THIS IS A WORK IN PROGRESS!

ARM Assembly Language and Linked Lists

The purpose of this tutorial is to introduce the linked list and to demonstrate how they can be set up and manipulated in ARM assembly language. We are carrying out this exercise because it demonstrate the way in which pointers and register indirect addressing is used in a practical application.

A list is a sequence of elements such as names, or even more complex items. Although we could create simple lists or tables in a computer application, we would soon run into problems when manipulating the information. Consider Figure 1 that contains a list of names. If we wish to sort the names in alphabetical order, we have to perform a lot of data movement. This is not difficult with small lists, but a list with a million items would take an excessively large amount of time.

The linked list solves the problem of list manipulation by separating *sequence* from *content*. Figure 2 demonstrate this concept. Two lists have been created: a list of pointers and a list of data items. Each of the pointers points to its associated data. Now, if you wish to change the order of the items, you can simply change the order of the pointers and leave the items alone. You can remove an item by deleting its pointer, or add an item by creating a new pointer.

Terminology

We refer to the elements of a list as nodes. A node is a record containing; for example, information about the data structure itself and user data.

In this introduction, the two fields of a node are a pointer that points to the next node, and a data field.

Figure 3 demonstrates the linked list where the list of pointers and data items are merged. All we have to do is to create a node with two parts: a *head* that contains a pointer to the next node, and a *tail* that contains the actual data item itself. By convention, the final pointer in the last node of the list is set to 0 (the null pointer) to show that the end of the list has been reached.

Figure 1 The simple list

Figure 2 The pointer list

Figure 3 The linked list

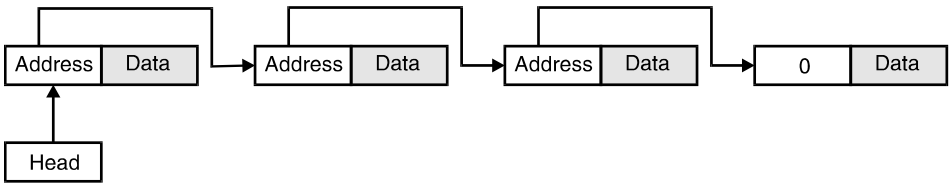


Figure 4 demonstrate how a simple linked list looks in memory. We're used a small memory and decimal arithmetic for simplicity.

Figure 4 The organization of a simple linked list in memory

Traversing the Linked List

Suppose we wish to add a new element to the end of a linked. We have to read the address field of the first element to locate the next element. Then we read the address field of this element in order to move to the next element. The list is traversed in this way until its end has been reached. We know that the end of the list has been located when the address of the next element is zero.

The following fragment of ARM code inserts a new item into a linked list whose nodes consist of a 32-bit pointer and a 32-bit data element at the next word location in memory; that is, if a pointer is at location *m*, its data is at location *m + 4*. Initially, the variable HEAD points to the first item in the list, and the variable NEW contains the address of the new item to be inserted.

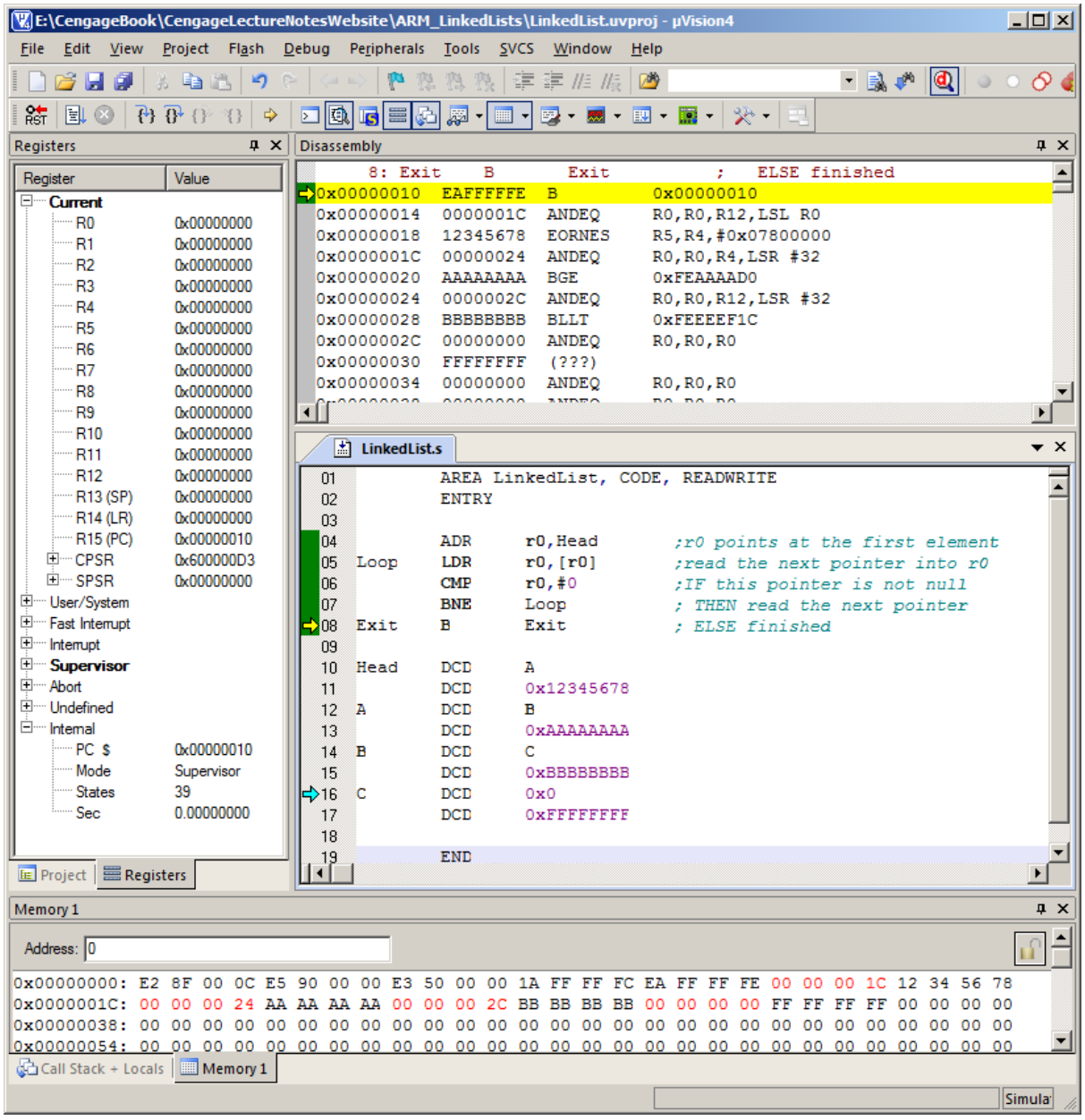
```
        ADR     r0,Head     ; r0 points at the first element
Loop    LDR     r0,[r0]     ; Repeat: read the next pointer into r0
        CMP     r0,#0       ; IF this pointer is not null
```

```
        BNE     Loop        ;       THEN read the next pointer
EXIT    B       Exit        ;       ELSE loop here
```

The key instruction is LDR `r0,[r0]` which reads the contents of memory pointed at by r0 and puts the result in r0. Since the value pointed at is the pointer to the next node, executing this instruction steps through the linked list. The CMP r0,#0 is used to exit the loop when the null pointer has been located.

The next step is to test this code fragment. In Figure 5 we provide a suitable test environment. The linked list begins at location Head. We have called the pointers Head, A, B, C. The data elements are 0x12345678, 0xAAAAAAAA, 0xBBBBBBBB, 0xCCCCCCCC, 0xFFFFFFFF, which you can see in the memory map.

Figure 5 Running the basic code in the Keil simulator



Let's extend the code to perform a useful function by inserting a new element at the current end of the linked list. In principle, we can put the new element anywhere in memory. In practice, we will assume that the memory area beyond the last element if currently free and that we can locate a new element there. All we now need is a register to hold the new data, and we will use r8 for that purpose.

We scan the link list as before. When we locate the final null pointer, we replace it with a pointer to the next element. A new element is created and we insert the data from r8. Finally, we set the pointer field of the new element to zero (Figure 6).

```
        LDR     r8,=0x12121212  ; Set up some dummy data for testing
        ADR     r0,Head         ; r0 points at the first element
Loop    LDR     r1,[r0]         ; Read the next pointer into r0
        CMP     r1,#0           ; IF this pointer is null
        BEQ     Insert          ;    THEN we're at the end (so insert new record)
        MOV     r0,r1           ;    ELSE update r0
        B       Loop            ;    and go round again
Insert                          ; Begin insertion
        ADD     r0,r0,#8        ; Update pointer to point at next free node
        STR     r0,[r0,#-8]     ; Store the pointer in the previous pointer field
        MOV     r1,#0           ; Set up the new null pointer
        STR     r1,[r0]         ; Save it in the new pointer field
        STR     r8,[r0,#4]      ; Pop in the new data field
Stop    B       Stop            ; Park here
```

Figure 6 Inserting a new element at the end of a linked list

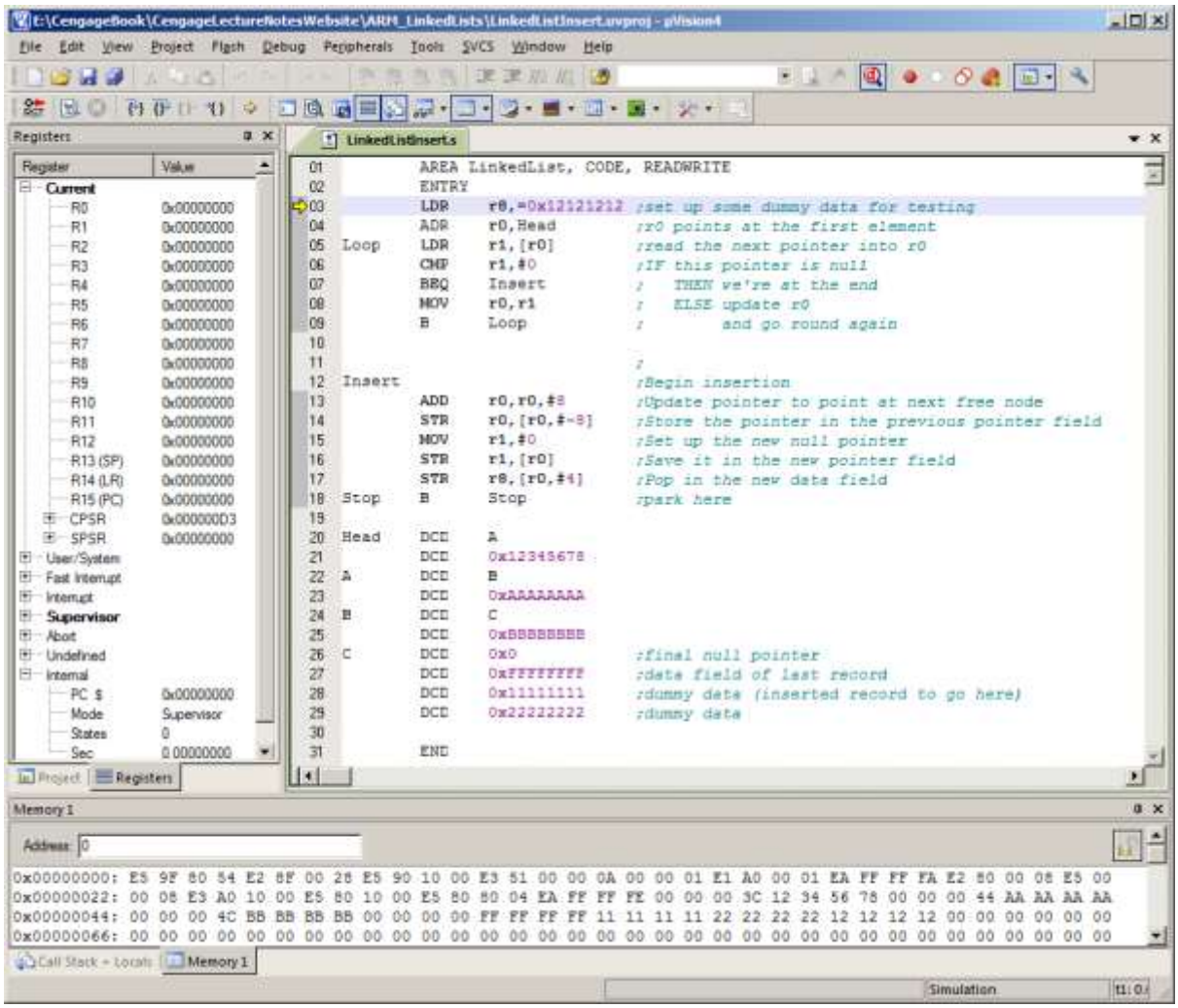


Figure 7 The memory map of the program before the insertion of a record

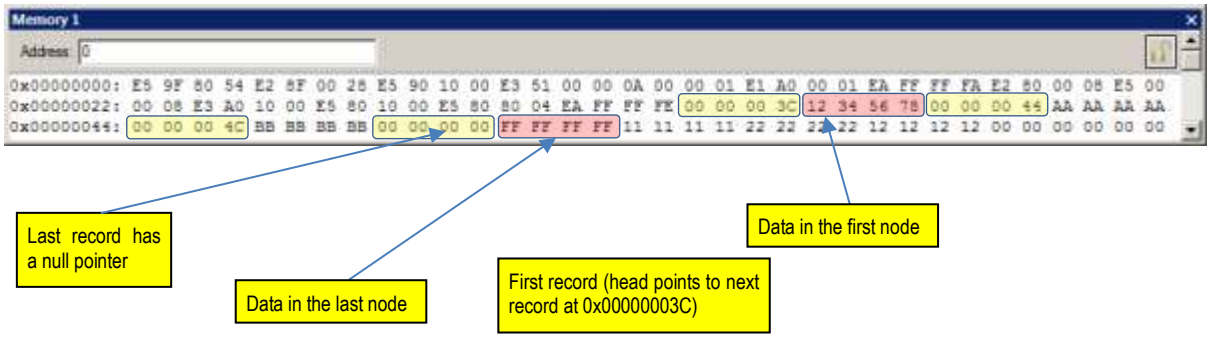
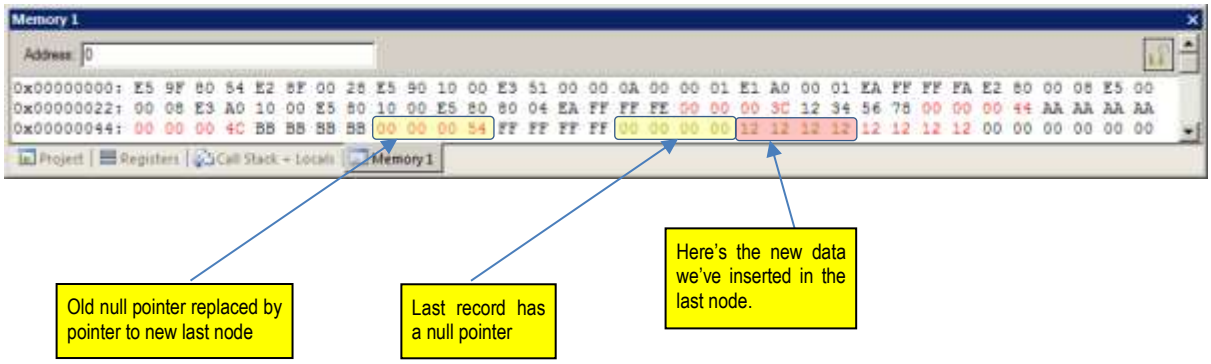


Figure 8 The memory map of the program after the insertion of a record



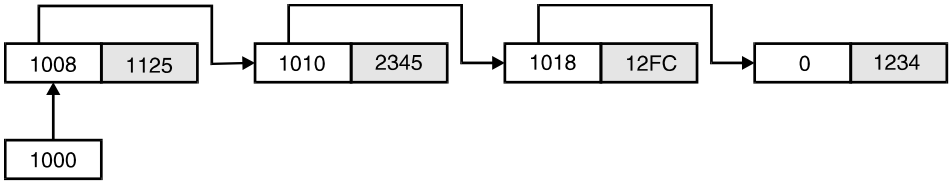
Searching the Linked List for a Maximum Value

The next example demonstrates how we can search a linked list to find the maximum, value of a particular record (data field).

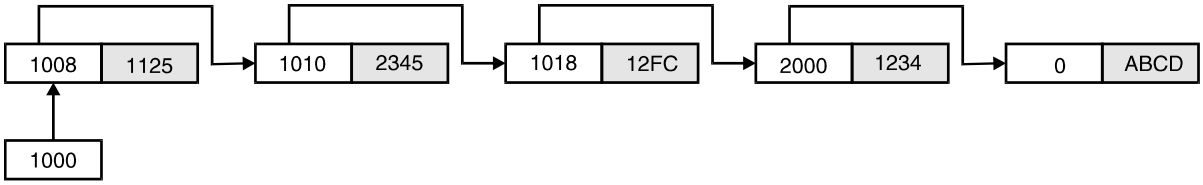
```
EXIT MOVEA.L  #NEW,A1   Pick up address of new element
     MOVE.L   A1,(A0)   Add new entry to end
     CLR.L    (A1)      Add new terminator
```

The following three diagrams illustrate a linked list before the insertion of an element, after the insertion of a new element, and the memory map before and after the insertion.

Example of a linked list



The effect of inserting an element into the linked list



Memory map of the linked list before and after the insertion of an element

Memory map of linked list before inserting an element

00001000	00001008
00001004	00001125
00001008	00001010
0000100C	00002345
00001010	00001018
00001014	000012FC
00001018	00000000
0000101C	00001212

Memory map of linked list after inserting an element

00001000	00001008
00001004	00001125
00001008	00001010
0000100C	00002345
00001010	00001018
00001014	000012FC
00001018	00002000
0000101C	00001212
00002000	00000000
00002004	0000ABCD

Note: The shaded memory elements represent data values

KKKKKKKKKKKKKKKKKKKKKKKKKKKKK

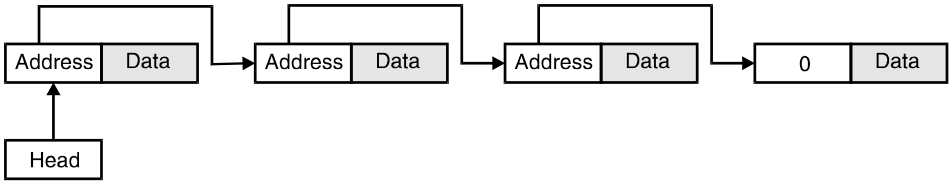
Example of the Use of Address Register Indirect Addressing

This is a mock ICA

An example of address register indirect addressing is provided by the *linked list*. The singly linked list is composed of a chain of units, each linked to its successor by a pointer (i.e., address). The last element in the list usually has a null (i.e., zero) address.

That is, an element in a linked list consists of a header (an address pointing to the next element) and a tail (the data stored by each element). A linked list is useful because you can sort the list simply by changing the pointers, rather than moving the elements themselves.

A linked list



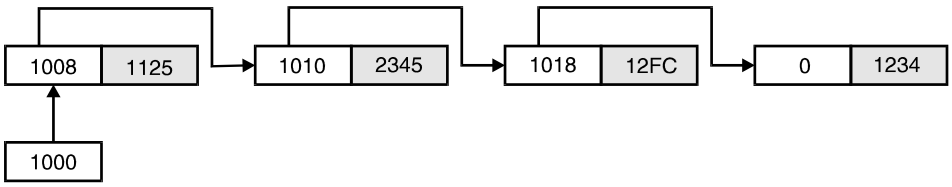
If address register A0 points at the first element in the linked list, the operation `MOVEA.L (A0),A0` reads the contents of the memory location pointed at by address register A0 (i.e., the pointer field of the first item in the list) and puts it in address register A0. The effect of this operation is to leave A0 pointing at the next element in the linked list. Each time this instruction is executed, A0 is advanced to point to the next element.

Suppose we wish to add a new element to the end of a linked list. We have to read the address field of the first element to locate the next element. Then we read the address field of this element in order to move to the next element. The list can be traversed in this way until its end is reached. We know that the end of the list has been located when the address of the next element is zero. The following fragment of code inserts a new item into the list. Initially, the longword variable `HEAD` points to the first item in the list, and the longword variable `NEW` contains the address of the new item to be inserted.

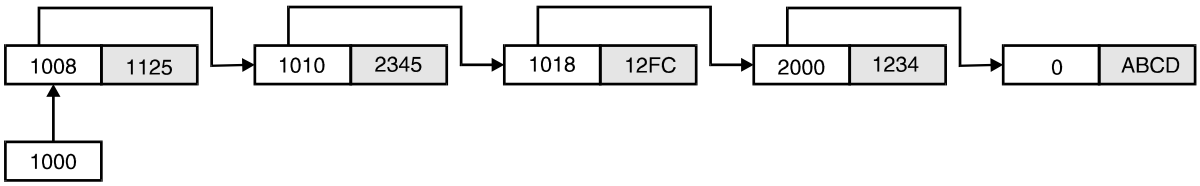
```
     LEA      HEAD,A0   A0 initially points to the start of the linked list
LOOP TST.L    (A0)      IF the address field = 0
     BEQ      EXIT         THEN exit
     MOVEA.L  (A0),A0      ELSE read the address of the next element
     BRA      LOOP      Continue

EXIT LEA      NEW,A1    Pick up address of new element
     MOVE.L   A1,(A0)   Add new entry to end of list
     CLR.L    (A1)      Insert the new terminator
```

The following three diagrams illustrate a linked list before the insertion of an element, after the insertion of a new element, and the memory map before and after the insertion.

Example of a linked list



The effect of inserting an element into the linked list



Memory map of the linked list before and after the insertion of an element

Memory map of linked list
before inserting an element

00001000	00001008
00001004	00001125
00001008	00001010
0000100C	00002345
00001010	00001018
00001014	000012FC
00001018	00000000
0000101C	00001212

Memory map of linked list
after inserting an element

00001000	00001008
00001004	00001125
00001008	00001010
0000100C	00002345
00001010	00001018
00001014	000012FC
00001018	00002000
0000101C	00001212

Note: The shaded memory
elements represent data values

00002000	00000000
00002004	0000ABCD

The Problem

a. Construct a linked list starting at location $1000 in memory. Assume that each element is composed of a longword pointer to the next element and a 4-byte data field. Create your own list and choose suitable numbers for the data elements. Create a list with 8 entries and remember that the pointer of the last element is zero.

b. Draw a diagram (i.e., memory map) for this linked list

c. Write a program that scans the linked list and returns (in data register D0), the value of the largest element in the list.

d. Write a program that puts the elements in the linked list in descending order. That is, the list must be scanned and the pointers modified so that each element points to the next lower element. The data fields of each entry in the list are not "moved" – only the pointers change.

e. Use the 68K simulator to test your program.

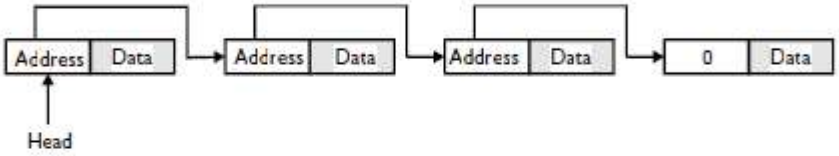
e. Use the 68K simulator to test your program.

KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK

Question 21. By means of diagrams explain the meaning of a *singly-linked list* and a *doubly-linked list*?

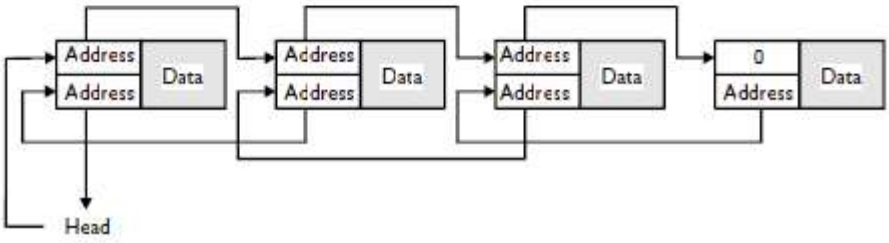
Answer

A singly-linked list is a data structure whose members are composed of two elements: a *head* and a *tail*. The tail is the data part of each member and its contents and structure depend on the nature of the list. The head is a pointer element that points to the next member of the list. The last member of the list may contain a pointer to zero or a pointer back to the first member of the list.



A doubly-linked list is similar to a single-linked list. However, each member of the list has two pointers — one to the *next* member in the list and one to the *previous* member of the list.

You can move through the members of a singly-linked list in one direction only. You can move through the members of a double-linked list in two directions.



KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK

Question 22. A singly-linked list, whose first element is pointed at by A0, consists of elements whose head is a 32-bit address pointing to the next element in the list and a variable-length tail. The tail may be of any length greater than 4 bytes. The last element in the list points to the null address zero. Write a program to search the list for an element whose tail begins with the word in data register D0.

Answer

```
LEA Start,A0 Point at the first element
Next TST.L (A0) Test the next pointer
BEQ Exit IF zero THEN end_of_list
CMP.W (4,A0),D0 Test for the element
BEQ Exit IF found THEN exit
MOVEA.L (A0),A0 Point to next element in the list
BRA Next Continue with search
Exit ...
```

Question 22. A singly-linked list, whose first element is pointed at by A0, consists of elements whose head is a 32-bit address pointing to the next element in the list and a variable-length tail. The tail may be of any length greater than 4 bytes. The last element in the list points to the null address zero. Write a program to search the list for an element whose tail begins with the word in data register D0.

Answer

```
        LEA     Start,A0    Point at the first element
Next    TST.L   (A0)        Test the next pointer
        BEQ     Exit        IF zero THEN end_of_list
        CMP.W   (4,A0),D0   Test for the element
        BEQ     Exit        IF found THEN exit
        MOVEA.L (A0),A0     Point to next element in the list
        BRA     Next        Continue with search
Exit    ...
```


```
LEA Start,A0 Point at the first element
Next TST.L (A0) Test the next pointer
BEQ Exit IF zero THEN end_of_list
CMP.W (4,A0),D0 Test for the element
BEQ Exit IF found THEN exit
```

```
MOVEA.L (A0),A0 Point to next element in the list
BRA Next Continue with search
Exit ...
```

```
        ADR   r1,MyArray      ;set up r1 to point to MyArray
        .
        LDR   r3,[r1]         ;read an element
        .       .
MyArray DCD   0x12345678
```

```
       AREA ARMtest, CODE, READONLY
       ENTRY
       MOV   r0,#0              ;clear total in r0
       MOV   r1,#10             ;FOR i = 1 to 10
Next   MUL   r2,r1,r1           ;  square number
       MLA   r0,r2,r1,r0        ;  cube number and add to total
       ADD   r1,r1,#1           ;  increment number
       CMP   r1,#11             ;  test for 1 past end of loop
       BNE   Next               ;END FOR
       END
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
       AREA LinkedList, CODE, READONLY
       ENTRY
       ADR   r0,List            ; r0 points at the list
Next   LDR   r1,(r0)            ; Repeat: read pointer field
       TST   r1                 ; test the pointer
       BEQ   Exit               ; leave on null pointer
       LDR   r2,(4,r1)          ; read the data element
       CMP   r2,r3              ; test for 1 past end of loop
       BNE   Next               ; continue if element not found

List   DCD   0x0000100012345678
       DCD   0x0000100012345678
       DCD   0x0000100012345678
       DCD   0x0000100012345678
       DCD   0x0000100012345678
       DCD   0x0000100012345678
       DCD   0x0000100088888888
Exit   B     Exit               ;endless loop for testing
       END
```