Daniel Davies- Modules task

List.c

The first task I completed was the "list.c" module. Regarding our conversation on Friday, I initially did implement this slightly differently to the specification you set out in list.h- however, after revising my solution, I decided that my initial solution had some small errors in it (compared to your list.h). Hence, I actually ended up implementing list.c to your specifications. In the end, I was able to produce a fully functional doubly linked list, implemented to your specifications set out in list.h (or at least, that is my impression). The main points to my implementation are as follows:

- There are two sentinel nodes in the list, one at each end, represented by the first and last nodes. These are used as a means of user error control in the program.
- The user is (given the impression that they are) in-between two nodes at all times. As stated in the "start" and "end" functions, the user is also able to occupy a position before the start of the list or after the end of the list.
- The current node points to the node after the gap that the user thinks they are in, and so the sentinel node at the end can be pointed to, whereas the one at the start cannot.
- The items in the list are generic, but rather than using "void *item" I used "char item[]" in the node structure
- 60 auto tests were created to test each function (mostly individually, some together in parts). This was across multiple data types, but mostly focused on int.

Since the current pointer points to the node after the gap that the user is seemingly in, the general trend of the functions were that any of the "Before" functions operated on the nodes that are previous to the current node, while any "After" functions operated on the data inside of the current node. This means that the user is always convinced that they are in a gap between two nodes. (There are of course checks on whether an operation should be carried out, depending on whether the user is beyond the end/ before the start of the list- e.g, a getBefore at the start of the list is illegal). An exception to this is the insert functions- since either way an element is placed at the same position in the list, but it is only the location of the current node that changes, depending on whether the user wanted to go before or after the current position in the list.

Extensions

For this task, rather than completing one large extension project, I completed a number of smaller sub-extensions, in order to practice pointers/ other techniques we have recently learned.

The first extension task I did was extend the list.c module with two extra functions. The first of these, as you'll see in the list.c program, is "insertAfterAfter". This function came about when playing around with the list.c module, and I thought it would be useful for the user to have a function that could "skip gaps", and rather than just being able to insert a value into the gap before the current node, also being able to insert a value into the gap after it, since this requires fewer operations and could possibly be used as a shorthand for going forward and then inserting (this is how I used it during playing around with pointers and lists). I also included a “traverse” and “traverseBack” function in list.c, which I used for testing, which traverses the list- from the front, and from the back, respectively.

The second extension I completed was a small program that uses the list.c functions to make use of a doubly linked list. This is given in the “mainProg.c” file. This was a simple program I created, mostly to firstly see how structures would be passed into a linked list (using the generic char[] data type), and also allowed me to try out dynamically allocating space to a possibly increasing size array using “realloc()”. The program simply sets up a structure, “person”, which holds some details about somebody. Once data has been filled into this structure, a pointer to the structure is passed into the linked list and stored. From this, I was able to experiment with retrieving the data, and then creating a smaller sub function to calculate the average age of all of the people stored in the linked list (using a dynamic array to store the ages). This program also allowed me to further test whether the list.c functions were performing correctly.

The final extension I completed was implementing the “map.h” API using a binary search tree, as given in your recommendations for the extension task, which gave me the interesting opportunity to find out how functions are passed to other functions as pointers in c. I was able to implement all of the necessary functions in map.h to produce a fully functioning map, with keys and data values. However, in contrast to the map.h specifications, rather than both the keys and data of the map being generic, the keys of my map.h were made to be of type “int” (the data remained generic, of type char[]), as unfortunately I was not able to implement a solution with both fields as void pointers. A copy of my modified map.h has been provided. As always, the functions inside this module have been auto tested.