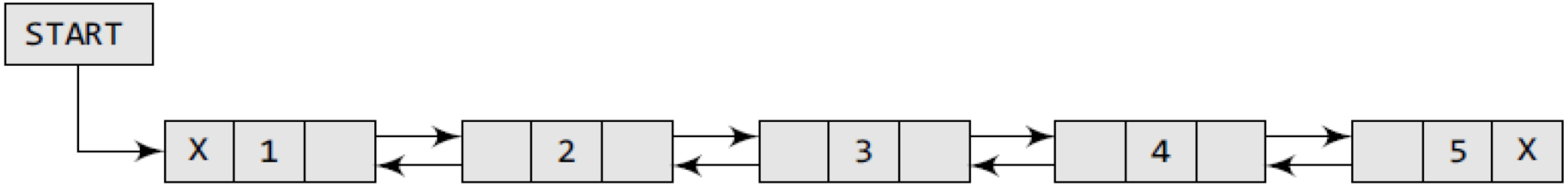
Double Linked List

Prof Muliaro Wafula

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.

Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.



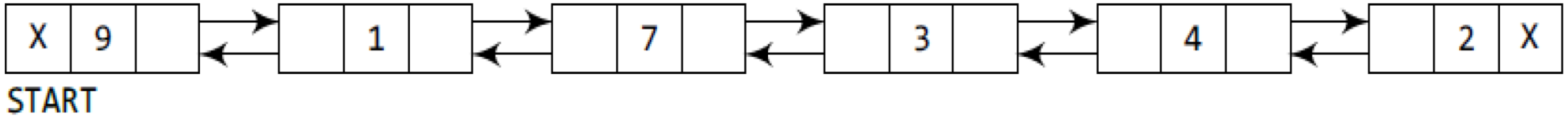
A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).

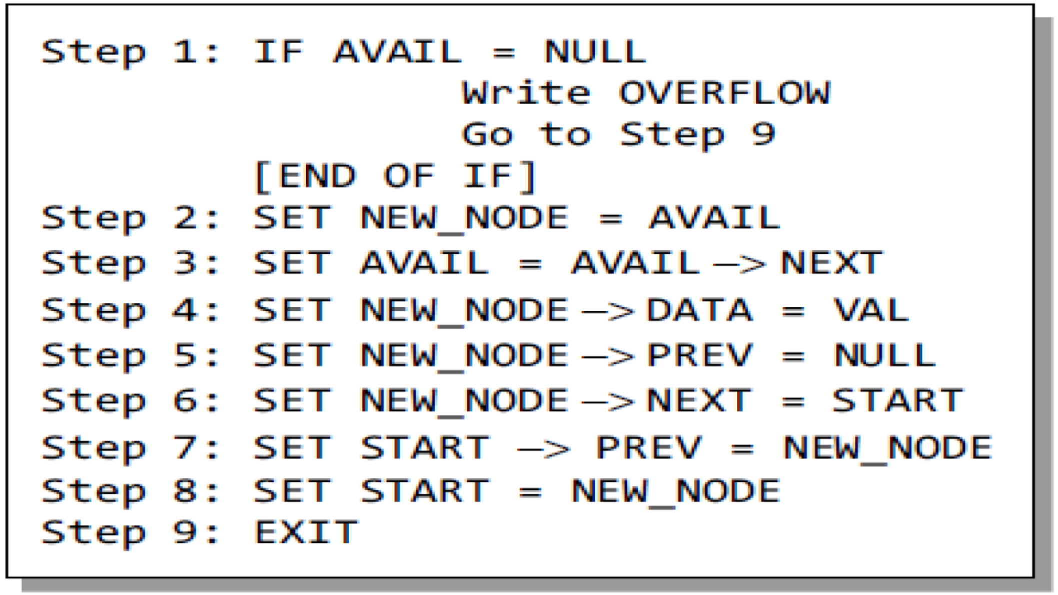
The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Let us view how a doubly linked list is maintained in the memory.

Inserting a Node at the start of List

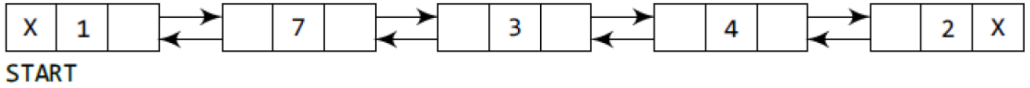
Add the new node before the START node. Now the new node becomes the first node of the list.



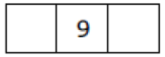


```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

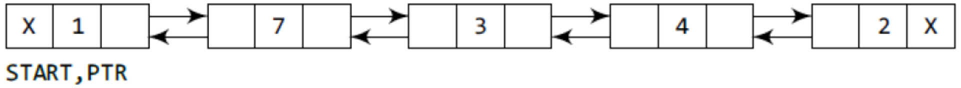
Inserting a Node in between nodes



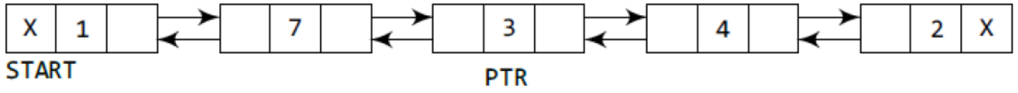
Allocate memory for the new node and initialize its DATA part to 9.



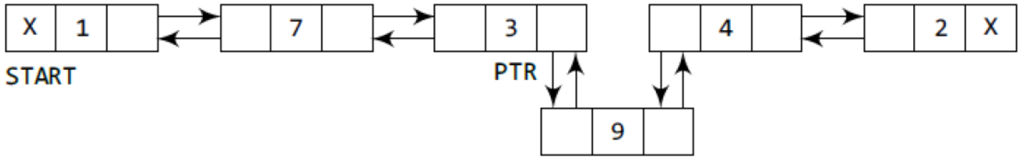
Take a pointer variable PTR and make it point to the first node of the list.

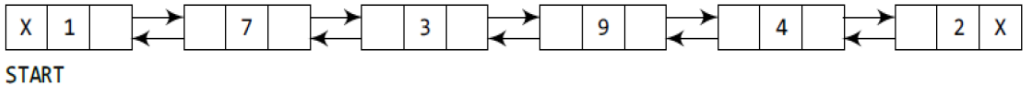


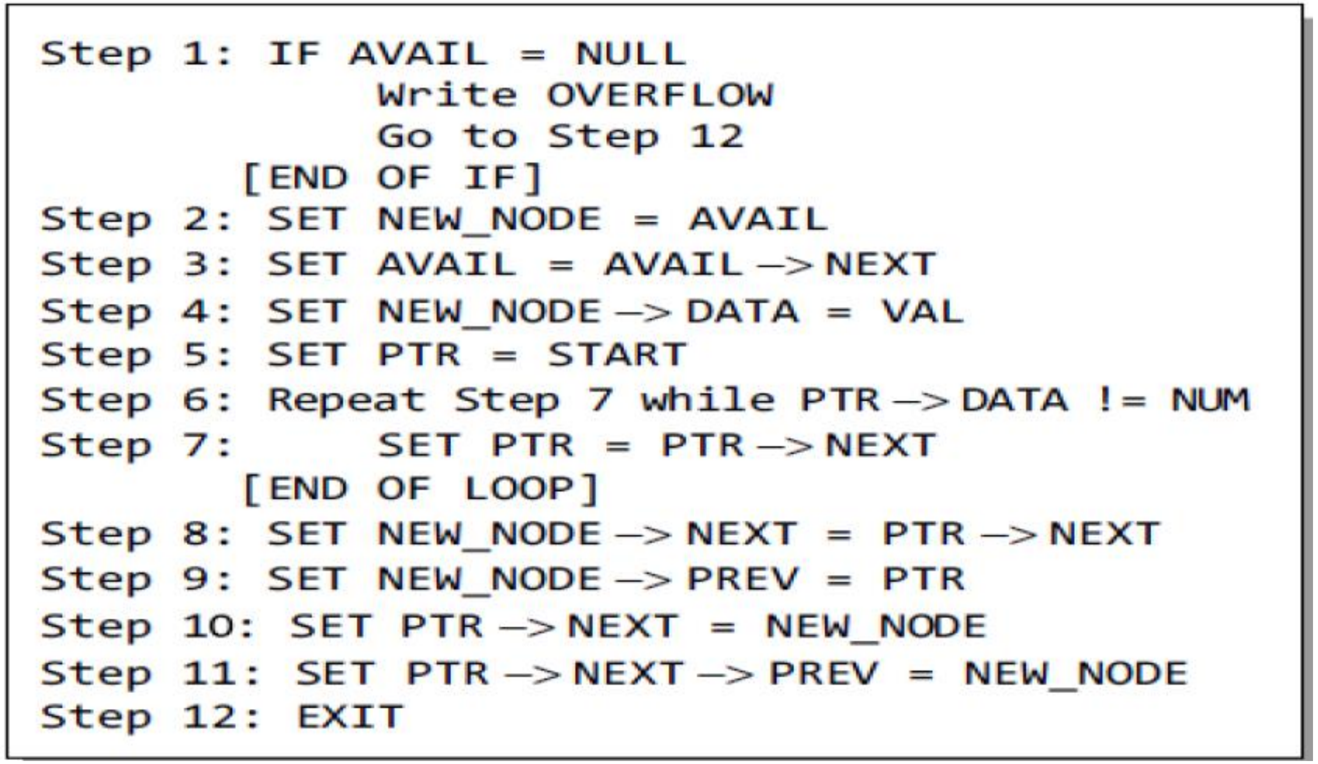
Move PTR further until the data part of PTR = value after which the node has to be inserted.



Insert the new node between PTR and the node succeeding it.







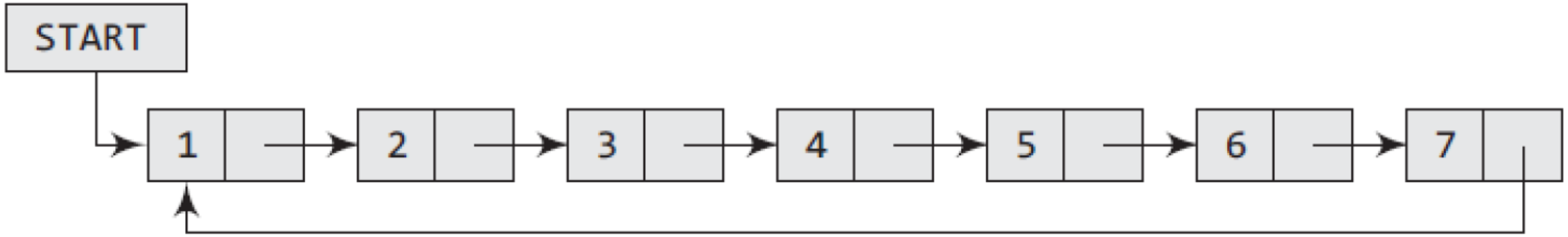
```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 12
          [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

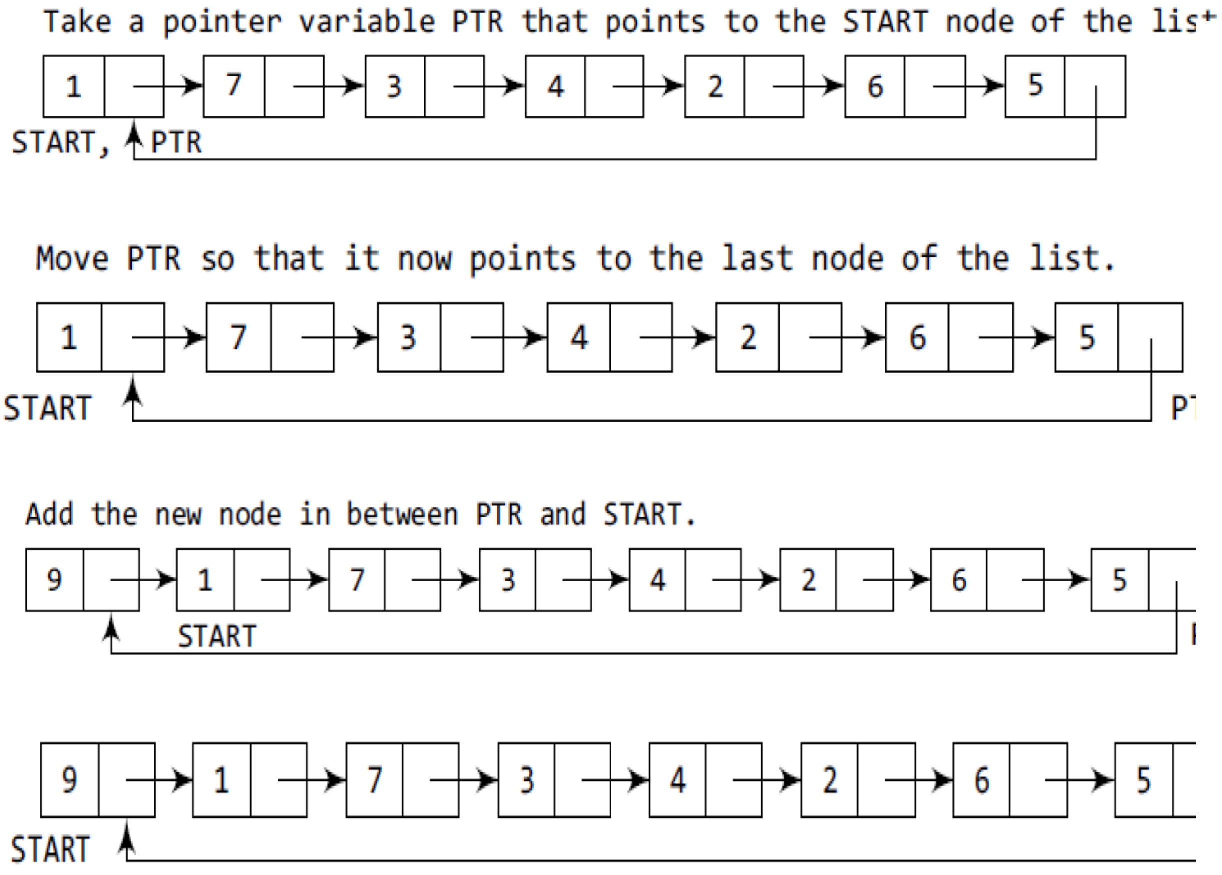
Circular Linked Lists

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

Note that there are no NULL values in the NEXT part of any of the nodes of list.

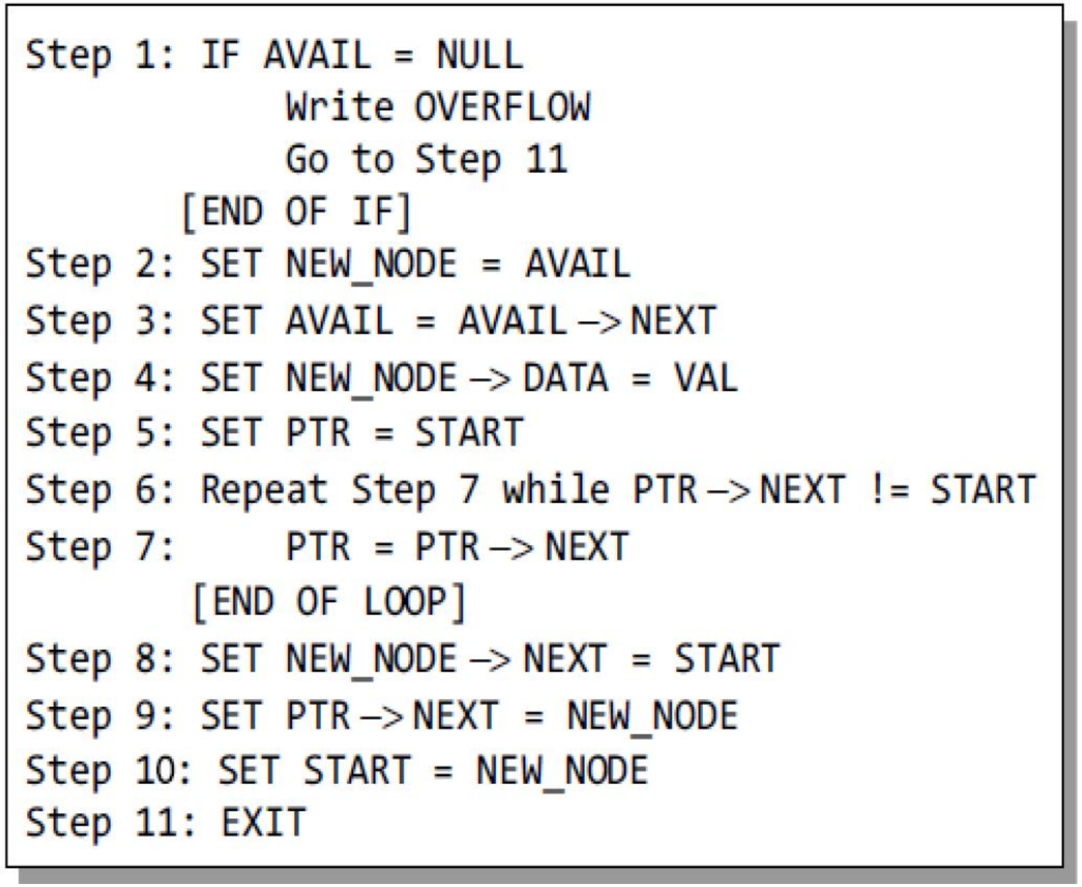




Take a pointer variable PTR that points to the START node of the list

Move PTR so that it now points to the last node of the list.

Add the new node in between PTR and START.



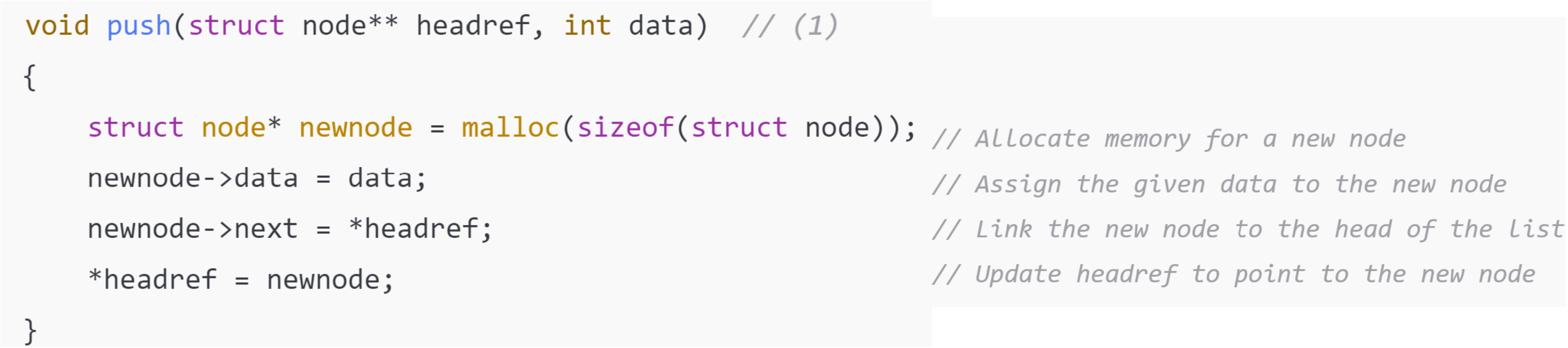
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
          [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:       PTR = PTR -> NEXT
          [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Summary-Linked List/Node

```
struct node {
    int info;           // This is an integer field to store the information (data)
    struct node *next;  // This is a pointer to the next "node" structure.
} *ptr;
```

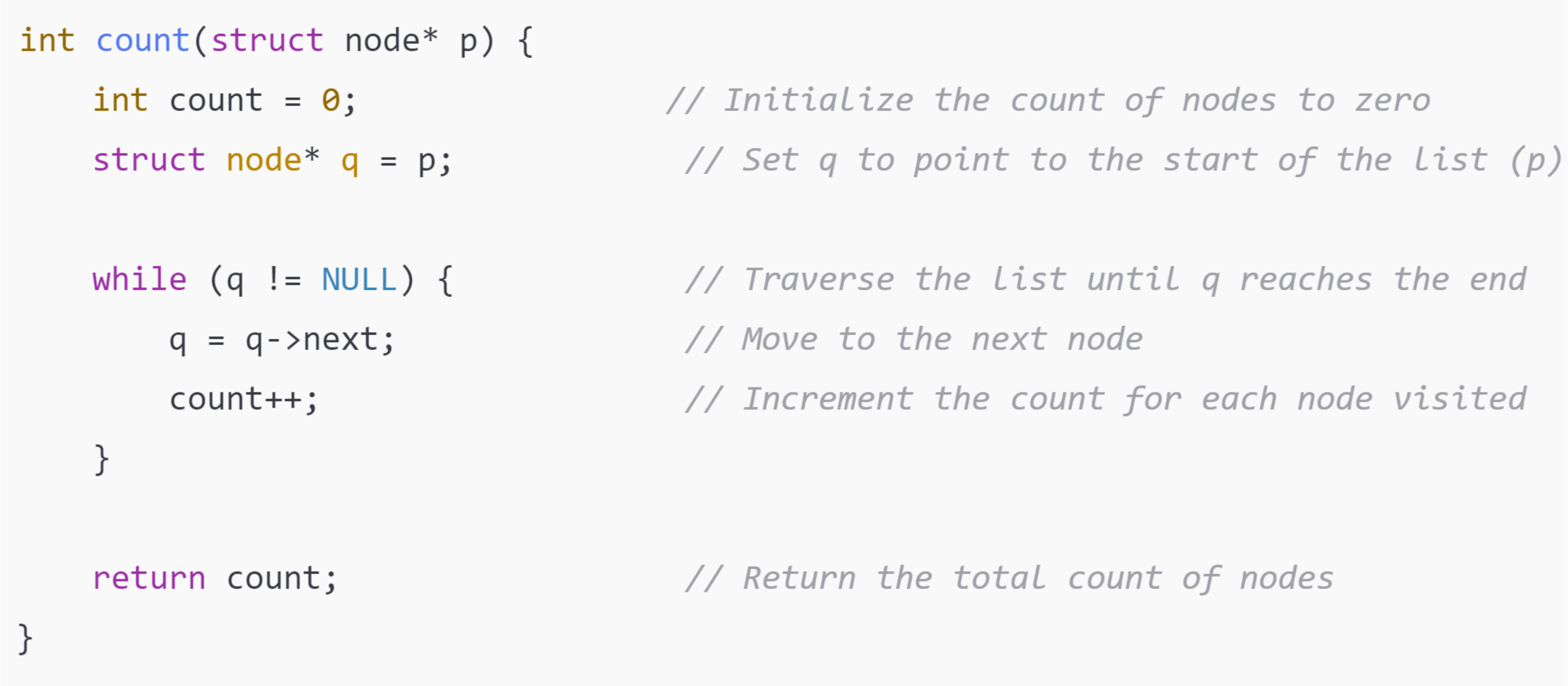
- `struct node`: This declares a new structure type called `node`.
- `int info;`: An integer field within the structure to hold some data.
- `struct node *next;`: A pointer named `next` that points to the *next* `node` structure in a linked list. This is what makes the structure "self-referential."

Insertion



```
void push(struct node** headref, int data)  // (1)
{
    struct node* newnode = malloc(sizeof(struct node)); // Allocate memory for a new node
    newnode->data = data;                               // Assign the given data to the new node
    newnode->next = *headref;                           // Link the new node to the head of the list
    *headref = newnode;                                 // Update headref to point to the new node
}
```

Incremental

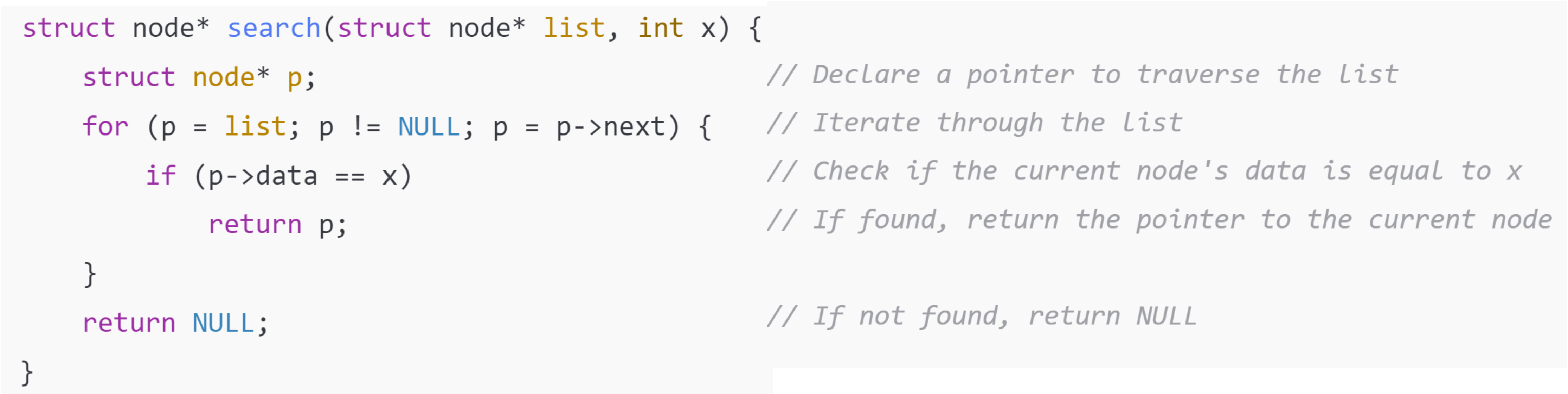


```c
int count(struct node* p) {
    int count = 0;                // Initialize the count of nodes to zero
    struct node* q = p;            // Set q to point to the start of the list (p)

    while (q != NULL) {            // Traverse the list until q reaches the end
        q = q->next;               // Move to the next node
        count++;                   // Increment the count for each node visited
    }

    return count;                  // Return the total count of nodes
}
```

Searching



```
struct node* search(struct node* list, int x) {
    struct node* p;                             // Declare a pointer to traverse the list
    for (p = list; p != NULL; p = p->next) {    // Iterate through the list
        if (p->data == x)                       // Check if the current node's data is equal to x
            return p;                           // If found, return the pointer to the current node
    }
    return NULL;                                // If not found, return NULL
}
```