Lock-Free FIFO Queues

J. Barnat and P. Ročkai

ParaDiSe
Parallel & Distributed
Systems Laboratory

Masaryk University
Czech Republic

redhat.

Motivation

- Many parallel algorithms require continuous communication.
- Lock-free FIFO can be used as a message queue.
- A light-weight alternative to MPI.

Formulation of the problem:

- Two threads, A and B, and a common queue, Q.
- Thread A adds items to Q.
- Thread B removes items from Q.

Reader/Writer Benchmark

- **reader:**

```
int n = 0, o = 0;
while ( n < items )
   if ( !q.empty() ) {
      int o = q.dequeue();
      assert_eq( n + 1, o );
      n = o;
   }
```

- **writer:**

```
int n = 0;
while ( n < items )
   q.enqueue( ++ n );
```

Reader/Writer Benchmark

Numbers in the slides:
Intel Core 2 Duo @ 2.4GHz
$16 \times 1024 \times 1024$ ($= 2^{24}$) items.

Download the sources from:
`http://web.mornfall.net/stuff/slides/fifos.tar.gz`

Naïve Mutexed Queue

- `std::deque< T > q; wibble::Mutex m;`
- **empty?:**

```
m.lock();
bool empty = q.empty();
m.unlock();
return empty;
```

- **enqueue:**

```
m.lock();
q.push_back();
m.unlock();
```

- **dequeue:**

```
m.lock();
T x = q.front();
q.pop_front();
m.unlock();
return x;
```

Naïve Mutexed Queue

Pros:

- Trivial to implement.
- Good **sequential** performance.

Cons:

- Very poor concurrent performance,
- due to lock contention.

(Reader/writer: about 3.5M integers per second.)

Can We Do Better?

- Yes!
- A FIFO queue can be implemented using 2 pointers/indices, such that each of them is only ever written by one of the threads.
- Careful sequencing is required for correctness.
- Emptiness check (done by the reader) needs to *look at* (but not change) the writer variable.

Bounded Queue: Circular Buffer

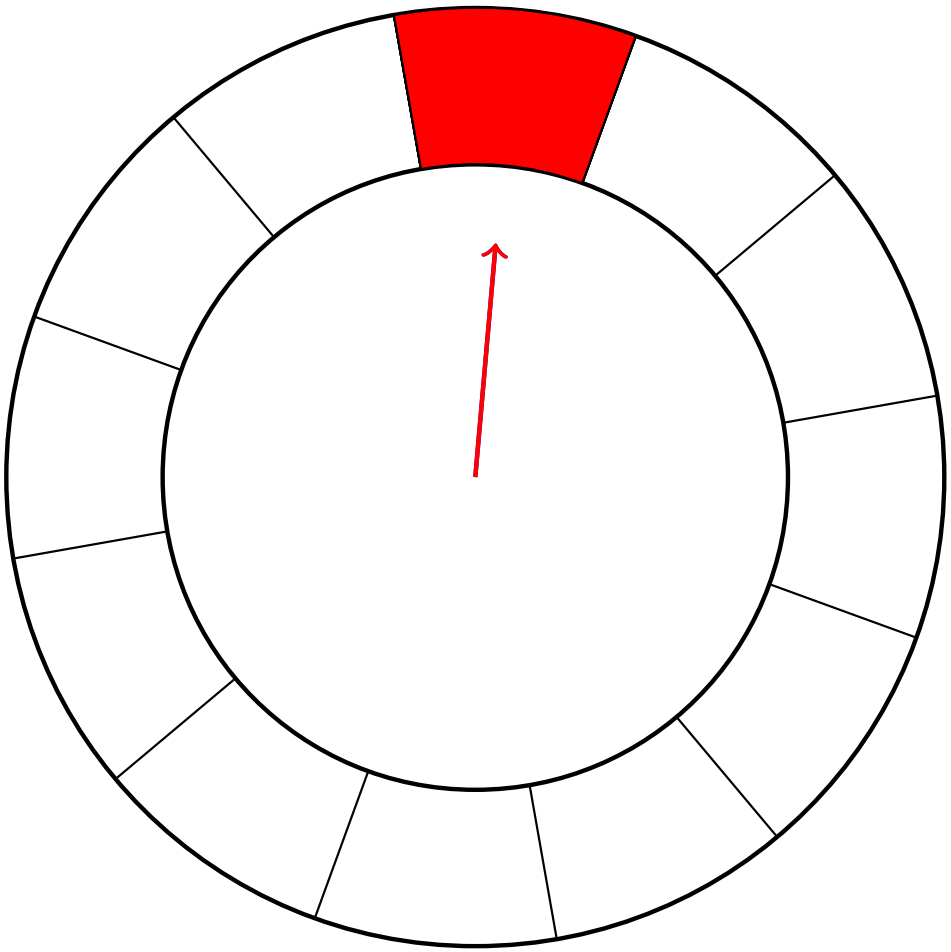
- `int reader = 0; T q[ size ]; int writer = 0;`
- **empty?:** `reader == writer;`
- **enqueue:**

```
while ( (writer + 1) % size == reader );
q[ writer ] = x;
writer = (writer + 1) % size;
```

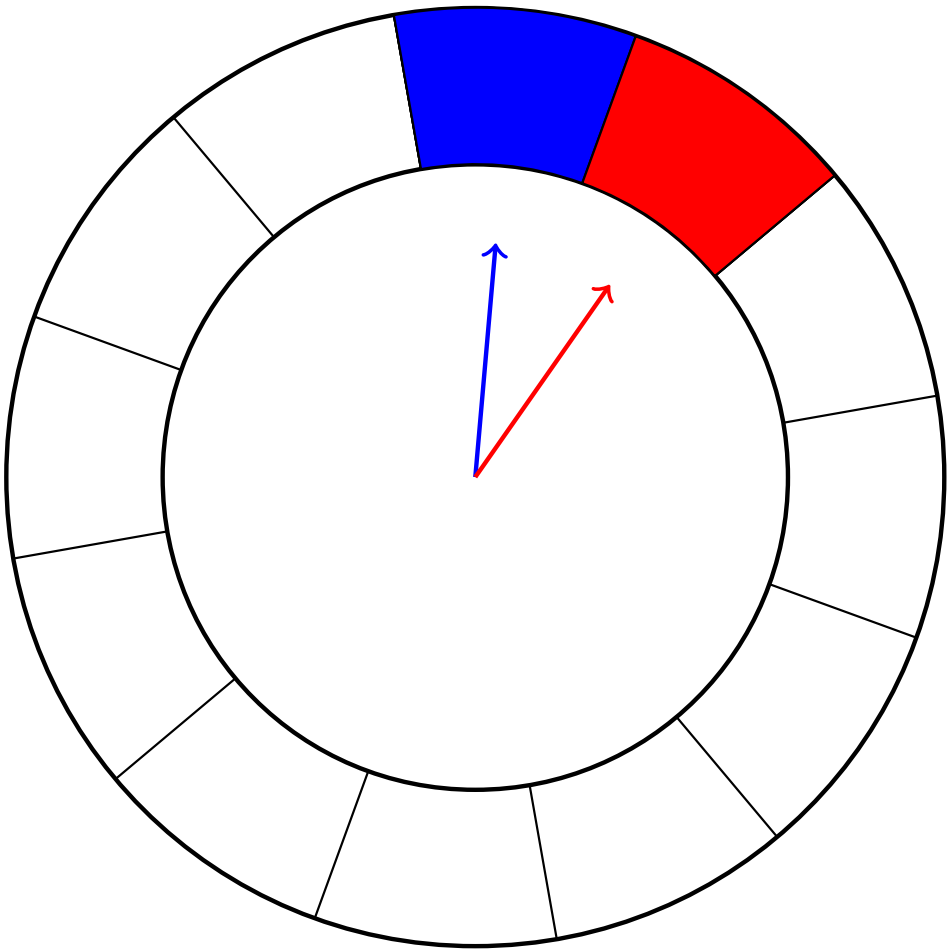
- **dequeue:**

```
T x = q[ reader ];
reader = (reader + 1) % size;
return x;
```

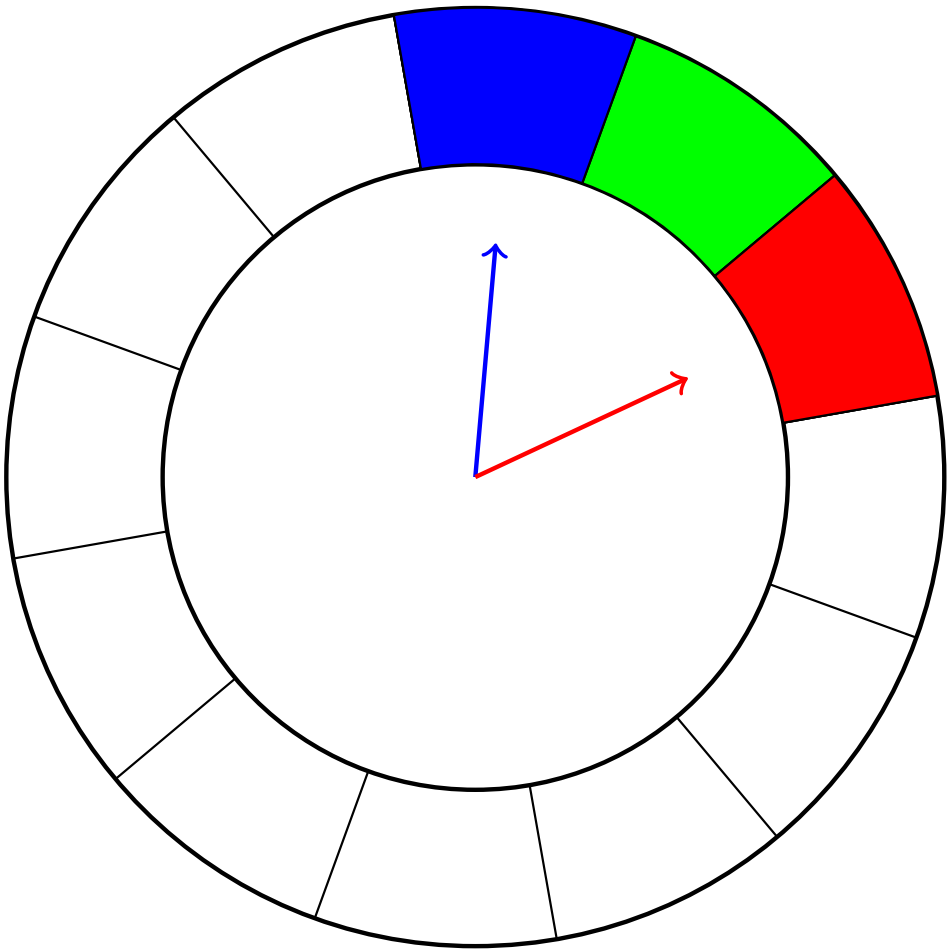
Bounded Queue: Circular Buffer



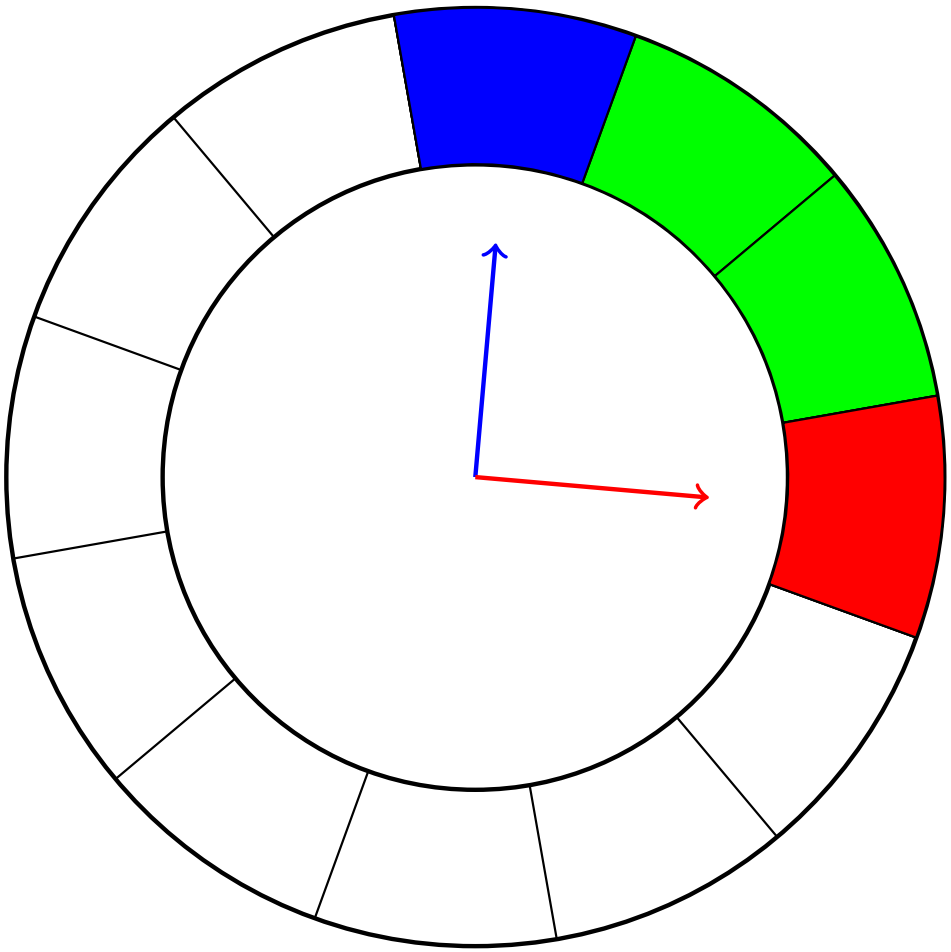
Bounded Queue: Circular Buffer



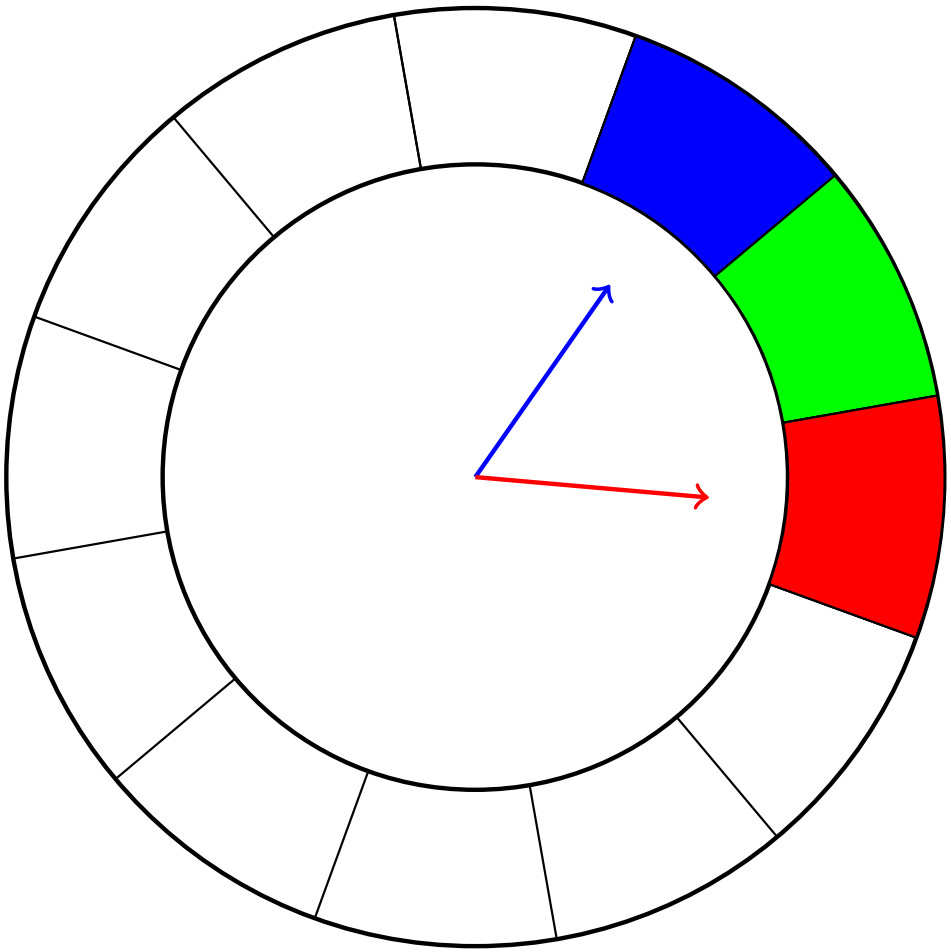
Bounded Queue: Circular Buffer



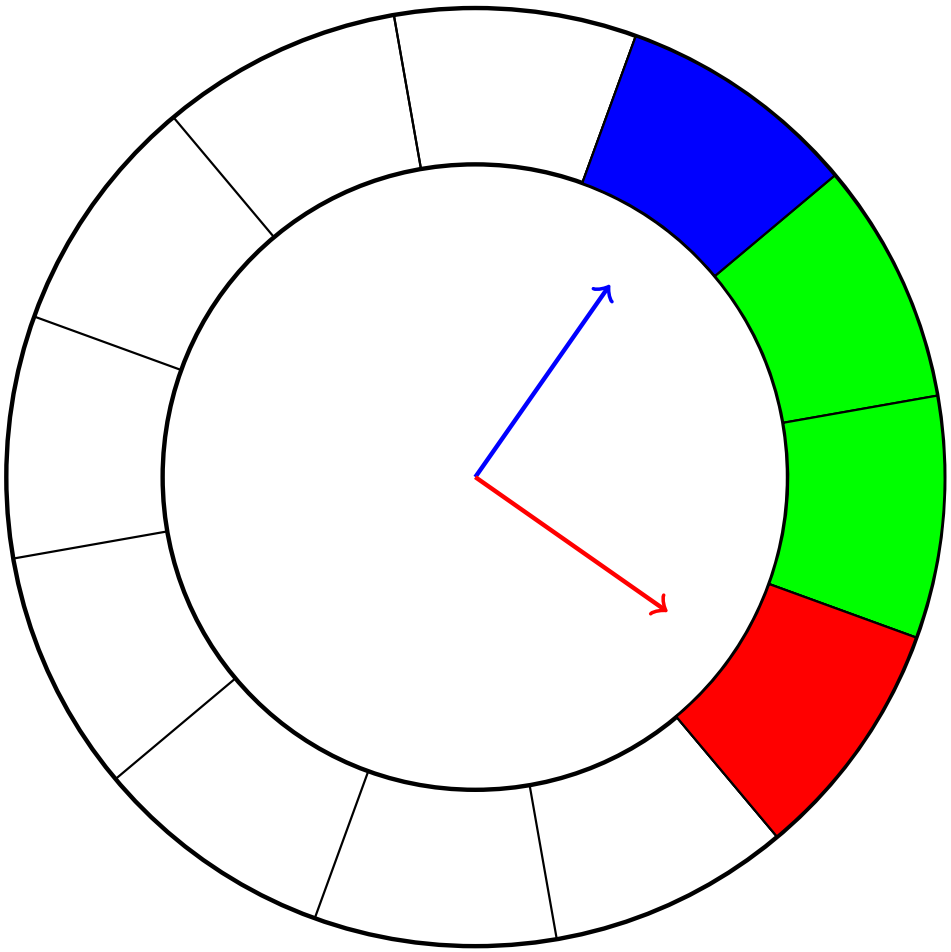
Bounded Queue: Circular Buffer



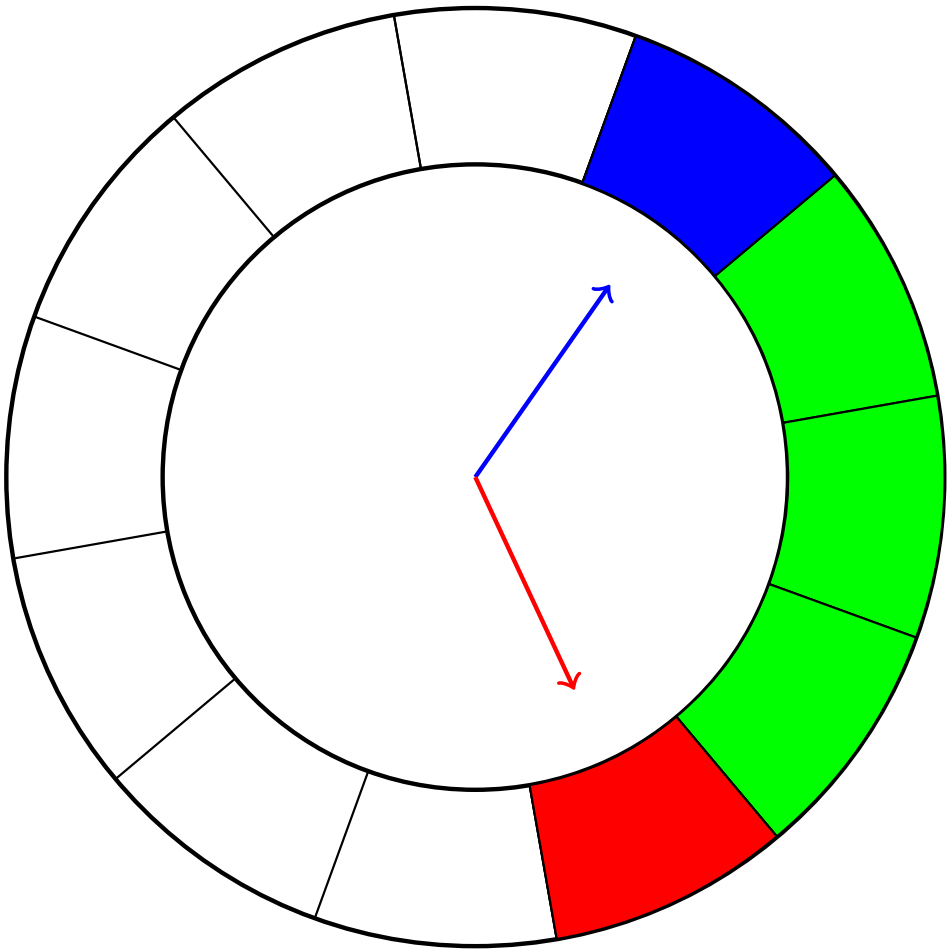
Bounded Queue: Circular Buffer



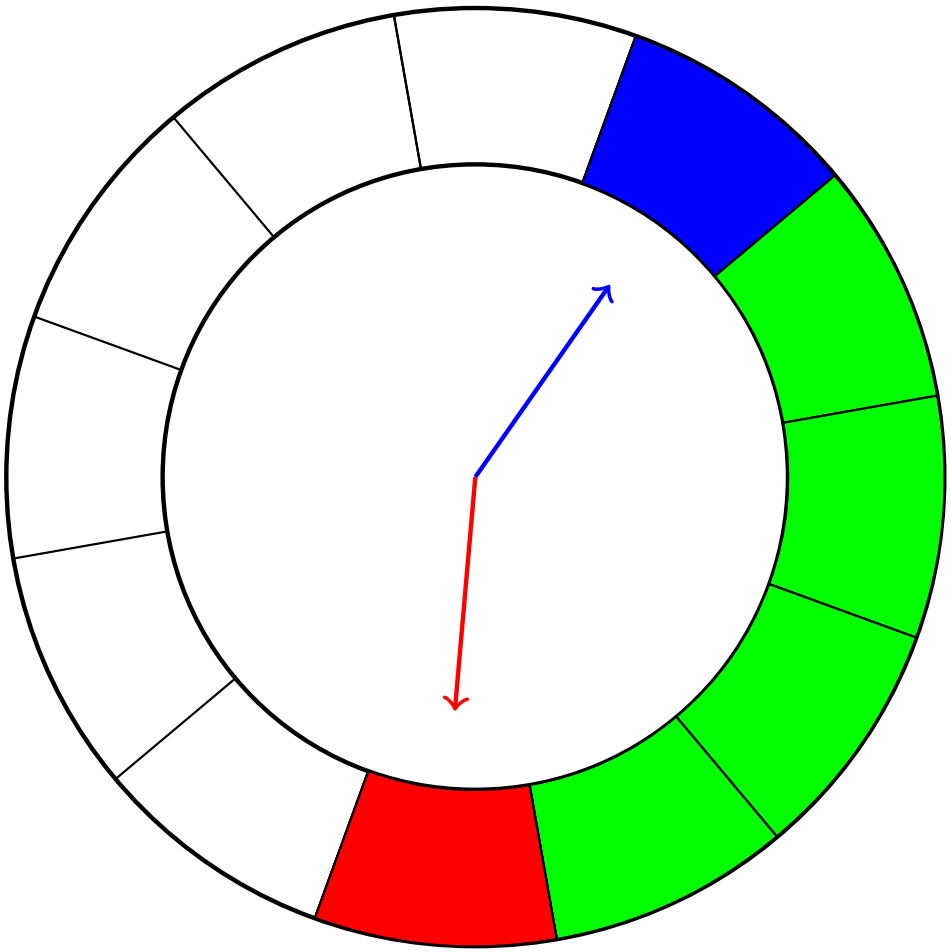
Bounded Queue: Circular Buffer



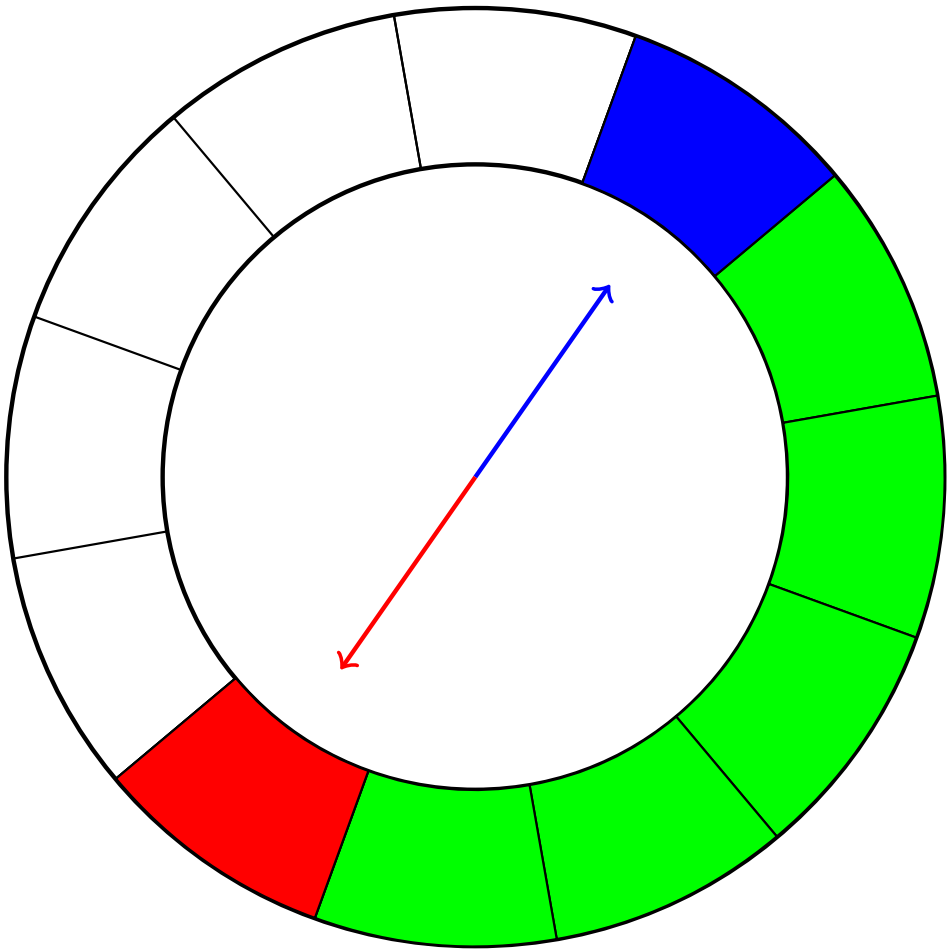
Bounded Queue: Circular Buffer



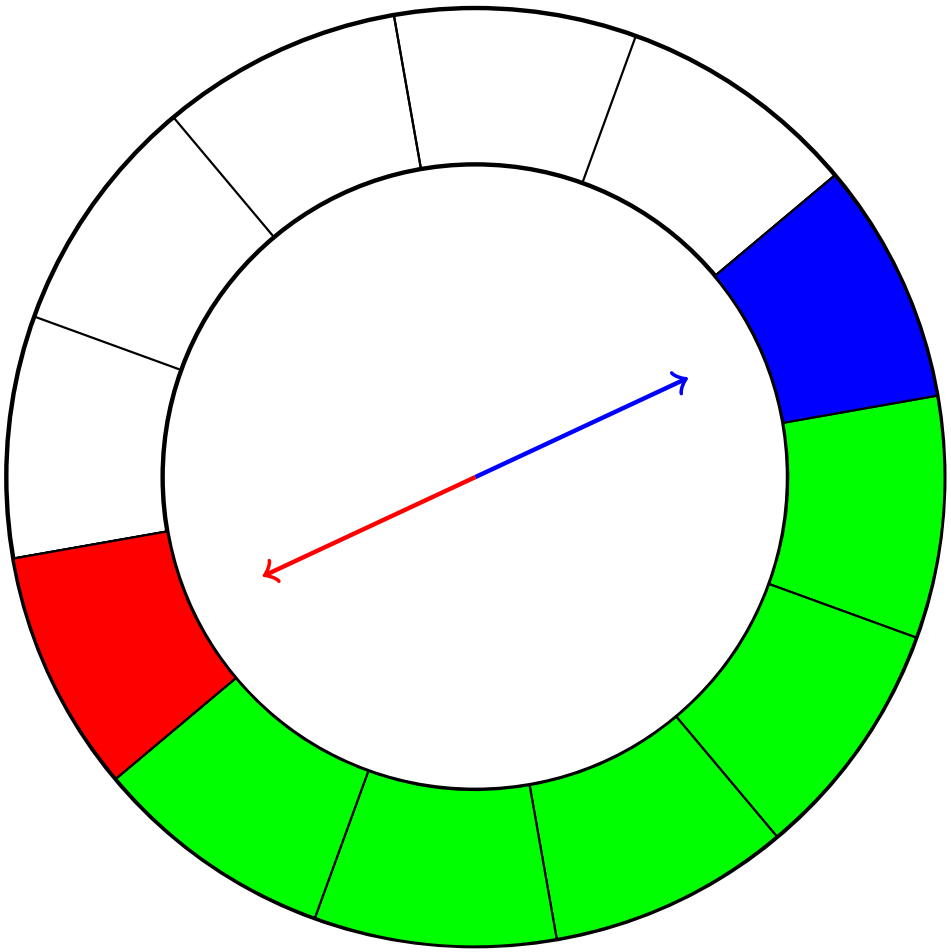
Bounded Queue: Circular Buffer



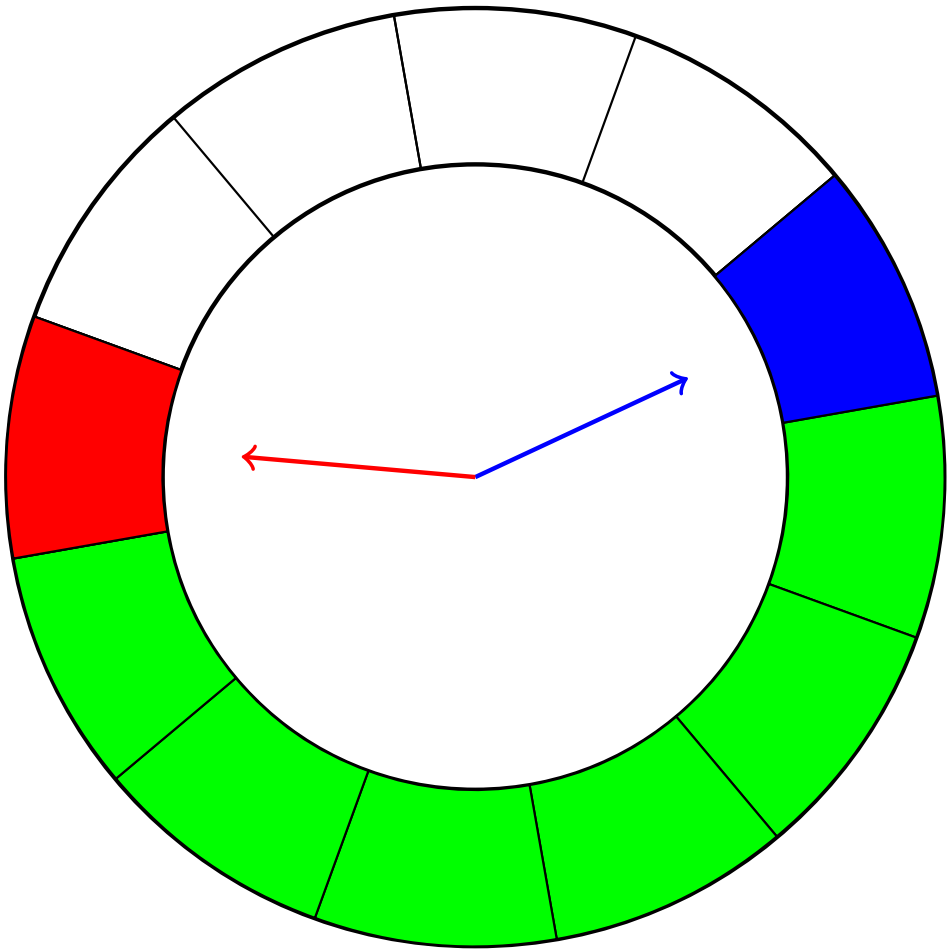
Bounded Queue: Circular Buffer



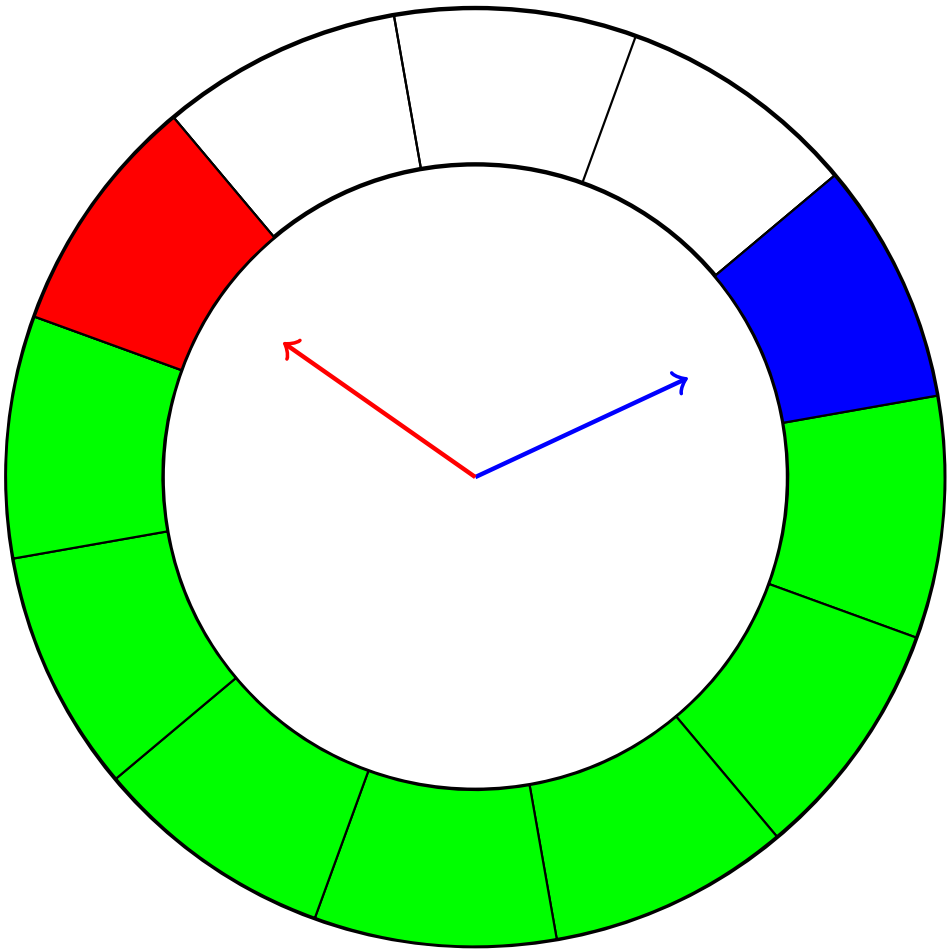
Bounded Queue: Circular Buffer



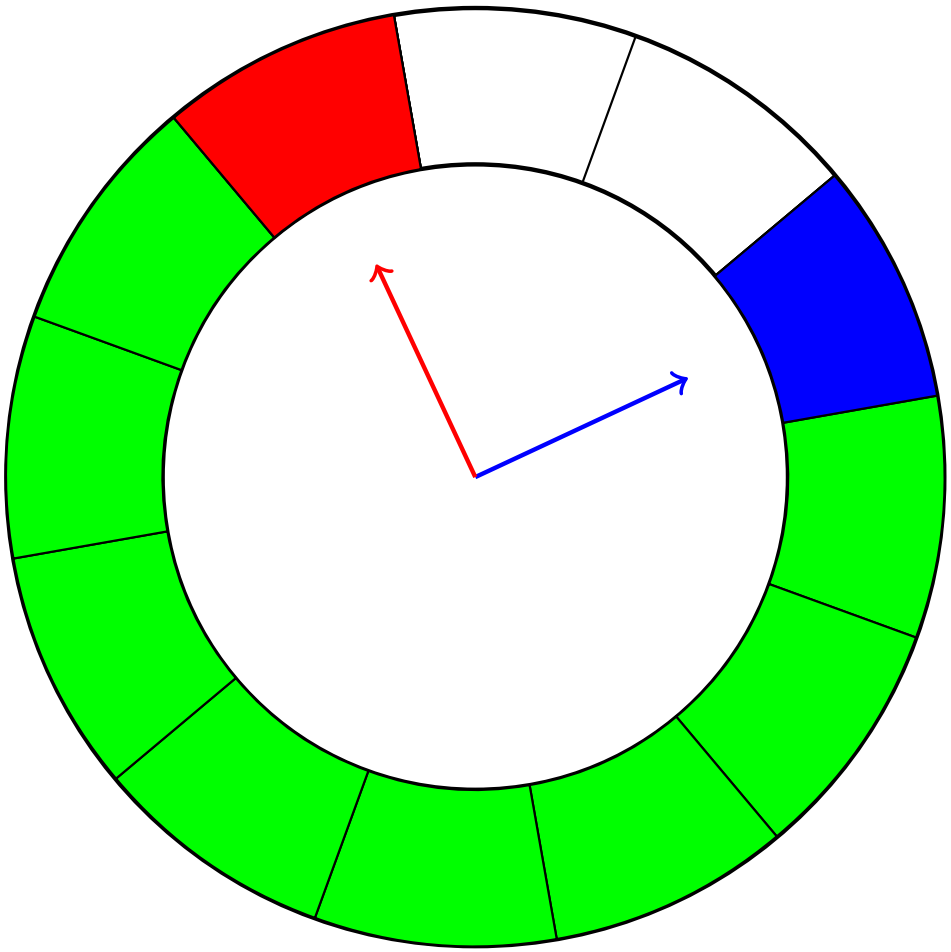
Bounded Queue: Circular Buffer



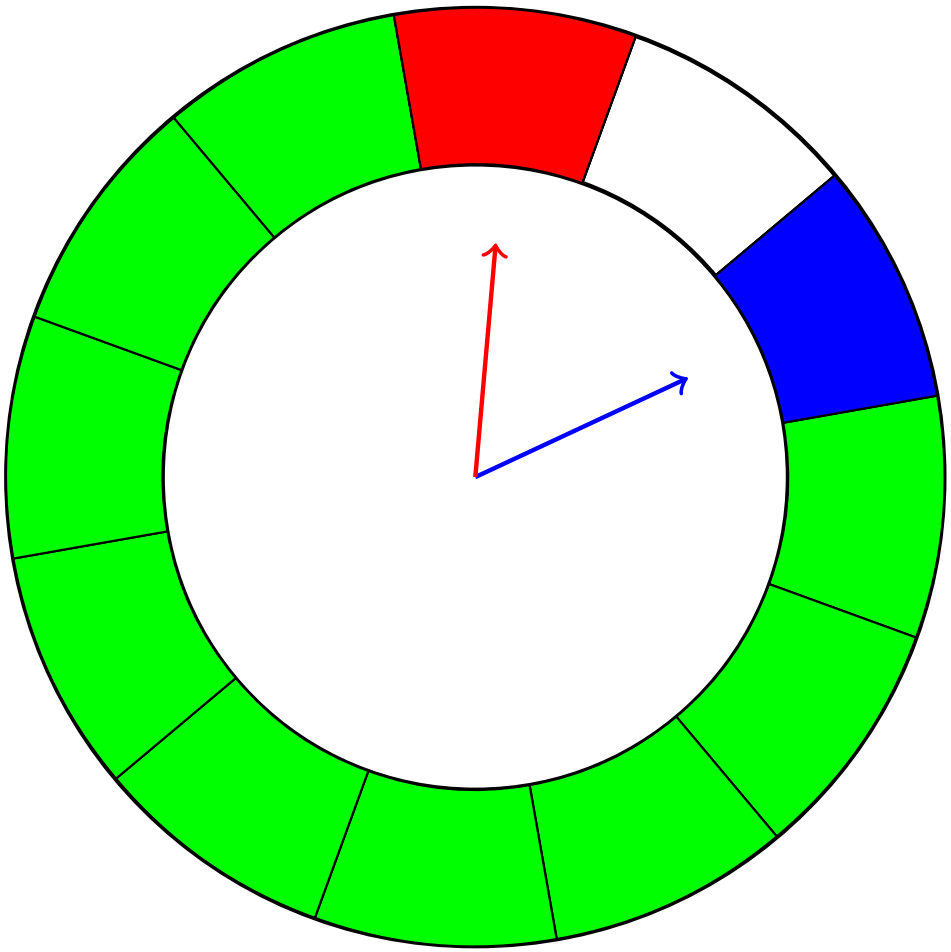
Bounded Queue: Circular Buffer



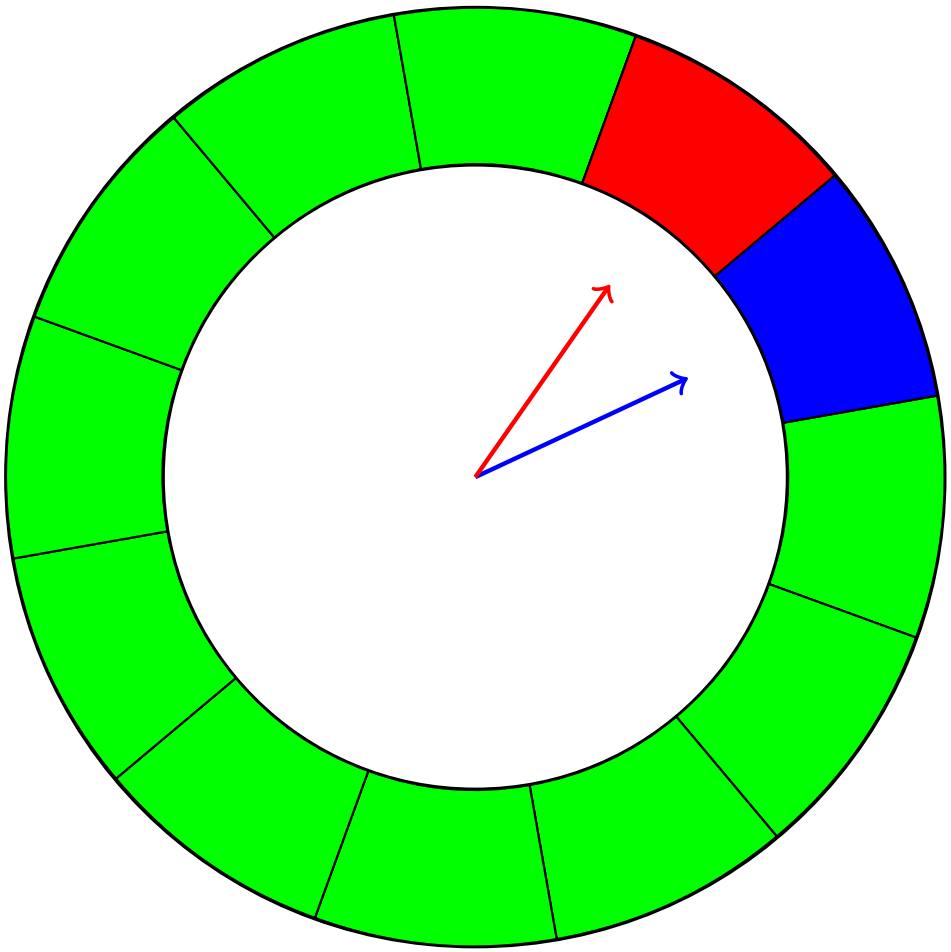
Bounded Queue: Circular Buffer



Bounded Queue: Circular Buffer

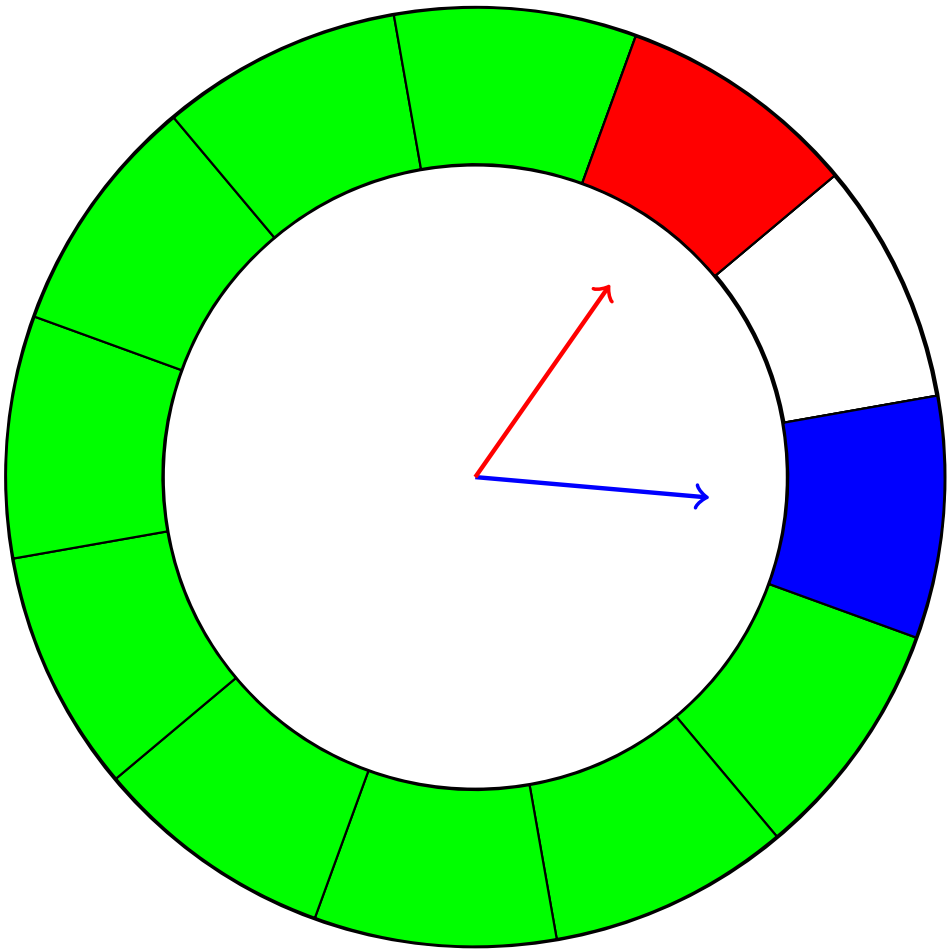


Bounded Queue: Circular Buffer



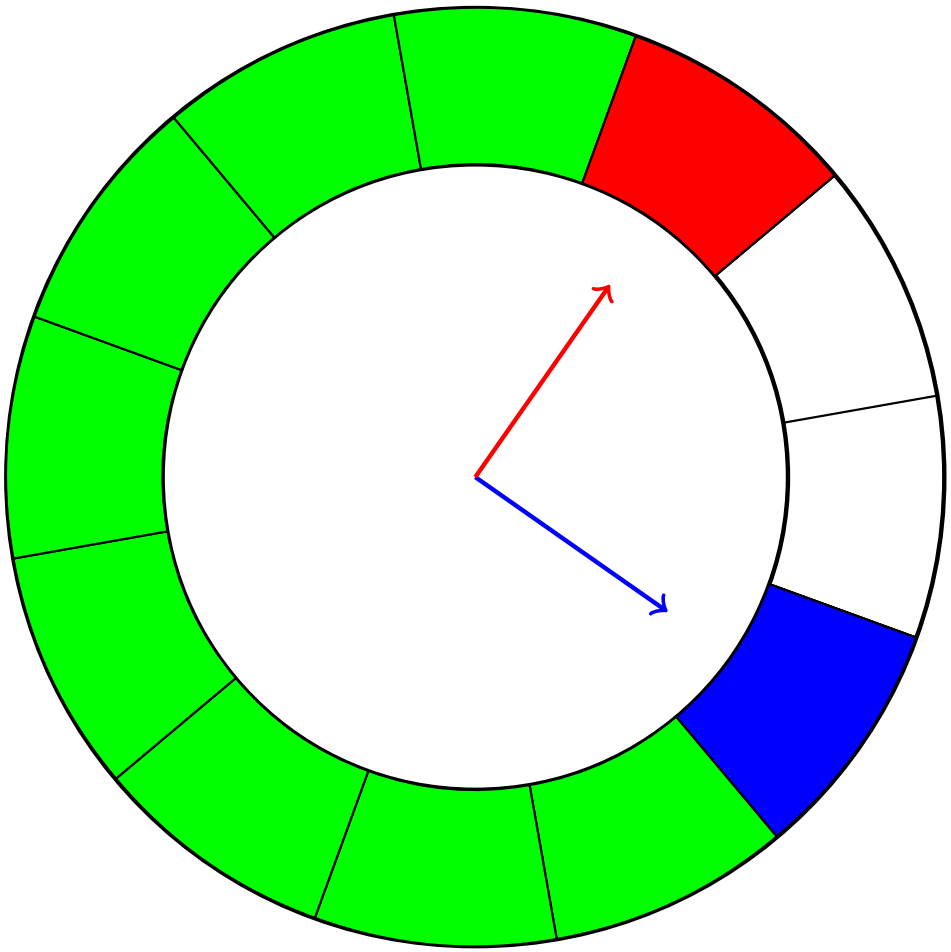
Writer blocks!

Bounded Queue: Circular Buffer

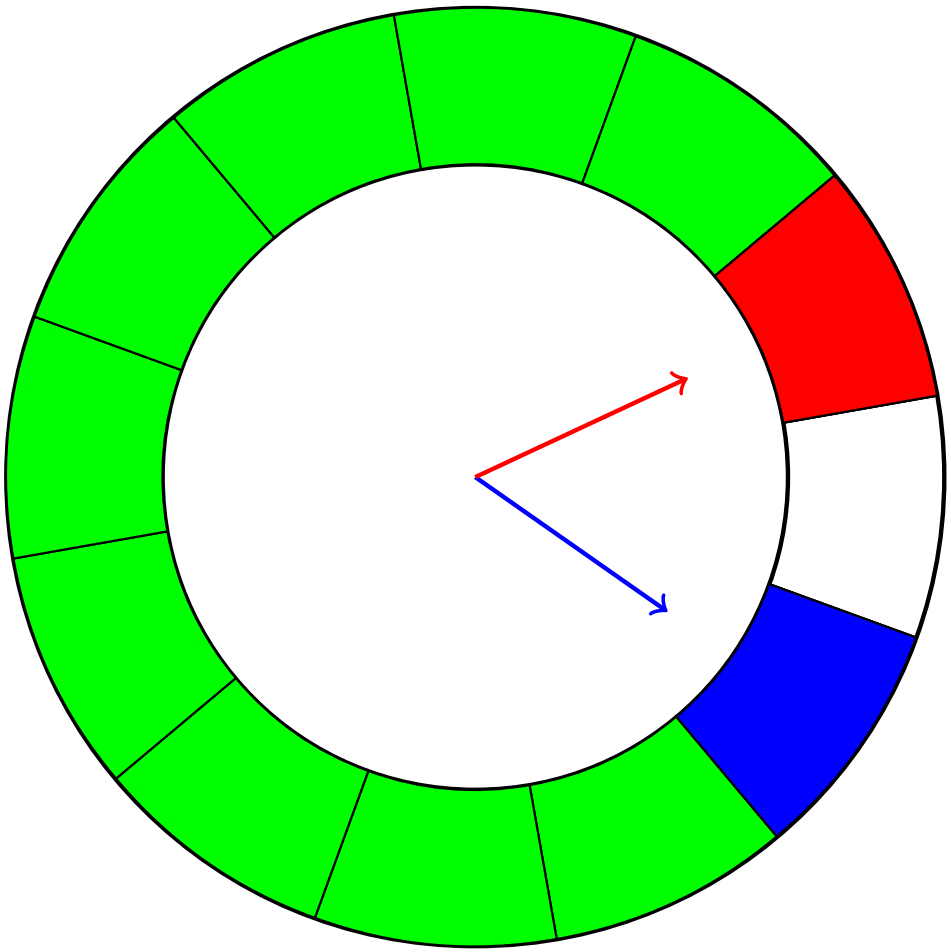


Writer is unblocked.

Bounded Queue: Circular Buffer



Bounded Queue: Circular Buffer



Bounded Queue: Circular Buffer

Pros:

- Simple to implement.
- Very good performance.

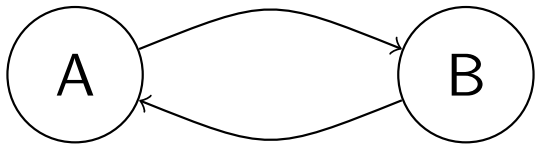
Cons:

- May slow down writers if readers are lagging.
- Prone to deadlocks.

(Reader/writer: about 26.7M integers per second.)

Bounded Queue Deadlock

- If there is a cycle in the communication graph,
- and the communication is unbounded.



1. Two channels, $A \to B$ and $B \to A$.
2. The channel $A \to B$ fills in.
3. A tries to write another item, but blocks.
4. B continues to write, but A is not reading (it is blocked).
5. Eventually, the channel $B \to A$ fills in.
6. Neither of A, B can make any further progress.

Unbounded Queue: Linked List

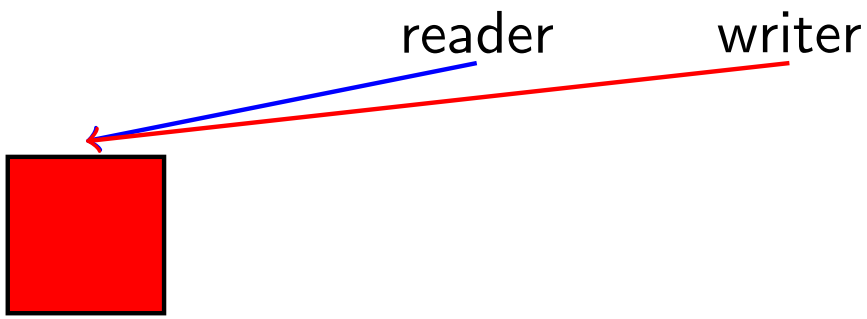
- **Node:** `Node *next; T value;`
- `Node *reader; Node *writer;`
- **empty?:** `reader == writer;`
- **enqueue:**

```
Node *n = new Node;
n->value = x;
writer->next = n;
writer = n;
```

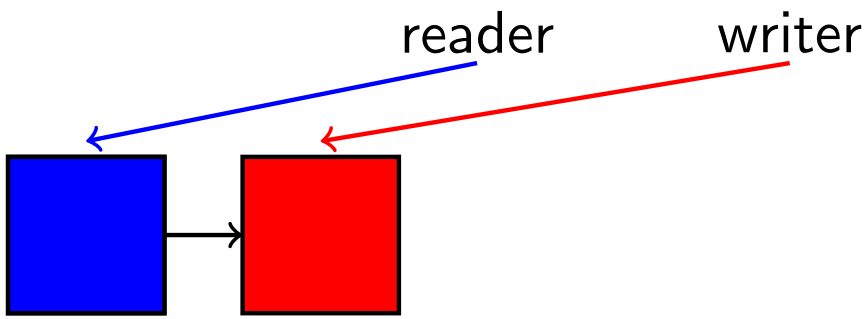
- **dequeue:**

```
Node volatile *n = reader;
assert( reader->next );
reader = reader->next;
delete n;
return reader->value;
```

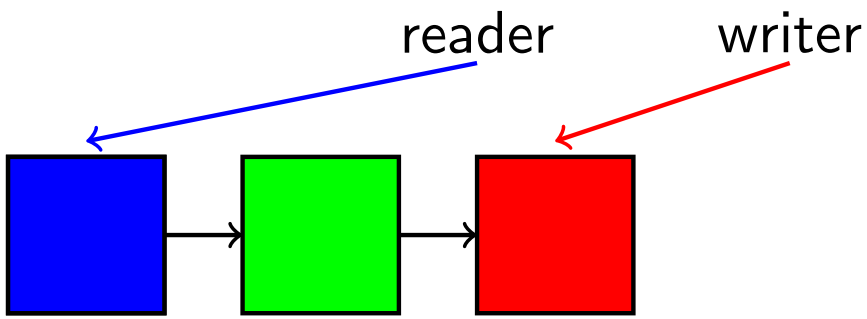
Unbounded Queue: Linked List



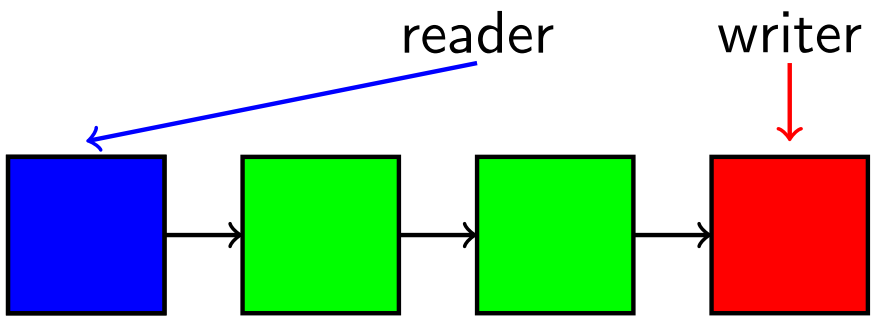
Unbounded Queue: Linked List



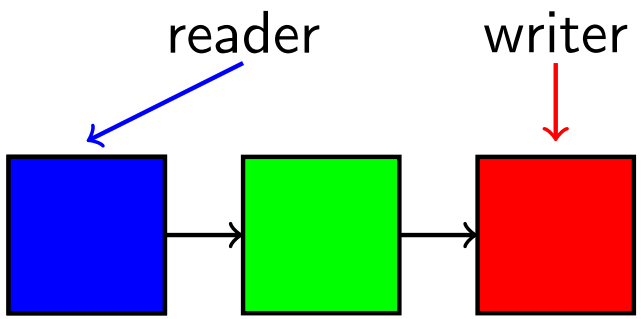
Unbounded Queue: Linked List



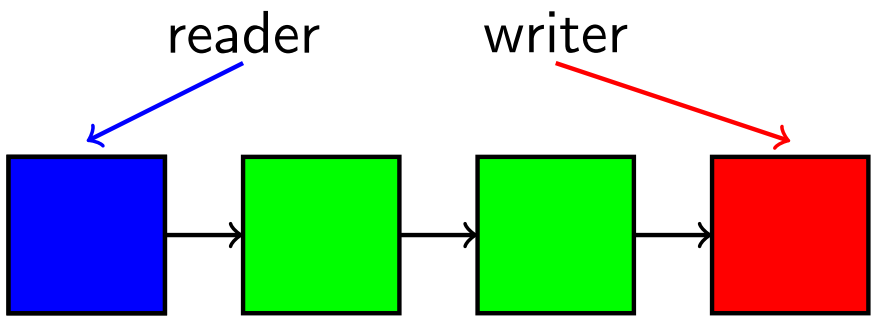
Unbounded Queue: Linked List



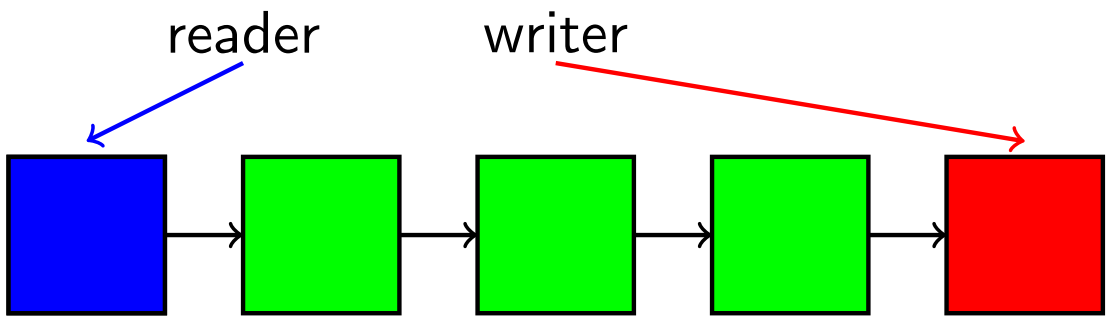
Unbounded Queue: Linked List



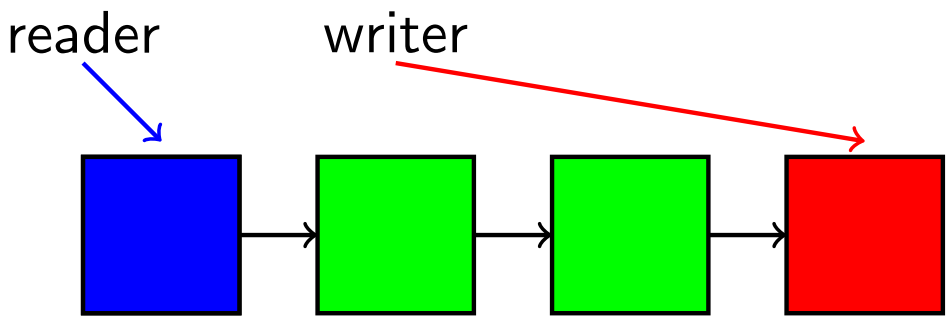
Unbounded Queue: Linked List



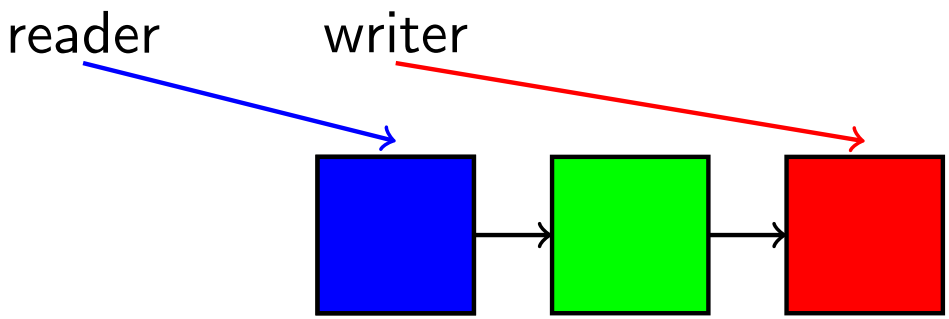
Unbounded Queue: Linked List



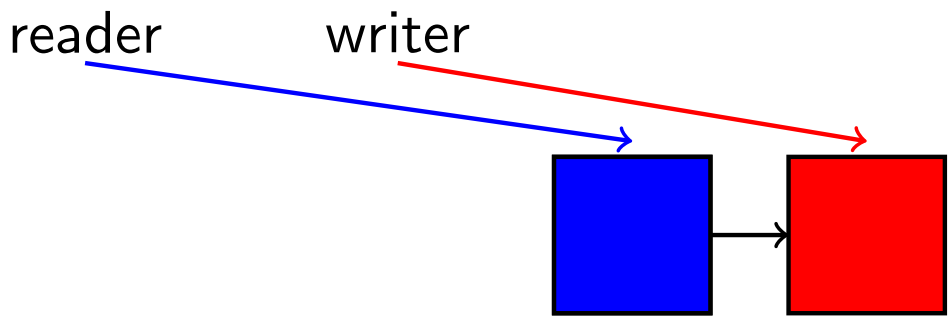
Unbounded Queue: Linked List



Unbounded Queue: Linked List



Unbounded Queue: Linked List



Unbounded Queue: Linked List

Pros:

- Simple to implement.
- Unbounded.

Cons:

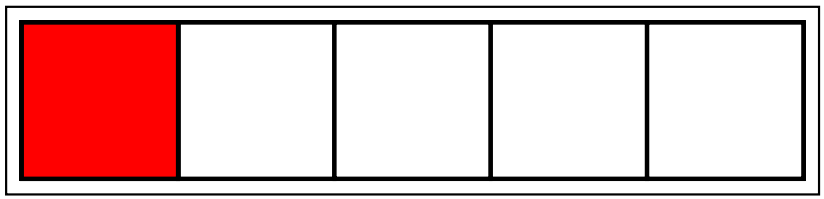
- Not very fast.
- Poor data locality.
- Lots of space overhead (extra pointer per item).
- **NB.** Prone to cache line sharing.

(Reader/writer: about 12M integers per second.)

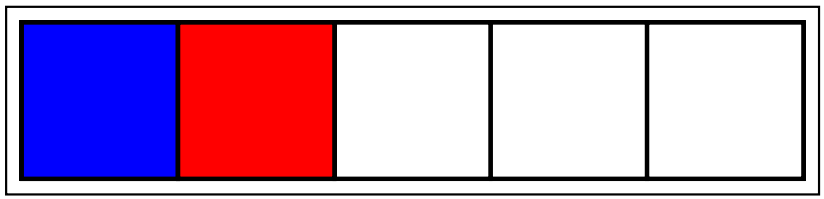
Unbounded Queue: Hybrid

- **Node:** `T *read; T buffer[ pernode ]; T *write;`
  `Node *next;`
- `Node *reader; Node *writer;`
- **empty?** `reader == writer &&`
  `reader->read >= reader->write;`
- **enqueue/dequeue**: Oops. This slide is too small for that.
- (but you can have the C++ sources)

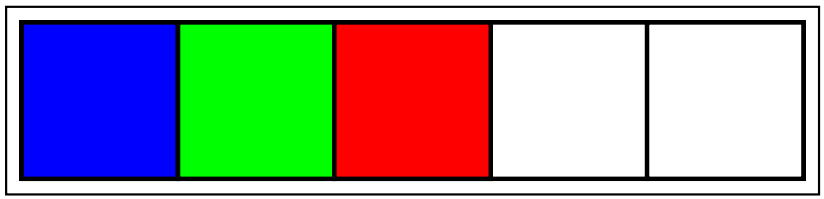
Unbounded Queue: Hybrid



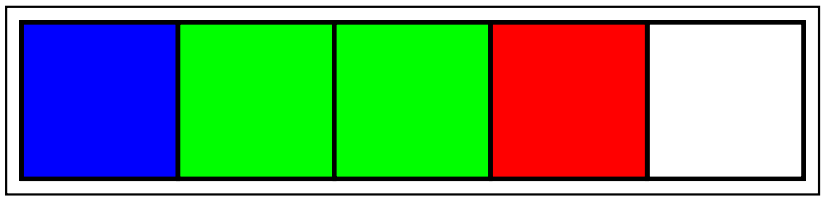
Unbounded Queue: Hybrid



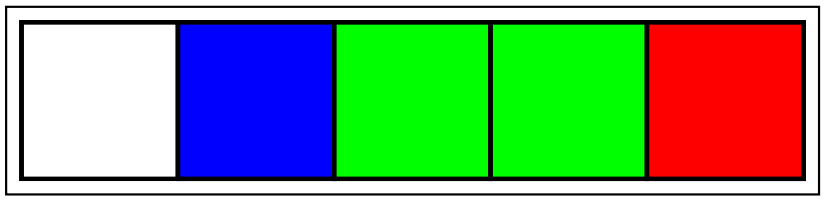
Unbounded Queue: Hybrid



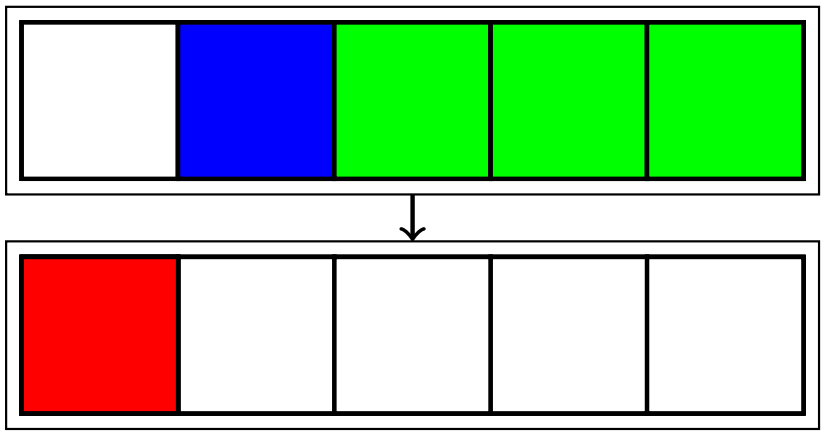
Unbounded Queue: Hybrid



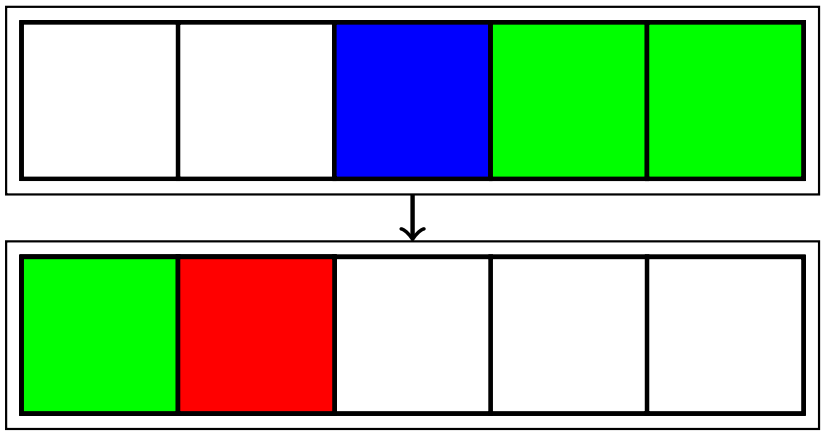
Unbounded Queue: Hybrid



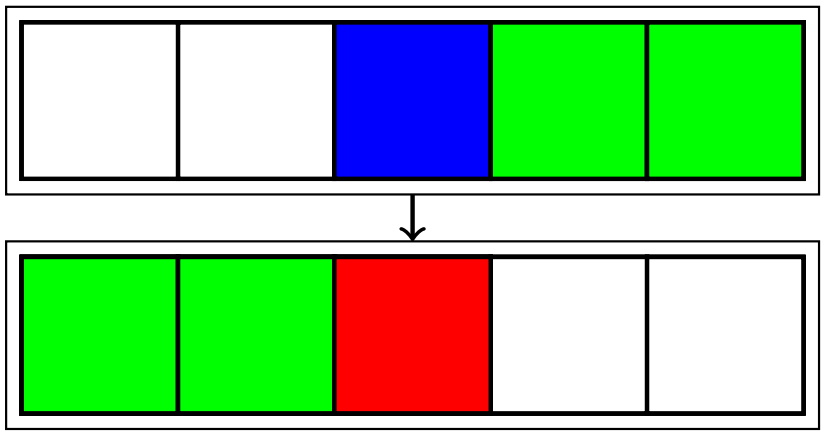
Unbounded Queue: Hybrid



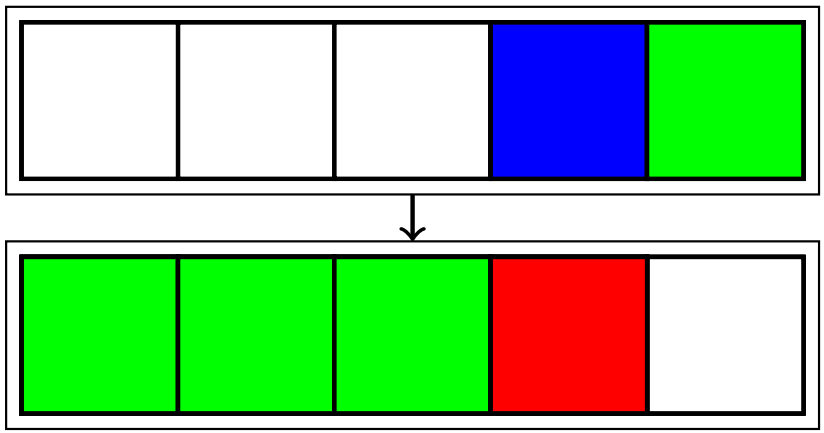
Unbounded Queue: Hybrid



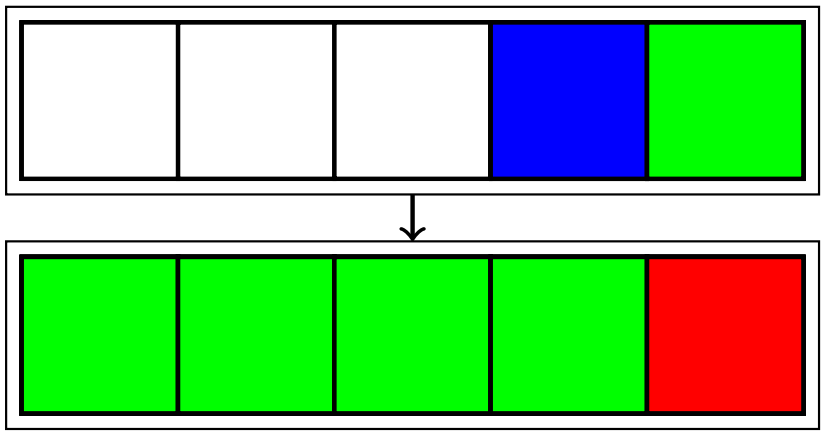
Unbounded Queue: Hybrid



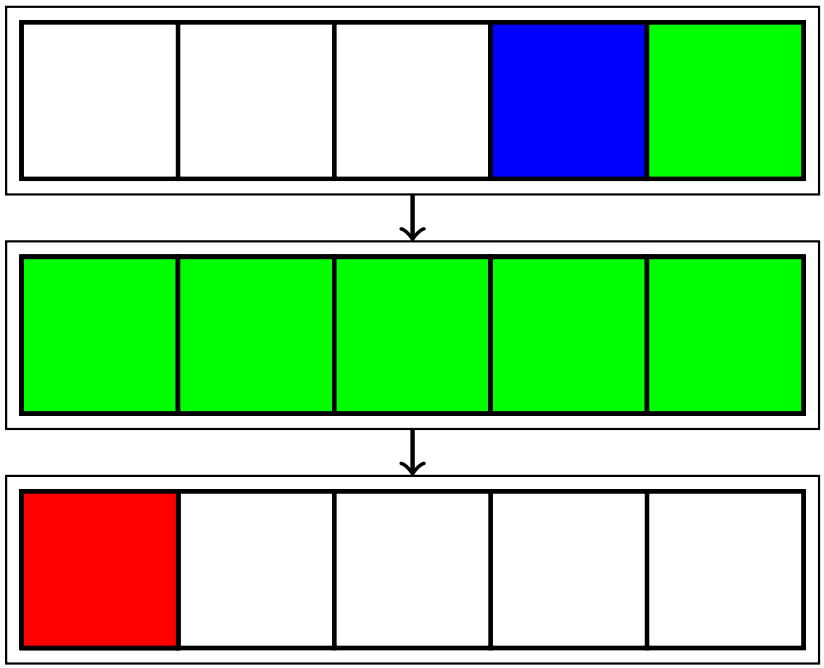
Unbounded Queue: Hybrid



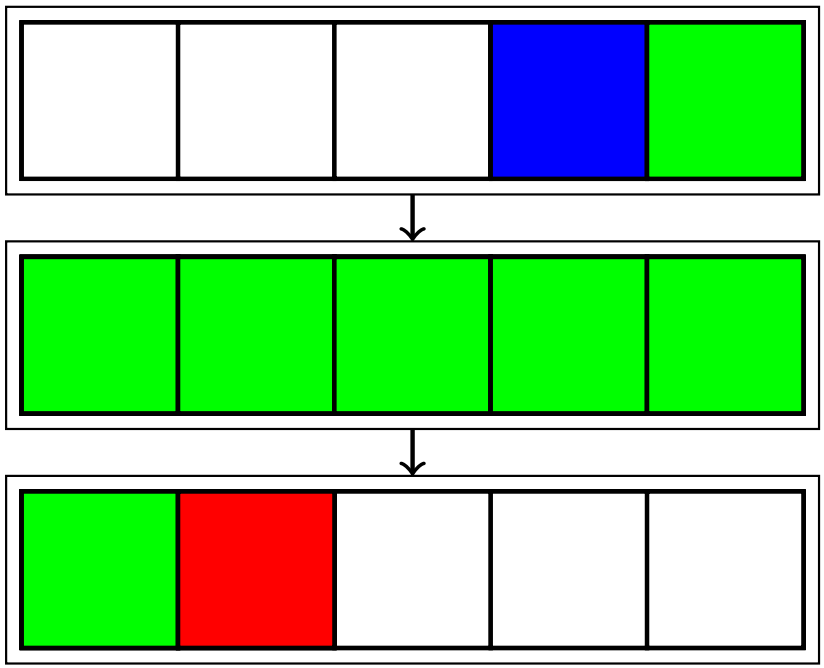
Unbounded Queue: Hybrid



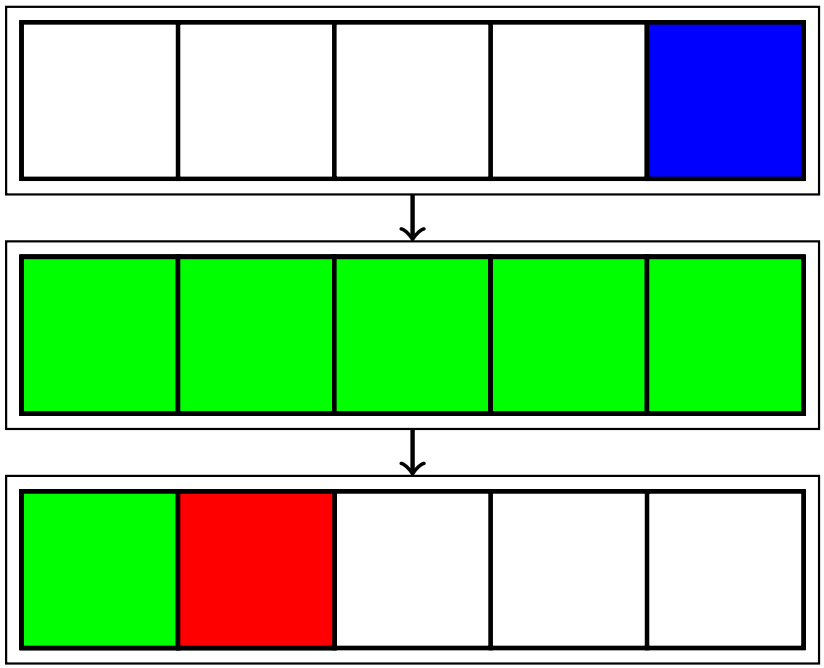
Unbounded Queue: Hybrid



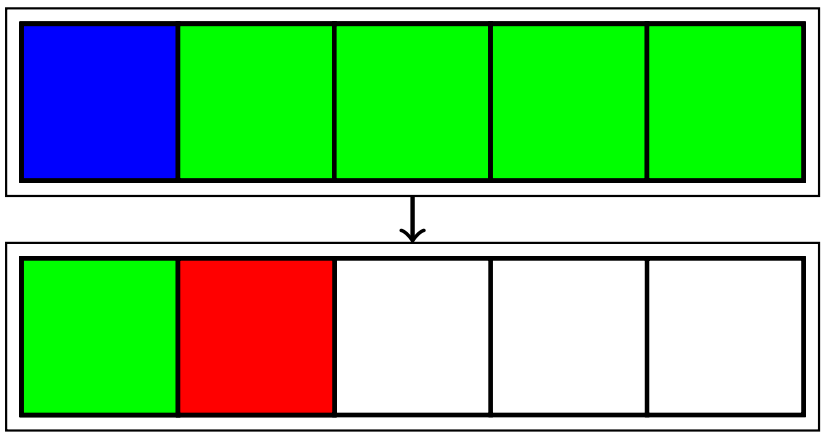
Unbounded Queue: Hybrid



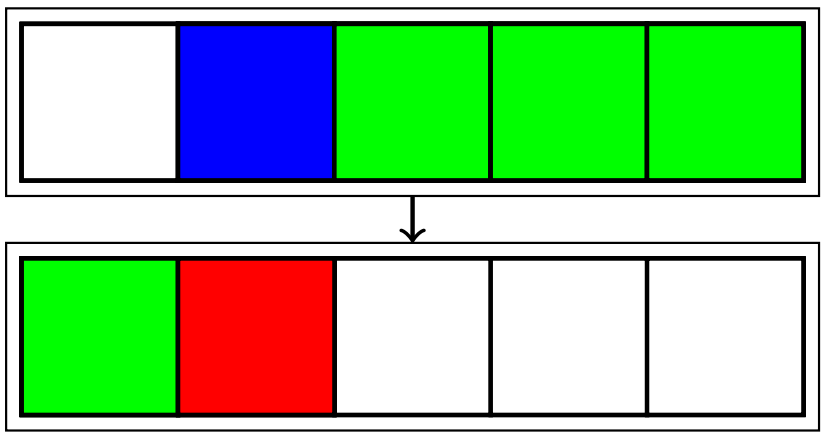
Unbounded Queue: Hybrid



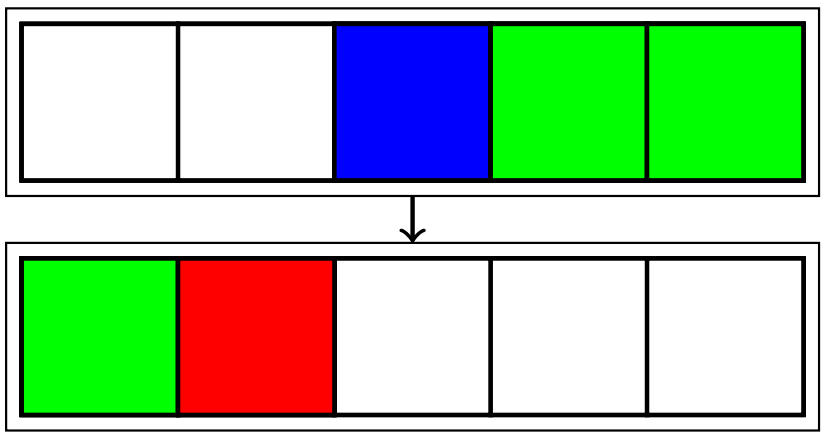
Unbounded Queue: Hybrid



Unbounded Queue: Hybrid



Unbounded Queue: Hybrid



Unbounded Queue: Hybrid

Pros:

- Very good concurrent performance.
- Low memory overhead.

Cons:

- Hard to implement.

Reader/writer: about 36M integers per second.
(**10x** speedup compared to the naïve queue implementation.)

Review

(live)