Linked List Exercises

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 31
1.0.0

We Could Create Functions That Don't Use Nodes

```
linked_list_t *linked_list_create();
void linked_list_free(linked_list_t *linked_list);
bool linked_list_empty(linked_list_t *linked_list);
int linked_list_length(linked_list_t *linked_list);
void linked_list_print(linked_list_t *linked_list);

void linked_list_insert_front(linked_list_t *linked_list, int val);
int linked_list_remove_front(linked_list_t *linked_list);

/* New functions */
bool linked_list_contains(linked_list_t *linked_list, int val);
bool linked_list_remove_first(linked_list_t *linked_list, int val);
int linked_list_remove_all(linked_list_t *linked_list, int val);
```

Let's Create a Function to Check for a Value

We'll write the following:

```
bool linked_list_contains(linked_list_t *linked_list, int val);
```

It should return `true` if `val` is in the linked list, `false` otherwise

Our Code to Check For a Value in a Linked List

```
bool linked_list_contains(linked_list_t *linked_list, int val) {
    node_t *current = linked_list->head;
    while (current != NULL) {
        if (current->val == val) {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

Let’s Create a Function to Remove the First Value

We’ll write the following:

```
bool linked_list_remove_first(linked_list_t *linked_list, int val);
```

It should return `true` if `val` got removed, `false` otherwise

This function should also `free` the node used to store the value

Removing and Freeing the First Matched Value

```
bool linked_list_remove_first(linked_list_t *linked_list, int val) {
    node_t *previous = NULL;
    node_t *current = linked_list->head;
    while (current != NULL) {
        if (current->val == val) {
            node_t *next = current->next;
            free(current);
            if (previous == NULL) { linked_list->head = next; }
            else                  { previous->next = next; }
            return true;
        }
        previous = current;
        current = current->next;
    }
    return false;
}
```

Let's Create a Function to Remove All Matching Values

We'll write the following:

```
bool linked_list_remove_all(linked_list_t *linked_list, int val);
```

It should return the number of values removed from the linked list

This function should also `free` the node used to store the value

We Can Write This Function Using `linked_list_remove_first`

```
int linked_list_remove_all(linked_list_t *linked_list, int val) {
    int removed = 0;
    while (linked_list_remove_first(linked_list, val)) {
        removed += 1;
    }
    return removed;
}
```

We Can Solve Some Problems Using Linked Lists

In general, you can do implement more complex arithmetic using linked lists

We could sum together all values in the linked list by:
removing two numbers from the front of the list,
adding the values together,
and insert the result to the front of the list

Repeat this process until only one number is left, that's the total sum

Computing the Sum of All Values in a Linked List

```
int linked_list_sum(linked_list_t *linked_list) {
    if (linked_list_empty(linked_list)) {
        return 0;
    }
    while (linked_list_length(linked_list) != 1) {
        int a = linked_list_remove_front(linked_list);
        int b = linked_list_remove_front(linked_list);
        linked_list_insert_front(linked_list, a + b);
    }
    return linked_list_remove_front(linked_list);
}
```

You Could Also Resize an Array

Unless you're constantly adding and removing from the front
arrays are generally faster, and use less space

How would we make an array larger?

We Could `malloc` Space for the Resized Array

Assume `int *array` that points to space for `int length` elements

```
int *new_array = malloc(new_length * sizeof(int));
if (new_array == NULL) {
    free(array);
    exit(EXIT_FAILURE);
}
for (int i = 0; i < min(length, new_length); ++i) {
    new_array[i] = array[i];
}
free(array);
array = new_array;
length = new_length;
```

Resizing an Allocation Exists in the C Standard Library

The function prototype is:

```
void *realloc(void *ptr, size_t new_size);
```

`ptr` is the pointer from a previous call to `malloc` (if `NULL`, it behaves Like `malloc`)
- returns `NULL` on error, and `ptr` isn't free'ed
- otherwise, returns a valid pointer to `new_size` bytes
  - (copying up to old size bytes if needed)

Note: C may be able to expand the original allocation (no copying needed!)

Using `realloc` to Grow or Shrink an Allocation

```
int *new_array = realloc(array, new_length * sizeof(int));
if (new_array == NULL) {
    free(array);
    exit(EXIT_FAILURE);
}
array = new_array;
length = new_length;
```

There is A Version of `malloc` That Initializes Memory

The function prototype is:

```
void* calloc(size_t num, size_t size);
```

It takes two arguments, and multiplies them together for you

This function also initializes all memory to `0`

Note: `malloc` May Zero Initialize the Values by "Luck"

```
int main(void) {
    int length = 4;
    int *array_malloc = malloc(length * sizeof(int));
    arrayPrint(array_malloc, length);
    int *array_calloc = calloc(length, sizeof(int));
    arrayPrint(array_calloc, length);
    free(array_malloc);
    free(array_calloc);
    return EXIT_SUCCESS;
}
```

When we run this program, we may see:

```
array: -1347440721 -1347440721 -1347440721 -1347440721
array: 0 0 0 0
```

If We Resize an Array Often, We May Want to Create a Vector

A vector is a data structure that is basically an array you can resize

```
typedef struct vector {
    int *data;
    int length;
    int capacity;
} vector_t;
```

We'll just keep the current pointer and length in a `struct`

We may also keep track of a capacity which is the number of values we can hold without reallocation

Creating a Vector Dynamically

```
vector_t *vector_create(int length) {
    vector_t *vector = malloc(sizeof(vector_t));
    vector->data = malloc(length * sizeof(int));
    if (vector->data == NULL) {
        exit(EXIT_FAILURE);
    }
    vector->length = length;
    vector->capacity = length;
    return vector;
}
```

Example Code to Resize a Vector, Reallocating if Needed

```
void vector_resize(vector_t *vector, int new_length) {
    if (new_length <= vector->capacity) {
        vector->length = new_length;
    }
    else {
        vector->data = realloc(vector->data, new_length * sizeof(int));
        if (vector->data == NULL) {
            exit(EXIT_FAILURE);
        }
        vector->length = new_length;
        vector->capacity = new_length;
    }
}
```

2022 Final Q14

Write a C function called reorder, the prototype of which is given below, that reorders the nodes in a linked list such that nodes with a value of `0` appear at the front of the linked list and nodes with any other integer value appear at the end of the linked list, while maintaining the original order of non-zero nodes.

Example 1:
Input List: `0 0 15 0 0 13 10`
Output List: `0 0 0 0 15 13 10`

Example 2:
Input List: `1 0 19 0 0 5 0`
Output List: `0 0 0 0 1 19 5`

Note: You are not allowed to copy or modify the data member in any of the nodes in the linked list. However, you can modify the next pointer in the nodes.

```
void reorder(LinkedList *list);
```

Another Alternative Solution for 2022 Final Q14

```
void reorder(linked_list_t *linked_list) {
    node_t *previous = NULL;
    node_t *current = linked_list->head;
    while (current != NULL) {
        if (current->val == 0) {
            if (previous != NULL) {
                node_t *next = current->next;
                current->next = linked_list->head;
                linked_list->head = current;
                previous->next = next;
                current = next;
                continue;
            }
        }
        previous = current;
        current = current->next;
    }
}
```