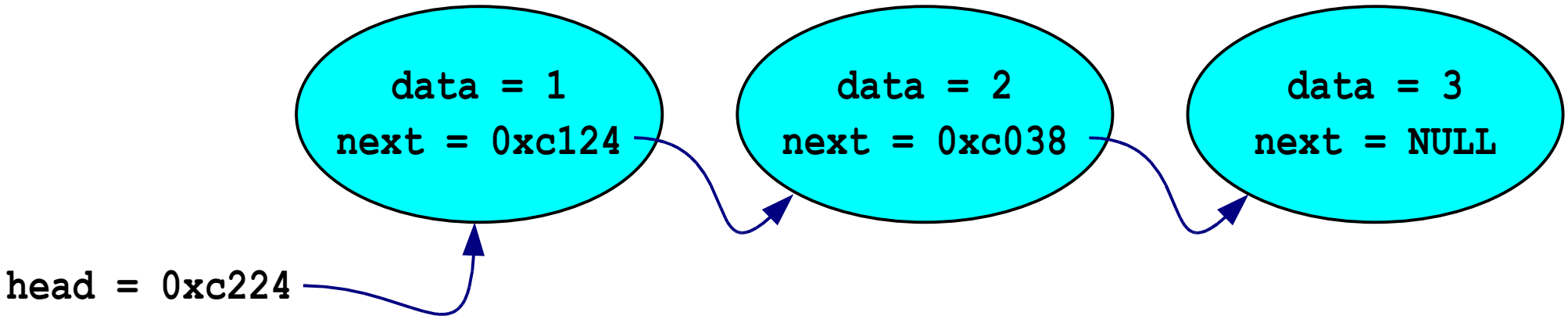
Lista z dowiązaniami

- Do tej pory poznaliśmy dwa rodzaje struktur danych, będących kolekcjami
  - tablica
  - stos
- Lista z dowiązaniami to kolejna kolekcja
- Tablice, stosy i listy przechowuja "elementy" dla programu "klienta".
- Konkretny typ elementu nie ma znaczenia, gdyż w zasadzie identyczna struktura może przechowywać elementy każdego typu.
  - Tablica / stos liczb całkowitych
  - Tablica / stos liczb zmiennoprzecinkowych
  - Tablica / stos kont klientów
  - Tablica / stos ...

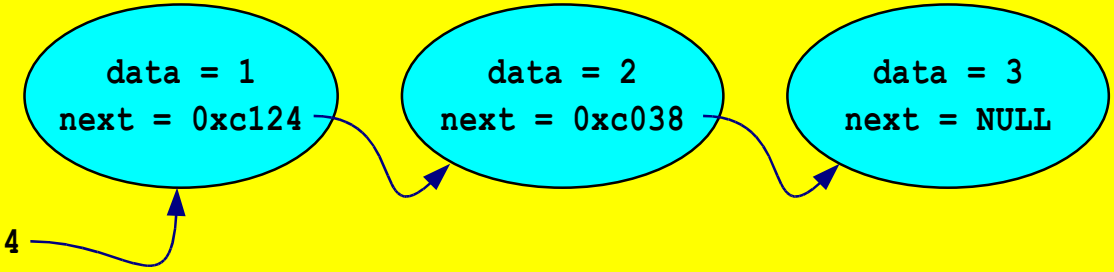
Struktura listy z dowiązaniami

- Tablica alokuje pamięć dla wszystkich swoich elementów w ciągłym obszarze pamięci
- LIsta alokuje pamięć dla każdego elementu oddzielnie we własnym bloku pamięci zwanym węzłem. Wszystkie węzły listy są połączone razem jak ogniwa w łańcuchu.
- Każdy węzeł zawiera dwa pola: "data" przechowujący dane dla klienta i "next" będące wskaźnikiem do kolejnego węzła należącego do listy.
- Każdy węzeł jest zaalokowany na stercie przy pomocy funkcji **`malloc()`**, więc istnieje dopóki pamięć nie zostanie zwolniona przy pomocy funkcji **`free()`**. Początek listy to wskaźnik do jej pierwszego węzła.



Tworzenie listy

- Poniższy przykład tworzy listę zawierająca trzy elementy



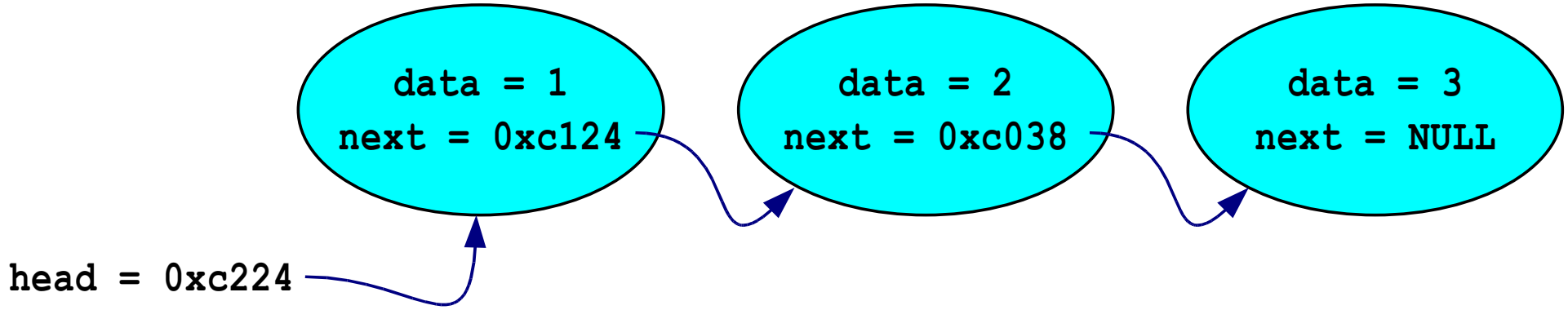

```
struct node
{
  int data;
  struct node *next;
};

/* 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;
}
```

Liczenie elementów listy

- Przekazujemy listę poprzez przekazanie wskaźnika do pierwszego węzła
- Iterujemy przez całą listę z użyciem lokalnego wskaźnika



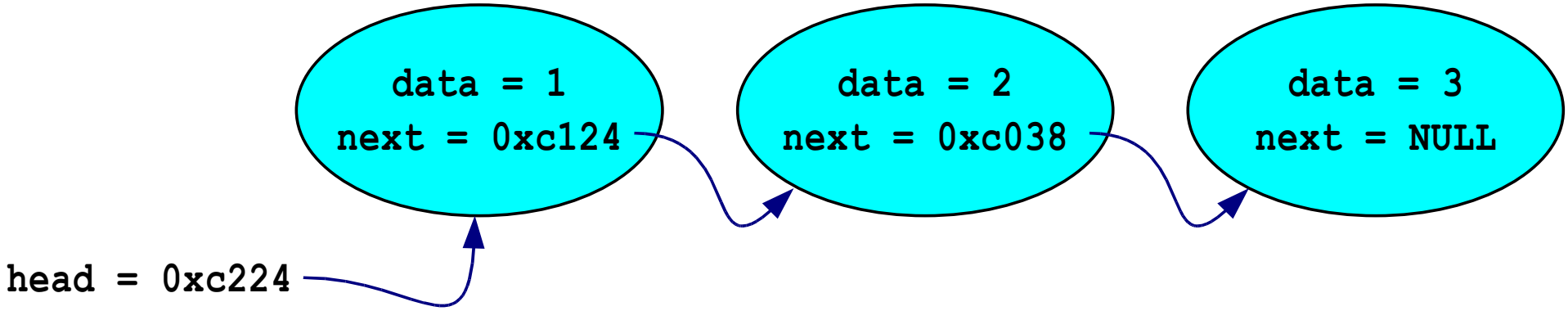

```
/* Given a linked list head pointer, compute
   and return the number of nodes in the list. */

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 */
}
```

Wyświetlenie elementów listy

- Iterujemy przez całą listę z użyciem lokalnego wskaźnika
- Wypisujemy dane zawarte w każdym węźle

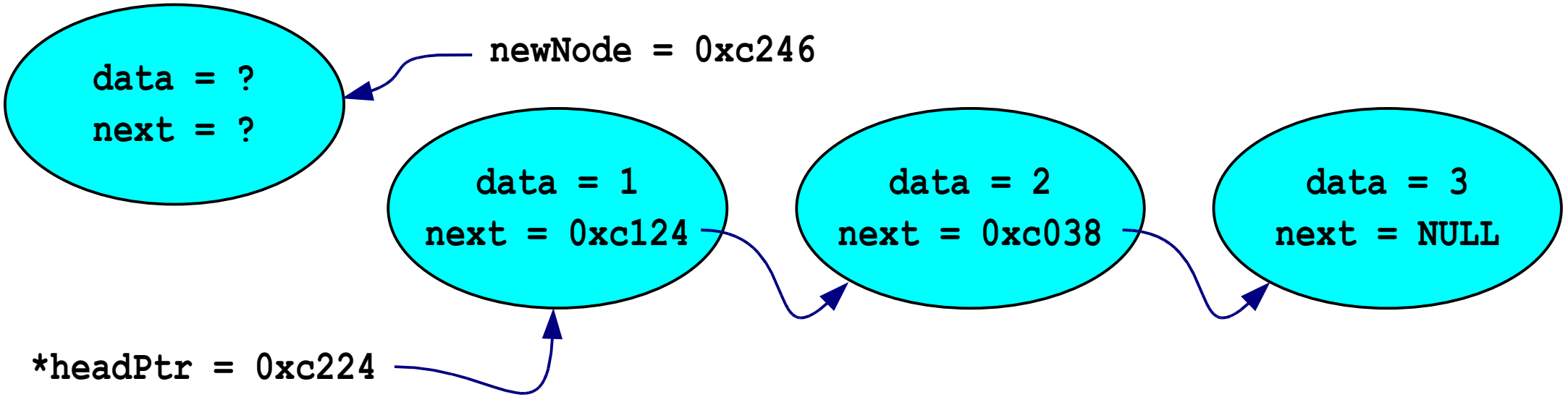



```
/* Given a linked list head pointer, display
   all numbers stored in the list. */

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

Dodawanie elementu na początku listy

- Przekazujemy wskaźnik do wskaźnika aby móc go modyfikować

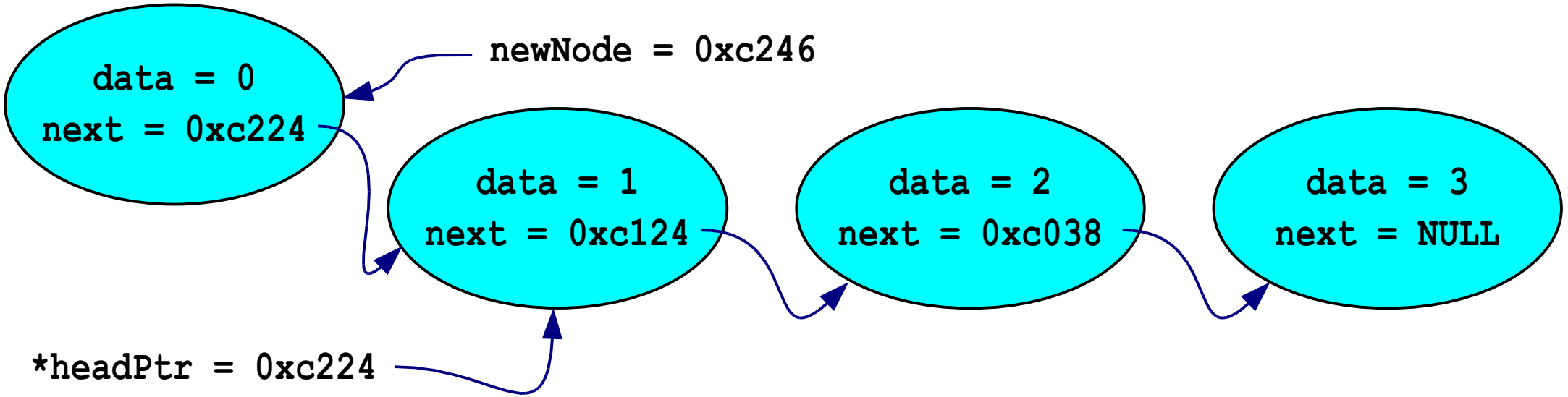



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

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

Dodawanie elementu na początku listy

- Przekazujemy wskaźnik do wskaźnika aby móc go modyfikować

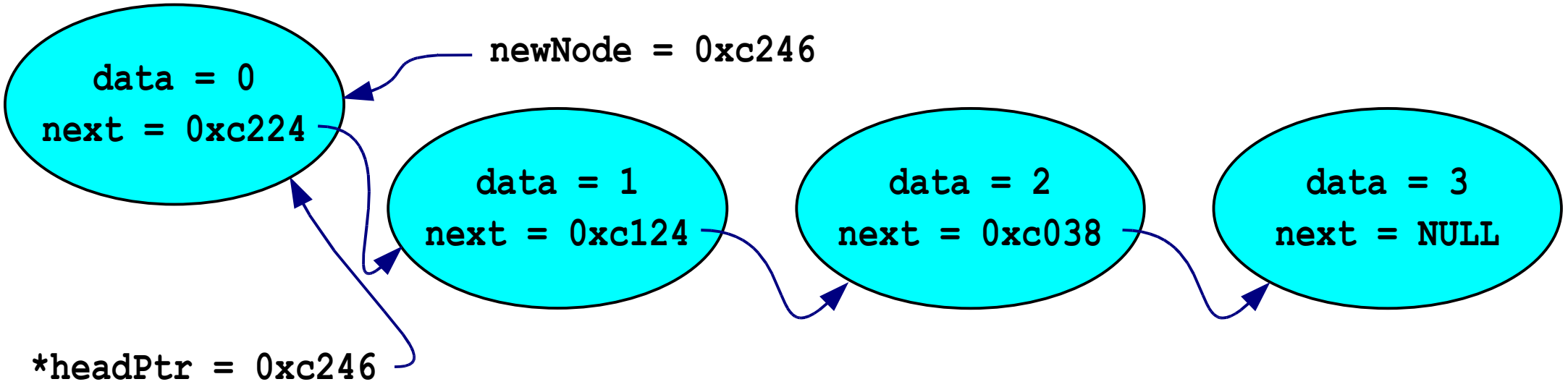



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

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

Dodawanie elementu na początku listy

- Przekazujemy wskaźnik do wskaźnika aby móc go modyfikować

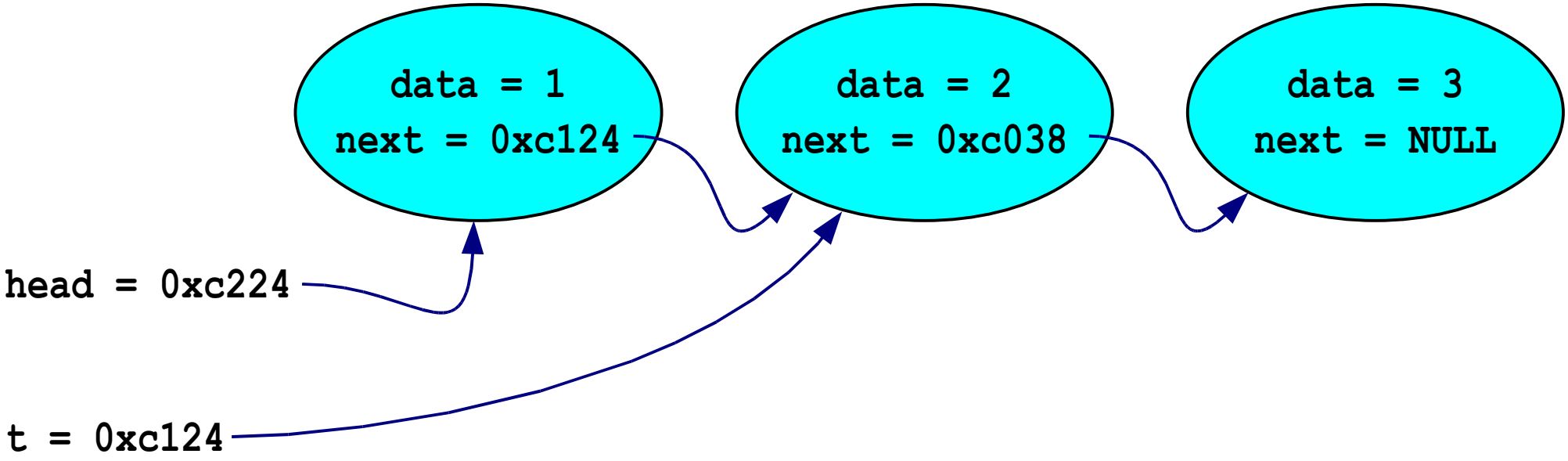



```
void Push (struct node **headPtr, int data)
{
  struct node *newNode = malloc (sizeof (struct node));
  newNode->data = data;
  newNode->next = *headPtr;
  *headPtr = newNode;
}

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

Zwolnienie listy

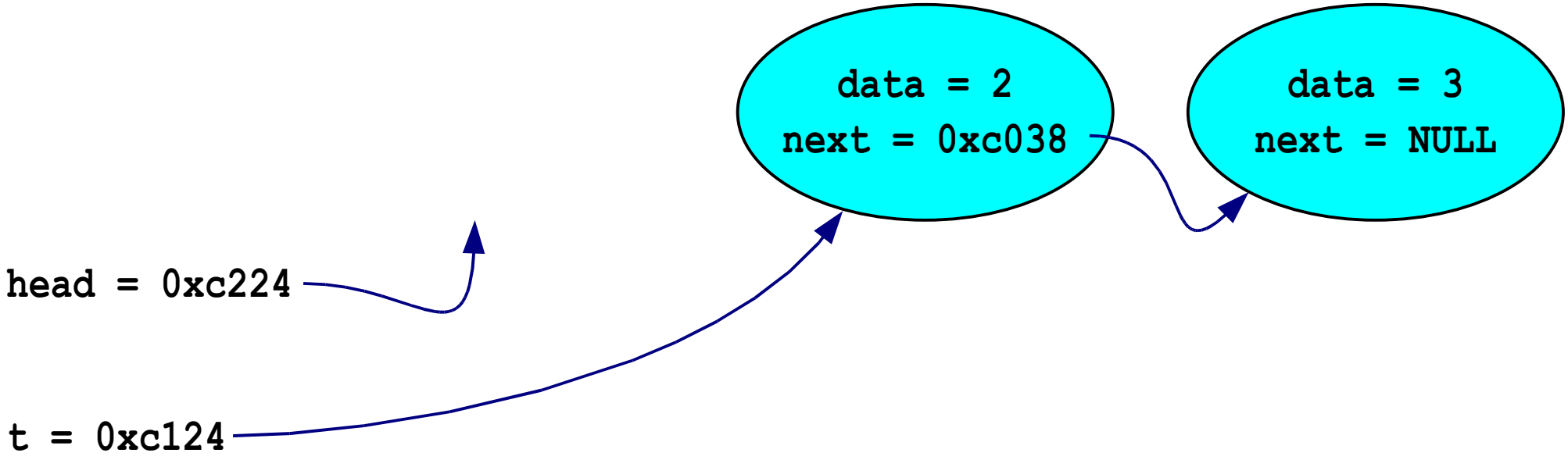
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

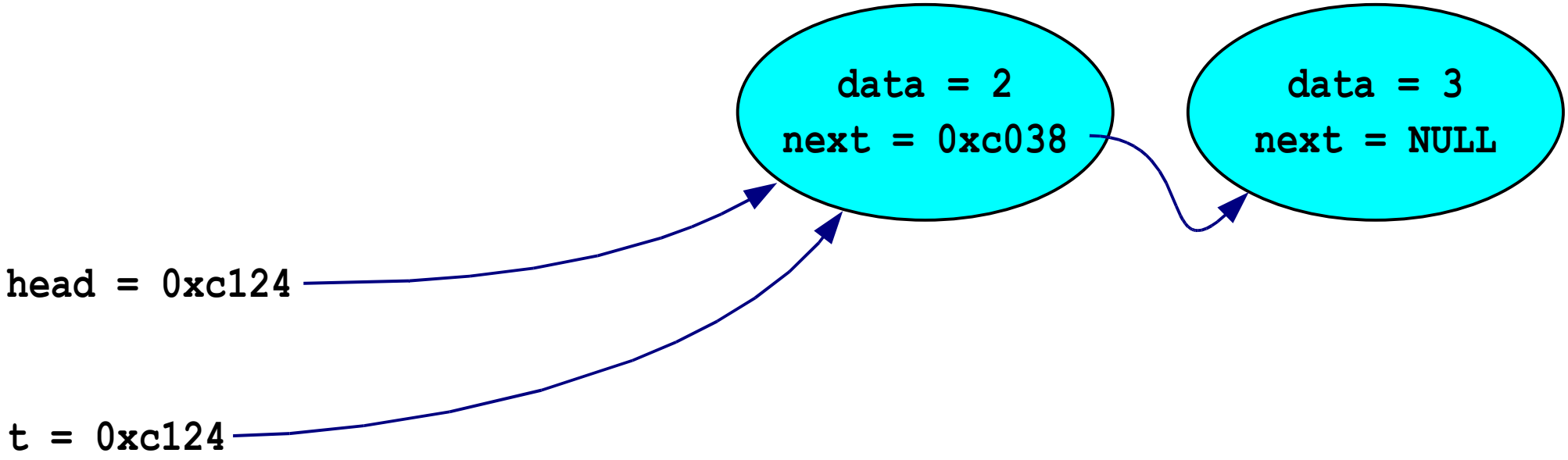
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

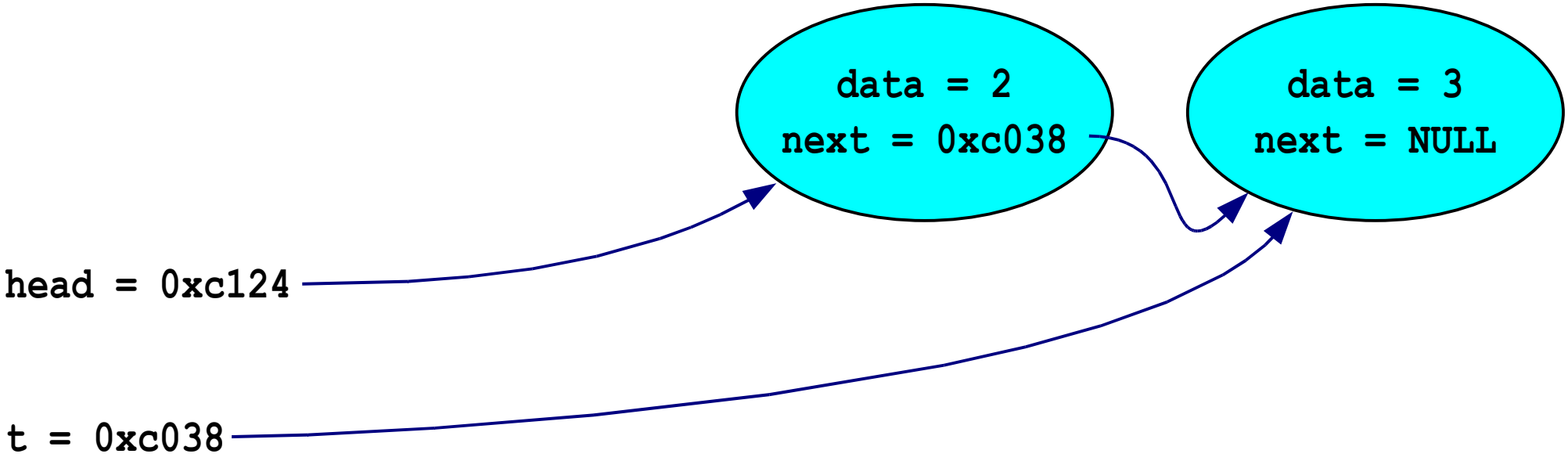
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

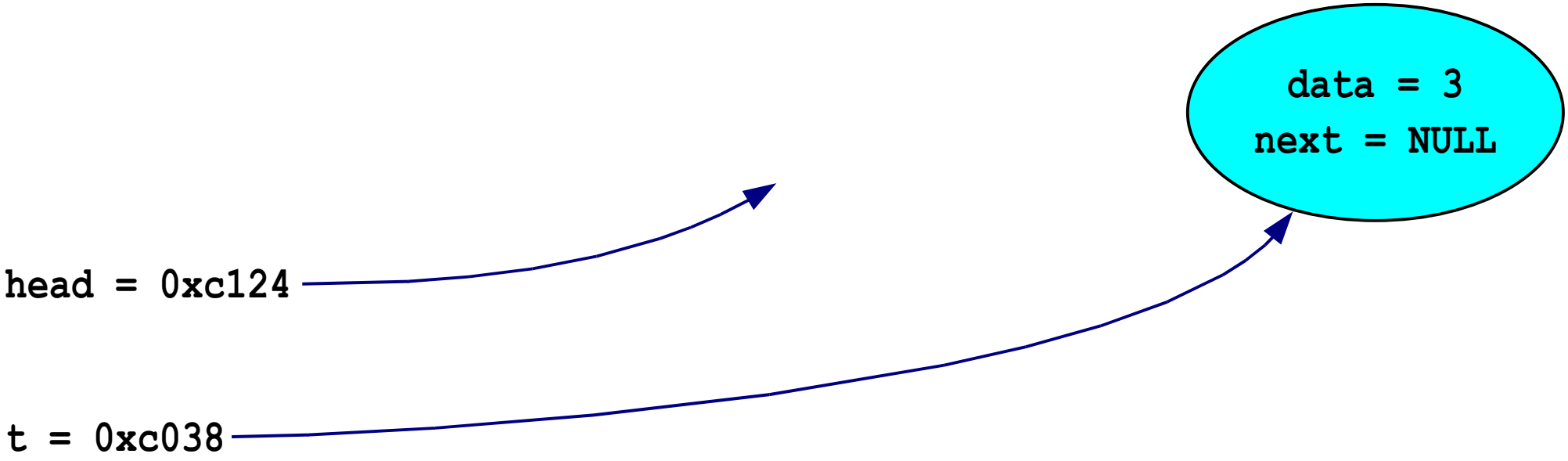
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

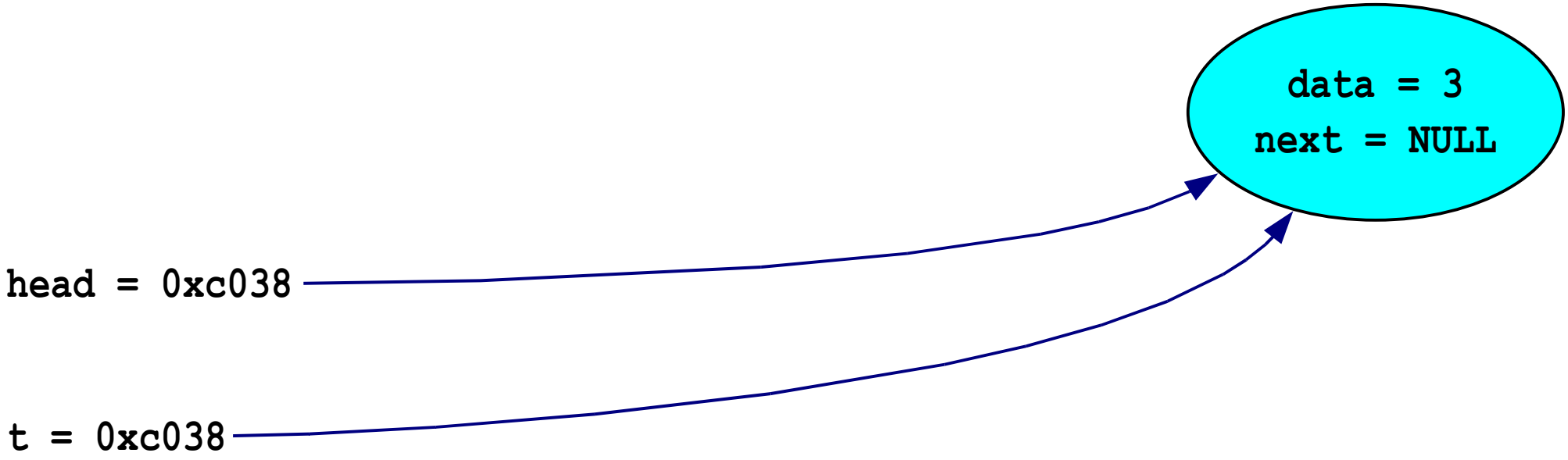
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

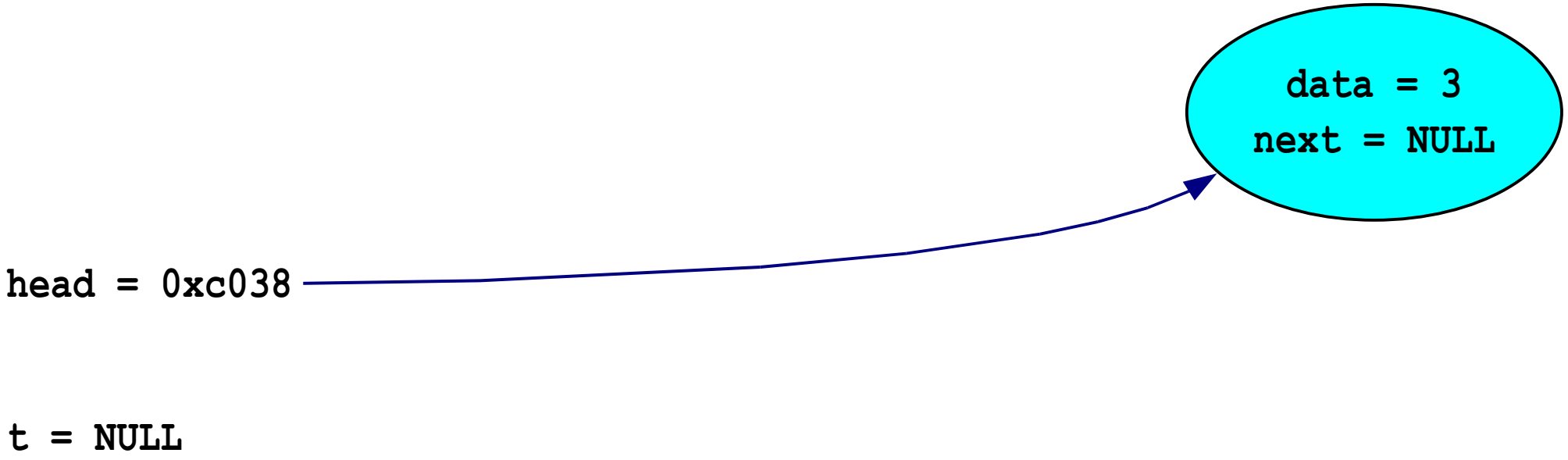
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

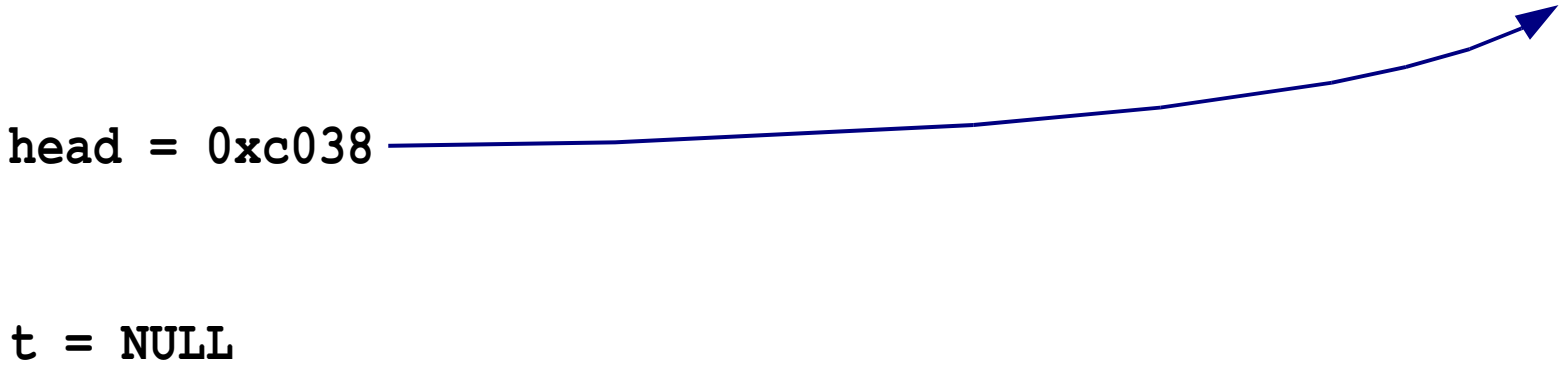
- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

- Zwalniamy każdy węzeł poczynając od początkowego




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

Zwolnienie listy

- Zwalniamy każdy węzeł poczynając od początkowego

```
head = NULL
```

```
t = NULL
```

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

Niepoprawna implementacja zwalniania listy

- Wersja poniżej jest krótsza, ale niepoprawna
  - Próbuje odczytać zawartość zwolnionego obszaru pamięci
  - Nie możemy wyeliminować zmiennej tymczasowej

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

Dodawanie węzła

- Funkcja dodaje węzeł na końcu listy

```
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;
    }
}
```

Kopiowanie listy

- Funkcja zwraca wskaźnik do kopii listy
- Nie ma potrzeby szczególnego traktowania pustej listy

```
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;
}
```

Praca domowa

- Napisać funkcję **`Pop()`** która jest przeciwieństwem funkcji **`Push()`**. **`Pop()`** pobiera niepustą listę, usuwa początkowy węzeł i zwraca dane pobrane z początkowego węzła.

```
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 */
}
```

- Napisać iteracyjną funkcję **`Reverse()`**, zamieniającą kolejność elementów na liście bez alokacji dodatkowej pamięci przez odpowiednią manipulację wskaźnikami **`.next`** i **`head`**.

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

Praca domowa

- Napisać funkcję **`Sort()`** sortującą listę w miejscu. Użyć algorytmu sortowania bąbelkowego.

```
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);
}
```

- Napisać kompletny zestaw funkcji (włączając powyższe zadania domowe) dla listy łańcuchów tekstowych. Lista powinna przechowywać kopie łańcuchów, więc nie należy zapominać o poprawnym zarządzaniu pamięcią.