Linked Lists

Pintos contains a linked list data structure in `lib/kernel/list.h` that is used for many different purposes. This linked list implementation is different from most other linked list implementations you may have encountered, because **it does not use any dynamic memory allocation**.

```
/* List element. */
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };
/* List. */
struct list
  {
    struct list_elem head;      /* List head. */
    struct list_elem tail;      /* List tail. */
  };
```

In a Pintos linked list, each list element contains a `struct list_elem`, which contains the pointers to the next and previous element. Because the list elements themselves have enough space to hold the prev and next pointers, we don't need to allocate any extra space to support our linked list. Here is an example of a linked list element which can hold an integer:

```
/* Integer linked list */
struct int_list_elem
  {
    int value;
    struct list_elem elem;
  };
```

Next, you must create a `struct list` to represent the whole list. Initialize it with `list_init()`.

```
/* Declare and initialize a list */
struct list my_list;
list_init (&my_list);
```

Now, you can declare a list element and add it to the end of the list. Notice that the second argument of `list_push_back()` is the address of a `struct list_elem`, not the `struct int_list_elem` itself.

```
/* Declare a list element. */
struct int_list_elem three = {3, {NULL, NULL}};
/* Add it to the list */
list_push_back (&my_list, &three.elem);
```

We can use the `list_entry()` macro to convert a generic `struct list_elem` into our custom `struct int_list_elem` type. Then, we can grab the `value` attribute and print it out:

```
/* Fetch elements from the list */
struct list_elem *first_list_element = list_begin (&my_list);
struct int_list_elem *first_integer = list_entry (first_list_element,
                                                  struct int_list_elem,
                                                  elem);
printf("The first element is: %d\n", first_integer->value);
```

By storing the prev and next pointers inside the structs themselves, we can avoid creating new `linked list element` containers. However, this also means that a `list_elem` can only be part of one list a time. Additionally, our list should be homogeneous (it should only contain one type of element).

The `list_entry()` macro works by computing the offset of the `elem` field inside of `struct int_list_elem`. In our example, this offset is 4 bytes. To convert a pointer to a generic `struct list_elem` to a pointer to our custom `struct int_list_elem`, the `list_entry()` just needs to subtract 4 bytes! (It also casts the pointer, in order to satisfy the C type system.)

Linked lists have 2 sentinel elements: the `head` and `tail` elements of the `struct list`. These sentinel elements can be distinguished by their `NULL` pointer values. Make sure to distinguish between functions that return the first actual element of a list and functions that return the sentinel `head` element of the list.

There are also functions that sort a link list (using quicksort) and functions that insert an element into a sorted list. These functions require you to provide a list element comparison function (see `/lib/kernel/list.h` for more details).