Linked list structure in C

Sam Serbouti serbouti@kth.se

September 25, 2024

Introduction

In this assignment, I shift from working with primitive data structures and arrays to more complex structures that use pointers to link elements. I start by implementing methods such as adding, finding, and removing items in the list, as well then evaluate the performance of appending one linked list to another as their size increases.

1 Implementing linked lists in C

Unlike arrays, (that are blocks of memory with a fixed size) linked lists consist of individual elements (`cell`) that hold data (`int value`) and a reference to the next cell in the sequence (`cell *tail`).

When implementing this data structure, I must be careful with having robust memory allocations (ie after every malloc(), I check if the memory was allocated successfully). Specifically, when I `append()` list b to list a, list a now owns the cells of list b. So, after appending, we only need to free the structure of list b, not its cells (because they are now part of a).

```c
void add(linked *lnk, int item) {
    cell *new_cell = ..
      ..(cell *)malloc(sizeof(cell));
    if (!new_cell) {
        fprintf(stderr, "error");
        exit(1);
    }
    new_cell->value = item;
    new_cell->tail = lnk->first;
    lnk->first = new_cell;
}

int length(linked *lnk) {
    int count = 0;
    cell *current = lnk->first;
    while (current != NULL) {
        count++;
        current = current->tail;
    }
    return count;
}

bool find(linked *lnk, int item) {
    cell *current = lnk->first;
    while (current != NULL) {
        if (current->value == item) {
            return true;
        }
        current = current->tail;
    }
    return false;
}
```

```c
void append(linked *a, linked *b) {
    if (a == NULL || b == NULL)
        return;
    if (a->first == NULL) {
        a->first = b->first;
        return;
    }
    cell *current = a->first;
    while (current->tail != NULL)
        {current = current->tail;}
    current->tail = b->first;
}

linked *init_list(int n) {
    linked *a = create_linked();
    for (int i = 0; i < n; i++) {
        add(a, i);
    }
    return a;
}
```

```c
void remove_item(linked *lnk, int item) {
    cell *current = lnk->first;
    cell *previous = NULL;
    while (current != NULL) {
        if (current->value == item) {
            if (previous == NULL) {
            lnk->first = current->tail;
            }
            else {
            previous->tail = current->tail;
            }
            free(current);
            return;
        }
        previous = current;
        current = current->tail;
    }
}
```

2 Appending runtime benchmarks

For each benchmark, I perform a warm-up benchmark operation with 9000000 cells. `benchmark1()` consists of setting list b's size to 100 cells and increasing list a's size from 100 to 10000000 on a log scale.
`benchmark2()` does the same but sets list a's size to 100 and increases list b's size similarly.

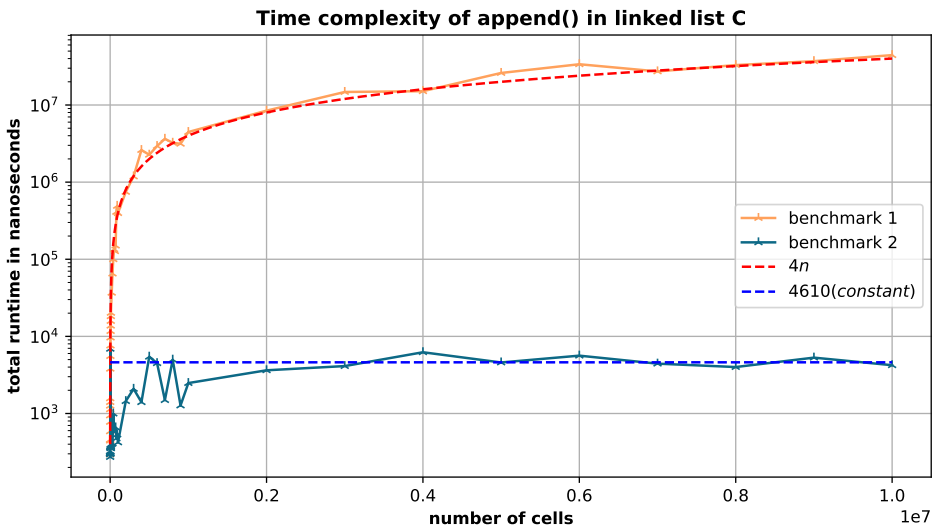

Figure 1: If the second list is fixed $\mathcal{O}(n)$ if first is fixed $\mathcal{O}(1)$

Observations

In `benchmark1()`, list a grows, and since the append function goes trhough all the elements in list a to find the last node, the time complexity is $\mathcal{O}(n)$.

But in `benchmark2()`, list a's size is fixed, and appending list b means just linking the last node of a to the first node of b. This happens in constant time, so the time complexity is $\mathcal{O}(1)$ regardless of the size of list b.

3 Comparing Linked lists with Arrays

Implementing append() using arrays

WHen using arrays instead of a linked list, the primary difference is that arrays have a fixed size. So to append one array to another I need to:

1. determine the total size `*new_len = len_a + len_b;`
2. allocate the new array `int *result = (int *)malloc((*new_len) * sizeof(int));`
3. copy all the elements from a to the new
4. copy all the elements from b to `new[len_a + i] = b[i];`
5. free old arrays a and b (if needed)

Time Complexity of Array Append

The time complexity of the append operation is $\mathcal{O}(len(a) + len(b))$ because I must copy all the elements from a, which takes $\mathcal{O}(len(a))$ time and then, copy all the elements from b, which takes $\mathcal{O}(len(b))$ time.

For the first benchmark this *might* not have an impact if $len(b) \ll len(a)$. But for `benchmark2()`, linked lists out-beat arrays:

	benchmark1 (setting b)	benchmark2 (setting a)
linked list	$\mathcal{O}(len(a))$	$\mathcal{O}(1)$
array	$\mathcal{O}(len(a) + len(b))$	$\mathcal{O}(len(a) + len(b))$

Thus, appending can be more efficient with linked list if you're just linking the last node of a to the first node of b because you don't need to traverse or copy the elements of b.

4 Making a stack out of linked list

A stack follows a LIFO (Last In, First Out) principle, where the last element added is the first one to be removed (using linked lists is useful because of it is dynamic). Stacks are caracterized by two operations : `push` and `pop`.

```
void push(linked *stack, int item) {
    add(stack, item);  //add item to front of the linked list
}

int pop(linked *stack) {
    if (stack->first == NULL) {return -1;}  //underflow

    int value = stack->first->value;
    remove(stack, value);
    return value;
}
```

	array	linked list
push	add item tat the end of array, could need resizing: $\mathcal{O}(n)$	$\mathcal{O}(1)$
pop	at the head of list, resizing is never needed: $\mathcal{O}(1)$	$\mathcal{O}(1)$

I can see that the only major downside of using linked lists for building a stack instead of an array, is the complexity of implementation. Additionally, the advantage of arrays is the proximity of memory slots used for the data structure that could have an impact on access time. Besides that, one should resort to linked lists.

Conclusion

Arrays are preferable when you need efficient index-based access (in an array, accessing an element by its index takes $\mathcal{O}(1)$ time ; in a linked list, it takes $\mathcal{O}(n)$ time because you must traverse the list from the head.) or when working with a fixed number of elements (more space-efficient and simpler to work with, as it avoids the overhead of storing pointers for each cell).

Linked lists provide more flexibility for dynamic stacks (the stack can grow or shrink as needed without worrying about the initial size, we don't need to resize the structure when it grows) but with extra memory overhead (each element in the stack requires extra memory for the pointer) and slightly slower access times compared to arrays (linked lists use scattered memory allocations, which can result in slower access compared to arrays) though push and pop are handled efficiently ($\mathcal{O}(1)$ since they involve adding or removing elements from the head of the list).