Shopping Cart Report

Built a Shopping Cart system using a pointer array where each element in the array is a pointer holding a memory address to some data value. In my program I input clothing items such as Nike’s clothing items, Adidas items, and New Balance items. Along with duplicate items in the list. By implementing a dynamic array, I had to manage memory and avoid memory leaks. In this report, I discuss the advantages and disadvantages of my approach and ways I could have improved it.

**The Pros**

By implementing an array which is a linear data structure that has elements in sequences - which are contiguous and is fixed in size. We are able to utilize memory space by adding elements to each block of memory space in sequences. In addition, when accessing elements in an array, it becomes more efficient because we can identify the location of an element with an array’s index. This direct access becomes a time complexity of O(1).

Another pro was every time our array reached half of its memory usage, I would double the capacity of the array. If the array had 10 capacity and after 5 elements, it would double to 20 capacity and so forth. This type of implementation was crucial to avoid the program from crashing.

**The Cons**

The cons with my implementation was when finding an item, I did not use any sorting or searching algorithms. For example, when finding an item to remove in an unsorted array, it would search every index in the array until it found the item. This time complexity is O(n) because it has to search every index. This can be improved by sorting the array in ascending or descending order and dividing the search in half. If we assume the array was sorted in alphabetical order, and we wanted to remove the item “Nike Wildhorse 10” in the array, we can search from the half to the end of the elements. And not have to search from index 0 to the half.

Another con of the implementation when removing an item is having to move all the elements that won’t be removed to a new array. If I needed to remove an element in index 23 and had 100 elements. I would have to loop through from the **first element** to the element before the i**tem\_to\_delete** and also use another loop to loop from the **item\_to\_delete + 1** to the end of the elements. This reallocation causes lots of overhead. You can imagine if there were thousands of elements or millions of elements, this can be expensive. Frequent resizing can be slow.

Lastly, I mentioned earlier that our array’s capacity would double every time it was half full. In most cases this is good when performance is important. However, if performance is not as important and memory is limited, then shrinking memory so that unused memory can be used. For my implementation of removing items, I did not implement a way to shrink or trim the size of my array’s capacity. This led to a lot of unused memory space.