There are many major differences between my own implementation compared to the one in the Linux kernel. Namely, the Linux implementation for add and remove contains various validation functions to see if an add or a remove is possible. This is good as linked lists heavily deal with memory allocation and pointing to objects and adding and removing entries to and from a corrupted list in a complex system can lead to potential security vulnerabilities.

```
/*
 * Performs the full set of list corruption checks before __list_add().
 * On list corruption reports a warning, and returns false.
 */
bool __list_valid_slowpath __list_add_valid_or_report(struct list_head *new,
                                                      struct list_head *prev,
                                                      struct list_head *next);

/*
 * Performs list corruption checks before __list_add(). Returns false if a
 * corruption is detected, true otherwise.
 *
 * With CONFIG_LIST_HARDENED only, performs minimal list integrity checking
 * inline to catch non-faulting corruptions, and only if a corruption is
 * detected calls the reporting function __list_add_valid_or_report().
 */
static __always_inline bool __list_add_valid(struct list_head *new,
                                             struct list_head *prev,
                                             struct list_head *next)
{
        bool ret = true;

        if (!IS_ENABLED(CONFIG_DEBUG_LIST)) {
                /*
                 * With the hardening version, elide checking if next and prev
                 * are NULL, since the immediate dereference of them below would
                 * result in a fault if NULL.
                 *
                 * With the reduced set of checks, we can afford to inline the
                 * checks, which also gives the compiler a chance to elide some
                 * of them completely if they can be proven at compile-time. If
                 * one of the pre-conditions does not hold, the slow-path will
                 * show a report which pre-condition failed.
                 */
                if (likely(next->prev == prev && prev->next == next && new != prev && new != next))
                        return true;
                ret = false;
        }

        ret &= __list_add_valid_or_report(new, prev, next);
        return ret;
}
```

The delete/remove method in the Linux implementation does not free the memory after removing the entry. It does however provide a variant of the remove function that reinitializes the delete entry as a part of the structure. The documentation within linux/list.h states that delete should be used if the user intends to discard the whole structure, whereas delete_init should be used if the user wants to delete the entry within the list but still keep the structure the list is attached to. This also allows for simpler and cleaner code, which can be incredibly useful when calling the structure repeatedly.

```
/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_del(struct list_head * prev, struct list_head * next)
{
        next->prev = prev;
        WRITE_ONCE(prev->next, next);
}

/*
 * Delete a list entry and clear the 'prev' pointer.
 *
 * This is a special-purpose list clearing method used in the networking code
 * for lists allocated as per-cpu, where we don't want to incur the extra
 * WRITE_ONCE() overhead of a regular list_del_init(). The code that uses this
 * needs to check the node 'prev' pointer instead of calling list_empty().
 */
static inline void __list_del_clearprev(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->prev = NULL;
}

static inline void __list_del_entry(struct list_head *entry)
{
        if (!__list_del_entry_valid(entry))
                return;

        __list_del(entry->prev, entry->next);
}

/**
 * list_del - deletes entry from list.
 * @entry: the element to delete from the list.
 * Note: list_empty() on entry does not return true after this, the entry is
 * in an undefined state.
 */
static inline void list_del(struct list_head *entry)
{
        __list_del_entry(entry);
        entry->next = LIST_POISON1;
        entry->prev = LIST_POISON2;
}
```

This separation allows for the end user to have greater control over memory management and only delegates the kernel to handle only linkage.

How the list is used is also fairly different from many traditional doubly linked list implementations. Rather than defining a list structure that will contain various objects, the kernel embeds the list within the object's structure.

```
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

/**
 * INIT_LIST_HEAD - Initialize a list_head structure
 * @list: list_head structure to be initialized.
 *
 * Initializes the list_head to point to itself.  If it is a list header,
 * the result is an empty list.
 */
static inline void INIT_LIST_HEAD(struct list_head *list)
{
        WRITE_ONCE(list->next, list);
        WRITE_ONCE(list->prev, list);
}

#ifdef CONFIG_LIST_HARDENED

#ifdef CONFIG_DEBUG_LIST
# define __list_valid_slowpath
#else
# define __list_valid_slowpath __cold __preserve_most
#endif
```

Example use case:

```
struct fox {
    unsigned long tail_length; /* length in centimeters of tail */
    unsigned long weight; /* weight in kilograms */
    bool is_fantastic; /* is this fox fantastic? */
    struct list_head list; /* list of all fox structures */
};
```

- Source: https://github.com/firmianay/Life-long-Learner/blob/master/linux-kernel-development/chapter-6.md

In my implementation, I defined the data within the list for the list to store and create pointers to. The Linux implementation does the inverse approach, utilizing a macro to take any list entry and return the structure holding that list's head. What this can do is allow for much greater flexibility in storing data, as the list code doesn't need to be changed to store the data itself but can be embedded into a structure to "store" the data associated with that one list. It also allows for multiple lists to be used in a single program to refer to different structures.

```
/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 *
 * WARNING: any const qualifier of @ptr is lost.
 */
#define container_of(ptr, type, member) ({                              \
        void *__mptr = (void *)(ptr);                                   \
        static_assert(__same_type(*(ptr), ((type *)0)->member) ||       \
                      __same_type(*(ptr), void),                        \
                      "pointer type mismatch in container_of()");       \
        ((type *)(__mptr - offsetof(type, member))); })
```