```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;


        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;


        Node* tail = Tail;
        Node* next = head->next;

        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {

                return data;
            }
        }
    }
}
```

queue

lock-free

GC

37 LOC

```
struct Node {
    data_t data;
    Node* node;
}
```

```
shared:
    Node* Head;
    Node* Tail;
```

```
void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}
```

```
void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}
```

```
data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

lock-free

SMR

37+6 LOC

Non-blocking Queue (Michael&Scott)

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}
```

```
void protect0(Node* ptr) {
    myRec->hp0 = ptr;
}

void protect1(Node* ptr) {
    myRec->hp1 = ptr;
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
    }
}
```

lock-free

HP

42 LOC

no reclamation

Non-blocking Queue (Michael&Scott)

```
struct Node {
    data_t data;
    Node* node;
}

shared:
    Node* Head;
    Node* Tail;

void init() {
    Head = new Node();
    Head->next = null;
    Tail = Head;
}

void enqueue(data_t val) {
    Node* node = new Node();
    node->data = val;
    node->next = null;
    while (true) {
        Node* tail = Tail;
        protect0(tail);
        if (Tail != tail) continue;
        Node* next = tail->next;
        if (Tail != tail) continue;
        if (next == null) {
            if (CAS(tail->next, null, node)) {
                CAS(Tail, tail, node);
            }
        } else {
            CAS(Tail, tail, next);
        }
    }
}

data_t dequeue() {
    while (true) {
        Node* head = Head;
        protect0(head);
        if (Head != head) continue;
        Node* tail = Tail;
        Node* next = head->next;
        protect1(next);
        if (Head != head) continue;
        if (head == tail) {
            if (next == null) return empty_t;
            else CAS(Tail, tail, next);
        } else {
            data = head->data;
            if (CAS(Head, head, next)) {
                retire(head);
                return data;
            }
        }
    }
}
```

queue

lock-free

SMR

37+6 LOC

```
struct Rec {
    Rec* next;
    Node* hp0;
    Node* hp1;
}

shared:
    Rec* HPRecs;

thread-local:
    Rec* myRec;
    List<Node*> retiredList;

void join() {
    myRec = new HPRec();
    while (true) {
        Rec* tmp = HPRecs;
        myRec->next = tmp;
        if (CAS(HPRecs, tmp, myRec)) {
            break;
        }
    }
}

void part() {
    unprotect(0);
    unprotect(1);
}

void protect0(Node* ptr) {
    myRec->hp0 = ptr;
}

void protect1(Node* ptr) {
    myRec->hp1 = ptr;
}

void retire(Node* ptr) {
    retiredList.add(ptr);
    if (*) reclaim();
}

void reclaim() {
    List<Node*> protectedList;
    Rec* tmp = HPRecs;
    while (tmp != null) {
        Node* hp0 = cur->hp0;
        Node* hp1 = cur->hp1;
        protectedList.add(hp0);
        protectedList.add(hp1);
        cur = cur->next;
    }
    for (Node* ptr : retiredList) {
        if (!protectedList.contains(ptr)) {
            retiredList.remove(ptr);
            delete ptr;
    }
}
```

lock-free

HP

no reclamation

42 LOC

Verification Challenges

1. unbounded shared heap
2. unbounded data domain
3. unbounded number of threads
4. fine-grained concurrency

GC