← Lab 3: linked lists and the heap

Due by midnight, Wednesday 9/26

In this lab, you'll write a simple linked list. You'll have to use `malloc` and `free` to make it.

If you haven't done linked lists before... who was your 445 instructor???

> Project 2 is gonna involve linked lists, so this lab is a way for you to refresh yourself. Please make sure you understand this stuff thoroughly.

Here is the Node struct that you will be using:

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

Here are the functions you will implement:

- `Node* create_node(int value)`
  - it will `malloc` a `Node`, set its value to `value`, set its next to `NULL`, and return it.
    - the lecture 7 slides show how to `malloc` a struct instance.
- `void list_print(Node* head)`
  - `head` points to the head of a linked list.
  - it will print the values of all the items in the list in this format: `5 -> 8 -> 2 -> 1`
- `Node* list_append(Node* head, int value)`
  - `head` points to the head of a linked list.
  - it will go to the **end** of the linked list, use `create_node(value)`, and put that new node at the end of the linked list.
  - then it will return that new node.
- `Node* list_prepend(Node* head, int value)`
  - `head` points to the head of a linked list.
  - it will use `create_node(value)`, make that node point to `head`, and return that new node.

- `void list_free(Node* head)`
  - `head` points to the head of a linked list, **or NULL.**
  - if `head` is NULL, do nothing.
  - otherwise, free all the nodes in the list.
    - **do not free a node before you get its `next` field.**
- `Node* list_remove(Node* head, int value)`
  - `head` points to the head of a linked list.
  - it will search for a node whose value `== value`.
    - if it finds that node, it will **remove it from the list** and `free` it.
      - there are special cases for removing the head and tail!
    - if it doesn't find that node, nothing happens.
  - it will return the **head of the list** (which may have changed or become NULL!)

Hints:

> Read these. It's not "cheating", it's important information.

- Don't write everything at once and then test it. Write one function, test that, and repeat.
- `for` loops fit together with linked lists very nicely.
- Write `list_print` as soon as you can, so you can test your code easier.
- Don't worry about `head == NULL` for anything other than `list_free`.
- For `list_remove`, when you are iterating over the list, don't move your pointer too far ahead...
  - (You have to know the previous node to remove a node.)

`main`

Here's a small driver I wrote. (You can put all your code in one `lab3.c` file.) Feel free to comment stuff out, add more stuff etc. to test your code more thoroughly.

```
int main() {
        // The comments at the ends of the lines show what lis
        Node* head = create_node(1);
        list_print(head);                   // 1
        Node* end = list_append(head, 2);
        list_print(head);                   // 1 -> 2
        end->next = create_node(3);
```

```
        list_print(head);                   // 1 -> 2 -> 3
        head = list_prepend(head, 0);
        list_print(head);                   // 0 -> 1 -> 2 -> 3
        list_append(head, 4);
        list_print(head);                   // 0 -> 1 -> 2 -> 3
        list_append(head, 5);
        list_print(head);                   // 0 -> 1 -> 2 -> 3

        head = list_remove(head, 5);
        list_print(head);                   // 0 -> 1 -> 2 -> 3
        head = list_remove(head, 3);
        list_print(head);                   // 0 -> 1 -> 2 -> 4
        head = list_remove(head, 0);
        list_print(head);                   // 1 -> 2 -> 4

        list_free(head);

        return 0;
}
```

`valgrind`

`valgrind` is a helpful tool. It can analyze your program's memory accesses, heap allocations, frees etc. at runtime to make sure you're not doing anything weird. It can help you find memory leaks, array-out-of-bounds errors, buffer overflows, double-frees, using freed heap memory etc...

To use it, you just put `valgrind` before the program you want to run:

```
valgrind ./lab3
```

It will print lines that look like `==12345==` that will point out issues with your program as it runs.

Common issues:

- **"Invalid read/write of size n"**
  - you are trying to access an invalid pointer.
  - keep reading: it will tell you *where* the read/write happens. Like, the file and line!
- **"LEAK SUMMARY"**

    - you have a memory leak.
    - basically, the problem is in `list_free`.
        - it says "Rerun with –leak-check=full to see details of leaked memory" but if you do that, it just shows where the memory was *allocated*, which isn't too useful for you.
- **"Invalid free() / delete / delete[] / realloc()"**
    - you are trying to call `free()` on something that you aren't allowed to.
    - like something that you already freed.
    - or a pointer that doesn't point to the heap.
    - keep reading the error!!!

Submission

You get the idea by now. (But replace `lab1` with `lab3`.)