

## Pointers and Linked-Lists

A **pointer** is a variable which does not contain any data. Instead, it points to a memory area where the data is stored. This makes sense when a very large amount of data must be stored, such as a picture.

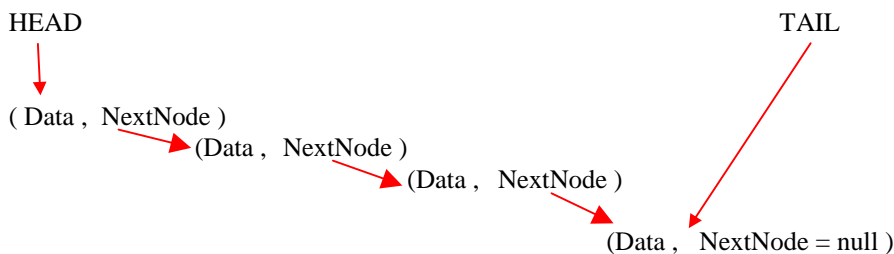
The **pointer variable** contains the **address** in memory where the actual data is stored. In a 32 bit operating system, a pointer variable contains a 32 bit integer address.

In this diagram, Airplane and Car are pointer variables.

```
Airplane -----> ( picture of airplane )
Car  -----> ( description of a car as a document )
```

A **linked-list** contains **nodes**. Each node contains some **data**, as well as a pointer to the **next** node.

**Head** is a pointer variable pointing to the first node in the list. To find the other nodes, a program must follow along the chain, from one node to the next. It is convenient to also keep a **Tail** pointer active which always points to the last node in the list. At the end, last node then points to **null (nothing)**.



To **ADD** a node to the list, the application (e.g. Java) must ask the operating system to **allocate** some memory where the node can be stored. The OS (e.g. Windows) reserves some space, and returns the **address** of that area to the application, which then stores the address in a pointer variable.

```
Node thisNode = new Node();
```

The NEW command asks for memory. The amount of memory reserved is determined by the definition of Node.

Once the new node has been created, it must be attached to the list. The next pointer in the last node must be changed to point at the new node.

```
tail.next = thisNode;
```

Usually, the application wants to store some data in the new node.

```
thisNode.data = word ;
```

Now, to preserve the **overall structure** and **integrity** of the linked-list, we must move the LAST pointer to point at the new TAIL node, and then set this last node to point at nothing.

```
tail = thisNode ;
tail.next = null;
```

A linked-list is **not** the same as an array. You **cannot** use [subscripts]. You **cannot** print the third node by writing:

```
output( head(3).data );      \ NO! This is NOT CORRECT!
```

This is meaningless. Instead, you can only **access** subsequent nodes by following pointers **sequentially**. To print the first three nodes in the list use these commands:

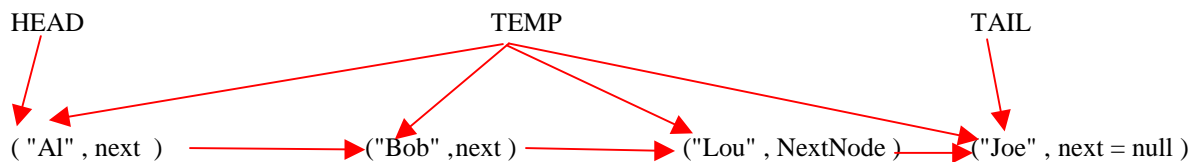
```
output( head.data );        \ the Data stored in the first node
output( head.next.data);    \ the Data in the second node
output( head.next.next.data); \ the Data in the third node
```

Accessing (printing or searching) the **entire list** requires a loop. The key idea here is a **temporary pointer** which moves along from one node to the next. A command to move a **temp** pointer from one node to the next is:

```
temp = temp.next ;
```

Here is the code to print the entire list:

```
Node temp ;
temp = head ;          \ make TEMP point at the first node
while ( temp != null ) \ checks for the null pointer
{                       \ at the end of the list
    output( temp.data );
    temp = temp.next    \ moves to the next node
}
```



Some practice questions:

(Q1) What does the following command accomplish?

```
head = head.next ;
```

(Q2) What does the following command accomplish?

```
head.next = head.next.next ;
```

(Q3) What does the following accomplish?

```
head.next = null ;
```

```
last = head ;
```

(Q4) The following is **always** an illegal command. Why?

```
output( last.next.data );
```

One might wonder, “Why would anyone go to all this trouble? Why not just use an array instead of all this confusion and aggravation with pointers?” First, arrays are generally limited to some maximum size - many programming languages only allow a maximum of 64 KB to be stored in an array. Second, the **maximum** size of an array is **declared** (e.g. Dim) at the beginning of the program, before running, and this maximum size **cannot be changed**. Arrays are **static** - their size doesn't change when the program is running - indeed, it doesn't ever change. Lastly, some operations like **inserting** in the middle of a list are very efficient using Linked-Lists – it is not necessary to "move" anything when inserting.

**Pointers** provide **dynamic memory allocation**. More memory can be given to a list when it needs it. Large applications might maintain several large lists of data (thousands or millions of items). Using dynamic allocation allows the application to **balance** memory usage so that all the **data-structures** can get enough memory when they need it.

*Besides all that, pointers are fun (well, most programmers think so), and they are **required** for the IB Higher Level examination.*

=====

The sample program LINKLIST allows the user to type in one word at a time, and slowly builds up a linked-list. You need to practice programming this type of **data-structure**. Complete the following exercises.

(1) Create a procedure which removes the first node in the list (the head). It simply needs to change the HEAD pointer to point at the second node. This can be achieved with a single command. However, you should properly **error-trap** this procedure so that it will not crash if the list is empty.

(2) Create a procedure which removes the second node in the list. It accomplishes this by changing HEAD.NEXTNODE to point at the third node in the list. Again, this can be accomplished by a single command, but use an if..then.. to prevent run-time errors on short lists.

(3) Create a procedure which **inserts** a new node between the first and second nodes. Thus, if the list contains Adam, Baker, and Charlie, then a new node Eddy would be inserted and the list would contain Adam, Eddy, Baker, Charlie. This is a little more difficult. A new node must be created. Then the first node must be forced to point at the new node, and the new node must point at the previously second node.

(4) Create a second list, with a pointer called HEAD2 pointing at the first node, and TAIL2 pointing at the last node. Now you should be able to type names into either list. Make a new procedure which **concatenates (joins)** the two lists. This is pretty easy - make the last node of the first list point to the first node of the second list. Then set TAIL equal to TAIL2 so that the HEAD...TAIL list contains all the nodes. Now set HEAD2 and TAIL2 to Nothing, so the second list is empty and the first list contains everything.

(5) Create a **search** procedure. It should ask for a name, then search through the list for a matching name. If the name is found, it returns “FOUND”. Otherwise, it returns "NOT FOUND".

(6) Change the ADD procedure so that it also works when the list is empty. Then the ADDFIRST procedure is no longer needed. This requires some if..then.. command to check whether the list is empty.

(7) Create a separate **LinkedList** class (without a UI). This must contain a HEAD pointer (object), a TAIL pointer, and methods to INITIALIZE the list (set HEAD and TAIL to NOTHING), to ADD a new node, and to DISPLAY the list. Further methods should be added as stated below. Your program will still need the **ListNode** module, but this still only contains the Data and NextNode data members.

(8) Add a function to **COUNT** the number of nodes in the list.

(9) Write a **DeleteHead** method which removes the first node in the list. This command is *almost* sufficient:

```
head = head.next ;
```

But you must add error trapping, in case the list is empty.

(10) By creating a second **LinkedList**, and using **DeleteHead** repeatedly, write a procedure to **REVERSE** a linked list. The result is stored in a new linked-list.