# Homework #4

**Released: 02-01-2018**
**Due: 02-08-2018 11:59pm**

We are going to implement a linked-list library and practice (shared) pointer operations. This homework comes with three files:

- `linked_lib.h` contains the declarations of the APIs of the `linked` library. See Section 1 for detailed introduction.
- `linked_lib.cpp` contains the actual implementation of the `linked` APIs.
- `linked_test.cpp` contains the unit tests of the `linked` library.

Most of the APIs accepts an argument of type `List&`, representing the head of a linked-list as we did in the example code in the class. Since it is a reference, changing that argument is the same as changing the variable it is referencing. For example, in the following code, `ptr` will be non-null and `*ptr` will be 9. Review the examples in the class if you are not familiar with this technique.

```
void init_with_9(shared_ptr<int>& node)
{
  node = make_shared<int>(9);
}

shared_ptr<int> ptr;
CHECK(ptr == nullptr);
init_with_9(ptr);
CHECK(ptr != nullptr && *ptr == 9);
```

There is no main program for this homework. When compiling, an executable `linked_test` will be built. This is the executable for unit testing. Similar to homework 2, we are going to implement the APIs, handle incorrect arguments and write unit tests for the library.

When doing this homework, it will be very helpful to draw a diagram for every statement involving pointers in the code. This is not part of the requirement, but just a way to help making it easier to track pointer operations.

## 1 Linked-List Library

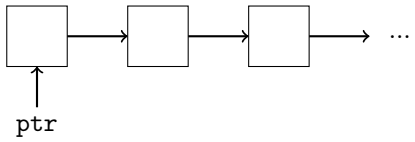
```
struct ListNode
{
  int data;
  std::shared_ptr<ListNode> next;
};

using List = std::shared_ptr<ListNode>;
```

Linked-lists are represented as shared pointers to `ListNode` structs. A `ListNode` struct contains a `data` field, and a shared-pointer field `next` pointing to further nodes, as in the examples in the class. Being a shared pointer, `nullptr` is also a valid linked-list which has no nodes.

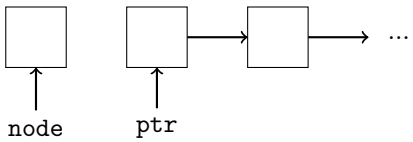
In this homework, we *guarantee* that all linked-lists will be valid. Any two linked-lists will not intersect with each other, and no linked-list contains a loop. Now, please complete the implementation of the following functions in `linked_lib.cpp`.

### 1.1 `List pop_front(List& front);`

The function `List pop_front(List& front);` removes the first node from the linked-list `front` and returns the original first node. `pop_front` should also set the `next` of the original first node to `nullptr`. For example, before invoking `pop_front` on `ptr`, we have



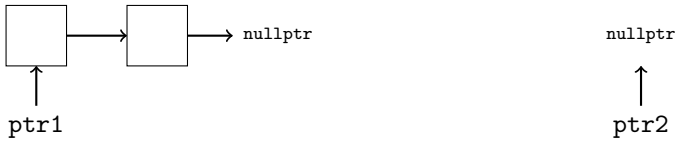

After running `node = pop_front(ptr);`, the first node of the linked-list pointed by `ptr` is removed. `ptr` now points to the second node, and the original first node is returned by `pop_front` with its `next` set to `nullptr`.



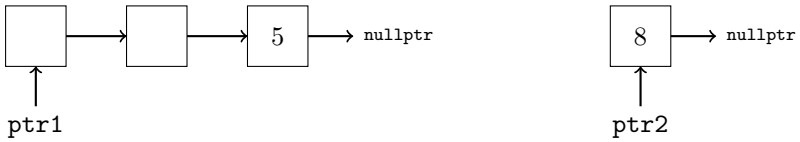

### 1.2 `void push_back(List& front, int data);`

`void push_back(List& front, int data);` creates a new `ListNode` containing `data` and insert that new node to the end of the linked-list pointed by `front`. If `front` is `nullptr`, simply make `front` point to the new node.

Before invoking `push_back`:




After running `push_back(ptr1, 5);` and `push_back(ptr2, 8);`:



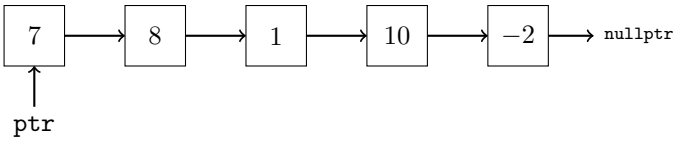

### 1.3 `int& nth_element(List front, size_t n);`

`int& nth_element(List front, size_t n);` returns the reference to the `data` in the $n$th element of the linked-list `front`, counting from zero. For example, `nth_element(ptr, 2);` returns the reference to the `data` that contains 5 in Section 1.2.

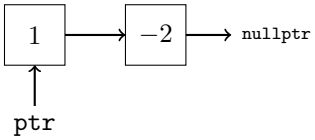
Note that you must not copy `data`, but directly return `p->data` for some node pointer `p`.

### 1.4 `void filter_lt(List& front, int limit);`

`void filter_lt(List& front, int limit);` deletes all elements that are greater than or equal to `limit` in the linked-list `front` while keeping all other elements intact. For example, before invoking `filter_lt`,




After running `filter_lt(ptr, 7);`:




Note that the nodes in the list after `filter_lt` should be the original nodes passed to it. It must not allocate, meaning it must not call `make_shared` either directly or indirectly.

## 2 Write Unit Tests and Handle Errors

As in homework 2, please throw a `runtime_error` for erroneous arguments. `push_back` will never fail; `filter_lt` simply does nothing for empty linked-lists. The only erroneous cases are:

- `pop_front` where `front` is `nullptr`.
- `nth_element` where `n` is out of bound. That is, `n` $\geq$ length of `front`.

Please also implement *proper* unit tests for every API. You have to figure out what cases there might be and implement a corresponding unit test to ensure that the API works properly under the assumption that the APIs will only be invoked with valid linked-lists.

There is one sample unit test provided in `linked_lib.cpp` demonstrating how you could write a simple test for linked-lists. The test starts by setting up a made-up linked-list as the input for the API, invoke the API, and then examing that the result is as expected. You may leave that test there or remove it.