```
/** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */
                    ALGORITHMS – SINGLY LINKED LINEAR LIST
/** ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ */


Function CreateNode(FIRST)
    Given a list pointed by FIRST this function acquires a node from the
    availabiity stack, whose Top is indicated by AVAIL. NEWW indicates a
    node pointer. The function returns a node pointer.

1.  Acquire a Node from Availability Stack
        NEWW = AVAIL
        AVAIL = LINK(AVAIL)

2.  Confirm Allocation
        If NEWW == NULL
            Write('Availability Stack UNDERFLOW.')
            Exit
        Else
            Write('Node Created')

3.  Return the Node
        Return NEWW
```

Function **InsertAtBeg(FIRST, KEY)**

Given a list pointed by FIRST and the data value KEY, this function inserts a node indicated by NEWW at the beginning of the list.

1. Create a Node

```
NEWW <== NODE
```

2. Is the Node available to use??

```
If NEWW = NULL
    Return FIRST
```

3. Initialize the Node

```
DATA(NEWW) = KEY
LINK(NEWW) = NULL
```

4. Is the list Empty??

```
If FIRST = NULL
    Return NEWW
```

5. Insert the node and return the list

```
LINK(NEWW) = FIRST
Return NEWW
```

```
Function InsertAtEnd(FIRST, KEY)
    Given a list pointed by FIRST and the data value KEY, this function
    inserts a node indicated by NEWW at the end of the list.

1.  Create a Node
        NEWW <== NODE

2.  Is the Node available to use??
        If NEWW = NULL
            Return FIRST

3.  Initialize the Node
        DATA(NEWW) = KEY
        LINK(NEWW) = NULL

4.  Is the list Empty??
        If FIRST = NULL
            Return NEWW

5.  Save the list pointer, FIRST
        TEMP = FIRST

6.  Traverse to the last node in the list
        Repeat Until LINK(TEMP) <> NULL
            TEMP = LINK(TEMP)
```

```
7.  Insert the node and return the list
        LINK(TEMP) = NEWW
        Return FIRST


Function DisplayLL(FIRST)
    Given a list pointed by FIRST this function prints the contents of
    the list.

1.  Is the list Empty??
        If FIRST = NULL
            Write ('The List is Empty...')
            Return

2.  Set the display prompt
        Write('List Contents: ')

3.  Traverse the list and print the node contents
        Repeat Until FIRST <> NULL
            Write('-->', DATA(FIRST))
            FIRST = LINK(FIRST)
        Return
```

```
Function LengthLL(FIRST)
    Given a list pointed by FIRST this function returns length of the list.
    LENGTH is local variable indicating length of list.

1.  Initialize LENGTH
        LENGTH = 0

2.  Is the list Empty??
        If FIRST = NULL
            Return LENGTH

3.  Traverse to the last node and increment LENGTH
        Repeat Until FIRST <> NULL
            LENGTH = LENGTH + 1
            FIRST = LINK(FIRST)

  /**Alternative to Step-3
    3.  Reset Length and Traverse the List till last Node
            LENGTH = 1
            Repeat Until LINK(FIRST) <> NULL
                FIRST = LINK(FIRST)
                LENGTH = LENGTH + 1
  */

4.  Return the Length
        Return LENGTH
```

Function **InsertAtPos(FIRST, KEY, POS)**

Given a list pointed by FIRST and the data value KEY, this function inserts a node indicated by NEWW at the indicated position, POS in the output list. LEN indicates the total number of nodes in the pointed by FIRST. TEMP is a local list pointer. NDX is local variable indicating the node index.

1. Initialize local variables

```
NDX = 0
LEN = LengthLL(FIRST)
```

2. Is the position valid??

```
If POS < 1 OR POS > LEN
    Write('Invalid Position, Insert Failed')
    Return FIRST
```

3. Create a node

```
NEWW <== NODE
```

4. Is the Node available to use??

```
If NEWW = NULL
    Return FIRST
```

5. Initialize the Node

```
DATA(NEWW) = KEY
LINK(NEWW) = NULL
```

```
6.  Is POS the start of list??
        If POS = 1
            LINK(NEWW) = FIRST
            Return NEWW

7.  Save the list pointer, FIRST
        TEMP = FIRST

8.  Traverse to the node at POS-2 in the list
        Repeat Until NDX < POS-2
            NDX = NDX + 1
            TEMP = LINK(TEMP)

9.  Insert the node and rearrange list pinters
        LINK(NEWW) = LINK(TEMP)
        LINK(TEMP) = NEWW

10. Return the list
        Return FIRST
```

```
Function SortLL(FIRST, MODE)
    This function put the list pointed by list pointer FIRST in either
    ascending order or descending order (indicated by MODE), and returns
    the list pointer, FIRST. IPTR and JPTR are local list pointers. KEY
    is local variable to hold DATA.

1.  Is the list empty? or has a single node??
        If FIRST = NULL or LINK(FIRST) = NULL
            Return FIRST

2.  Save the list pointer
        IPTR = FIRST

3.  Apply Bubble Sort
        For IPTR = FIRST to LINK(IPTR) <> NULL Step LINK(IPTR)

            For JPTR = LINK(IPTR) to JPTR <> NULL Step LINK(JPTR)

                Select MODE from

                    Case 0: /* Ascending Sequence */
                            If DATA(IPTR) > DATA(JPTR)
                                KEY = DATA(IPTR)
                                DATA(IPTR) = DATA(JPTR)
                                DATA(JPTR) = KEY
```

```
                    Case 1: /* Descending Sequence */
                            If DATA(IPTR) < DATA(JPTR)
                                KEY = DATA(IPTR)
                                DATA(IPTR) = DATA(JPTR)
                                DATA(JPTR) = KEY
                End Select

4.  Return the list
        Return FIRST


Function InsertOrdered(FIRST, KEY)
    Given a ascending order list pointed by FIRST and the data value KEY,
    this function inserts a node indicated by NEWW in the list. TEMP is
    a local list pointer.

1.  Create a Node
        NEWW <== NODE

2.  Is the Node available to use??
        If NEWW = NULL
            Return FIRST

3.  Initialize the Node
        DATA(NEWW) = KEY
        LINK(NEWW) = NULL
```

```
4.  Is the list Empty??
        If FIRST = NULL
            Return NEWW

5.  If new node precedes first node
        If DATA(FIRST) >= DATA(NEWW)
            LINK(NEWW) = FIRST
            Return NEWW

6.  Save the list pointer, FIRST
        TEMP = FIRST

7.  Traverse till the predecessor of new node
        Repeat While LINK(TEMP) <> NULL and DATA(LINK(TEMP)) <= DATA(NEWW)
            TEMP = LINK(TEMP)

8.  Insert the node and return the list
        LINK(NEWW) = LINK(TEMP)
        LINK(TEMP) = NEWW
        Return FIRST
```

Function **DeleteAtBeg(FIRST, KEY)**

Given a list pointed by FIRST and a data variable KEY, this function deletes a node at the start of the list and updates KEY with data contents of the deleted node. It returns the updated list, FIRST. TEMP is a local list pointer.

1. Is List Empty??

```
If FIRST = NULL
    Write('Empty List, Delete Failed.')
    Return FIRST
```

2. Delete the Node

```
KEY = DATA(TEMP)    /*Save Data */
TEMP = FIRST
FIRST = LINK(FIRST)
Restore(TEMP)       /*Return Node to Availability Stack */
```

3. Return the list

```
Return FIRST
```

```
Function DeleteAtEnd(FIRST, KEY)
    Given a list pointed by FIRST and a data variable KEY, this function
    deletes a node at the start of the list and updates KEY with data
    contents of the deleted node. It returns the updated list, FIRST.
    TEMP is a local list pointer.

1.  Is List Empty??
        If FIRST = NULL
            Write('Empty List, Delete Failed.')
            Return FIRST

2. If the list has only node
        If LINK(FIRST) = NULL
            KEY = DATA(FIRST)
            Restore(FIRST)
            Return NULL

3.  Save the list pointer
        TEMP = FIRST

4.  Traverse till last node
        Repeat While LINK(LINK(TEMP)) <> NULL
            TEMP = LINK(TEMP)
```

```
5.  Delete the node and rearrange pointers
        KEY = DATA(LINK(TEMP))  /*Save Data */
        Restore(LINK(TEMP))     /*Return Node to Availability Stack */
        LINK(TEMP) = NULL       /*Set last link to NULL */

6.  Return the list
        Return FIRST
```

Function **CopyLL(FIRST)**

Given a list pointed by FIRST this function creates a duplicate of the list. The duplicate list is pointed by DUPL. NEWW, TEMP and CPY are temporary pointer variables.

```
1.  Is the list empty??
        If FIRST = NULL
            Return NULL

2.  Copy the first node.
        NEWW <== NODE
        DATA(NEWW) = DATA(FIRST)
        LINK(NEWW) = NULL
        DUPL = NEWW

3.  Save the list pointer.
        TEMP = FIRST
```

```
4.  Traverse remainder of the list
        Repeat While LINK(TEMP) <> NULL
            /* Update CPY */
            CPY = NEWW
            TEMP = LINK(TEMP)

            /*Copy the node */
            NEWW <== NODE
            DATA(NEWW) = DATA(TEMP)
            LINK(NEWW) = NULL
            LINK(CPY) = NEWW

5.  Set the link of last node and return list
        LINK(NEWW) = NULL; /*May be Omitted*/
        Return DUPL


Function reverseLL(FIRST)
    Given a list pointed by FIRST this function reverses the list. It
    returns the list pointer reversed list. PTR1, PTR2 and REV local
    list variables.

1.  Is the list empty??
        If FIRST = NULL
            Return NULL
```

```
2.  Initialize the pointers
        REV = FIRST                 /*Pointer to partially reversed list */
        PTR2 = LINK(LINK(FIRST))    /*Looking ahead by 2 nodes */
        PTR1 = LINK(FIRST)          /*Looking ahead a node */

3.  Put first node in reversed list
        LINK(REV) = NULL;
        LINK(PTR) = REV;

4.  Traverse remainder of the list and reassign node pointers
        Repeat While PTR2 <> NULL
            REV = PTR1
            PTR1 = PTR2
            PTR2 = LINK(PTR2)
            LINK(PTR1) = REV

5.  Return the reversed list
        Return PTR1
```

Function DeleteAddress(FIRST, NEWW)

Given a list pointed by FIRST and a pointer NEWW, which denotes the address of the node to be deleted, this function perform indicated deletion. TEMP is a local list pointer.

```
1.  Is List Empty??
        If FIRST = NULL
            Write('Empty List, Delete Failed.')
            Return FIRST

2.  Delete the first node??
        If TEMP = FIRST
            FIRST = LINK(FIRST)
            Restore(TEMP)
            Return FIRST

3.  Save the list pointer
        TEMP = FIRST

4.  Search for NEWW
        Repeat while LINK(TEMP) <> NULL and LINK(TEMP) <> NEWW
            TEMP = LINK(TEMP)

5.  Delete the node, if found and rearrange the pointers
        If LINK(TEMP) <> NULL
            LINK(TEMP) = LINK(NEWW)
            Restore(NEWW)
        Else
            Write('Node not Found, Delete Failed')
            Return FIRST
```

Function **mergeOrderedLL(FIRST, SECOND)**

Given two ordered [ascending sequence] lists pointed by FIRST and SECOND respectivley, this function combines these lists into a new list pointed by THIRD. NEWW and TAIL are temporary pointer variables.

1. Are both list empty??

```
    If FIRST = NULL and SECOND = NULL
        return NULL
```

2. Initialize the pointers

```
    THIRD = TEMP = TAIL  = NULL
```

3. Set the Search [both lists are non empty]

```
    Repeat While FIRST <> NULL and SECOND <> NULL
        NEWW <== NODE          /*Create Node*/
        LINK(NEWW) = NULL

        If DATA(FIRST) <= DATA(SECOND)
            DATA(NEWW) = DATA(FIRST)
            FIRST = LINK(FIRST)          /* advance FIRST */
        Else
            DATA(NEWW) = DATA(SECOND)
            SECOND = LINK(SECOND)        /* advance SECOND */

       IF THIRD = NULL
            THIRD = TAIL = NEWW
```

```
            ELSE
                 LINK(TAIL) = NEWW
                 TAIL = NEWW
```

4. If SECOND has exhausted, append nodes in FIRST to THIRD.

```
        Repeat While FIRST <> NULL
            NEWW <== NODE          /*Create Node*/
            LINK(NEWW) = NULL
            DATA(NEWW) = DATA(FIRST)
            FIRST = LINK(FIRST)
            LINK(TAIL) = NEWW
            TAIL = NEWW
```

4. If FIRST has exhausted, append nodes in SECOND to THIRD.

```
        Repeat While SECOND <> NULL
            NEWW <== NODE           /*Create Node*/
            LINK(NEWW) = NULL
            DATA(NEWW) = DATA(SECOND)
            SECOND = LINK(SECOND)
            LINK(TAIL) = NEWW
            TAIL = NEWW
```

5. Return the merged list

```
        Return THIRD
```

CIRCULAR LINKED LIST

Function **InsertAtBegLL(FIRST, KEY)**

Given a circular linked linear list pointed by FIRST and the data value KEY, this function inserts a node indicated by NEWW at the beginning of the list.

1. Create a Node

```
NEWW <== NODE
```

2. Is the Node available to use??

```
If NEWW = NULL
    Return FIRST
```

3. Initialize the Node

```
DATA(NEWW) = KEY
LINK(NEWW) = NULL
```

4. Is the list Empty??

```
If FIRST = NULL
    LINK(NEWW) = NEWW
    Return NEWW
```

5. Save the list pointer

```
TEMP = FIRST
```

6. Traverse till the last Node
```
    Repeat While LINK(TEMP) <> FIRST
        TEMP = LINK(TEMP)
```

7. Insert the node and return the list
```
    LINK(NEWW) = FIRST
    LINK(TEMP) = NEWW
    Return NEWW
```

Procedure **InsertAtBegLLTail(FIRST, TAIL, KEY)**

Given a circular linked linear list pointed by FIRST and the data value KEY, this procedure inserts a node indicated by NEWW at the beginning of the list. TAIL is input-output list pointer pointing to end of the list.

1. Create a Node
```
    NEWW <== NODE
```

2. Is the Node available to use??
```
    If NEWW = NULL
        Return FIRST
```

3. Initialize the Node
```
    DATA(NEWW) = KEY
    LINK(NEWW) = NULL
```

```
4.  Initialize Pointer
        TAIL = FIRST

5.  Is the list Empty??
        If FIRST = NULL
            LINK(NEWW) = NEWW
            TAIL = NEWW
            Return NEWW

6.  Insert the node and return the list
        LINK(NEWW) = FIRST
        LINK(TAIL) = NEWW
        Return NEWW
```

Procedure **InsertAtEndLLTail(FIRST, TAIL, KEY)**

Given a circular linked linear list pointed by FIRST and the data value KEY, this procedure inserts a node indicated by NEWW at the end of the list. TAIL is input-output list pointer pointing to end of the list.

```
1.  Create a Node
        NEWW <== NODE

2.  Is the Node available to use??
        If NEWW = NULL
            Return FIRST
```

```
3.  Initialize the Node
        DATA(NEWW) = KEY
        LINK(NEWW) = NULL

4.  Is the list empty??
        If FIRST = NULL
            FIRST = NEWW
            LINK(NEWW) = FIRST
            TAIL = FIRST
            Return(FIRST

5.  Insert the node and return the list
        LINK(TAIL) = NEWW
        LINK(NEWW) = FIRST
        TAIL = LINK(TAIL)
        Return(FIRST
```

Procedure **InsertOrderedLLTail(FIRST, TAIL, KEY)**

Given a ordered (increasing sequence) circular linked linear list pointed by FIRST and the data value KEY, this procedure inserts a node indicated by NEWW at the appropriate location in the list. TAIL is input-output list pointer pointing to end of the list.

1. Create a Node

```
NEWW <== NODE
```

2. Is the Node available to use??

```
If NEWW = NULL
    Return FIRST
```

3. Initialize the Node

```
DATA(NEWW) = KEY
LINK(NEWW) = NULL
```

4. Is the list empty??

```
If FIRST = NULL
    FIRST = NEWW
    LINK(NEWW) = FIRST
    TAIL = FIRST
    Return(FIRST
```

```
5.  Is the node preceeds first node??
        If DATA(FIRST) <= DATA(NEWW)
            LINK(NEWW) = FIRST
            LINK(TAIL) = NEWW
            Return(NEWW)

6.  Save the list pointer
        TEMP = FIRST

7.  Traverse till predecessor of new node
        Repeat While LINK(TEMP) <> FIRST and DATA(LINK(TEMP)) <= DATA(NEWW)
            TEMP = LINK(TEMP)

8.  Insert the node
        LINK(NEWW) = LINK(TEMP)

9.  Is node succeds the last node??
        If DATA(LINK(TEMP)) < DATA(NEWW)
            LINK(TAIL) = NEWW
            TAIL = LINK(TAIL)

10. Set the chain and return the list
        TAIL = LINK(TAIL)
        Return FIRST
```