Linked Lists

Muhammad Shahwaiz Abbasi

VT 2026

Introduction

This report presents the fifth assignment for the Data Structures and Algorithms (ID1021) course.

We will be implementing the basic linked list operations such as `add, remove, length, find and append`. Then we will analyze the runtime complexity of the append function. At last, we will compare linked lists to arrays and discuss the pros and cons of using a dynamic stack with a linked list.

For benchmark the time was measured in nanoseconds using a function from `Assignment 1`.

The objective of this assignment is to understand how linked lists work and the implementation of their core functions.

What is a Linked List?

Linked list is a simple linked structure. Linked list holds a sequence of cells but only have access to the first cell in the sequence. These cells contain integers and unlike arrays the cells are not stored next to each other. They are all over the memory. So one cell contains the value and also the reference or *pointer* to the next cell in the sequence. If a cell's reference is a *null pointer* then it is the last item in the list. Linked Lists are useful because they can dynamically adjust the length when adding or removing cells.

The following is the blueprint for linked list that holds a sequence of integer:

```
typedef struct cell {
    int value;
    struct cell *tail;
} cell;
typedef struct linked {
    cell *first;
```

```
} linked;
```

Basic Operations

In this assignment, we were first asked to implement the basic operations for linked lists. These functions are as follows:

```
void linked_add(linked *lnk, int item){
    cell *new = (cell*)malloc(sizeof(cell));
    new->value = item;
    new->tail = lnk->first;
    lnk->first = new;
}
int linked_length(linked *lnk){
    int leng = 0;
    cell *nxt = lnk->first;
    while (nxt != NULL) {
        leng++;
        nxt = nxt->tail;
    }
    return leng;
}
bool linked_find(linked *lnk, int item){
    cell *nxt = lnk->first;
    while (nxt != NULL) {
        if (nxt->value == item) {
            return true;
        }
        nxt = nxt->tail;
    }
    return false;
}
void linked_remove(linked *lnk, int item){
    cell *nxt = lnk->first;
    cell *prev = NULL;
    while (nxt != NULL) {
        if (nxt->value == item) {
            if (prev == NULL) {
                lnk->first = nxt->tail;
            } else {
                prev->tail = nxt->tail;
            }
            break;
        }
        prev = nxt;
```

```
        nxt = nxt->tail;
    }
```

The last method we were asked to make an `append` function. This function connects one list to another list.

```
void linked_append(linked *a, linked *b) {
    cell *nxt = a->first;
    cell *prev = NULL;
    while(nxt != NULL) {
        prev = nxt;
        nxt = nxt->tail;
    }
    if (prev != NULL){
        prev->tail = b->first;
    }
    else{
        a->first = b->first;
    }
    b->first = NULL;
}
```

In this code the last cell of the first list is pointing to NULL but after implementation of the append function, that pointer is updated to point to the first cell of the second list.

Benchmarking

After implementing the operations, our task is now to benchmark the append operation to determine the time complexity of varying size linked list `a` with a fixed size linked list `b`. We benchmark from dynamic `a` to fixed `b`. Then we reverse the process, benchmarking from a fixed size list `a` to a varying size list `b`. For Dynamic to fixed. We set the size of `b` to 100 and increased `a's` size from 1,000 to 100,000. For the fixed to dynamic, we swapped these size parameters.

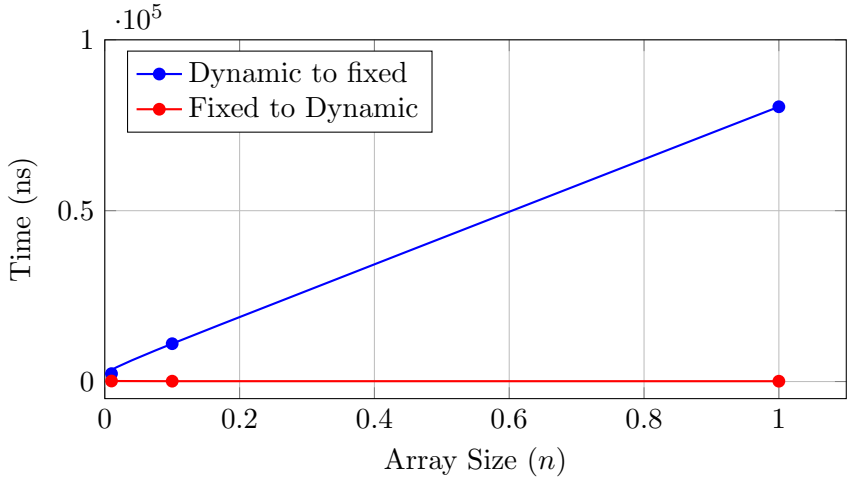

Figure 1: Dynamic to fixed vs Fixed to Dynamic Performance

Looking at `Figure 1`, we can clearly see that fixed a and dynamic b is more time efficient, as it has a time complexity of $\mathcal{O}(1)$ whereas dynamic a to fixed b has the complexity of $\mathcal{O}(n)$. The reason why this happens is that in dynamic a to fixed b we have to go through a to find the last cell and then merge the b into a. But when the size is fixed (fixed a to dynamic b) it take almost same time to goes thorough a to find the last cell.

Comparison on Arrays Using the Append Operation

When adding more elements to an array, we have to create a new array with the length needed and copy all elements from the original array a. We then use another `for` loop to continue adding the elements of the array b into the newly created array. This results in a time complexity of $\mathcal{O}(n + m)$ where n and m are the sizes of the two array being merged in the operation.

Linked List Stack, Pros and Cons

Push and pop are the two primary operations of a stack. Push adds an element to the top and pop removes the element from the top. In a linked list implementation, the top of the stack is represented by the first element of the list. To implement the push operation, we simply insert a new cell at the beginning of the list. Similarly, the pop operation is implemented by removing the first cell. An array based stack stores elements next to each other which makes time complexity for Push and Pop function also $\mathcal{O}(1)$. This is memory efficient because it does not contain any pointers but if we have to change the size of an array then the time complexity becomes $\mathcal{O}(n)$. From the information above, we can conclude that arrays based stack are faster than the Linked list stack.

- **Pros:**
  - A linked list can easily change its length by adding or removing cells without creating a new list or copying all the elements unlike an array, which takes $\mathcal{O}(n)$ time for such operations. Push and Pop in Linked list has a time complexity of $\mathcal{O}(1)$.
  - It also saves us from stack overflow as it can dynamically adjust the size.
- **Cons:**
  - A linked list uses memory inefficiently because each cell requires extra space for its data and for a pointer to the next cell.
  - Accessing an element is slower because of pointer dereferencing and dynamic memory allocation.