Section 1 Lesson 7

Section 1 lesson 7 search

I Discuss new data structures, algorithms, or language features

This lesson demonstrates how to use data structures for their best properties. Linked lists for their ease of adding nodes. Building trees to keep a list sorted. And arrays for the ability to perform a quick binary search. By creating a hybrid node with pointers for both lists and trees the best structure was easy to choose.

II Describe functions and modules

Include the libraries necessary to fulfill the functions used in this application. Define the size of the employee list at the top of the code. Create the data node structure, then a hybrid data structure node capable of handling both linked lists and trees. Define a data node array for the search routine. Define a newData() function to handle the data portion of the problem and define a newNode() function to take care of the hybrid data structure part. Define an addNode() function to build a linked list along with its associated printData() function. Define the treeInsert() function to build a tree from walking our linked list of data with its associated print function. The function putInArray() walks the tree (which is inherently sorted) putting the data into a data array for our binary search algorithm. Since the binary search algorithm requires an indexed list I used the simplest form – a data array. The writeData() function walks the linked list and saves the data on a disk file. The addXNode() is a helper function used by readData() to build a new linked list from the data file. The aforementioned readData() function reads the data file, allocates the data and the data structure nodes, fills their fields, then uses addXNode() to build a new linked list.

main() begins by seeding the random number generator. Then it sets up pointers to two lists and a tree with one working pointer, ptr. We have a for loop create a singly linked list of employee data. Next we walk the list, and print it out as a sanity check. We write the data to a disk file then use a while loop to create a tree from our linked list. Print some white space and print the data in the tree. This will demonstrate that the data is now in sorted order. Next we build an array of employee data from the tree, which is naturally in sorted order. Now that we have a sorted array of employee data, we can run the binary search algorithm on it. The user is asked for a number (in this case it is searching for age). The function binSearch() returns a pointer to the first correct employee record. The query loop runs until the user inputs a -1 as a stop value. Next, we read the data from the file we created earlier and print it out. The printData() function has the capability of showing where the data is located in

memory. Compare this print out to the very first one and you will see the data is now in a different place. This is because it has been copied, not simply pointed at.

III Compile and run the code

- Open a terminal window
- Navigate to … lesson7 using cd (change directory in Linux or Windows using MinGW)
- Type **make** which will link and compile the application called search (in Linux, in Windows it is called search.exe)
- Type **./search** (in Linux) or **search** (in Windows) to run the application
- Read search.c in your editor to find how the code creates the application's output

On my Linux box the files are located at:

```
kevin@Hermes:~/files/share/BD_2nd_Edition/1_tools/S1L7_search
```

On my Windows box they are here:

```
U:\BD_2nd_Edition\lessons\1_tools\S1L7_search
```

Your file will be located wherever you unpacked BD_2nd_Edition.

IV Describe code line by line

I expanded the comment section in this lesson to describe the steps followed as the application runs. Then I #included our now familiar libraries. Starting with line 19 you see a #define statement you can use to change the number of employees you create. #define statements are a convenient way to designate a fixed value in a location easily accessible to the programmer. #define statement are either at the very top of files right after the #include statements or they are located in the separate header files called. #define statements are traditionally in all caps, so when you see them interspersed throughout the code, you can easily recognize them. The value is set in stone at compile time; there is no way to change it via your app.

Line 21 has your application specific data node structure. On line 30 represents a single node of the data structure. In this case I have included a pointer to the data, a pointer for the linked list, plus two links for the tree structure. This hybrid structure lets me access the data in multiple ways. Next, on line 37, is an array of pointers to data nodes. This is filled from the tree then used during the search algorithm.

Instead of the inline creation of nodes, I now have functions for all the steps. First, on line 39, you create a new data node and fill it with random data. This function returns a pointer to the new node, so it can be put into a list, or a tree of structure nodes. The next function takes the data node and creates a structure node around it. This is used by the addNode() function (line 64) to make a linked list. I wrote a printData() function to make the print out easier and consistent. Got tired of typing all the -> too.

The function treeInsert() is used by main() to read the singly linked list, then string the nodes together into a tree. When the function has finished you can scan the data either as a tree or as a singly linked list. The data does not move it just gets threaded differently. But, the neat part is, by scanning the list into the tree you now have a sorted list. Some algorithms build data structures to sort huge amounts of data, on each time step. Tree are a valuable data structure. I commented out the lines which change the sort index. You can now pick age, salary, or seniority with lines 113, 114, and 115.

I left some of my debugging lines in place. It is always nice to know if your data structures are working correctly. One lost pointer and you're screwed.

printInOrder() is a function which walks the tree from the smallest (leftmost) element to the largest, printing out data along the way. It is recursive but should be quite straightforward.

putInArray() walks the tree once again, but this time, it puts its elements into the struct node *array[] we created on line 37. This lets us use the binary search algorithm, starting on line 168.

Binary search is a divide and conquer technique. With each step, it solves the same problem on half of the remaining data. This repeats until the answer trivially pops out . Once again, this function is recursive and self explanatory. The reason you put the data elements into an array, is because you need the indices to be able to manipulate them mathematically. Like so: middle = (start + end) / 2 to find the middle. You just can't do that with a tree or a linked list. But it is difficult to sort using an array because you need to rewrite all of the data during each of the moves. However, if you abstract the data away from the structure like we've been doing, it is not so costly. Trees and linked lists are much cooler.

Now we come to the I/O functions. writeData() is pretty simple. Just walk the list pointed to by head and fprintf() the data you want stored. First, save the count, so you'll know how many records to read. Otherwise, you would need to look for the end of the file. Whichever way you like is fine. I like to know how many elements I need to read right up front. Belt and suspenders!

Line 198 opens the data file "search.dat" in a write mode. The "w+" designates writing with the possibility of rewriting. In this case the next write will over-write the previous file. fopen() returns a file pointer fp. This is how you walk through the stored file. You can rewind, or fast forward, to

wherever you want to read. Normally I don't care to do that, I just read everything from beginning to end and not worry about where fp is pointing. Fancy always bites you! But memory used to be tiny and expensive, so we needed workarounds just to get by.

On line 209 we have addXNode(), for add external node, as compared to addNode() which created its data from thin air using random numbers. addXNode() takes the file's data and puts it into a singly linked list. It is used by the next function, on line 234, readData(). Because you can only save the data, not the data structures, you need to rebuild the nodes from the raw data.

Instead of creating functions to do this, I took parts of other functions, modified a function into a new one, and added new lines of code. This is my normal process for writing code. First you have chaos and then you massage it into order. When I am done building I normally have a few cruft.c files lying around with my notes and the scaffolding necessary to build the final app. A professional chunk of code has a lot of conditionals included so you can run the code for separate machines, various OSes, and for diagnostics.

Now we come to main, where you will see how all the parts are used.

lines 270 - 274 set up the variables and pointers.

line 276 is a simple for loop creating a singly linked list out of random data. addNode() calls newNode() which calls newData(). They allocate and fill the two types of nodes: data nodes and data structure nodes.

Next we walk the linked list printing data as we go.

Line 288 has us saving the raw file. You'll notice the id data field is filled with a sequential number. When you walk the linked list it will be in "id" order.

Line 290 starts walking the linked list again, only this time the data is inserted into a tree structure. Simply by inserting nodes into the tree you are sorting them on the index you choose.

Line 298 calls printInorder(root). This walks the tree from the leftmost element to the rightmost element. and shows you they are ordered by age (in this case).

Line 300 walks the tree we just created and puts the data into an array. This allows us to use the binary search algorithm.

On line 306 & 307 the user is asked for a number. The app wants to know which age you are looking for. On line 311 the data you entered is sent to binSearch() to see if it is in the array. If you find the index of choice, that element is printed out by printData(). Otherwise, you get the pithy message "Index not found".

Lastly, starting on line 323, you read the data you stored to build a new linked list. readData() returns the pointer to the head of the new list. On line 325 you use that to walk the new list and printData() it onto the screen. You will notice, that now the memory locations have nothing to do with those of the original data. That is because you had to allocate new memory to make new nodes.

V Describe makefile line by line

Line 3 chooses the g++ compiler.

Lines 5 through 9 compile and link the source code into the executable search.

Lines 11 through 13 cleanup for us.

VI Deeper

This is a lesson about using data structures to manipulate data. We start by filling a singly linked list with data. Then we build a tree data structure from our list, without changing the original list. By building the tree, we automatically sort the list. Now we walk the tree and fill an array from each node. The array data structure allows us to use a simple search technique, called a binary search, on the sorted data. Since you can index each location of an array, you can compare field values to your key by partitioning the array. Remember the index values of the first and the last array locations. Now divide to total length in half, for your first 'guess'. If the data value equals the key value you are done. If the key is smaller than your 'guess', then recursively search the first half of the list. Otherwise recurse on the second half of the list. This method keeps splitting the remaining list in half until you find the right record, or your top and bottom index values 'cross'. The general technique is called divide and conquer. In this case, you eliminate half of the remaining list at each step. Instead of scanning all N records, you scan many fewer; $\log_2 N$ to be exact, because you are halving the list each time, the logarithmic base is 2. In some cases the divide and conquer technique can give $\log_4 N$ or better speed up. For this application we are using a one dimensional list, and breaking it in half at each step. If your data is in a grid, you could use a 2D version of divide and conquer, called a quad-tree. And, if you wish, you could move to the 3D world and use an oct-tree representation for $\log_8 N$ speed. Indexing for the latter two cases is more complex, but if you can narrow in on the local neighborhood you will see faster results.

I could have used separate data structures for the singly-linked list and the tree but thought it was better to hybridize the two. By creating a structure node with both linked list pointers and tree pointers I can thread the data however I like. Our first data structure change, sorts the list while the next structure change allows simple searching. As a side task, I wrote some saving and reading functions so you can store your creations on the hard drive. I am still using the 'main() on the bottom with the functions preceding it' style. You will see it has become cumbersome, due to all of the functions I've created.

After you have digested this lesson, you will understand why it is good to abstract the data away from the data structure. In this app, the data never moves until you write and then read it again. In each print out I have included the address of the chunk of data. If you took a hex debugger to main memory you could find each little bit of data in each structure at those addresses.

Binary search algorithm

The binary search algorithm is easy to write. All you really need is an indexed list of data. The index is the key. Imagine yourself looking up a word in the index of a textbook. The index references you to a single page number, or more. Think of the page number as an index into your list. You say to yourself, page 231 must be a little before the middle of this book, and your fingers open the book about half way and you check the page number. Then you use your thumb to move another chunk of pages in the desired direction. When you get very close to your chosen page, you leaf through them one by one by one until you reach your goal. Well, the binary search algorithm is very close to what you just did with your fingers and brain. Only it is binary so you go ½ of the way then ¼ of the way then 1/8 of the way iteratively, until you reach your index. If your data is evenly distributed throughout the entire range, then your search is very fast ($\log_2 n$). If the data is unevenly distributed, it takes a few more iterations before you find your goal. However, it is well worth the extra effort to build the index to your list. With a sorted linked list you would add an array keeping track of the memory location of each node. Putting your data temporarily into an array is probably the simplest way to build your index. I usually use the simplest method to lessen errors, and so I don't confuse myself.

VII Assignments

1. Uncomment lines 213, 214 & 230 to see where the data is located in memory. You can see each node is right where you created it. Only the pointers to other nodes change in value. This is why using a pointer to the data instead of including the data within the structure of the list or tree is so much faster. One step of abstraction saves you time and makes your code more reusable.
2. Change the search loop to ask for age instead of just a number.
3. Change the tree insert function to sort on another field like salary or seniority. Basically just uncomment one line and comment the previous one.
4. Add more fields to the employee record and modify all the print statements accordingly.
5. Rewrite code so addXNode() is not used.

VIII Links

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

http://www.algolist.net/Algorithms/Binary_search

https://www.geeksforgeeks.org/binary-search/

https://www.cprogramming.com/tutorial/cfileio.html

https://en.wikipedia.org/wiki/C_file_input/output

https://www.tutorialspoint.com/cprogramming/c_file_io.htm

https://stackoverflow.com/questions/24892270/beneftis-of-hybrid-data-structures-on-efficiency

https://www.researchgate.net/publication/220807459_Hybrid_and_Custom_Data_Structures_Evolution_of_the_Data_Structures_Course

https://pdfs.semanticscholar.org/a0d3/306999eacc7fab93955eb1223eef10312708.pdf

https://www.cg.tuwien.ac.at/research/publications/2016/Labschuetz_Matthias_2016_AHD/