Doubly-Linked List Challenge

Matthew CntKillMe

December 27, 2018

Contents

1 Introduction

Your task is to implement a doubly-linked list data structure with the given interface. You will be judged based on the following criteria (in order of most important to least important):

- Correct implementation of the interface.
- Proper memory management.
- Code readability, reuse, and performance.

The additional challenges are optional, however it is recommended for those who want more of a challenge. For the sake of simplicity, assume no boundary errors and memory allocation errors can take place.

2 Required Interface

```c
struct linked_list;
struct node;
typedef double value_t;

typedef struct linked_list
{
        struct node* first;
        struct node* last;
        size_t size;
} linked_list;

typedef struct node
{
        struct node* prev;
        struct node* next;
        value_t value;
} node;

/**
 * Initialize a linked_list object.
 */
void linked_list_init(linked_list* list);

/**
 * Copies a linked_list and all of its elements.
 * The two lists should be fully independent of each other.
 * Assume the destination list is initialized but empty.
 */
void linked_list_copy(linked_list* dest, const linked_list* src);
```

```c
/**
 * Clears a linked_list of all its elements.
 */
void linked_list_clear(linked_list* list);

/**
 * Resizes a linked_list to the given size. For newly created nodes, initialize
 *  them with the given value.
 */
void linked_list_resize(linked_list* list, size_t newSize, value_t value);

/**
 * Returns the size (number of elements) of a linked_list.
 */
size_t linked_list_size(const linked_list* list);

/**
 * Returns the first element of a linked_list
 * Assume the list is not empty.
 */
value_t linked_list_front(const linked_list* list);
/**
 * Returns the last element of a linked_list
 * Assume the list is not empty.
 */
value_t linked_list_back(const linked_list* list);

/**
 * Adds an element with the given value to the end of a linked_list.
 */
void linked_list_push_front(linked_list* list, value_t value);

/**
 * Adds an element with the given value to the beginning of a linked_list.
 */
void linked_list_push_back(linked_list* list, value_t value);

/**
 * Removes the element at the beginning of a linked_list and returns it.
 * Assume the list is not empty.
 */
value_t linked_list_pop_front(linked_list* list);

/**
 * Removes the element at the end of a linked_list and returns it.
```

```c
 * Assume the list is not empty.
 */
value_t linked_list_pop_back(linked_list* list);

/**
 * Returns the element at the given index of a linked_list.
 * Assume idx is in the range [0, size)
 */
value_t linked_list_get(const linked_list* list, size_t idx);

/**
 * Alters the element at the given index of a linked_list and
 *  returns the old value.
 * Assume idx is in the range [0, size)
 */
value_t linked_list_set(linked_list* list, size_t idx, value_t newValue);
```

3 Additional Challenges

3.1 Extra Functionality

```c
typedef int (*comparator_t)(value_t, value_t);
typedef void (*callback_t)(value_t);

/**
 * Reverses the elements of a linked_list.
 */
void linked_list_reverse(linked_list* list);

/**
 * Sorts the elements of a linked_list in the order defined by a comparator.
 */
void linked_list_sort(linked_list* list, comparator_t comparator);

/**
* Appends one linked_list to the end of another.
* The source linked_list should become an empty linked list.
*/
void linked_list_append(linked_list* dest, linked_list* src);

/**
 * Iterates over a linked_list and invokes a callback for each element.
 */
void linked_list_foreach(const linked_list* list, callback_t callback);
```

```c
/**
 * Swaps the elements of two linked_lists.
 */
void linked_list_swap(linked_list* list1, linked_list* link2);
```

3.2 Iterators

An iterator is a *pointer-like* object that refers to an element. It is *invalidated* when the associated node has been destroyed.

```c
typedef node* iter_t;
typedef const node* const_iter_t;

/**
 * Returns an iterator to the first element of a linked_list.
 * If the list is empty, the end iterator is returned.
 */
iter_t linked_list_begin(const linked_list* list);

/**
 * Returns an iterator to one after the last element of a linked_list.
 */
iter_t linked_list_end(const linked_list* list);

/**
 * Returns the element associated with an iterator.
 * Assume iter is in the range [begin, end).
 */
value_t linked_list_read(const linked_list* list, const_iter_t iter);

/**
 * Alters the element associated with an iterator and returns the old value.
 * Assume iter is in the range [begin, end).
 */
value_t linked_list_write(const linked_list* list, iter_t iter);

/**
 * Advances an iterator by a number of steps, a negative step
 *  indicates advancing backwards.
 * Assume iter + steps will be in the range [begin, end].
 */
iter_t linked_list_advance(const linked_list* list, iter_t iter, ptrdiff_t steps);

/**
 * Inserts an element before a given iterator and
 *  returns an iterator to the new element.
```

```
 */
iter_t linked_list_insert(linked_list* list, iter_t iter, value_t value);

/**
 * Erases an element at the given iterator and
 *  returns the iterator following the erased element.
 * Assume iter is in the range [begin, end).
 */
iter_t linked_list_erase(linked_list* list, iter_t iter);

/**
 * Returns the distance between two nodes, negative if first comes after last.
 */
ptrdiff_t linked_list_dist(linked_list* list, const_iter_t first,
  const_iter_t last);
```

3.3 Extra Iterator Functionality

```
/**
 * Inserts some number elements before the given iterator that are
 *  initialized with then given value.
 */
iter_t linked_list_insert_many(linked_list* list, iter_t begin, size_t count,
  value_t value);

/**
 * Erases all elements in the range [first, last)
 * Assume dist(first, last) is non-negative and first != end.
 */
iter_t linked_list_erase_range(linked_list* list, iter_t first, iter_t last);

/**
 * Swaps the nodes associated with the two iterators.
 * Assume iter1, iter2 are in the range [begin, end).
 */
void linked_list_swap_nodes(linked_list* list, iter_t iter1, iter_t iter2);

/**
 * Reverses the nodes of a linked_list by their elements from [first, last).
 * Assume dist(first, last) is non-negative and first != end.
 */
void linked_list_reverse_nodes(linked_list* list, iter_t first, iter_t last);

/**
 * Sorts the nodes of a linked_list by their elements from [first, last)
 *  in the order defined by a comparator.
```

```
 * Assume dist(first, last) is non-negative, and and first != end.
 */
void linked_list_sort_nodes(linked_list* list, iter_t first, iter_t last,
  comparator_t comparator);
```

4 Interface Reference

The expected runtime complexity does not account for library function calls, assume those are O(1).
Variable n refers to the size of the linked_list.

Interface Reference		
linked_list_*	**Runtime Complexity**	**Iterator Invalidation**
new	$O(1)$	
copy	$O(n)$	
free	$O(n)$	[begin, end)
clear	$O(n)$	[begin, end)
resize	$O(\|n - newSize\|)$	[begin+newSize, end)
size	$O(1)$	
front	$O(1)$	
back	$O(1)$	
push_front	$O(1)$	
push_back	$O(1)$	
pop_front	$O(1)$	last
pop_back	$O(1)$	first
get	$O(idx)$	
set	$O(idx)$	
reverse	$O(n)$	undefined
sort	$O(n^2)$	undefined
append	$O(1)$	
foreach	$O(n)$	
swap	$O(1)$	
begin	$O(1)$	
end	$O(1)$	
read	$O(1)$	
write	$O(1)$	
advance	$O(steps)$	
insert	$O(1)$	
erase	$O(1)$	iter
dist	$O(\|dist(first, last)\|)$	
insert_many	$O(count)$	
erase_range	$O(\|dist(first, last)\|)$	[first, last)
swap_nodes	$O(1)$	
reverse_nodes	$O(dist(first, last))$	
sort_nodes	$O(dist^2(first, last))$	