```cpp
#include <algorithm>    // find(), move(), move_backward(), equal(), swap(), lexicographical_compare()
#include <cstddef>      // size_t
#include <initializer_list>
#include <iomanip>      // setw()
#include <iterator>     // distance(), next()
#include <stdexcept>    // logic_error
#include <string>

#include "Book.hpp"
#include "BookList.hpp"



// As a rule, I strongly recommend avoiding macros, unless there is a compelling reason - this is such a case. This really does need
// to be a macro and not a function due to the way the preprocessor expands the source code location information.  It's important to
// have these expanded where they are used, and not here. But I just can't bring myself to writing this, and getting it correct,
// everywhere it is used.  Note:  C++20 will change this technique with the introduction of the source_location class. Also note the
// usage of having the preprocessor concatenate two string literals separated only by whitespace.
#define exception_location "\n detected in function \"" + std::string(__func__) +  "\"     \
                           "\n at line " + std::to_string( __LINE__ ) +                    \
                           "\n in file \"" __FILE__ "\""




/*********************************************************************
**  Private implementations, types, and objects
*********************************************************************/
bool BookList::containersAreConsistant() const
{
  // Sizes of all containers must be equal to each other
  if(    _books_array_size != _books_vector.size()
      || _books_array_size != _books_dl_list.size()
      || _books_array_size !=  books_sl_list_size() ) return false;

  // Element content and order must be equal to each other
  auto current_array_position   = _books_array  .cbegin();
  auto current_vector_position  = _books_vector .cbegin();
  auto current_dl_list_position = _books_dl_list.cbegin();
  auto current_sl_list_position = _books_sl_list.cbegin();

  while( current_vector_position != _books_vector.cend() )
  {
    if(    *current_array_position != *current_vector_position
        || *current_array_position != *current_dl_list_position
        || *current_array_position != *current_sl_list_position ) return false;

    // Advance the iterators to the next element in unison
    ++current_array_position;
    ++current_vector_position;
    ++current_dl_list_position;
    ++current_sl_list_position;
  }

  return true;
}
```

```
63 // Calculate the size of the singly linked list on demand
64 std::size_t BookList::books_sl_list_size() const
65 {
66   #ifndef STUDENT_TO_DO_REGION
67     /// Some implementations of a singly linked list maintain the size (number of elements in the list).  std::forward_list does
68     /// not. The size of singly linked list must be calculated on demand by walking the list from beginning to end counting the
69     /// number of elements visited.  The STL's std::distance() function does that, or you can write your own loop.
70     if constexpr( (true) )
71       return std::distance( _books_sl_list.cbegin(), _books_sl_list.cend() );                          // distance() walks the list
72     else
73     {
74       // alternative implementation
75       std::size_t size = 0;
76       for( auto current = _books_sl_list.cbegin(); current != _books_sl_list.cend(); ++size, ++current ); // body intentionally empty
77       return size;
78     }
79   #endif
80 }
81
82
83
84
85
86
87
88
89
90
91
92 /*************************************************************************
93 **  Constructors, destructor, and assignment operators
94 *************************************************************************/
95 // Rule of 6 - I wanted a tailored assignment operator, so I should (best practice) write the other too
96 BookList::BookList()                                 = default;
97
98 BookList::BookList( const BookList  & other )        = default;
99 BookList::BookList(       BookList && other )        = default;
100
101 BookList & BookList::operator=( BookList    rhs )    { swap( rhs ); return *this; }
102 BookList & BookList::operator=( BookList && rhs )    = default;
103
104 BookList::~BookList()                                = default;
105
106
107
108 BookList::BookList( const std::initializer_list<Book> & initList )
109   : _books_vector ( initList.begin(), initList.end() ),
110     _books_dl_list( initList.begin(), initList.end() ),
111     _books_sl_list( initList.begin(), initList.end() )
112 {
113   // Unlike the other containers that are expandable, the array has a fixed capacity N.  Copy only the first N elements of the
114   // initialization list into the array.
115   for( auto p = initList.begin();  _books_array_size < _books_array.size()  &&  p != initList.end();   ++_books_array_size, ++p)
116   {
117     _books_array[_books_array_size] = *p;
118   }
119 }
120
121
122
123 BookList & BookList::operator+=( const std::initializer_list<Book> & rhs )
124 {
```

```
125   #ifndef STUDENT_TO_DO_REGION
126     /// Concatenate the right hand side book list of books to this list by repeatedly inserting at the bottom of this book list.
127     /// The input type is a container of books accessible with iterators like all the other containers.  The constructor above gives
128     /// an example.  Use BookList::insert() to insert at the bottom.
129     for( const auto & book : rhs ) insert( book, Position::BOTTOM );
130   #endif
131
132   // Verify the internal book list state is still consistent amongst the four containers
133   if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
134   return *this;
135 }
136
137
138
139 BookList & BookList::operator+=( const BookList & rhs )
140 {
141   #ifndef STUDENT_TO_DO_REGION
142     /// Concatenate the right hand side book list of books to this list by repeatedly inserting at the bottom of this book list.
143     /// All the rhs containers (array, vector, list, and forward_list) contain the same information, so pick just one to traverse.
144     /// Walk the container you picked inserting its books to the bottom of this book list. Use BookList::insert() to insert at the
145     /// bottom.
146     for( const auto & book : rhs._books_sl_list ) insert( book, Position::BOTTOM );
147   #endif
148
149   // Verify the internal book list state is still consistent amongst the four containers
150   if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
151   return *this;
152 }
153
154
155
156
157
158
159
160 /*************************************************************************
161 **  Queries
162 *************************************************************************/
163 std::size_t BookList::size() const
164 {
165   // Verify the internal book list state is still consistent amongst the four containers
166   if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
167
168   #ifndef STUDENT_TO_DO_REGION
169     /// All the containers are the same size, so pick one and return the size of that.  Since the forward_list has to calculate the
170     /// size on demand, stay away from using that one.
171     return _books_vector.size();
172   #endif
173 }
174
175
176
177 std::size_t BookList::find( const Book & book ) const
178 {
179   // Verify the internal book list state is still consistent amongst the four containers
180   if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
181
182   #ifndef STUDENT_TO_DO_REGION
183     /// Locate the book in this book list and return the zero-based position of that book.  If the book does not exist, return the
184     /// size of this book list as an indicator the book does not exist.  The book will be in the same position in all the containers
185     /// (array, vector, list, and forward_list) so pick just one of those to search.  The STL provides the find() function that is a
186     /// perfect fit here, but you may also write your own loop.
```

```
187
188     // C++17 added std::optional which provides an alternative way to indicate the book was not found.  I may adopt that in a future release
189     //
190     // I prefer the "if constexpr" to the "#if 0" as a means to stub out large sections of code because the "if constexpr" will still
191     // compile all branches (and hence check of errors), where the "#if 0" does not.
192     if constexpr( (true) )
193     {
194       // get the iterator of the book
195       // The content of all the containers is the same, so the offset will be the same in each - so pick one
196       return std::find( _books_vector.cbegin(), _books_vector.cend(), book ) - _books_vector.cbegin();  // subtract to convert iterator to offset
197     }
198
199     else
200     {
201       // alternative implementation
202       // The content of all the containers is the same, so the offset will be the same in each - so pick one
203       for( std::size_t offset = 0; offset < _books_vector.size(); ++offset )   if( _books_vector[offset] == book ) return offset;
204
205       return _books_vector.size();
206     }
207   #endif
208 }
209
210
211
212
213
214
215
216
217
218
219
220
221 /*************************************************************************
222 **  Mutators
223 *************************************************************************/
224 void BookList::insert( const Book & book, Position position )
225 {
226   // Convert the TOP and BOTTOM enumerations to an offset and delegate the work
227   if     ( position == Position::TOP    )  insert( book, 0      );
228   else if( position == Position::BOTTOM )  insert( book, size() );
229   else throw std::logic_error( "Unexpected insertion position" exception_location );  // Programmer error.  Should never hit this!
230 }
231
232
233
234 void BookList::insert( const Book & book, std::size_t offsetFromTop )       // insert new book at offsetFromTop, which places it before the current book at offsetFromTop
235 {
236   // Validate offset parameter before attempting the insertion.  std::size_t is an unsigned type, so no need to check for negative
237   // offsets, and an offset equal to the size of the list says to insert at the end (bottom) of the list.  Anything greater than the
238   // current size is an error.
239   if( offsetFromTop > size() ) throw InvalidOffset_Ex( "Insertion position beyond end of current list size" exception_location );
240
241
242   /**********  Prevent duplicate entries  ********************/
243   #ifndef STUDENT_TO_DO_REGION
244     /// Silently discard duplicate items from getting added to the book list.  If the to-be-inserted book is already in the list,
245     /// simply return.
246     if( find( book ) != size() ) return;
247   #endif
248
```

```
249
250
251
252     // Inserting into the book list means you insert the book into each of the containers (array, vector, list, and forward_list).
253     // Because the data structure concept is different for each container, the way a book gets inserted is a little different for
254     // each.  You are to insert the book into each container such that the ordering of all the containers is the same.  A check is
255     // made at the end of this function to verify the contents of all four containers are indeed the same.
256
257
258     /*********  Insert into array  ********************/
259     {
260       #ifndef STUDENT_TO_DO_REGION
261         /// Unlike the other containers, std::array has no insert() function, so you have to write it yourself. Insert into the array
262         /// by shifting all the items at and after the insertion point (offsetFromTop) to the right opening a gap in the array that
263         /// can be populated with the given book.  Remember that arrays have fixed capacity and cannot grow, so make sure there is
264         /// room in the array for another book before you start by verifying _books_array_size is less than _books_array.size().  If
265         /// not, throw CapacityExceeded_ex.  Also remember that you must keep track of the number of valid books in your array, so
266         /// don't forget to adjust _books_array_size.
267
268         // Array has a fixed size and cannot be increased.  Make sure another book will fit.
269         if( _books_array_size >= _books_array.size() ) throw CapacityExceeded_Ex( "Cannot fit another book into fixed size array already at capacity" exception_location );
270
271         ///
272         /// open a hole to insert new book by shifting to the right everything at and after the insertion point.
273         /// For example:  a[8] = a[7];  a[7] = a[6];  a[6] = a[5];  and so on.
274         /// std::move_backward will be helpful, or write your own loop.
275         ///
276         /// See function FixedVector::insert() in FixedVector.hpp in our Sequence Container Implementation Examples, and
277         /// RationalArray::insert() in RationalArray.cpp in our Rational Number Case Study examples.
278         std::move_backward( _books_array.begin() + offsetFromTop,              // start of a range to move elements from
279                             _books_array.begin() + _books_array_size,          // end of a range to move elements from. This element is not moved
280                             _books_array.begin() + _books_array_size + 1 );    // position of one past the final element in the destination range
281
282
283         // insert the book and increment size
284         _books_array[offsetFromTop] = book;
285         ++_books_array_size;
286       #endif
287     }  // Insert into array
288
289
290
291     /*********  Insert into vector  ********************/
292     {
293       #ifndef STUDENT_TO_DO_REGION
294         /// The vector STL container std::vector has an insert function, which can be directly used here.  But that function takes a
295         /// pointer (or more accurately, an iterator) that points to the book to insert before.  You need to convert the zero-based
296         /// offset from the top to an iterator by advancing _books_vector.begin() offsetFromTop times.  The STL has a function called
297         /// std::next() that does that, or you can use simple pointer arithmetic to calculate it.
298         ///
299         /// Behind the scenes, std::vector::insert() shifts to the right everything at and after the insertion point, just like you
300         /// did for the array above.
301
302         // Sense you were given an offset and not a direct iterator to the insertion point, you need to advance _books_vector.begin()
303         // offsetFromTop times.
304         _books_vector.insert( _books_vector.begin() + offsetFromTop, book );
305       #endif
306     } // Insert into vector
307
308
309
310     /*********  Insert into doubly linked list  *********/
```

```
311   {
312     #ifndef STUDENT_TO_DO_REGION
313       /// The doubly linked list STL container std::list has an insert function, which can be directly used here.  But that function
314       /// takes a pointer (or more accurately, an iterator) that points to the book to insert before.  You need to convert the
315       /// zero-based offset from the top to an iterator by advancing _books_dl_list.begin() offsetFromTop times.  The STL has a
316       /// function called std::next() that does that, or you can write your own loop.
317
318       // Sense you were given an offset and not a direct iterator to the insertion point, you need to walk the list to locate the
319       // insertion point
320       _books_dl_list.insert( std::next( _books_dl_list.begin(), offsetFromTop ), book );
321     #endif
322   } // Insert into doubly linked list
323
324
325
326   /*********  Insert into singly linked list  *********/
327   {
328     #ifndef STUDENT_TO_DO_REGION
329       /// The singly linked list STL container std::forward_list has an insert function, which can be directly used here.  But that
330       /// function inserts AFTER the book pointed to, not before like the other containers.  A singly linked list cannot look
331       /// backwards, only forward.  You need to convert the zero-based offset from the top to an iterator by advancing
332       /// _books_sl_list.before_begin() offsetFromTop times.  The STL has a function called std::next() that does that, or you can
333       /// write your own loop.
334
335       // Can only insert after a node, and we need to insert before.  begin() returns an iterator to the first node, but
336       // before_begin() returns an iterator to something before the first node
337       _books_sl_list.insert_after( std::next( _books_sl_list.before_begin(), offsetFromTop ), book );
338     #endif
339   } // Insert into singly linked list
340
341   // Verify the internal book list state is still consistent amongst the four containers
342     if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
343   } // insert( const Book & book, std::size_t offsetFromTop )
344
345
346
347   void BookList::remove( const Book & book )
348   {
349     remove( find( book ) );
350   }
351
352
353
354   void BookList::remove( std::size_t offsetFromTop )
355   {
356     // Removing from the book list means you remove the book from each of the containers (array, vector, list, and forward_list).
357     // Because the data structure concept is different for each container, the way an book gets removed is a little different for
358     // each.  You are to remove the book from each container such that the ordering of all the containers is the same.  A check is
359     // made at the end of this function to verify the contents of all four containers are indeed the same.
360
361     if( offsetFromTop >= size() ) return;                                    // no change occurs if (zero-based) offsetFromTop >= size()
362
363     /*********  Remove from array  **********/
364     {
365       #ifndef STUDENT_TO_DO_REGION
366         /// Close the hole created by shifting to the left everything at and after the remove point.
367         /// For example:  a[5] = a[6];  a[6] = a[7];  a[7] = a[8];  and so on
368         ///
369         /// std::move() will be helpful, or write your own loop.  Also remember that you must keep track of the number of valid books
370         /// in your array, so don't forget to adjust _books_array_size.
371         ///
372         /// See function FixedVector<T>::erase() in FixedVector.hpp in our Sequence Container Implementation Examples, and
```

```
373       /// RationalArray::remove() in RationalArray.cpp in our Rational Number Case Study examples.
374       std::move( _books_array.begin() + offsetFromTop + 1,                        // start of a range to move elements from
375                  _books_array.begin() + _books_array_size,                        // end of a range to move elements from. This element is not moved
376                  _books_array.begin() + offsetFromTop );                          // the beginning of the destination range
377
378
379
380       --_books_array_size;
381       _books_array[_books_array_size] = {};                                       // overwrite the new "hole" with a default book
382     #endif
383   } // Remove from array
384
385
386
387   /*********  Remove from vector  ********************/
388   {
389     #ifndef STUDENT_TO_DO_REGION
390       /// The vector STL container std::vector has an erase function, which can be directly used here.  But that function takes a
391       /// pointer (or more accurately, an iterator) that points to the book to be removed.  You need to convert the zero-based
392       /// offset from the top to an iterator by advancing _books_vector.begin() offsetFromTop times.  The STL has a function called
393       /// std::next() that does that, or you can use simple pointer arithmetic to calculate it.
394       ///
395       /// Behind the scenes, std::vector::erase() shifts to the left everything after the insertion point, just like you did for the
396       /// array above.
397
398       // Sense you were given an offset and not a direct iterator to the insertion point, you need to advance
399       // _books_vector.begin() offsetFromTop times.
400       _books_vector.erase( _books_vector.begin() + offsetFromTop );
401     #endif
402   } // Remove from vector
403
404
405
406   /*********  Remove from doubly linked list  *********/
407   {
408     #ifndef STUDENT_TO_DO_REGION
409       /// The doubly linked list STL container std::list has an erase function, which can be directly used here.  But that function
410       /// takes a pointer (or more accurately, an iterator) that points to the book to remove.  You need to convert the zero-based
411       /// offset from the top to an iterator by advancing _books_dl_list.begin() offsetFromTop times.  The STL has a function called
412       /// std::next() that does that, or you can write your own loop.
413
414       // Sense we were given an offset and not a direct iterator to the insertion point, need to walk the list to locate the insertion
415       // point
416       _books_dl_list.erase( std::next( _books_dl_list.begin(), offsetFromTop ) );
417     #endif
418   } // Remove from doubly linked list
419
420
421
422   /*********  Remove from singly linked list  *********/
423   {
424     #ifndef STUDENT_TO_DO_REGION
425       /// The singly linked list STL container std::forward_list has an erase function, which can be directly used here.  But that
426       /// function erases AFTER the book pointed to, not the one pointed to like the other containers.  A singly linked list cannot
427       /// look backwards, only forward.  You need to convert the zero-based offset from the top to an iterator by advancing
428       /// _books_sl_list.before_begin() offsetFromTop times.  The STL has a function called std::next() that does that, or you can
429       /// write your own loop.
430
431       // Can only erase after a node, and we need to erase before.  begin() returns an iterator to the first node, but
432       // before_begin() returns an iterator to something before the first node
433       _books_sl_list.erase_after ( std::next( _books_sl_list.before_begin(), offsetFromTop ) );
434     #endif
```

```
435   } // Remove from singly linked list
436
437   // Verify the internal book list state is still consistent amongst the four containers
438   if( !containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
439 } // remove( std::size_t offsetFromTop )
440
441
442
443 void BookList::moveToTop( const Book & book )
444 {
445   #ifndef STUDENT_TO_DO_REGION
446     /// If the book exists, then remove and reinsert it.  Else do nothing.  Use BookList::find() to determine if the book exists in
447     /// this book list.
448     if( auto offset = find( book ); offset != size() )
449     {
450       remove( offset );
451       insert( book, Position::TOP );
452     }
453   #endif
454 }
455
456
457
458 void BookList::swap( BookList & rhs ) noexcept
459 {
460   if( this == &rhs ) return;
461
462   _books_array  .swap( rhs._books_array   );
463   _books_vector .swap( rhs._books_vector  );
464   _books_dl_list.swap( rhs._books_dl_list );
465   _books_sl_list.swap( rhs._books_sl_list );
466
467   std::swap( _books_array_size, rhs._books_array_size );
468 }
469
470
471
472
473
474
475
476
477
478
479
480
481 /*************************************************************************
482 **  Insertion and Extraction Operators
483 *************************************************************************/
484 std::ostream & operator<<( std::ostream & stream, const BookList & bookList )
485 {
486   if( !bookList.containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
487
488   unsigned count = 0;
489   for( const auto & book : bookList._books_sl_list )   stream << '\n' << std::setw(5) << count++ << ":  " << book;
490
491   return stream;
492 }
493
494
495
496 std::istream & operator>>( std::istream & stream, BookList & bookList )
```

```
497 {
498   if( !bookList.containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
499
500   for( Book book; stream >> book; )   bookList.insert( book, BookList::Position::BOTTOM );
501
502   return stream;
503 }
504
505
506
507
508
509
510
511
512
513
514
515
516 /*************************************************************************
517 **  Relational Operators
518 *************************************************************************/
519 bool operator==( const BookList & lhs, const BookList & rhs )
520 {
521   return !( lhs < rhs )  &&  !( rhs < lhs );
522 }
523
524
525
526 bool operator<( const BookList & lhs, const BookList & rhs )
527 {
528   if( !lhs.containersAreConsistant() || !rhs.containersAreConsistant() ) throw BookList::InvalidInternalState_Ex( "Container consistency error" exception_location );
529
530   // comparing arrays using std::array::operator==() will compare every element in the array (which is usually what you want).  But
531   // in this case, only the first N elements are valid, so compare only those elements.  This is implicitly handled in the other
532   // containers because size() and end() are already adjusted for only the valid elements.
533   auto begin_lhs = lhs._books_array.cbegin();
534   auto begin_rhs = rhs._books_array.cbegin();
535
536   auto end_lhs   = begin_lhs + lhs._books_array_size;
537   auto end_rhs   = begin_rhs + rhs._books_array_size;
538
539   // C++20's spaceship operator should optimize this algorithm, but for now ...
540   if( !std::equal( begin_lhs, end_lhs, begin_rhs, end_rhs ) )  return std::lexicographical_compare ( begin_lhs, end_lhs, begin_rhs, end_rhs );
541   if( lhs._books_vector  != rhs._books_vector              )  return lhs._books_vector  < rhs._books_vector;
542   if( lhs._books_dl_list != rhs._books_dl_list            )  return lhs._books_dl_list < rhs._books_dl_list;
543   if( lhs._books_sl_list != rhs._books_sl_list            )  return lhs._books_sl_list < rhs._books_sl_list;
544
545   // At this point, all attributes are equal to each other, so the lhs cannot be less than the rhs
546   return false;
547 }
548
549 bool operator!=( const BookList & lhs, const BookList & rhs ) { return !( lhs == rhs ); }
550 bool operator<=( const BookList & lhs, const BookList & rhs ) { return !( rhs  < lhs ); }
551 bool operator> ( const BookList & lhs, const BookList & rhs ) { return  ( rhs  < lhs ); }
552 bool operator>=( const BookList & lhs, const BookList & rhs ) { return !( lhs  < rhs ); }
553
```