Linked lists

Raul P. Pelaez

October 22, 2025

Contents

1 Introduction **1**

2 Basic concepts **1**

3 Implementation **2**
3.1 Type definitions 2
3.2 Basic operations 4
3.2.1 Creating a linked list 4
3.2.2 Inserting a node 4
3.2.3 Traversing the list 6
3.2.4 Deleting a node 7
3.2.5 Freeing the list 8

4 Exercises **9**
4.1 Linked list 9
4.2 Doubly linked list 10

1 Introduction

Today, we will learn about linked lists in C. Linked lists are a fundamental data structure that allows us to store and manage collections of data efficiently. Unlike arrays, linked lists do not require contiguous memory allocation, making them more flexible for dynamic data storage. You would use linked lists when you need to frequently insert or delete elements, as these operations can be performed in constant time ($O(1)$). On the contrary, adding or removing an element from an array requires shifting elements, and perhaps even reallocating memory, which is in general an $O(n)$ operation.

On the other hand, linked lists have some downsides. They require more memory per element due to the storage of pointers, and accessing elements is slower than in arrays because it requires traversing the list from the head to the desired node. In particular, accessing an element is a linear time operation ($O(n)$) in linked lists, compared to constant time ($O(1)$) in arrays (just dereferencing a pointer). Additionally, each element in a linked list, by its nature, is allocated separated in memory from the other elements, and on the heap (which is slower to access than the stack). This can lead to cache misses and reduced performance due to poor locality of reference.

A linked list is a really bad choice when you need random access to elements, or when memory usage (in terms of access speed per element) is a concern. In such cases, arrays (be it static or dynamic) are a better choice. Finally, the API for linked lists is more complex than for arrays, as we need to manage pointers and memory allocation manually. On the other hand, accessing an arbitrary element in an array is as simple as dereferencing a pointer.

2 Basic concepts

A linked list is a collection of nodes, where each node contains data and a pointer to the next node in the sequence. The first node is called the head, and the last node points to NULL, indicating the end

of the list. There are several types of linked lists, including singly linked lists, doubly linked lists, and circular linked lists. In this lesson, we will focus on singly linked lists.

Here is a simple representation of a singly linked list:

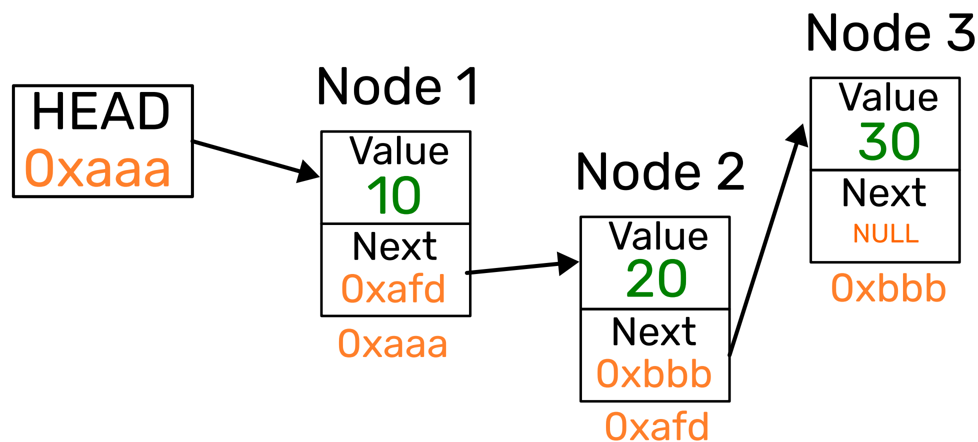


Figure 1: A linked list that stores integer values. Each box represents a node, which holds two pieces of information: the integer value and a pointer to the next node in the list. The last node points to NULL, indicating the end of the list. Below each node, we show the memory address where that node is stored. The head pointer points to the first node in the list. The `next` pointer of each node points to the memory address of the subsequent node, allowing traversal through the list.

Advanced

A doubly linked list
A doubly linked list is really similar to a singly linked list, but each node contains two pointers: one pointing to the next node and another pointing to the previous node. This allows for traversal in both directions, making certain operations more efficient at the cost of increased memory usage per node and and a more complex implementation.
On the other hand, a circular linked list is a variation where the last node points back to the first node, creating a circular structure. This can be useful for applications that require continuous traversal of the list without needing to check for the end.

3 Implementation

There are countless ways to implement a linked list in C, but we will focus on a simple implementation that covers the basic operations: creation, insertion, deletion, and traversal.

3.1 Type definitions

We will start by defining the structure of a node in the linked list:

```c
// linked_list.h
#pragma once

typedef struct{
  int value;
} Data;

struct Node;

struct Node{
  Data data;
  struct Node* next;
```

```c
};

typedef struct Node Node;

typedef struct {
  Node* head;
} LinkedList;
```

There are some mysterious contraptions in this code, lets break it down. In this snippet, we are defining three structures, `Data`, `Node` and `LinkedList`. The `Data` structure is a simple structure that holds an integer value, we define it so that it is easy to change what the list stores later on. The `Node` structure represents a single node in the linked list, containing a `Data` object and a pointer to the next node in the list. The `LinkedList` structure contains a pointer to the head of the list. We technically do not need the `LinkedList` structure, which is just a pointer to the first `Node` (AKA the `HEAD`), but it makes it easier to manage the list as a whole. For instance, we can write functions that take something of type `LinkedList` as input, instead of a pointer to a `Node`, which might or might not be the `HEAD`.

The surprising bit in this code is perhaps the fact that the `Node` structure contains a pointer to `Node`. We are referring to a type inside its own definition! This is possible because we are using a forward declaration of the `Node` structure. The line `struct Node;` tells the compiler that there is a structure called `Node`, but we will define it later. This allows us to use pointers to `Node` in the definition of the `Node` structure itself. We cannot use the `typedef` trick when doing a forward declaration, so we have to use the full `struct Node*` syntax.

> **Info**
>
> **Forward declaration of structures**
> We can "forward declare" a function in C, for instance:
>
> ```c
> int add(int a, int b); // Forward declaration
> int main() {
>   int result = add(2, 3); // Call the function
>   return 0;
> }
> int add(int a, int b) { // Function definition
>   return a + b;
> }
> ```
>
> The forward declaration tells the compiler that there is a function called `add` that takes two integers as input and returns an integer. This allows us to call the function in `main` before we define it. The actual definition of the function comes later. We use this technique to, for instance, separate declaration and definition of functions in header files and source files.
> Turns out we can do the same thing with structures. We can tell the compiler that, at some point, a structure called `something` will be defined by writing:
>
> ```c
> struct something; // Forward declaration
> ```
>
> As long as we then define it *somewhere*, the compiler will be happy.

Finally, I am defining an alias for the `struct Node` type, so that we can just write `Node` instead of `struct Node`. This is just a convenience, and not strictly necessary.

Advanced

An even more confusing way to do define the Node

We can actually skip the forward declaration and the alias by defining `Node` as follows:

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

Notice that we are writing `Node` twice, once in the first line, and once in the last line. The first `Node` is the name of the structure, while the second `Node` is the name of the type we are defining. This works because the compiler knows that we are defining a new type, so it can differentiate between the two uses of `Node`.

It is equivalent to the forward declaration we did before. But perhaps much more mind bending.

3.2 Basic operations

Thus far, the only thing we can do with our linked list is to create it. We will now implement some basic operations that we can perform on the linked list.

3.2.1 Creating a linked list

To create a linked list, we need to allocate memory for the `LinkedList` structure and initialize the `head` pointer to NULL, indicating that the list is empty. Let us assume that the basic type definitions are in a file called `linked_list.h`. We can implement the `create_linked_list` function as follows:

```
// appended to the previous linked_list.h
#include <stdlib.h>
/**
 *@brief Create a new empty linked list
 *@return A pointer to the newly created linked list
*/
LinkedList* linked_list_create() {
  LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
  list->head = NULL;
  return list;
}
```

Why store the `LinkedList` in the heap instead of just declaring it as a variable like `LinkedList list;`? We could do that, but as you will see in a moment, we will have no other choice than to store the nodes in the heap. If we want to be consistent, we should also store the `LinkedList` in the heap.

At this point, we have a function that creates an empty linked list. We can use it as follows:

```
#include "linked_list.h"

int main(){
  LinkedList* list = linked_list_create();
  printf("HEAD points to: %p\n", list->head);
  free(list);
  return 0;
}
```

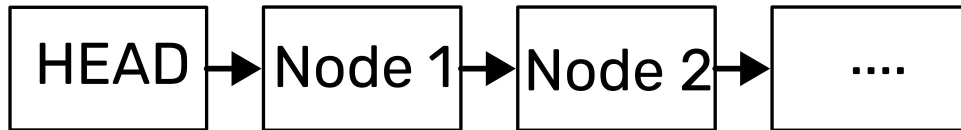
Not very exciting, but we have created an empty linked list. The `head` pointer is NULL, indicating that the list is empty.

3.2.2 Inserting a node

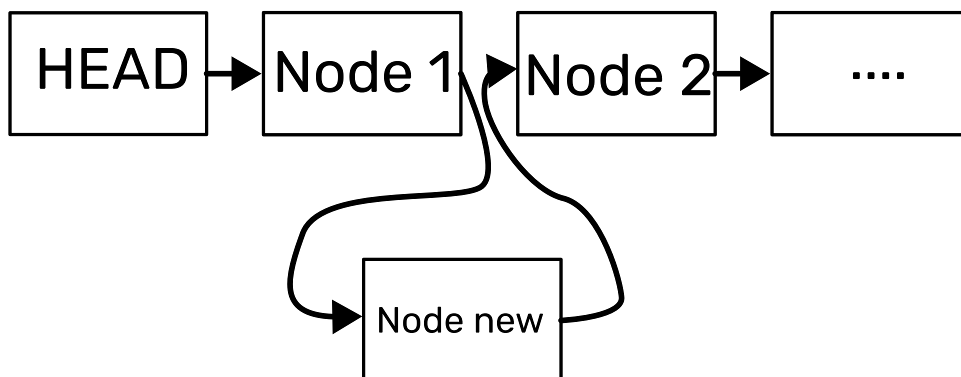
To insert a node into the linked list, we need to create a new node, set its data, and update the pointers accordingly. We will implement a function called `insert` that inserts a new node after a given node. If the given node is NULL, we will insert the new node at the head of the list.

Inserting a node anywhere in the list is a constant time operation ($O(1)$), as we only need to update a few pointers. In particular, we need to create the new node, then, to insert it after a given node, we need to update the `next` pointer of the new node to point to the `next` pointer of the given node, and then update the `next` pointer of the given node to point to the new node. If we are inserting at the head, we need to update the `next` pointer of the new node to point to the current head, and then update the head pointer to point to the new node.

I know its a mouthful, but it is easier to understand with a diagram: We start with a list that looks like this:



Say that we want to insert a new node between 1 and 2, the `instert` function is going to create a `Node` called "new" and place it like this:



There is a special case when we want to insert at an empty list, or at the head of a non-empty list. In this case, we want to insert the new node before the current head.

```c
// appended to the previous linked_list.h
/**
 *@brief Insert a new node after the given node
 *@param list The linked list to insert the node into
 *@param prev_node The node after which to insert the new node. If NULL, insert at the head.
 *@param data The data to store in the new node
 *@return 0 on success, -1 on failure
*/
int linked_list_insert(LinkedList* list, Node* prev_node, Data data) {
  Node* new_node = (Node*)malloc(sizeof(Node));
  if (!new_node) {
    return -1; // Memory allocation failed
  }
  new_node->data = data;
  Node* next = prev_node ? prev_node->next : list->head;
  new_node->next = next;
  if (prev_node) {
    prev_node->next = new_node;
  } else {
    list->head = new_node;
  }
  return 0;
}
```

Insight

Memory leaks?
We are writing a lot of `malloc`, but not a lot of `free`. It might seem as if we are leaking the `Node` in this function, but we are not. The new node is now part of the linked list, and we will free it when we free the entire list. We will see how to do that later. The point is that we are not loosing track of the newly created pointer, it is accessible from the `head` of the list.

Info

The ternary operator
The line `result = condition? value_if_true : value_if_false;` is using the ternary operator, which is a shorthand way of writing an if-else statement. It evaluates the `condition`, and if it is true, it assigns `value_if_true` to `result`; otherwise, it assigns `value_if_false` to `result`.
Example

```
int a = 10;
int b = 20;
int max = (a > b) ? a : b; // max will be 20
```

The ternary operator is often used for simple conditional assignments like this one.

The insert function can be used as follows:

```
#include "linked_list.h"
#include <stdio.h>
int main(){
  LinkedList* list = create_linked_list();
  Data data1 = {1};
  Data data2 = {2};
  Data data3 = {3};
  linked_list_insert(list, NULL, data1); // Insert 1 at the head
  linked_list_insert(list, list->head, data2); // Insert 2 after 1
  linked_list_insert(list, list->head, data3); // Insert 3 after 1
  // List is now 1 -> 3 -> 2
  free(list);
  return 0;
}
```

3.2.3 Traversing the list

To traverse the linked list, we need to start from the head and follow the `next` pointers until we reach the end of the list (when the `next` pointer is NULL). We will implement a function called `print_list` that prints the values of the nodes in the list.

```
// appended to the previous linked_list.h
#include <stdio.h>
/**
 *@brief Print the values in the linked list
 *@param list The linked list to print
*/
void linked_list_print(LinkedList* list) {
  Node* current = list->head;
  while (current) {
    printf("%d -> ", current->data.value);
    current = current->next;
  }
  printf("NULL\n");
}
```

The `print_list` function can be used as follows:

```c
#include "linked_list.h"
#include <stdio.h>
int main(){
  LinkedList* list = linked_list_create();
  Data data1 = {1};
  Data data2 = {2};
  Data data3 = {3};
  linked_list_insert(list, NULL, data1); // Insert 1 at the head
  linked_list_insert(list, list->head, data2); // Insert 2 after 1
  linked_list_insert(list, list->head, data3); // Insert 3 after 1
  // List is now 1 -> 3 -> 2
  linked_list_print(list);
  // We have a memory leak now, but we will fix that later
  return 0;
}
```

Output

```
1 -> 3 -> 2 -> NULL
```

3.2.4 Deleting a node

To delete a node from the linked list, we need to update the pointers accordingly and free the memory allocated for the node. We will implement a function called `delete_node` that deletes a node. Deleting a node is also a constant time operation ($O(1)$), as we only need to update a few pointers. In essence, we eliminate the target node and point the previous node to the next node of the target node. If we are deleting the head, we need to update the head pointer to point to the next node of the current head.

```c
// appended to the previous linked_list.h
#include <stdlib.h>
/**
 *@brief Delete a node from the linked list
 *@param list The linked list to delete the node from
 *@param target The node to delete
 *@return 0 on success, -1 on failure
*/
int linked_list_delete(LinkedList* list, Node* target) {
  if (!list->head || !target) {
    return -1; // List is empty or target is NULL
  }
  if (list->head == target) {
    list->head = target->next;
    free(target);
    return 0;
  }
  Node* current = list->head;
  while (current->next && current->next != target) {
    current = current->next;
  }
  if (!current->next) {
    return -1; // Target not found
  }
  current->next = target->next;
  free(target);
  return 0;
}
```

Again, the head node is a special case, as we need to update the head pointer. If the target node is not the head, we need to traverse the list to find the node before the target node, and then update its `next` pointer to skip the target node. The `delete_node` function can be used as follows:

```c
#include "linked_list.h"
#include <stdio.h>
int main(){
  LinkedList* list = linked_list_create();
  Data data1 = {1};
  Data data2 = {2};
  Data data3 = {3};
  linked_list_insert(list, NULL, data1); // Insert 1 at the head
  linked_list_insert(list, list->head, data2); // Insert 2 after 1
  linked_list_insert(list, list->head, data3); // Insert 3 after 1
  // List is now 1 -> 3 -> 2
  linked_list_print(list);
  linked_list_delete(list, list->head->next); // Delete 3
  // List is now 1 -> 2
  linked_list_print(list);
  // We have a memory leak now, but we will fix that later
  return 0;
}
```

Output

```
1 -> 3 -> 2 -> NULL
1 -> 2 -> NULL
```

3.2.5 Freeing the list

The code above does contain a memory leak, as we are not freeing the list, which entails freeing all the nodes on top of the `LinkedList` structure itself. We will implement a function called `linked_list_free` that frees the entire linked list.

```c
// appended to the previous linked_list.h
/**
 *@brief Free the entire linked list
 *@param list A pointer to the linked list to free. After freeing, the pointer is set to NULL.
*/
void linked_list_free(LinkedList** list) {
  if (!list || !*list) {
    return; // List is already NULL
  }
  Node* current = (*list)->head;
  while (current) {
    Node* next = current->next;
    free(current);
    current = next;
  }
  free(*list);
  *list = NULL;
}
```

The `linked_list_free` function can be used as follows:

```c
#include "linked_list.h"
#include <stdio.h>
#include <assert.h>
int main(){
  LinkedList* list = linked_list_create();
  Data data1 = {1};
  Data data2 = {2};
  Data data3 = {3};
  linked_list_insert(list, NULL, data1); // Insert 1 at the head
  linked_list_insert(list, list->head, data2); // Insert 2 after 1
```

```c
    linked_list_insert(list, list->head, data3); // Insert 3 after 1
    // List is now 1 -> 3 -> 2
    linked_list_print(list);
    linked_list_delete(list, list->head->next); // Delete 3
    // List is now 1 -> 2
    linked_list_print(list);
    linked_list_free(&list); // Free the entire list
    assert(list == NULL); // list should be NULL after freeing
    printf("List freed successfully\n");
    return 0;
}
```

Output

```
1 -> 3 -> 2 -> NULL
1 -> 2 -> NULL
List freed successfully
```

4 Exercises

4.1 Linked list

Goal

Generic traversal
Traversing the linked list to operate on each data element requires writing a bunch of boilerplate, error-prone, code. A code that must be repeated for each operation we need to perform on the list. The `linked_list_print` function is an example of this. Instead, we would like to write a generic traversal function that takes an "operation" as input and applies it to each node in the list. Write such a function.
I provide here a skeleton of the function you need to implement:

```c
/**
 *@brief Traverse the linked list and apply the given operation to each node
 *@param list The linked list to traverse
 *@param op The operation to apply to each node,
 *          a function that takes a Data* as input and returns void
 */
void linked_list_traverse(LinkedList* list, /* Your operator type here */ op) {
    // Your code here
}
```

Try your function by implementing an operation that prints the value of each node, and passing it to the `linked_list_traverse` function. It should behave the same as the `linked_list_print` function.

hint

- One of the arguments of the function should be a function pointer that takes a `Data*` as input and returns void.

Milestone

Make the operator function a `typedef`, call it `LinkedListOp`.

Milestone

Make sure you have proper error handling in your function. e.g., check if the list is NULL.
Use unit testing for this. A test could be passing a `NULL` pointer to the `linked_list_traverse` function and checking that it handles the error gracefully (e.g., by returning without crashing).

Advanced milestone

Reduction
Implement an operator function that sums all the values in the linked list using the `linked_list_traverse` function.

hint

- This operator is a bit special, as it needs to maintain some state (the running sum) between calls. You will need to get a bit creative here.

Advanced milestone

No globals allowed
If your solution to the previous milestone used global variables, re-implement it without using any global state.

hint

- You might need to modify the signature of the operator function to accept an additional argument that holds the state (the running sum). But for an extra challenge, try to do it without modifying the signature of the operator function.
- A possible approach involves the keyword `static` inside the operator function, look it up to see how it can help you.

4.2 Doubly linked list

Advanced

Goal

Looking for a challenge?
A doubly linked list is similar to a singly linked list, but each node contains two pointers: one pointing to the next node and another pointing to the previous node. This allows for traversal in both directions. Modify the linked list API above to support doubly linked lists.