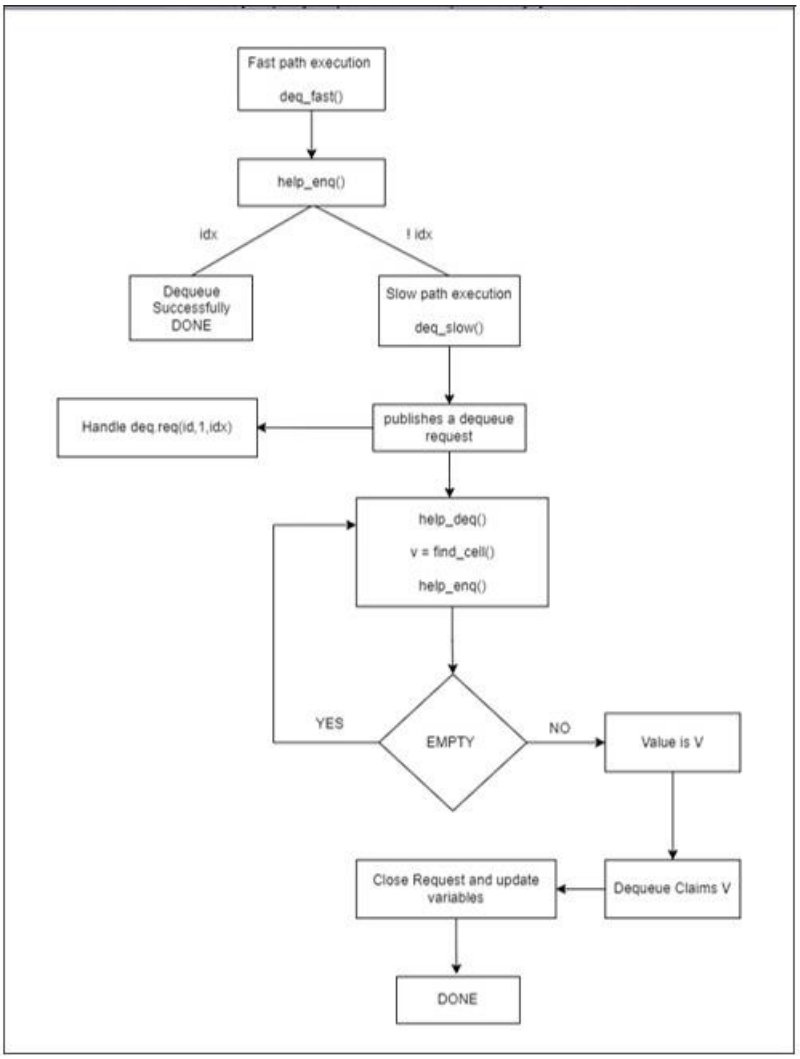
How Enqueue is Wait-Free



2. Dequeue

Likewise, a fast-path dequeue may encounter challenges in achieving success due to enqueued values being acquired by competing dequeuers. To ensure the wait-free nature of dequeues, each dequeuer retains a pointer to a dequeue peer. Upon successfully retrieving a value, a dequeuer assists its dequeue peer if the peer has an outstanding dequeue request. On the other hand, a slow-path dequeue and its assisting entities navigate through the queue to locate a value for dequeuing. A dequeue operation on the slow-path is bound to succeed eventually, as all competing dequeues will collaborate to facilitate its completion.

How Dequeue is Wait-Free



Ensure Linearizability:-

To ensure linearizability following invariants are taken into consideration for each case:-

1.Enqueue:-

Invariant 1: An enqueue can only deposit a value into a cell Q[i], where i<T when the enqueue completes.

Invariant 2: A cell Q[i] can only be reserved for enqueue requests r where r.id <= i.

Invariant 3: An enqueue helper might return EMPTY to the dequeuewhich it called on cell Q[i] only if no pending enqueue can put a valueinto Q[i] and the helper sees T <= i.

2.Dequeue:-

Invariant 1: A dequeue can only take a value from a cell Q[i] wherei<H where the dequeue completes

Invariant 2: After some helper saw a cell Q[i] with a "bottom" when T <= i and announced Q[i], a dequeue might return"EMPTY" a. i.e The dequeue may return EMPTY if somehelp_enqoperationon the last announced cell returned EMPTY.

Invariant 3: For a specific dequeue request k, every cell Q[i]: k.id <=i<= k.idx has been visited by at least one helper function.

Invariant 4: No new candidate may be announced in the future, if an announced candidate cell satisfies a dequeueoperation's request.

Flow of Execution:-

1. Enqueue

Initially when an enqueue is called, the thread tries to execute enq_fast (the fast path) until it succeeds by itself trying FAA or if it hits "patience" . Ifrequired, it will finish the enqueue using the slow-path execution methodology(enq_slow), which is sureto succeed. When fast path fails, it returnsthe cell index it obtained with FAA in an output parameter named "id" . This index nowbecomes the id of theenqueue request which isused by the slow-path method.

The initiation of the slow path execution for enqueuing starts with the publication of its enqueue request in the designated format. Subsequent to the completion of the publishing, for seeking assistance, a slow-path enqueue consistently strives for autonomous success by acquiring indices of extra cells in the queue through FAA. It endeavors to deposit its value into each potential cell. This iterative process persists until the enqueue successfully deposits its value into a cell or receives assistance from a competing dequeuer to achieve completion. Each dequeuer attempts to aid enqueue requests by invoking help enqon each cell it accesses

2. Dequeue

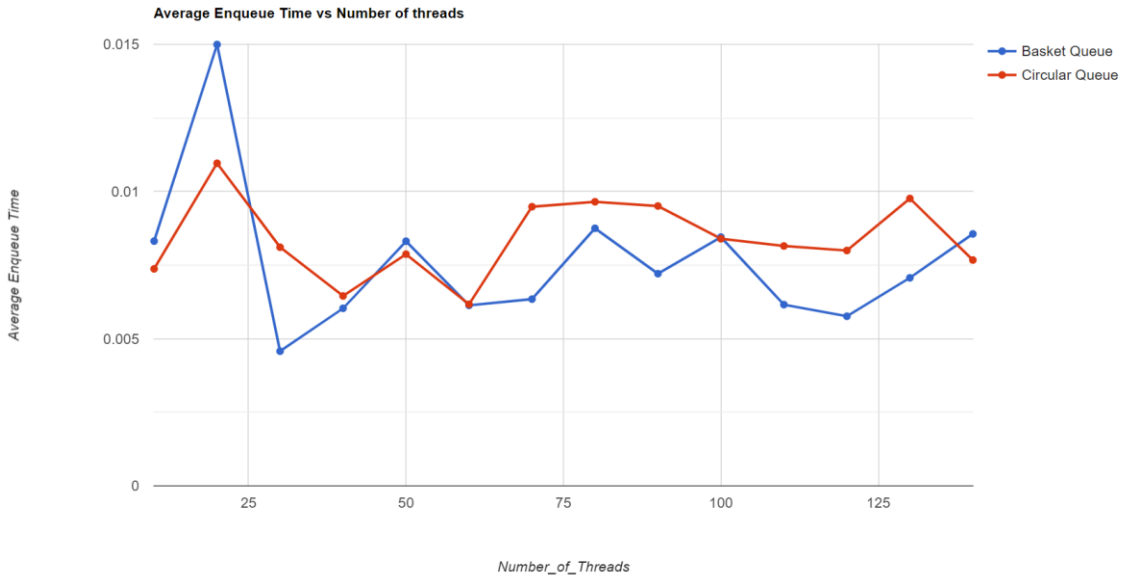
A fast-path dequeue closely resembles the standard dequeue process, with the additional step of invoking help enq on cells to attempt securing a value, providing assistance to enqueuers when necessary. If help enq yields a value, the dequeuer is required to assert the value by utilizing it. In cases where help enq returns "Bottom" or the dequeue process fails to claim the provided value, deqfast outputs the cell's index. This index serves as the identifier for the dequeue request on the slow-path , and the function returns "bottom" to signify the failure. In the event of a successful value claim, the dequeuer must aid its dequeue counterpart before returning the value.

A slow-path dequeuer commences by issuing a dequeue request to declare its intention. Subsequently, both the dequeuer and its assisting entities invoke helpdeq to finalize the pending request. As the dequeuer itself calls help deq to fulfill its own request, both the dequeuer and the assisting dequeuers are denoted as helpers. The help deq function guarantees the request's completion before returning.

Results and Graph Analysis :-

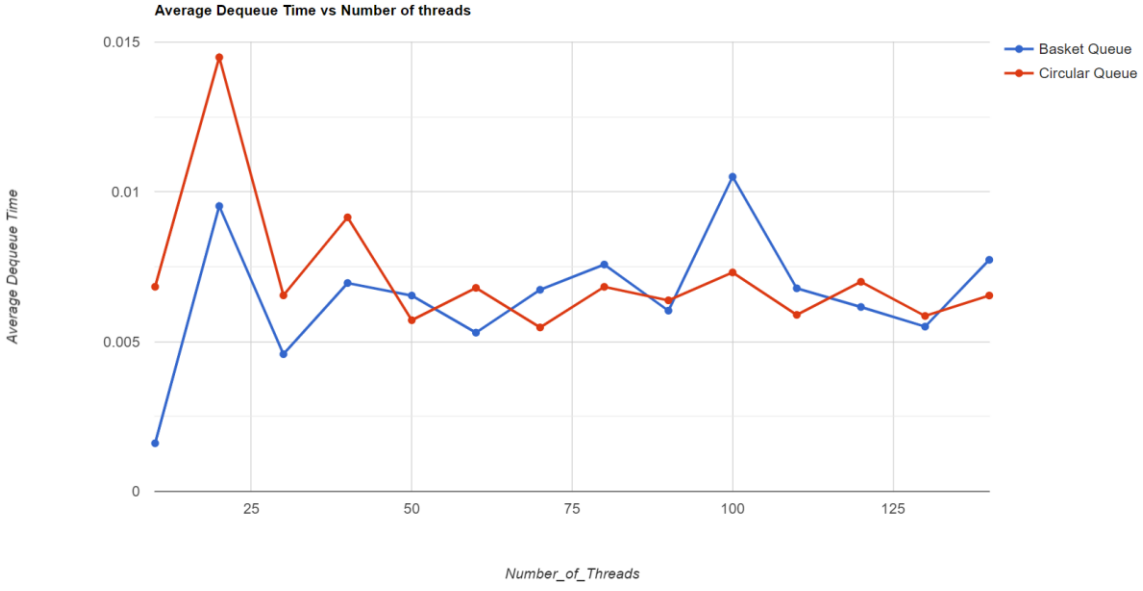
For all the graphs we have taken number of threads in the range 10 – 140, in increments of 10. i.e. 10,20,30,…..,140.

1. Average Enqueue time vs No-of-threads



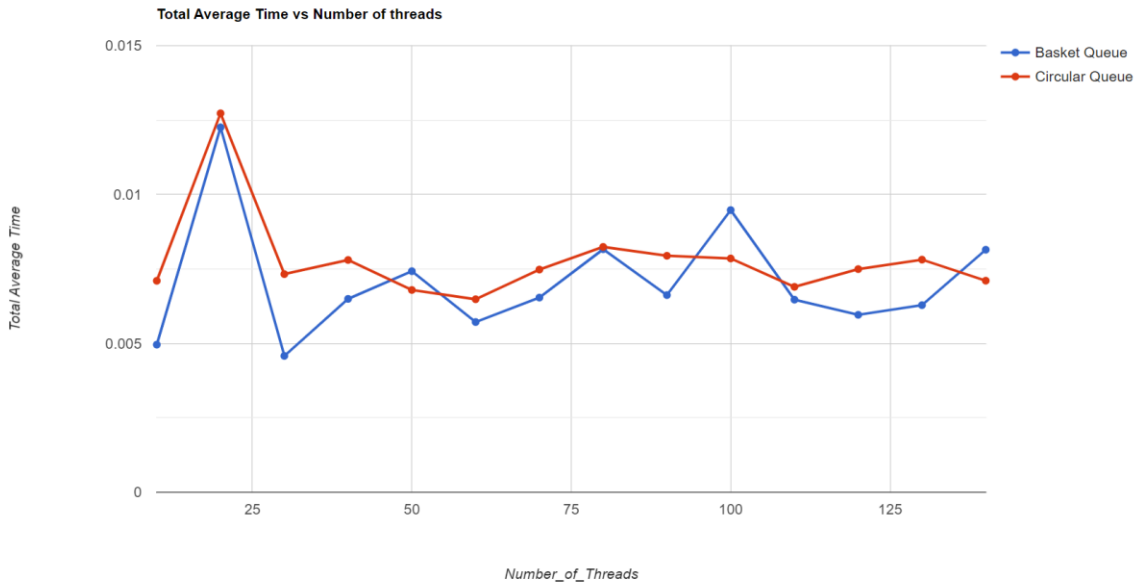
The above graph shows a plot between Average time taken by the enqueue operations and number of threads, it is evident that initially when the number of threads were increased from 10 to 20 there is sharp increase and immediately after that there is a fall and from that point we see fluctuations with increase in number of threads in the curves for basket queue within a constant range of (0.005-0.01) micro seconds. We also observe that when number of threads is greater than60 we see a trend, that the curve of circular queue is always above the curve of Basket queue. This indicates that Basket queue performs better than Circular queue after number of threads=60 till number of threads=130.

2. Average Dequeue time vs No-of-threads



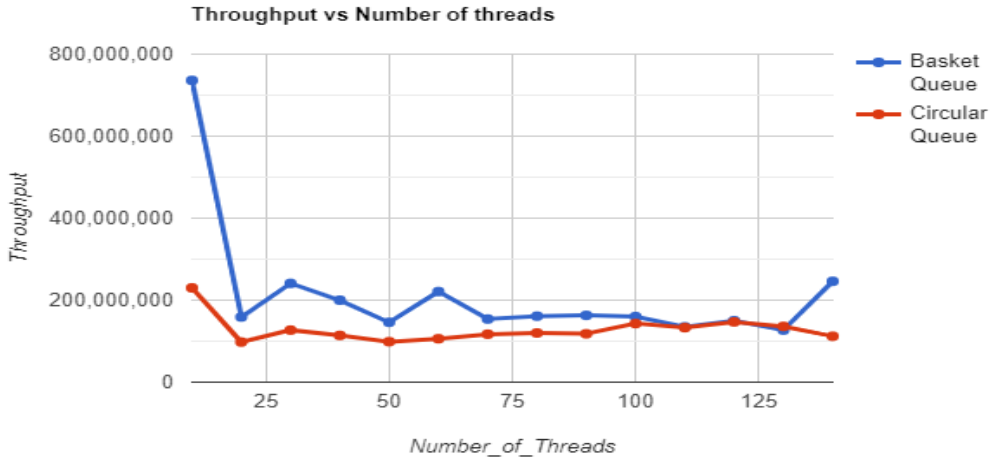
The above graph shows a plot between Average dequeue time and number of threads. We see that average dequeue time increases as the number of threads increase from 10 to 20. But average dequeue time decreases as we increase the number of threads from 20 to 30. After number of threads=30, we see that there is no strictly increasing or decreasing trends in the curves of basket and circular queue. The curves fluctuate between the range 0.0045 – 0.011 micro seconds.

3. Total Average Time vs No-of-threads



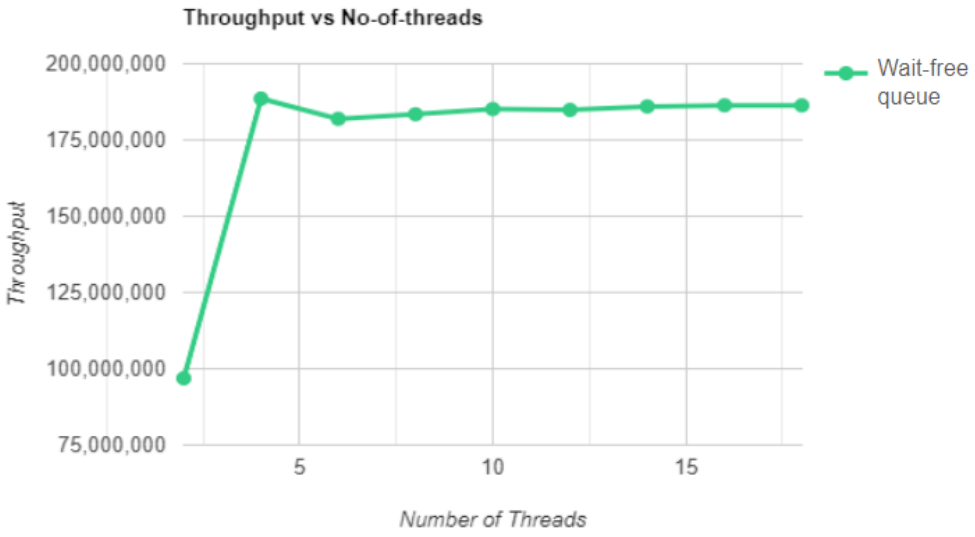
Similarly, The above graph shows a plot between Total Average time and number of threads.We see that Total average time increases as the number of threads increase from 10 to 20. But average dequeue time decreases as we increase the number of threads from 20 to 30. After number of threads=30, we see that there is no strictly increasing or decreasing trends in the curves of basket and circular queue. The curves fluctuate between the range 0.005 – 0.009 micro seconds.

4. Throughput vs No-of-threads (Circular queue versus Basket Queue)



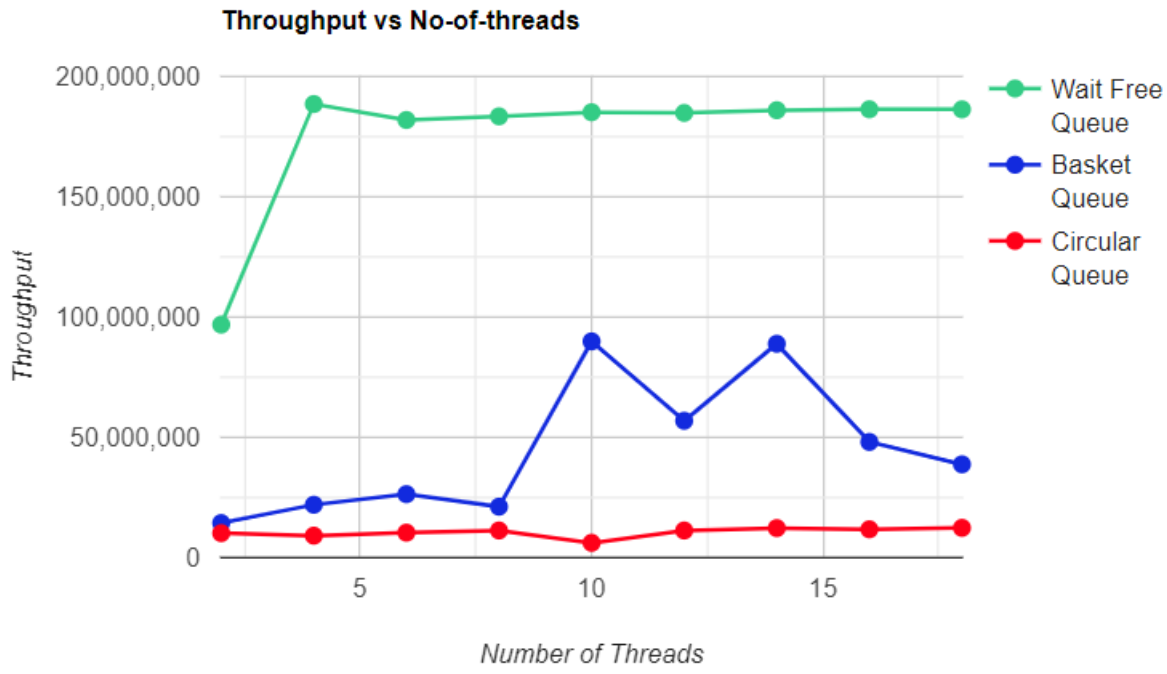
In the above graph, we observe a plot between Throughput(The total number of operations per second) and number of threads, where throughput faces a sharp decrease initially with increase in number of threads from 10 to 20 in the curves for basket queue and circular queue, with circular queue performing slightly poor compared to basket queue. After that point the curves show fluctuations, but we see that the trend is neither strictly increasing nor strictly decreasing.

5. Throughput vs No-of-threads (Wait Free Queue)



In the above graph, we observe a plot between Throughput(The total number of operations per second) and number of threads. For smaller threads, of range 2 to 20 in the multiples of 2, we see wait free queue's performance increases sharply and after a certain point attains a stable growth as the number of threads increases.

6. Throughput vs No-of-threads (Wait Free queue vs Circular queue vs Basket Queue)



In the above graph, we observe a plot between Throughput(The total number of operations per second) and number of threads. For smaller threads, of range 2 to 20 in the multiples of 2, we see wait free queue is performing significantly better compared to circular queue and basket queue. We also observe that basket queue is still performing comparatively better than circular queue even for smaller number of threads.

System Specifications :

Model Name:	**MacBook Air**
Model Identifier:	**Mac14,2**
Chip:	**Apple M2**
Total Number of Cores:	**8 (4 performance and 4 efficiency)**
Memory:	**8 GB**
System Firmware Version:	**10151.41.12**
OS Loader Version:	**10151.41.12**

References

- The Baskets Queue by Moshe Hoffman1, and Nir Shav (Baskets Queue)
- A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue by Ruslan Nikolaev (Circular Queue)
- A Wait-free Queue as Fast as Fetch-and-Add Chaoran Yang John Mellor-Crummey