20 Lists Module (lists.hhf)

The list.bodyhhf library module provides a class data type and a set of functions to manipulate linked lists within a program.

20.1 The Lists Module

To use the list functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "lists.hhf" )
or
#include( "stdlib.hhf" )
```

20.1.0.1 List Data Types

The HLA Standard Library provides a generic list abstract data type via the lists module. The lists module provides two classes: a generic list class and a generic, abstract, *node*_t class. These classes have (approximately) the following definitions:

```
    nodePtr_t :pointer to node_t;
node_t:
    class

        var
            Prev: pointer to node_t;
            Next: pointer to node_t;

        procedure create; @returns( "esi" ); @external;
        method destroy;                      @abstract;
        method cmpNodes( n:nodePtr_t );        @abstract;

    endclass;

list_t:
    class

        var
            Head:       pointer to node_t;
            Tail:       pointer to node_t;
            Cnt:        uns32;
            align(4);

      procedure create; @returns( "esi" );
      method destroy;
      method numNodes; @returns( "eax" );
      method append_index( var n:node_t; posn: dword );
             @returns( "esi" );

      method append_node( var n:node_t; var after: node_t );
         @returns( "esi" );

      method append_last( var n:node_t ); @returns( "esi" );
      method insert_index( var n:node_t; posn:dword ); @returns( "esi" );
      method insert_node( var n:node_t; var before:node_t );
         @returns( "esi" );

      method insert_first( var n:node_t ); @returns( "esi" );
```

```
        method delete_index( posn:dword );
        method delete_node( var n:node_t );
        method delete_first;
        method delete_last;
        method index( posn:dword );
        method xchgNodes( n1:nodePtr_t; n2:nodePtr_t );
        method sort;
        method reverse;
        method search( cmpThunk:thunk );
        iterator nodeInList;
        iterator nodeInListReversed;
        iterator filteredNodeInList( t:thunk );
        iterator filteredNodeInListReversed( t:thunk );

      endclass;
```

The *node_t* class is an abstract base class from which you must derive a node type for the nodes in your list. You would normally override the *node_t.create* procedure and write a procedure that specifically allocates storage for an object of type *node_t* and initializes any important data fields. If you like, your overloaded *create* procedure can call *node_t.create* to initialize the link fields of the node you create, although this is not strictly necessary.

The *node_t.destroy* method is an abstract method that you must override. The *list_t.destory* method calls *node_t.destroy* (or, at least, your overloaded version of it) in order to free the storage associated with a given node. A typical concrete implementation of this function looks like the following:

```
method MyNode.destroy; @nodisplay; @noframe;
begin destroy;

  // On entry, ESI points at the current node object.
  // Free the storage associated with this node.

  if( isInHeap( esi )) then

     free( esi );

  endif;

end destroy;
```

The *node_t.cmpNodes* method is another abstract method you may need to override. This method compares the current node (referenced by *this*) against the node whose address the caller passes as the single argument. This method compares the two nodes and sets the carry and zero flags in a manner consistent with an unsigned integer comparison (that is, it sets the carry flag if the *this* node is less than the parameter node; it sets the zero flag if the two nodes are equal; it clears these two flags if the opposite conditions hold). The *list_t.sort* and *list_t.search* functions use *node_t.cmpNodes*; if you use either of these functions in the list_t objects you create, you will need to provide a concrete implementation of the *node_t.cmpNodes* method. Note that because *node_t.cmpNodes* is an abstract method, there is no default implementation for this function – you must provide a concrete implementation if you call it or you call some other function that calls it. Here is a sample implementation that demonstrates this:

```
method MyNode.cmpNodes;
var
  thisSave
begin cmpNodes;

  // Assume there is a "keyID" signed integer field in MyNode and
  // when we compare the two nodes we simply compare the int32 values
  // and set the flags for an unsigned comparison.

  push( eax );
```

```
    mov( n, eax );
    mov( (type MyNode [eax]).keyID, eax );
    cmp( eax, this.keyID );
    if( @l ) then

        stc();// Make @b. Note that Z is clear

    else

        clc();// Make @nb.

        // Note that Z is set appropriately at this point.

    endif;
    pop( eax );

end cmpNodes;
```

For a typical example of an overloaded node_t class, see the listDemo.hla example in the HLA examples subdirectory.

The *list_t* class is an abstract data type used to maintain lists of nodes. Internally, the *list_t* class represents lists of nodes using a doubly-linked list, although your applications should not be aware of the internal implementation. Likewise, for efficiency reasons the *list_t* class maintains a pointer to the head of the list, a pointer to the tail of the list, and a count of the number of nodes currently in the list. Your applications should ignore these fields (note that you can obtain the number of nodes in the list by calling the *numNodes* method) and treat the fields as private to the class.

20.2 List_t Class Function Types

In most HLA classes, there are three types of functions: (static) procedures, (dynamic), and (dynamic) iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). Static procedure calls are very efficient, but you lose the benefits of inheritence and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second effiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though exteremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The HLA Standard Library predefines two classes: *list_t* and *virtualList_t*. They differ in how they define the functions appearing in the class types. The *list_t* type uses static procedures for all functions, the *virtualList_t* type uses methods for all class functions. Therefore, *list_t* objects will make direct calls to all the functions (and only link in the procedures you actually call); however, *list_t* objects do not support function polymorphism in derived classes. The *virtualList_t* type does support polymorphism for all the class methods, but whenever you use this data type you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *list_t* and *virtualList_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program (better yet, don't use both types in the same program – use *virtualList_t* if you need polymorphism).

20.3 Creating New List Class Types

As it turns out, the only difference between a method and a procedure (in HLA) is how that method/procedure is called. The actual function code is identical regardless of the declaration (the reason HLA supports method and procedure declarations is so that it can determine how to populate the VMT and to determine how to call the function). By pulling some tricks, it's quite possible to call a procedure using the method invocation scheme or call a method using a direct call (like a static procedure). The Standard Library list class module takes advantage of this trick to make it possible to create new list classes with a user-selectable set of procedures and methods. This allows you to create a custom list type that uses methods for those functions you want to override (as methods) and use procedures for those functions you don't call or will never override (as virtual methods). Indeed, the *list_t* and *virtualList_t* data types were created using this technique. The *list_t* data type was created specifying all functions as procedures, the *virtualList_t* data type was created specifying all functions as methods. By using the *_hla.make_listClass* macro, you can create new date data types that have any combination of procedures and methods.

`_hla.make_listClass( className, "<list of methods>" )`

_hla.make_listClass is a macro that generates a new data type. As such, you should only invoke this macro in an HLA type declaration section. This macro requires two arguments: a class name and a string containing the list of methods to use in the new data type. The method list string must contain a sequence of method names (typically separated by spaces, though this isn't strictly necessary) from the following list:

```
destroy
numNodes
appendIndex
appendNode
appendLast
insertIndex
insertNode
insertFirst
deleteIndex
deleteNode
deleteFirst
deleteFast
index
xchgNodes
sort
reverse
search
```

Here is *_hla.make_listClass* macro invocation that creates the *virtualList* type:

```
type
  _hla.make_listClass
  (
      virtualList_t,
      "destroy"
      "numNodes"
      "appendIndex"
      "appendNode"
      "appendLast"
      "insertIndex"
      "insertNode"
      "insertFirst"
      "deleteIndex"
      "deleteNode"
      "deleteFirst"
      "deleteFast"
      "index"
      "xchgNodes"
      "sort"
      "reverse"
      "search"
  );
```

(For those unfamiliar with the syntax, HLA automatically concatenates string literals that are separated by nothing but whitespace; therefore, this macro contains exactly two arguments, the *virtualList_t* name and a single string containing the concatenation of all the strings above.)

From this macro invocation, HLA creates a new data type using methods for each of the names appearing in the string argument. If a particular date function's name is not present in the *_hla.make_listClass* macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the *list_t* data type (which uses static procedures for all the list functions):

```
type
                _hla.make_listClass( list_t, " " );
```

Because the function string does not contain any of the list function names, the *_hla.make_listClass* macro generates static procedures for all the list functions.

The *list_t* type is great if you don't need to create a derived list class that allows you to polymorphically override any of the list functions. If you do need to create methods for certain functions and you don't mind linking in all the list class functions (and you don't mind the extra overhead of a method call, even for those functions you're not overloading), the *virtualList_t* data type is convenient to use because it makes all the functions virtual (that is, methods). Probably 99% of the time you won't be calling the list functions very often, so the overhead of using method invocations for all list functions is irrelevant. In those rare cases where you do need to support polymorphism for a few list functions but don't want to link in the entire set of list functions, or you don't want to pay the overhead for indirect calls to functions that are never polymorphic, you can create a new list class type that specifies exactly which functions require polymorphism.

For example, if you want to create a date class that overrides the definition of the sort and search functions, you could declare that new type thusly:

```
type
   _hla.make_listClass
    (
        MyListClass,
        "sort"
        "search"
    );
```

This new class type (*MyListClass*) has two methods, *sort* and *search*, and all the other list functions are static procedures. This allows you to create a derived class that overloads the *sort* and *search* methods and access those methods when using a generic *MyListClass* pointer, e.g.,

```
type
   derivedMyListClass :
     class inherits( MyListClass );

         override method sort;
         override method search;

     endclass;
```

Again, it is important for you to understand that types created by *_hla.make_listClass* are base types. They are not derived from any other class (e.g., *virtualList* is not derived from *list* or vice-versa). The types created by the *_hla.make_listClass* macro are independent and incompatible types. For this reason, you should avoid using different base list class types in your program. Pick (or create) a base list class and use that one exclusively in an application. You'll avoid confusion by following this rule.

20.4 List Procedures, Methods, and Iterators

Because you can create your own list data types, describing list functions as procedures or methods is somewhat inaccurate. In the sections that follow, a function is described as a "procedure" if it is always a static procedure and you cannot override that (this only applies to the constructor); a function is described as a "method" if you can create a new data type and define that function to be a static procedure or a dynamic method via the *hla.make_listClass* macro. Note that the four iterators defined in the list class (*list_t.nodeInList,*

list_t.nodeInlistReversed, list_t.filteredNodeInList, and *list_t.filteredNodeInListReversed*) are always dynamic iterators, you cannot change their definition.

As is typical for the Standard Library documentation when describing classes and objects, this chapter does not provide any examples of low-level assembly language calls to the various methods in the *list_t* class. The assumption here is that someone who is doing object-oriented programming in assembly language is perfectly happy using the high-level method calls (particularly as the low-level method invocations are rather messy). If you're an exception to this rule, please consult the HLA documentation for details on making direct (low-level) calls to class methods and iterators.

The calling sequence examples appearing throughout this chapter use the following object declarations:

```
static
  sList:virtualList_t;
  pList:pointer to virtualList_t;
```

Note that the calling sequences are exactly the same for static and virtual objects. That is, you could replace the two *virtualList_t* data types above with *list_t* and the examples would all still be syntactically correct.

When discussing methods, the following sections claim that any call to a method will wipe out the value in the EDI register. This is true if the class data type actually uses methods. If you've created a new list data type using *_hla.make_listClass* and you've defined a function to be a procedure rather than a method, then the call is direct and it does not necessarily disturb the value of the EDI register. However, you should not make this assumption. Some methods might actually assume that it's okay to disturb the value in EDI as it was used to hold the VMT address for the call. Better safe than sorry – assume that if it's a method, EDI's value gets modifed.

20.5 List Constructor and Destructor

`procedure list_t.create; @returns( "esi" );`

This is the standard constructor for the list class. If you call this class procedure via *list_t.create()* it will allocate storage for a new *list_t* object, initialize the fields of that object (to the empty list), and return a pointer to that *list_t* object in ESI. If you call this class procedure via *someListVarName.create()* then this procedure will initialize the (presumably) allocated *list_t* object (again, to the empty list).

```
HLA high-level calling sequence examples:


// Constructor call that allocates storage for a list object:

virtualList_t.create();
mov( esi, pList );

// Constructor call that initializes an already-allocated object:

sList.create();
```

`method list_t.destroy;`

This method frees the storage associated with each node in the list (if the individual nodes were allocated on the heap), it then frees the storage associated with the *list_t* object itself, assuming the list was allocated on the heap. Note that successful execution of this method requires that you create a derived class from the abstract base class *node_t* and that you've overridden the *node_t.destroy* method. The *list_t.destroy* method deallocates the nodes in the list by calling the *node_t.destroy* method for each node in the list.

```
HLA high-level calling sequence examples:


sList.destroy();
pList.destroy();
```

20.6 Accessor Functions

method list_t.numNodes; @returns("eax");

This function returns the number of nodes currently in the list in the EAX register. You should always call this routine rather than access the *list_t.Cnt* field directly.

```
HLA high-level calling sequence examples:

  sList.numNodes();
  mov( eax, sNumNodes );

  pList.numNodes();
  mov( eax, pNumNodes );
```

20.6.0.1 Adding Nodes to a List

#macro list_t.append(node, posn);
#macro list_t.append(node, node);
#macro list_t.append(node);

The *list_t.append* macro provides function overloading on the *list_t.append_index, list_t.append_node,* and *list_t.appendLast* functions. The *list_t.append* macro checks the number and type of the parameters and calls the appropriate *list_t.append_** function whose signature matches the argument list. See the discussion of the following three methods for details on the specific calls.

method list_t.append_index(var n:node_t; posn: dword); @returns("esi");

This method appends node *n* to the list after node *posn* in the list. If *posn* is greater than or equal to the number of nodes in the list, then this method appends node *n* to the end of the list. Normally, you would not call this method directly. Instead, you would use the

```
sList.append(n,posn);
```

macro to call this method. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

```
HLA high-level calling sequence examples:

  sList.append_index( MyNodePtr, 4 );// Append after fifth node
  pList.append( MyNodePtr, 5 );// Append after sixth node
```

method list_t.append_node(var n:node_t; var after: node_t);
@returns("esi");

This method inserts node *n* in the object list immediately after node *after* in that list. This method assumes that *after* is a node in the object's list; it does not validate this fact. Therefore, you must ensure that *after* is a member of the object's list. Normally, you would not call this function directly; instead, you would invoke the

```
listVar.append( n, after);
```

macro to do the work. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

```
HLA high-level calling sequence examples:

  // Append NewNode after the NodeInList node:

  sList.append_node( NewNode, NodeInList );

  // Append anotherNewNode after someNodeInpList:

  pList.append( AnotherNewNode, someNodeInpList );
```

method list_t.append_last(var n:node_t); @returns("esi");

This method appends node *n* to the end of the object list. Normally you would not call this method directly, instead you would just invoke the macro:

```
listVar.append(n);
```

This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

```
HLA high-level calling sequence examples:

  // Append NewNode at the end of the list:

  sList.append_last( NewNode );

  // Append anotherNewNode at the end of the pList:

  pList.append( AnotherNewNode );
```

#macro list_t.insert(node, posn);
#macro list_t.insert(node, node);
#macro list_t.insert(node);

The *list_t.insert* macro provides function overloading on the *list_t.insert_index, list_t.insert_node,* and *list_t.insertFirst* functions. The *list_t.insert* macro checks the number and type of the parameters and calls the appropriate *list_t.insert_** function whose signature matches the argument list. See the discussion of the following three methods for details on the specific calls.

method list_t.insert_index(var n:node_t; posn:dword); @returns("esi");

This method inserts node *n* before the $posn^{th}$ node in the list. If *posn* is greater than or equal to the number of nodes in the list, this method simply appends the node to the end of the list (remember, nodes are numbered from 0..Cnt-1; so if *posn=Cnt* then that would imply inserting the node at the end of the list). Normally you would not call this method directly; instead, you'll invoke the

```
listVar.insert( n, posn);
```

macro to do the job. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

```
HLA high-level calling sequence examples:

  sList.insert_index( MyNodePtr, 4 );// Insert before fifth node
  pList.insert( MyNodePtr, 5 );// Insert before sixth node
```

```
method list_t.insert_node( var n:node_t; var before:node_t );
    @returns( "esi" );
```

This method inserts node n before node before in the object's list. This method assumes that before is an actual member of the list, it does not verify this prior to insertion. You would not normally call this routine directly. Instead, invoke the *listVar.insert(n, before)* macro to do th actual work. This function returns a pointer to node n in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Insert NewNode before the NodeInList node:

sList.insert_node( NewNode, NodeInList );

// Insert anotherNewNode before someNodeInpList:

pList.insert( AnotherNewNode, someNodeInpList );
```

```
method list_t.insert_first( var n:node_t ); @returns( "esi" );
```

This function inserts node n at the beginning of the object's list. You would not normally call this method directly; you should normally invoke the *listVar.insert(n)* macro and let it do all the work. This function returns a pointer to node n in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Insert NewNode at the beginning of the list:

sList.insert_last( NewNode );

// Insert anotherNewNode at the start of the pList:

pList.insert( AnotherNewNode );
```

20.7 Removing Nodes From a List

```
#macro list_t.delete( posn );
#macro list_t.delete( node );
#macro list_t.delete();
```

These macros overload the *list_t.delete_index, list_t.delete_node,* and *list_t.delete_first* methods in the list class. The macro determines which of these methods to call by testing the number and types of the macro's arguments. Note that this macro does not overload the *list_t.delete_last* method as it does not have a unique signature (i.e., *list_t.delete_last's* signature would be identical to *list_t.delete_first's*).

```
method list_t.delete_index( posn:dword ); @returns( "esi" );
```

This method removes the $posn^{th}$ node from the list and returns a pointer to this node in ESI. Normally you would invoke the

```
list_t.delete( posn );
```

macro rather than calling this method directly.

HLA high-level calling sequence examples:

```
sList.delete_index( 4 );// Delete the fifth node
pList.insert( ecx );// Delete the node whose index is in ECX
```

method list_t.delete_node(var n:node_t); @returns("esi");

This method removes node *n* from the list and returns a pointer to this node in ESI. This method assumes that node *n* actually is in the list; it does not verify this. Normally, you would invoke the

```
list_t.delete(n);
```

macro rather than call this method directly.

HLA high-level calling sequence examples:

```
// Delete the deleteMe node:

sList.delete_node( deleteMe );
mov( esi, deleted_node );

// Delete anotherUselessNode:

pList.delete( anotherUselessNode );
mov( esi, deleted_node_too );
```

method list_t.delete_first; @returns("esi");

This method removes the first node from the list and returns a pointer to this node in ESI. Normally you would not call this method directly but you would invoke the

```
list_t.delete();
```

macro instead.

HLA high-level calling sequence examples:

```
// Delete the node at the beginning of the list:

sList.delete_first();
mov( esi, deleted_node );

pList.delete();
mov( esi, deleted_node_too );
```

method list_t.delete_last; @returns("esi");

This method removes the last node from the list and returns a pointer to this node in ESI.

HLA high-level calling sequence examples:

```
// Delete the node at the end of the list:

sList.delete_last();
mov( esi, deleted_node );
```

20.8 Accessing Nodes in a List

method list_t.index(posn:dword); @returns("esi");

This method returns a pointer to the *posn*th node in the list in the ESI register. It returns NULL if the list is empty. It returns the address of the last node in the list if *posn* >= *Cnt*.

HLA high-level calling sequence examples:

```
// Access the 5th node in the list:

sList.index( 4 );
mov( esi, fifth_node );
```

iterator list_t.nodeInList;

This iterator returns a pointer to each node in a list in the ESI register. This iterator traverses the list forward – from the beginning of the list to the end of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

HLA high-level calling sequence examples:

```
// Traverse the entire list:

foreach sList.nodeInList() do

   << Do something with the node pointer in ESI… >>

endfor;

foreach pList.nodeInList() do

   << Do something with the node pointer in ESI… >>

endfor;
```

iterator list_t.nodeInListReversed;

This iterator returns a pointer to each node in a list in the ESI register. This iterator traverses the list backward, from the end of the list to the beginning of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

HLA high-level calling sequence examples:

```
// Traverse the entire list backwards:

foreach sList.nodeInListReversed() do

   << Do something with the node pointer in ESI… >>

endfor;

foreach pList.nodeInListReversed() do

   << Do something with the node pointer in ESI… >>
```

```
    endfor;
```

iterator list_t.filteredNodeInList(t:thunk);

This iterator traverses the list and returns a pointer to each node that is "approved" by the thunk *t*. This iterator traverses the list forward – from the beginning of the list to the end of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

On each FOREACH loop iteration, the *list_t.filteredNodeInList* iterator will call the t thunk and pass it a pointer to the current node in ESI. The (caller-defined) thunk will test that node (application-specific) and return true in AL if the FOREACH loop should iterate on that particular node; the thunk should return false in AL if the FOREACH loop should skip that particular node in the interation sequence.

```
HLA high-level calling sequence examples:

    // Traverse the list and operate on all nodes whose
    // "j" field is greater than or equal to 10:

    foreach
        sList.filteredNodeInList
        (
            thunk
            #{
                xor( eax, eax );
                cmp( (type MyNode [esi]).j, 10 );
                setae( al );
            }#
        )
    do

        << Do something with the node pointer in ESI… >>

    endfor;

    // Traverse the list and operate on all nodes whose
    // "j" field is equal to the "k" field:

    foreach
        sList.filteredNodeInList
        (
            thunk
            #{
                mov( (type MyNode [esi]).j, eax );
                cmp( eax, (type MyNode [esi]).k );
                mov( 0, eax );
                sete( al );
            }#
        )
    do

        << Do something with the node pointer in ESI… >>

    endfor;
```

iterator list_t.filteredNodeInListReversed(t:thunk);

This iterator behaves just like *list_t.filteredNodeInList* except that it traverses the list backwards. As for *list_t.filteredNodeInList*, you must provide a thunk that approves or rejects each node in the list. Only approved

nodes are passed along to the body of the FOREACH loop. Like most iterators, you normally use this iterator within a FOREACH loop.

```
HLA high-level calling sequence examples:

  // Traverse the list and operate on all nodes whose
  // "j" field is greater than or equal to 10:

  foreach
     sList.filteredNodeInListReversed
     (
        thunk
        #{
           xor( eax, eax );
           cmp( (type MyNode [esi]).j, 10 );
           setae( al );
        }#
     )
  do

     << Do something with the node pointer in ESI… >>

  endfor;

  // Traverse the list and operate on all nodes whose
  // "j" field is equal to the "k" field:

  foreach
     sList.filteredNodeInListReversed
     (
        thunk
        #{
           mov( (type MyNode [esi]).j, eax );
           cmp( eax, (type MyNode [esi]).k );
           mov( 0, eax );
           sete( al );
        }#
     )
  do

     << Do something with the node pointer in ESI… >>

  endfor;
```

20.9 Miscellaneous List Functions

`method list_t.reverse;`

This method reverses the nodes in the list. That is, the first node becomes the last node, the second node becomes the second to the last node, …, and the last node becomes the first node. This function only changes the (private) Next and Prev pointers in each node, it does not physically move the nodes around in memory (so pointers to the nodes remain valid) nor does it change any other data in the nodes in the list.

```
HLA high-level calling sequence examples:

  // Reverse the lists:
```

```
sList.reverse();
pList.reverse();
```

method list_t.xchgNodes(n1:nodePtr_t; n2:nodePtr_t);

This method exchanges two nodes in the list. Specifically, this function swaps the nodes' (private) *Next* and *Prev* fields and, if necessary, updates the beginning and ending node pointers in the list object. The *n1* and *n2* parameters must contain pointers to nodes within the list or this function will produce undefined results. This function does not check *n1* or *n2* to verify that they are within the list, it is the caller's responsibility to ensure this.

HLA high-level calling sequence examples:

```
// Exchange the 4th and 7th nodes in the list:

sList.index( 3 );
mov( eax, ebx ); // Get address of the 4th node.
sList.index( 6 );// Get the address of the 7th node.
sList.xchgNodes( eax, ebx );// Exchange the two nodes.
```

method list_t.sort;

This method sorts the nodes in the list in ascending order. This function invokes the *node_t* class *cmpNodes* method in order to sort the list, so if you use this function you must provide a concrete implementation of the *cmpNodes* method or the HLA run-time system will raise an "Abstract Method Executed" exception.

Note that if you want to sort the list in descending order, and you don't otherwise need to sort it in ascending order, you can define the *node_t.cmpNodes* method to reverse the state of the carry flag (that is, return carry set on greater than and carry clear on less than or equal). However, be careful if you do this as those semantics will exist for all lists that try to sort the particular *node_t* class you've defined this way. Perhaps a better solution would be to overload the list class you've defined and create a new *sort* procedure that sorts the data in descending order. Or simply modify the list class source code and add a *list_t.sortReverse* function.

HLA high-level calling sequence examples:

```
// Sort the lists:

sList.sort();
pList.sort();
```

method list_t.search(cmpThunk:thunk); @returns("eax");

This method searches for a specific node in the list (starting at the front of the list and sequentially searching toward the end of the list). It executes the *cmpThunk* thunk on each node until either the thunk returns true in AL (that is, it does not return false in AL) or it reaches the end of the list. If the thunk ever returns true, then this function returns a pointer to the associated node in the EAX register. If the search function scans the entire list and *cmpThunk* always returns false, then this function will return NULL (zero) in EAX upon reaching the end of the list.

This function is very similar to the *list_t.filteredNodeInList* function except that it does not iterate on every matching node in the list. Instead, this function returns only the first matching occurrence found in the list.

HLA high-level calling sequence examples:

```
// Search for the first node whose
// "j" field is equal to 10:
```

```
sList.search
(
    thunk
    #{
        xor( eax, eax );
        cmp( (type MyNode [esi]).j, 10 );
        sete( al );
    }#
);
if( eax <> NULL ) then

    << do something with the node pointed at by EAX… >

endif;
```