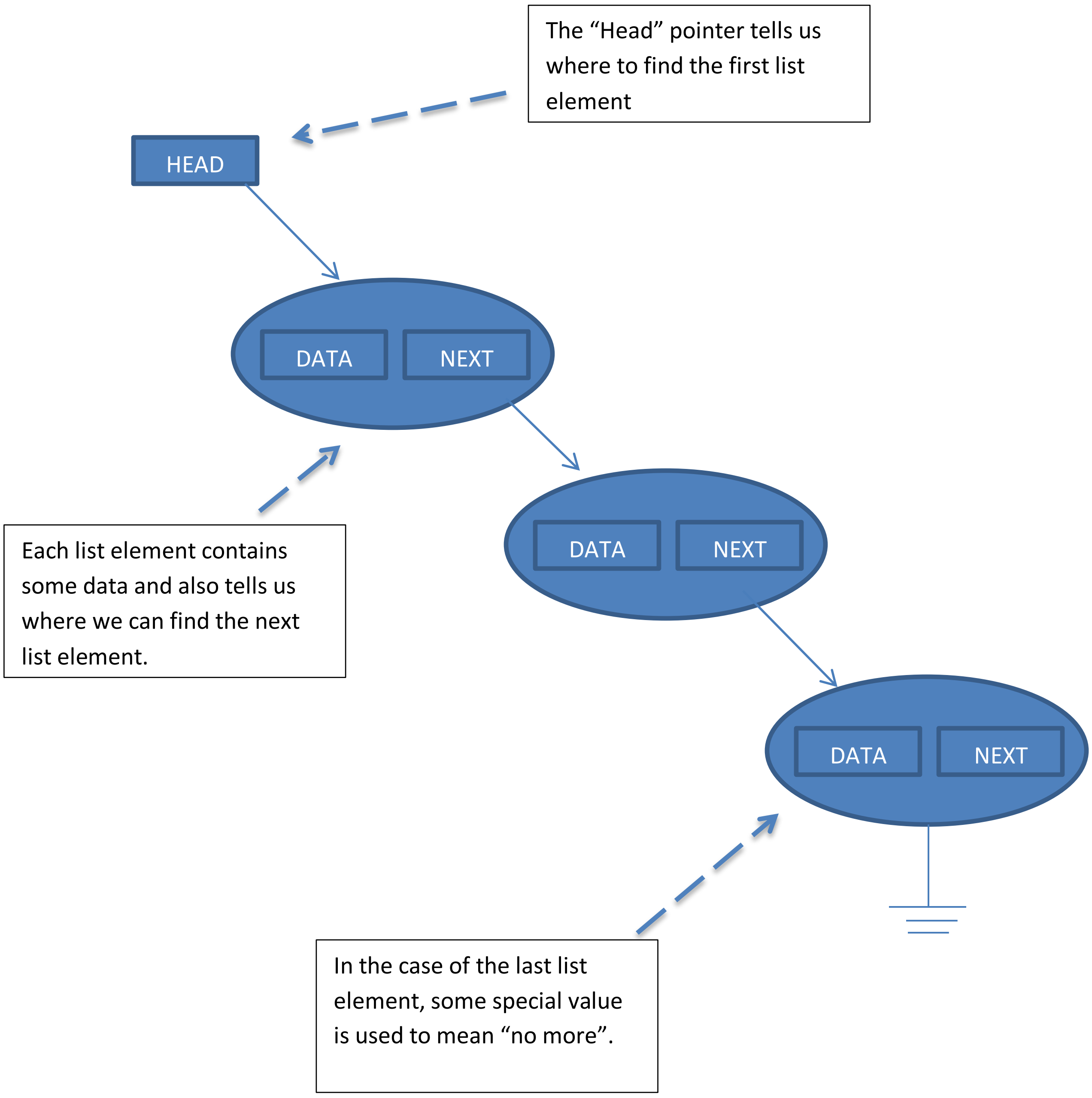
# LINKED LISTS



The “Head” pointer tells us where to find the first list element

Each list element contains some data and also tells us where we can find the next list element.

In the case of the last list element, some special value is used to mean “no more”.

In the C world, the elements of a linked list are structures, the head and next pointers are pointer variables, and “NULL” is used as the special “no more elements” value.

We will begin by considering a list of "int" values in this case list elements (hereafter called "nodes") are as follows:

```
Struct LNode {
int data;
LNode *next;
};
```

And the head pointer is a pointer (initially NULL, meaning that the list is completely empty) to a node.

```
LNode *head= NULL;
```

The following code reads in values (until -1 is entered) and insert each value entered at the beginning of the list. It assumes the node declaration given above Keep in mind that "n -> data" is equivalent to "(*n).data". The value is stored into the "data" component of the structure pointed to by "n"

```
Int value;
LNode *n;                    // "n" for "new"
for (;;) {
printf("Enter a value(-1 to end) \n");
scanf("%d \n",value);
  if (value== -1) {
      break;
  }
n = new LNode ():           // create a new node
n -> data= value;           // place the data in the node
                              // insert new node into the list (at the beginning)
n -> next= head;          // make the node refer to the first existing list element

head = n;                 // make the head pointer refer to the newly created node
}
```

This particular code always adds new values at the beginning of the list.

Having built a list, we might want to print out all of the values in it. The following code does this.

```
LNode *c;                    // c for "current"
c = head;                    // while we've more nodes to visit
  while (c != NULL) {
  printf("%d \n",data);    // output the value on the current node
                             // and advance to the next one
  c=c->next;                // advance to next node
  }
```

Note that this code will function correctly (output nothing) if the list is empty (if the head pointer is null). The process of proceeding through a list node by node (as this code does) is called a "list traversal". The process can be very neatly implemented with a "for" loop, as shown below.

```
for (c= head; c != NULL; c = c -> next) {
printf("%d \n", c->value);        // output the value on the current node
}
```

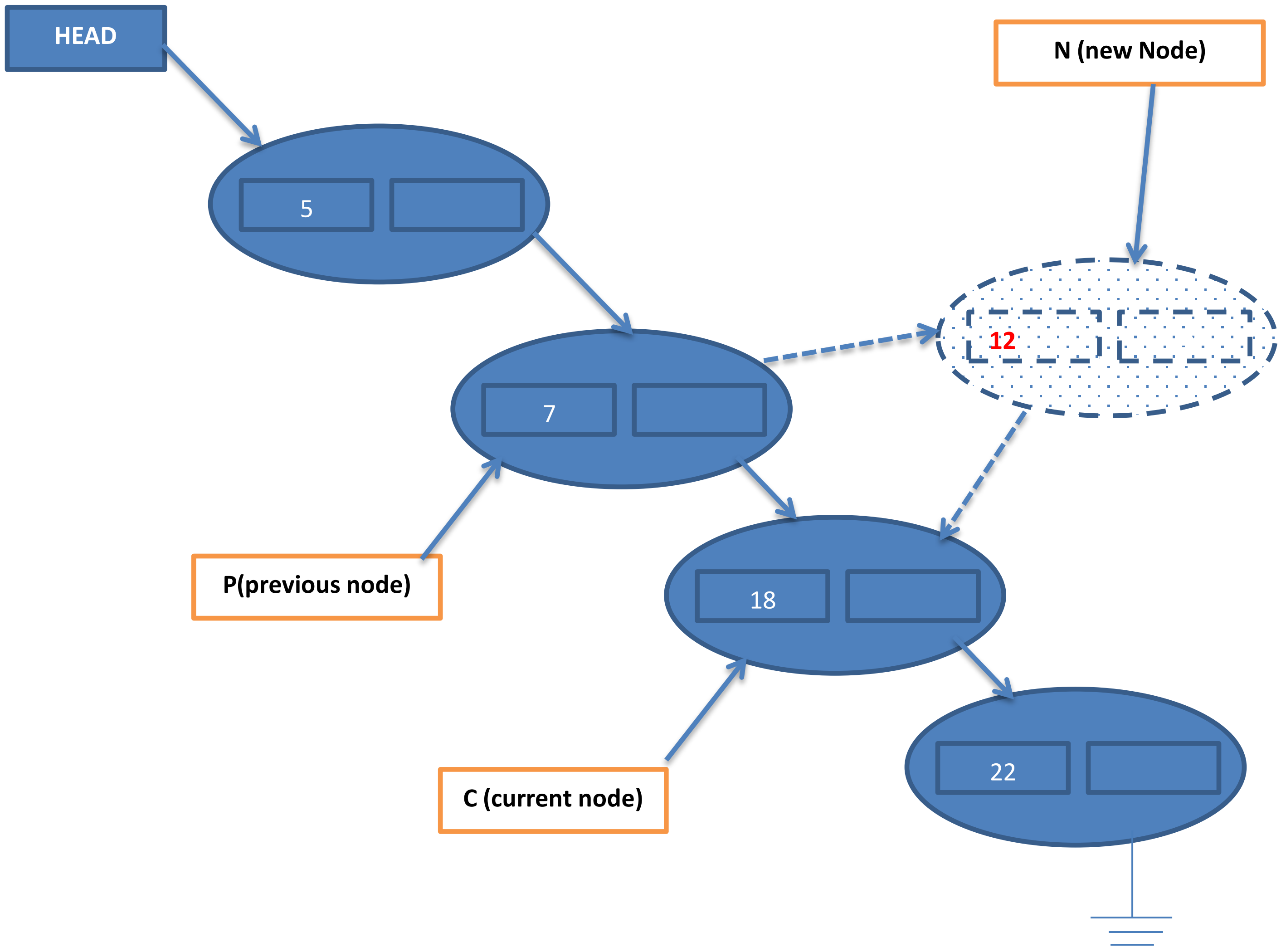
Imagine that we would like to delete the first element of our list. The following code deletes the first element of the list and leaves the value that was in it in variable “Value”.

```
LNode *c;
int value;                   // just to be safe, we begin by ensuring that there is in fact a first element
if (head != NULL){       //(i.e. that the list is not empty)
value = head -> data; // extract value from element
c = head;                    // keep track of the first node
head= head -> next;   // de-link the node from the list
delete c;                    //delete the node
}
```

Note that this code does nothing if the list is empty.

If new values are always inserted at the beginning of a list, and all deletions remove the first list element, the list acts as a "stack” (a ’first-in, last-out’ or FILO, container).

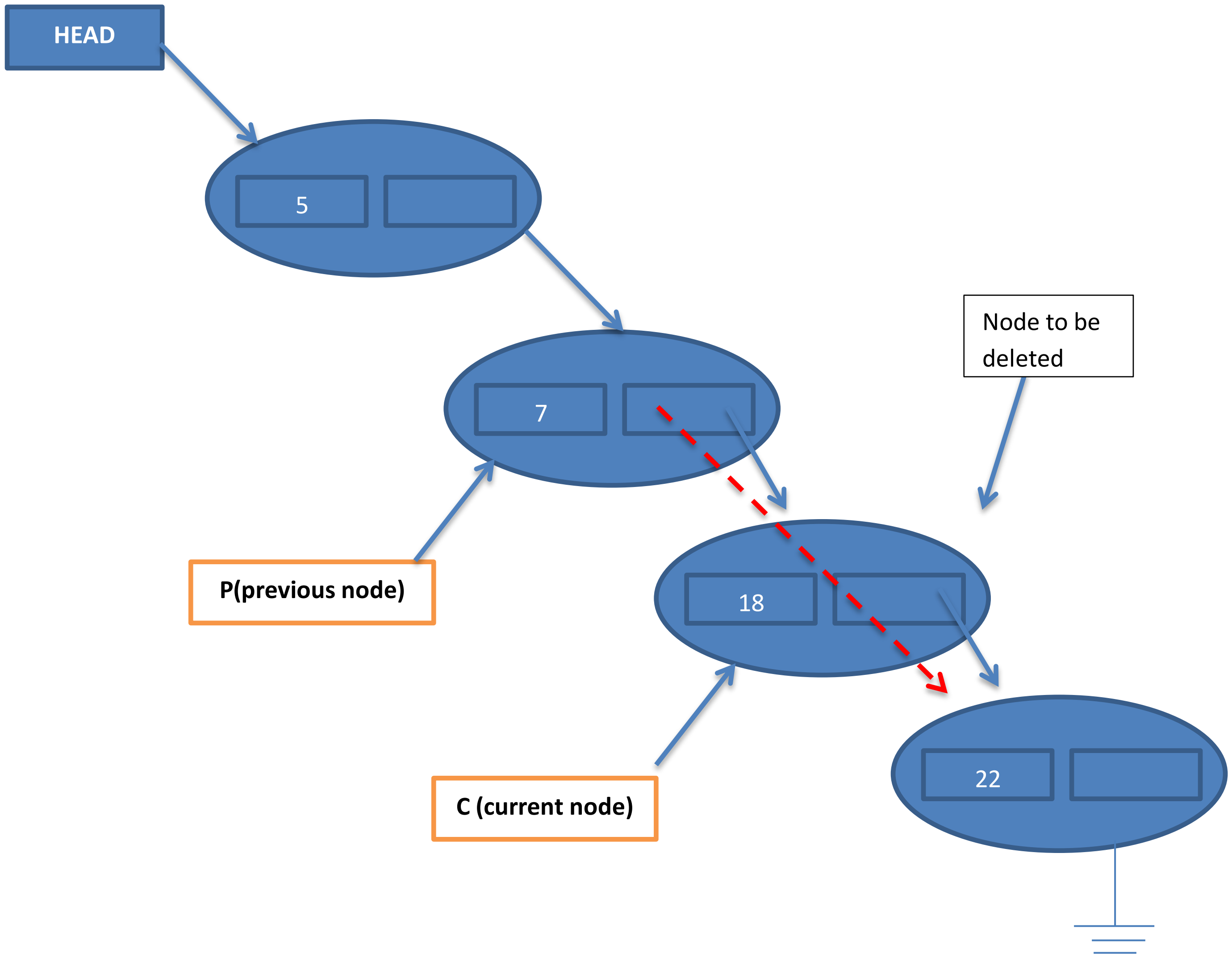
If we want to keep the contents of a list in order (i.e. maintain a sorted list), we cannot always insert new values at the beginning of the list, but must instead insert them where they belong' The diagram on the next page illustrates the insertion of 12 into a sorted linked list which already contains 5,7, 18, and 22.



Inserting in the middle of a list is a two stage process it is first necessary to locate the position. Within the list, at which the new node is to be inserted. This process involves two pointers, the "previous" (p) and "current" (c) pointers. They are positioned so that "p" points to the node before the point of insertion, and "c" points to the node after the point of insertion. The actual insertion may then be accomplished by the following two assignments.

```
n -> next= c;
p -> next= n;
```

Whether or not a list is sorted, we may want to delete an element other than the first element. The diagram below illustrates the deletion of 18 from a List containing 5, 7, 18, and 22.



If the node to be deleted is somewhere in the middle of the list, the node can then be de-linked from the list with the following single statement.

```
p->next=c->next;        // de-link node from list.

LNode *P: NULL, *c= head;
                        // keep looking until we either hit the end of
                        // the list or find the node to be deleted.
While ((c!= NULL) && (value != c -> data) ) {
        p=c; c=c->next;
}
```

If c runs off the end of the list, it means that the node we're looking for doesn't exist.

```
if (c ==NULL) {
  return;            // or whatever.
}
```

Note that, if the list is sorted, we need not go right to the end of the list before concluding that the value we're looking for doesn't exist. Instead we can give up as soon as we come across a value which is bigger than the value we're looking for.

Once the point of deletion has been found, there are only two cases to consider. Either "p" is NULL (the node is the first list node) or it isn't (the node isn't the first list node). In either case the general idea is the same. We must update whatever is before the node to be deleted so that it points to the node after the node being deleted. If there is a previous node, it is updated, and, if there isn't, the head pointer is updated instead.

```
if (p== NULL) {                          // node is the first list node
   head= c -> next;
  } else {                               // node isn't the first list node
            p -> next= c -> next;
 }
delete c;
```