Gradual Verification C0 for Minimum Priority Queue

Definition of min priority queue:

When an element is dequeued from a min priority queue, it will be the one with the smallest priority value among all elements in the queue. The element with the lowest priority value is always at the front of the queue.

Verified functions:

createMinPriQueue(int value)

enqueue(MinPriorityQueue *q, int value)

dequeue(MinPriorityQueue *q)

Commands to run:

```
cd gvc0
sbt
run -x ./minprique_linkedlist.c0
run -s ./minprique_linkedlist.c0

run -x ./createQ_test1_success.c0
run -x ./createQ_test2_success.c0

cd silicon-gv
export Z3_EXE=/usr/bin/z3
sbt
run ../gvc0/minprique_linkedlist.vpr
```

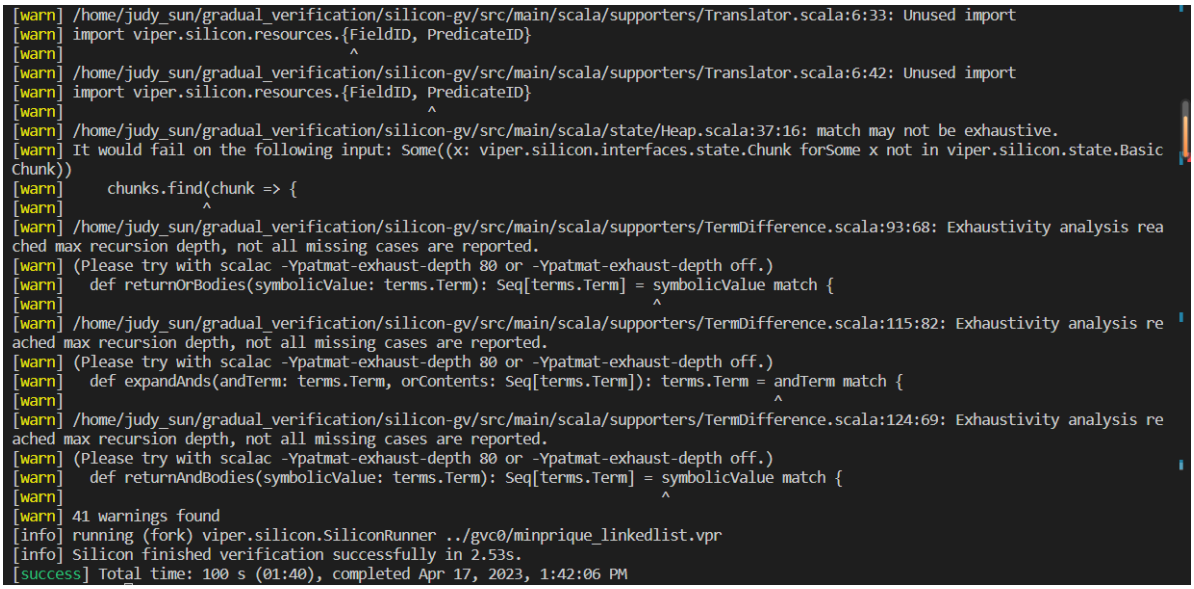
Write the struct of minimum priority queue:

code:

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    bool deleted;
};
typedef struct Node Node;
```

```
struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

int main () {
    return 0;
}
```

terminal output:



Write the predicates

trial 1

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    struct Node *prev;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
```

```
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL || start->val <= start->next->val) &&
            isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start) && isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    acc(start) && isMinPQHelper(start, start->val);
@*/

//-----------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires isMinPQ(q->head);
    //@ ensures isMinPQ(q->head);
{
    Node *newNode = alloc(struct Node);
    newNode->val = value;
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
    } else { // insert after head
        Node *curr = q->head;
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant ? && acc(curr) && acc(curr->next) && isSorted(curr)
&& isSorted(curr->next) && isMinPQHelper(curr, curr->val) && isMinPQHelper(curr-
>next, curr->next->val);
        {
            curr = curr->next;
        }
```

```
        newNode->next = curr->next;
        curr->next = newNode;
    }
    q->size++;
}

int main () {
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -s minprique_linkedlist.c0
[info] running (fork) gvc.Main -s minprique_linkedlist.c0
[info] [*] – Mon Apr 17 19:04:49 EDT 2023
[info] [x] – Errors:
[info] 70:37 - Subject of acc() expression must be a field or dereference
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 2 s, completed Apr 17, 2023, 7:04:50 PM
```

trial 2

```
/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
@*/
```

terminal output:

```
[warn] 41 warnings found
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 5 errors in 4.59s:
[info]   [0] Predicate might not be well-formed. There might be insufficient permission to access start.Node$val. (minprique
_linkedlist.vpr@11.1)
[info]   [1] Predicate might not be well-formed. There might be insufficient permission to access start.Node$next.Node$val.
(minprique_linkedlist.vpr@19.1)
[info]   [2] Contract might not be well-formed. There might be insufficient permission to access $result.MinPriorityQueue$he
ad. (minprique_linkedlist.vpr@25.11)
[info]   [3] Contract might not be well-formed. There might be insufficient permission to access q.MinPriorityQueue$head. (m
inprique_linkedlist.vpr@35.12)
[info]   [4] Method call might fail. There might be insufficient permission to access $result.MinPriorityQueue$head. (minpri
que_linkedlist.vpr@61.3)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 159 s (02:39), completed Apr 18, 2023, 12:12:28 AM
```

trial 3

```
/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL ? true : (acc(start->next->val) &&
            start->val <= start->next->val)) && isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ? true : acc(start->val) && acc(start->next) &&
isMinPQHelper(start, start->val);
@*/
```

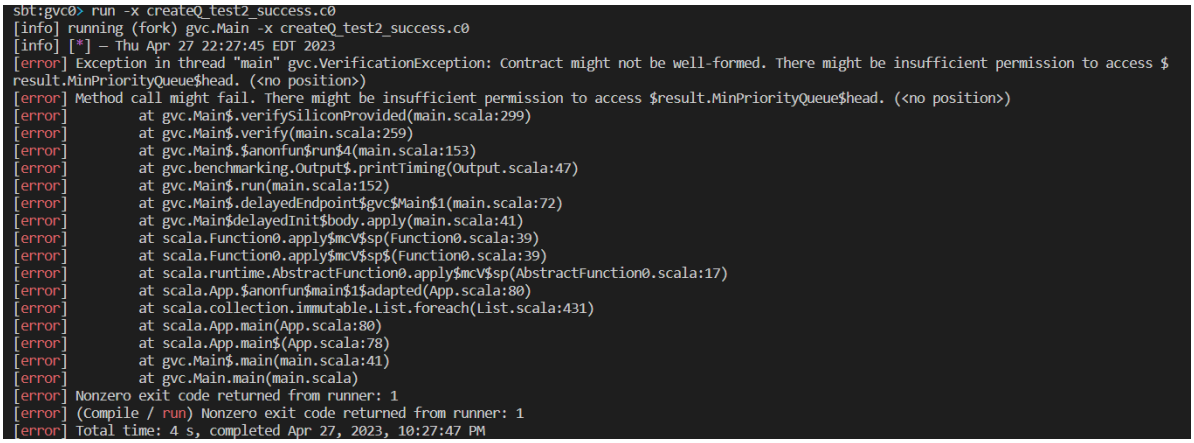
terminal output:

```
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:124:69: Exhaustivity an
alysis reached max recursion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnAndBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] 41 warnings found
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 2 errors in 3.39s:
[info]   [0] Postcondition of createMinPriQueue might not hold. There might be insufficient permission to access isMinPQ($re
sult.MinPriorityQueue$head). (minprique_linkedlist.vpr@25.11)
[info]   [1] Contract might not be well-formed. There might be insufficient permission to access q.MinPriorityQueue$head. (m
inprique_linkedlist.vpr@35.12)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 101 s (01:41), completed Apr 18, 2023, 2:08:21 AM
```

trial 4

```
/*@
predicate isMinPQ(Node *start) =
    start == NULL ? true : (acc(start->val) && acc(start->next) &&
isMinPQHelper(start->next, start->val));
@*/
```

terminal output:



trial 5: success

```
/*@
predicate orderedListSegWithPrev(struct Node *start, struct Node *end, int prev,
int endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      start->val >= prev && orderedListSegWithPrev(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate orderedListSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      orderedListSegWithPrev(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
  orderedListSeg(start, NULL, -1);
@*/

//-----------------------------------------------------------------lemmas
// Lemma:
```

```
void appendLemmaLoopBody(struct Node *a, struct Node *b, struct Node *c, int
aPrev, int cPrev, int bVal, int cVal, bool bDeleted)
  /*@
    requires orderedListSegWithPrev(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && acc(c->deleted) && c->val == cVal
&&
              c->val >= cPrev && orderedListSegWithPrev(c->next, NULL, c->val,
-1)
            )
        ) &&
        ( (b == c) ?
            ( true )
            :
            (
              acc(b->val) && acc(b->next) && acc(b->deleted) && b->val == bVal
&&
              orderedListSegWithPrev(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures orderedListSegWithPrev(a, c, aPrev, cVal) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && acc(c->deleted) && c->val == cVal
&&
              c->val >= cPrev && orderedListSegWithPrev(c->next, NULL, c->val,
-1)
            )
        ) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  } else {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    appendLemmaLoopBody(a->next, b, c, a->val, cPrev, bVal, cVal, b->deleted);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  }
}

void appendLemmaAfterLoopBody(struct Node *a, struct Node *b, struct Node *c,
int aPrev, int bVal, int cVal, bool bDeleted)
  /*@
    requires orderedListSegWithPrev(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && acc(c->deleted) &&
c->val == cVal ) &&
```

```
        ( (b == c) ?
            ( true )
            :
            (
              acc(b->val) && acc(b->next) && acc(b->deleted) && b->val == bVal
&&
              orderedListSegWithPrev(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures orderedListSegWithPrev(a, c, aPrev, cVal) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && acc(c->deleted) &&
c->val == cVal) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  } else {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    appendLemmaAfterLoopBody(a->next, b, c, a->val, bVal, cVal, b->deleted);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  }
}
```

Write the createMinPriQueue() function

trial 1

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
    (start == NULL) ?
        true
```

```
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL ? true : (acc(start->next->val) &&
            start->val <= start->next->val)) && isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ? true : acc(start->val) && acc(start->next) &&
isMinPQHelper(start, start->val);
@*/

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ fold isMinPQ(q->head);
    return q;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -x createQ_test2_success.c0
[info] running (fork) gvc.Main -x createQ_test2_success.c0
[info] [*] – Thu Apr 27 22:24:05 EDT 2023
[error] Exception in thread "main" gvc.VerificationException: Contract might not be well-formed. There might be insufficient permission to access $
result.MinPriorityQueue$head. (<no position>)
[error] Method call might fail. There might be insufficient permission to access $result.MinPriorityQueue$head. (<no position>)
[error]         at gvc.Main$.verifySiliconProvided(main.scala:299)
[error]         at gvc.Main$.verify(main.scala:259)
[error]         at gvc.Main$.$anonfun$run$4(main.scala:153)
[error]         at gvc.benchmarking.Output$.printTiming(Output.scala:47)
[error]         at gvc.Main$.run(main.scala:152)
[error]         at gvc.Main$.delayedEndpoint$gvc$Main$1(main.scala:72)
[error]         at gvc.Main$delayedInit$body.apply(main.scala:41)
[error]         at scala.Function0.apply$mcV$sp(Function0.scala:39)
[error]         at scala.Function0.apply$mcV$sp$(Function0.scala:39)
[error]         at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
[error]         at scala.App.$anonfun$main$1$adapted(App.scala:80)
[error]         at scala.collection.immutable.List.foreach(List.scala:431)
[error]         at scala.App.main(App.scala:80)
[error]         at scala.App.main$(App.scala:78)
[error]         at gvc.Main$.main(main.scala:41)
[error]         at gvc.Main.main(main.scala)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 4 s, completed Apr 27, 2023, 10:24:08 PM
```

trial 2

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL ? true : (acc(start->next->val) &&
            start->val <= start->next->val)) && isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
```

```
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isMinPQHelper(start, start->val)
        );
@*/

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ unfold isMinPQ(q->head);
    //@ fold isMinPQ(q->head);
    return q;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -x createQ_test2_success.c0
[info] running (fork) gvc.Main -x createQ_test2_success.c0
[info] [*] – Thu Apr 27 22:51:43 EDT 2023
[error] Exception in thread "main" gvc.VerificationException: Unfolding isMinPQ(q.MinPriorityQueue$head) might fail. There might be insufficient pe
rmission to access isMinPQ(q.MinPriorityQueue$head). (<no position>)
[error]         at gvc.Main$.verifySiliconProvided(main.scala:299)
[error]         at gvc.Main$.verify(main.scala:259)
[error]         at gvc.Main$.$anonfun$run$4(main.scala:153)
[error]         at gvc.benchmarking.Output$.printTiming(Output.scala:47)
[error]         at gvc.Main$.run(main.scala:152)
[error]         at gvc.Main$.delayedEndpoint$gvc$Main$1(main.scala:72)
[error]         at gvc.Main$delayedInit$body.apply(main.scala:41)
[error]         at scala.Function0.apply$mcV$sp(Function0.scala:39)
[error]         at scala.Function0.apply$mcV$sp$(Function0.scala:39)
[error]         at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
[error]         at scala.App.$anonfun$main$1$adapted(App.scala:80)
[error]         at scala.collection.immutable.List.foreach(List.scala:431)
[error]         at scala.App.main(App.scala:80)
[error]         at scala.App.main$(App.scala:78)
[error]         at gvc.Main$.main(main.scala:41)
[error]         at gvc.Main.main(main.scala)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 4 s, completed Apr 27, 2023, 10:51:46 PM
```

trial 3

```
/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL ? true : (acc(start->next->val) &&
            start->val <= start->next->val)) && isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isMinPQHelper(start->next, start->val)
        );
@*/

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ assert acc(q->head) && isMinPQ(q->head);
    return q;
}
```

terminal output:

```
sbt:gvc0> run -x createQ_test2_success.c0
[info] running (fork) gvc.Main -x createQ_test2_success.c0
[info] [*] – Thu Apr 27 22:31:43 EDT 2023
[error] Exception in thread "main" gvc.VerificationException: Assert might fail. There might be insufficient permission to access isMinPQ(q.MinPrio
rityQueue$head). (<no position>)
[error]         at gvc.Main$.verifySiliconProvided(main.scala:299)
[error]         at gvc.Main$.verify(main.scala:259)
[error]         at gvc.Main$.$anonfun$run$4(main.scala:153)
[error]         at gvc.benchmarking.Output$.printTiming(Output.scala:47)
[error]         at gvc.Main$.run(main.scala:152)
[error]         at gvc.Main$.delayedEndpoint$gvc$Main$1(main.scala:72)
[error]         at gvc.Main$delayedInit$body.apply(main.scala:41)
[error]         at scala.Function0.apply$mcV$sp(Function0.scala:39)
[error]         at scala.Function0.apply$mcV$sp$(Function0.scala:39)
[error]         at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
[error]         at scala.App.$anonfun$main$1$adapted(App.scala:80)
[error]         at scala.collection.immutable.List.foreach(List.scala:431)
[error]         at scala.App.main(App.scala:80)
[error]         at scala.App.main$(App.scala:78)
[error]         at gvc.Main$.main(main.scala:41)
[error]         at gvc.Main.main(main.scala)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 4 s, completed Apr 27, 2023, 10:31:46 PM
```

trial 4: success with question mark:

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
// check if the linked list is in ascending order (recursion)
// 1. if the current node is NULL (end of list), already sorted, return true
// 2. check the value of the current node <= value of next node, recursion

predicate isSorted(Node *start) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            (start->next == NULL ? true : (acc(start->next->val) &&
            start->val <= start->next->val)) && isSorted(start->next)
        );
@*/

/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
        );
```

```
@*/

/*@
predicate isMinPQ(Node *start) =
  start == NULL ? true :
  (acc(start->val) && acc(start->next) && isMinPQHelper(start, start->val));
@*/

//---------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures ? && acc((\result)->size) && isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ assert acc(q->head) && acc(q->size);
    //@ fold isMinPQ(q->head);
    return q;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -x minprique_linkedlist.c0
[info] running (fork) gvc.Main -x minprique_linkedlist.c0
[info] [*] – Thu Apr 27 21:48:51 EDT 2023
[info] 0
[success] Total time: 4 s, completed Apr 27, 2023, 9:48:54 PM
```

trial 5: complete success

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;
```

```
/*@
predicate sortedSegHelper(struct Node *start, struct Node *end, int prev, int
endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) &&
      start->val >= prev && sortedSegHelper(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate sortedSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) &&
      sortedSegHelper(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            sortedSeg(start, NULL, -1)
        );
@*/

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ fold isMinPQ(q->head);
    return q;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -x createQ_test2_success.c0
[info] running (fork) gvc.Main -x createQ_test2_success.c0
[info] [*] — Thu Apr 27 22:59:36 EDT 2023
[info] 0
[success] Total time: 4 s, completed Apr 27, 2023, 10:59:39 PM
```

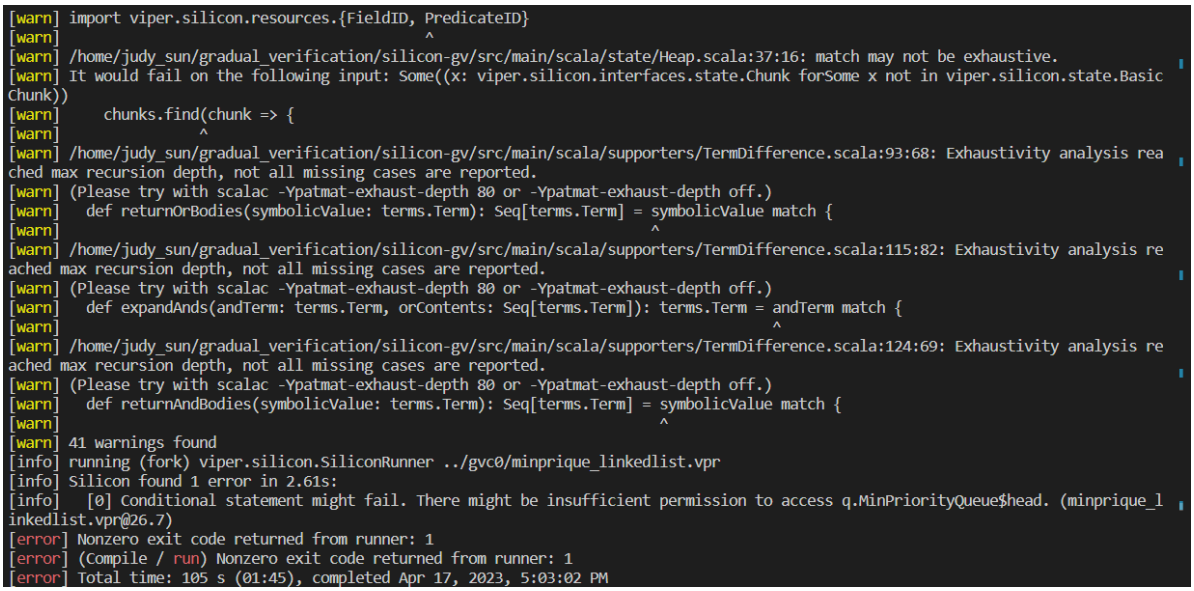
Write enqueue() function

trial 1:

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    struct Node *prev;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

MinPriorityQueue* createMinPriQueue()
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
{
    Node *newNode = alloc(struct Node);
    newNode->val = value;
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
    } else { // insert after head
        Node *curr = q->head;
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant ? && acc(curr->next) && acc(curr->next->val);
        {
            curr = curr->next;
        }
        newNode->next = curr->next;
        curr->next = newNode;
    }
    q->size++;
}
```

```
int main () {
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:



trial 2

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    struct Node *prev;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;


/*@
predicate isMinPQHelper(Node *start, int minVal) =
    (start == NULL) ?
        true
    :
        (
            acc(start->next) && isSorted(start) && start->val >= minVal &&
            isMinPQHelper(start->next, start->val)
```

```
        );
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ? true : acc(start->next) && isMinPQHelper(start, start->val);
@*/

//-----------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires isMinPQ(q->head);
    //@ ensures isMinPQ(q->head);
{
    Node *newNode = alloc(struct Node);
    newNode->val = value;
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
    } else { // insert after head
        Node *curr = q->head;
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant isSorted(q->head) && isMinPQHelper(q->head, curr-
>val) && acc(curr->next);
        {
            curr = curr->next;
        }
        newNode->next = curr->next;
        curr->next = newNode;
    }
    q->size++;
}

int main () {
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:

```
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def expandAnds(andTerm: terms.Term, orContents: Seq[terms.Term]): terms.Term = andTerm match {
[warn]                                                                                 ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:124:69: Exhaustivity analysis reached max recursion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnAndBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] 41 warnings found
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 6 errors in 3.33s:
[info]   [0] Predicate might not be well-formed. There might be insufficient permission to access start.Node$val. (minprique_linkedlist.vpr@11.1)
[info]   [1] Predicate might not be well-formed. There might be insufficient permission to access start.Node$val. (minprique_linkedlist.vpr@15.1)
[info]   [2] Predicate might not be well-formed. There might be insufficient permission to access start.Node$next.Node$val. (minprique_linkedlist.vpr@19.1)
[info]   [3] Contract might not be well-formed. There might be insufficient permission to access $result.MinPriorityQueue$head. (minprique_linkedlist.vpr@25.11)
[info]   [4] Contract might not be well-formed. There might be insufficient permission to access q.MinPriorityQueue$head. (minprique_linkedlist.vpr@35.12)
[info]   [5] Method call might fail. There might be insufficient permission to access $result.MinPriorityQueue$head. (minprique_linkedlist.vpr@61.3)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 108 s (01:48), completed Apr 17, 2023, 8:26:27 PM
```

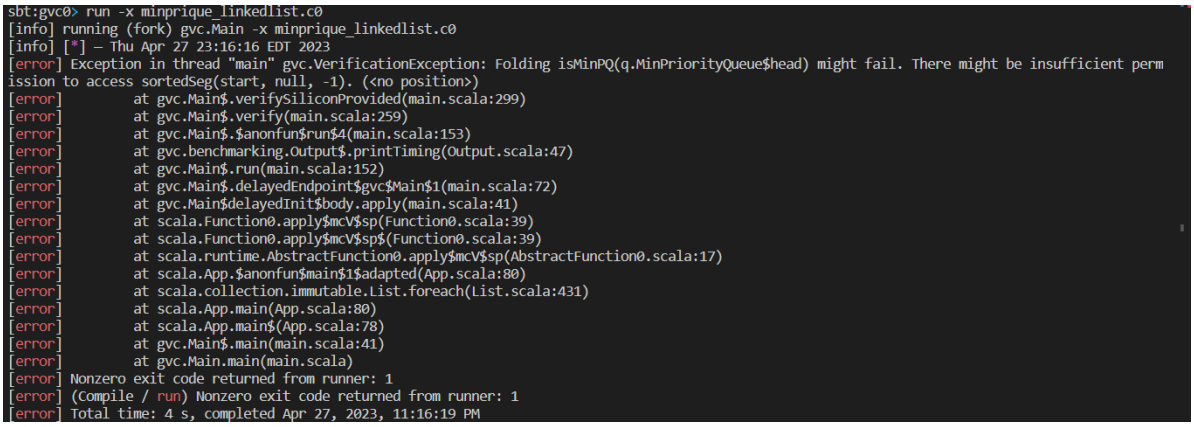
trial 3

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
predicate sortedSegHelper(struct Node *start, struct Node *end, int prev, int
endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) &&
      start->val >= prev && sortedSegHelper(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate sortedSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) &&
      sortedSegHelper(start->next, end, start->val, endVal)
    ) ;
@*/
```

```
/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            sortedSeg(start, NULL, -1)
        );
@*/

//-------------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ fold isMinPQ(q->head);
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    Node *newNode = alloc(struct Node);
    newNode->val = value;

    //@ unfold isMinPQ(q->head);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
    } else { // insert after head
        Node *curr = q->head;

        //@ unfold sortedSeg(curr, NULL, -1);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next);
        //@ loop_invariant sortedSeg(q->head, curr, q->head->val) && q->head-
>val <= value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && curr->next->val >= curr->val && sortedSegHelper(curr-
>next->next, NULL, curr->next->val, -1);
        {
            //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
            curr = curr->next;
            //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);
        }
        newNode->next = curr->next;
```

```
        curr->next = newNode;
        //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);
        //@ fold sortedSeg(curr, NULL, -1);
    }
    q->size++;
    //@ fold isMinPQ(q->head);
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:



corresponding vpr file:

```
field MinPriorityQueue$head: Ref

field MinPriorityQueue$size: Int

field Node$next: Ref

field Node$val: Int

predicate isMinPQ(start: Ref) {
  (start == null ? true : acc(start.Node$val, write) && acc(start.Node$next,
write) && acc(sortedSeg(start, null, -1), write))
}

predicate sortedSeg(start: Ref, end: Ref, endVal: Int) {
  (start == end ? true : acc(start.Node$val, write) && acc(start.Node$next,
write) && acc(sortedSegHelper(start.Node$next, end, start.Node$val, endVal),
write))
}
```

```
predicate sortedSegHelper(start: Ref, end: Ref, prev: Int, endVal: Int) {
  (start == end ? (end == null ? true : endVal >= prev) : acc(start.Node$val,
write) && acc(start.Node$next, write) && start.Node$val >= prev &&
acc(sortedSegHelper(start.Node$next, end, start.Node$val, endVal), write))
}

method createMinPriQueue() returns ($result: Ref)
  requires true
  ensures acc($result.MinPriorityQueue$size, write) &&
acc($result.MinPriorityQueue$head, write) &&
acc(isMinPQ($result.MinPriorityQueue$head), write)
{
  var q: Ref
  q := new(MinPriorityQueue$head, MinPriorityQueue$size)
  q.MinPriorityQueue$head := null
  q.MinPriorityQueue$size := 0
  fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  $result := q
}

method enqueue(q: Ref, value: Int)
  requires acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
  ensures acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
{
  var newNode: Ref
  var curr: Ref
  newNode := new(Node$val, Node$next)
  newNode.Node$val := value
  unfold acc(isMinPQ(q.MinPriorityQueue$head), write)
  if (q.MinPriorityQueue$head == null || value <=
q.MinPriorityQueue$head.Node$val) {
    newNode.Node$next := q.MinPriorityQueue$head
    q.MinPriorityQueue$head := newNode
  } else {
    curr := q.MinPriorityQueue$head
    unfold acc(sortedSeg(curr, null, -1), write)
    while (curr.Node$next != null && curr.Node$next.Node$val < value)
      invariant acc(curr.Node$val, write) && acc(curr.Node$next, write) &&
(acc(sortedSeg(q.MinPriorityQueue$head, curr, q.MinPriorityQueue$head.Node$val),
write) && q.MinPriorityQueue$head.Node$val <= value) && (curr.Node$next == null
? true : acc(curr.Node$next.Node$next, write) && acc(curr.Node$next.Node$val,
write) && curr.Node$next.Node$val >= curr.Node$val &&
acc(sortedSegHelper(curr.Node$next.Node$next, null, curr.Node$next.Node$val,
-1), write))
    {
      unfold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1),
write)
      curr := curr.Node$next
      fold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1), write)
    }
    newNode.Node$next := curr.Node$next
    curr.Node$next := newNode
    fold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1), write)
```

```
    fold acc(sortedSeg(curr, null, -1), write)
  }
  q.MinPriorityQueue$size := q.MinPriorityQueue$size + 1
  fold acc(isMinPQ(q.MinPriorityQueue$head), write)
}

method main() returns ($result: Int)
  requires true
  ensures true
{
  var q: Ref
  q := createMinPriQueue()
  enqueue(q, 10)
  enqueue(q, 5)
  enqueue(q, 20)
  $result := 0
}

method printint(value: Int)
{

}

method printchar(value: Int)
{

}

method printbool(value: Bool)
{

}

method flush()
{

}

method mod(r: Int, l: Int) returns ($result: Int)
{
  $result := 0
}

method rand(prev: Int) returns ($result: Int)
{
  $result := 0
}

method readStress() returns ($result: Int)
{
  $result := 0
}
```

```
[warn] import viper.silicon.resources.{FieldID, PredicateID}
[warn]                                      ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/state/Heap.scala:37:16: match may not be exhaustive.
[warn] It would fail on the following input: Some((x: viper.silicon.interfaces.state.Chunk forSome x not in viper.silicon.state.BasicChunk))
[warn]     chunks.find(chunk => {
[warn]                ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:93:68: Exhaustivity analysis reached max recur
sion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnOrBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:115:82: Exhaustivity analysis reached max recu
rsion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def expandAnds(andTerm: terms.Term, orContents: Seq[terms.Term]): terms.Term = andTerm match {
[warn]                                                                                 ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:124:69: Exhaustivity analysis reached max recu
rsion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnAndBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] 41 warnings found
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 1 error in 3.20s:
[info]   [0] Folding isMinPQ(q.MinPriorityQueue$head) might fail. There might be insufficient permission to access sortedSeg(start, null, -1). (min
prique_linkedlist.vpr@61.3)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 103 s (01:43), completed Apr 27, 2023, 11:20:18 PM
```

trial 4

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
predicate sortedSegHelper(struct Node *start, struct Node *end, int prev, int
endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) &&
      start->val >= prev && sortedSegHelper(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate sortedSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) &&
      sortedSegHelper(start->next, end, start->val, endVal)
    ) ;
@*/
```

```
/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            sortedSeg(start, NULL, -1)
        );
@*/

//-------------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ fold isMinPQ(q->head);
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    Node *newNode = alloc(struct Node);
    newNode->val = value;
    //@ unfold isMinPQ(q->head);
    //@ unfold sortedSeg(q->head, NULL, -1);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
        //@ fold sortedSegHelper(q->head->next, NULL, q->head->val, -1);
        //@ fold sortedSeg(q->head, NULL, -1);
    } else { // insert after head
        Node *curr = q->head;

        //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
        //@ fold sortedSeg(q->head, curr, curr->val);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next);
        //@ loop_invariant sortedSeg(q->head, curr, curr->val) && curr->val <=
value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && curr->next->val >= curr->val && sortedSegHelper(curr-
>next->next, NULL, curr->next->val, -1);
        {
            //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
            curr = curr->next;
```

```
            //@ fold sortedSeg(q->head, curr, curr->val);
            //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
        }
        newNode->next = curr->next;
        curr->next = newNode;
        //@ fold sortedSegHelper(curr->next->next, NULL, curr->next->val, -1);
        //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);
        //@ unfold sortedSeg(q->head, curr, curr->val);
    }
    q->size++;
    //@ fold sortedSeg(q->head, NULL, -1);
    //@ fold isMinPQ(q->head);
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:

```
sbt:gvc0> run -x minprique_linkedlist.c0
[info] running (fork) gvc.Main -x minprique_linkedlist.c0
[info] [*] – Fri Apr 28 13:18:26 EDT 2023
[error] Exception in thread "main" gvc.VerificationException: Unfolding sortedSeg(q.MinPriorityQueue$head, null, -1) might fail. There might be ins
ufficient permission to access sortedSeg(q.MinPriorityQueue$head, null, -1). (<no position>)
[error]         at gvc.Main$.verifySiliconProvided(main.scala:299)
[error]         at gvc.Main$.verify(main.scala:259)
[error]         at gvc.Main$.$anonfun$run$4(main.scala:153)
[error]         at gvc.benchmarking.Output$.printTiming(Output.scala:47)
[error]         at gvc.Main$.run(main.scala:152)
[error]         at gvc.Main$.delayedEndpoint$gvc$Main$1(main.scala:72)
[error]         at gvc.Main$delayedInit$body.apply(main.scala:41)
[error]         at scala.Function0.apply$mcV$sp(Function0.scala:39)
[error]         at scala.Function0.apply$mcV$sp$(Function0.scala:39)
[error]         at scala.runtime.AbstractFunction0.apply$mcV$sp(AbstractFunction0.scala:17)
[error]         at scala.App.$anonfun$main$1$adapted(App.scala:80)
[error]         at scala.collection.immutable.List.foreach(List.scala:431)
[error]         at scala.App.main(App.scala:80)
[error]         at scala.App.main$(App.scala:78)
[error]         at gvc.Main$.main(main.scala:41)
[error]         at gvc.Main.main(main.scala)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 4 s, completed Apr 28, 2023, 1:18:29 PM
```

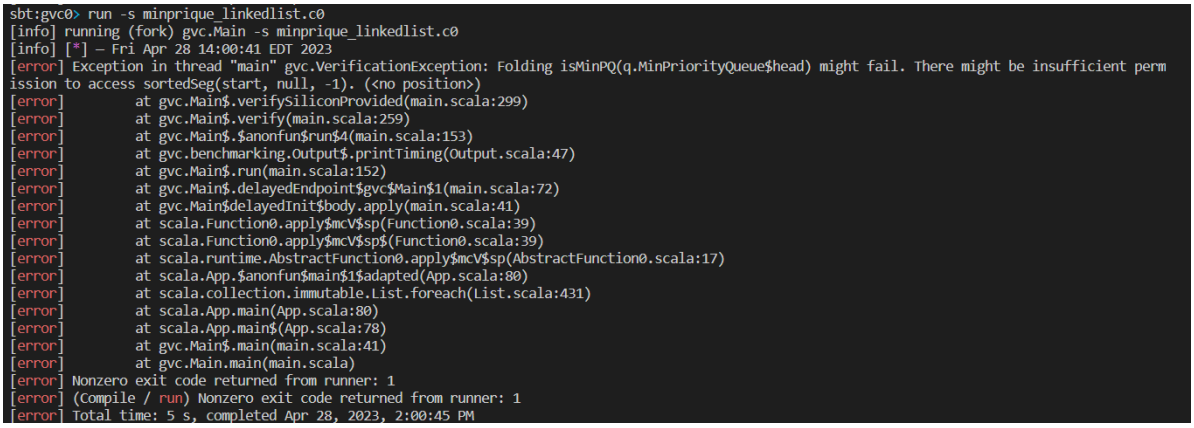
trial 5

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;
```

```
/*@
predicate sortedSegHelper(struct Node *start, struct Node *end, int prev, int
endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) &&
      start->val >= prev && sortedSegHelper(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate sortedSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) &&
      sortedSegHelper(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
    start == NULL ?
        true
    :
        (
            acc(start->val) && acc(start->next) &&
            sortedSeg(start, NULL, -1)
        );
@*/

//-------------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue()
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = NULL;
    q->size = 0;
    //@ fold isMinPQ(q->head);
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
```

```
    Node *newNode = alloc(struct Node);
    newNode->val = value;
    //@ unfold isMinPQ(q->head);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        newNode->next = q->head;
        q->head = newNode;
    } else { // insert after head
        Node *curr = q->head;
        //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
        //@ fold sortedSeg(q->head, curr, curr->val);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next);
        //@ loop_invariant sortedSeg(q->head, curr, curr->val) && curr->val <=
value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && curr->next->val >= curr->val && sortedSegHelper(curr-
>next->next, NULL, curr->next->val, -1);
        {
            curr = curr->next;
            //@ unfold sortedSeg(q->head, curr, curr->val);
            //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);
        }
        newNode->next = curr->next;
        curr->next = newNode;
        //@ fold sortedSegHelper(newNode->next, NULL, newNode->val, -1);
        //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);

        //@ fold sortedSeg(q->head, NULL, -1);
    }
    q->size++;
    //@ fold isMinPQ(q->head);
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue();
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:



corresponding vpr file:

```
field MinPriorityQueue$head: Ref

field MinPriorityQueue$size: Int

field Node$next: Ref

field Node$val: Int

predicate isMinPQ(start: Ref) {
  (start == null ? true : acc(start.Node$val, write) && acc(start.Node$next,
write) && acc(sortedSeg(start, null, -1), write))
}

predicate sortedSeg(start: Ref, end: Ref, endVal: Int) {
  (start == end ? true : acc(start.Node$val, write) && acc(start.Node$next,
write) && acc(sortedSegHelper(start.Node$next, end, start.Node$val, endVal),
write))
}

predicate sortedSegHelper(start: Ref, end: Ref, prev: Int, endVal: Int) {
  (start == end ? (end == null ? true : endVal >= prev) : acc(start.Node$val,
write) && acc(start.Node$next, write) && start.Node$val >= prev &&
acc(sortedSegHelper(start.Node$next, end, start.Node$val, endVal), write))
}

method createMinPriQueue() returns ($result: Ref)
  requires true
  ensures acc($result.MinPriorityQueue$size, write) &&
acc($result.MinPriorityQueue$head, write) &&
acc(isMinPQ($result.MinPriorityQueue$head), write)
{
  var q: Ref
  q := new(MinPriorityQueue$head, MinPriorityQueue$size)
  q.MinPriorityQueue$head := null
  q.MinPriorityQueue$size := 0
  fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  $result := q
}

method enqueue(q: Ref, value: Int)
```

```
  requires acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
  ensures acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
{
  var newNode: Ref
  var curr: Ref
  newNode := new(Node$val, Node$next)
  newNode.Node$val := value
  unfold acc(isMinPQ(q.MinPriorityQueue$head), write)
  if (q.MinPriorityQueue$head == null || value <=
q.MinPriorityQueue$head.Node$val) {
    newNode.Node$next := q.MinPriorityQueue$head
    q.MinPriorityQueue$head := newNode
  } else {
    curr := q.MinPriorityQueue$head
    unfold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1), write)
    fold acc(sortedSeg(q.MinPriorityQueue$head, curr, curr.Node$val), write)
    while (curr.Node$next != null && curr.Node$next.Node$val < value)
      invariant acc(curr.Node$val, write) && acc(curr.Node$next, write) &&
(acc(sortedSeg(q.MinPriorityQueue$head, curr, curr.Node$val), write) &&
curr.Node$val <= value) && (curr.Node$next == null ? true :
acc(curr.Node$next.Node$next, write) && acc(curr.Node$next.Node$val, write) &&
curr.Node$next.Node$val >= curr.Node$val &&
acc(sortedSegHelper(curr.Node$next.Node$next, null, curr.Node$next.Node$val,
-1), write))
    {
      curr := curr.Node$next
      unfold acc(sortedSeg(q.MinPriorityQueue$head, curr, curr.Node$val), write)
      fold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1), write)
    }
    newNode.Node$next := curr.Node$next
    curr.Node$next := newNode
    fold acc(sortedSegHelper(newNode.Node$next, null, newNode.Node$val, -1),
write)
    fold acc(sortedSegHelper(curr.Node$next, null, curr.Node$val, -1), write)
    fold acc(sortedSeg(q.MinPriorityQueue$head, null, -1), write)
  }
  q.MinPriorityQueue$size := q.MinPriorityQueue$size + 1
  fold acc(isMinPQ(q.MinPriorityQueue$head), write)
}

method main() returns ($result: Int)
  requires true
  ensures true
{
  var q: Ref
  q := createMinPriQueue()
  enqueue(q, 10)
  enqueue(q, 5)
  enqueue(q, 20)
  $result := 0
}
```

```
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnOrBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:115:82: Exhaustivity analysis reached max recu
rsion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def expandAnds(andTerm: terms.Term, orContents: Seq[terms.Term]): terms.Term = andTerm match {
[warn]                                                                               ^
[warn] /home/judy_sun/gradual_verification/silicon-gv/src/main/scala/supporters/TermDifference.scala:124:69: Exhaustivity analysis reached max recu
rsion depth, not all missing cases are reported.
[warn] (Please try with scalac -Ypatmat-exhaust-depth 80 or -Ypatmat-exhaust-depth off.)
[warn]   def returnAndBodies(symbolicValue: terms.Term): Seq[terms.Term] = symbolicValue match {
[warn]                                                                    ^
[warn] 41 warnings found
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 1 error in 3.62s:
[info]   [0] Folding isMinPQ(q.MinPriorityQueue$head) might fail. There might be insufficient permission to access sortedSeg(start, null, -1). (min
prique_linkedlist.vpr@63.3)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 97 s (01:37), completed Apr 28, 2023, 2:03:57 PM
```

trial 6: success

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
predicate sortedSegHelper(struct Node *start, struct Node *end, int prev, int
endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) &&
      start->val >= prev && sortedSegHelper(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate sortedSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) &&
      sortedSegHelper(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
  sortedSeg(start, NULL, -1);
```

```
@*/

//-------------------------------------------------------------------------lemmas
// Lemma:
void appendLemmaLoopBody(struct Node *a, struct Node *b, struct Node *c, int
aPrev, int cPrev, int bVal, int cVal)
  /*@
    requires sortedSegHelper(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && c->val == cVal &&
              c->val >= cPrev && sortedSegHelper(c->next, NULL, c->val, -1)
            )
        ) &&
        ( (b == c) ?
            ( true )
            :
            (
              acc(b->val) && acc(b->next) && b->val == bVal &&
              sortedSegHelper(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures sortedSegHelper(a, c, aPrev, cVal) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && c->val == cVal &&
              c->val >= cPrev && sortedSegHelper(c->next, NULL, c->val, -1)
            )
        ) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
    //@ unfold sortedSegHelper(a, b, aPrev, bVal);
    //@ fold sortedSegHelper(a, c, aPrev, cVal);
  } else {
    //@ unfold sortedSegHelper(a, b, aPrev, bVal);
    appendLemmaLoopBody(a->next, b, c, a->val, cPrev, bVal, cVal);
    //@ fold sortedSegHelper(a, c, aPrev, cVal);
  }
}

void appendLemmaAfterLoopBody(struct Node *a, struct Node *b, struct Node *c,
int aPrev, int bVal, int cVal)
  /*@
    requires sortedSegHelper(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && c->val == cVal )
&&
        ( (b == c) ?
```

```
            ( true )
            :
            (
              acc(b->val) && acc(b->next) && b->val == bVal &&
              sortedSegHelper(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures sortedSegHelper(a, c, aPrev, cVal) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && c->val == cVal ) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
    //@ unfold sortedSegHelper(a, b, aPrev, bVal);
    //@ fold sortedSegHelper(a, c, aPrev, cVal);
  } else {
    //@ unfold sortedSegHelper(a, b, aPrev, bVal);
    appendLemmaAfterLoopBody(a->next, b, c, a->val, bVal, cVal);
    //@ fold sortedSegHelper(a, c, aPrev, cVal);
  }
}

//-------------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue(int value)
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = alloc(struct Node);
    q->head->val = value;
    q->head->next = NULL;
    q->size = 1;
    //@ fold sortedSegHelper(q->head->next, NULL, q->head->val, -1);
    //@ fold sortedSeg(q->head, NULL, -1);
    //@ fold isMinPQ(q->head);
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold sortedSeg(q->head, NULL, -1);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = q->head;
        q->head = newNode;
        //@ fold sortedSegHelper(newNode->next, NULL, newNode->val, -1);
```

```
        //@ fold sortedSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    } else { // insert after head
        Node *curr = q->head;
        //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
        //@ fold sortedSeg(q->head, curr, curr->val);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next);
        //@ loop_invariant acc(q->head);
        //@ loop_invariant sortedSeg(q->head, curr, curr->val) && curr->val <=
value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && curr->next->val >= curr->val && sortedSegHelper(curr-
>next->next, NULL, curr->next->val, -1);
        {
            Node *prev = curr;
            curr = prev->next;
            //@ unfold sortedSeg(q->head, prev, prev->val);
            //@ fold sortedSegHelper(prev->next, curr, prev->val, curr->val);

            if (q->head == prev) {
            } else {
              appendLemmaLoopBody(q->head->next, prev, curr, q->head->val, prev-
>val, prev->val, curr->val);
            }

            //@ fold sortedSeg(q->head, curr, curr->val);
            //@ unfold sortedSegHelper(curr->next, NULL, curr->val, -1);
        }
        struct Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = curr->next;
        curr->next = newNode;
        //@ fold sortedSegHelper(newNode->next, NULL, newNode->val, -1);
        //@ fold sortedSegHelper(curr->next, NULL, curr->val, -1);

        //@ unfold sortedSeg(q->head, curr, curr->val);
        if (q->head == curr) {
        } else {
          appendLemmaAfterLoopBody(q->head->next, curr, NULL, q->head->val,
curr->val, -1);
        }

        //@ fold sortedSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    }
    q->size++;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue(1);
    enqueue(q, 10);
```

```
    enqueue(q, 5);
    enqueue(q, 20);
    return 0;
}
```

terminal output:

```
sbt:Silicon> run ../gvc0/minprique_linkedlist.vpr
[info] compiling 1 Scala source to /home/judy_sun/gradual_verification/silicon-gv/target/scala-2.12/classes ...
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
       [info] Silicon finished verification successfully in 21.32s.
[success] Total time: 24 s, completed May 15, 2023, 11:16:47 AM
sbt:Silicon>
JLine: do you wish to see all 632 possibilities (632 lines)?
```

```
sbt:gvc0> run -x minprique_linkedlist.c0
[info] compiling 1 Scala source to /home/judy_sun/gradual_verification/gvc0/silicon/target/scala-2.12/classes ...
[info] running (fork) gvc.Main -x minprique_linkedlist.c0
[info] [*] – Mon May 15 11:18:21 EDT 2023
[info] 0
[success] Total time: 24 s, completed May 15, 2023, 11:18:42 AM
```

Writing dequeue() function

trial 1: directly delete the node

```
Node* dequeue(MinPriorityQueue *q)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold orderedListSeg(q->head, NULL, -1);
    if (q->head == NULL) {
        // Queue is empty, nothing to dequeue
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
        return NULL;
    } else {
        //@ unfold orderedListSegWithPrev(q->head->next, NULL, q->head->val,
-1);
        Node *temp = q->head;
        q->head = q->head->next;
        //@ assert orderedListSegWithPrev(q->head, NULL, q->head->val, -1);
        //@ fold orderedListSegWithPrev(q->head, NULL, q->head->val, -1);
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
        //@ fold released(temp);
        q->size--;
        return temp;
    }
}
```

terminal output:

```
[error] Exception in thread "main" gvc.VerificationException: Postcondition of dequeue might not hold. There might be insufficient
permission to access released($result). (<no position>)
```

trial 2: success by changing struct, add Boolean deleted

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    bool deleted;
};
typedef struct Node Node;

struct MinPriorityQueue {
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
predicate orderedListSegWithPrev(struct Node *start, struct Node *end, int prev,
int endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      start->val >= prev && orderedListSegWithPrev(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate orderedListSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      orderedListSegWithPrev(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
  orderedListSeg(start, NULL, -1);
@*/

void dequeue(MinPriorityQueue *q)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold orderedListSeg(q->head, NULL, -1);
    if (q->head == NULL) {
        // Queue is empty, nothing to dequeue
        //@ fold orderedListSeg(q->head, NULL, -1);
```

```
        //@ fold isMinPQ(q->head);
        return;
    } else {
        //@ unfold orderedListSegWithPrev(q->head, NULL, q->head->val, -1);
        q->head->deleted = true; // Mark node as deleted
        //@ fold orderedListSegWithPrev(q->head, NULL, q->head->val, -1);
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    }
    q->size--;
}
```

terminal output:

```
sbt:Silicon> run ../gvc0/minprique_linkedlist.vpr
[info] compiling 1 Scala source to /home/judy_sun/gradual_verification/silicon-gv/target/scala-2.12/classes ...
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon found 2 errors in 8.02s:
[info]   [0] Unfolding orderedListSegWithPrev(q.MinPriorityQueue$head, null, q.MinPriorityQueue$head.Node$val, -1) might f
ail. There might be insufficient permission to access orderedListSegWithPrev(q.MinPriorityQueue$head, null, q.MinPriorityQ
ueue$head.Node$val, -1). (minprique_linkedlist.vpr@83.5)
[info]   [1] Folding orderedListSeg(q.MinPriorityQueue$head, null, -1) might fail. There might be insufficient permission
to access start.Node$deleted. (minprique_linkedlist.vpr@141.5)
[error] Nonzero exit code returned from runner: 1
[error] (Compile / run) Nonzero exit code returned from runner: 1
[error] Total time: 11 s, completed May 17, 2023, 3:55:49 PM
```

trial 3: success by revising enqueue() function to add access to deleted Boolean

```
void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold orderedListSeg(q->head, NULL, -1);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = q->head;
        q->head = newNode;
        //@ fold orderedListSegWithPrev(newNode->next, NULL, newNode->val, -1);
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    } else { // insert after head
        Node *curr = q->head;
        //@ unfold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);
        //@ fold orderedListSeg(q->head, curr, curr->val);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next) && acc(curr-
>deleted);
        //@ loop_invariant acc(q->head);
        //@ loop_invariant orderedListSeg(q->head, curr, curr->val) && curr->val
<= value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && acc(curr->next->deleted) && curr->next->val >= curr-
>val && orderedListSegWithPrev(curr->next->next, NULL, curr->next->val, -1);
        {
            Node *prev = curr;
            curr = prev->next;
            //@ unfold orderedListSeg(q->head, prev, prev->val);
```

```
            //@ fold orderedListSegWithPrev(prev->next, curr, prev->val, curr->val);

            if (q->head == prev) {
            } else {
              appendLemmaLoopBody(q->head->next, prev, curr, q->head->val, prev->val, prev->val, curr->val, prev->deleted);
            }

            //@ fold orderedListSeg(q->head, curr, curr->val);
            //@ unfold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);
        }
        struct Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = curr->next;
        newNode->deleted = false;
        curr->next = newNode;
        //@ fold orderedListSegWithPrev(newNode->next, NULL, newNode->val, -1);
        //@ fold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);

        //@ unfold orderedListSeg(q->head, curr, curr->val);
        if (q->head == curr) {
        } else {
          appendLemmaAfterLoopBody(q->head->next, curr, NULL, q->head->val, curr->val, -1, q->head->deleted);
        }

        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    }
    q->size++;
}
```

terminal output:

```
sbt:Silicon> run ../gvc0/minprique_linkedlist.vpr
[info] compiling 1 Scala source to /home/judy_sun/gradual_verification/silicon-gv/target/scala-2.12/classes ...
[info] running (fork) viper.silicon.SiliconRunner ../gvc0/minprique_linkedlist.vpr
[info] Silicon finished verification successfully in 31.48s.
[success] Total time: 34 s, completed May 17, 2023, 4:39:02 PM
```

Complete code

1. gvc0 code

```
#use <conio>
#use <stress>
struct Node {
    int val;
    struct Node *next;
    bool deleted;
};
typedef struct Node Node;

struct MinPriorityQueue {
```

```
    Node *head;
    int size;
};
typedef struct MinPriorityQueue MinPriorityQueue;

/*@
predicate orderedListSegWithPrev(struct Node *start, struct Node *end, int prev,
int endVal) =
  (start == end) ?
    ( (end == NULL) ? true : endVal >= prev )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      start->val >= prev && orderedListSegWithPrev(start->next, end, start->val,
endVal)
    ) ;
@*/

/*@
predicate orderedListSeg(struct Node *start, struct Node *end, int endVal) =
  (start == end) ?
    ( true )
    :
    (
      acc(start->val) && acc(start->next) && acc(start->deleted) &&
      orderedListSegWithPrev(start->next, end, start->val, endVal)
    ) ;
@*/

/*@
predicate isMinPQ(Node *start) =
  orderedListSeg(start, NULL, -1);
@*/

//-----------------------------------------------------------------------lemmas
// Lemma:
void appendLemmaLoopBody(struct Node *a, struct Node *b, struct Node *c, int
aPrev, int cPrev, int bVal, int cVal, bool bDeleted)
  /*@
    requires orderedListSegWithPrev(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && acc(c->deleted) && c->val == cVal
&&
              c->val >= cPrev && orderedListSegWithPrev(c->next, NULL, c->val,
-1)
            )
        ) &&
        ( (b == c) ?
            ( true )
            :
            (
```

```
              acc(b->val) && acc(b->next) && acc(b->deleted) && b->val == bVal
&&
              orderedListSegWithPrev(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures orderedListSegWithPrev(a, c, aPrev, cVal) &&
        ( (c == NULL) ?
            ( true )
            :
            ( acc(c->val) && acc(c->next) && acc(c->deleted) && c->val == cVal
&&
              c->val >= cPrev && orderedListSegWithPrev(c->next, NULL, c->val,
-1)
            )
        ) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  } else {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    appendLemmaLoopBody(a->next, b, c, a->val, cPrev, bVal, cVal, b->deleted);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  }
}

void appendLemmaAfterLoopBody(struct Node *a, struct Node *b, struct Node *c,
int aPrev, int bVal, int cVal, bool bDeleted)
  /*@
    requires orderedListSegWithPrev(a, b, aPrev, bVal) &&
        ( (b == c) ? bVal == cVal : true ) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && acc(c->deleted) &&
c->val == cVal ) &&
        ( (b == c) ?
            ( true )
            :
            (
              acc(b->val) && acc(b->next) && acc(b->deleted) && b->val == bVal
&&
              orderedListSegWithPrev(b->next, c, b->val, cVal)
            )
        ) ;
  @*/
  /*@
    ensures orderedListSegWithPrev(a, c, aPrev, cVal) &&
        ( (c == NULL) ? true : acc(c->val) && acc(c->next) && acc(c->deleted) &&
c->val == cVal) ;
  @*/
{
  if (b == c) {
  } else if (a == b) {
```

```
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  } else {
    //@ unfold orderedListSegWithPrev(a, b, aPrev, bVal);
    appendLemmaAfterLoopBody(a->next, b, c, a->val, bVal, cVal, b->deleted);
    //@ fold orderedListSegWithPrev(a, c, aPrev, cVal);
  }
}

//-------------------------------------------------------------------------verified
code starts below

MinPriorityQueue* createMinPriQueue(int value)
    //@ requires true;
    //@ ensures acc((\result)->size) && acc((\result)->head) &&
isMinPQ((\result)->head);
{
    MinPriorityQueue *q = alloc(struct MinPriorityQueue);
    q->head = alloc(struct Node);
    q->head->val = value;
    q->head->next = NULL;
    q->size = 1;
    //@ fold orderedListSegWithPrev(q->head->next, NULL, q->head->val, -1);
    //@ fold orderedListSeg(q->head, NULL, -1);
    //@ fold isMinPQ(q->head);
    return q;
}

void enqueue(MinPriorityQueue *q, int value)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold orderedListSeg(q->head, NULL, -1);
    if (q->head == NULL || value <= q->head->val) { // insert at head
        Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = q->head;
        q->head = newNode;
        //@ fold orderedListSegWithPrev(newNode->next, NULL, newNode->val, -1);
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    } else { // insert after head
        Node *curr = q->head;
        //@ unfold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);
        //@ fold orderedListSeg(q->head, curr, curr->val);
        while (curr->next != NULL && curr->next->val < value)
        //@ loop_invariant acc(curr->val) && acc(curr->next) && acc(curr-
>deleted);
        //@ loop_invariant acc(q->head);
        //@ loop_invariant orderedListSeg(q->head, curr, curr->val) && curr->val
<= value;
        //@ loop_invariant (curr->next == NULL) ? (true) : acc(curr->next->next)
&& acc(curr->next->val) && acc(curr->next->deleted) && curr->next->val >= curr-
>val && orderedListSegWithPrev(curr->next->next, NULL, curr->next->val, -1);
```

```
        {
            Node *prev = curr;
            curr = prev->next;
            //@ unfold orderedListSeg(q->head, prev, prev->val);
            //@ fold orderedListSegWithPrev(prev->next, curr, prev->val, curr->val);

            if (q->head == prev) {
            } else {
              appendLemmaLoopBody(q->head->next, prev, curr, q->head->val, prev->val, prev->val, curr->val, prev->deleted);
            }

            //@ fold orderedListSeg(q->head, curr, curr->val);
            //@ unfold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);
        }
        struct Node *newNode = alloc(struct Node);
        newNode->val = value;
        newNode->next = curr->next;
        newNode->deleted = false;
        curr->next = newNode;
        //@ fold orderedListSegWithPrev(newNode->next, NULL, newNode->val, -1);
        //@ fold orderedListSegWithPrev(curr->next, NULL, curr->val, -1);

        //@ unfold orderedListSeg(q->head, curr, curr->val);
        if (q->head == curr) {
        } else {
          appendLemmaAfterLoopBody(q->head->next, curr, NULL, q->head->val, curr->val, -1, q->head->deleted);
        }

        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
    }
    q->size++;
}

void dequeue(MinPriorityQueue *q)
    //@ requires acc(q->head) && acc(q->size) && isMinPQ(q->head);
    //@ ensures acc(q->head) && acc(q->size) && isMinPQ(q->head);
{
    //@ unfold isMinPQ(q->head);
    //@ unfold orderedListSeg(q->head, NULL, -1);
    if (q->head == NULL) {
        // Queue is empty, nothing to dequeue
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
        return;
    } else {
        //@ unfold orderedListSegWithPrev(q->head->next, NULL, q->head->val, -1);
        q->head->deleted = true; // Mark node as deleted
        //@ fold orderedListSegWithPrev(q->head->next, NULL, q->head->val, -1);
        //@ fold orderedListSeg(q->head, NULL, -1);
        //@ fold isMinPQ(q->head);
```

```
    }
    q->size--;
}

int main ()
    //@ requires true;
    //@ ensures true;
{
    MinPriorityQueue *q = createMinPriQueue(1);
    enqueue(q, 10);
    enqueue(q, 5);
    enqueue(q, 20);
    // Node* deletedNode;
    // deletedNode = dequeue(q);
    // deletedNode = dequeue(q);
    // deletedNode = dequeue(q);
    dequeue(q);
    dequeue(q);
    dequeue(q);
    return 0;
}
```

2. vpr code

```
field MinPriorityQueue$head: Ref

field MinPriorityQueue$size: Int

field Node$deleted: Bool

field Node$next: Ref

field Node$val: Int

predicate isMinPQ(start: Ref) {
  acc(orderedListSeg(start, null, -1), write)
}

predicate orderedListSeg(start: Ref, end: Ref, endVal: Int) {
  (start == end ? true : acc(start.Node$val, write) && acc(start.Node$next,
write) && acc(start.Node$deleted, write) &&
acc(orderedListSegWithPrev(start.Node$next, end, start.Node$val, endVal),
write))
}

predicate orderedListSegWithPrev(start: Ref, end: Ref, prev: Int, endVal: Int) {
  (start == end ? (end == null ? true : endVal >= prev) : acc(start.Node$val,
write) && acc(start.Node$next, write) && acc(start.Node$deleted, write) &&
start.Node$val >= prev && acc(orderedListSegWithPrev(start.Node$next, end,
start.Node$val, endVal), write))
}
```

```
method appendLemmaAfterLoopBody(a: Ref, b: Ref, c: Ref, aPrev: Int, bVal: Int,
cVal: Int, bDeleted: Bool)
  requires acc(orderedListSegWithPrev(a, b, aPrev, bVal), write) && (b == c ?
bVal == cVal : true) && (c == null ? true : acc(c.Node$val, write) &&
acc(c.Node$next, write) && acc(c.Node$deleted, write) && c.Node$val == cVal) &&
(b == c ? true : acc(b.Node$val, write) && acc(b.Node$next, write) &&
acc(b.Node$deleted, write) && b.Node$val == bVal &&
acc(orderedListSegWithPrev(b.Node$next, c, b.Node$val, cVal), write))
  ensures acc(orderedListSegWithPrev(a, c, aPrev, cVal), write) && (c == null ?
true : acc(c.Node$val, write) && acc(c.Node$next, write) && acc(c.Node$deleted,
write) && c.Node$val == cVal)
{
  if (b == c) {

  } elseif (a == b) {
    unfold acc(orderedListSegWithPrev(a, b, aPrev, bVal), write)
    fold acc(orderedListSegWithPrev(a, c, aPrev, cVal), write)
  } else {
    unfold acc(orderedListSegWithPrev(a, b, aPrev, bVal), write)
    appendLemmaAfterLoopBody(a.Node$next, b, c, a.Node$val, bVal, cVal,
b.Node$deleted)
    fold acc(orderedListSegWithPrev(a, c, aPrev, cVal), write)
  }
}

method appendLemmaLoopBody(a: Ref, b: Ref, c: Ref, aPrev: Int, cPrev: Int, bVal:
Int, cVal: Int, bDeleted: Bool)
  requires acc(orderedListSegWithPrev(a, b, aPrev, bVal), write) && (b == c ?
bVal == cVal : true) && (c == null ? true : acc(c.Node$val, write) &&
acc(c.Node$next, write) && acc(c.Node$deleted, write) && c.Node$val == cVal &&
c.Node$val >= cPrev && acc(orderedListSegWithPrev(c.Node$next, null, c.Node$val,
-1), write)) && (b == c ? true : acc(b.Node$val, write) && acc(b.Node$next,
write) && acc(b.Node$deleted, write) && b.Node$val == bVal &&
acc(orderedListSegWithPrev(b.Node$next, c, b.Node$val, cVal), write))
  ensures acc(orderedListSegWithPrev(a, c, aPrev, cVal), write) && (c == null ?
true : acc(c.Node$val, write) && acc(c.Node$next, write) && acc(c.Node$deleted,
write) && c.Node$val == cVal && c.Node$val >= cPrev &&
acc(orderedListSegWithPrev(c.Node$next, null, c.Node$val, -1), write))
{
  if (b == c) {

  } elseif (a == b) {
    unfold acc(orderedListSegWithPrev(a, b, aPrev, bVal), write)
    fold acc(orderedListSegWithPrev(a, c, aPrev, cVal), write)
  } else {
    unfold acc(orderedListSegWithPrev(a, b, aPrev, bVal), write)
    appendLemmaLoopBody(a.Node$next, b, c, a.Node$val, cPrev, bVal, cVal,
b.Node$deleted)
    fold acc(orderedListSegWithPrev(a, c, aPrev, cVal), write)
  }
}

method createMinPriQueue(value: Int) returns ($result: Ref)
  requires true
```

```
  ensures acc($result.MinPriorityQueue$size, write) &&
acc($result.MinPriorityQueue$head, write) &&
acc(isMinPQ($result.MinPriorityQueue$head), write)
{
  var q: Ref
  var _: Ref
  q := new(MinPriorityQueue$head, MinPriorityQueue$size)
  _ := new(Node$val, Node$next, Node$deleted)
  q.MinPriorityQueue$head := _
  q.MinPriorityQueue$head.Node$val := value
  q.MinPriorityQueue$head.Node$next := null
  q.MinPriorityQueue$size := 1
  fold acc(orderedListSegWithPrev(q.MinPriorityQueue$head.Node$next, null,
q.MinPriorityQueue$head.Node$val, -1), write)
  fold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
  fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  $result := q
}

method dequeue(q: Ref)
  requires acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
  ensures acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
{
  unfold acc(isMinPQ(q.MinPriorityQueue$head), write)
  unfold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
  if (q.MinPriorityQueue$head == null) {
    fold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
    fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  } else {
    unfold acc(orderedListSegWithPrev(q.MinPriorityQueue$head.Node$next, null,
q.MinPriorityQueue$head.Node$val, -1), write)
    q.MinPriorityQueue$head.Node$deleted := true
    fold acc(orderedListSegWithPrev(q.MinPriorityQueue$head.Node$next, null,
q.MinPriorityQueue$head.Node$val, -1), write)
    fold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
    fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  }
  q.MinPriorityQueue$size := q.MinPriorityQueue$size - 1
}

method enqueue(q: Ref, value: Int)
  requires acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
  ensures acc(q.MinPriorityQueue$head, write) && acc(q.MinPriorityQueue$size,
write) && acc(isMinPQ(q.MinPriorityQueue$head), write)
{
  var newNode: Ref
  var curr: Ref
  var newNode1: Ref
  var prev: Ref
  unfold acc(isMinPQ(q.MinPriorityQueue$head), write)
  unfold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
```

```
  if (q.MinPriorityQueue$head == null || value <=
q.MinPriorityQueue$head.Node$val) {
    newNode := new(Node$val, Node$next, Node$deleted)
    newNode.Node$val := value
    newNode.Node$next := q.MinPriorityQueue$head
    q.MinPriorityQueue$head := newNode
    fold acc(orderedListSegWithPrev(newNode.Node$next, null, newNode.Node$val,
-1), write)
    fold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
    fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  } else {
    curr := q.MinPriorityQueue$head
    unfold acc(orderedListSegWithPrev(curr.Node$next, null, curr.Node$val, -1),
write)
    fold acc(orderedListSeg(q.MinPriorityQueue$head, curr, curr.Node$val),
write)
    while (curr.Node$next != null && curr.Node$next.Node$val < value)
      invariant acc(curr.Node$val, write) && acc(curr.Node$next, write) &&
acc(curr.Node$deleted, write) && acc(q.MinPriorityQueue$head, write) &&
(acc(orderedListSeg(q.MinPriorityQueue$head, curr, curr.Node$val), write) &&
curr.Node$val <= value) && (curr.Node$next == null ? true :
acc(curr.Node$next.Node$next, write) && acc(curr.Node$next.Node$val, write) &&
acc(curr.Node$next.Node$deleted, write) && curr.Node$next.Node$val >=
curr.Node$val && acc(orderedListSegWithPrev(curr.Node$next.Node$next, null,
curr.Node$next.Node$val, -1), write))
    {
      prev := curr
      curr := prev.Node$next
      unfold acc(orderedListSeg(q.MinPriorityQueue$head, prev, prev.Node$val),
write)
      fold acc(orderedListSegWithPrev(prev.Node$next, curr, prev.Node$val,
curr.Node$val), write)
      if (q.MinPriorityQueue$head == prev) {

      } else {
        appendLemmaLoopBody(q.MinPriorityQueue$head.Node$next, prev, curr,
q.MinPriorityQueue$head.Node$val, prev.Node$val, prev.Node$val, curr.Node$val,
prev.Node$deleted)
      }
      fold acc(orderedListSeg(q.MinPriorityQueue$head, curr, curr.Node$val),
write)
      unfold acc(orderedListSegWithPrev(curr.Node$next, null, curr.Node$val,
-1), write)
    }
    newNode1 := new(Node$val, Node$next, Node$deleted)
    newNode1.Node$val := value
    newNode1.Node$next := curr.Node$next
    newNode1.Node$deleted := false
    curr.Node$next := newNode1
    fold acc(orderedListSegWithPrev(newNode1.Node$next, null, newNode1.Node$val,
-1), write)
    fold acc(orderedListSegWithPrev(curr.Node$next, null, curr.Node$val, -1),
write)
    unfold acc(orderedListSeg(q.MinPriorityQueue$head, curr, curr.Node$val),
write)
```

```
    if (q.MinPriorityQueue$head == curr) {

    } else {
      appendLemmaAfterLoopBody(q.MinPriorityQueue$head.Node$next, curr, null,
q.MinPriorityQueue$head.Node$val, curr.Node$val, -1,
q.MinPriorityQueue$head.Node$deleted)
    }
    fold acc(orderedListSeg(q.MinPriorityQueue$head, null, -1), write)
    fold acc(isMinPQ(q.MinPriorityQueue$head), write)
  }
  q.MinPriorityQueue$size := q.MinPriorityQueue$size + 1
}

method main() returns ($result: Int)
  requires true
  ensures true
{
  var q: Ref
  q := createMinPriQueue(1)
  enqueue(q, 10)
  enqueue(q, 5)
  enqueue(q, 20)
  dequeue(q)
  dequeue(q)
  dequeue(q)
  $result := 0
}

method printint(value: Int)
{

}

method printchar(value: Int)
{

}

method printbool(value: Bool)
{

}

method flush()
{

}

method mod(r: Int, l: Int) returns ($result: Int)
{
  $result := 0
}

method rand(prev: Int) returns ($result: Int)
{
```

```
  $result := 0
}

method readStress() returns ($result: Int)
{
  $result := 0
}
```

3. Restrictions

In the implementation of dequeue() function, instead of directly deleting the node, I add a Boolean flag for each node indicating whether it is "deleted" or not. This design may leave deleted nodes in the queue, so periodic cleaning might be necessary.