```
151     updateHeap(agent);
152 }
153
154 void
155 ThreeTierHeapEventQueue::enqueueEvent(xxxx::Agent* agent, xxxx::Event* event) {
156     ASSERT(agent != NULL);
157     ASSERT(event != NULL);
158     ASSERT( agent->tier2 != NULL );
159     ASSERT(getIndex(agent) < agentList.size());
160     // A convenience reference to tier2 list of buckets
161     Tier2List& tier2 = *agent->tier2;
162     // Use binary search O(log n) to find match or insert position
163     agentBktCount += tier2.size();
164     Tier2List::iterator iter =
165         std::lower_bound(tier2.begin(), tier2.end(), event, lessThanPtr);
166     // There are 3 cases: 1. we found matching bucket, 2: iterator
167     // to bucket with higher recvTime, or 3: tier2.end().
168     if (iter == tier2.end()) {
169         tier2.emplace_back(makeTier2Entry(event));  // add new entry to end.
170     } else if ((*iter)->getReceiveTime() == event->getReceiveTime()) {
171         // We found an existing bucket. Append this event to this
172         // existing bucket.
173         (*iter)->updateContainer(event);
174     } else {
175         // If there is no bucket with a matching receive time in Tier2
176         // vector, then insert an instance of HOETier2Entry (aka
177         // bucket) into the vector at the appropriate position.
178         ASSERT((*iter)->getReceiveTime() > event->getReceiveTime());
179         tier2.emplace(iter, makeTier2Entry(event));
180     }
181     // ASSERT(std::is_sorted(tier2.begin(), tier2.end()));
182 }
183
184 void
185 ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent,
186                                  xxxx::EventContainer& events) {
187     ASSERT(agent != NULL);
188     // Note: events container may be empty!
189     ASSERT(getIndex(agent) < agentList.size());
190     // Add all events to tier2 entries appropriately.
191     for (xxxx::Event* event : events) {
192         // Enqueue event but don't waste time fixing-up heap yet for
193         // this agent.  We will do it at the end after all events are
194         // added.  However, we don't increase reference counts in this
195         // API.
196         enqueueEvent(agent, event);
197     }
198     // Clear out all the events in the incoming container
199     events.clear();
200     // Update the location of this agent on the heap as needed.
201     updateHeap(agent);
202 }
203
204 int
205 ThreeTierHeapEventQueue::eraseAfter(xxxx::Agent* dest,
206                                     const xxxx::AgentID sender,
207                                     const xxxx::Time sentTime) {
208     int  numRemoved = 0;
209     ASSERT( dest->tier2 != NULL );
210     Tier2List& tier2eventPQ = *dest->tier2;
211     long currIdx = tier2eventPQ.size() - 1;
212     while (!tier2eventPQ.empty() && (currIdx >= 0)) {
213         if (tier2eventPQ[currIdx]->getReceiveTime() > sentTime) {
214             std::vector<xxxx::Event*>& eventList =
215                 tier2eventPQ[currIdx]->getEventList();
216             size_t index = 0;
217             while (!eventList.empty() && (index < eventList.size())) {
218                 Event* const evt = eventList[index];
219                 ASSERT(evt != NULL);
220                 if (isFutureEvent(sender, sentTime, evt)) {
221                     evt->decreaseReference();
222                     numRemoved++;
223                     eventList[index] = eventList.back();
224                     eventList.pop_back();
225                 } else {
```



```
226                     index++;  // onto next event in this bucket
227                 }
228             }
229             // If all events are canceled then this bucket needs to be
230             // removed from the tier2 entry.
231             if (eventList.empty()) {
232                 tier2Recycler.emplace_back(tier2eventPQ[currIdx]);
233                 tier2eventPQ.erase(tier2eventPQ.begin() + currIdx);
234             }
235         }
236         currIdx--;
237     }
238     // Update the 1st tier heap for scheduling.
239     updateHeap(dest);
240     // Return number of events canceled to track statistics.
241     return numRemoved;
242 }
243
244 void
245 ThreeTierHeapEventQueue::reportStats(std::ostream& os) {
246     UNUSED_PARAM(os);
247     const long comps = std::log2(agentList.size()) *
248         avgSchedBktSize.getCount() + fixHeapSwapCount.getSum();
249     os << "Average #buckets per agent  : " << agentBktCount     << std::endl;
250     os << "Average scheduled bucket size: " << avgSchedBktSize  << std::endl;
251     os << "Average fixHeap compares    : " << fixHeapSwapCount << std::endl;
252     os << "Compare estimate            : " << comps            << std::endl;
253 }
254
255 void
256 ThreeTierHeapEventQueue::prettyPrint(std::ostream& os) const {
257     os << "HeapOfVectorsEventQueue::prettyPrint() : not implemented.\n";
258 }
259
260 size_t
261 ThreeTierHeapEventQueue::getIndex(xxxx::Agent *agent) const {
262     ASSERT(agent != NULL);
263     size_t index = reinterpret_cast<size_t>(agent->fibHeapPtr);
264     ASSERT(index < agentList.size());
265     ASSERT(agentList[index] == agent);
266     return index;
267 }
268
269 size_t
270 ThreeTierHeapEventQueue::updateHeap(xxxx::Agent* agent) {
271     ASSERT(agent != NULL);
272     size_t index = getIndex(agent);
273     if (agent->oldTopTime != getTopTime(agent)) {
274         index = fixHeap(index);
275         // Update the position of the agent in the scheduler's heap
276         // Validate
277         ASSERT(agentList[index] == agent);
278         ASSERT(getIndex(agent) == index);
279         // Update time value as well for future access
280         agent->oldTopTime = getTopTime(agent);
281         // Validation check.
282         ASSERT(getTopTime(agentList[0]) <= getTopTime(agentList[1]));
283     }
284     // Return the new index position of the agent
285     return index;
286 }
287
288 size_t
289 ThreeTierHeapEventQueue::fixHeap(size_t currPos) {
290     ASSERT(currPos < agentList.size());
291     xxxx::Agent* value    = agentList[currPos];
292     const size_t len      = (agentList.size() - 1) / 2;
293     size_t secondChild    = currPos;
294     int           opCount  = 0;
295     // This code was borrowed from libstdc++ implementation to ensure
296     // that the fix-ups are consistent with std::make_heap API.
297     while (secondChild < len)  {
298         secondChild = 2 * (secondChild + 1);
299         if (compare(agentList[secondChild], agentList[secondChild - 1])) {
300             secondChild--;
```