Linked Lists

Edward Sharp

14-02-25

1 Introduction

This report aims to explore linked lists, implement basic methods for them, measure results and link them theoretically to their time complexities. Additionally, we will explore the implementation of a stack using linked lists and discuss execution time differences.

2 Implementation Details

The linked list data structure consists of individual nodes implemented as a ‘struct‘. Each node stores a value and a pointer to the next node in the sequence. A linked list stores a pointer to the first node, referred to as its ‘head‘.

2.1 Creating and Freeing a Linked List

The linked list and its nodes are dynamically allocated memory using ‘malloc‘. The following methods initialize and free a linked list:

```
linked *linked_create() {
    linked *new = (linked*)malloc(sizeof(linked));
    new->first = NULL;
    return new;
}

void linked_free(linked *lnk) {
    cell *nxt = lnk->first;
    while (nxt != NULL) {
        cell *tmp = nxt->tail;
        free(nxt);
        nxt = tmp;
    }
    free(lnk);
}
```

2.2 Adding a Node to the Beginning

The ‘linked_add‘ function allocates memory for a new node, assigns the given value, and links it as the head of the list.

```
void linked_add(linked *lnk, int item) {
    cell *new = (cell*)malloc(sizeof(cell));
    new->value = item;
    new->tail = lnk->first;
    lnk->first = new;
}
```

3 Benchmarks

To study the time complexity of the ‘`linked_append`‘ operation, benchmarks were performed by varying the size of the linked lists.

3.1 Varying the Size of List A

In the first benchmark, the size of list ‘a‘ was varied while keeping the size of list ‘b‘ constant. The code used for this initialization is given below:

```
linked *linked_init(int n) {
    linked *lnk = linked_create();
    for (int i = 0; i < n; i++) {
        linked_add(lnk, i);
    }
    return lnk;
}
```

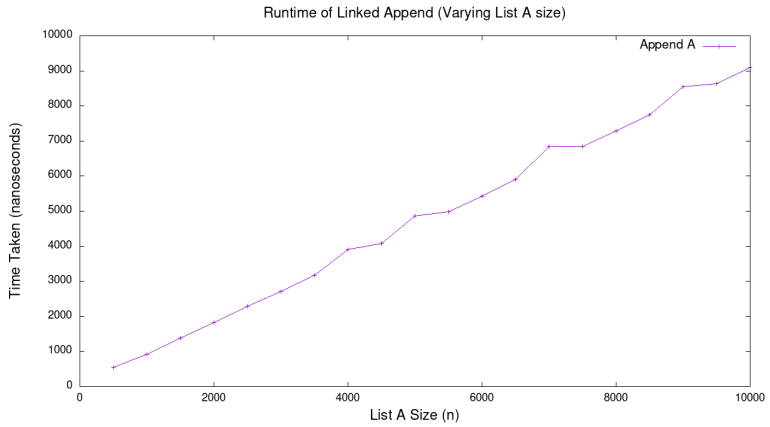

Figure 1: Runtime of `linked_append` varying the size of list a.

The results showed that the runtime of ‘`linked_append`‘ increases linearly with the size of list ‘a‘, indicating a time complexity of $O(n)$. This can be observed in Figure 1.

3.2 Varying the Size of List B

In the second benchmark, the size of list ‘b‘ was varied while keeping the size of list ‘a‘ constant. The results demonstrated that the runtime remains constant, confirming that the time complexity is not dependent on the size of list ‘b‘.

This behavior occurs because the implementation of ‘`linked_append`‘ does not involve traversing list ‘b‘. Instead, the function only traverses list ‘a‘ to find the last element and links the head of list ‘b‘ as its new tail, which is done in $O(1)$ time with respect to the size of list ‘b‘.

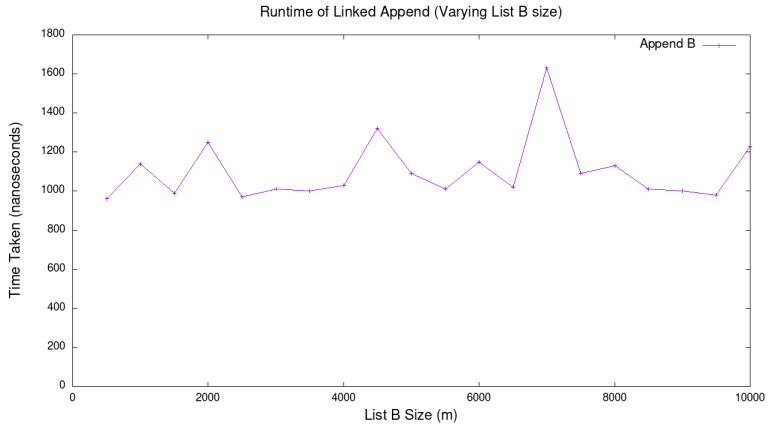

Figure 2: Runtime of `linked_append` varying the size of list b.

Thus, the results confirm that appending list ‘b‘ to list ‘a‘ is independent of the size of list ‘b‘ and has a constant time complexity, $O(1)$, with respect to the size of list ‘b‘.

4 Comparison with Arrays

If the ‘append‘ operation were implemented using arrays, it would involve creating a new array with enough space to store the contents of both input arrays, copying elements from both arrays into the new array, and releasing the original arrays.

This process would result in a time complexity of $O(n + m)$, where n and m are the sizes of the two arrays. Compared to linked lists, arrays require additional memory allocation and copying overhead but provide faster random access times.

The results of the benchmarks highlight that while arrays offer faster construction for smaller datasets, linked lists handle dynamic memory operations and growth more efficiently, particularly in scenarios where constant time complexity for appending is required.

5 Stack with Linked Lists

Using a linked list, a stack can be efficiently implemented. The ‘linked_add‘ operation can serve as the ‘push‘ operation, adding elements to the top of the stack. The ‘pop‘ operation can be implemented by removing and returning the first element of the linked list.

When compared to arrays, linked list stacks offer a dynamic growth advantage without requiring reallocation. However, they incur additional overhead due to dynamic memory allocation and pointer management.