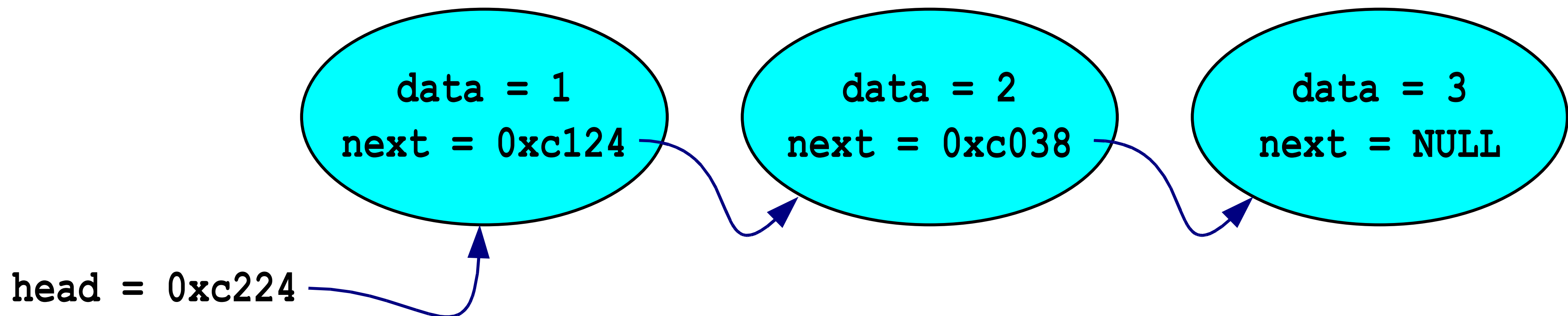
Linked List

- So far, you know two types of data structures, which are collections of data
  - arrays
  - stacks
- Linked lists are another collection type.
- Arrays, stacks and linked lists store "elements" on behalf of "client" code.
- The specific type of element is not important since essentially the same structure works to store elements of any type.
  - Arrays / stacks of integers
  - Arrays / stacks of doubles
  - Arrays / stacks of customer accounts
  - Arrays / stacks of ...

LInked List Structure

- An array allocates memory for all its elements lumped together as one block of memory.
- A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". All nodes of a list are connected together like the links in a chain.
- Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.
- Each node is allocated in the heap with a call to **`malloc()`**, so the node memory continues to exist until it is explicitly deallocated with a call to **`free()`**. The front of the list is a pointer to the first node.



Create List

- The example below will create the three-element list

```
struct node
{
  int data;
  struct node *next;
};
```

data = 1
next = 0xc124

data = 2
next = 0xc038

data = 3
next = NULL

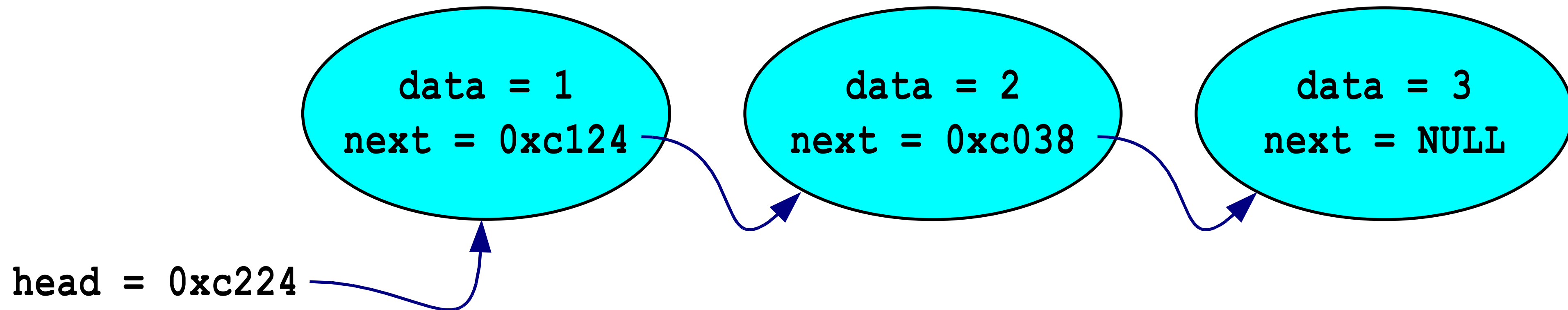
head = 0xc224

```
/* Build the list {1, 2, 3} in the heap and store
   its head pointer in a local stack variable.
   Returns the head pointer to the caller. */

struct node * BuildOneTwoThree ()
{
  struct node *head = NULL;
  struct node *second = NULL;
  struct node *third = NULL;
  head = malloc (sizeof (struct node)); /* allocate 3 nodes in the heap */
  second = malloc (sizeof (struct node));
  third = malloc (sizeof (struct node));
  head->data = 1;                /* setup first node */
  head->next = second;
  second->data = 2;              /* setup second node */
  second->next = third;
  third->data = 3;               /* setup third link */
  third->next = NULL;
/* At this point, the linked list referenced by "head"
   matches the list in the drawing. */
  return head;
}
```

Count Elements in the List

- Pass the list by passing the head pointer
- Iterate over the list with a local pointer



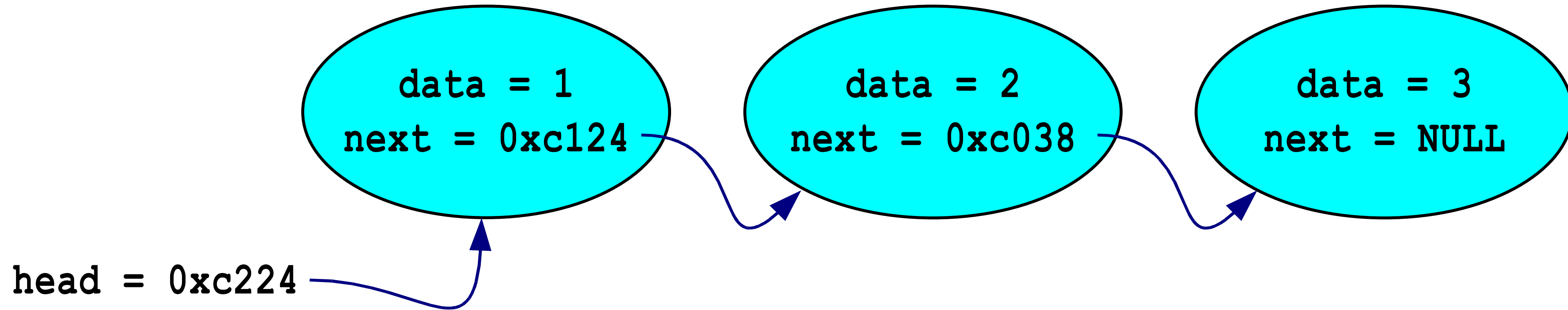

```
/* Given a linked list head pointer, compute
   and return the number of nodes in the list. */

int Length (struct node *head)
{
  struct node *current = head;
  int count = 0;
  while (current != NULL)
    {
      count++;
      current = current->next;
    }
  return count;
}

void LengthTest ()
{
  struct node *myList = BuildOneTwoThree ();
  int len = Length (myList);    /* results in len == 3 */
}
```

Display Elements in the List

- Iterate over the list with the local pointer
- Print the data contained in each node

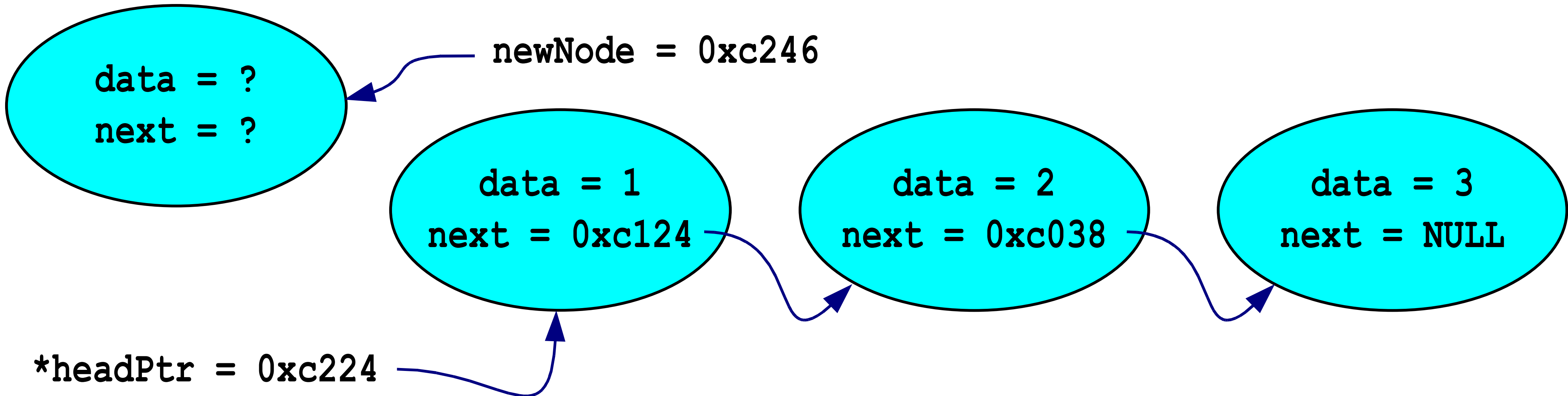



```
/* Given a linked list head pointer, display
   all numbers stored in the list. */

void Display (struct node *head)
{
  struct node *current = head;
  while (current != NULL)
    {
      printf("%d ",current->data);
      current = current->next;
    }
  printf("\n");
}
```

Add Element in Front of the List

- Pass the pointer to the head to be able to modify it

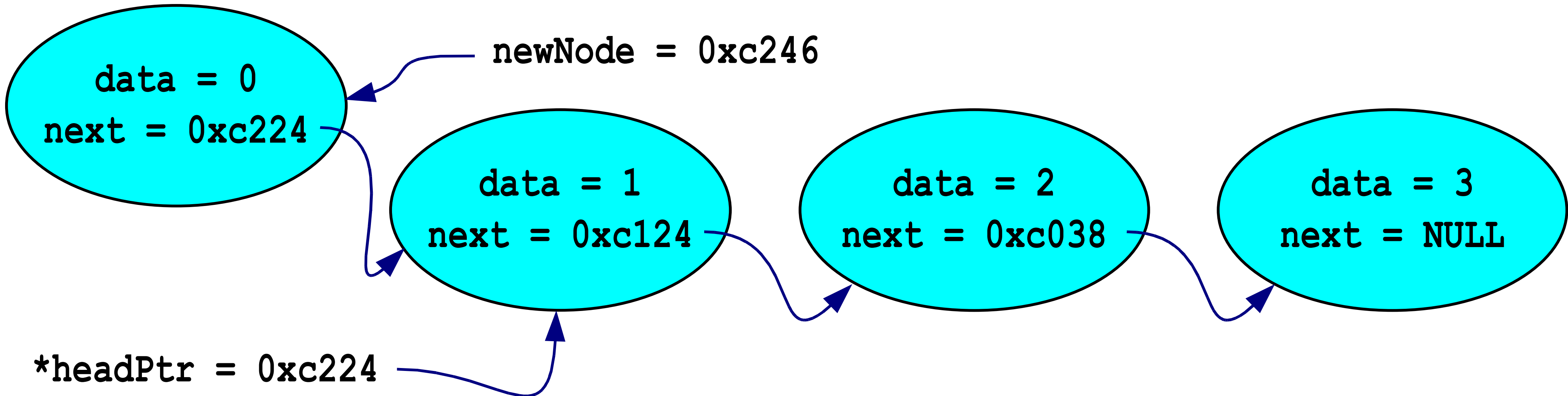



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

void PushTest ()
{
  struct node *head = BuildOneTwoThree ();
  Push (&head, 0);                /* note the & */
  Push (&head, 13);
  /* head is now the list {13, 0, 1, 2, 3} */
}
```

Add Element in Front of the List

- Pass the pointer to the head to be able to modify it

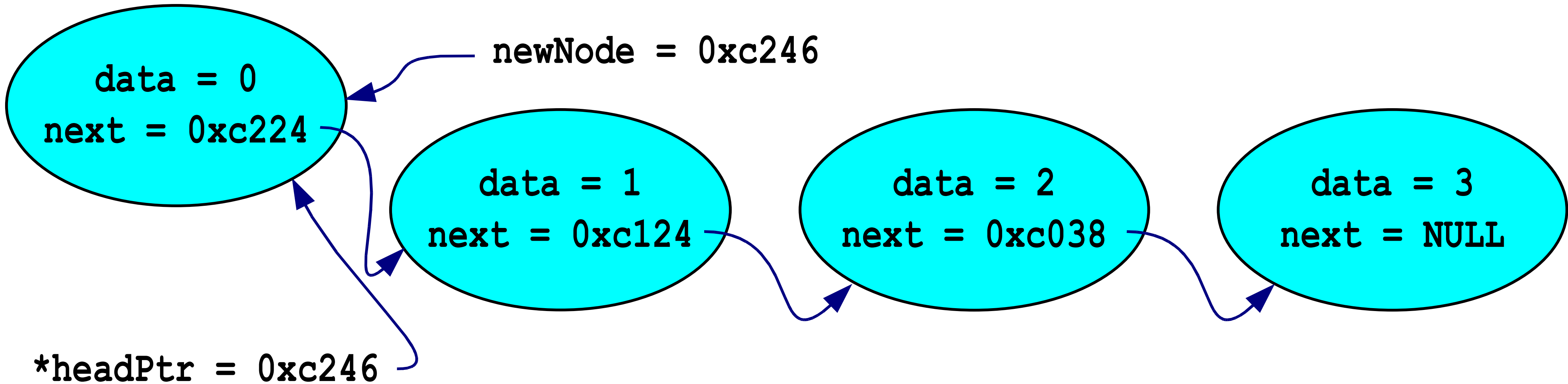



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

void PushTest ()
{
  struct node *head = BuildOneTwoThree ();
  Push (&head, 0);                /* note the & */
  Push (&head, 13);
  /* head is now the list {13, 0, 1, 2, 3} */
}
```

Add Element in Front of the List

- Pass the pointer to the head to be able to modify it

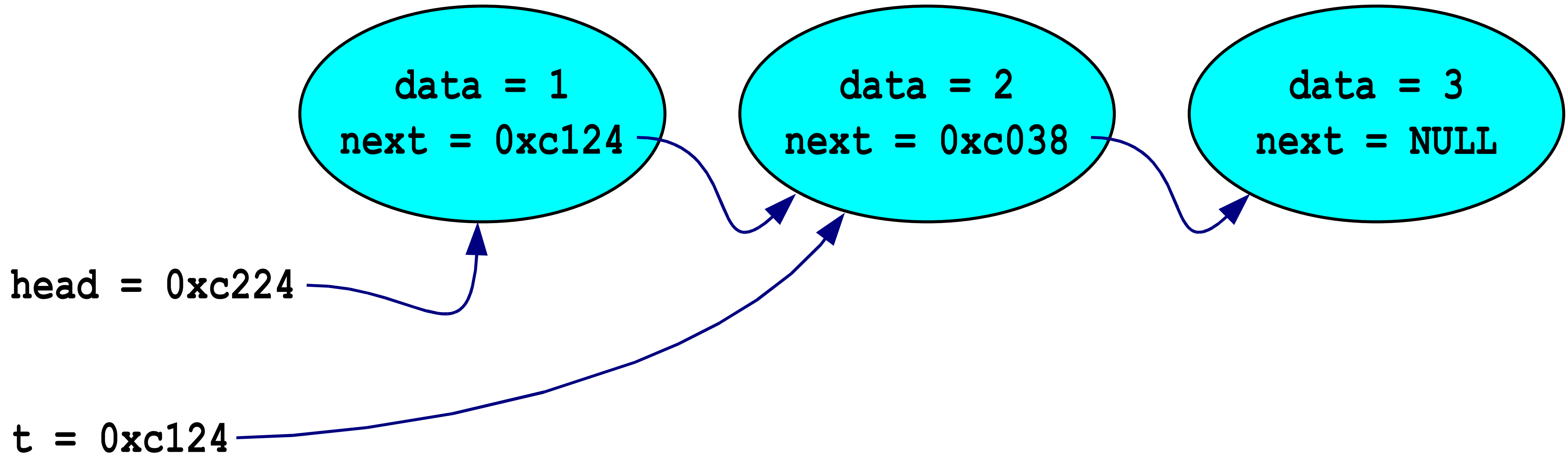



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

void PushTest ()
{
  struct node *head = BuildOneTwoThree ();
  Push (&head, 0);                /* note the & */
  Push (&head, 13);
  /* head is now the list {13, 0, 1, 2, 3} */
}
```

Free List

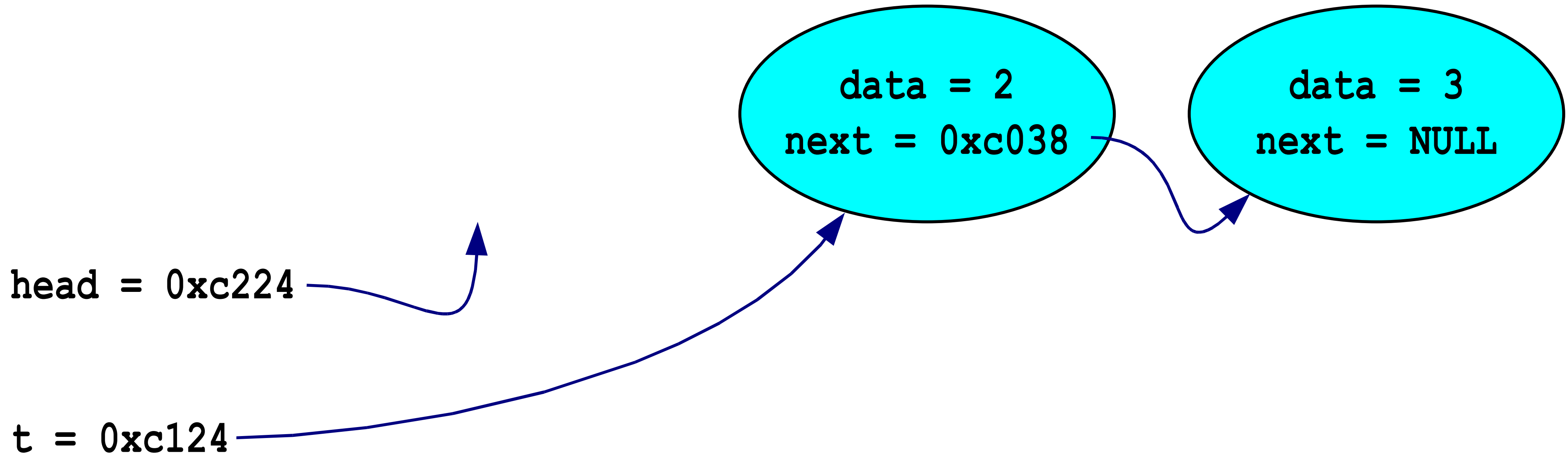
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

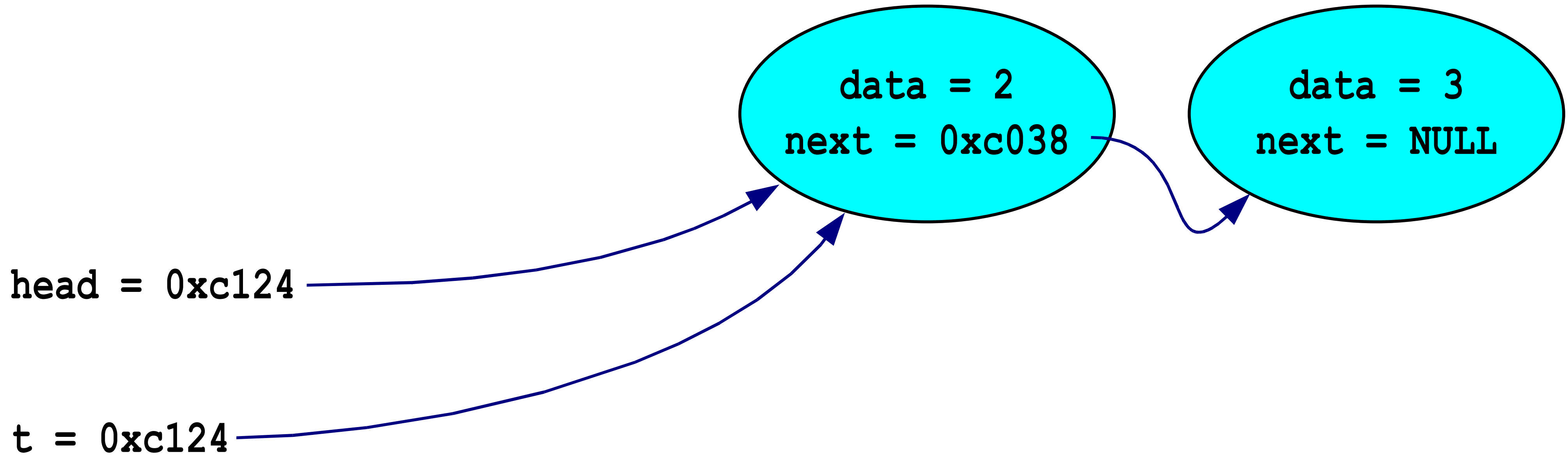
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

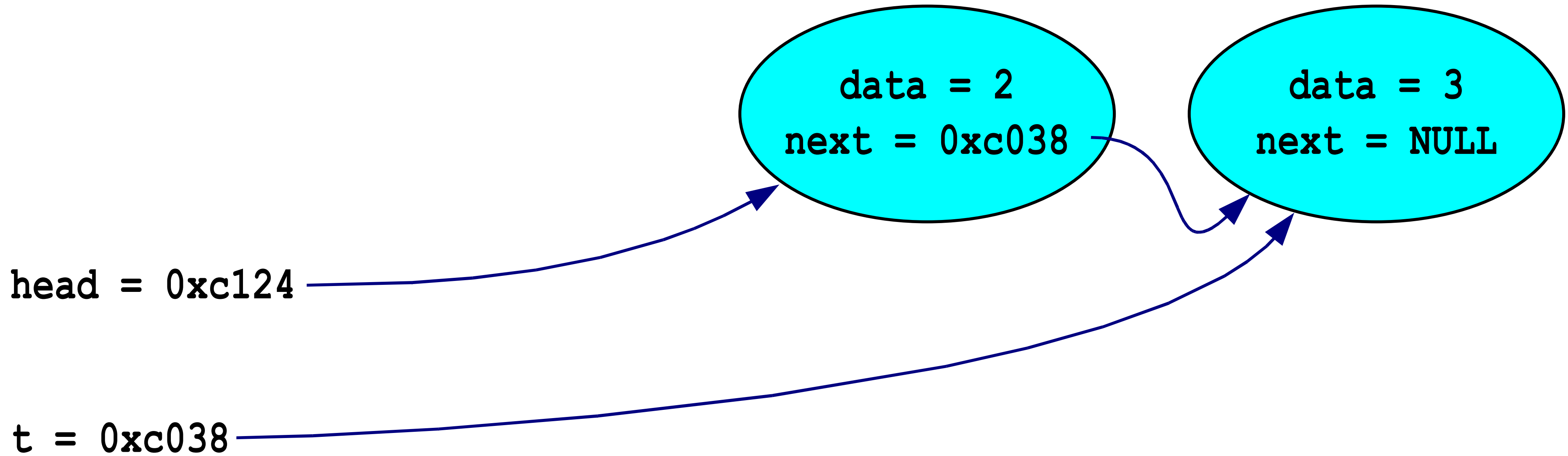
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

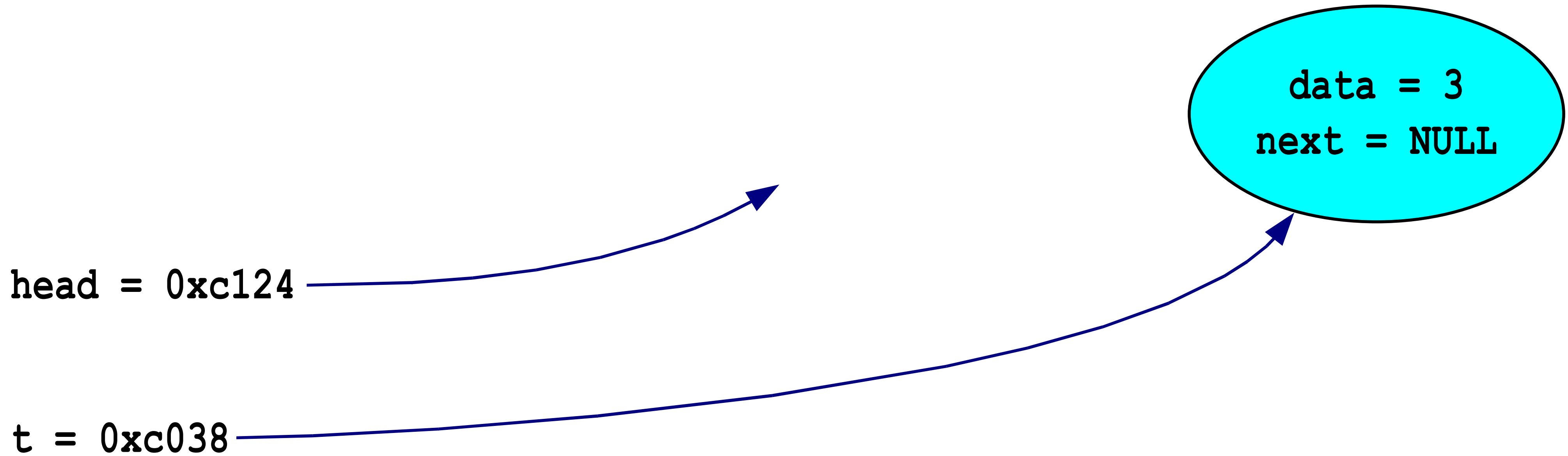
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

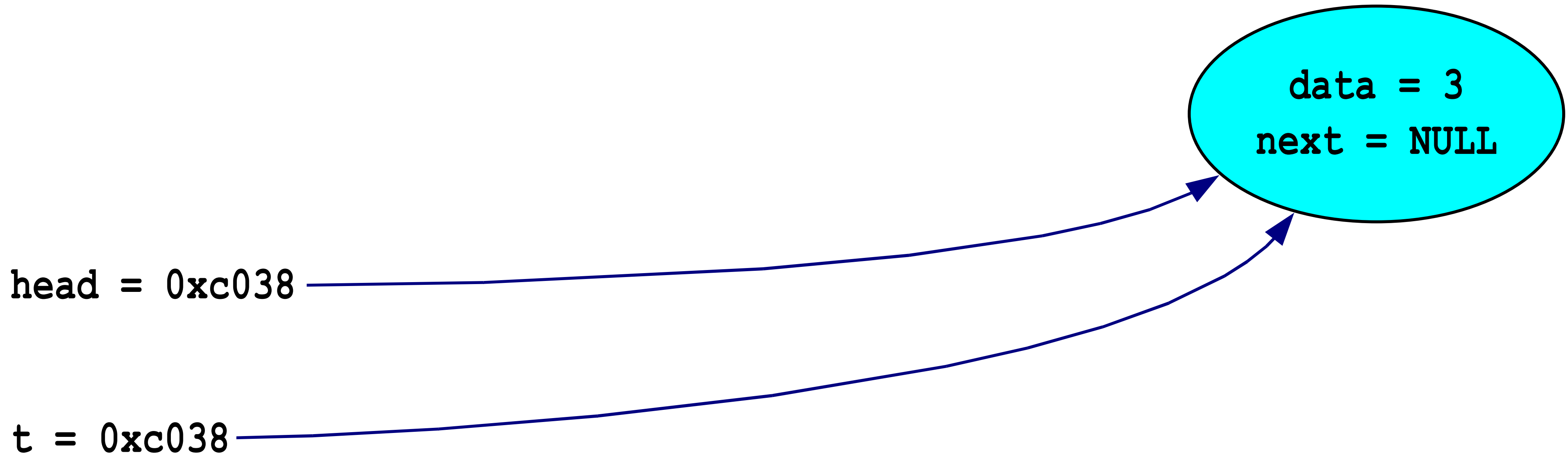
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

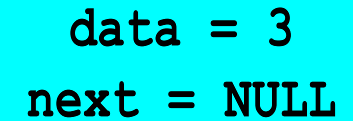
- Free each node starting from the beginning




```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

- Free each node starting from the beginning



head = 0xc038

t = NULL

```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

- Free each node starting from the beginning

head = 0xc038

t = NULL

```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Free List

- Free each node starting from the beginning

`head = NULL`

`t = NULL`

```
void FreeList (struct node *head)
{
  while (head)
    {
      struct node *t = head->next;
      free (head);
      head = t;
    };
}
```

Incorrect Implementation of Free List

- Version below is shorter, but incorrect
  - It attempts to access the freed memory area
  - We cannot get rid of the temporary variable

```
void FreeList (struct node *head)
{
  while (head)
    {
      free (head);
      head=head->next;
    };
}
```

Append Node

- Function appends the node at the end of the list

```
void AppendNode (struct node **headRef, int num)
{
  struct node *current = *headRef;
  struct node *newNode;
  newNode = malloc (sizeof (struct node));
  newNode->data = num;
  newNode->next = NULL;
  /* special case for length 0 */
  if (current == NULL)
    {
      *headRef = newNode;
    }
  else
    {
      /* Locate the last node */
      while (current->next != NULL)
        {
          current = current->next;
        }
      current->next = newNode;
    }
}
```

Copy List

- Function returns the pointer to the copy of the list
- Note there is no need for special case for an empty list

```
struct node * CopyList (struct node *src)
{
  struct node *head = NULL;
  struct node **dst=&head;
  while (src)
    {
      *dst = malloc (sizeof (struct node));
      (*dst)->data = src->data;
      (*dst)->next = NULL;
      src = src->next;
      dst = &((*dst)->next);
    }
  return head;
}
```

Homework Problems

- Write a **`Pop()`** function that is the inverse of **`Push()`**. **`Pop()`** takes a non-empty list, deletes the head node, and returns the head node's data.

```
void PopTest() {
  struct node* head = BuildOneTwoThree(); // build {1, 2, 3}
  int a = Pop(&head); /* deletes "1" node and returns 1 */
  int b = Pop(&head); /* deletes "2" node and returns 2 */
  int c = Pop(&head); /* deletes "3" node and returns 3 */
  int len = Length(head); /* the list is now empty, so len == 0 */
}
```

- Write an iterative **`Reverse()`** function that reverses a list in place by rearranging all the **`.next`** pointers and the **`head`** pointer.

```
void ReverseTest() {
  struct node* head;
  head = BuildOneTwoThree();
  Reverse(&head);
  /* head now points to the list {3, 2, 1} */
  FreeList(head);
}
```

Homework Problems

- Write the **`Sort()`** functions that sorts the list in place in ascending order. Use the *bubblesort* algorithm.

```
void SortTest() {
  struct node* head = NULL;
  int i;
  for(i=0;i<10;i++)
    Push(&head,i)
  /* head now points to the list {9, 8, ... , 1, 0} */
  Sort(&head);
  /* head now points to the list {0, 1, 2, ... , 9} */
  FreeList(head);
}
```

- Write a complete set of functions (including above homework problems) for the list of zero-terminated strings. List should store the copies of strings, so do not forget about the proper memory management.