LECTURE 9:

LINKED LISTS

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)

Beijing Dublin International College

Much of this lecture is adapted from: http://cslibrary.stanford.edu/103/LinkedListBasics.pdf

Arrays

- Arrays are probably the most common way to store **multiple elements of the same type**.
- They also make it very **quick to access** an element using an index.
- However, they do have some **drawbacks**:
  - They have a **fixed size**: we must know the size we want before we create it (or re-create it when it gets full, using `realloc(...)`).
  - It is **difficult to add** new items into an array, as all the elements afterwards must be shifted one place right to make room.
  - It is **difficult to remove** items from an array, as all the later elements must be shifted one place left to fill the gap.
    - These are both slow operations in large arrays.

Linked Lists

- A **linked list** is an alternative way of storing many items of the same type.

- In general, they are strong in the areas that arrays are weak.
  - However they do this by being more complicated!

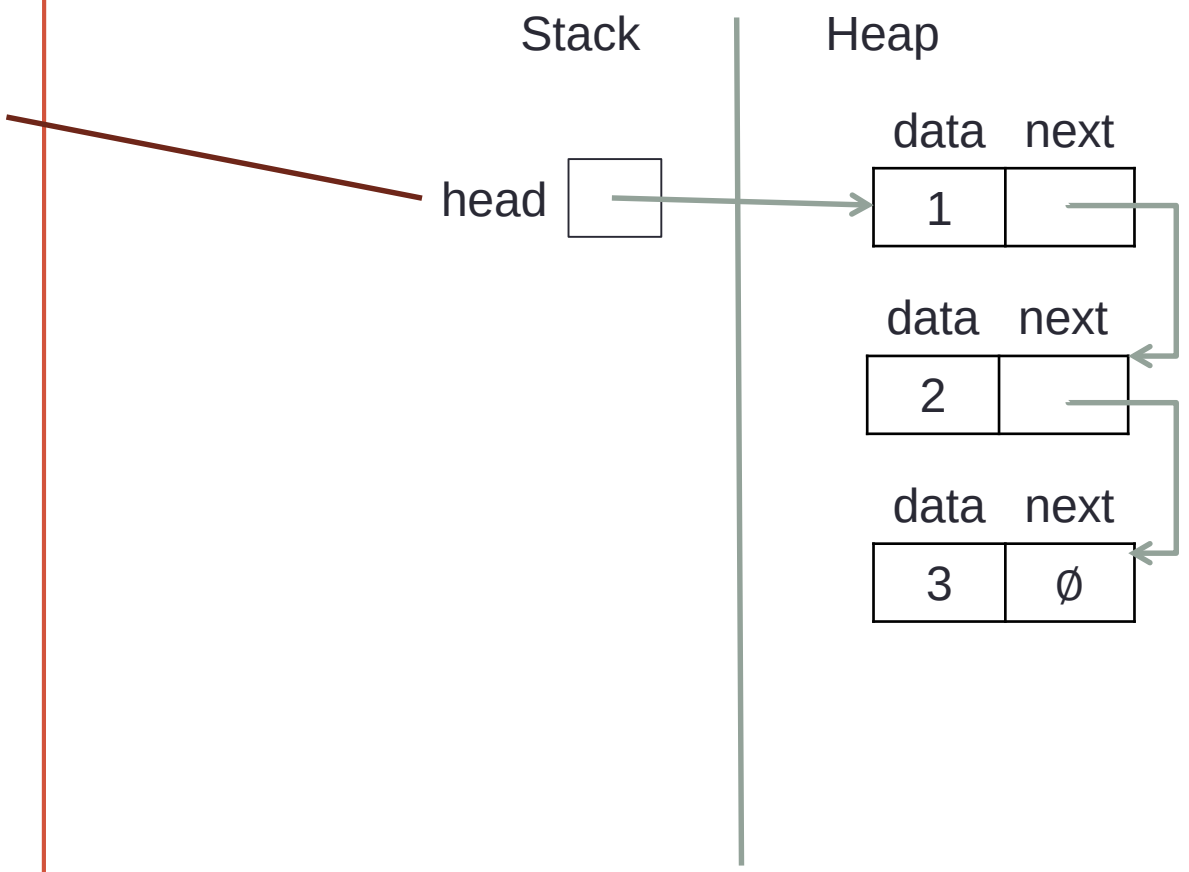
- Instead of allocating one large block of memory where everything is stored (like an array), linked lists allocate a separate piece of memory for each element to be stored.

Thinking about Linked Lists

- In a linked list, every element is stored in a **node**.
- The list is created by using pointers to connect nodes like links in a chain.
- Each node is a structure with two fields:
  - **data**: stores the element;
  - **next**: a pointer to the next node in the list.
- Nodes are allocated on the heap using `malloc(...)`
- It is also important that we keep a pointer to the first node in the list, so we can access it.

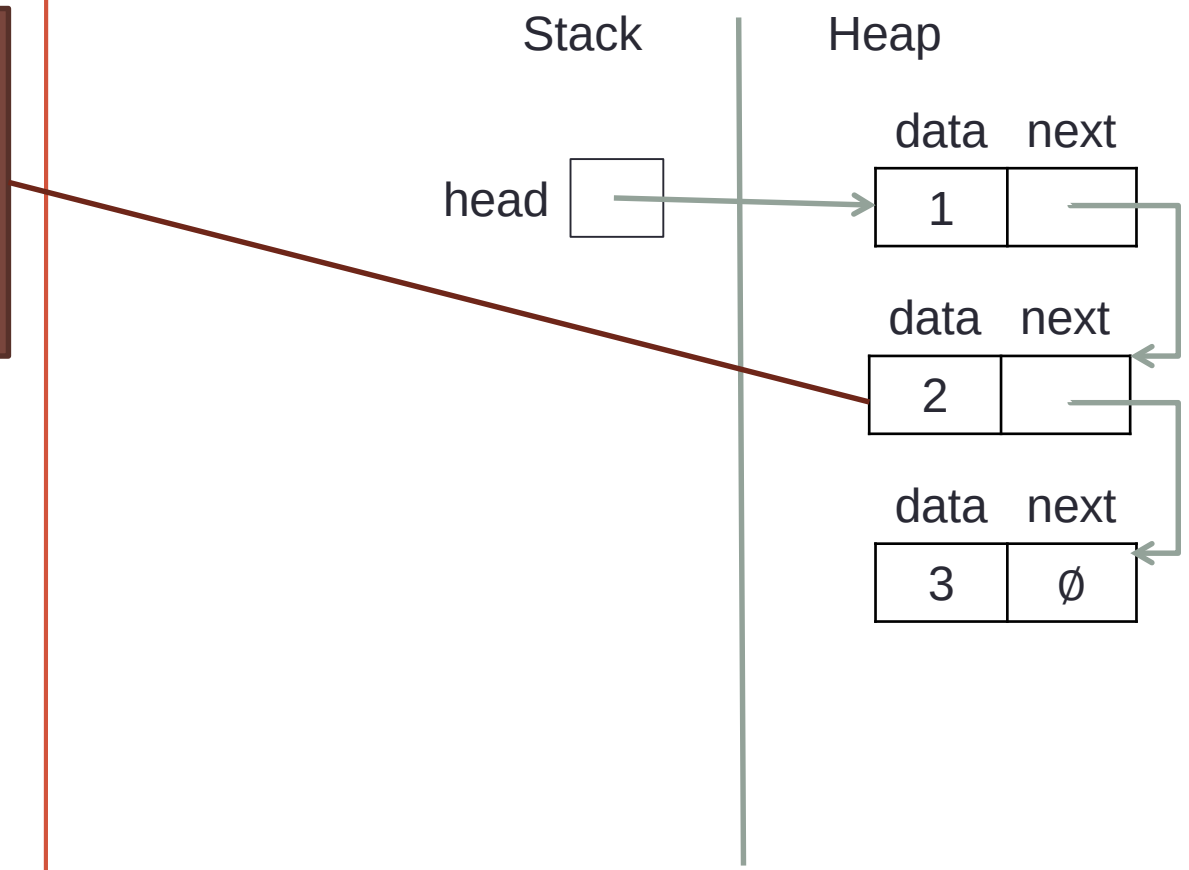
Thinking about Linked Lists

A pointer (called the head) contains the address of the first node, so we can access the list.



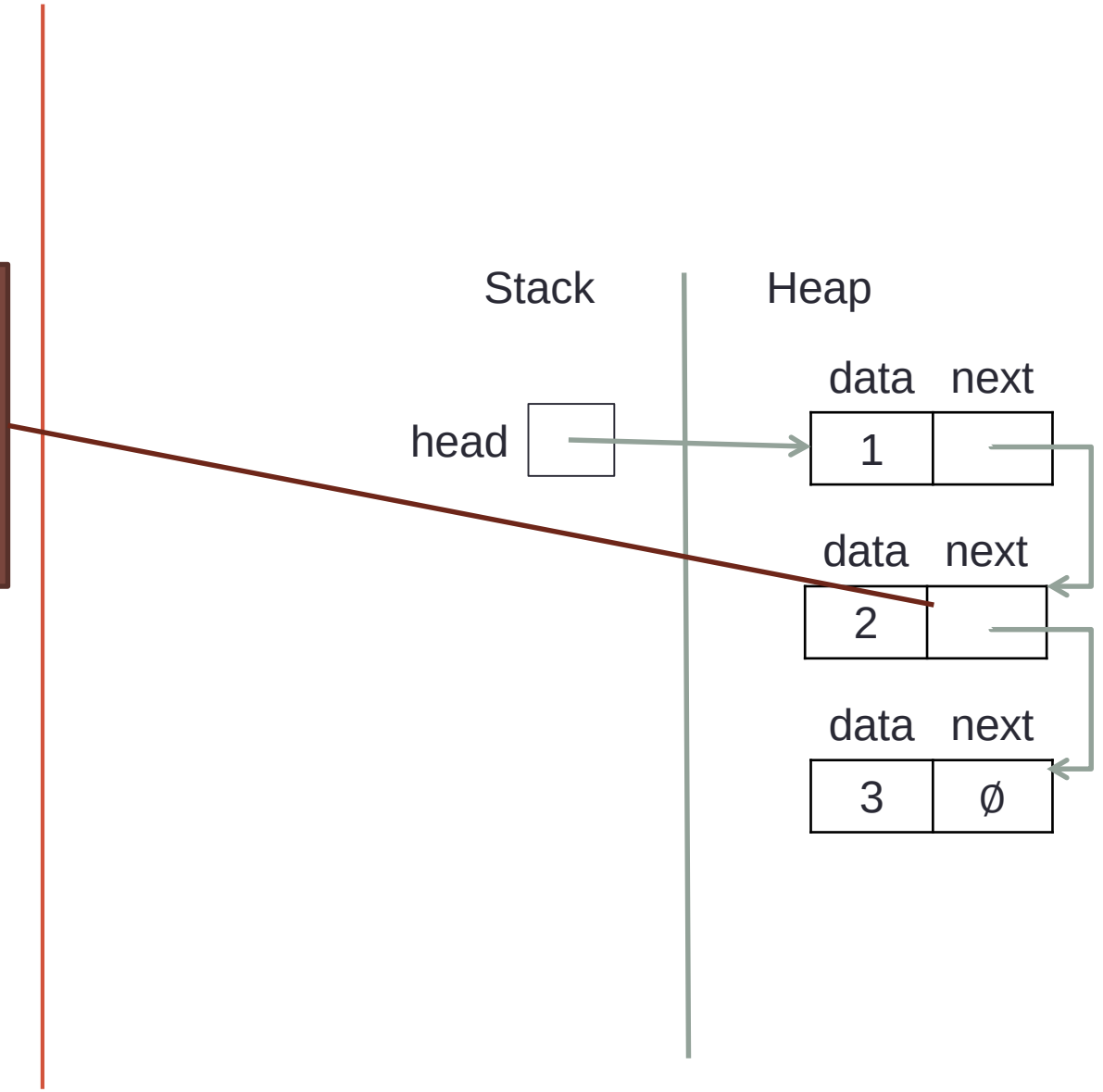
Thinking about Linked Lists

Each node stores one data element (in this example, they are `int` values)

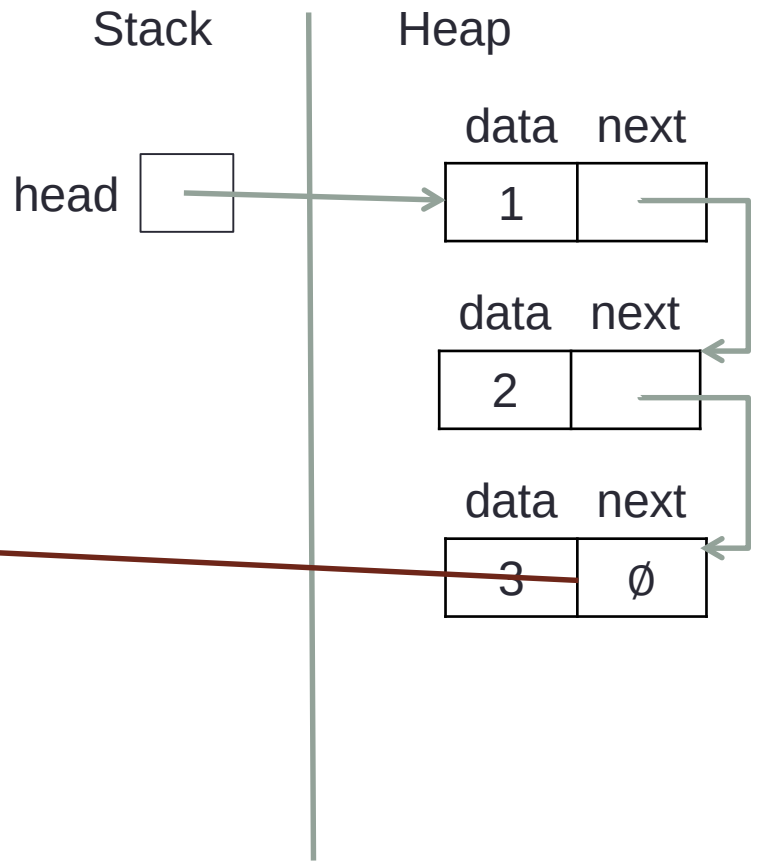


Thinking about Linked Lists

Each node stores one next pointer, to the next node in the list.



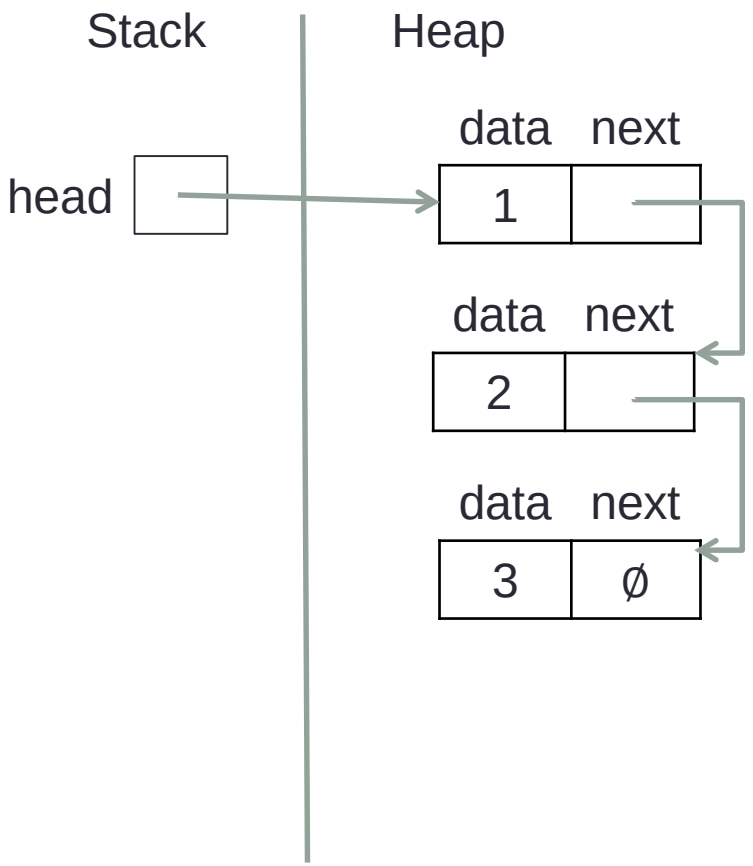
Thinking about Linked Lists



For the last node, the next pointer is NULL.

Thinking about Linked Lists

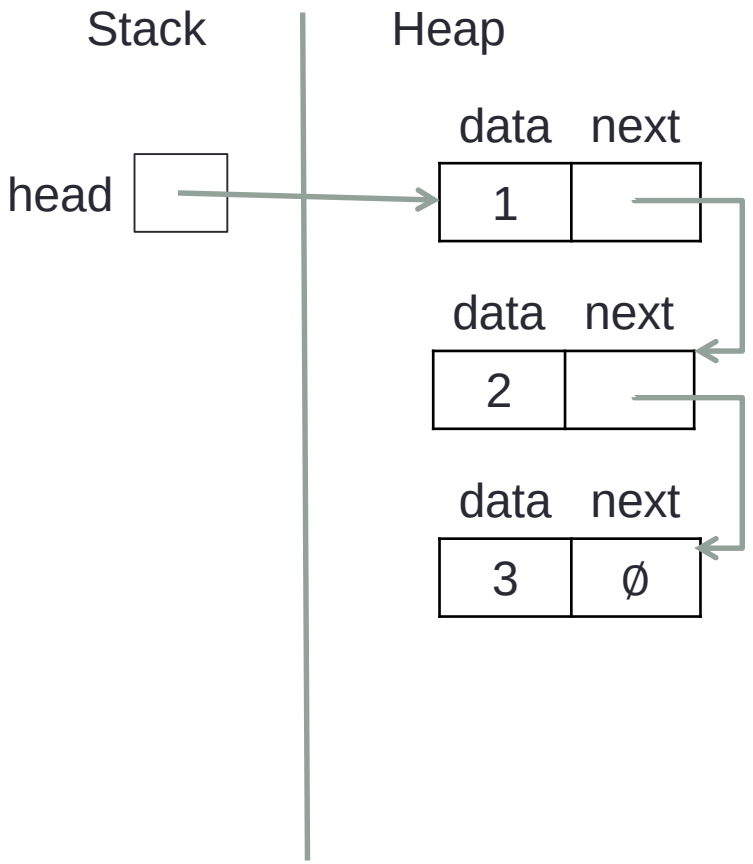
The overall list works by connecting nodes together using their next pointers. Nodes are allocated on the heap using `malloc(...)`.



Thinking about Linked Lists

Code can access any node in the list by following the next pointers, starting at the head of the list.

We will look at how we can build a list, and then how we can use it.



The node structure

- To be able to create this list, we must create a `Node` structure.
- Remember, a `Node` consists of:
  - a `data` field (an `int`, for our example)
  - a `next` field (a pointer to another `Node`).

- We can create a typedef for this structure like this:

```
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

This is a little different to what we have seen before.

Inside the struct, we must refer to "`struct Node`" and not "`Node`", because it is refererring to itself.

Building a Linked List

- Now, we will look at a function that will create a Linked List that stores the elements {1, 2, 3}.

- It will create this list on the heap using `malloc(...)` and return a pointer to the first node in the list (the "head" of the list).

- Let's call this function:

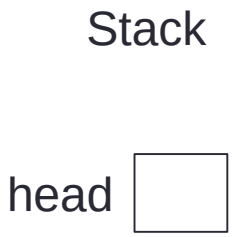
```
Node* build_one_two_three();
```

```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```

Stack

Heap

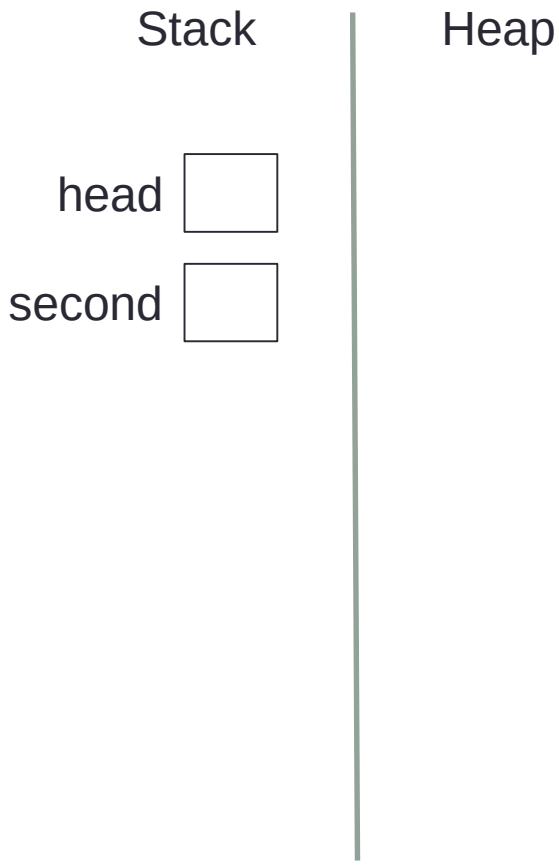
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



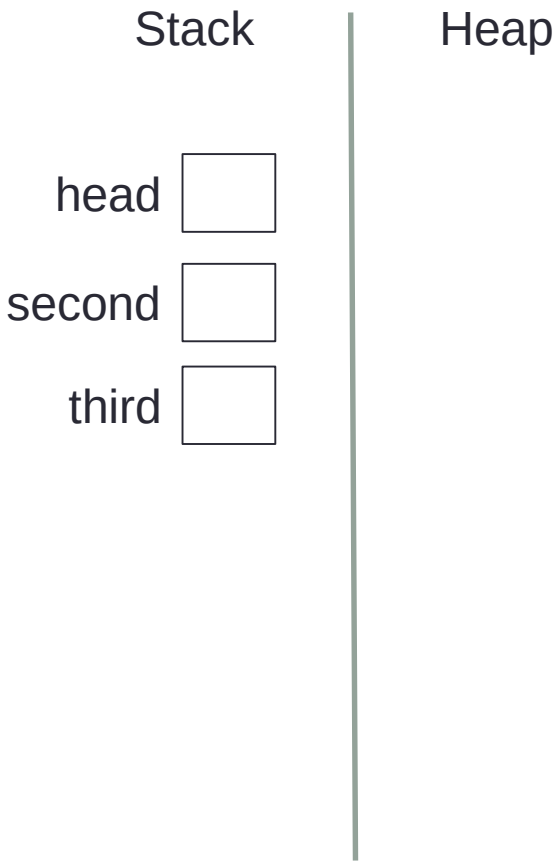
In these diagrams, where a variable is shown as empty, it means that its value is unknown (because it could be anything).

Also, to keep things simple and make the code easier to understand, I am not checking if `malloc()` succeeded.

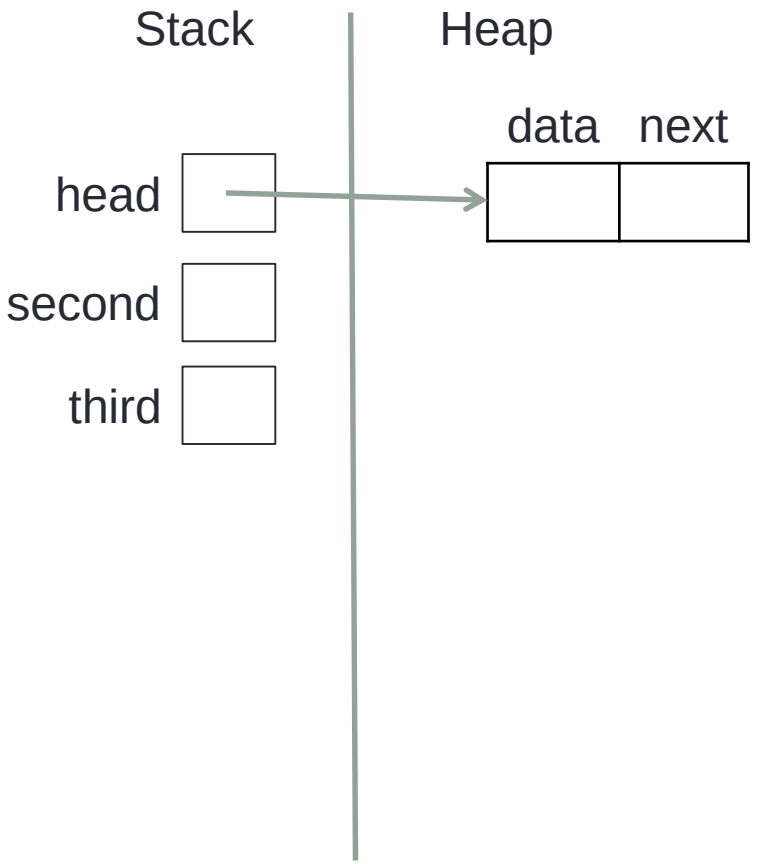
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



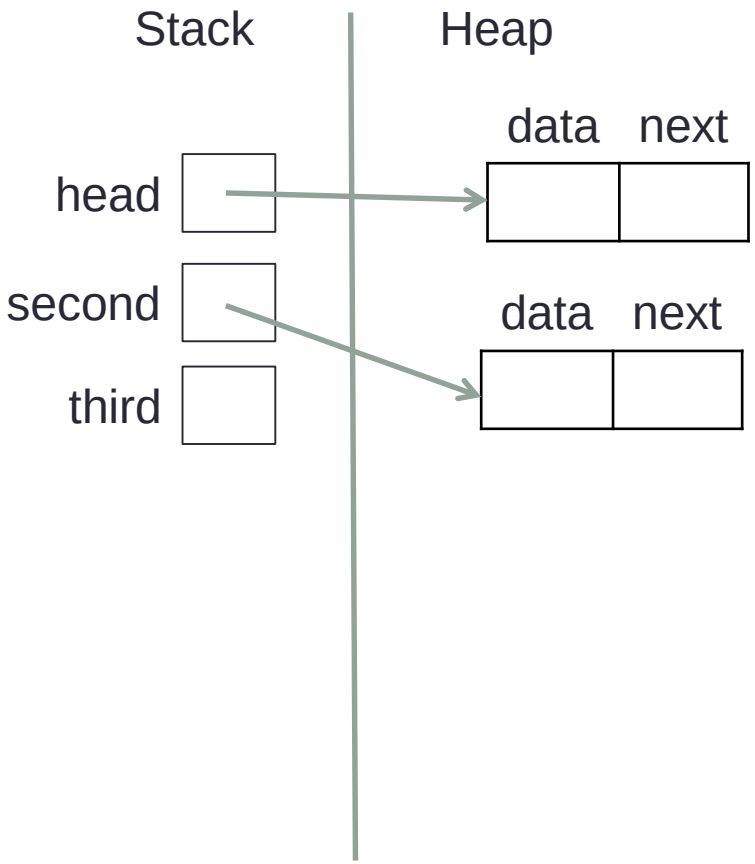
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



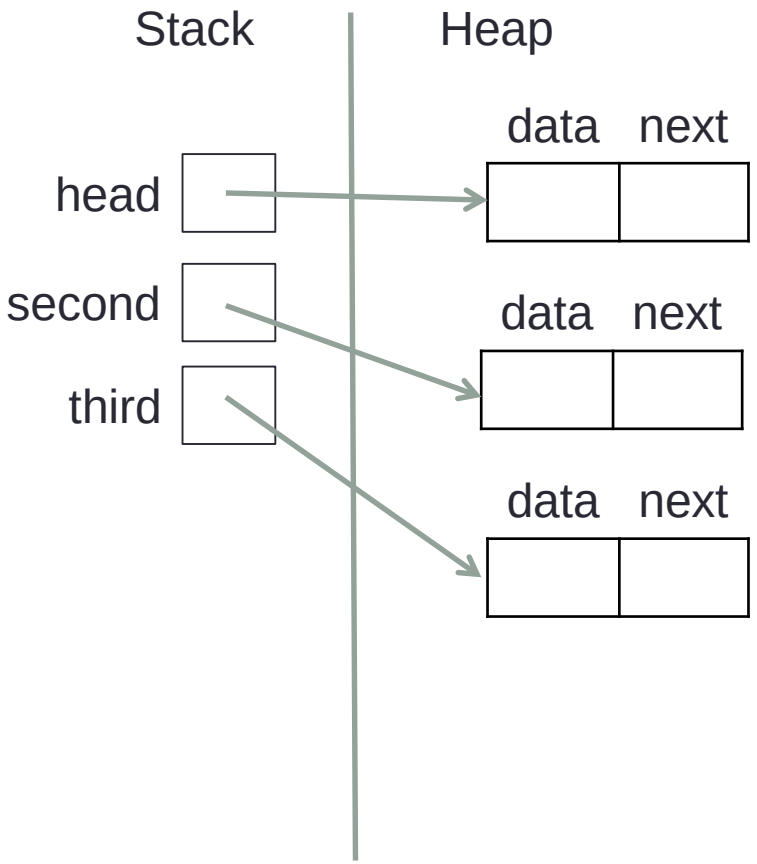
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



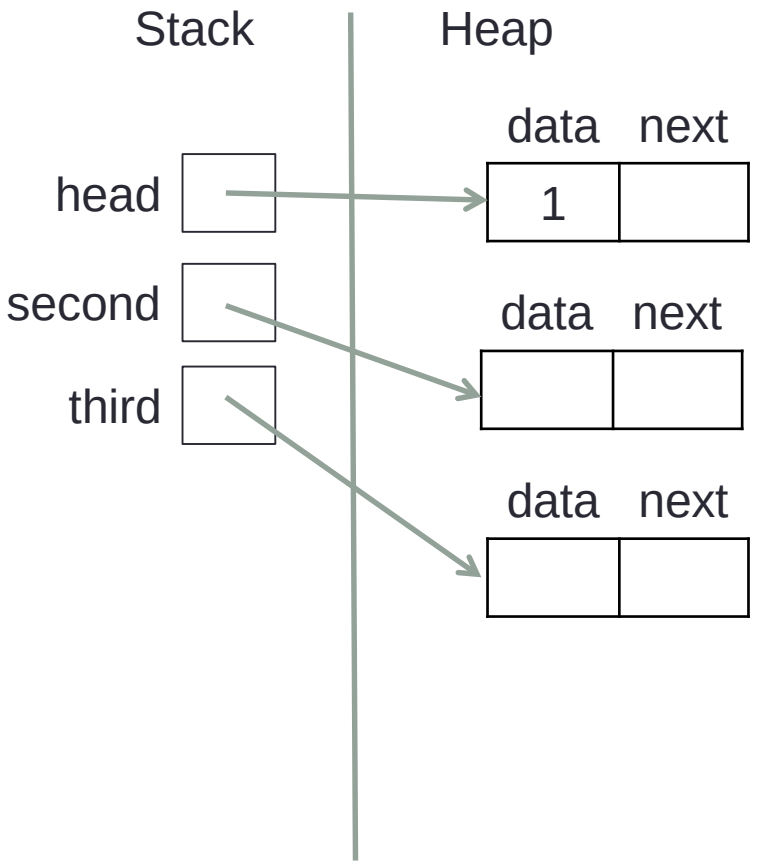
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



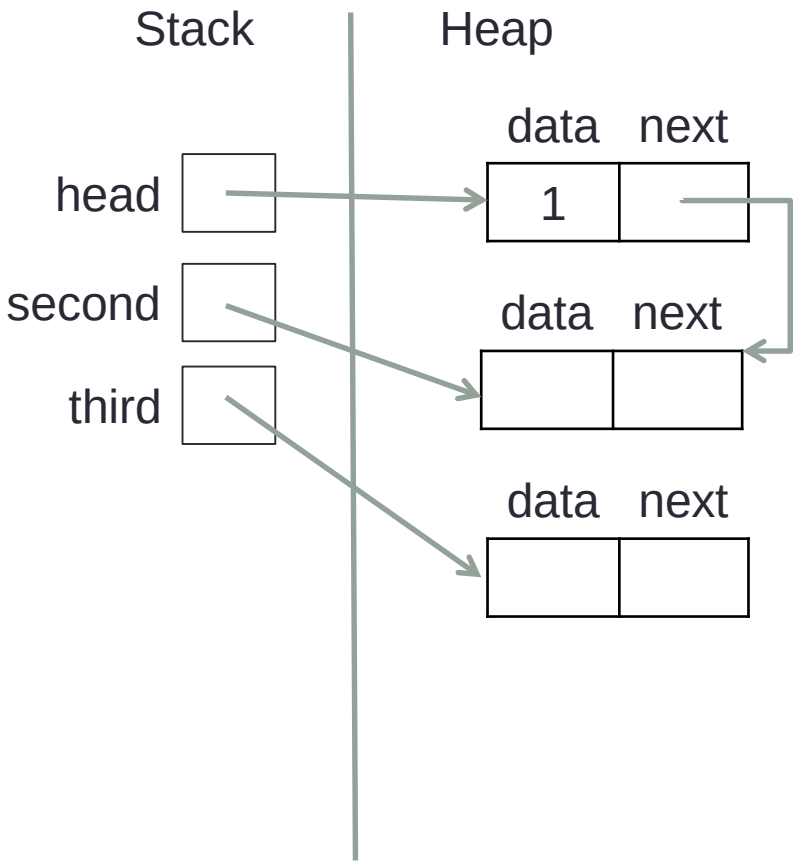
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



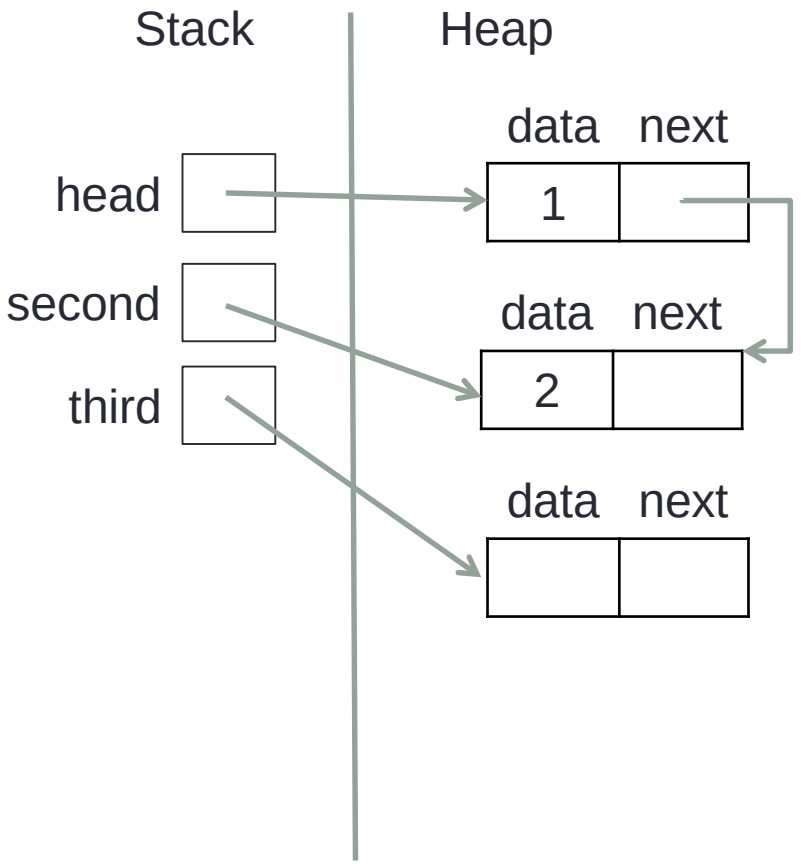
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



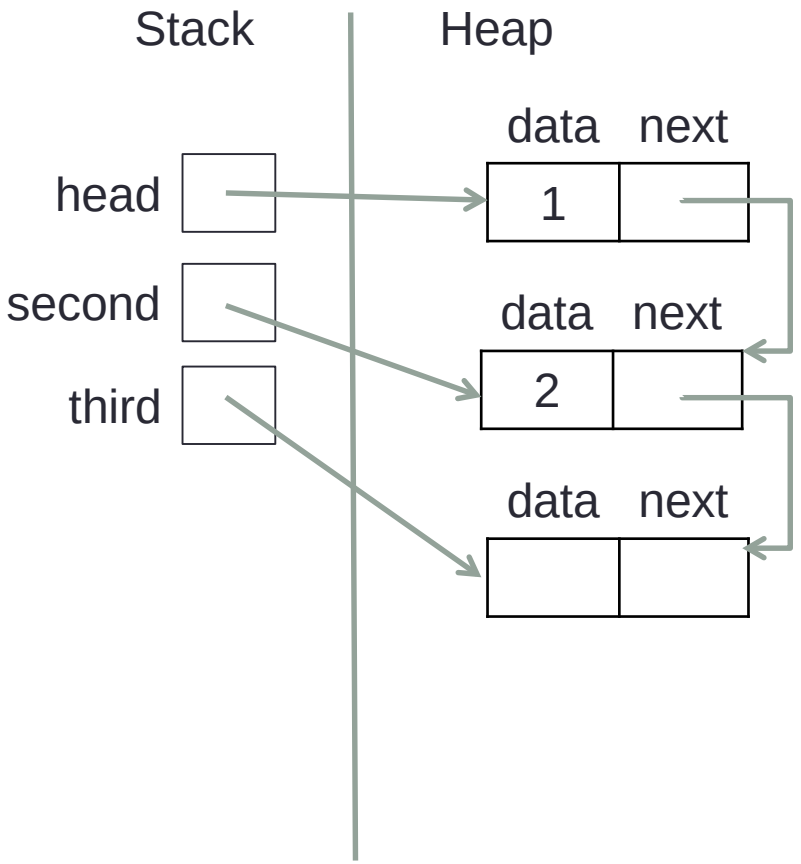
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



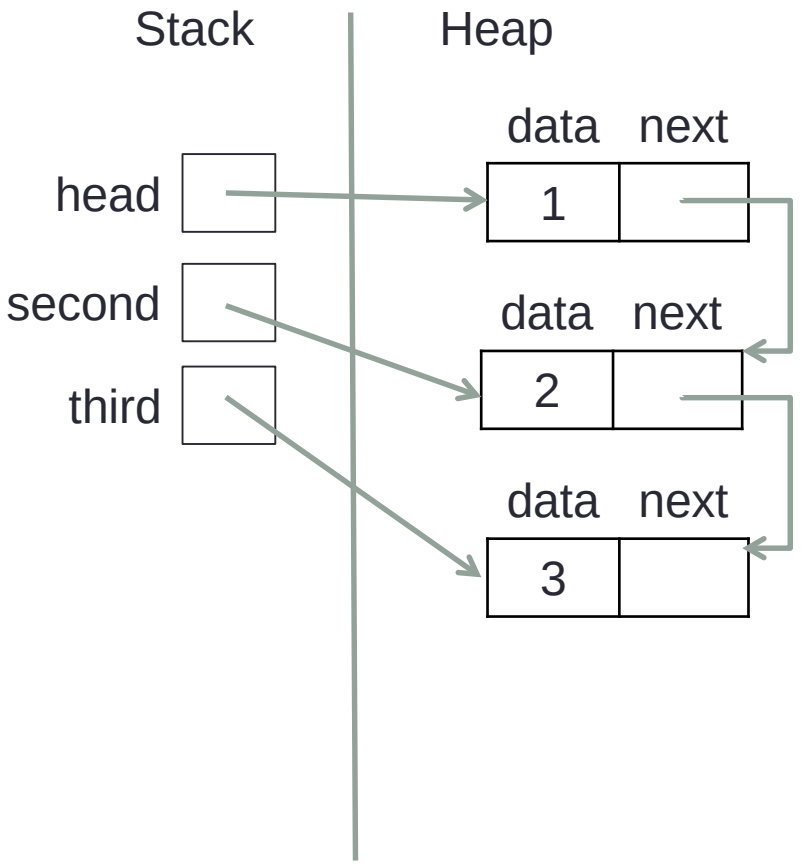
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



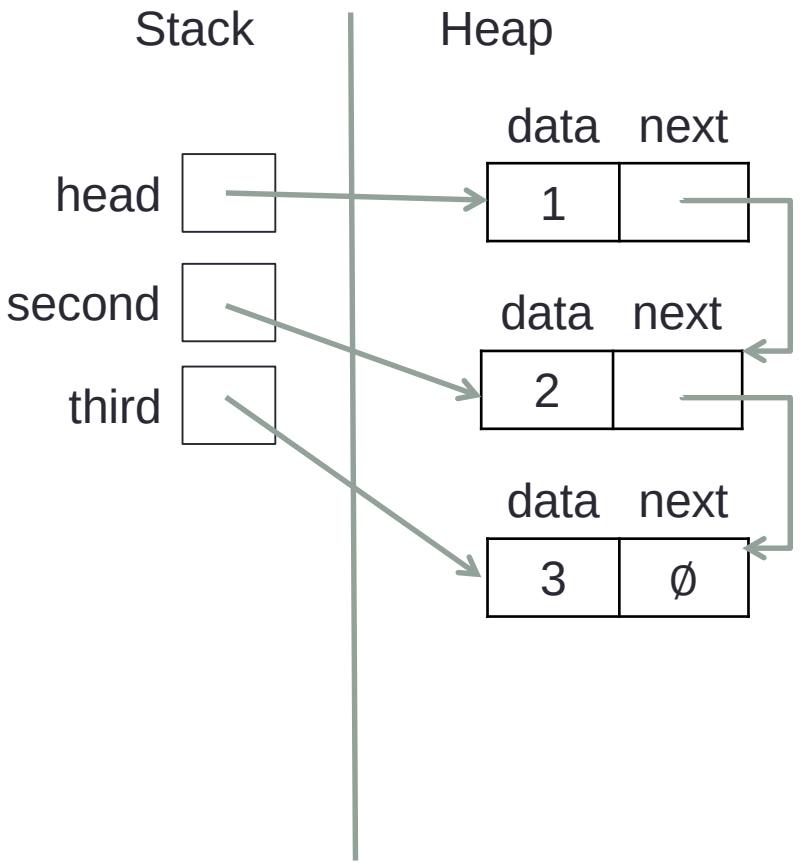
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```

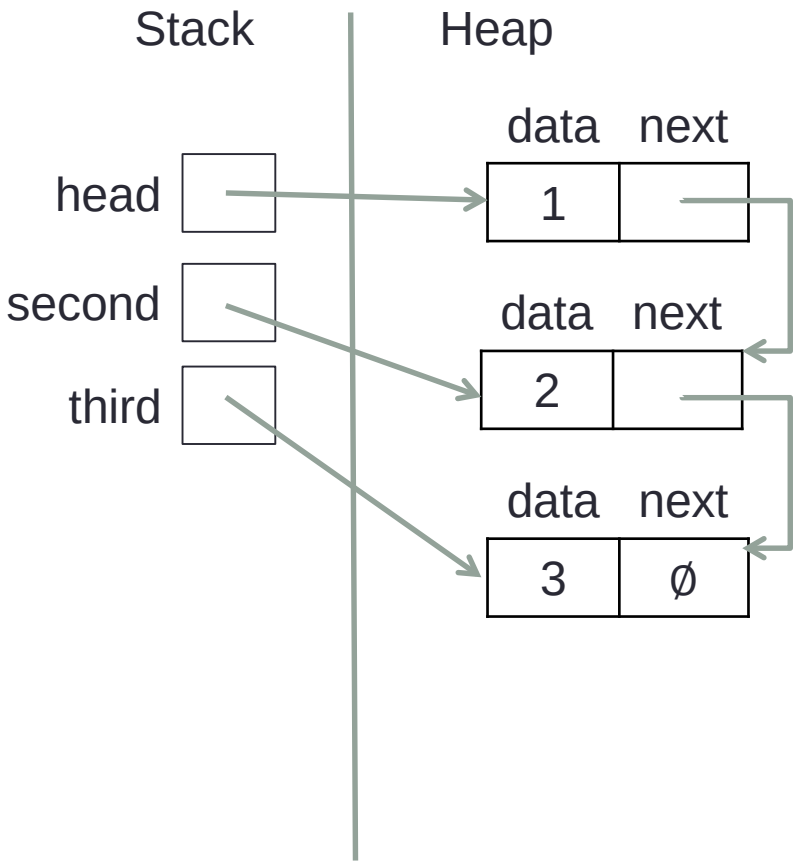


```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```



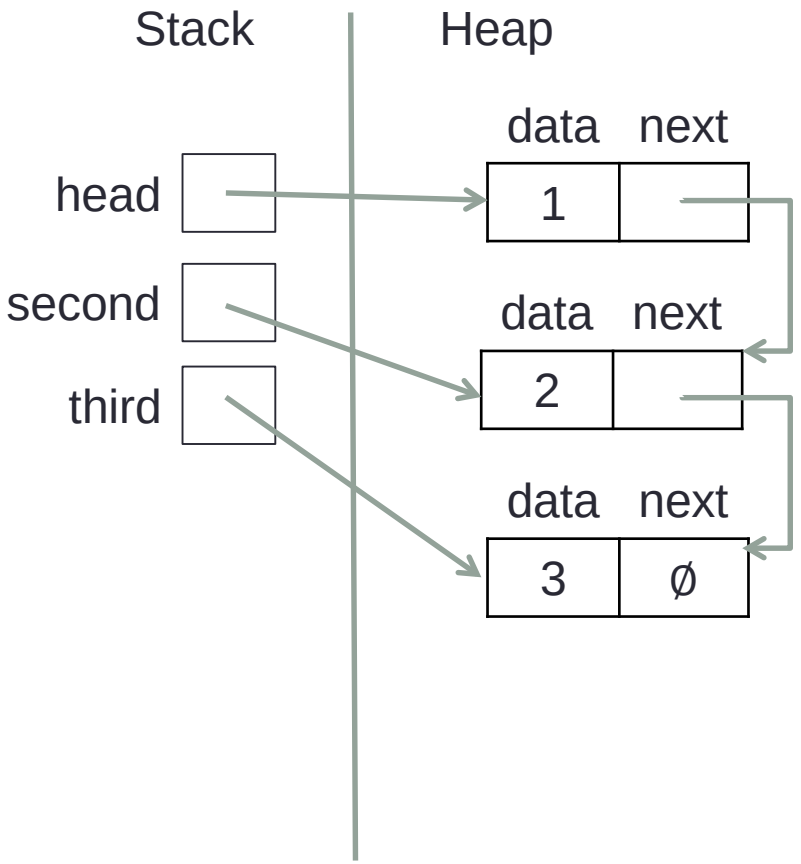
```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```

Note that in the end, we didn't need second or third (the pointers) except to do the 'linking'. We returned the first pointer (head).



```
Node* build_one_two_three() {
  Node *head;
  Node *second;
  Node *third;
  head = malloc(sizeof(Node));
  second = malloc(sizeof(Node));
  third = malloc(sizeof(Node));
  head->data = 1;
  head->next = second;
  second->data = 2;
  second->next = third;
  third->data = 3;
  third->next = NULL;
  return head;
}
```

Note that in the end, we didn't need second or third (the pointers) except to do the 'linking'. We returned the first pointer (head).



Finding the length of a list

- Our linked list will need many functions for it to be able to do things (say, delete a node), and so we can get information from our list (like its length).

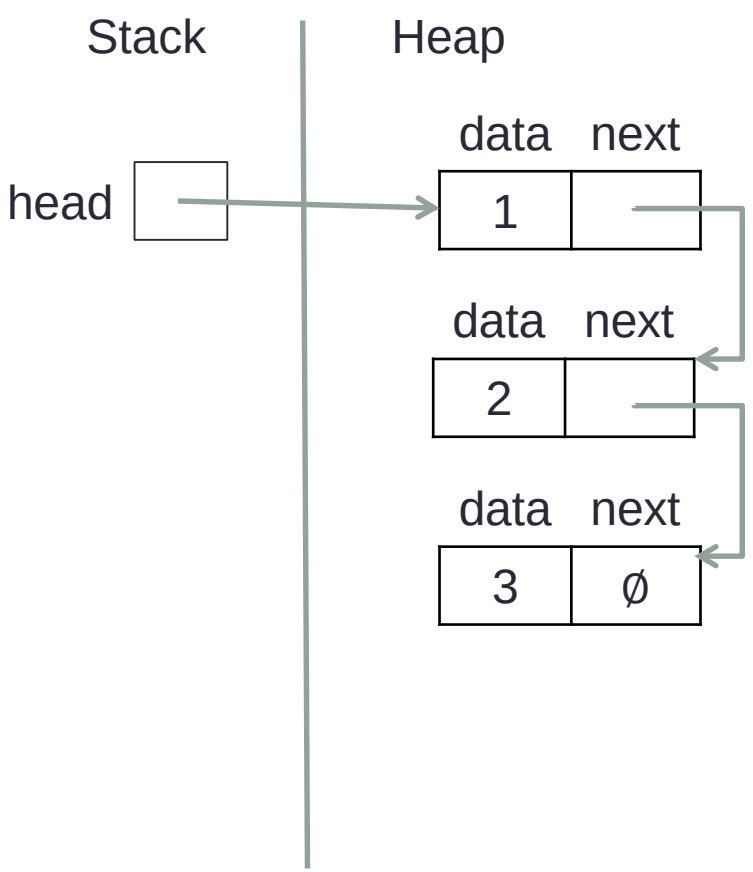
- The `length(...)` function computes the number of elements in the list (and returns it as an `int`).

- It is quite simple, but it will show us how we can iterate through the list to visit each of the nodes in it.
  - This is useful for many list operations.

- We will pass the head of this list to the function:

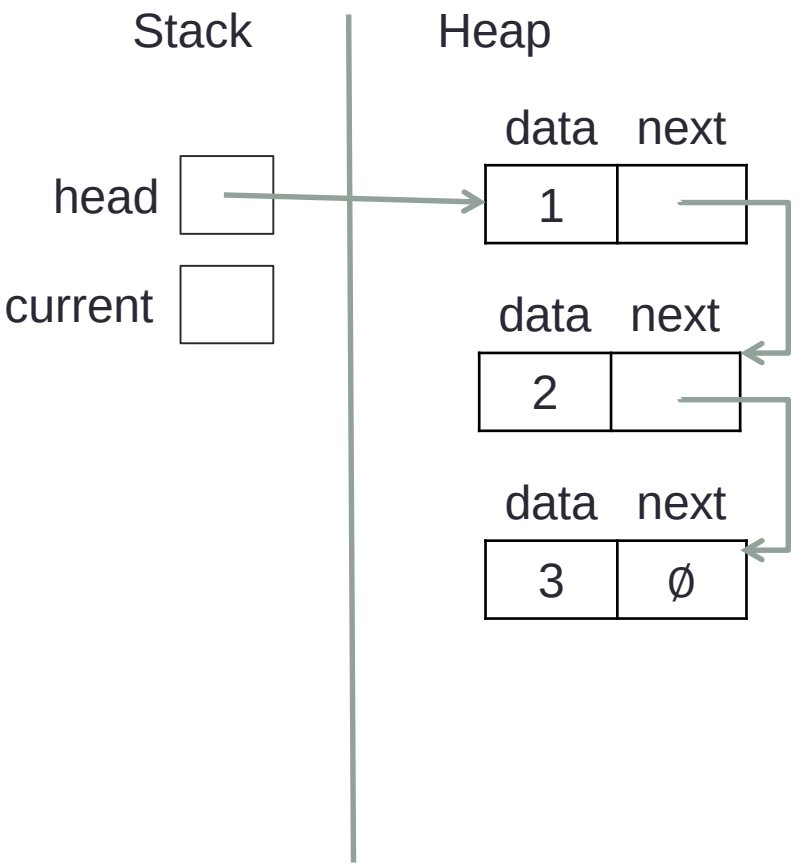
```
int length( Node *head );
```

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



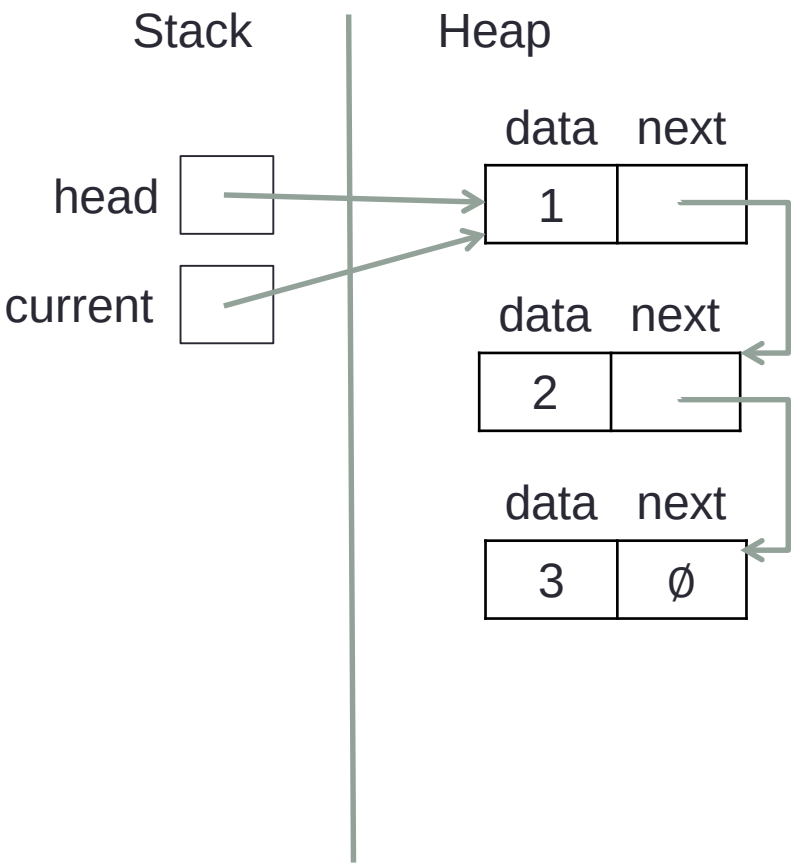
```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```

Create a pointer variable called `current` to keep track of where we are in the list.



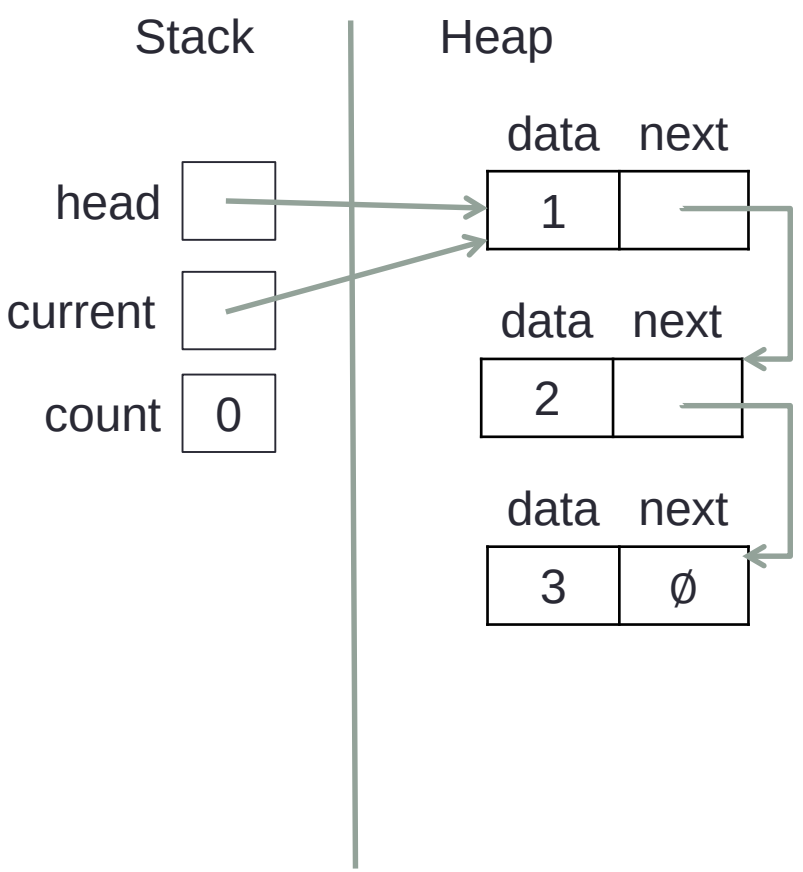
```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```

At the beginning, current points to the same node as head, to indicate that we are at the beginning of the list.



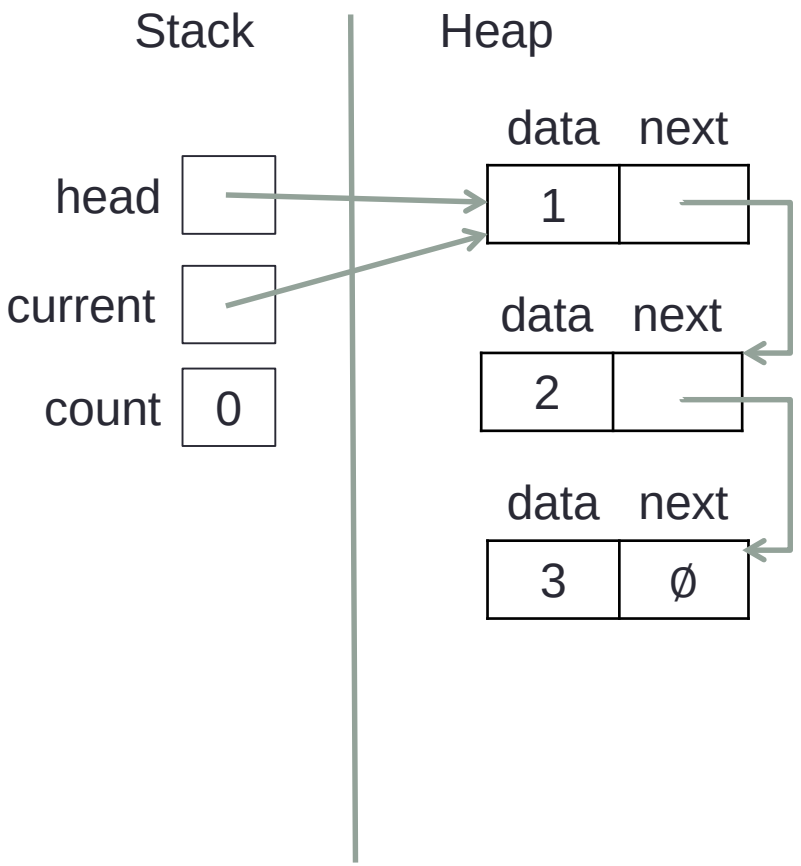
`count` will store the length of the list when we are finished. So far, we have not counted anything, so it begins at 0.

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



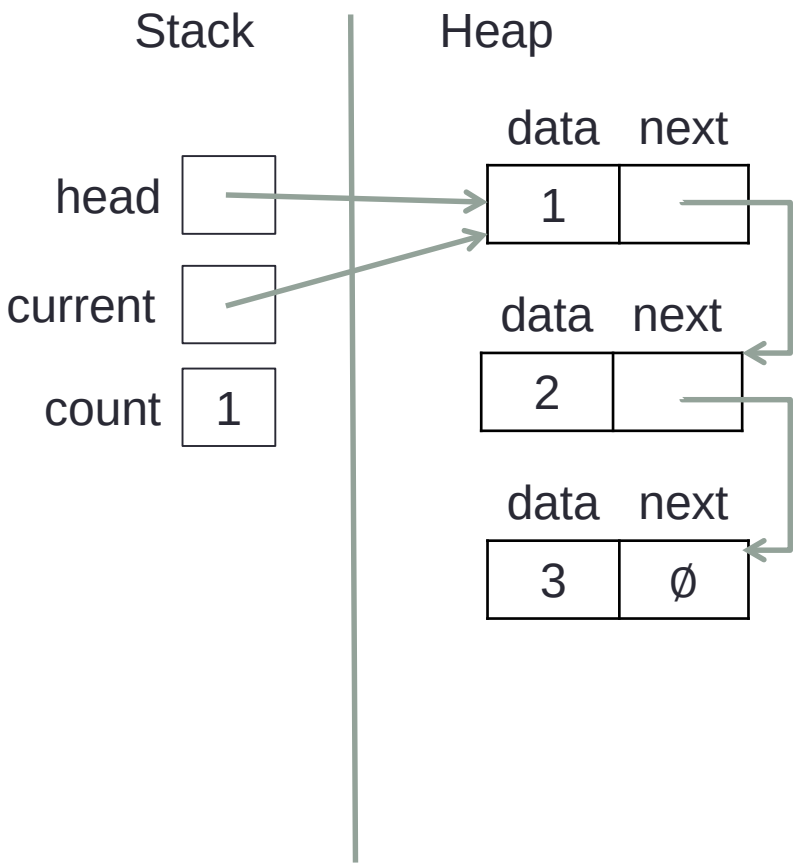
Remember that the last `next` node in the list is `NULL`. This is how we know to stop the loop.

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



Increase the `count` by 1 (this counts the first node).

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```

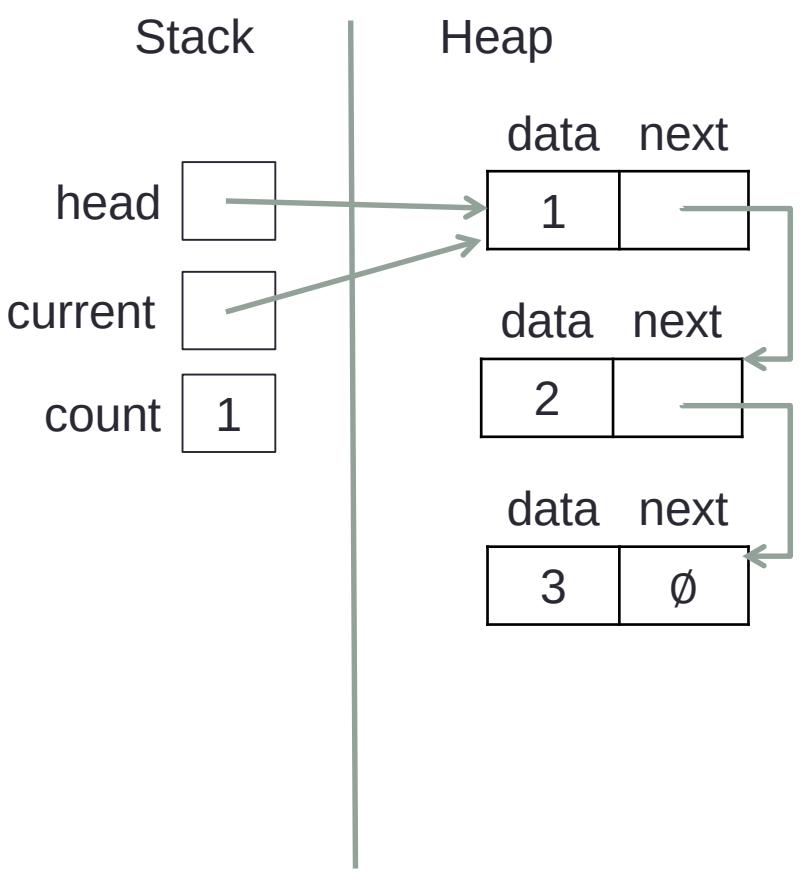


Change current to the next node.

Before this line, `current` points to the first node in the list.

So `current->next` is the second node.

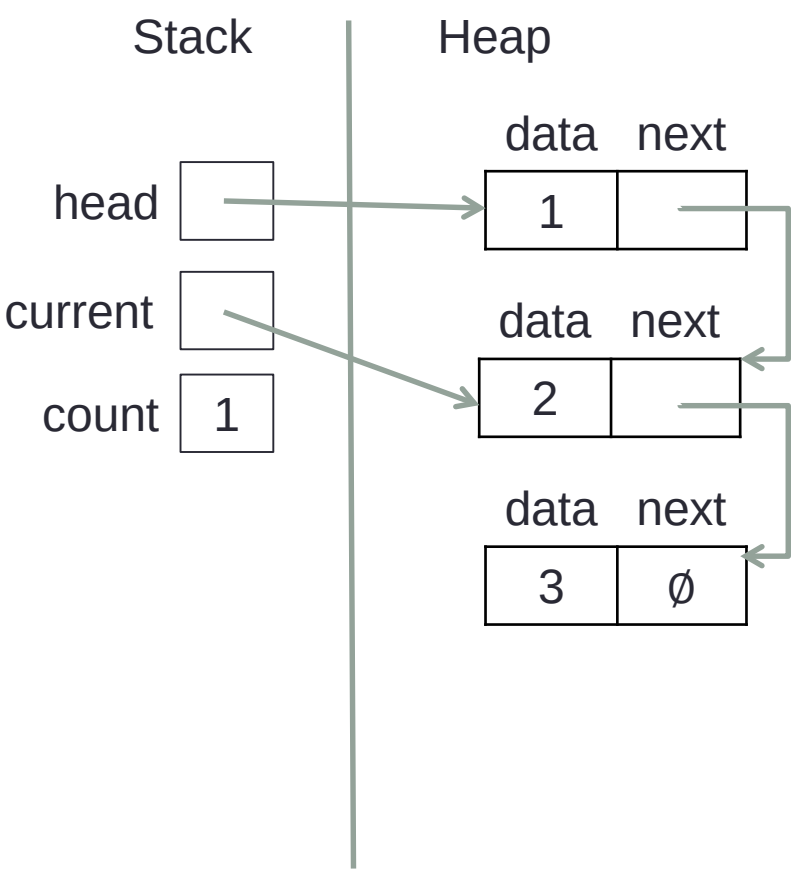
```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



Therefore, we store a pointer to the second node in `current`.

This is how we move through the list.

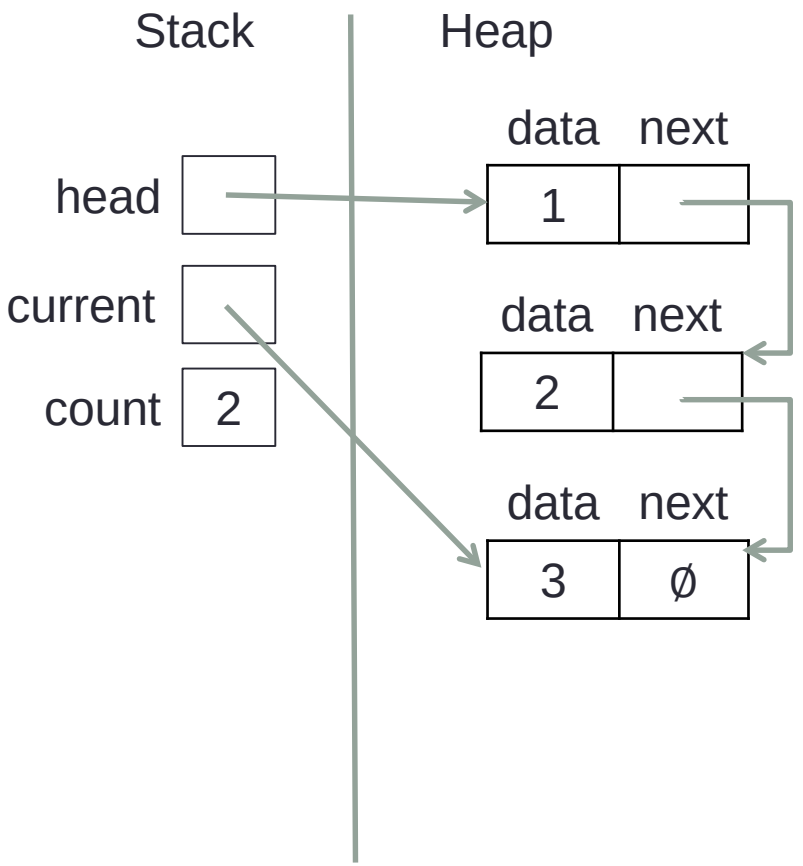
```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



The same thing happens during the second iteration:

- `current` is not `NULL`
- increase `count` (to count second node)
- set `current` to the next (third) node.

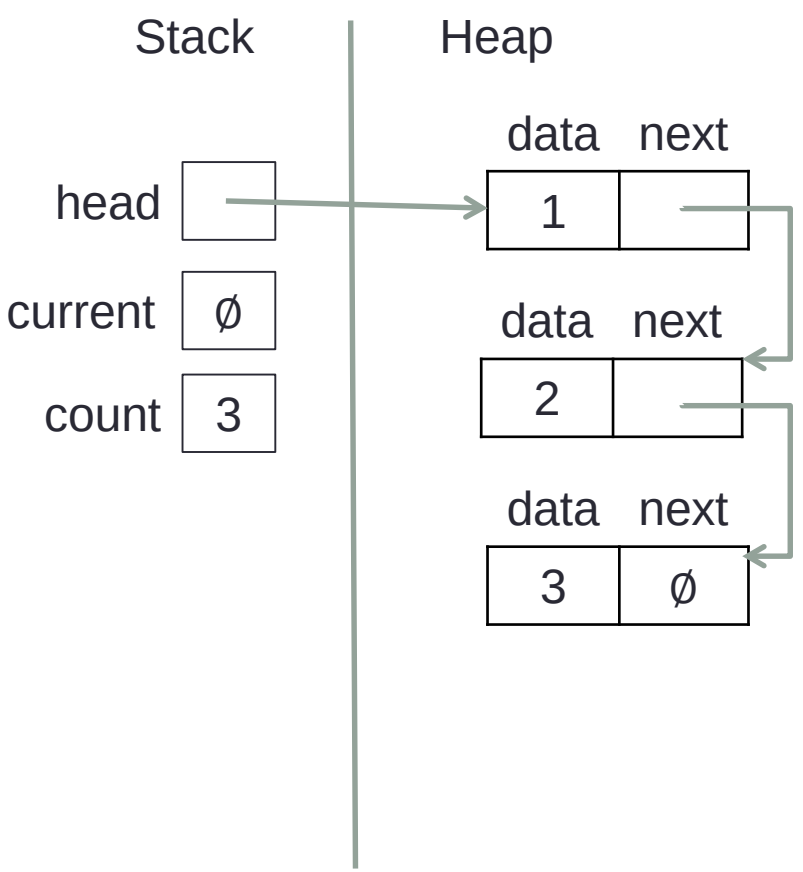
```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



The third iteration is slightly different:

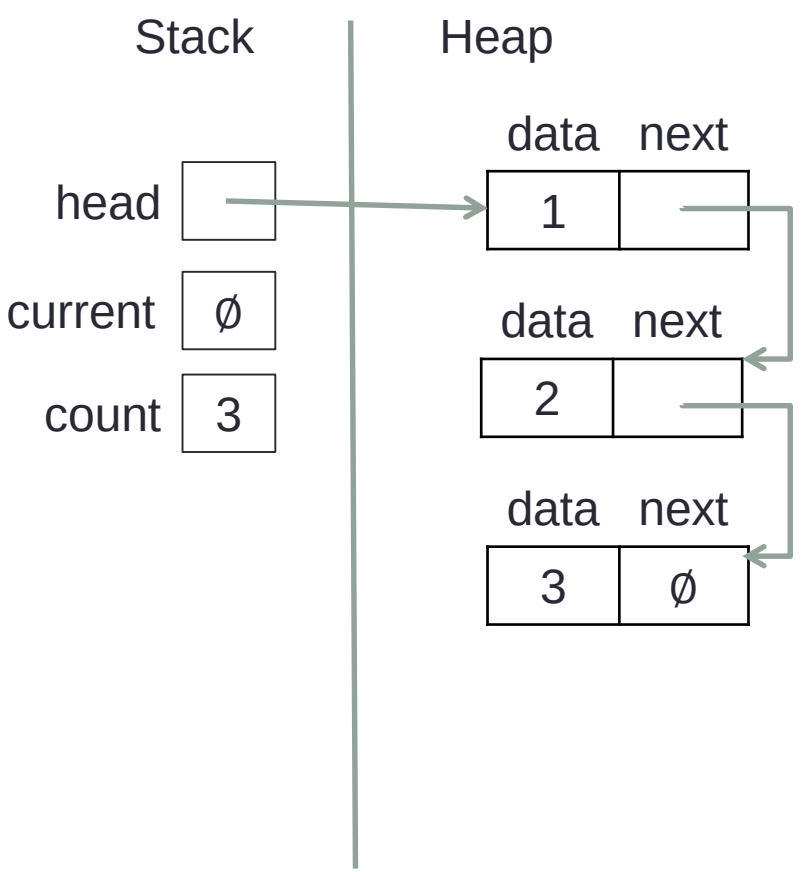
- `current` is not `NULL`
- increase `count` (to count third node)
- set `current` to the `next` field in the third node (this is NULL, as it is the last node)

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



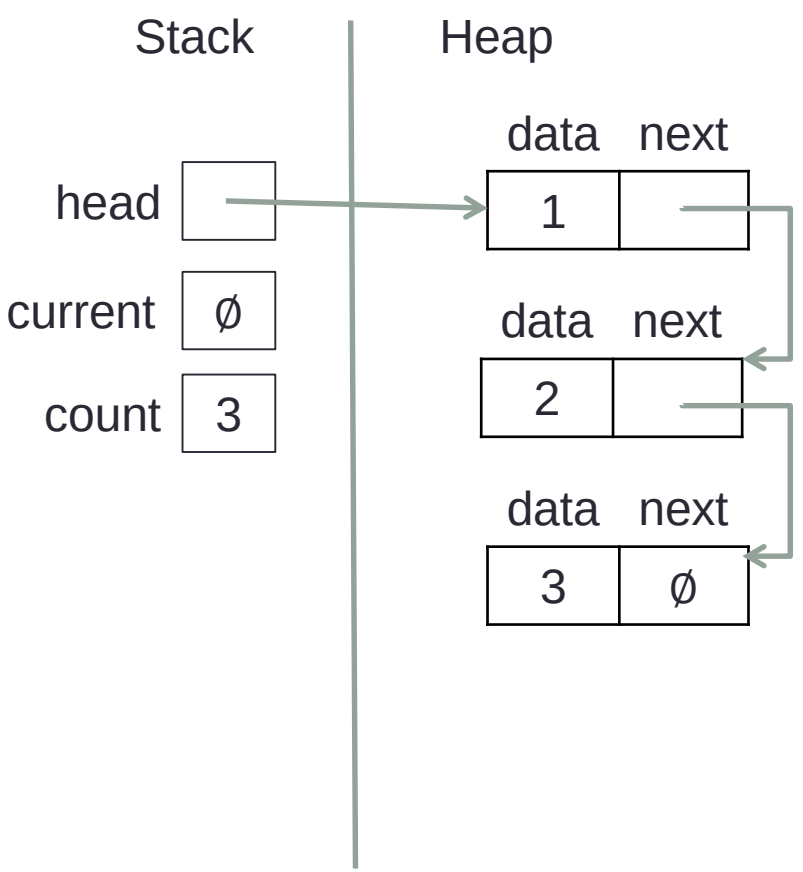
The next time the `while(...)` condition is checked, `current` is `NULL` so the loop exits.

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



The function returns a length of 3.

```
int length(Node *head) {
  Node *current;
  current = head;
  int count = 0;
  while( current != NULL ) {
    count++;
    current = current->next;
  }
  return count;
}
```



Using these together

```
int main() {
    Node* list_head;
    list_head = build_one_two_three();
    int len = length( list_head );
    printf( "Length is: %d\n", len );
}
```

File: length.c

Building a List

- The `build_one_two_three()` function is a good example of using pointers and structures to create a linked list.
- However, it is not very useful for making lists in general.
- We should make a function that can add a new element to the beginning of a list, no matter how long it is.
- We can use this function as many times as we like to build up a list.

Building a List

- Before we write any code, we should think about the steps we need to take to add to the beginning of a list:
  1. *Allocate:* Create a new node on the heap and set its data to be the element that we want to store.
  2. *Link Next:* Set the next pointer of the new node to the node that was at the beginning of the list.
  3. *Link Head:* Change the head pointer to point at the new node (so that it is now first in the list)
- To keep things simple, let's keep the head of the list in our main() function, and get our new function to return the new list head whenever something is added:

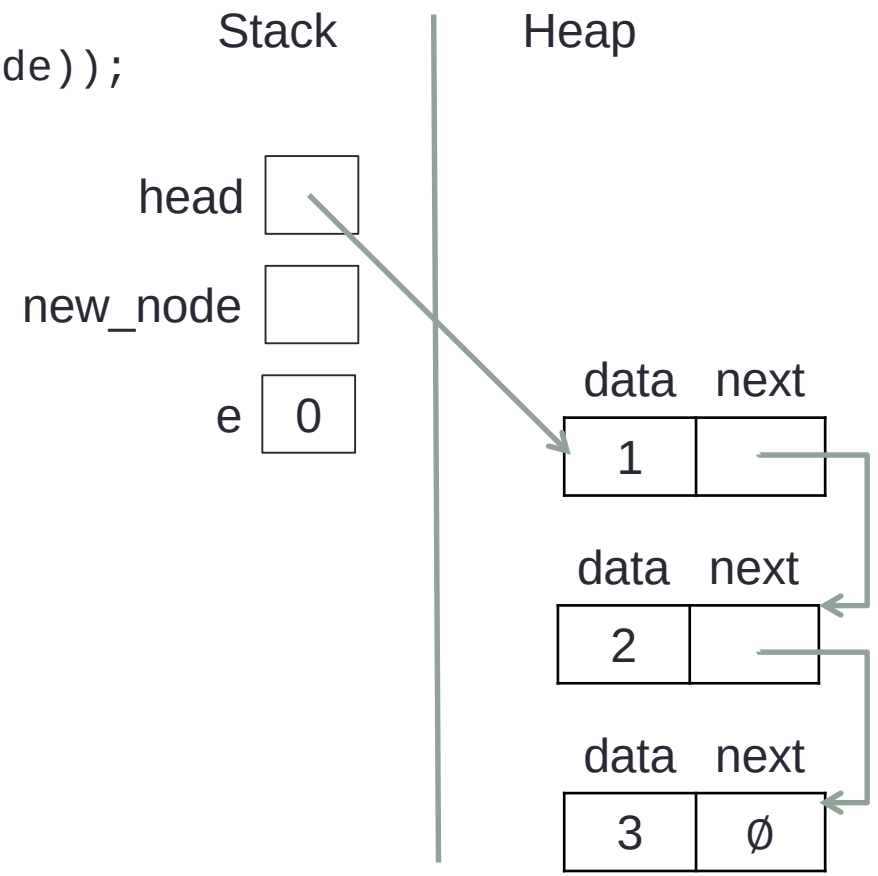
```
Node* add_element(Node *head, int e);
```

Building a List

- To test it, let's add 0 to the beginning of the list we already have, so we change our code to be:

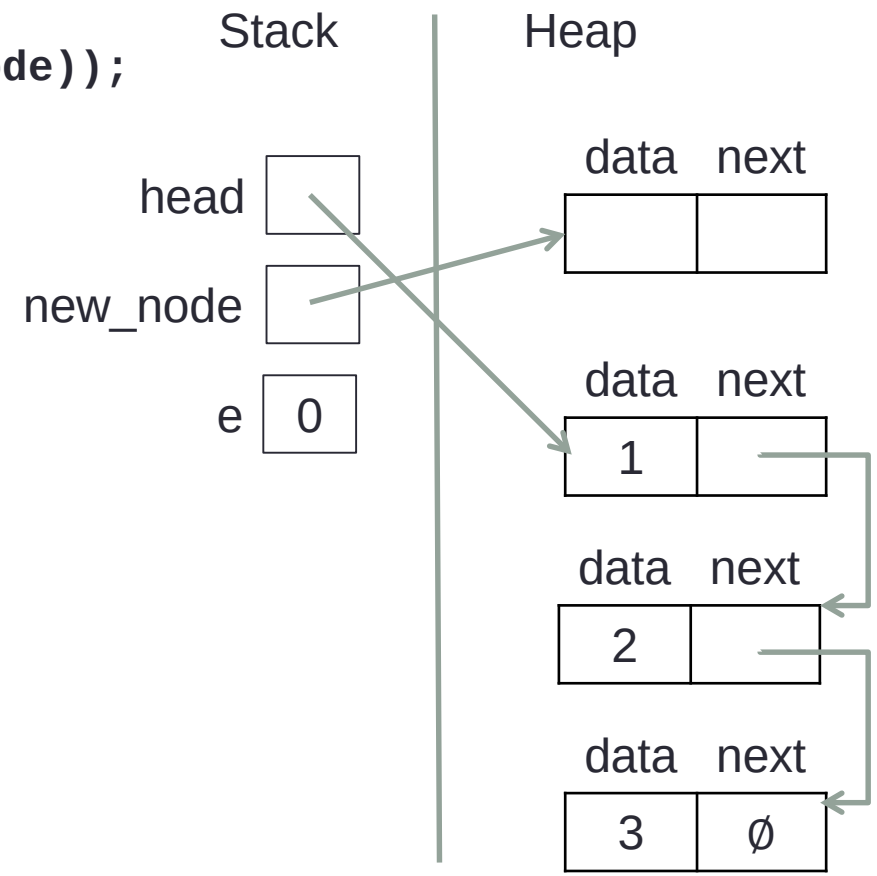
```
int main() {
    Node* list_head;
    list_head = build_one_two_three();
    list_head = add_element( list_head, 0 );
    int len = length( list_head );
    printf( "Length is: %d\n", len );
}
```

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



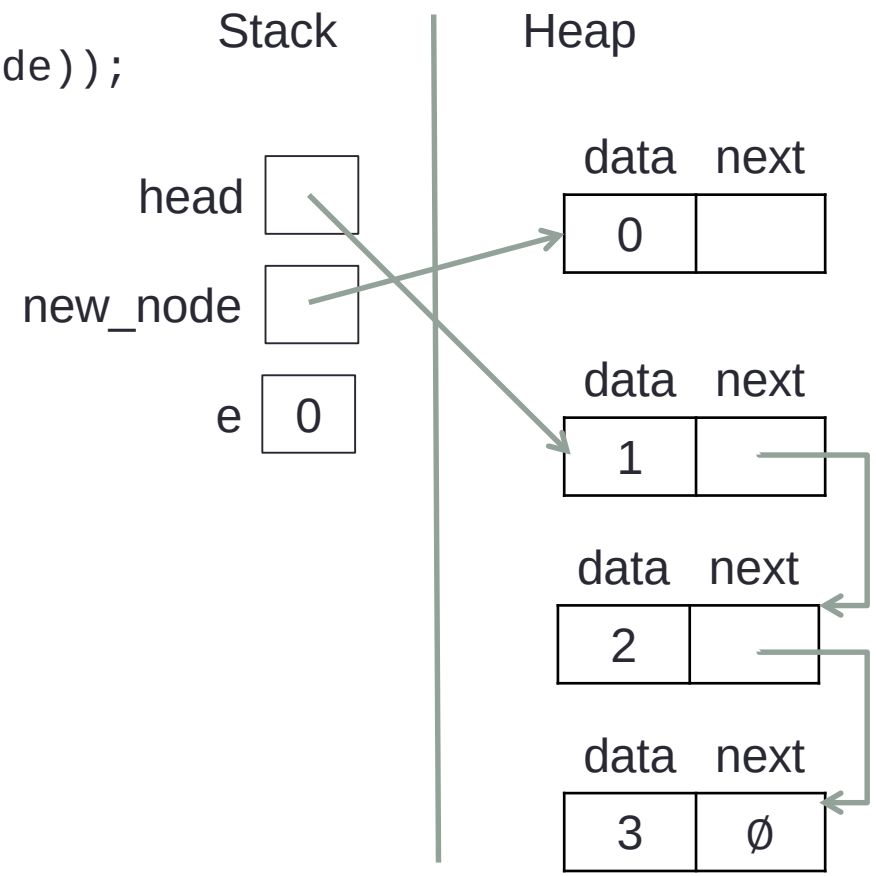
1. Allocate: Allocate a new node to store the new element.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



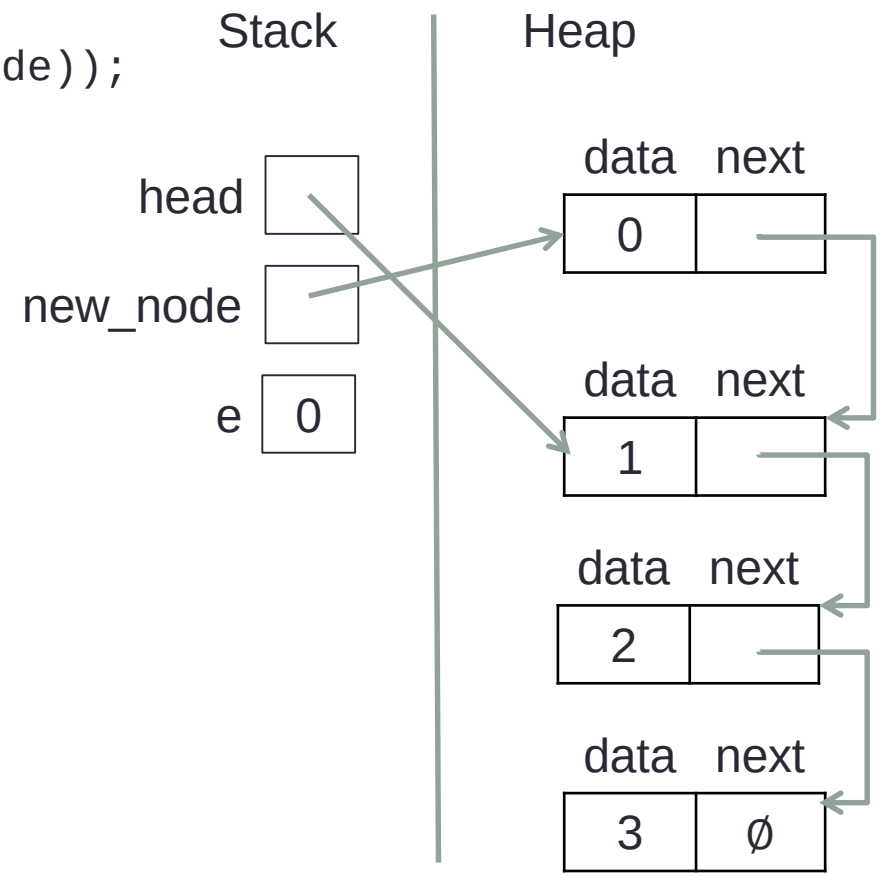
1. *Allocate:* Allocate a new node to store the new element.
   Set its data to be the element.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



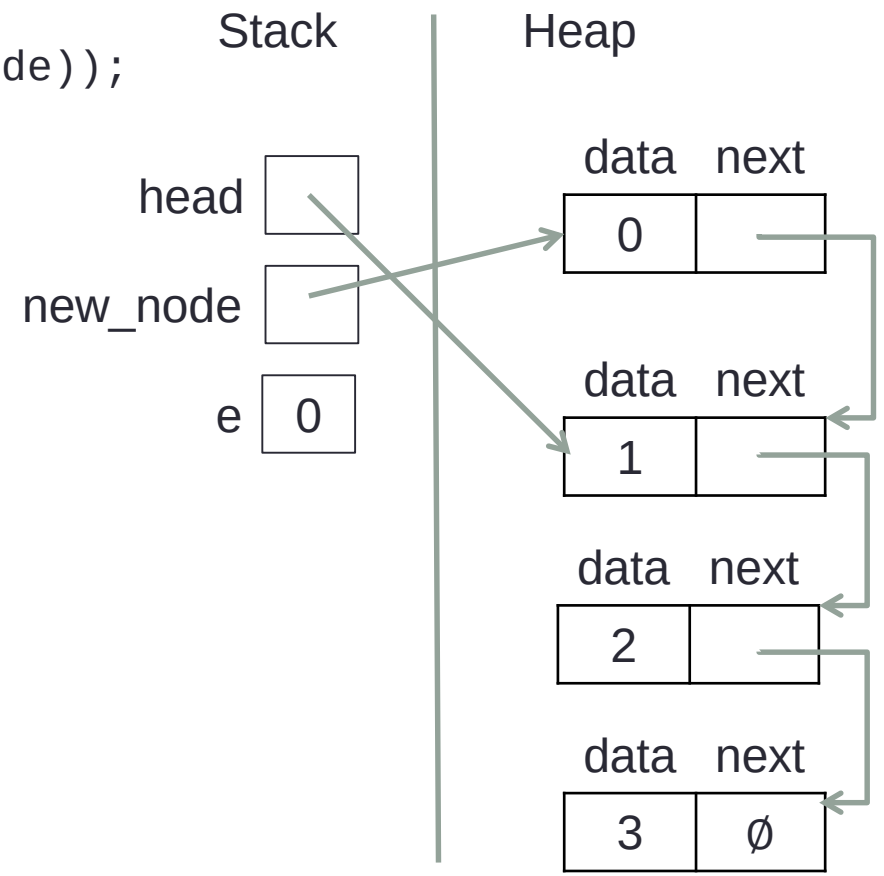
2. Link Next: Set the next pointer of the new node to be the old head of the list.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```

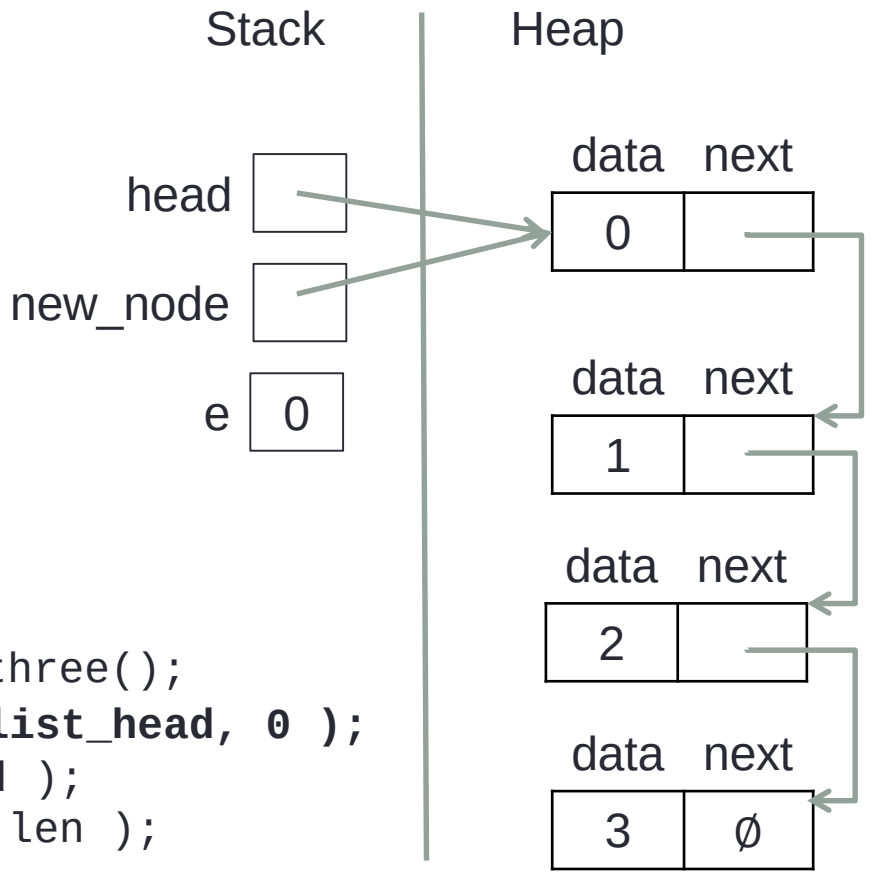


3. Link Head: Return the new head of the list, so it can be used in `main()`.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



Update Head: The new node is set to be the new head by `main()` when this function returns.




```
int main() {
    Node* list_head;
    list_head = build_one_two_three();
    list_head = add_element( list_head, 0 );
    int len = length( list_head );
    printf( "Length is: %d\n", len );
}
```

Building the List

- Now that we have a function to add an element to a list, we don't need `build_one_two_three()` anymore.
- We should be able to use our `add_element(...)` function to add **all** the elements to the list.
- **BUT:** We must first think about the beginning: how can we represent an empty list?
  - **Answer:** If there are no nodes in the list, then the head must be NULL.

Building the List

- Now our program becomes:

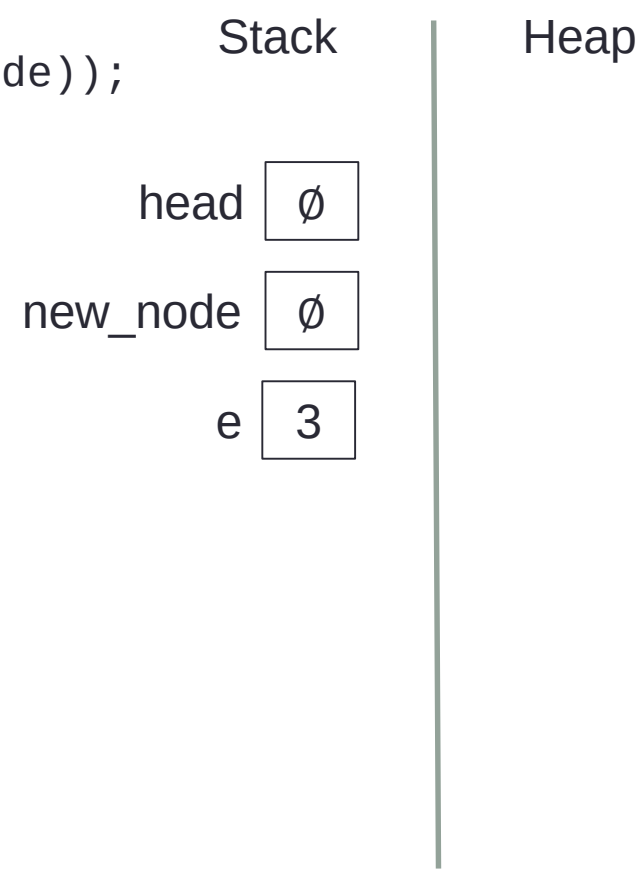
```
int main() {
    Node* list_head;
    list_head = NULL;
    list_head = add_element( list_head, 3 );
    list_head = add_element( list_head, 2 );
    list_head = add_element( list_head, 1 );
    list_head = add_element( list_head, 0 );

    int len = length( list_head );
    printf( "Length is: %d\n", len );
}
```

File: build.c

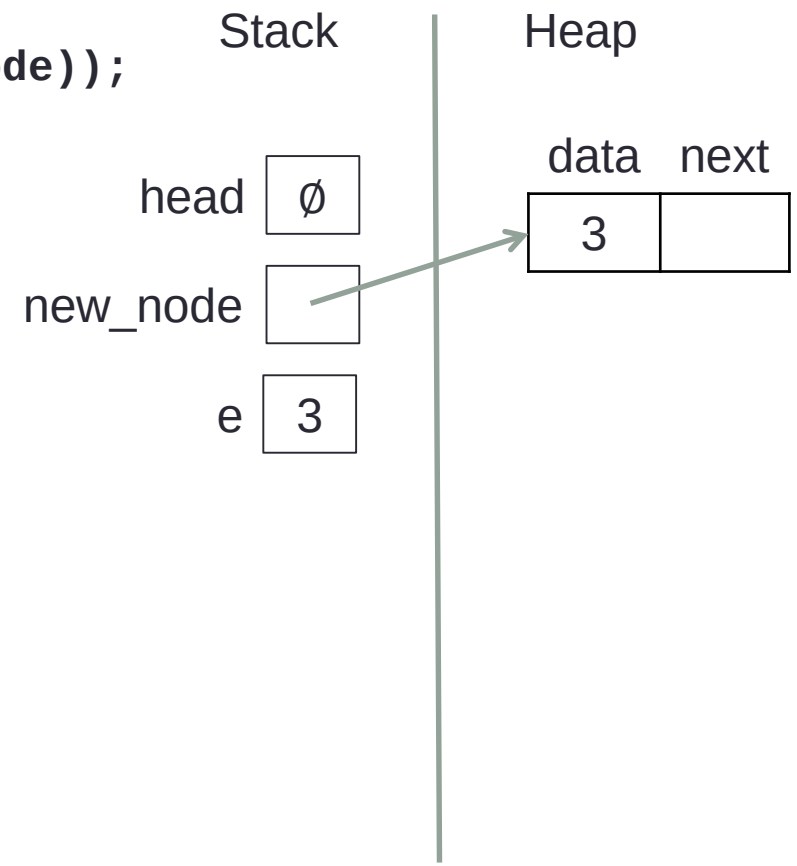
BUT: Are we *sure* this works for an empty list?

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



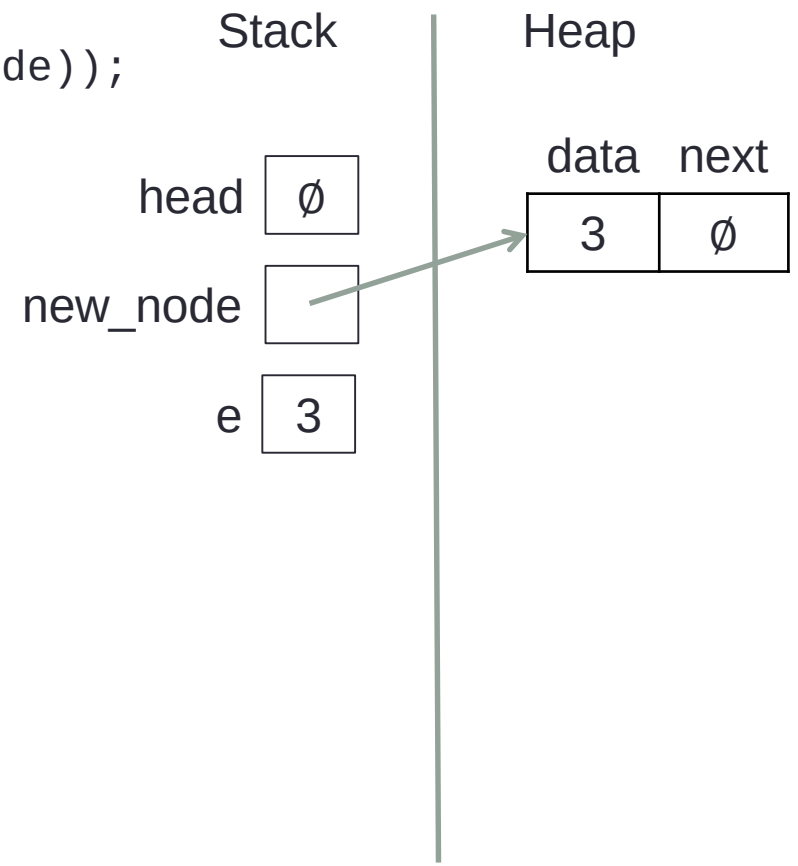
1. Allocate: Create a new node to store the element, and store the element in its data field.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```

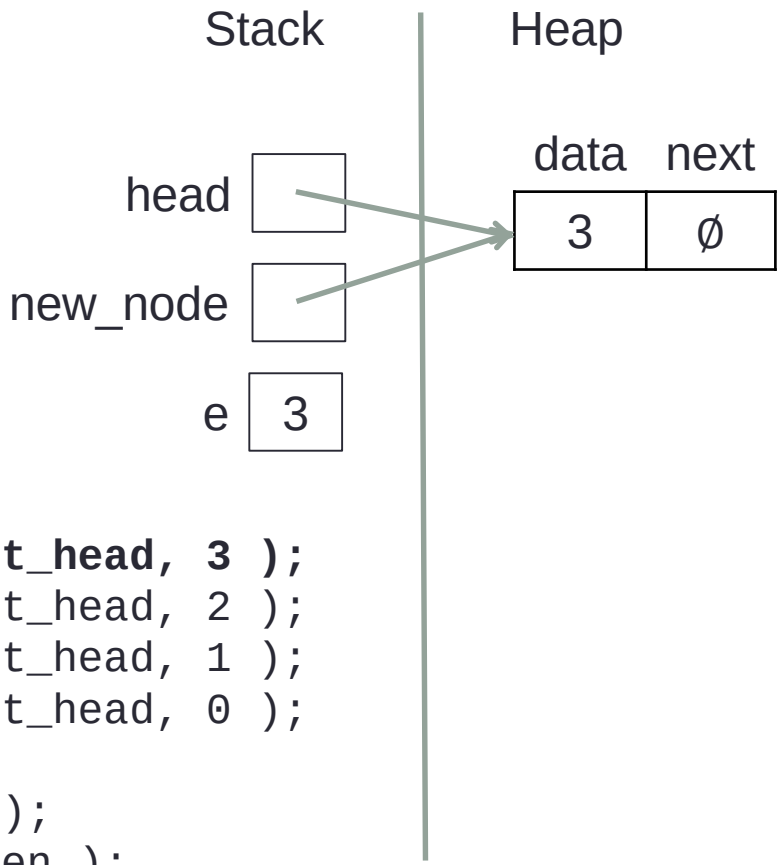


2. Link Next: Set the next pointer of the new node to be the old head of the list.

```
Node* add_element(Node *head, int e) {
   Node *new_node;
   new_node = malloc(sizeof(Node));
   new_node->data = e;
   new_node->next = head;
   return new_node;
}
```



Update Head: The new node is set to be the new head by `main()` when this function returns.




```
int main() {
   Node* list_head;
   list_head = NULL;
   list_head = add_element( list_head, 3 );
   list_head = add_element( list_head, 2 );
   list_head = add_element( list_head, 1 );
   list_head = add_element( list_head, 0 );

   int len = length( list_head );
   printf( "Length is: %d\n", len );
}
```

Finally, printing...

- Finally, let's write a function to iterate through the list and print its elements.

- We would like it to print nicely like this:

{0, 1, 2, 3, 4}

- It should take the list's head node as a parameter, as usual:

```
void print_list( Node *head );
```

Printing a List

Try drawing a memory diagram to see how this one works.

```
void print_list( Node *head ) {
   Node *current;
   current = head;
   while( current != NULL ) {
      if ( current == head ) {
         printf( "{ %d", current->data );
      }
      else if ( current->next != NULL ) {
         printf( ", %d", current->data );
      }
      else {
         printf( ", %d }\n", current->data );
      }
      current = current->next;
   }
}
```

file: print.c

Our list so far…

- Our list is not complete, there are still things we can't do with it, like:
  - Get the value at a given node
  - Adding nodes anywhere but at the beginning
  - Deleting nodes
  - … there are many more.

- We'll explore some of these in the lab.