Joseph Schinzer

Gabe Imes

Eric Bridgens

Data Structures Lab 7

The objectives explored in this assignment included pointers, ordered lists, as well as exceptions, polymorphism, and overloading operators. Pointers were used heavily in this assignment as we were creating an ordered list of pointers that reference the data that we were working with. In a professional setting, pointers in C++ development are extremely important because they allow you to work with data indirectly, as well as allowing helpful tricks such as dynamic memory allocation, allowing you to change array sizes at runtime. Ordered lists were the focus of this lab and prove very useful as a data structure, allowing you to keep information in order as it is added or removed from a list. In practice, ordered lists could be very useful for keeping track of information or records based on certain values, like dates for example. Additionally, we continued to work with exceptions, polymorphism and operator overloading in this lab, allowing us to add functionality such as creating different add methods, throwing under/overflow errors, and allowing comparison between 2 different ordered list objects.

The first version of the ordered list would insert the first item at the front of the list. To insert an item the method would start at the beginning of the list and compare each item in the list to the value being inserted to find where to insert the value. To remove an item from the list, the position of the list containing the value would be set to null, and the other items would be moved down one position in the list. An advantage of this version is that it is not very complex and simple to create. A disadvantage of this version is that it must iterate through the entire list to find where to store the item in the list, and when an item is removed each item in the list must be moved. We believe that this version will perform the worst out of the three.

The second version of this list also inserts the first item to the front of the list. To insert an item the method would start in the middle of the list and check if the value would go to the left or right side of the list, it would then check each item on that half of the list and find the correct position to insert this item at. The remove method is the same as the method of the first version. An advantage of this version is that it only needs to sort through one half of the list which makes it faster than the first version. A disadvantage of this version is that it is more difficult to create, and when removing an item from a list each item must be moved either forwards or backwards. We believe that this version will perform the best out of the three.

The third version of the list inserts the first item to the front of the list. The add item method works similar to task 2, but if there is a null pointer in the spot that the item is intended to go the item takes the spot of that null pointer. However, when an item is removed from the structure instead of reorganizing the list, a null pointer is left in the spot of the item. The advantage of this structure is that it does not require the list to be reorganized after an item is removed, and when an item is added it could be added to the spot of the null pointer. The disadvantage of this structure is that it is not organized. We believe that this version will perform the second best out of the three.

To simplify running our program we let the user input the size of the array. We also created a for loop to add or remove the items for the specified amount. The items being added would be randomly generated by a random number generator. To find the efficiency of each ordered list, we calculated the efficiency using the counts for each comparison and move and dividing their sum by the size. In order to get the results 100 times instead of running the program 100 times we created a for loop that would collect the results 100 times. Our results showed that the third version of the structure was the most efficient, while the second version was the second most efficient. Our prediction about the first version being the least efficient was correct.