Paper 011-2009

Creating Common Information Structures Using List's Stored in Data Step Hash Objects

Shawn Edney, ThotWave Technologies, Chapel Hill, NC

ABSTRACT

The ability of the SAS® 9 DATA step HASH object to resize the amount of information that can be stored during DATA step execution opens up a number of opportunities to exploit. Significantly, it provides a dynamic framework on which you can mimic the behavior of other common information structures, such as array lists, stacks, queues, deques, and linked lists. This paper shows how this is accomplished by using the HASH object to store list data and to implement custom code to provide list functionality. This paper provides a brief introduction to lists, followed by a series of discussions and example code for implementing several common information structures.

KEYWORDS: DEQUE, HASH, LINKED, LIST, QUEUE, STACK

INTRODUCTION

This paper strives to illustrate, through brief discussion and examples, how to fabricate common information structures such as array lists, stacks, queues, deques and linked lists, all of which will be built upon the version 9.1.3 hash object by implementing straightforward list structures. The purpose of implementing such data structures is to assist in abstracting the details of computational algorithms allowing the SAS programmer to implement the algorithm with code that more closely matches the standard data structure's model logic.

The paper has two major sections. The first gives a very brief introduction to lists, list properties and list methods with which we will be concerned. The second section presents data structure examples and discusses how to implement the logic for each structure using the hash object.

While this paper will present example code "recipes" that can be added to the SAS programmer's toolbox, its main goal is to illustrate how to leverage the list structure through the data step hash object to implement information structures that SAS does not natively support in the data step.

As this paper is focused on fairly advanced concepts implemented using advanced SAS coding techniques it is assumed that the reader has a good understanding of

- Hash object code syntax
- The behavior of the hash object and its related functions
- "Link" and "Return" statements
- Various forms of "Do While" loops

The details of these concepts will only be explained when it helps to illustrate a particular point related to implementing a particular data structure or leveraging the hash object.

There are many potential openings to discuss and illustrate ways that macro code could be used to simplify or modularize the presented code but this paper will not venture into the world of macro.

LISTS

The "list" is the major structural idea that underpins all of the examples shown in this paper. A list is basically what comes to mind, a sequence of data items. The lists covered in this paper though will have carefully crafted properties that define, for each item in the list, the item's relative location within the particular kind of list we are using. These location properties change slightly depending on the type of list we are using. Strictly speaking the lists we will be concerned with fall within the definition of "Linked List" but we will refer to them simply as "lists". The types of lists we will consider are linear lists and circular lists.

A linear list's most notable property is that it has "ends". The ends are often labeled with names such as: top, bottom, left, right, start or finish. The English alphabet is an example of a linear list in that the "ends" are the letters A and Z. These "ends" are the major distinguishing characteristic between linear and circular lists.

Circular list's, as you might have guessed have no "ends" within the list. The circular list has the property that if you start at one item in the list and move through the list in a consistent direction you will eventually arrive back at the point where you started. An example of a circular list might be the numbers on a standard twelve hour clock face. If we observe the hour hand, we can watch as it moves continuously in a clockwise direction, eventually coming back to the hour where we started observing its movement.

Now we need to define what is contained in our list. For our discussion we will focus on individual groups of data, each of which we will call a "node". Think of a node as the data you have access to via a single hash key. With this in mind a node is further defined as a particular group of information that is made up of an index "key" and the associated data that is accessed via the key. We will only consider lists that use a single key variable in our hash table structure. The data associated with a key will be simply called "node data".

Since we are going to be storing our data in hash tables we have to make our keys, in an individual list, either numeric or character type. All the examples we will cover in this paper will have numeric typed keys. This is primarily because it makes sorting keys straightforward and simplifies the logic needed to implement each information structure's associated functions. You might be thinking that this still sounds similar to an array which uses integer keys but as you will see in some of the examples, we will not limit ourselves solely to integer keys. For some information structures we will purposely create real numbers to use as keys.

Now to the item data that is indexed by each key. Because we are using the hash object to hold our list we can conveniently store multiple node data fields of type character or numeric. That is we can have character and numeric data indexed by the same key in the same list at the same time. This is of course a normal and convenient property of the data step hash object. One other notable point to make about node data is that while it will contain the process data it will also include additional data fields that are there solely to support the properties or methods of the particular type of information structure we are implementing. Of particular note will be link fields that provide information on the relative location of the node.

Now that we have covered what is in the list lets briefly touch on the abstract methods that we will create in order to actually implement a list. In order to make full use of the data structures presented we will need to be able to perform each of the functions listed in Table 1. Note that some of these methods are not strictly necessary from a computer science point of view but rather necessary from a SAS coding perspective or at least the particular coding created in the examples.

Table 1: List methods

Need to have a method to*:	Description
Construct an empty list (CONSTRUCTOR)	This is the action of creating the list framework to actually hold data. This action will be accomplished by initializing the appropriate hash table for the particular information structure we are using.
Test if the list is empty (IS_EMPTY)	This is simply a method to test if a data structure is empty (the hash table has no data).
Return a reference to the "head" node (HEAD)	This is a method to return the key for a "special" node. Often this is set to the first, or last, list node in a linear list.
Traverse the list (TRAVERSE)	This method will provide the logic to move from one node to the next.
Retrieve data (RETRIEVE)	Methods to return the node data associated with a particular node key.
Update data (UPDATE)	Method for updating node data of an existing node.
Insert a new node of data into the list (INSERT)	This method adds a node anywhere in the list, relative to an existing node unless the list is empty, in which case we just add the value to the list.
Remove a node from the list (DELETE_NODE)	This method removes a specified node.
Delete a list (DELETE_LIST)	Method of deleting (deconstructing) the underlying data structure (ultimately a hash table). This will often occur by completing (ending) data step execution which will cause the associated hash object to be deleted but it is possible and sometimes desirable to programmatically delete the hash during data step execution so that it can be re-instantiated (constructed).

* The keyword in parenthesis that follows each method will be used to help point the reader to the list method when presenting details associated with each information structure.

The last list property we need to discuss is the type of "link" that can exist between data items. For our discussion, a link is a variable in each item data that contains the key value to the next item in the list. We will use special logic to deal with the items that may not have a "next" data item. This situation arises when we reach the end of a linear list or a circular list has only one item. Regardless of whether we are talking about a linear list or a circular list, if we only have one item data field that contains a link to the next item, in a particular direction, we call this type of list a "single linked" list. If we add a second data item that allows us to traverse the list in the opposite direction then we have a "double linked" list. We will consider one variation on this. It is possible to constrain item keys such that the key to each item provides enough information to tell us what the next key must be. We will refer to this type of link as an "implicit" link while the first type we covered will be referred to as "explicit" link. The only implicitly linked list we will consider will be a double linked linear list. The implicit/explicit terms will only be used when discussing implicit linked lists. If the link type isn't specified you should assume we are talking about explicit linked lists.

As much of the this paper will deal with discussing how links between nodes are being used we are going to define the upper case terms NEXT and PREV to represent node data representing the index keys to neighboring nodes with NEXT always being a key in the opposite directions from PREV and the directions being consistent for each keyword. The included examples will usually use hash object data field names "next" and "prev" to represent this information. Table 2 lists the example illustrations for each list type we will consider.

Table 2: Index of illustrated list examples by type

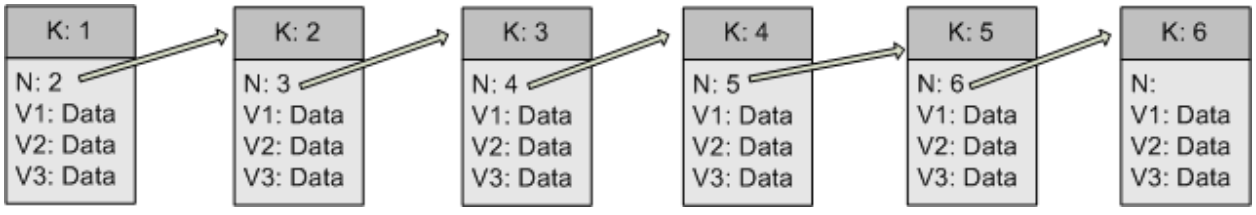
List Type	Linear List			Circular Lists	
	Implicitly Linked	Explicitly linked		Explicitly Linked	
	Double	Single	double	Single	double
Illustration #	1	2	3	4	5

Within the list illustrations the "K" values are for node keys, the "N" and "P" values are the next and previous links and the "V" items are the specific data item fields we want to store in our information structure. If a structure has N or P values then it is a single linked list. If it has N and P values then it is a double linked list. If no N or P value is presented then it is an implicitly linked list.

Illustration 1: Implicit Double Linked Linear list

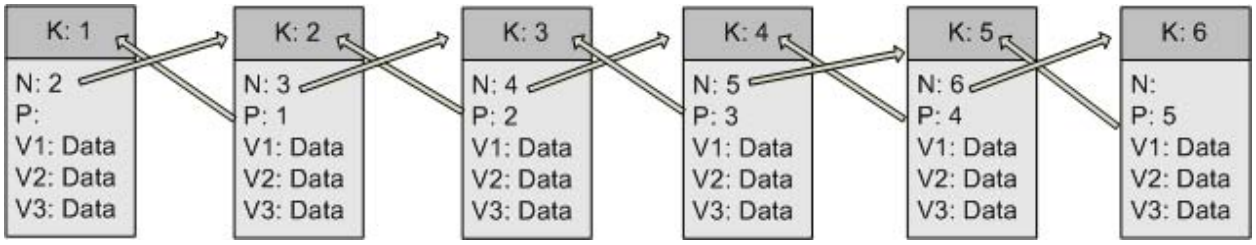
The implicitly linked list does not have data specifying the key value to the NEXT or PREV value. Instead it relies on logic that says the node keys will be integers such that there is never a missing key value between two keys.

Illustration 2: Single Linked Linear List



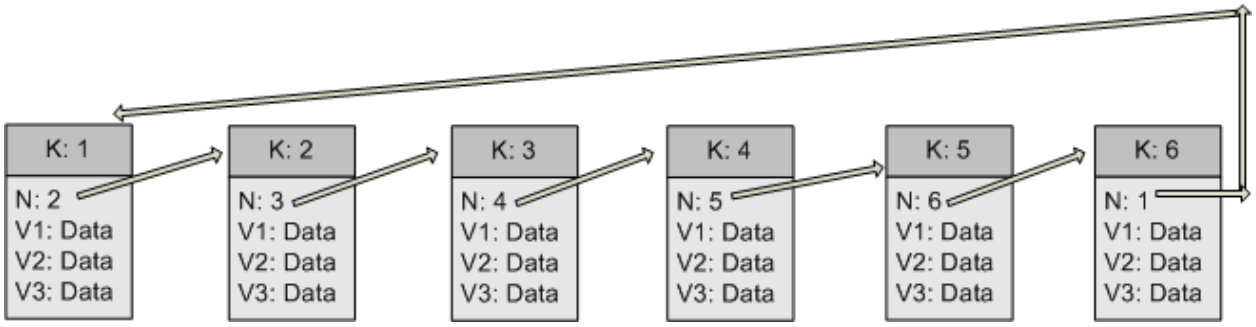
The single linked linear list nodes contain links to the following node, except for the last node, which contains a missing value for NEXT.

Illustration 3: Double Linked Linear List



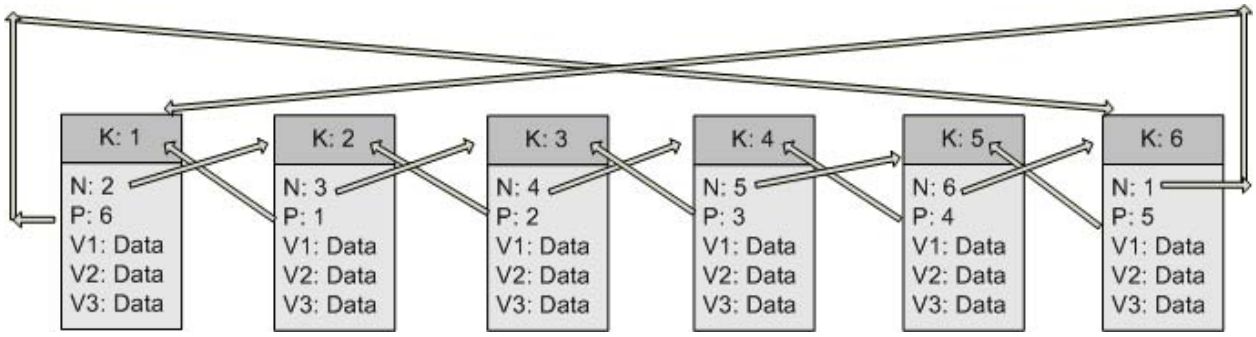
The linear double linked list contains explicit node data containing the link to neighboring nodes except for the end nodes which have a missing value for either NEXT or PREV.

Ilustration 4: Single Linked Circular List



In a single linked circular list every node contains a NEXT value.

Illustration 5: Double Linked Circular list



The double linked circular list is similar to the linear double linked list with the exception that all node data NEXT and PREV values contain the key values of neighboring nodes.

LIST METHOD DETAILS

CONSTRUCTOR METHOD

The constructor method is used to instantiate the underlying hash object upon which we will fashion our list based data structure. Careful planning is needed to correctly code the constructor code. While we will be using basic patterns similar to those found in SAS help documentation for instantiating hash objects you will need to make sure you take care of the following hash properties.

- Define the node key with the proper type (numeric in our examples) and in most cases make sure to define the node key within the data as well (or provide a second variable that duplicates the key)
- Define any node data fields that are needed by the list methods that are to be implemented. In most cases

this will included data fields specifying the node keys for one or both of the neighboring nodes. All of our examples will use field names NEXT and/or PREV to specify the neighboring nodes. Also be sure to set the data type to match the node key type.

- Define any process node data fields appropriately. These are the primary data fields we will want to retrieve when implementing an algorithm depending upon our data structure.

An example constructor for a double linked list would be:

```
/* CONSTRUCTOR */
if _n_=1 then do;
  length location prev ikey next 8 char_data $ 1 num_data 8;
  declare hash harray(ordered:"a");
  declare hiter iarray("harray");
  harray.definekey("location");
  harray.definedata("prev", "ikey", "next", "char_data", "num_data");
  harray.definedone();
  call missing(location, prev, ikey, next, char_data, num_data);
end;
```

Note that this constructor looks like a normal hash instantiation pattern. The hash properties that have been defined include

- The hash object has been instantiate as an ascending, ordered hash object. This is necessary for the hash iterator mentioned below but also provides us with a quick method for retrieving “first” and “last” nodes in the hash table.
- A hash iterator “iarray” has been defined providing us with a native method for iterating through the hash object, starting at either the first or last node.
- A numeric, node key field “location”.
- A numeric node data item field “ikey” that will always match the value in “location”. We could have just defined location as both a key and data but this example uses a second field.
- Two numeric node data fields next and prev (generally referred to as NEXT and PREV) to define the two neighboring node keys.
- A data field “char_data” to hold process character data of length 1.
- A data field “num_data” to hold process numeric data.

Note that we don’t have any indication of whether this constructor is for a linear or circular list. This is because it could function for either depending on how the list is populated and maintained.

HEAD METHOD

The HEAD method is used to store and update one or more non-hash variables that contain the node key value for one or more specific nodes within the list structure. Usually the HEAD method is used to track the first or last node in a list. There is also a case for storing a random node within say a circular list to provide a starting node for other methods, most notably TRAVERSE (TRAVERSE will be covered in a following section).

All of the examples in this paper that implement the HEAD method will use retained variables to hold HEAD node keys. For our examples, when an explicit HEAD method value is implemented, it will always be populated unless the hash object contains no data. A snippet of code implementing an explicit variable HEAD method might look like:

```
/* Variable to hold the key to an item that can serve as "head" */
retain head_key .;
…
<code that inserts, updates, or deletes a node causing the head key to be updated>
/* variable “location” is the key to the new head node */
head_key = location;
```

Because we are using an underlying hash object as our data structure for lists and forcing the use of appropriately (numerically) ordered keys, it is possible to create an ordered hash object and use the hash object’s “first” and “last” functions to return the nodes that will usually be of importance when considering a HEAD method. In this case we will not create explicitly retained variables but rather depend on the ability of the hash object’s “first” and “last” functions to retrieve the node specified.

IS_EMPTY METHOD

The IS_EMPTY method provides a mechanism for testing if our underlying hash object contains any data. This will be most important when identifying that there is list data available for processing or when identifying that we are about to

add the first node to our structure. In the case of adding our first node we might need to provide special logic to properly deal with the lack of neighboring nodes when populating the values in the NEXT and/ or PREV fields, if they exist.

Two possible methods for implementing IS_EMPTY would include testing for the number of items contained in the hash object, using the “NUM_ITEMS” attribute or testing if the “head” reference, associated with the “HEAD” method is populated with a non-missing value.

Implementing the IS_EMPTY method using the hash object “NUM_ITEMS” attribute is basically a test that says if there is one or more keys in the hash object then it is not empty. An example would look like:

```
/* IS_EMPTY */
/* harray is an instantiated hash object */
  rc=harray.num_items;
  put 'Our list has ' rc ' items';
  if rc > 0 then put 'Our list is not Empty';
  else put 'Our list is Empty';
```

Implementing the IS_EMPTY method using the HEAD method value is simply a matter of testing if the HEAD method value is missing. The code for this method then might look like:

```
  if head_key = . then put 'Our list is Empty';
  else put 'Our list is not Empty';
```

RETRIEVE METHOD

The RETRIEVE method is used to return node data values. It is one of the simpler methods to implement in that we simply use the hash object FIND method to return the node data associated with a particular key. Most of the complications associated with the RETRIEVE method are associated with guarding against the unplanned overwriting of node data fields in use in the datastep.

As the RETREIVE method will often be utilizing a NEXT or PREV value from a previous retrieval we need to briefly mention a SAS 9.1.3 software bug that causes unexpected hash behavior when using the value from a node data field as the key in following node retrievals. To avoid the bug behavior it is necessary to follow a simple, albeit odd coding pattern. For numeric data use code like:

```
rc=htest.find(key:numeric_data + 0);
```

For character data you do:

```
 rc=htest.find(key:character_data ||'');
```

Notice that we involved the key variable in a trivial function that will not alter the original value. This is sufficient to guard against the bug. For more information on this bug see the following document.

Problem Note 13188: DATA step component object FIND method not updating properly

http://support.sas.com/kb/13/188.html

This issue has been fixed in SAS version 9.2.

TRAVERSE METHOD

The TRAVERSE method is a sequential access method that can be implemented through numerous means depending upon the type of list implemented and the underlying hash object that was constructed. To implement a TRAVERSE method our examples will depend on two coding patterns implemented singly or in conjunction. These code patterns:

- Utilize the hash iterator to move from node to node based upon carefully ordered node keys
- Utilize predefined NEXT and/or PREV hash object node data fields to point us to neighboring nodes

Most of our examples will make use of explicitly defined link variables NEXT and/or PREV to provide information on neighboring nodes but one example will not use NEXT or PREV. The lack of an explicitly defined “link” variable creates what was previously referred to as an implicitly linked list. The specific properties of this kind of list will be discussed in a later section.

Let’s take a closer at the TRAVERSE patterns we want to use.

TRAVERSE METHOD USING THE HASH ITERATOR FUNCTIONS

Using the hash iterator would seem to be a no brainer for implementing the TRAVERSE method and in some cases it is exactly what we need. If we are careful to index our list nodes with keys that create the necessary order, the hash iterator performs quite nicely as a TRAVERSE method for lists that only perform actions at the “ends” of the list, as

defined by the hash object functions, “FIRST” and “LAST”. It is also quite useful when we need to traverse over the whole list.

The hash iterator though performs poorly when we need to implement a TRAVERSE method that allows us to start anywhere in the list and then traverse away from the starting point. In this case the hash iterator would have to start at one end of the list, iterate (TRAVERSE) to the starting point, then traverse nodes in the desired direction away from the relative starting node. Once at the starting point the hash iterator makes it simple to traverse in either direction but the overhead of traversing to the starting node can add a significant amount of unnecessary processing overhead.

TRAVERSE METHOD USING NEIGHBORING NODE KEYS STORED IN NODE DATA

One method for getting around the limitations of the hash iterator functions is to store one or both of the neighboring nodes index keys within each individual node’s data. In the included examples this is done by creating NEXT and/or PREV node data fields that are populated with the neighboring node keys as nodes are inserted. This allows us to retrieve a single node and then traverse in any direction that is available via the NEXT and PREV fields that have been defined. The logical steps involved in this type of traversal are:

1. Retrieve the starting node (RETREIVE). The starting node might be a node with a special property, such as a HEAD node, or a node with a specific index key.
2. If the appropriate link variable (NEXT or PREV) retrieved in the previous step contains a missing value then stop TRAVERSE process because a following node does not exist
3. Use the NEXT or PREV value, as appropriate, as the key for retrieval (RETRIEVE) of the following node
4. Go back to step 2

This method pattern functions well but it does have some drawbacks. It increases the amount of data that must be stored in the hash object and it increases the processing that must be completed in order to implement INSERT and DELETE_NODE methods. The increased processing is due to the need to update neighboring NEXT and PREV fields during these operations. This method’s shortcomings though can be an acceptable trade off for the shortcomings of the hash iterator solution which needs to start at a list end.

This is a good place to point out that we haven’t been exactly true to the spirit of linked lists, as implemented/described above. Linked lists are structures that generally allow only for sequential access of data with the exception of HEAD nodes that are available directly by nature of an externally stored and updated “pointer”(in our case a variable holding the head node index key). The fact that we are building our lists on hash objects gives us this additional functionality of a built in random access mechanism. This bit of functionality is something we will want to take advantage of when convenient but it is also a property we might wish to ignore when trying to implement a specific documented algorithm that depends on the functionality of a list based information structure that provides sequential access.

UPDATE METHOD

The UPDATE method is used to update node data values. It is easily implemented using the hash object REPLACE function. The most common mistake made with UPDATE methods is forgetting to update all of the node data fields appropriately even if only one field is being changed.

One safe pattern is to:

1. Retrieve any data you need from nodes other than the node to be updated and save to alternate variables
2. Retrieve the node that is to be updated
3. Modify any data fields that are to be updated
4. Make sure to set the node key variable appropriately if it was not included in the hash data fields (not auto-updated during a RETRIEVE)
5. Update the node using the hash object “replace” function.

INSERT METHOD

The INSERT method is used to add a new node to the hash table. While the act of actually adding a new node may be straightforward using the hash object ADD function, an insert may require:

- Testing if this will be the first node and updating NEXT and/or PREV values appropriately if used
- The new node to have its key value calculated using the key values in use by the soon to be neighboring node(s)
- Updating the new neighboring node(s), NEXT and/or PREV node key values to point to the newly inserted node
- Updating the HEAD value if the new node will also be the new HEAD node

DELETE_NODE METHOD

The DELETE_NODE method removes a node from the hash table. This function is basically the reverse of the INSERT function. We can easily remove a node from the hash table using the hash object REMOVE function but depending on the type of list being implemented we may need to:

- Update the current neighboring nodes NEXT and PREV values to point to one another rather than the soon to be deleted node(if NEXT and PREV are used)
- Update the HEAD value if the current node(about to be deleted) is the current HEAD node. To properly update the HEAD value it may be necessary to test if the soon to be deleted node is the only node in the hash object.

DELETE_LIST METHOD

The DELETE_LIST method is used to delete the hash object. This occurs normally at the termination of the data step but if there is a need to delete the hash object during data step execution the hash object DELETE function provides an easy mechanism for performing the deletion. This method might be of interest when there is the need to iterate over lists, where lists need to be periodically deleted, reconstructed and populated with new data.

INFORMATION STRUCTURES

The individual information structures will now be covered.

LINEAR SINGLE LINKED LIST

The Linear Single Linked List is simply an ordered list of 0 or more nodes that are indexed in such a way that each node contains a data element that defines the next, but not the previous, node in the list. This means that having only NEXT available, limits us to traversing the list in only one direction. While this is technically true by definition of a single linked list, the fact that we are implementing our information structures using hash tables, with properly ordered numeric keys, means we will actually always have more functionality available. As we work our way up to more complicated structures these dis-congruencies between the structure we are trying to implement and what is possible via the underlying structure, the hash object, will become less apparent.

As this is the first information structure to be covered, the code will be covered in more detail than other sections to help illustrate how to build and use hash based list structures. Generally there will be a short discussion of a section of code followed immediately by the code block just discussed.

To reduce the complexity of the example code we will be using a data _null_ step. The first important statement is the declaration and initialization of the variable head_key. The variable head_key will be used to identify the first node in our list and whether there is a first node. If no data exists in our list then head_key is set to missing. The retain statement is not specifically necessary in this example but often you will want to retain this value.

```
data _null_;
  retain head_key .; /* HEAD */
  /* number of list items to create. Only important here to create
     a small set of example data in a later step */
  num_items_create=6;
```

Next we create the hash object that will hold our single linked list. Care should be taken to plan what data should be contained within the hash object. The following CONSTRUCTOR code has the properties:

- Fields
  - location – numeric field that contains the hash object key
  - ikey – numeric data field that will always match the node key (location). This example could have added location as both a key and a data field but we will use the second field “ikey” to illustrate when location is not updated.
  - next – numeric field that contains the key to the following node if a following node exists. If a following node does not exist NEXT is set to missing. The field NEXT is what creates the “single link” property of our list.
  - char_data - character field of dummy data used to simulate additional node data
  - num_data – numeric field of dummy data used to simulate additional node data
- Note that we have not defined a hash iterator.

```
/* CONSTRUCTOR */
  if _n_=1 then do;
    length location ikey next 8 char_data $ 1 num_data 8;
    declare hash harray();
    harray.definekey("location");
    harray.definedata( "ikey", "next", "char_data", "num_data");
    harray.definedone();
    call missing(location, ikey, next, char_data, num_data);
  end;
```

Now that we have our list information structure constructed lets test if there is any data in the list. For our single linked list that has a HEAD value defined we will rely on the value contained within the head_key variable to identify if the list is empty. Such a test would look like:

```
  /* Test 1 - IS_EMPTY */
  put 'Test 1 - List should be empty';
  if head_key = . then put 'Our list is Empty';
  else put 'Our list is not Empty';
```

But what if we didn't have a HEAD value defined for this list? Well in that case a convenient method would be to use the hash object attribute "num_items" to query the number of items in the list (hash object). Implementing this logic would look something like:

```
/* Test 2 - IS_EMPTY */
  put 'Test 2 - List should be empty';
  rc=harray.num_items;
  put 'Our list has ' rc ' items';
  if rc > 0 then put 'Our list is not Empty';
  else put 'Our list is Empty';
```

In order to do something productive with our list we need to add some nodes of data. The following code loads some dummy data into the list. Note that care is taken to load the data in an orderly fashion so that the keys are ordered and the NEXT values point to the following node. We also update the head_key value.

```
  /* INSERT 1 - load a group of ordered data into our array list */
  do i=1 to num_items_create;
    c=byte(i+64); /* on a windows machine the byte function will return a
                     letter for the range of values used in this example */
    n=i*2; /* this is just to create some numeric data */

    ikey = i;
    if i < num_items_create then next=i+1; /* create a single link to the next item
*/
    else next=.;

    /* we are going to purposely set the head key to the first key */
    if i=1 then head_key=1; /* HEAD */

    /* INSERT */
    rc=harray.add(key:i, data:ikey, data:next, data:c, data:n);
  end;
```

Now what if we want to add a new node to the beginning of the list? To perform this task we need to complete the steps

1. Test if there is already data in the list. This is important so that we know how to initialize the new node key and NEXT value.
2. Insert the new node with the appropriate key and NEXT value. We don't have to worry about a previous first node, if one exists, because it doesn't contain any information on previous nodes.
3. Update the head_key value to the index key used by the new node

The implementation of this logic would look like:

```
  /* INSERT 2 */
  if head_key ne . then do; /* HEAD */
    put 'We are about to add a new first node to a non-empty list';
    /* INSERT */
```

```
      rc=harray.add(key:head_key - 1, data:head_key - 1, data:head_key, data:'Z',
  data:9);
      head_key=head_key - 1; /* update HEAD */
    end;
    else do;
      put 'We are about to add a node to an empty list';
      /* INSERT */
      rc=harray.add(key:1, data:1, data:., data:'X', data:9);
      head_key=1; /* update HEAD */
    end;
```

Now we want to look at how to insert a node following an existing node. This leads us to an interesting difference between the general implementation of a list and the implementation we are considering based on the hash object. Usually a single linked list only provided sequential access to data. That is when you want to locate a node within the list you have to start at one end of the list and use the available node links (NEXT and/or PREV) to traverse the list and locate the desired node. But in our case, having implemented our list on top of a hash object, we enjoy the ability to have random access or in other words we have the ability to locate (return the data from) specific nodes without the necessity of traversing the list. Because we have random access in this case we will locate the node we wish to follow by using a single application of the hash "find" method. A later example will illustrate when we still need to use sequential access.

Now back to the action of inserting a new node following an existing node. The steps we will use to complete this task are

1. Locate the node we wish to insert after, and RETRIEVE the NEXT value
2. Calculate a key for the new node based on the keys of the soon to be neighboring node keys. In our case we will make a key that will numerically be halfway between the two neighboring keys.
3. Store the current NEXT value of the node that will precede our new node so that the new node can be updated with the correct NEXT value.
4. Update the NEXT value of the node that will precede our new node with the node key calculated in step 2.
5. Add the new node using the key calculated in step 2 and the NEXT value stored in step 3.

Coded this would like:

```
    /* INSERT 3 */
    /* Variable insert_after will hold the index key for the node we wish to follow
  */
    insert_after=4;
    rc=harray.find(key:insert_after); /* RETRIEVE */
    new_node_key  = (ikey + next) / 2 ;
    new_node_next = next;
    next=new_node_key;
    location=ikey;
    rc=harray.replace(); /* UPDATE */
    rc=harray.add(key:new_node_key, data:new_node_key, data:new_node_next, data:'Y',
  data:0); /* INSERT */
```

We have one more INSERT method to consider but first let's consider the TRAVERSE method.

Strictly speaking TRAVERSE provides us with a mechanism for accessing nodes in a sequential manner. Because our current example is based on a single linked linear list we only have enough information contained in each node to direct us to the following node thus allowing us to traverse the list in only one direction. The steps for traversing the entire list would be

1. Retrieve the starting node.
    a. In our example that means using our HEAD method if we want to start at the first node. If the HEAD value is missing then we do not have any data in our list and the TRAVERSE method stops processing.
    b. If we want to start at a specific node with a known key then we retrieve the known node by its provided key.
2. Test the NEXT value retrieved in the previous step to check if there is a following node (NEXT value is non-missing). If NEXT is missing TRAVERSE stops.
3. Retrieve the node with index key matching NEXT from the previous step.
4. Go back to step 2.

An example TRAVERSE method that begins at the HEAD node and then traverses the list, writing all the hash data to the log would be:

```
/* TRAVERSE 1*/
location=.; /* re-initialize location variable to show it is not updated */
put 'Data returned in TRAVERSE 1';
/* Do we have any data to traverse? if so then step through all the data */
if head_key ne . then do; /* HEAD */
  put 'location  ikey  next  char_data  num_data';
  put '----------------------------------------';
  next=head_key; /* HEAD */
  do while(next ne .);
    rc=harray.find(key:next+0); /* TRAVERSE and RETRIEVE */
    put location +7 ikey best3. +4 next best3. +5 char_data +9 num_data;
  end;
end;
put '========================================';
put;
```

Now lets consider our last INSERT example. We now know how to insert a value at the beginning of a list, and within a list but let's consider how we might go about adding a value to the end of our list. This task is different from our previous two insert scenarios because we do not know what node is currently the last node. We need to find the current last node because we need to update the NEXT field in the current last node to point to the new last node we will be inserting. Since we don't have the current last node stored in some kind of HEAD value we will need to traverse the list and find the last node. Once we find the current last node we can add our new last node and update the old last node's NEXT value to point to the new following node. There is one special case to consider though. What if the list does not currently contain any data? In this case we would need to add the new node to the list and update head_key. All of this logic can be contained in the code:

```
/* INSERT 4*/
/* The purpose of this step is to add a new node at the end
   of the list */
/* Do we have any data to traverse? if so then step through
   all the data and find the last node */
if head_key ne . then do; /* HEAD */
  put 'We are about to add a new last node to a non empty list';
  /* there is data in the list */
  next=head_key; /* HEAD */
  do while(next ne .);
    rc=harray.find(key:next+0); /* TRAVERSE and RETRIEVE */
  end;
  next=ikey + 1;
  location=ikey; /* index key for current last node */
  rc=harray.replace(); /* UPDATE */
  rc=harray.add(key:next+0, data:next+0, data:., data:'Z', data:-1);
end;
else do;
put 'We are about to add a new last node to an empty list';
  rc=harray.add(key:1, data:1, data:., data:'Z', data:-1); /* INSERT */
  head_key = 1; /* HEAD */
end;
put '========================================';
put;
```

The previous code might make one consider that if the hash object had used an alternate constructor that included ordering the hash object, via the "ordered" option and including a hash iterator it would have been possible to directly access the current last node via the hash "last" function. This will be an important property to consider when planning a CONSTRUCTOR but it was purposely not used in this example in order to illustrate the action of traversing the whole list to locate a node with a particular property or key. In this example we are searching for a node with the property "last" as defined by our logic.

Now we have reached the point of considering methods for deleting nodes. First we will consider how we would delete the first node. This turns out to be fairly easy for this list because we know which node is first because of our HEAD method and we don't have to update the link to neighboring nodes because the first node does not have any nodes pointing to it. Thus we just need to delete the first node, if it exists, and update the HEAD value appropriately. Coded this would look like:

```
/* DELETE_NODE 1*/
/* Purpose of this step is to delete the first node */
```

```
    if head_key ne . then do;
      put 'A first node is available for deletion';
      rc=harray.find(key:head_key);
      rc=harray.remove(key:head_key);
      head_key=next; /* this works even if we delete the only node */
    end;
    else put 'A first node was not available for deletion';
    put '=======================================';
    put;
```

Note that in the next example the previous code is assumed to be contained within a block of code labeled "delete_first_node". It was presented without link and return statements to make the purpose clear.

What if we want to delete a node that isn't the first node? Well in that case we need to consider how we find the node to be deleted and then also how we update the node that currently points to the soon to be deleted node. For the current list, all nodes except the first will have a node pointing to them via their node data value NEXT. A code version of this would look like:

```
    /* DELETE_NODE 2*/
    /* Purpose of this step is to delete a specific node within the list.
       We will assume that we know the node exists which also implies
       that the list is non-empty */
    delete_node=2;

    if head_key = delete_node then do;
      /* We link to a general method here */
      link delete_first_node; /* see the DELETE_NODE 1 code block */
    end;
    else do;
      next=head_key; /* HEAD */
      /* first locate the node preceding the node we are about to delete */
      do while(next ne delete_node);
        rc=harray.find(key:next+0); /* RETRIEVE */
      end;
      preceding_node=ikey;
      /* retrieve the data from the node marked for deletion. next+0 = delete_node */
      rc=harray.find(key:next+0); /* RETRIEVE */
      preceding_node_new_next=next;
      rc=harray.remove(key:ikey);
      rc=harray.find(key:preceding_node);
      next=preceding_node_new_next;
      location=preceding_node;
      /* now update preceding node to have the next value that the deleted node
  contained */
      rc=harray.replace(); /* UPDATE */
      rc=harray.find();
    end;
```

The last method to consider is how to delete a list. It is a trivial matter. Just submit the following.

```
    /* DELETE_LIST 1 */
    rc=harray.delete(); /*DELETE_LIST */
```

LINEAR DOUBLE LINKED LIST

The Linear Double Linked List is an ordered list of zero or more nodes that are indexed in such a way that each node contains a data element that defines both the next and the previous, node in the list. The only exception is the end nodes which contain a missing value for either the NEXT or PREV value depending on which end of the list the nodes are located. Refer to Illustration 3 for a graphical representation of this particular list.

Because the list methods for linear double linked lists are similar to linear single linked lists and the single linked list methods were thoroughly covered we will only briefly mention a few distinguishing points for the double linked variety. The primary points of concern are:

- The CONSTRUCTOR will need to account for both NEXT and PREV values.
- When processing INSERT or DELETE_NODE methods it will be necessary to update both neighboring

nodes link data to point to either the new node in the case of an INSERT or point to one another in the case of a DELETE_NODE. In the case of INSERT the new node must point to two neighbors, assuming we are not inserting at an end.

STACK

Stacks are information structures based on lists where the methods INSERT and DELETE_NODE are restricted to functioning at only one particular "end" of a single linked list. The node data link will always point to the node that was added before the current node.

A common stack example is to consider a can of tennis balls. As balls are added to the can (INSERT) only the last ball added to the can is accessible at any one time (RETREIVE). This also implies that if say three balls have been added to the can and we want to retrieve the middle ball we need to retrieve the top ball (last ball added) so that the middle ball will be available for access.

Generally speaking stacks operate by using three primary methods called "push", "pop" and "peek". From the standpoint of the list functions we have been using we can define each stack function as:

- Push – INSERT, always performed at the same end of the list
- Pop – RETRIEVE followed by DELETE_ITEM, always performed on the last item inserted
- Peek – RETRIEVE, always performed on the last item inserted

Now that we have some understanding of the stack's structure, constraints and methods let's consider how we might construct the underlying hash object to implement a stack. This brings us to two important, related questions.

- Do we want to use explicit or implicit links?
- Do we want to use a HEAD function that updates a head key variable to track the last node inserted or do we want to use an ordered hash object that provides the hash functions "first" and "last" to implement our HEAD function?

It is possible to construct a usable hash object regardless of our answers to these two questions but the amount of code we must write to implement the necessary methods will probably be the factor that pushes us to a particular solution. In our case we should consider

- All our inserts and deletes will occur at the end of the list and each node contains a link to only the preceding node. This means that when inserting or deleting a new node we do not need to update values in any other node. Additionally this provides us the ability to use regular, equally spaced node keys such as integers.
- If regularly spaced item keys can always be expected then we could depend on implicit links without having the need to store and maintain explicit links within our node data.
- A HEAD method that utilizes an updated variable to hold the node key for the head node is probably the best way to go if the stack is going to be large but the use of an ordered hash object makes the coding a lot easier and causes limited performance issues when used with smaller sized lists. Small and large in this case are pretty arbitrary though.

Considering the points above, the example stack code implementation will have the following properties

- Numeric integer node keys
- *Implicit* links between nodes (no explicit link data stored in the node)
- The underlying hash object will be ordered allowing us to use the "last" function to implement our HEAD method.

The example itself will showcase how we can implement much of the necessary functionality by utilizing "link/return" statements to encapsulate the stack methods within a data step that used a do-while loop to read through a dataset. While the example is contrived, the pattern for working with list functions in this manner can be beneficial. The code for this information structure will not be covered in detail, beyond the comments included within the code itself.

STACK CODE EXAMPLE

```
data _null_;
  /* CONSTRUCTOR */
  if _n_=1 then do;
    length location location 8 char_data $ 1 num_data 8;
    declare hash hstack(ordered:"a");
    declare hiter istack("hstack");
    hstack.definekey("location");
    hstack.definedata("location", "char_data", "num_data");
    hstack.definedone();
```

```
  call missing(location, char_data, num_data);
end;

/* loop through a dataset and add some data to our stack */
do while(not eof);
  set sashelp.class end=eof;
  /* INSERT */
  link PUSH; /* add the sex and weight of each student to the stack */

  sex=.; char_data=.; /* sex value is lost due to processing logic */

  /* For some contrived reason we now need to remove the last stack value if
     it was for a females or anyone under 14. Note that age is still
     retained for current stack value but not sex */
  if age < 14 then link POP; /* remove those under age 14 */
  else do; /* otherwise check for females */
    link PEEK; /* RETREIVE */
    if char_data='F' then link POP;
  end;
end;

/* Now lets output the data to see what we have just to make sure
   we have what we expected */
link number_of_nodes;
put 'Number of items in stack: ' rc;
put;

put 'Stack_Location char_data(sex) num_data(weight)';
put '-------------------------------------------';

/* this loop depends on the leave function */
do while(1);
  link IS_EMPTY;
  if rc=0 then leave;
  link POP;
  put location +17 char_data +12 num_data;
end;
stop;

HEAD: /* this will also be used  as an embedded IS_EMPTY function */
rc=istack.last();
/* if rc ne 0 then no data was available */
return;

IS_EMPTY:
/* this will be our main IS_EMPTY function because it won't overwrite
   node data values when called */
rc=hstack.num_items;
/* if rc=0 then no data is available */
return;

number_of_nodes:
/* not used here but added to show an example of a helper method */
rc=hstack.num_items;
/* rc=number of items in the stack */
return;

PUSH:
link HEAD;
/* if there isn't a head node then set first location value */
if rc ne 0 then location=0;
location=location+1;
/* the next two lines for setting char_data and num_data migh have
```

```
   needed to live outside this link code in order to deal with the
   specific logical needs but in this case it works well */
char_data=sex;
num_data=weight;
rc=hstack.add(); /* INSERT */
return;

POP:
link HEAD;
if rc = 0 then do;
  /* if there is a head node then delete it*/
  /* we will use the next function to release the hold the
     iterator has on the last value from our use of the "last" function.
     The "next" function does not actually succeed since we are already
     at the last value. */
  rc=istack.next();
  rc=hstack.remove(); /* DELETE_NODE */
end;
return;

PEEK:
link HEAD;
rc=istack.next(); /* see POP for explaination for this line */
return;
run;
```

DEQUE AND QUEUE

The moment you see “QUEUE” it would be expected for you to be saying to yourself “don’t we have the lag function to implement as a queue?”. Well yes, but if you have ever tried to use the lag function in a complicated data step where you were not using the default looping mechanism you know how hard it can be to correctly use the lag function. That is why we will consider a queue structure here. The queue we will discuss will be customizable and controllable. But first let’s take a closer look at what deques and queues are from a list perspective and also show how the stack we just considered is actually related.

Knuth[1] defined stacks, queues and deques as

- A *stack* is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list.
- A *queue* is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.
- A *deque* (“double ended queue”) is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list.

From this description is can be seen that deque (pronounced “deck”, like a deck of cards) is the most general of the three structures but each is built upon linear lists that have specific constraints on how the list methods can affect the list.

The queue and deque can both be implemented in a manner similar to the stack presented above. One difference though is that since it is important to know which end a method is to function upon, we will label the ends “LEFT” and “RIGHT” to make it clear to which end we are referring.

When it comes to working with a deque it might be helpful to imagine two stacks where the heads become “LEFT” and “RIGHT” and the “other” ends are pushed together in the middle of the new list. With that in mind it would allow us to use our previous stack code methods with only slight modification. In particular we would need to specify which end we were trying to perform HEAD, INSERT or DELETE_NODE methods upon. To do this you might create new methods like “LEFT_POP” and “RIGHT_POP” or perhaps just create a new HEAD method that used a second variable to identify if you wanted to act upon the LEFT or RIGHT end. To make this change you could simply use the “first” and “last” hash object functions as appropriate to act upon LEFT or RIGHT.

If you follow the deque logic presented but then constrain your list to only allow POP to occur at one end and PUSH to occur at the other end you wind up with a queue. This queue will be under your explicit control unlike the lag function which depends on the default data step looping mechanism. So while implementing this custom queue might require more code than a simple lag function it might also provide a lot more functionality when implementing complicated algorithms that rely on a queue.

Code examples for queues and deques will not be included.

ARRAYLIST

For our discussion an arraylist is an array-like structure that can dynamically resize itself as needed. Think of it as a structure that parallels the standard data step array with the added properties that it doesn't need to know how many values it will contain at compile time and it can contain multiple data item fields. Implementing this information structure in the datastep is similar to implementing the list based queue discussed above. The points to consider are

- UPDATE is allowed to update node data but not the node's index key
- INSERT is handled explicitly and only allowed at the ends (no PUSH here)
- RETREIVE is handled explicitly but only allowed at the ends (no POP – RETREIVE and DELETE_NODE)
- DELETE_NODE is handled explicitly and only allowed at the ends (no POP)

With these points in mind it should be straightforward to implement an arraylist. Of course the performance of a native SAS array will be far superior to any arraylist based upon a hash object but there may be cases where an arraylist is just the tool needed to solve a problem.
No code example for arraylist's is included in this paper.

CIRCULAR LINKED LISTS

As SAS programmers we are used to finding clever ways to deal with linear list of items. Circular lists structures though are a very different issue. If you want to implement a circular list structure without a list based hash object you might likely wind up with a solution that makes clever use of arrays or the set statement using the point option. The code logic though can get fairly onerous as the size of the list grows, the number of data items to deal with increases or functions such as insert and delete need to be implemented. This leads to why circular lists based upon the hash object are so useful. The hash object makes all the hard stuff straightforward. If we need to do an INSERT, the hash will happily accept another value. If we need to do a RETRIEVE or a DELETE_ITEM, these methods are also easily done with a hash based list.

CIRCULAR SINGLE LINKED LISTS

Circular single linked lists will receive only a passing mention here. The circular double linked list discussion in conjunction with the information presented on linear single linked lists should provide sufficient details for implementing circular single linked lists. See Illustration 4 for a graphical representation of circular single linked lists.

CIRCULAR DOUBLE LINKED LISTS

Circular double linked lists may be the most necessary and thus the most powerful hash/list based information structure presented in this paper. It's necessary because there are few ways to deal with data as a circular list while also having the ability to:

- expand its size to accommodate any amount of data (within system limits of course)
- allow deletion of any node
- allow new nodes to be inserted anywhere in the list.
- provide sequential access to list nodes
- provide random access to list nodes
- hold multiple items of node data in a single node (key)

Certainly there are clever, but complicated, ways to accomplish these properties through arrays and possibly datasets but the author feels the hash based list methods are superior due to the straightforward manner in which the list can be used once the appropriate list methods are coded.

Code examples for implementing circular double linked lists will not be presented. Review Illustration 5 and the discussion of linear single linked lists for code patterns necessary to implement this information structure.

APPLICATION – CIRCULAR DOUBLE LINKED LISTS

The author came to believe in the power of list based information structures based on data step hash objects while working to write custom code for various GIS related tasks.

One of the difficult issues one faces when working with map datasets within the data step is the need to deal with individual polygons of map data. Large complicated polygons are most easily represented as a circular list of coordinates. But without a dependable and consistent information structure to hold this kind of data, it is hard to

implement graphical and GIS algorithms using the SAS data step. Specifically the author found that implementing circular double linked lists was the only way to handle the problem of clipping polygons to other polygons.

A brief pictorial list of the problems that pushed the author to hash based lists is shown in Illustrations 6, 7 and 8. Illustration 6 was meant to simply create a contour map (actually only contour lines) over a map of the lower 48 states. The circles are the location of the data used to interpolate the surface.

Illustration 7 quickly followed Illustration 6 because well, everything looks better in color. Illustration 8 is where things really got complicated though. The goal was to take the data from say illustration 7 and clip off all the information outside the boundary of the lower 48 states. There are a number of ways to make it “appear” that the map has been clipped but the author wanted to find a solution for true clipping of the image so that the output image size would be reduced and a general solution could be created for generating similar maps for any region.

Illustration 8 is Proc GMAP output of Illustration 7 data after applying the clipping algorithm and transforming the data to a SAS Map dataset. Illustrations 6 and 7 were created by plotting raw line and polygon data using the SAS Data Step Graphics Interface (DSGI).

ILLUSTRATION 6:

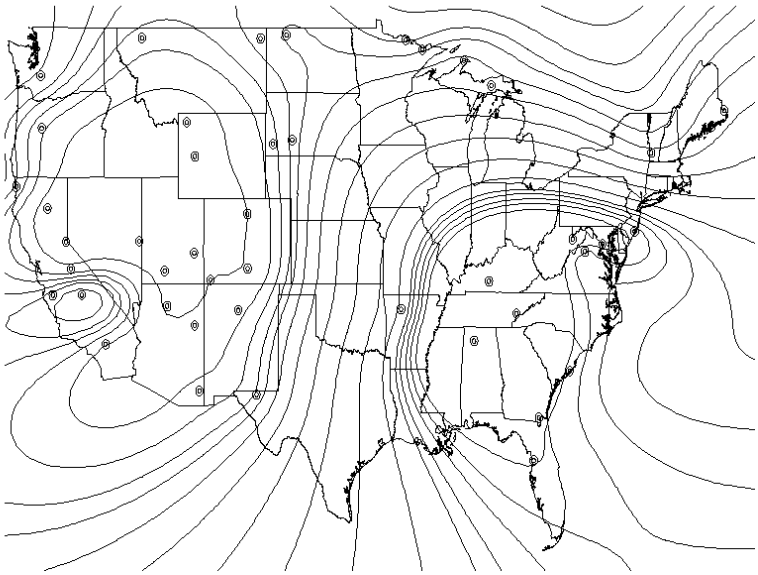


ILLUSTRATION 7:

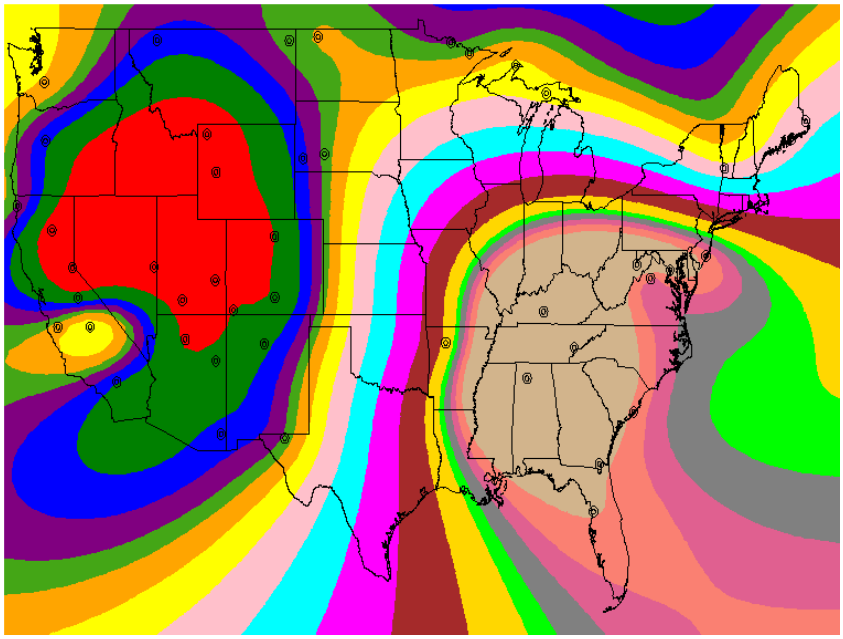
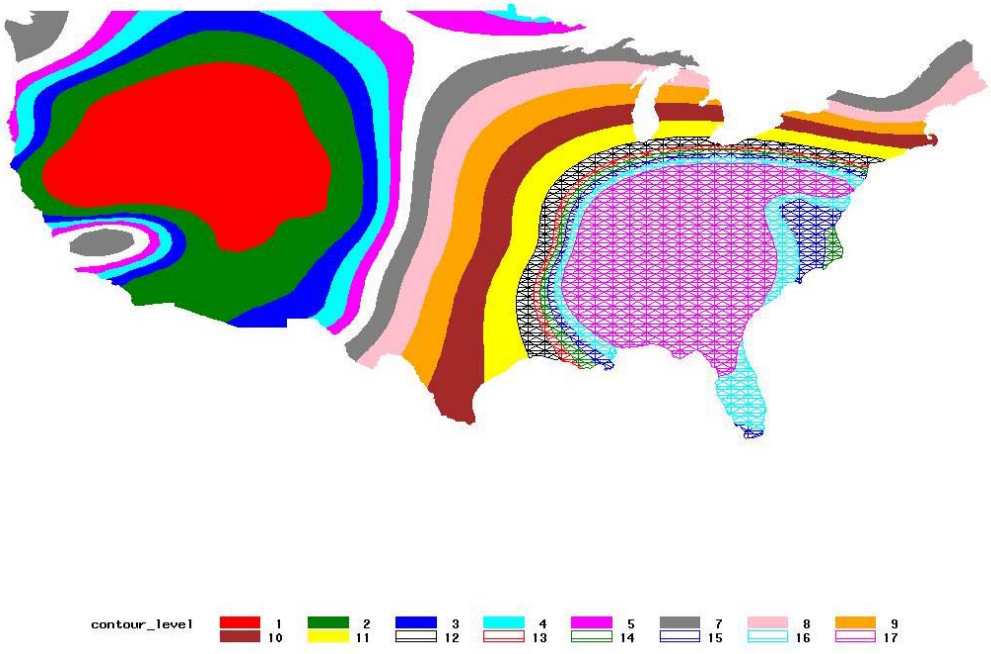


ILLUSTRATION 8:



CONCLUSION

List based information structures based upon the SAS version 9.1.3 hash object can provide SAS programmers with a wide range of coding flexibility when implementing complicated algorithms that call for list based information structures. The coding can be somewhat involved but it is straightforward and consistent. The addition of these information structures should make it possible for SAS programmers to tackle complicated problems that in the past would have been relegated to one of the traditional compiled programming languages.

REFERENCES

1 Knuth, The Art of Computer Programming, 1st edition, Vol. 1, p235

CONTACT INFORMATION

Contact the author at:

Shawn Edney
ThotWave Technologies
510 Meadowmont Village Circle #192
Chapel Hill, NC 27517
800-591-THOT
sedney@thotwave.com
http://www.thotwave.com