```cpp
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_H
#define THREE_TIER_HEAP_EVENT_QUEUE_H

#include <vector>
#include <stack>
#include <algorithm>
#include "Avg.h"
#include "EventQueue.h"
#include "TwoTierHeapOfVectorsEventQueue.h"

BEGIN_NAMESPACE(xxxx)

/** Class to encapsulate information for Tier2 entries/buckets.

        This is a simple class that encapsulates a list of events (in
        eventList) all with exactly the same receive time to a given
        agent.  These objects are cached/reused by the scheduler queue to
        reduce memory allocation operations, particularly for the
        eventList because memory management turns out to be he most
        expensive operation.
*/
class HOETier2Entry {
private:
    /** The receive time of events in this tier2 entry. Note that all
        the events in this entry are must/are concurrent -- that is,
        destined for the same agent at the same time.
     */
    Time recvTime;

    /** The list of entities in this HOE entry class */
    std::vector<xxxx::Event*> eventList;

public:
    /** Constructor to create a tier2 entry with 1 initial event in
        it.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    HOETier2Entry(xxxx::Event* event) : recvTime(event->getReceiveTime()),
                                        eventList(1, event) {}

    /** Reset the information in this tier2 entry.

        This method is synonymous to the constructor except, it is
        used to reset/recycle an existing tier2 entry.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    void reset(xxxx::Event* event) {
        recvTime = event->getReceiveTime();
        eventList.clear();
        eventList.emplace_back(event);
    }

    /** Appends events to the EventContainer list.
     *
     *  The method is used to append concurrent events to their respective
     *  position in the tier2 container.
     */
    void updateContainer(xxxx::Event* event){
        eventList.emplace_back(event);
    }

    /** Obtain pointer to the first event in this list.

        \return Pointer to the first event in this list. This return
        value cannot/should-not be NULL.
    */
    inline xxxx::Event* getEvent() const {
        return eventList.front();
    }
```

```cpp

    /** \brief compares the receive times of events

        The method is used to determine whether or not an event
        already exists in the tier2 container.

        \returns True if lhs receiveTime is equal to rhs receiveTime
    */
    inline bool operator==(const HOETier2Entry &rhs) {
        return (this->recvTime == rhs.recvTime);
    }

    inline bool operator<(const HOETier2Entry &rhs) {
        return (this->recvTime < rhs.recvTime);
    }

    inline Time getReceiveTime() const {
        return recvTime;
    }

    inline const std::vector<xxxx::Event*>& getEventList() const {
        return eventList;
    }

    inline std::vector<xxxx::Event*>& getEventList() {
        return eventList;
    }
};

/** A three-tier-heap aka "3tHeap" or "heap-of-heap" event queue for
    managing events.

    <p>This class provides a heap-of-heap based event queue for
    managing events for simulation.  The two-tiers are organized as
    follows:</p>

    <p><u>First tier:</u> This class uses standard C++ algorithms
    (such as: \c std::make_heap, \c std::push_heap, \c std::pop_heap)
    to manage a heap of events for each agent.  The events are stored
    in a backing std::vector in each agent.  It is the same per-agent
    infrastructure as used by Fibonacci heap (implemented in AgentPQ
    class).</p>

    <p><u>Second tier:</u> This class specifically handles the
    necessary behavior of the second tier of operations -- that is
    scheduling of agents by maintaining heap of agents.</p>

    \note On the long run it would be better to avoid reliance on
    std::push_heap or std::pop_heap methods due to implicit dependence
    in the fixHeap() method.
*/
class ThreeTierHeapEventQueue : public EventQueue {
public:
    /** The constructor for the TwoTierHeapEventQueue.

        The default (and only) constructor for this class.  The
        constructor does not have any specific task to perform other
        than set a suitable identifier in the base class.
    */
    ThreeTierHeapEventQueue();

    /** The destructor.

        The destructor does not have any special tasks to perform.
        All the dynamic memory management is handled by the standard
        containers (namely std::vector) that is used internally by
        this class.
    */
    ~ThreeTierHeapEventQueue();

    /** Add/register an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>

        <p>This class uses the supplied agent pointer to setup the
        list of agents managed and scheduled by this class.</p>

        \param[in,out] agent A pointer to the agent to be registered.
        This value is not used.

        \return This method returns the iterator to the position of
        the agent in its internal vector as a cross-reference to be
        stored in an agent.
    */
    virtual void* addAgent(xxxx::Agent* agent);

    /** Remove/unregister an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>

        <p>This method removes all events scheduled for the specified
        agent in its internal data structures.</p>

        \param[in,out] agent A pointer to the agent whose events are
        to be removed from the vector managed by this class.
    */
    void removeAgent(xxxx::Agent* agent) override;

    /** Determine if the event queue is empty.

        This method implements the base class API to report if any
        events are pending to be processed in the event queue.

        \return This method returns true if the event queue of the
        top-agent is logically empty.
    */
    virtual bool empty() {
        return (agentList.empty() || top()->tier2->empty());
    }

    /** Obtain pointer to the highest priority (lowest receive time)
        event.

        This method can be used to obtain a pointer to the highest
        priority event in this event queue, without de-queuing the
        event.

        \note The event returned by this method is not dequeued.

        \return A pointer to the next event to be processed.  If the
        queue is empty then this method returns NULL.
    */
    virtual xxxx::Event* front();

    /** Method to obtain the next batch of events to be processed by
        one agent.

        <p>In XXXX agents are scheduled to process all events at a
        given simulation time.  The next concurrent events (i.e.,
        events with the same receive time) with the lowest time stamp
        are to be placed in the supplied event container.  The event
        container is then passed to the corresponding agent for
        further processing.</p>

        <p>This method essentially delegates the dequeue process to
        the agent with the next lowest timestamp.  Once the agent has
        been dequeued, this method fixes the heap by placing the
        top-agent in its appropriate location in the heap.</p>

        \param[out] events The event container in which the next set
        of concurrent events are to be placed.  Note that the order of
        concurrent events in the event container is unspecified.
    */
    virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);

    /** Enqueue a new event.
```

```

        This method must be used to enqueue/add an event to this event
        queue.  Once added the reference count on the event is
        increased.  This method adds the event to the specified agent.
        Next this method fixes the heap to ensure that the agent with
        the least-time-stamp is at the top of the heap.  This method
        essentially uses an internal helper method to accomplish its
        tasks.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.

        \param[in] event The event to be enqueued.  This parameter can
        never be NULL.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);

    /** Enqueue a batch of events.

        This method can be used to enqueue/add a batch of events to
        this event queue.  Once added the reference count on each one
        of the events is increased.  This method provides a convenient
        approach to enqueue a batch of events, particularly after a
        rollback.  Next this method fixes the heap to ensure that the
        agent with the least-time-stamp is at the top of the heap.
        This method uses an internal helper method to accomplish its
        tasks.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.  Currently, this value is
        not used.

        \param[in] event The list of events to be enqueued.  This
        container can and will be be empty in certain situations.  The
        reference counts of the events in the container remains
        unmodified.  The list of events become part of the event
        queue.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);

    /** Dequeue all events sent by an agent after a given time.

        This method implements the base class API method.  This method
        can be used to remove/erase events sent by a given agent after
        a given simulation time.  This API is needed to cancel events
        during a rollback.  Next this method fixes the heap to ensure
        that the agent with the least-time-stamp is at the top of the
        heap.

        \param[in] dest The agent whose currently scheduled events
        are to be checked and cleaned-up.  This agent must be a valid
        agent that has been registered/added to this event queue.  The
        pointer cannot be NULL.  This parameter is not used.

        \param[in] sender The ID of the agent whose events have to be
        removed.  This agent must be a valid agent that has been
        registered/added to this event queue.  The pointer cannot be
        NULL.

        \param[in] sentTime The time from which the events are to be
        removed.  All events (including those sent at this time) sent
        by the sender agent are removed from this event queue.

        \return This method returns the number of events actually
        removed.
    */
    virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
                           const xxxx::Time sentTime);

    /** Print full contents of scheduler queue to given output stream.

        This is a convenience method that is used primarily for
        troubleshooting purposes.  This method prints all the events
        in this queue, with each event on its own line.

        \param[out] os The output stream to which the contents of the
        queue are to be written.
    */
    virtual void prettyPrint(std::ostream& os) const;

    /** Method to report aggregate statistics.

        This method is invoked at the end of simulation after all
        agents on this rank have been finalized.  This method can
        report any aggregate statistics from the event
        queue. Currently, this method does not have any additional
        statistics to report.

        \param[out] os The output stream to which the statistics are
        to be written.
    */
    virtual void reportStats(std::ostream& os);

protected:
    /** Enqueue a new event.

        This method must be used to enqueue/add an event to this event
        queue.  Once added the reference count on the event is
        increased.  This method adds the event to the specified agent.
        Next this method fixes the heap to ensure that the agent with
        the least-time-stamp is at the top of the heap.

        \param[in] agent The agent to which the event is to be
        scheduled.  This agent corresponds to the agent ID returned by
        event->getReceiverAgentID() method.

        \param[in] event The event to be enqueued.  This parameter can
        never be NULL.
    */
    virtual void enqueueEvent(xxxx::Agent* agent, xxxx::Event* event);

    /** Convenience method to remove events.

        This is an internal convenience method that is used to remove
        the front (i.e., events with lowest timestamp) event list from this
        queue.
    */
    void pop_front(xxxx::Agent* agent);

    /** Convenience method to obtain the top-most or front agent.

        This method can be used to obtain a pointer to the top/front
        agent -- that is, the agent with the lowest timestamp event to
        be scheduled next.

        \return A pointer to the top-most agent in this heap.
    */
    inline xxxx::Agent* top() {
        return agentList.front();
    }

    /** Convenience method to get the top-event time for a given
        agent.

        This method returns the top event time in the vector of events queue.
        If agent’s vector of event queue is empty, then it returns infinity.

        \return The receive time of top event’s recv time or
        TIME_INFINITY if vector is empty.
    */
    inline xxxx::Time getTopTime(const xxxx::Agent* const agent) const {
        return agent->tier2->empty() ? TIME_INFINITY :
            agent->tier2->front()->getReceiveTime();
    }

    /** Comparator method to sort events in the heap.

        This is the comparator method that is passed to various
```

```
        standard C++ algorithms to organize events as a heap.  This
        comparator method gives first preference to receive time of
        events.  Tie between two events with the same recieve time is
        broken based on the receiver agent ID.

        \param[in] lhs The left-hand-side event to be used for
        comparison.  This parameter cannot be NULL.

        \param[in] rhs The right-hand-side event to be used for
        comparison. This parameter cannot be NULL.

        \return This method returns if lhs < rhs, i.e., the lhs event
        should be scheduled before the rhs event.
    */
    inline bool compare(const Agent *lhs, const Agent * rhs) const {
        return getTopTime(lhs) >= getTopTime(rhs);
    }

    /** Comparator method to sort events in the heap.

        This is the comparator method that is passed to various
        standard C++ algorithms to organize events as a heap.  This
        comparator method gives first preference to receive time of
        events.  Tie between two events with the same recieve time is
        broken based on the receiver agent ID.

        \param[in] lhs The left-hand-side event to be used for
        comparison.  This parameter cannot be NULL.

        \param[in] rhs The right-hand-side event to be used for
        comparison. This parameter cannot be NULL.

        \return This method returns if lhs < rhs, i.e., the lhs event
        should be scheduled before the rhs event.
    */
    inline static bool lessThan(const HOETier2Entry& lhs,
                                const xxxx::Event* const event) {
        return (lhs.getReceiveTime() < event->getReceiveTime());
    }

    /** Comparator method to sort events in the heap.

        This is the comparator method that is passed to various
        standard C++ algorithms to organize events as a heap.  This
        comparator method gives first preference to receive time of
        events.  Tie between two events with the same recieve time is
        broken based on the receiver agent ID.

        \param[in] lhs The left-hand-side event to be used for
        comparison.  This parameter cannot be NULL.

        \param[in] rhs The right-hand-side event to be used for
        comparison. This parameter cannot be NULL.

        \return This method returns if lhs < rhs, i.e., the lhs event
        should be scheduled before the rhs event.
    */
    inline static bool lessThanPtr(const HOETier2Entry* const lhs,
                                   const xxxx::Event* const event) {
        return (lhs->getReceiveTime() < event->getReceiveTime());
    }

    /** The getNextEvents method.

        This method is a helper that will grab the next set of events
        to be processed for a given agent.  This method is invoked in
        dequeueNextAgentEvents() method in this class.

        \param[out] container The reference of the container into
        which events should be added.
    */
    void getNextEvents(Agent* agent, EventContainer& container);

        /** Obtain the current index of the agent from it’s
        cross-reference.

        This method is a refactored utility method that has been
        introduced to streamline the code.  This method essentially
        obtains the index position of the given agent in the agentList
        vector from the agent’s fibHeapPtr corss−reference.  This
        cross−reference is consistently updated by the various methods
        in this class to enable rapid access to the location of the
        agent.

        \param[in] agent The agent whose index value in the agentList
        is to be determined.

        \return The index position of the agent in the agentList
        vector (if all checks pass).
    */
    size_t getIndex(xxxx::Agent *agent) const;

    /** Update position of agent in the scheduler’s heap.

        This is an internal helper method that is used to update the
        position of an agent in the scheduler’s heap.  This method
        essentially performs sanity checks, uses the fixHeap() method
        to update position of the agent, and updates cross references
        for future use.

        \param[in] agent The agent whose position in the heap is to be
        updated.  This pointer cannot be NULL.

        \return This method returns the updated index position of the
        agent in agentList (the vector that serves as storage for the
        heap).
    */
    size_t updateHeap(xxxx::Agent* agent);

    /** Fix−up the location of the agent in the heap.

        This method can be used to update the location of an agent in
        the heap.

        \note The implementation for this method has been heavily
        borrowed from libstdc++’s code base to ensure that heap
        updates are consistent with std::make_heap API.
        Unfortunately, this does imply that there is a chance this
        method may be incompatible with future versions.

        \param[in] currPos The current position of the agent in the
        heap whose position is to be updated.  This value is the index
        position of the agent in the agentList vector.

        \return This method returns the new position of the agent in
        the agentList vector.
    */
    size_t fixHeap(size_t currPos);

    /** Convenience method to determine if an event is a future event.

        This method is a helper method used in the eraseAfter() method
        to determine if a given event is a future event from a given
        sender agent.

        \param[in] sender The sender agent to be used in comparison.

        \param[in] sendTime The reference time for comparison

        \param[in] evt The event to be checked if it is future event.

        \return This method returns true if the event is sent from a
        given sender agent and its send time is greater−or−equal to
        the given sentTime.
     */
    inline bool
    isFutureEvent(const xxxx::AgentID sender, const xxxx::Time sentTime,
                  const xxxx::Event* evt) const {
        return ((evt−>getSenderAgentID() == sender) &&
                (evt−>getSentTime() >= sentTime));
```

```
    }

    /** Helper method to reuse tier2 entries or create a new one.

        This method is a convenience method to recycle tier2 entry
        object is available.  If the recycle bin is empty, then this
        method creates a new object.

        \param[in] event The event to be used to initialize and to be
        contained in the newly created tier2 entry.

        \return A tier2 entry initialized and containing the given
        event.
     */
    HOETier2Entry* makeTier2Entry(xxxx::Event* event) {
        if (!tier2Recycler.empty()) {
            HOETier2Entry* entry = tier2Recycler.back();
            tier2Recycler.pop_back();
            entry−>reset(event);
            return entry;
        }
        return new HOETier2Entry(event);
    }

private:
    /** The backing storage for events managed by this class.

        This vector contains the list of agents being managed by the
        class.  The agents in the vector are stored and maintained as
        a heap.  The heap is created and managed using standard C++
        algorithms, namely: \c std::make_heap, \c std::push_heap, and
        \c std::pop_heap.
    */
    std::vector<xxxx::Agent*> agentList;

    /** Stats object to track the average tier−2 bucket size.  This
        value is the one that primarily determines if tier−2
        operations are going to be optimal or not.  Higher this value,
        the better for this event queue.
    */
    Avg avgSchedBktSize;

    /** The number of times the fixHeap method performed heap−fixing
        operations.  This variable is fine−grained in that it
        accumulates the average number of compares that occur to fix
        up the heap of agents.  The fixHeap method has a q1 = O(log
        n1) compares.  So if this method is called m times, the
        statistics reports (q1 + q2 + ... + qm) / m.
    */
    Avg fixHeapSwapCount;

    /** The average queue size for each agent.  This value determines
        the time takes to find a bucket into which an event is to be
        inserted.
    */
    Avg agentBktCount;

    /** A stack to recycle Tier2 entries to minimize memory allocation
        calls for these small blocks used in this queue.
    */
    std::deque<HOETier2Entry*> tier2Recycler;
};

END_NAMESPACE(xxxx)

#endif
```

```cpp
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_CPP
#define THREE_TIER_HEAP_EVENT_QUEUE_CPP

#include "ThreeTierHeapEventQueue.h"
#include <algorithm>

BEGIN_NAMESPACE(xxxx)

// A convenience shortcut used just in this source file
using Tier2List = std::deque<HOETier2Entry*>;
// using Tier2List = std::vector<Tier2Entry>;

ThreeTierHeapEventQueue::ThreeTierHeapEventQueue() :
    EventQueue("HeapOfVectorsEventQueue") {
    // Nothing else to be done.
}

ThreeTierHeapEventQueue::~ThreeTierHeapEventQueue() {
    // Clear up memory allocated for HOETier2Entry
    for (HOETier2Entry* entry : tier2Recycler) {
        delete entry;
    }
}

void*
ThreeTierHeapEventQueue::addAgent(xxxx::Agent* agent) {
    agentList.push_back(agent);
    // Create the vector that is used to manage events for the agent.
    agent->tier2 = new Tier2List();
    return reinterpret_cast<void*>(agentList.size() - 1);
}

void
ThreeTierHeapEventQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    ASSERT(!empty());
    // Decrease reference count for all events in the agent event queue
    // before agent removal.
    ASSERT( agent->tier2 != NULL );
    // Logically remove events in this agent's tier2 queues/buckets
    Tier2List& tier2eventPQ = *agent->tier2;
    for (xxxx::HOETier2Entry* bucket : tier2eventPQ) {
        for (Event* evt : bucket->getEventList()) {
            evt->decreaseReference();  // logically remove event
        }
        // Free the memory reserved for this bucket
        delete bucket;
    }
    // Clear out tier2 queue (so this agent's time becomes PINFINITY)
    agent->tier2->clear();
    // Update the heap to place agent with LTSF
    updateHeap(agent);
}

xxxx::Event*
ThreeTierHeapEventQueue::front() {
    return (!top()->tier2->empty()) ? top()->tier2->front()->getEvent() : NULL;
}

void
ThreeTierHeapEventQueue::pop_front(xxxx::Agent* agent) {
    // Decrease reference count for all events in the front of the
    // agent event queue before the list of events is removed from the
    // event queue.
    std::vector<xxxx::Event*>& eventList =
        agent->tier2->front()->getEventList();
    for (Event* evt: eventList) {
        evt->decreaseReference();
    }
    // agent->tier2->erase(agent->tier2->begin());
    tier2Recycler.emplace_back(agent->tier2->front());
    agent->tier2->pop_front();
}

void
```

```cpp
ThreeTierHeapEventQueue::getNextEvents(Agent* agent,
                                       EventContainer& container) {
    ASSERT(container.empty());
    ASSERT(agent->tier2 != NULL);
    Tier2List& tier2 = *agent->tier2;
    ASSERT(tier2.front()->getEvent() != NULL);
    // All events in tier2 front should have same receive times
    const xxxx::Time eventTime = tier2.front()->getReceiveTime();
    // Copy all the events out of the tier2 front into the return contianer
    // container = std::move(agent->tier2->front().getEventList());
    std::vector<xxxx::Event*>& evtList = tier2.front()->getEventList();
    container.assign(evtList.begin(), evtList.end());
    DEBUG({
            // Do validation checks on the events in tier2
            for (const Event* event : container) {
                //  All events must have the same receive time
                ASSERT( event->getReceiveTime() == eventTime );

                // We should never process an anti-message.
                if (event->isAntiMessage()) {
                    std::cerr << "Anti-message Processing: " << *event
                              << std::endl;
                    std::cerr << "Trying to process an anti-message event, "
                              << "please notify XXXX developers of this issue"
                              << std::endl;
                    abort();
                }
                // Ensure that the top event is greater than LVT
                if (event->getReceiveTime() <= agent->getTime(Agent::LVT)) {
                    std::cerr << "Agent is being scheduled to process "
                              << "an event ("
                              << *event << ") that is at or below it LVT (LVT="
                              << agent->getTime(Agent::LVT) << ", GVT="
                              << agent->getTime(Agent::GVT)
                              << "). This is a serious error. Aborting.\n";
                    std::cerr << *agent << std::endl;
                    abort();
                }
                // Ensure reference counts are consistent.
                ASSERT(event->getReferenceCount() < 3);
                DEBUG(std::cout << "Delivering: " << *event << std::endl);
            }
        });
    // Recycle the entry at the beginning of the queue.
    tier2Recycler.emplace_back(tier2.front());
    tier2.pop_front();
    // std::rotate(tier2.begin(), tier2.begin() + 1, tier2.end());
    // tier2.pop_back();
    // Track bucket/block size statistics
    avgSchedBktSize += container.size();
}

void
ThreeTierHeapEventQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (!empty()) {
        // Get agent and validate.
        xxxx::Agent* const agent = top();
        ASSERT(agent != NULL);
        ASSERT(getIndex(agent) == 0);
        // Have the events give up its next set of events
        getNextEvents(agent, events);
        ASSERT(!events.empty());
        // Fix the position of this agent in the scheduler's heap.
        updateHeap(agent);
    }
}

void
ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    // Use helper method (just below this one) to add event and fix-up
    // the queue.  First Increase event reference count for every
    // event added to the event queue.
    ASSERT( event->getReferenceCount() < 2 );
    event->increaseReference();
    enqueueEvent(agent, event);
    updateHeap(agent);
}

void
ThreeTierHeapEventQueue::enqueueEvent(xxxx::Agent* agent, xxxx::Event* event) {
    ASSERT(agent != NULL);
    ASSERT(event != NULL);
    ASSERT( agent->tier2 != NULL );
    ASSERT(getIndex(agent) < agentList.size());
    // A convenience reference to tier2 list of buckets
    Tier2List& tier2 = *agent->tier2;
    // Use binary search O(log n) to find match or insert position
    agentBktCount += tier2.size();
    Tier2List::iterator iter =
        std::lower_bound(tier2.begin(), tier2.end(), event, lessThanPtr);
    // There are 3 cases: 1. we found matching bucket, 2: iterator
    // to bucket with higher recvTime, or 3: tier2.end().
    if (iter == tier2.end()) {
        tier2.emplace_back(makeTier2Entry(event));  // add new entry to end.
    } else if ((*iter)->getReceiveTime() == event->getReceiveTime()) {
        // We found an existing bucket. Append this event to this
        // existing bucket.
        (*iter)->updateContainer(event);
    } else {
        // If there is no bucket with a matching receive time in Tier2
        // vector, then insert an instance of HOETier2Entry (aka
        // bucket) into the vector at the appropriate position.
        ASSERT((*iter)->getReceiveTime() > event->getReceiveTime());
        tier2.emplace(iter, makeTier2Entry(event));
    }
    // ASSERT(std::is_sorted(tier2.begin(), tier2.end()));
}

void
ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent,
                                 xxxx::EventContainer& events) {
    ASSERT(agent != NULL);
    // Note: events container may be empty!
    ASSERT(getIndex(agent) < agentList.size());
    // Add all events to tier2 entries appropriately.
    for (xxxx::Event* event : events) {
        // Enqueue event but don't waste time fixing-up heap yet for
        // this agent.  We will do it at the end after all events are
        // added.  However, we don't increase reference counts in this
        // API.
        enqueueEvent(agent, event);
    }
    // Clear out all the events in the incoming container
    events.clear();
    // Update the location of this agent on the heap as needed.
    updateHeap(agent);
}

int
ThreeTierHeapEventQueue::eraseAfter(xxxx::Agent* dest,
                                    const xxxx::AgentID sender,
                                    const xxxx::Time sentTime) {
    int  numRemoved = 0;
    ASSERT( dest->tier2 != NULL );
    Tier2List& tier2eventPQ = *dest->tier2;
    long currIdx = tier2eventPQ.size() - 1;
    while (!tier2eventPQ.empty() && (currIdx >= 0)) {
        if (tier2eventPQ[currIdx]->getReceiveTime() > sentTime) {
            std::vector<xxxx::Event*>& eventList =
                tier2eventPQ[currIdx]->getEventList();
            size_t index = 0;
            while (!eventList.empty() && (index < eventList.size())) {
                Event* const evt = eventList[index];
                ASSERT(evt != NULL);
                if (isFutureEvent(sender, sentTime, evt)) {
                    evt->decreaseReference();
                    numRemoved++;
                    eventList[index] = eventList.back();
                    eventList.pop_back();
                } else {
```

```cpp
                    index++;  // onto next event in this bucket
                }
            }
            // If all events are canceled then this bucket needs to be
            // removed from the tier2 entry.
            if (eventList.empty()) {
                tier2Recycler.emplace_back(tier2eventPQ[currIdx]);
                tier2eventPQ.erase(tier2eventPQ.begin() + currIdx);
            }
        }
        currIdx--;
    }
    // Update the 1st tier heap for scheduling.
    updateHeap(dest);
    // Return number of events canceled to track statistics.
    return numRemoved;
}

void
ThreeTierHeapEventQueue::reportStats(std::ostream& os) {
    UNUSED_PARAM(os);
    const long comps = std::log2(agentList.size()) *
        avgSchedBktSize.getCount() + fixHeapSwapCount.getSum();
    os << "Average #buckets per agent  : " << agentBktCount    << std::endl;
    os << "Average scheduled bucket size: " << avgSchedBktSize  << std::endl;
    os << "Average fixHeap compares    : " << fixHeapSwapCount << std::endl;
    os << "Compare estimate            : " << comps            << std::endl;
}

void
ThreeTierHeapEventQueue::prettyPrint(std::ostream& os) const {
    os << "HeapOfVectorsEventQueue::prettyPrint() : not implemented.\n";
}

size_t
ThreeTierHeapEventQueue::getIndex(xxxx::Agent *agent) const {
    ASSERT(agent != NULL);
    size_t index = reinterpret_cast<size_t>(agent->fibHeapPtr);
    ASSERT(index < agentList.size());
    ASSERT(agentList[index] == agent);
    return index;
}

size_t
ThreeTierHeapEventQueue::updateHeap(xxxx::Agent* agent) {
    ASSERT(agent != NULL);
    size_t index = getIndex(agent);
    if (agent->oldTopTime != getTopTime(agent)) {
        index = fixHeap(index);
        // Update the position of the agent in the scheduler's heap
        // Validate
        ASSERT(agentList[index] == agent);
        ASSERT(getIndex(agent) == index);
        // Update time value as well for future access
        agent->oldTopTime = getTopTime(agent);
        // Validation check.
        ASSERT(getTopTime(agentList[0]) <= getTopTime(agentList[1]));
    }
    // Return the new index position of the agent
    return index;
}

size_t
ThreeTierHeapEventQueue::fixHeap(size_t currPos) {
    ASSERT(currPos < agentList.size());
    xxxx::Agent* value      = agentList[currPos];
    const size_t len        = (agentList.size() - 1) / 2;
    size_t secondChild      = currPos;
    int          opCount    = 0;
    // This code was borrowed from libstdc++ implementation to ensure
    // that the fix-ups are consistent with std::make_heap API.
    while (secondChild < len)  {
        secondChild = 2 * (secondChild + 1);
        if (compare(agentList[secondChild], agentList[secondChild - 1])) {
            secondChild--;
        }
        agentList[currPos] = std::move(agentList[secondChild]);
        agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
        currPos = secondChild;
        opCount++;  // track statistics on number of operations performed
    }
    if (((agentList.size() & 1) == 0) &&
        (secondChild == (agentList.size() - 2) / 2)) {
        secondChild        = 2 * (secondChild + 1);
        agentList[currPos] = std::move(agentList[secondChild - 1]);
        agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
        currPos            = secondChild - 1;
        opCount++;  // track statistics on number of operations performed
    }
    // Use libstdc++’s internal method to fix-up the vector from the
    // given location.
    // std::__push_heap(agentList.begin(), currPos, 0, value,
    //                  __gnu_cxx::__ops::__iter_comp_val(compare));

    size_t parent = (currPos - 1) / 2;
    while ((currPos > 0) && (compare(agentList[parent],value))) {
        agentList[currPos] = std::move(agentList[parent]);
        agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
        currPos = parent;
        parent  = (currPos - 1) / 2;
        opCount++;  // track statistics on number of operations performed
    }
    agentList[currPos] = value;
    agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
    // Update aggregate statistics
    fixHeapSwapCount += opCount;
    // Return the final index position for the agent
    return currPos;
}

END_NAMESPACE(xxxx)

#endif
```

```
#ifndef TWO_TIER_LADDER_QUEUE_H
#define TWO_TIER_LADDER_QUEUE_H

#include <forward_list>
#include <queue>
#include <vector>
#include <typeinfo>
#include <set>
#include "Avg.h"
#include "Event.h"
#include "EventQueue.h"

/** \file LadderQueue.h

    \brief Enhancement of LadderQueue to improve performance of
    Optimistic Parallel Simulations by minimizing rollbacks.

    The LadderQueue data structure is detailed in the following paper:

    W. Tang, R. Goh, and I. Thng, "Ladder queue: An O(1) priority
    queue structure for large-scale discrete event simulation", ACM
    TOMACS, Vol 15, Issue 3, Pages 175--204, July 2005. URL:
    http://doi.acm.org/10.1145/1103323.1103324

    <p>One major disadvantage of the LadderQueue is that canceling
    events due to a rollback is expensive -- the whole queue has to be
    scanned.</p>

    <p>In order to reduce the overhead of scanning events for
    canceling, this TwoTierLadderQueue further subdivides each bucket
    in Top and ladder Rung to store events in a 2nd Tier based on
    their sender's ID.  It uses a simple hash function on the sender's
    AgentID to identify the 2nd tier bucket and enqueue's the event
    into that bucket.  Currently, the hash function is simply
    implemented as a modulo t2k, with t2k being an implementation
    dependent value.  Since second tier buckets (implemented as
    std::vector) are preallocated, Small t2k values increase 2nd tier
    bucket sizes increasing time for cancellation.  On the other hand
    if buckets are not used, then the space/time invested to create
    them can become an overhead.</p>

    <p>Note that the TwoTierLadderQueue would have very similar
    characteristics to LadderQueue in sequential or 1 process
    simulation as there are no rollbacks</p>
*/

// Bucket size after which new rung is created in ladder
#define LQ2T_THRESH 50

/** \def LQ2T_STATS(x)

    \brief Define a convenient macro for conditionally compiling
    additional statistics collection regarding ladder queue.

    Define a custom macro LQ2T_STATS (note the all caps) macro to be
    used to conditionally compile in debugging code to generate
    detailed logs.  This helps to minimize code modification to insert
    and remove debugging messages.
*/
#define COMMA ,
#define LQ2T_STATS(x) x
// #define LQ2T_STATS(x)

BEGIN_NAMESPACE(xxxx)

/** A convenience alias for list of events maintained by a sub-bucket. */
using BktEventList = std::vector<xxxx::Event*>;

/** Alias to the data structure for holding a vector of sub-buckets in
    a TwoTierBucket.
*/
using SubBucketList = std::vector<BktEventList>;

constexpr bool SenderID   = true;
constexpr bool ReceiverID = false;
```

```

/** A generic two tier bucket that is used for both Top and Rungs of
    the 2-tier ladderQ.

    <p>This bucket does not store events in it directly.  Instead it
    splits into t2k sub-buckets based on a simple hash function.
    Currently, the hash function is simply implemented as a modulo
    t2k, with t2k being an implementation dependent value.  Splitting
    the events into sub-buckets makes event cancellations easier.
    </p>
*/
class TwoTierBucket {
public:
    /** The shared parameter indicating the number of sub-buckets to
        be used in each 2-tier bucket.  This value defaults to 32.  It
        is overriden by by command-line argument when ladder queue is
        used.
    */
    static int t2k;

    /** Constructor to create a bucket with fixed number (i.e., t2k)
        of tier-2 lists.
    */
    TwoTierBucket() : subBuckets(t2k), count(0) {}

    /** A move constructor to facilitate moving objects (if needed).

        \param[in,out] src The source object whose data is to be moved
        into this.  The source object does not contain any useful
        information after the move is complete.
    */
    TwoTierBucket(TwoTierBucket&& src) : subBuckets(std::move(src.subBuckets)),
                                         count(std::move(src.count)) {
        // Reset count in source to aid debugging.
        src.count = 0;
    }

    /** The destructor for this class.

        The destructor decreases the reference count on all the events
        in its list to free-up any pending events.
    */
    ~TwoTierBucket();

    /** The hash function used to distribute events into sub-buckets.

        \param[in] sender The sender's ID to be hashed.

        \return The hash value based on sender ID.  The return value,
        say hash, must be in the range 0 <= hash < t2k.
    */
    inline int hash(const xxxx::AgentID sender) const {
        // Use a simple hashing function for now.
        return (sender % t2k);
    }

    /** Add an event to this TwoTier bucket based on sender's ID

        This method is a template specialization to use the sender's
        ID for hashing to find sub-bucket. The event is added to the
        sub-bucket identified using the hash function in this class.

        \param[in] event The event to be added to this bucket.  This
        method does not alter the refernece counts on events (as the
        top-level TwoTierLadderQueue performs the reference count
        management).
    */
    template <bool Sendr, typename std::enable_if<Sendr>::type* = nullptr>
    void push_back(xxxx::Event* event) {
        const size_t subBktIdx = hash(event->getSenderAgentID());
        subBuckets[subBktIdx].push_back(event);
        count++;
    }

    /** Add an event to this TwoTier bucket based on receiver's ID
```

```

        This method is a template specialization to use the receiver’s
        ID for hashing to find sub-bucket. The event is added to the
        sub-bucket identified using the hash function in this class.

        \param[in] event The event to be added to this bucket.  This
        method does not alter the refernece counts on events (as the
        top-level TwoTierLadderQueue performs the reference count
        management).
    */
    template <bool Recvr, typename std::enable_if<!Recvr>::type* = nullptr>
    void push_back(xxxx::Event* event) {
        const size_t subBktIdx = hash(event->getReceiverAgentID());
        subBuckets[subBktIdx].push_back(event);
        count++;
    }

    /** Move all the events from the given two tier bucket into this
        bucket.

        This method moves all the events from the sub-buckets in
        srcBkt to corresponding sub-buckets in this.

        \param[in,out] srcBkt The bucket from where the events are to
        be moved into this bucket.
    */
    void push_back(TwoTierBucket&& srcBkt);

    /** Helper method to move all events from a 2-tier bucket into a
        single list of events.

        This method is a convenience method that is used by the bottom
        tier to combine all the events from various sub-buckets into a
        single list of events.

        \param[out] dest The destination event list to which all the
        events are to added.

        \param[in,out] srcBkt The bucket from where the events are to
        be moved.  After this call the srcBkt will not have any events
        in it.
    */
    static void push_back(BktEventList& dest, TwoTierBucket&& srcBkt);

    /** Obtain the count of events which includes the events in sub-buckets.

        \return The sum of events in all of the sub-buckets.
    */
    size_t size() const {
        ASSERT( getEventCount() == count );
        return count;
    }

    /** Convenience method to determine if the bucket is empty.

        \return This method returns true if this bucket does not
        contain any events in it.
     */
    bool empty() const {
        ASSERT( getEventCount() == count );
        return (count == 0);
    }

    /** Convenience method to remove all events sent by the sender
        at-or-after the given send Time.

        This method removes computes the sub-bucket that contains all
        the events for this sender and removes events from that
        sub-bucket.

        \param[in] sender The sender agent whose events are to be
        removed.

        \param[in] sendTime The time at-or-after which events from the
        sender are to be removed from the given list.
```

```

        \param[in,out] scans Statistics object (if stats is enabled)
        to track number of events scanned.

        \return This method returns the number of events that were
        removed.
    */
    int remove_after(xxxx::AgentID sender, const Time sendTime
                     LQ2T_STATS(COMMA Avg& scans));

    /** Remove all events in this bucket for a given receiver agent
        ID.

        This is a convenience method that removes all events for a
        given receiver agent in this bucket.  This method is used to
        remove events scheduled for an agent, when an agent is removed
        from the scheduler.  This method has to search through all the
        sub-buckets because the condition is based on receiver (and
        not sender).

        \param[in] receiver The receier ID whose events are to be
        removed from the sub-buckets.

        \return This method returns the number of events removed.
    */
    int remove(xxxx::AgentID receiver);

    /** This method is purely for troubleshooting one scenario
        where an event would get stuck in the ladder and not get
        scheduled correctly.

        \param[in] recvTime The time to be used for checking to see if
        sub-buckets have an event before this time.

        \return Returns true if an event before this receiveTime (for
        any agent) is pending in a sub-bucket.
    */
    bool haveBefore(const Time recvTime) const;

    /** Convenience method to remove all events from a given sender
        that were sent at-or-after the given sendTime.

        This method linearly scans the given event list, checks, and
        removes all events that were sent by the sender at-or-after
        the specified send time.

        \note This method assumes unsorted list of events and does not
        preserve order of events if an event is cancelled -- this is
        because Events to be removed are moved to the back and popped
        to reduce deletion time.

        \param[in,out] list The list of events from where all events
        for the sender are to be removed.  This method linearly scans
        through this list.  If events are removed, the order of events
        in the list is not preserved.

        \param[in] sender The sender agent whose events are to be
        removed.

        \param[in] sendTime The time at-or-after which events from the
        sender are to be removed from the given list.

        \return This method returns the number of events that were
        removed.
    */
    static int remove_after(BktEventList& list, xxxx::AgentID sender,
                            const xxxx::Time sendTime);

    /** Remove all events in a given event list for a given receiver
        agent ID.

        This is a convenience/helper method that removes all events
        for a given receiver agent in a given sub-bucket/list.  This
        method is used to remove events scheduled for an agent, when
        an agent is removed from the scheduler.  This method has to
        search through all the events in the list to remove them.

        \param[in] subBkt The sub-bucket or list from where events are
        to be removed.

        \param[in] receiver The receier ID whose events are to be
        removed from the sub-buckets.

        \return This method returns the number of events removed.
    */
    static int remove(BktEventList& list, xxxx::AgentID receiver);

    /** Obtain a reference to the list of sub-buckets in this bucket.

        \return Mutable reference to the list of sub-buckets in this
        bucket.
    */
    SubBucketList& getSubBuckets() { return subBuckets; }

    /** Convenience method to reset count of events in this bucket to
        zero.

        This method is used in TwoTierRung operations to reset the
        events in this bucket to zero, after events have been moved
        out of this bucket.
    */
    void resetCount() {
        count = 0;
    }

protected:
    /** Return sum of events in each sub-bucket.

        This method is used purely for validation/debugging.  This
        method iterates over each sub-bucket in the list and returns
        the actual count of events.  This value must be consistent
        with the value in the count instance variable.

        \return The actual sum of events in various sub-buckets.
    */
    size_t getEventCount() const;

    /** Clear out all the events in this bucket.

        This method clears out all the events in various sub-buckets
        in this 2-tier bucket.  It also sets count to zero.

        \note This method decreases references on any pending events.
    */
    void clear();

private:
    /** The list of tier-2 sub-buckets that contain events distributed
        based on the hash of the receiver's ID.
    */
    SubBucketList subBuckets;

    /** The total number of events in all of the sub-buckets.  This
        information is primary used to quickly respond to the size()
        method calls
    */
    size_t count;
};

/** The class that forms the Top rung of a 2-tier ladder queue.

    The top-rung of the 2-tier ladder queue behaves similar to the
    ladder queue with respect to managing time stamps.  However, the
    organization is different -- events are not stored in a linear
    list.  Instead they are stored in sub-buckets based on a hash of
    the sender agent's ID.

    \note Do not call push_back directly.  Instead use the add method
    in this class to add events.
*/
```

```
class TwoTierTop : public TwoTierBucket {
    friend class TwoTierRung;
    friend class TwoTierLadderQueue;
public:
    /** Construct and initialize top to empty state.

        The constructor uses a convenience method in this class to
        reset the timestamps to zero.
    */
    TwoTierTop() {
        reset();
    }

    /** The destructor

        Currently the destructor has nothing much to do as the base
        class does all of the necessary clean-ups.
     */
    ~TwoTierTop() {}

    /** Method to add events to top and update current minimum and
        maximum time stamp values.

        \param[in] event The event to be added to the top.  This
        pointer cannot be NULL.
    */
    void add(xxxx::Event* event);

    /** Return the current start-time for top.

        \note This value changes when events are added/removed. So
        don't think about caching this value.

        \return The current start time. This value is used for
        scheduling events and creating rungs.
    */
    Time getStartTime() const { return topStart; }

    /** Returns the minimum timestamp of events in this rung.

        \note This value changes when events are added/removed. So
        don't think about caching this value.

        \return The minimum timestamp of events in this rung.
    */
    Time getMinTime() const { return minTS; }

    /** Returns the maximum timestamp of events in this rung.

        \note This value changes when events are added/removed. So
        don't think about caching this value.

        \return The maximum event timestamp in this rung.
    */
    Time getMaxTime() const { return maxTS; }

    /** Convenience method to determine if given time is within the
        <i>current</i> minmum and maximum time.

        \param[in] ts The timestamp value to be checked.

        \return This method returns true if getMinTime() <= ts <=
        getMaxTime().  Otherwise it returns false.
     */
    bool contains(const Time ts) const {
        return (ts >= minTS) && (ts <= maxTS);
    }

    /** Convenience method compute the bucket size for the top-level
        rung of the TwoTierLadder queue.

        \return The suggested bucket width (in terms of time) for the
        top-level rung of the TwoTierLadder.
    */
    double getBucketWidth() const {
        DEBUG(std::cout << "minTS=" << minTS << ", maxTS=" << maxTS
              << ", size=" << size() << std::endl);
        return std::max((maxTS - minTS + size() - 1.0) / size(), 0.01);
    }

protected:
    /** Helper method to reset top either during construction or
        whenever it is emptied to move events into the ladder.

        \param[in] topStart An optional start time for the top rung.
    */
    void reset(const Time topStart = 0);

private:
    /** Instsance variable to track the current minimum timestamp of
        events in top.  This value changes each time a new event is
        added to the top via the add emthod.
    */
    xxxx::Time minTS;

    /** Instsance variable to track the current maximum timestamp of
        events in top.  This value changes each time a new event is
        added to the top via the add emthod.
    */
    xxxx::Time maxTS;

    /** Instsance variable to track the last time top was reset.  This
        is used for debugging/troubleshooting purposes.
    */
    xxxx::Time topStart;
};

/** The bottom most rung of the TwoTierLadder queue.  The bottom rung
    is the same as that of the standard ladder queue.  However, in
    2-tier ladder queue, the size of the bottom has been relaxed.  So
    bottom can be pretty long.  This implies that 2-tier ladder queue
    will not be O(1).  It will be O(n log n).  However, it should
    perform just fine as the ladder queue.

    \note In XXXX we have an API requirement/guarantee that all the
    concurrent events we have will be scheduled simultaneously.  This
    eases agent development in many applications.  Consequently, it is
    imperative that bottom be allowed to be long to contain all
    concurrent events.

    \note Do not use front() / back() to access the first event in
    bottom. Instead use the first_event() method.
*/
class OneTierBottom : public BktEventList {
public:
    /** The default and only constructor.  It does not have any
        special work to do as the base class handles most of the
        tasks.
    */
    OneTierBottom() {}

    /** Add events from a TwoTierBucket into the bottom.

        This method is used to bulk move events from a rung of the
        ladder (or top) into the bottom.  The events are added and
        sorted in preparation for scheduling.

        \param bucket The 2-tier bucket from where events are to be
        moved into the bottom rung.
    */
    void enqueue(TwoTierBucket&& bucket);

    /** Add a single event to the bottom rung.

        This method uses binary-search (O(log n)) to insert an event
        into the bottom.

        \param[in] event The event to be added to the bottom rung.
        This pointer cannot be NULL.  No operations are done on the
        reference-counters in this method.
```

```
    */
    void enqueue(xxxx::Event* event);

    /** Convenience method to dequeue events after a given time.

        \param[in] sender The sender agent whose events are to be
        removed.

        \param[in] sendTime The time at-or-after which events from the
        sender are to be removed from the given list.
    */
    int remove_after(xxxx::AgentID sender, const Time sendTime);

    /** Remove all events for a given receiver agent in the bucket
        encapsulated by this object.

        This is a convenience method that removes all events for a
        given receiver agent in this object.  This method is used to
        remove events scheduled for an agent, when an agent is removed
        from the scheduler.
    */
    int remove(xxxx::AgentID receiver) {
        // Use static convenience method do to this task.
        return TwoTierBucket::remove(*this, receiver);
    }

    /** Convenience method used to dequee the next set of events for
        scheduling.

        This method is used to provide necessary implemntation to
        interface with the XXXX scheduler.  This method dequeues the
        next batch of the concurrent events for processing by a given
        agent.

        \param[out] events The container to which all the events to be
        processed is to be added.
    */
    void dequeueNextAgentEvents(xxxx::EventContainer& events);

    /** Convenience method for debugging/troubleshooting.

        \return The highest timestamp from the events in the bottom.
        If no events are present this method returns TIME_INFINITY.
    */
    xxxx::Time maxTime() const {
        // purely for debugging
        return (!empty() ? front()->getReceiveTime() : TIME_INFINITY);
    }

    /** Convenience method for debugging/troubleshooting.

        \return The minimum timestamp from the events in the bottom.
        If no events are present this method returns TIME_INFINITY.
    */
    xxxx::Time findMinTime() const {
        // purely for debugging
        return (!empty() ? back()->getReceiveTime() : TIME_INFINITY);
    }

    /** Method to determine the range of receive time values currently
        in bottom.  This value is typically used to decide if it is
        worth moving events from bottom into the ladder.

        \return The difference in maximum and minimum receive
        timestamp of events in the bottom.  This value is zero if all
        events have the same receive time.  If the bottom is empty,
        then this method also returns zero.
    */
    xxxx::Time getTimeRange() const {
        if (empty()) {
            return 0;
        }
        return (front()->getReceiveTime() - back()->getReceiveTime());
    }

    /** Determine bucket width to move bottom into ladder.

        This method is invoked only when the ladder is empty and the
        bottom is long and needs to be moved into the ladder.  This
        method must compute and return the preferred bucket width.

        \note If the bottom is empty this method returns bucket width
        of 0.
     */
    double getBucketWidth() const;

    /** Convenience method to check if the entries in the bottom are
        sorted correctly.  This method is purely used for
        troubleshooting/debugging.
    */
    void validate() const;

    /** Event comparison function used by various structures in ladder
        queue.

        \param[in] lhs The left-hand-side event for comparison.  The
        pointer cannot be NULL.

        \param[in] lhs The right-hand-side event for comparison.  The
        pointer cannot be NULL.

        \return This method returns true if lhs is less than rhs.
        That is, lhs should be scheduled before rhs.
    */
    static inline bool compare(const xxxx::Event* const lhs,
                               const xxxx::Event* const rhs) {
        return ((lhs->getReceiveTime() > rhs->getReceiveTime()) ||
                ((lhs->getReceiveTime() == rhs->getReceiveTime() &&
                  (lhs->getReceiverAgentID() > rhs->getReceiverAgentID()))));
    }

    /** Convenience method to check to see if bottom has events before
        the specified receive time.

        This method is used for troubleshooting/debugging only.

        \param[in] recvTime The receive time for checking.

        \return Returns true if an event before this receiveTime (for
        any agent) is pending in the bottom rung.
    */
    bool haveBefore(const Time recvTime) const {
        return (findMinTime() <= recvTime);
    }

    /** Convenience method to consistently access the first event in
        the bottom, consistent with the way bottom is sorted.

        \note Calling this method when the bottom is empty has
        undefined behavior.

        \return The next event with the lowest time stamp.
    */
    xxxx::Event* first_event() {
        return back();
    }

protected:
    // Currently this class does not have any protected members.

private:
    // Currently this class does not have any private members
};

/** Class that represents one rung in the 2-tier ladder queue.

    The 2-tier rung uses the same strategy for receive time-based
    bucket creation as the regular ladder queue.  However, the
    organization of each bucket is different -- events are not stored
    in a linear list.  Instead they are stored in sub-buckets based on
```

```
    a hash of the sender agent's ID.
*/
class TwoTierRung {
public:
    /** The constructor to create an empty rung.

        The constructor merely initializes all the instance variables
        to default initial values to create an empty rung.
    */
    TwoTierRung() : rStartTS(TIME_INFINITY), rCurrTS(TIME_INFINITY),
                    bucketWidth(0), currBucket(0), rungEventCount(0) {
        LQ2T_STATS(maxBkts = 0);
    }

    /** A move constructor required to quickly move rungs in a ladder
        to shrink/grow it.

        \param[in] src The source rung from where events are to be
        copied.
    */
    TwoTierRung(TwoTierRung&& src) :
        rStartTS(src.rStartTS), rCurrTS(src.rCurrTS),
        bucketWidth(src.bucketWidth), currBucket(src.currBucket),
        bucketList(std::move(src.bucketList)),
        rungEventCount(src.rungEventCount) {
        LQ2T_STATS(maxBkts = src.maxBkts);
    }

    /** Convenience constructor to create a rung using events from the
        top rung.

        This is a delegating constructor that delegates the actual
        tasks to the overloaded constructor.

        \param[in] top The top bucket from where the events are to be
        created.
    */
    explicit TwoTierRung(TwoTierTop&& top) :
        TwoTierRung(std::move(top), top.getMinTime(),
                    top.getBucketWidth()) {
            // Reset of top counters etc. is done by caller in
            // TwoTierLadderQueue::populateBottom()
    }

    /** Convenience constructor to create a rung with events from a
        given bucket.

        \param[in,out] bkt The bucket from where events are to be
        moved into this newly created rung.  After this operation data
        in the bucket is cleared.

        \param[in] rStart The start time for this rung.

        \param[in] bucketWidth The delta in receive time for each
        bucket in this rung.  The bucketWidth must be > 0.
    */
    TwoTierRung(TwoTierBucket&& bkt, const Time rStart,
                const double bucketWidth) : rungEventCount(0) {
        move(std::move(bkt), rStart, bucketWidth);
    }

    /** Convenience method initialize a rung by moving events from a
        given bucket.

        It is assumed that this rung is empty prior to this operation.

        \param[in,out] bkt The bucket from where events are to be
        moved into this rung.  After this operation data in the bucket
        is cleared.

        \param[in] rStart The start time for this rung.

        \param[in] bucketWidth The delta in receive time for each
        bucket in this rung.  The bucketWidth must be > 0.
    */
    void move(TwoTierBucket&& bucket, const Time rStart,
              const double bucketWidth);

    /** Convenience constructor to create a rung with events from the
        bottom rung.

        This operation is used to redistribute bottom to the ladder
        ensures that the bottom does not get too long.

        \param[in,out] bottom The bottom rung from where events are to
        be moved into this newly created rung.  After this operation
        bottom will be empty.

        \param[in] rStart The start time for this rung.

        \param[in] bucketWidth The delta in receive time for each
        bucket in this rung.  The bucketWidth must be > 0.
    */
    TwoTierRung(OneTierBottom&& bottom, const Time rStart,
                const double bucketWidth) : rungEventCount(0) {
        move(std::move(bottom), rStart, bucketWidth);
    }

    /** Convenience method to create a rung with events from the
        bottom rung.

        This operation is used to redistribute bottom to the ladder
        ensures that the bottom does not get too long.  This method
        assumes that the this rung is empty to begin with.

        \param[in,out] bottom The bottom rung from where events are to
        be moved into this newly created rung.  After this operation
        bottom will be empty.

        \param[in] rStart The start time for this rung.

        \param[in] bucketWidth The delta in receive time for each
        bucket in this rung.  The bucketWidth must be > 0.
    */
    void move(OneTierBottom&& bottom, const Time rStart,
              const double bucketWidth);

    /** Remove the next bucket in this rung for moving to another rung
        in the ladder.

        This method must be used to remove the next bucket from this
        rung.  The bucket is logically removed (or moved) out of this
        rung.

        \param[out] bktTime The simulation receive time associated
        with the bucket being moved out.
    */
    TwoTierBucket&& removeNextBucket(xxxx::Time& bktTime);

    /** Determine if this rung is empty.

        This is a convenience method that is used to determine if this
        rung contains any events to be processed.

        \return This method returns true if the rung does not have any
        events -- i.e., when the rung is empty.
    */
    bool empty() const { return (rungEventCount == 0); }

    /** Add an event to suitable bucket in this rung.

        This method computes a bucket index (based on equation #2 in
        LQ paper) using the formula:

        \code
        size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
        \endcode

        \param[in] event The event to be added to a suitable bucket in
        this rung.
```

```
    */
    void enqueue(xxxx::Event* event);

    /** Obtain the start time for this rung.

        This method returns the rung starting time that was set when
        this rung was created.

        \return The starting time of this rung that determines the
        lowest timestamp event that can be added to this rung.
    */
    xxxx::Time getStartTime() const { return rStartTS; }

    /** Obtain the bucket width (i.e., difference in receive times for
        adjacent buckets) for this rung.

        This method returns the bucket width that was set when this
        rung was created.

        \return The bucket with for this rung.
    */
    double getBucketWidth() const { return bucketWidth; }

    /** The current bucket value in this ladder queue.

        The current minimum time of events that can be added to this
        rung of the ladder eueue.

        \return The minimum timestamp of events that can be added to
        the rung of this ladder queue.
    */
    xxxx::Time getCurrTime() const {
        return rCurrTS;
    }

    /** The maximum receive time value of event that can be added to
        this rung.

        \return The maximum receive time of an event that can be added
        to a bucket in this rung.
    */
    xxxx::Time getMaxRungTime() const {
        return rStartTS + (bucketList.size() * bucketWidth);
    }

    /** Convenience method to determine if a given event can be added
        to this rung.

        \param[in] event The event whose receive time is to be used to
        check to see if it can be added to this ladder.

        \return Returns true if the event can be added to this rung.
        Otherwise it returns false.
    */
    bool canContain(xxxx::Event* event) const;

    /** Remove all events from the given sender sent at-or-after the
        specified send time from all buckets in this rung.

        This method linearly scans the buckets, checks, and removes
        all events that were sent by the sender at-or-after the
        specified send time.

        \param[in] sender The sender agent whose events are to be
        removed.

        \param[in] sendTime The time at-or-after which events from the
        sender are to be removed from the given list.

        \param[out] ceScanRung The stats object to be updated with
        number of events scanned in the buckets in this rung.

        \return This method returns the total number of events that
        were removed from this rung.
    */
    int remove_after(xxxx::AgentID sender, const Time sendTime
                     LQ2T_STATS(COMMA Avg& ceScanRung));

    /** Remove all events for a given receiver agent in this rung.

        This is a convenience method that removes all events for a
        given receiver agent in this rung.  This method is used to
        remove events scheduled for an agent, when an agent is removed
        from the scheduler.

        \param[in] receiver The receiving agent ID whose events are to
        be removed from all the buckets in this rugn.

        \param[out] ceScanRung The stats object to be updated with
        number of events scanned in the buckets in this rung.
    */
    int remove(xxxx::AgentID receiver
               LQ2T_STATS(COMMA Avg& ceScanRung));

    /** Check to ensure that the number of events in various buckets
        matches the count instance variable.

        This method is used only for troubleshooting/debugging
        purposes.  If counts don't match then assert fails in this
        method causing the simulation to abort.
    */
    void validateEventCounts() const;

    /** Print a user-friendly version of the events in this queue.

        Currently this method is not implemented.
    */
    void prettyPrint(std::ostream& os) const;

    /** Update the statistics object with data from this rung.

        \param[out] avgBktCnt Update the average number of buckets in
        this rung.
    */
    void updateStats(Avg& avgBktCnt) const;

    /** Convenience method to determine if the current bucket in this
        rung is empty.

        \return This method returns true if the current bucket in this
        rung is empty.
    */
    bool isCurrBucketEmpty() const {
        return (currBucket >= bucketList.size() ||
                bucketList[currBucket].empty());
    }

    /** This method is purely for troubleshooting one scenario where
        an event would get stuck in the ladder and not get scheduled
        correctly.

        \param[in] recvTime The time to be used for checking to see if
        sub-buckets have an event before this time.

        \return Returns true if an event before this receiveTime (for
        any agent) is pending in a sub-bucket.
    */
    bool haveBefore(const Time recvTime) const;

protected:
    // Currently this class does not have any protected members.

private:
    /** The lowest timestamp event that can be added to this rung.
        This value is set when a rung is created and is never changed
        during the lifetime of this rung.
    */
    xxxx::Time rStartTS;

    /** The timestamp of the lowest event that can be currently added
```

```
        to this rung.  This value logically starts with rStartTS and
        grows to the time stamp of last bucket in this rung as buckets
        are dequeued from this rung.
    */
    xxxx::Time rCurrTS;

    /** The width of the bucket in simulation receive time
        differences.  This value can be fractional.
    */
    double bucketWidth;

    /** The index of the current bucket on this rung to which events
        can be added.  This is also the next bucket that will be
        dequeued from the rung.
    */
    size_t currBucket;

    /** The deque containing the set of vectors in this bucket list.
     */
    std::deque<TwoTierBucket> bucketList;

    /** Total number of events still present in this rung.  This is
        used to report size and check for empty quickly.
    */
    int rungEventCount;

    /** Statistics object to track the maximum number of buckets used
        in this rung */
    LQ2T_STATS(size_t maxBkts);
};

/** The top-level 2-tier ladder queue

    <p>This class represents the top-level 2-tier ladder queue class
    that interfaces with the XXXX scheduler.  This class implements
    the top-level logic associated with ladder queue to enqueue,
    dequeue, and cancel events from the ladder queue.</p>

    <p>The logic for most of the operations is consistent with those
    proposed by the Tang et. al, except for the following:

    <ol>

    <li>The size of the bottom is not restricted.  So events are never
    moved from bottom back into the ladder.</li>

    <li> The number of buckets in a rung is restricted to 100</li>

    </ol>

    </p>
*/
class TwoTierLadderQueue : public EventQueue {
public:
    /** The constructor that creates an empty ladder queue.

        The constructor also initializes various statistics variables
        used by the this queue to report detailed statistics about its
        operations at the end of simulation.
     */
    TwoTierLadderQueue() : EventQueue("LadderQueue"), nRung(0),
                           ladderEventCount(0) {
        ladder.reserve(MaxRungs);
        LQ2T_STATS(ceTop    = ceLadder   = ceBot  = 0);
        LQ2T_STATS(insTop   = insLadder  = insBot = 0);
        LQ2T_STATS(maxRungs = maxBotSize = 0);
    }

    /** The destructor.

        Currently the destructor does not have anything special to do
        as the different encapsulated objects handle all the necessary
        clean-up.
    */
    ~TwoTierLadderQueue() {}

    /** Enqueue an event into the laadder queue.

        Depending on the scenario the event is appropriately added to
        one of: top, ladder rung, or the bottom.

        \param[in] e The event to be enqueued for scheduling in the
        ladder queue.
    */
    void enqueue(xxxx::Event* e);

    /** Cancel all events from a given sender that were sent
        at-or-after the specified send time.

        This method essentially calls the corresponding method(s) in
        top, rung, and bottom to cancel pending events.

        \param[in] sender The sender agent whose events are to be
        removed.

        \param[in] sendTime The time at-or-after which events from the
        sender are to be removed from the given list.

        \return This method returns the number of events that were
        removed.
    */
    int remove_after(xxxx::AgentID sender, const Time sendTime);

    /** Determine if the ladder queue is empty.

        Implements the interface method used by XXXX::Scheduler.

        \return Returns true if top, ladder, and bottom are all empty
        -- i.e., there are no pending events.
    */
    virtual bool empty() {
        return top.empty() && (ladderEventCount == 0) &&  bottom.empty();
    }

    /** Implementation for method used by XXXX::Scheduler.

        This method is called by XXXX kernel to inform the scheduler
        queue about an agent being added during initialization.  The
        ladder queue does not utilize this information and
        consequently this method does not have any special operation
        to perform.

        \param[in] agent The agent being added.  This pointer is not
        really used.

        \return This method simply returns nullptr as the ladder queue
        does not use any cross references in xxxx::Agent for its
        operations.
    */
    virtual void* addAgent(xxxx::Agent* agent);

    /** Remove an agent just before simulation completes.

        This method is invoked by the XXXX kernel to inform that an
        agent is being removed.  This method removes all pending
        events for the specified agent from the ladder queue.

        \param[in] agent The agent whose sender ID is used to remove
        all pending events in the top, rungs, and bottom.
    */
    virtual void removeAgent(xxxx::Agent* agent);

    /** Implement interface method to peek at the next event to
        schedule.

        \note In order to enable peeking of the front event, the
        bottom may need to get populated.

        \return A pointer to the next event to schedule (if any).  The
        event is not dequeued.
```

```
     */
    virtual xxxx::Event* front();

    /** This method is used to provide necessary implemntation to
        interface with the XXXX scheduler.  This method dequeues the
        next batch of the concurrent events for processing by a given
        agent.

        \param[out] events The container to which all the events to be
        processed is to be added.
    */
    virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);

    /** Add an event to be scheduled to this ladder queue.

        This method implements the core API used by agents to schedule
        events for each other.

        \param[in] agent The receiver agent for which the event is
        scheduled.  This pointer is not used.

        \param[in] event The event to be scheduled. This simply calls
        the overloaded enqueue method.  The reference count on the
        event is increased by this method to account for this event
        being present in the ladder queue.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);

    /** Enqueue a batch of events

        This API to schedule a block of events.  This API is typically
        used after a rollback.

        \param[in] agent The receiver agent for which the event is
        scheduled.  This pointer is not used.

        \param[in] events The list of events to be scheduled. This
        simply calls the overloaded enqueue method to enqueue one
        event at a time.
    */
    virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);

    /** Implement XXXX kernel API to cancel all events sent by a given
        agent after a given time.

        \param[in] dest The destination agent whose events are to be
        cancelled.

        \param[in] sender The sender agent ID whose events are to be
        cancelled.

        \param[in] sentTime The send time at-or-after which all events
        from the sender are to be cancelled.
    */
    virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
                           const xxxx::Time sentTime);

    /** Print a human understandable version of the events in this
        queue.

        currently this method is not implemented.
     */
    virtual void prettyPrint(std::ostream& os) const;


    /** Convenience method to check to see if ladder queue has events
        before the specified receive time.

        This method is used for troubleshooting/debugging only.

        \param[in] recvTime The receive time for checking.

        \return Returns true if an event before this receiveTime (for
        any agent) is pending in the bottom rung.
    */
    bool haveBefore(const Time recvTime,
                    const bool checkBottom = false) const;

    /** Method to report aggregate statistics.

        This method is invoked at the end of simulation after all
        agents on this rank have been finalized.  This method is meant
        to report any aggregate statistics from this queue.  This
        method writes statistics only if LQ2T_STATS macro is enabled.

        \param[out] os The output stream to which the statistics are
        to be written.
    */
    virtual void reportStats(std::ostream& os);

    /** The maximum number of rungs that are normally created in the
        ladder queue.  The default value for this set to 8 based on
        the value suggested by Tang et. al. in the original Ladder
        Queue paper.  However, this value can make some difference in
        the overall performance and possibly fine tuned to suit the
        application needs based on the concurrency and number of
        events in the model.
    */
    static size_t MaxRungs;

protected:
    /** Check and create rungs in the ladder and return the next
        bucket of events from the ladder.

        This method implements the corresponding recurseRung method
        from the LQ paper. Refer to the paper for the details.
    */
    TwoTierBucket&& recurseRung();

    /** This is a convenience method that is used to move events from
        the ladder into bottom.

        This method moves events from teh current bucket in the last
        rung of the ladder into the bottom.  If this method is called
        when bottom is not empty, it does not perform any operation
        and returns immediately.  If the ladder does not have any
        events, but the top has events, then this method first moves
        events from top-rung into the ladder and then removes events
        from the last rung into the bottom-rung.
    */
    void populateBottom();

    /** Method to create a new ladder rung from the current bottom.

        This method should be called only when the following 2
        conditions are met:

        1. Length of bottom is > LQ2T_THRESH

        2. The bottom has events that are at different time stamps --
           that is bottom.getTimeRange() > 0.

        \return This method returns the index of the rung created so
        that the caller can readily work with that rung.
    */
    int createRungFromBottom();

private:
    TwoTierTop top;

    /** The ladder in the queue.  The lader consists of a set of
        rungs.  The currently used rung in the ladder is indicated by
        the nRung instance variable.  If the ladder is empty, then
        nRung is (or should be) 0
    */
    std::vector<TwoTierRung> ladder;

    /** The currently used last rung in the ladder queue.  If the
        ladder is empty, then nRung is (or should be) 0.  Otherwise
        this value is (or should be) in the range 0 < nRung <=
```

```
        ladder.size().  Rungs below nRung are not used and they do not
        contain any events to be scheduled.
    */
    size_t nRung;

    /** Instance variable to track the current number of pending
        events in all the rungs of the ladder.  This is a convenience
        instance variable to quickly detect pending events in the
        ladder without having to iterate through each rung.
    */
    int ladderEventCount;

    OneTierBottom bottom;

    LQ2T_STATS(Avg ceTop);
    LQ2T_STATS(Avg ceBot);
    LQ2T_STATS(Avg ceLadder);

    LQ2T_STATS(Avg ceScanTop);
    LQ2T_STATS(Avg ceScanLadder);

    /** The ceScanBot statistic tracks size of bottom rung scanned
        when at one (or more) events were canceled from bottom.
    */
    LQ2T_STATS(Avg ceScanBot);

    /** The ceNoCanScanBot statistic tracks size of bottom rung
        scanned but did not cancel any events.
    */
    LQ2T_STATS(Avg ceNoCanScanBot);

    LQ2T_STATS(int insTop);
    LQ2T_STATS(int insLadder);
    LQ2T_STATS(int insBot);
    LQ2T_STATS(size_t maxRungs);
    LQ2T_STATS(Avg avgBktCnt);
    LQ2T_STATS(Avg botLen);
    LQ2T_STATS(Avg avgBktWidth);

    /** Gague to track the number of events and times bottom was
        redistributed to the last rung of the ladder.

        Redistributing bottom to the ladder ensures that the bottom
        does not get too long.  But it is an expensive operation
        because all the sorting that was done is lost.  So it is a
        balance and we track and report this number for reference.
    */
    LQ2T_STATS(Avg botToRung);

    /** Gauge to track the maximum length of bottom.  The length of
        bottom plays an important role in the overall performance of
        the ladder queue.
    */
    LQ2T_STATS(size_t maxBotSize);
};


END_NAMESPACE(xxxx)

#endif
```

```
#ifndef TWO_TIER_LADDER_QUEUE_CPP
#define TWO_TIER_LADDER_QUEUE_CPP

#include <algorithm>
#include <functional>
#include "TwoTierLadderQueue.h"

// The maximum number of buckets 1 rung can have.
#define MAX_BUCKETS 100

/** The number of sub-buckets to be used in each 2-tier bucket */
int xxxx::TwoTierBucket::t2k = 32;

// ----------------------[ TwoTierBucket methods ]-----------------------

xxxx::TwoTierBucket::~TwoTierBucket() {
    clear();
}

void
xxxx::TwoTierBucket::clear() {
    for (BktEventList& subBkt : subBuckets) {
        for (xxxx::Event* event : subBkt) {
            event->decreaseReference();
        }
    }
    count = 0;
}

void
xxxx::TwoTierBucket::push_back(TwoTierBucket&& srcBkt) {
    ASSERT(srcBkt.subBuckets.size() == (size_t) t2k);
    ASSERT(srcBkt.subBuckets.size() == subBuckets.size());
    // Move evens from srcBkt into corresponding sub-buckets
    for (int idx = 0; (idx < t2k); idx++) {
        // Obtain reference to subbucket to be moved.
        BktEventList& src  = srcBkt.subBuckets[idx];
        BktEventList& dest = subBuckets[idx];
        // Move events from src to dest.
        dest.insert(dest.end(), src.begin(), src.end());
        // Update counters (also used to troubleshooting).
        count += src.size();
        // Clear out the source as events have been logically moved
        // out of it.
        src.clear();
    }
}

void
xxxx::TwoTierBucket::push_back(BktEventList& dest, TwoTierBucket&& srcBkt) {
    // Move all entries from each sub-bucket in srcBkt to the end of dest.
    for (BktEventList& subBkt : srcBkt.subBuckets) {
        dest.insert(dest.end(), subBkt.begin(), subBkt.end());
        subBkt.clear();
    }
    // Reset count as part of move semantics
    srcBkt.count = 0;
}

int
xxxx::TwoTierBucket::remove_after(xxxx::AgentID sender, const Time sendTime
                                  LQ2T_STATS(COMMA Avg& scans)) {
    const size_t subBktIdx = hash(sender);
    LQ2T_STATS(scans += subBuckets[subBktIdx].size());
    int removedCount = remove_after(subBuckets[subBktIdx], sender, sendTime);
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// Helper method to remove events from a sub-bucket.
int
xxxx::TwoTierBucket::remove_after(BktEventList& list, xxxx::AgentID sender,
                                  const Time sendTime) {
    size_t removedCount = 0;
    size_t curr = 0;
```

```
    while (curr < list.size()) {
        xxxx::Event* const event = list[curr];
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // To minimize removal time replace entry with last one
            // and pop the last entry off.
            list[curr] = list.back();
            list.pop_back();
        } else {
            curr++;  // on to the next event in the list
        }
    }
    return removedCount;
}

// This method is not performance critical as it is only called once
// at the end of simulation.
int
xxxx::TwoTierBucket::remove(xxxx::AgentID receiver) {
    size_t removedCount = 0;
    // Remove events from each sub-bucket.
    for (BktEventList& list : subBuckets) {
        // Use helper method to remove events.
        removedCount += remove(list, receiver);
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// static helper method also used by OneTierBottom.  This is called
// few times at the end of simulation.  So it is not performance
// critical.
int
xxxx::TwoTierBucket::remove(BktEventList& list, xxxx::AgentID receiver) {
    int removedCount = 0;  // statistics tracking
    // Linear scan through events in a given sub-bucket
    BktEventList::iterator curr = list.begin();
    while (curr != list.end()) {
        if ((*curr)->getReceiverAgentID() == receiver) {
            (*curr)->decreaseReference();
            curr = list.erase(curr);
            removedCount++;
        } else {
            curr++;
        }
    }
    return removedCount;  // let caller know the events removed
}

// This method is not performance critical.  It is used only for
// troubleshooting/debugging
bool
xxxx::TwoTierBucket::haveBefore(const Time recvTime) const {
    for (const BktEventList& list : subBuckets) {
        for (const xxxx::Event* const event : list) {
            if (event->getReceiveTime() <= recvTime) {
                return true;
            }
        }
    }
    return false;
}

// Actually counts events in each bucket for validation purposes.
// This method is not performance critical.  It is used only for
// troubleshooting/debugging
size_t
xxxx::TwoTierBucket::getEventCount() const {
    int sum = 0;
    for (const BktEventList& subBkt : subBuckets) {
        sum += subBkt.size();
    }
```

```cpp
    return sum;  // total number of events
}

// ------------------------[ TwoTierTop methods ]------------------------

// Helper method called from constructor and when events are moved
// from top into ladder.
void
xxxx::TwoTierTop::reset(const Time startTime) {
    minTS    = TIME_INFINITY;
    maxTS    = 0;
    topStart = startTime;
    clear();
}

void
xxxx::TwoTierTop::add(xxxx::Event* event) {
    push_back<SenderID>(event);   // Call base-class method.
    // Update running timestamps.
    minTS = std::min(minTS, event->getReceiveTime());
    maxTS = std::max(maxTS, event->getReceiveTime());
}

// ------------------------[ OneTierBottom methods ]------------------------

void
xxxx::OneTierBottom::enqueue(xxxx::TwoTierBucket&& bucket) {
    // Move events from bucket into the bottom.
    TwoTierBucket::push_back(*this, std::move(bucket));
    // Now sort the whole bottom O(n*log(n)) operation
    std::sort(begin(), end(), OneTierBottom::compare);
    DEBUG(validate());
}

void
xxxx::OneTierBottom::enqueue(xxxx::Event* event) {
    BktEventList::iterator iter =
        std::lower_bound(begin(), end(), event, compare);
    insert(iter, event);  // base class method.
    DEBUG(validate());
}

void
xxxx::OneTierBottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (empty()) {
        return;  // no events to provide
    }
    // Reference information used for checking in the loop below.
    const xxxx::Event*  nextEvt  = back();
    const xxxx::AgentID receiver = nextEvt->getReceiverAgentID();
    const xxxx::Time    currTime = nextEvt->getReceiveTime();
    // Move all events from bottom to the events-container for scheduling.
    do {
        // Back event is the lowest timestamp (or highest priority)
        // based on sorting order in OneTierBottom::compare()
        xxxx::Event* event = back();
        events.push_back(event);
        pop_back();   // remove from bottom.
        // erase(begin());
        // Check and work with the next event.
        nextEvt = (!empty() ? back() : NULL);
        DEBUG(std::cout << "Delivering: " << *event << std::endl);
    } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
             TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
    DEBUG(validate());
}

double
xxxx::OneTierBottom::getBucketWidth() const {
    if (empty()) {
        return 0;
    }
    ASSERT(front() != NULL);
    ASSERT(back()  != NULL);
    // Assumes that bottom is sorted with the lowest timestamp at the
```

```cpp
    // end for fast pop_back
    const double maxTS = front()->getReceiveTime();
    const double minTS = back()->getReceiveTime();
    return (maxTS - minTS + size() - 1.0) / size();
}

int
xxxx::OneTierBottom::remove_after(xxxx::AgentID sender, const Time sendTime) {
    // Since bucket is sorted we can shortcircuit scan if last event’s
    // time is less-or-equal to sendTime.
    if (empty() || (sendTime >= front()->getReceiveTime())) {
        return -1;  // Since bucket does not have events to be cancelled.
    }
    size_t removedCount = 0;
    iterator curr = begin();
    while (curr != end()) {
        xxxx::Event* const event = *curr;
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // In sorted mode we have to preserve the order. So
            // cannot swap & pop in this situation
            curr = erase(curr);
        } else {
            curr++;  // onto next event
        }
    }
    return removedCount;
}

// This method is used only for debugging. So it is not performance
// critical.
void
xxxx::OneTierBottom::validate() const {
    if (empty()) {
        return;  // yes. bottom is valid.
    }
    // Ensure events are sorted in timestamp order.
    BktEventList::const_iterator next = cbegin();
    BktEventList::const_iterator prev = next++;
    while ((next != cend()) &&
           ((*next)->getReceiveTime() >= (*prev)->getReceiveTime())) {
        prev = next++;
    }
    if (next != cend()) {
        std::cout << "Error in LadderQueue.Bottom: Event " << **next
                  << " was found after " << **prev << std::endl;
    }
    ASSERT( next == cend() );
}

// ------------------------[ TwoTierRung methods ]------------------------

void
xxxx::TwoTierRung::move(TwoTierBucket&& bkt, const Time minTS,
                        const double bktWidth) {
    // Setup starting & current timestamp for this rung.
    rStartTS = rCurrTS = minTS;
    // Ensure bucket width is not ridiculously small
    bucketWidth = bktWidth;
    currBucket  = 0;  // current bucket in this rung.
    // Initialize variable to track maximum bucket count
    LQ2T_STATS(maxBkts = 0);
    DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
    ASSERT(bucketWidth > 0);
    ASSERT(rungEventCount == 0);
    // Move events from given bucket into buckets in this Rung.
    DEBUG(std::cout << "Adding " << bkt.size() << " events to rung\n");
    for (BktEventList& list : bkt.getSubBuckets()) {
        // Add all events from sub-buckets to various buckets in this rung.
        while (!list.empty()) {
            // Remove event from the top linked list.
            xxxx::Event* event = list.back();
            list.pop_back();
            // Add to the appropriate bucket in this rung using a
            // helper method in this class.
            enqueue(event);
        }
    }
    // Reset bucket counters as we have moved all the events out
    bkt.resetCount();
    DEBUG(validateEventCounts());
}

void
xxxx::TwoTierRung::move(OneTierBottom&& bottom, const Time rStart,
                        const double bktWidth) {
    rStartTS = rCurrTS = rStart;
    // Ensure bucket width is not ridiculously small
    bucketWidth = bktWidth;
    currBucket  = 0;  // current bucket in this rung.
    ASSERT(rungEventCount == 0);
    // Initialize variable to track maximum bucket count
    LQ2T_STATS(maxBkts = 0);
    DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
    ASSERT(bucketWidth > 0);
    ASSERT(rungEventCount == 0);
    // Move events from bottom into buckets in this Rung.
    DEBUG(std::cout << "Adding " << bottom.size() << " events to rung\n");
    for (xxxx::Event* event : bottom) {
        // Add to the appropriate bucket in this rung using a
        // helper method in this class.
        enqueue(event);
    }
    // Finally clear out the events in bottom.
    bottom.clear();
    DEBUG(validateEventCounts());
}


bool
xxxx::TwoTierRung::canContain(xxxx::Event* event) const {
    const xxxx::Time recvTime = event−>getReceiveTime();
    const int bucketNum = (recvTime − rStartTS) / bucketWidth;
    return ((bucketNum >= (int) currBucket) && (recvTime >= rStartTS));
}

void
xxxx::TwoTierRung::enqueue(xxxx::Event* event) {
    ASSERT(event != NULL);
    ASSERT(event−>getReceiveTime() >= getCurrTime());
    // Compute bucket for this event based on equation #2 in LQ paper.
    size_t bucketNum = (event−>getReceiveTime() − rStartTS) / bucketWidth;
    ASSERT(bucketNum >= currBucket);
    if (bucketNum >= bucketList.size()) {
        // Ensure bucket list of sufficient size
        bucketList.resize(bucketNum + 1);
        // update variable to track maximum bucket count
        LQ2T_STATS(maxBkts = std::max(maxBkts, bucketList.size()));
    }
    ASSERT(bucketNum < bucketList.size());
    // Add event into appropriate bucket
    bucketList[bucketNum].push_back<SenderID>(event);
    // Track number of events added to this Rung
    rungEventCount++;
}

xxxx::TwoTierBucket&&
xxxx::TwoTierRung::removeNextBucket(xxxx::Time& bktTime) {
    ASSERT(!empty());
    ASSERT(currBucket < bucketList.size());
    // Find next non−empty bucket in this rung (there has to be one as
    // the previous asserts passed necessary checks)
    while ((currBucket < bucketList.size()) && bucketList[currBucket].empty()) {
        currBucket++;
    }
    DEBUG(validateEventCounts());
    ASSERT(currBucket < bucketList.size());
```

```
    ASSERT(!bucketList[currBucket].empty());
    // Track events that will be removed when this method returns
    rungEventCount −= bucketList[currBucket].size();
    ASSERT(rungEventCount >= 0);
    // Save information about the bucket to be removed & returned.
    const int retBkt = currBucket;
    bktTime = rStartTS + (retBkt * bucketWidth);
    // Advance current bucket to next time.
    currBucket++;
    rCurrTS = rStartTS + (currBucket * bucketWidth);
    // Sanity check on counters...
    if (currBucket >= bucketList.size()) {
        ASSERT(rungEventCount == 0);
    }
    return std::move(bucketList[retBkt]);
}

int
xxxx::TwoTierRung::remove_after(xxxx::AgentID sender, const Time sendTime
                                LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty() || (sendTime > getMaxRungTime())) {
        return 0;  // no events removed.
    }
    // Check each bucket in this rung and cancel out events.
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty() &&
            (rStartTS + (bktNum + 1) * bucketWidth) >= sendTime) {
            // Have the bucket remove necessary event(s) and update stats
            numRemoved +=
                bucketList[bktNum].remove_after(sender, sendTime
                                                LQ2T_STATS(COMMA ceScanRung));
        }
    }
    // Update events left in this rung.
    rungEventCount −= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

int
xxxx::TwoTierRung::remove(xxxx::AgentID receiver
                          LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty()) {
        return 0;  // no events to be removed.
    }
    // Have each bucket in the rung remove events
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty()) {
            // This stat needs to be tracked by the bucket and not here.
            LQ2T_STATS(ceScanRung += bucketList[bktNum].size());
            // Remove appropriate set of events.
            numRemoved += bucketList[bktNum].remove(receiver);
        }
    }
    rungEventCount −= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

void
xxxx::TwoTierRung::validateEventCounts() const {
    int numEvents = 0;
    for (const auto& bucket : bucketList)  {
        numEvents += bucket.size();
    }
    if (numEvents != rungEventCount) {
        DEBUG(std::cout << "Rung event count mismatch! Expecting: "
                        << rungEventCount << " events, but found: "
                        << numEvents << "." << std::endl);
        ASSERT(numEvents == rungEventCount);
    }
}
```

```
// Method called just before a rung is removed from the ladder queue.
void
xxxx::TwoTierRung::updateStats(Avg& avgBktCnt) const {
    LQ2T_STATS(avgBktCnt += maxBkts);
}

// This method is used only for troubleshooting/debugging purposes.
bool
xxxx::TwoTierRung::haveBefore(const Time recvTime) const {
    for (size_t i = 0; (i < bucketList.size()); i++) {
        if (bucketList[i].haveBefore(recvTime)) {
            return true;
        }
    }
    return false;
}

void
xxxx::TwoTierRung::prettyPrint(std::ostream& os) const {
    // Compute minimum, maximum, empty, and average bucket sizes.
    size_t minBkt = −1U, maxBkt = 0, emptyBkt = 0, sizeSum = 0;
    for (const TwoTierBucket& bkt : bucketList) {
        if (!bkt.empty()) {
            minBkt   = std::min(minBkt, bkt.size());
            maxBkt   = std::max(maxBkt, bkt.size());
            sizeSum += bkt.size();
        } else {
            emptyBkt++;
        }
    }
    double avgBktSz = sizeSum / (double) (bucketList.size() − emptyBkt);
    os << "start time="    << rStartTS     << ", curr time=" << rCurrTS
       << ", bkt. width=" << bucketWidth << ", bkt count=" << bucketList.size()
       << ", curr buckt=" << currBucket  << ", events="     << rungEventCount
       << ", min bkt="     << minBkt       << ", maxBkt="     << maxBkt
       << ", empty bkt="  << emptyBkt     << ", avg size="  << avgBktSz
       << std::endl;
}

// −−−−−−−−−−−−−−−−−−−−[ TwoTierLadderQueue methods ]−−−−−−−−−−−−−−−−−−−−

// The maximum number of rungs typically allowed in the ladder.  This
// value is set to 8 by default based on Tang et. al.  It can be set
// via command−line parameter −−lq−max−rungs 8.
size_t xxxx::TwoTierLadderQueue::MaxRungs = 8;

void
xxxx::TwoTierLadderQueue::reportStats(std::ostream& os) {
    UNUSED_PARAM(os);
    LQ2T_STATS({
            // Collect final bucket counts from the ladder
            for (size_t i = 0; (i < nRung); i++) {
                ladder[i].updateStats(avgBktCnt);
            }
            // Compute net number of compares for ladderQ
            // const long comps = log2(botLen.getMean()) * botLen.getSum();
            // std::make_heap has 3N time complexity.
            const long comps = 3 * botLen.getSum() +
                log2(botLen.getMean()) * botLen.getSum() / 3;
            os << "Events cancelled from top  : "    << ceTop
               << "\nEvents scanned in top      : " << ceScanTop
               << "\nEvents cancelled from ladder: " << ceLadder
               << "\nEvents scanned from ladder  : " << ceScanLadder
               << "\nEvents cancelled from bottom: " << ceBot
               << "\nEvents scanned from bottom  : " << ceScanBot
               << "\nNo cancel scans of bottom   : " << ceNoCanScanBot
               << "\nInserts into top          : " << insTop
               << "\nInserts into rungs        : " << insLadder
               << "\nInserts into bottom       : " << insBot
               << "\nMax rung count            : " << maxRungs
               << "\nAverage #buckets per rung   : " << avgBktCnt
               << "\nAverage bottom size       : " << botLen
               << "\nMax bottom size           : " << maxBotSize
               << "\nAverage bucket width      : " << avgBktWidth
               << "\nBottom to rung operations   : " << botToRung
```

```
               << "\nCompare estimate       : " << comps
               << std::endl;
        });
}

void
xxxx::TwoTierLadderQueue::enqueue(xxxx::Event* event) {
    if (top.getStartTime() < event−>getReceiveTime()) {
        DEBUG(std::cout << "Added to top: " << *event << std::endl);
        top.add(event);
        LQ2T_STATS(insTop++);
        return;
    }
    // Try to see if the event fits in the ladder. nRung is max rung index
    size_t rung = 0;
    while ((rung < nRung) && !ladder[rung].canContain(event)) {
        ASSERT((rung == 0) || ladder[rung].empty() ||
               (ladder[rung − 1].getCurrTime() >= ladder[rung].getCurrTime()));
        rung++;
    }
    if (rung < nRung) {
        DEBUG(ASSERT(bottom.empty() ||
                     (event−>getReceiveTime() > bottom.maxTime())));
        ladder[rung].enqueue(event);
        ladderEventCount++;  // Track events added to the ladder
        DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
                        << bottom.maxTime() << "): " << *event << "\n");
        LQ2T_STATS(insLadder++);
        return;
    }
    // Event does not fit in the ladder. It must go into bottom.
    // However, to ensure good performance we must keep bottom short.
    if ((bottom.size() > LQ2T_THRESH) && (bottom.getTimeRange() > 0)) {
        // Move events from bottom into ladder rung
        rung = createRungFromBottom();
        ASSERT(rung == nRung − 1);
        ASSERT(rung < ladder.size());
        // Due to rollback−reprocessing the event may be even
        // earlier than the last rung we just created!
        if (ladder[rung].canContain(event)) {
            ladder[rung].enqueue(event);
            ladderEventCount++;
            DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
                            << bottom.maxTime() << "): " << *event << "\n");
            LQ2T_STATS(insLadder++);
            return;
        }
    }
    // At this point, event must go into bottom, so enqueue it.
    bottom.enqueue(event);
    LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
    DEBUG(ASSERT(!haveBefore(bottom.first_event()−>getReceiveTime())));
    DEBUG(std::cout << "Added to bottom: " << *event << std::endl);
    LQ2T_STATS(insBot++);
}

// Implementation close to the version from the paper.
xxxx::TwoTierBucket&&
xxxx::TwoTierLadderQueue::recurseRung() {
    ASSERT(!empty());
    ASSERT(nRung > 0);
    ASSERT(!ladder.empty());
    // Now the last rung in ladder is the rung that has the next
    // bucket of events.
    xxxx::Time bktTime    = 0;  // set by removeNextBucket call below
    TwoTierRung& lastRung = ladder[nRung − 1];
    TwoTierBucket&& bkt   = lastRung.removeNextBucket(bktTime);
    ASSERT(!bkt.empty());
    ASSERT(!ladder.empty());
    // Check and create new rung in the ladder if the bucket is large.
    if ((bkt.size() > LQ2T_THRESH) && (nRung < MaxRungs)) {
        // Note: Here bucket width can dip a bit low. But that is
        // needed to ensure consistent ladder setup.
        const double bucketWidth = (lastRung.getBucketWidth() + bkt.size() −
                                    1.0) / bkt.size();
        // Create a new rung in the ladder
        nRung++;
        if (nRung > ladder.size()) {
            ladder.push_back(TwoTierRung(std::move(bkt), bktTime, bucketWidth));
            ASSERT(nRung == ladder.size());
        } else {
            ladder[nRung - 1].move(std::move(bkt), bktTime, bucketWidth);
        }
        DEBUG(std::cout << "2. Bucket width: " << bucketWidth << std::endl);
        LQ2T_STATS(avgBktWidth += bucketWidth);
        LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
        return recurseRung();  // Recurse now looking at newly added rung
    }
    // Track events being removed from the ladder
    ladderEventCount -= bkt.size();
    ASSERT(ladderEventCount >= 0);
    // Return bucket being removed.
    return std::move(bkt);
}

// Move events from ladder (or top) into bottom.
void
xxxx::TwoTierLadderQueue::populateBottom() {
    if (!bottom.empty()) {
        return;
    }
    if (ladderEventCount == 0) {   // nRung == -1
        if (top.empty()) {
            // There are no events in the ladder queue in this case
            ASSERT(empty());
            return;
        }
        // Move all events from top into buckets in first rung of the ladder!
        nRung++;
        ASSERT(nRung == 1);
        ladderEventCount += top.size();     // Track events in ladder
        // Move events to ladder
        if (nRung > ladder.size()) {
            ladder.push_back(TwoTierRung(std::move(top)));
            ASSERT(nRung == ladder.size());
        } else {
            ladder[nRung - 1].move(std::move(top), top.getMinTime(),
                                   top.getBucketWidth());
        }
        // Reset top counters and update the values of topStart for
        // next Epoch
        top.reset(top.getMaxTime());
        LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
        LQ2T_STATS(avgBktWidth += ladder.back().getBucketWidth());
        DEBUG(std::cout << "3. Bucket width: "
                        << ladder.back().getBucketWidth() << std::endl);
        DEBUG(prettyPrint(std::cout));
        ASSERT(top.empty());
    }
    // Bottom is empty. So we need to move events from the current
    // bucket in the ladder to bottom.
    ASSERT(!ladder.empty());
    ASSERT(bottom.empty());
    bottom.enqueue(recurseRung());  // Transfer bucket_k into bottom
    ASSERT(!bottom.empty());
    LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
    DEBUG(ASSERT(!haveBefore(bottom.first_event()->getReceiveTime())));
    LQ2T_STATS(botLen += bottom.size());
    // Clear out the rungs if we have used-up the last bucket in the ladder.
    while (nRung > 0 && ladder[nRung - 1].empty()) {
        LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
        nRung--;  // Logically remove rung from ladder
    }
}

int
xxxx::TwoTierLadderQueue::createRungFromBottom() {
    ASSERT(!bottom.empty());
    ASSERT(bottom.getTimeRange() > 0);
    DEBUG(std::cout << "Moving events from bottom to a new rung. Bottom has "
```

```
                    << bottom.size() << " events." << std::endl);
    // Compute the start time and bucket width for the rung.  Note
    // that with rollbacks, ladder can be empty and that situation
    // needs to be handled.
    const double bucketWidth = (ladder.empty() ? bottom.getBucketWidth() :
                                ladder[nRung - 1].getBucketWidth());
    // The paper computes rStart as RCur[NRung-1].  However, due to
    // rollback-reprocessing the bottom may have events that are below
    // RCur[NRung-1].  Consequently, we use the minimum of the two
    // values as as rstart
    const Time ladBkTime = ((nRung > 0) ? ladder[nRung - 1].getCurrTime() :
                            TIME_INFINITY);
    const Time rStart = std::min(ladBkTime,
                                 bottom.first_event()->getReceiveTime());
    ASSERT(rStart < ladBkTime);
    ASSERT(bottom.maxTime() < ladBkTime);
    ASSERT(bottom.maxTime() <= top.getStartTime());
    // Create a new rung and add it to the ladder.
    DEBUG(std::cout << "Moving bottom to rung. Events: " << bottom.size()
                    << ", rStart = " << rStart << ", bucketWidth = "
                    << (bucketWidth / bottom.size()) << std::endl);
    ladderEventCount += bottom.size();  // Update ladder event count
    LQ2T_STATS(botToRung += bottom.size());
    const double bktWidth = (bucketWidth + bottom.size() - 1.0) / bottom.size();
    DEBUG(std::cout << "bktWidth = " << bktWidth << std::endl);
    // Add rung and move move bottom into the last rung of the ladder.
    nRung++;
    if (nRung > ladder.size()) {
        ladder.push_back(TwoTierRung(std::move(bottom), rStart, bktWidth));
        ASSERT(nRung == ladder.size());
    } else {
        ladder[nRung - 1].move(std::move(bottom), rStart, bktWidth);
    }
    DEBUG(std::cout << "1. Bucket width: " << bktWidth << std::endl);
    LQ2T_STATS(avgBktWidth += bktWidth);
    LQ2T_STATS(maxRungs = std::max(maxRungs, nRung)); // Track max rungs
    ASSERT(bottom.empty());
    return nRung - 1;
}

int
xxxx::TwoTierLadderQueue::remove_after(xxxx::AgentID sender,
                                       const Time sendTime) {
    // Check and cancel entries in top rung.
    int numRemoved = top.remove_after(sender, sendTime
                                      LQ2T_STATS(COMMA ceScanTop));
    LQ2T_STATS(ceTop += numRemoved);
    // Cancel out events in each rung of the ladder.
    for (size_t rung = 0; (rung < nRung); rung++) {
        const int rungEvtRemoved =
            ladder[rung].remove_after(sender, sendTime
                                      LQ2T_STATS(COMMA ceScanLadder));
        ladderEventCount  -= rungEvtRemoved;
        numRemoved        += rungEvtRemoved;
        LQ2T_STATS(ceLadder += rungEvtRemoved);
    }
    // Clear out the rungs in ladder that are now empty after event
    // cancellations.
    while (nRung > 0 && ladder[nRung - 1].empty()) {
        LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
        nRung--;  // Logically remove rung from ladder
    }
    // Save original size of bottom to track stats.
    LQ2T_STATS(const size_t botSize  = bottom.size());
    // Cancel events from bottom.
    const int botRemoved  = bottom.remove_after(sender, sendTime);
    if (botRemoved > -1) {
        numRemoved += botRemoved;
        // Update statistics counters
        LQ2T_STATS(ceBot += botRemoved);
        LQ2T_STATS(ceScanBot += botSize);
        LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
    }
    return numRemoved;
}
```

```cpp

// This method is purely for debugging. So performance is not
// important
bool
xxxx::TwoTierLadderQueue::haveBefore(const Time recvTime,
                                     const bool checkBottom) const {
    // Check top
    if (top.haveBefore(recvTime)) {
        std::cout << "Top has event that is <= " << recvTime << std::endl;
        prettyPrint(std::cout);
        return true;
    }
    // Check each rung of the ladder
    for (size_t rung = 0; (rung < nRung); rung++) {
        if (ladder[rung].haveBefore(recvTime)) {
            std::cout << "Rung #" << rung << " has event that is <= "
                      << recvTime << std::endl;
            prettyPrint(std::cout);
            return true;
        }
    }
    // Check bottom rung.
    if (checkBottom && bottom.haveBefore(recvTime)) {
        std::cout << "Bottom has event that is <= " << recvTime << std::endl;
        prettyPrint(std::cout);
        return true;
    }
    // When control drops here it mean the whole 2-tier ladder queue
    // does not have an event with timestamp lower than recvTime.
    return false;
}

// ------------------------[  EventQueue implementation ]----------------------

void*
xxxx::TwoTierLadderQueue::addAgent(xxxx::Agent* agent) {
    UNUSED_PARAM(agent);
    return NULL;  // 2-tier queue has no cross-references to store in agent
}

void
xxxx::TwoTierLadderQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    const AgentID receiver = agent->getAgentID();
    // Remove events for agent from top
    LQ2T_STATS(ceScanTop += top.size());
    int numRemoved    = top.remove(receiver);
    LQ2T_STATS(ceTop += numRemoved);

    // Next remove events for agent from all the rungs in the ladder
    for (TwoTierRung& rung : ladder) {
        int rungEvtRemoved = rung.remove(agent->getAgentID()
                                         LQ2T_STATS(COMMA ceBot));
        ladderEventCount  -= rungEvtRemoved;
        numRemoved        += rungEvtRemoved;
        LQ2T_STATS(ceLadder += rungEvtRemoved);
    }
    // Finally remove events from bottom for the agent.
    LQ2T_STATS(const size_t botSize = bottom.size());
    const int botRemoved  = bottom.remove(receiver);
    LQ2T_STATS(ceScanBot += botSize);
    LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
    numRemoved           += botRemoved;
    LQ2T_STATS(ceBot     += botRemoved);
}


xxxx::Event*
xxxx::TwoTierLadderQueue::front() {
    if (empty()) {
        // Nothing to return.
        return NULL;
    }
    if (bottom.empty()) {
        populateBottom();
```

```cpp
        DEBUG(prettyPrint(std::cout));
    }
    ASSERT(!bottom.empty());
    return bottom.first_event();
}

void
xxxx::TwoTierLadderQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (empty()) {
        // No events to dequeue.
        return;
    }
    // We only dequeue from bottom. So ensure it has events in it.
    if (bottom.empty()) {
        // Move events from top or a ladder rung into bottom.
        populateBottom();
    }
    ASSERT(!bottom.empty());
    bottom.dequeueNextAgentEvents(events);
    ASSERT(!events.empty());
    DEBUG(ASSERT(!haveBefore(events.front()->getReceiveTime())));
}

// The main interface method used by XXXX to schedule event.
void
xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    UNUSED_PARAM(agent);
    event->increaseReference();
    enqueue(event);
}

// Method for block addition (typically used during rollback recovery)
void
xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent,
                                  xxxx::EventContainer& events) {
    UNUSED_PARAM(agent);
    for (auto& curr : events) {
        enqueue(curr);
    }
    events.clear();
}

// Method to cancel all events in the 2-tier heap.
int
xxxx::TwoTierLadderQueue::eraseAfter(xxxx::Agent* dest,
                                     const xxxx::AgentID sender,
                                     const xxxx::Time sentTime) {
    UNUSED_PARAM(dest);
    return remove_after(sender, sentTime);
}

void
xxxx::TwoTierLadderQueue::prettyPrint(std::ostream& os) const {
    // Print information on top.
    os << "Top: Events=" << top.size()
       << ", startTime=" << top.getStartTime()
       << ", minTime="   << top.getMinTime()
       << ", maxTime="   << top.getMaxTime()  << std::endl;
    // Print info on each rung of the ladder
    std::cout << "Ladder (rungs=" << nRung << ", size="
              << ladder.size() << "):\n";
    for (size_t i = 0; (i < nRung); i++) {
        os << "[" << i << "]: ";
        ladder[i].prettyPrint(os);
    }
    // Print info on bottom
    os << "Bottom: Events=" << bottom.size()
       << ", min="        << (!bottom.empty() ? bottom.findMinTime() : -1.0)
       << ", max="        << (!bottom.empty() ? bottom.maxTime()     : -1.0)
       << std::endl;
}

#endif
```