Section 1 Lesson 5

Section 1 lesson 5 sortLink

I Discuss new data structures, algorithms, or language features

We will examine the doubly linked list in this lesson. We also separate the data from the data structure. The data is no longer included inside the linking nodes, but rather in a node of its own. Now we can standardize our linked list tools. This lesson shows how sorting a linked list is more efficient than sorting an array. During the sorting process the data never moves, by using a pointer swap function only the pointers to a node change. Analysis shows Donald Shell's sorting routine has a worst case of $O(n^{3/2})$ and an average case of $O(n^{5/4})$.

II Describe functions and modules

There are two applications in this lesson. The first is shellSort.c. shellSort() takes as input a pointer to an array and the number of elements. It sorts elements at closer and closer intervals as the sort proceeds, comparing first farther elements and then closer elements. It performs as well as more complex algorithms, up to n=10,000 or more. Our main() function asks the user for the number of elements to generate for the array. This line would be trapped for incorrect input in a more complex application. The array is defined on the next line made possible by using the g++ compiler. Then the array is filled with random data. The shellSort() routine is called then we print out our sorted array of data.

The second application is sortLink.c, where we start with our data structure (formerly db), then we define a linked list node which points at a data node. Next, we define a few working node pointers. The function newData() allocates space for a new data node then fills each field with random data. We add another node to our linked list with addNode() while calling newData() to create its payload. A helper function is added, called swap(), which adjusts the pointers in the linked list. The data never moves. Our next function uses the simple selection sort algorithm to rearrange the links in our list.

main() seeds the random number generator from the present time and sets the employee count to 12. A while loop is used to create and add the correct number of nodes to our linked list of data. Next, we verify everything is in place with a while loop to print out the list's data. Call sort, add some white space, and print out the list again only this time in sorted order. Now we add a few more nodes of data to our list. Add more white space and print the new list which is partially sorted until the newly added nodes appear. This shows where these new nodes were added.

III Compile and run the code

- Open a terminal window
- Navigate to … lesson5 using cd (change directory in Linux or Windows using MinGW)
- Type **make** to link and compile the application called sortLink (in Windows it is called sortLink.exe)
- Type **./sortLink** (in Linux) or **sortLink** (in Windows) to run the application
- Read sortLink.c in your editor to see how the code creates the application's output

IV Describe code line by line

sortLink.c

The first 13 lines are fairly standard – the comment header and the #includes. Our data structure has three integer fields. The linked list node structure has pointers to the data node, the next node, and previous node. This allows us to keep most of our linked list tools intact. There is no need to change the data pointers by keeping the data node name generic, instead of calling it employeeData. You just change the data node contents and the list is none the wiser. Line 29 provides us with our working pointers to point to the head and tail of the list and another to scan the list. Tail could be eliminated, but that would require a few modifications to my working list code. It takes up little space, so I kept it even though it's redundant.

Our first function: newData() is a helper function for addNode(). newData() allocates enough space for a data node, and returns a pointer so you can fill that space with randomly generated data. Ages range from 20 to 70 years, salaries from $30,000 to $95,000, and seniority from 0 to 45 years. The next function, addNode(), allocates space for a node, then adds a filled data node to an empty or growing list.

The next function, swap(), is also called a helper function called by sort(). It swaps the data pointers without moving any data. In fact, the order of the list doesn't change, only the pointers to the data. You should write some code to print out the addresses of the nodes, and of the data, so you will see they don't change after a sort. What does change is the address pointed to by the data pointer, in your list nodes. By doing things this way, by not moving the data, you save a lot of time and make the code simpler. You are only swapping addresses not actual data, which is faster.

The sort() routine is pretty simple. But, with a data set of this size, it is good enough. If I were planning on sorting tens or hundreds of thousands of objects, I would choose a more complex and faster routine. Here the selection sort is best because it is easy to explain. Using two pointers you scan the list. The first pointer points to the starting location of the list. The next pointer scans the list, until it finds the smallest member of the set. Once it finds the smallest it swaps it with the value in the first location. Then the start pointer steps to the next node in the list, then the other pointer scans the list looking for the next smallest value. Lines 89, 90 and 96 you have seen many times before, it is how you walk a list to its end. Lines 86, 87, and 98 perform the same functions for the outer loop. You are done once your start pointer reaches the end of the list. The list is now sorted on your key field, which in this case is age. Change line 92 for a new key field or change < into >= to sort into reverse order. As I mentioned, this is a very simple sorting algorithm to understand.

Next we come to our main() function. It begins by setting the seed for the random number generator. In this case we use the time_t variable t. In line 105, we use the & operator to get the address of t. srand() then points at the address of t, then uses the value to seed the generator. The loop, defined on lines 108 through 112, creates our linked list of random data. Then we walk the list printing out all the data stored in it, with lines 115 through 123. Set the pointer to the head of the list, walk the ep pointer until it returns NULL, print the three data fields and the address of the data set, then point to the next node in the list and repeat.

Line 125 calls our sort routine to walk the list, ordering the data by increasing age. Lines 130 through 138 walk the list again using exactly the same code as shown in lines 115 through 123. I inlined the code here but it could easily be built into its own function, called from both places; plus the 147 through 155 location too. I just copied and pasted to keep all of the code easy to read, and easy to follow. I would not write my code this way unless I was using it to explain or demonstrate something. I would make a print() function and call it three times. Lines 140 through 143 are used to add more nodes to the list. In this case I used a for loop but the first time I created nodes I used a while loop. They do similar things, they just do them a little differently. Your choice.

Our main() created 12 nodes, printed them, sorted them, printed them again, added 5 more nodes and printed once more. This contrasts the previously sorted list with those at the end which were not sorted but added later. Feel free to rewrite this section to add more nodes, sort them, and print them. Change the sort routine to use different key values, then add thousands of new records to the list to check how long it takes to sort a short list as compared to a long list.

There is a function at the end of the code file which is not incorporated into the main function. Deletion is not used in this lesson but I keep the code around for later. It is tested and works well. deleteNode() takes a pointer to a node in your list as input. Four cases need to be satisfied: delete the only member of the list, delete the first node of the list, delete a node in the middle of the list, or delete the last node of the list.

Lines 171 through 177 cover the single member case. If your next pointer is NULL and your previous pointer is NULL, you know there are no other nodes. You free the data memory (line 173), free the list node (line 174), then print a friendly note telling the user his list is empty.

Lines 182 through 185 show the case where you want to delete the first node in the list. here->prev points at NULL so it's at the beginning. head points at the beginning of the list so you aim it at the node after the first node with head = here->next. Then you clean up by pointing prev back to NULL, instead of to the node you are deleting.

The if case, lines 179 and 180, is when you are not pointing at the beginning of the list. here->prev->next used to point back at itself, but you want to skip over this node, and point to the one before with here->prev->next = here->next.   If you are not at the beginning (lines 182 through 185) then set the head pointer to the node after the first one (head = here->next) and fix its previous pointer (head->prev = NULL).

Lines 187 through 193 handle the case where you are going to delete the final node in the list. Let's look at the else case first. Set the tail to point to the node before the last one (tail = here->prev). Then set the end of the list to NULL (tail->next = NULL). The if portion handles the case when the chosen node is not at the end of the list. Before the deletion here->next->prev points back at here but after the deletion we want it to point to the node prior to the one we are deleting (here->prev->next = here->next)

Make sure to draw some node insertion and deletion diagrams for yourself. If you want to become facile with linked lists and pointers, you will need pictures with nodes and arrows. Trust me, it does not seem to make sense, until you have seen how it works with diagrams. You need to be careful not to delete a link before you have connected both the next and prev links, around the node to be deleted. If you do things in the wrong order you will wind up pointing somewhere odd, and you will have a node floating in space with no pointers to it. That is called a memory leak. This node was allocated from your heap memory but it was not returned when you were done using it. Lines 173 and 174 are examples of freeing the data memory, then freeing the node memory.

Because I placed a return statement at the end of the almost empty list case the two free commands of lines 173 and 174 do not conflict with lines 195 and 196 at the end of the routine. Those two lines cover the end case, the beginning case, and the middle cases. You could easily rewrite this delete function to be smaller but less easy to read. This routine covers the middle node case twice so there is a little redundancy. But it has worked well for me for years, I am not going to change it now unless absolutely necessary. You can rewrite and test it if you like for homework.

shellSort.c

Here we have an example of a function with its associated test setup. This is how I implement and test algorithms before I put them into another application. shellSort() is a faster sorting algorithm which you can incorporate into sortLink.c for comparison to the present selection sort algorithm. shellSort() is more complex but much faster than selection sorting. Where selection sort works in O(n^2) time, a Shell sort can work in O(n lg n) time. Remember lg n is a log base 2 function. Notice line 18, where your increment value changes by ½ at each iteration. This is your logarithmic reduction. Shell sort is a divide and conquer routine. It sorts ever decreasing size lists. In this case, the routine is sorting a simple array.

Our main() function asks the user for the number of elements in the list, creates an array of that size, and then asks the user for data to be placed in each array location. I changed that so it adds the numbers randomly, instead of having the user input each number separately. It is just a test of the routine. Once our data is in the array we call the sort routine and print the sorted list.

I often write temporary wrappers around functions, so I can test them independently of the program I am writing. These test harnesses come in handy when you want to debug an algorithm or data structure thoroughly. Create data sets with problem cases to test whether the algorithm can handle them correctly. Fix the algorithm until it does, by adding error traps, and improving your code.

If you want to use ShellSort() inside of sortLink() you will need to rewrite the exchange and comparison portions of the routine. Instead of moving array values you will change link pointers. The comparison (the if statement on line 25) which currently reads

```
if (temp < array[j-increment])
```

will need to be changed to something resembling

```
if (ptr->data->age < start->data->age)
```

Once you have rewritten ShellSort() you can use it as your sort routine inside sortLink.c. Then test it on larger data sets.

V Describe makefile line by line

Here is a new twist on makefiles. I put a conditional statement at the beginning so you could choose which of two apps to compile and link. If, on line 4, XX is equated to anything, rather than nothing, sortLink is created. Otherwise shellSort is made. Once again, I have chosen g++ as my compiler on line 5. This allows me to use a few C++ features in C application code. Line 7 is where the conditional value XX is checked and the compile path is chosen. In lines 9 through 13 sortLink is compiled and then linked into an executable. Else (line 15) lines 17 through 21 are followed, making shellSort from its application code. Think of line 23 as a closing bracket in a regular if else statement. Lines 26 through 33 are used for cleanup checking the conditional again for which path to take.

VI Deeper

This lesson builds on the last two, adding a new data structure: the doubly linked list. Normally, when you want a list, you don't use an array. It is not easy to manipulate as you saw with the array sort in lesson S1L4. This lesson shows you how easy it is to use doubly linked lists to hold, arrange, and access the data. I have also abstracted the data away from the data structure, by using two structs. One for the data, the other for the list structure. It really helps during the sorting step. By making the list tools more generic, you will reuse them more frequently, allowing you to use your time more efficiently.

Starting with line 15 we build a data structure called data and currently filled with ints: age, salary, & seniority. Later we'll create structures which hold strings, but avoid their complexity for now. I am trying to show you how data structures work with algorithms, and making it simple. Complexity can come later.

On line 22 we build the data structure itself with a struct of pointers. One pointer to the data, while the other two point to the next node and to the previous node. You'll notice I have a commented line with another data structure: a tree. We will use it in lesson S1L6.

I am showing you another technique for writing code. In this style, the main() function comes last. All helper functions precede the function which calls them. By writing code in this order you don't need to declare things up front, because that happens automatically due to the order in which you wrote the code. This is not the normal way to leave the code in the end though. This is a way to write code on the fly and not have to deal with the fiddly bits. Later you can put the main() function right up front, declare each function, and put all that into a header file, along with the variables, e.g. “sortLink.h”. Makes much cleaner code, but writing it this manner lets you see every part in a 'flat' fashion. Once code is mature I would advocate it be written in the latter style and use a special header file.

Side note: #include statements are written two ways

```
#include <stdio.h>    and    #include "sortLink.h"
```

The first notation is for header files built into your C library. The latter is for include files written by the programmer. Annotating it this way helps you differentiate between the two.

Line 33, newData() allocates the memory for your new data and then creates the data from random choices. This function returns a pointer to the data for the next function to use.

Line 49, addNode() allocates the space for a new node, calls the newData() function to create the data to fill it, then adds the new node to the doubly linked list. If it is an empty list, the pointer adjustments are simpler. Adding to an existing list is one pointer adjustment more difficult. Each new node is added to the end of the list.

There are two pointers set for your use when you return from addNode(), head and tail. They point to the start and end of the list respectively. This way you can walk the list from either end. Two way linking aids you when you want to add a new node in the middle of the list.

However, I have written the new data structure to be independent of the linked list structure. By abstracting away the data from the data structure I have made the functions simpler. Thus, the nodes are always in the linked list data structure, while the data each node points to remains in its own chunk of memory. The same memory location where it was first created. Only the pointers pointing to that data change. That really saves time and makes the code simpler. If you want, you can add a printf()

line to show the addresses of the data, both before and after the sort. You will see they do not change. I just added this

```
printf("data is here %p \n", ep->data);
```

The %p gives you the address in the correct format.

We chose the selection sort algorithm again. You can see how simple it is, starting at line 70 with the swap() function. To swap two items requires a third point of reference so I made the new pointer *temp as a helper. C = A, A = B, B = C swaps A with B.

I am trying to show you a number of things in this example. Data structures with the doubly linked list. Abstracting data from the data structure. The repeated idiom of walking a list. A double pointer i.e. ptr->data->age to retrieve a field of data, or here->next->prev = here->prev during deletion of a node. Sorting using pointers. Using functions instead of inline code. In fact you could make a function out of the walk-the-list-and-print routine quite easily. Then call it before and after the sort. It would make main() more compact and readable too.

Make sure to read the comments. I added more so you could better understand each chunk of code.

I included deleteNode() so you could see how to free up memory which you have allocated. Memory leaks can be a problem if you are not fastidious about balancing each malloc() with a free(). I try to write this sort of code in tandem so I don't forget to give back what I have requested. Garbage collection, the cleaning up of memory fragments the OS gets tossed when apps are done with them, is an open area of research. Remember, there are two kinds of memory you can access: heap memory and stack memory. When you use malloc() you are grabbing a chunk of memory from the heap. When you create an array during compilation you are using memory in the stack space. When you write a recursive function (more of them coming up with trees) you create stack frames, another use of stack memory.

When your operating system runs your application it assigns a specific amount of memory space to it. The stack grows from one end of that memory space and the heap is allocated from the other end. When those two memory pointers overlap you get real problems. You can crash the computer or send weird stuff across the ‘net. Not a good thing in any case, so watch for memory leaks. Work in parallel by matching memory allocation with each memory deallocation, they should balance each other.

Working in parallel is also good advice for braces and parentheses. My editor gives me two matched braces when I type the opening one. Same with parentheses. By getting the editor to do this for me prevents many errors. My editor also colors the matching brace or parenthesis when I click on the other. That helps me find the structure, and make sure all the braces are correct. So, make sure to type () or {} or "" when needed just for clarity and error avoidance. By using the arrow keys you can get back to where you want to add things, quickly.

The makefile has a little something different too. It is a conditional makefile. I was working on two chunks of code in the editor at the same time and was too lazy to create a new folder with all the files and another makefile too. I just wanted a quick and dirty makefile for what I was doing. So I looked up how to write conditional make rules. XX is my conditional variable, which I can set to anything or nothing because ifdef XX is only looking for existence of a value, not what it is set to. “Does XX have a value or doesn't it?’, is what ifdef is asking. Kind of existential :) Since I added the ss, the file compiles then links sortLink. If you eliminate ss then shellSort.c is built instead.

By the way, I created a few output files by the simple expedient of using the shell to work for me thusly:

```
sortLink > sort.txt
```

That line runs the code I just compiled and saves its output to the file sort.txt That way you can run the example with much larger count size, and still see the output. Like I said Unix was designed around text files, text input, and text output. Makes the OS very useable and simple. By installing MinGW you gained that flexibility, plus you get to use Windows simultaneously.

Next, I am going to show you how to sort with a tree structure, plus a little about how to save and open files of data. It is a lot like printing except you use fprintf() instead of printf(). Reading a file is a bit more complex but I'll get there too. We've covered a bit of sorting; some searching will come in handy soon too. Writing code is all about three things: data, data structures, and algorithms. Once you understand that you are far ahead of most folks. Knowing which data structure to use and which algorithm requires being exposed to them, then picking the wrong one. I think you learn more by making mistakes than you do by being correct all the time. Perfection is boring :)

VII Assignments

1. Change the direction of your sort in sortLink.c
2. Change which field you’re sorting on in sortLink.c
3. Incorporate the deleteNode code in sortLink.c
4. Sort the list then delete the 5th through the 8th node. Add more new nodes. Then print out the new list.
5. Take the shell sorting code from shellSort, modify it to sort the linked list in sortLink. Run timings on the new code from n = 10 to n = 100,000
6. Rewrite and test the delete function.
7. Add time commands to check the speed of sortLink on various number of elements. As the numbers get higher you may want to comment out the print statements. You are just comparing the speed of the algorithm vs the size of n.

VIII Links

https://en.wikipedia.org/wiki/Shellsort

http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L12-ShellSort.htm

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html

https://en.wikipedia.org/wiki/Linked_list

https://www.interviewbit.com/courses/programming/topics/linked-lists/

https://medium.com/basecs/whats-a-linked-list-anyway-part-1-d8b7e6508b9d

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

https://en.wikipedia.org/wiki/Big_O_notation

https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

https://medium.freecodecamp.org/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4

http://web.mit.edu/16.070/www/lecture/big_o.pdf

https://yourbasic.org/algorithms/big-o-notation-explained/

https://www.geeksforgeeks.org/delete-a-node-in-a-doubly-linked-list/

https://www.geeksforgeeks.org/delete-doubly-linked-list-node-given-position/

https://www.tutorialspoint.com/learn_c_by_examples/remove_data_from_doubly_linked_list.htm

http://cs-people.bu.edu/jalon/cs113-01/dlistlab/dlist.html