Group 7

CSCI 313

HW 1-- Question 2: Binary Search

First, certain header files are included to allow for the usage of random variables, for filling our array and linked list, and measuring execution times (stdio.h, chrono, etc.), so we can measure how long each binary search takes to execute. Then code for a structure for a node is created so a linked list can be made. The empty pointer to 'next' is for the next node in the list. Since there is no 'next node' yet, it is set to "{nullptr}". The 'data' is for the data value to be passed into the node, which is also set to no value. Next, we establish functions for inserting nodes into a linked list, a recursive binary search, an iterative binary search, a linked binary search, and a function for performing all the loops.

In the main function, a random seed is called with srand(time(nullptr)). Without this, we would get the same random numbers when rand() is called. Then a constant int is declared to be used as the size of the array and linked list. The sizes used are one million, ten million, and a hundred million, and these sizes are passed into the loops function.

Since our array will be in the millions, we create an array on the heap instead of on the stack (int* array = new int[size] vs. int array[size]). We use heap instead of stack because the stack is more limited in size, so trying to fill a stack array with this many elements will crash the program, as I've had to deal with many times before figuring out what was wrong. We also need a 'delete[] array' line later for deallocating memory after we are finished using the array, which will be after the second binary search using the array.

A for loop is used to set random variables to the array. Since the elements must be higher than the previous, we set the first element outside of the for loop with rand(), then we can set the later elements to be higher than the previous with array[i] = rand()+array[i-1]. A random target number to search for within the arrays and linked list is created.

The clock is called before and after the recursive search to record the execution time of the search, then it is outputted. The results of the search are outputted first; was the target number found in the array? An if statement determines this. Then it outputs how long it took the recursive search to execute. The same thing is repeated for the iterative search; the results of the search are shown, and then the execution time.

For the linked list, I was unable to find a way to append over ten thousand nodes without the program crashing. So I decided to try a different route in testing one, ten, and one hundred million nodes. I divided the size by ten thousand for the linked list so it would have sizes of a hundred, a thousand, and ten thousand, then repeated the linked search ten thousand times to simulate searching those millions of nodes.

First, a linked list is declared with struct Node* head = new Node(). Then we directly pass in a value into the first node of the linked list with 'head->data = array[0]' because, if we didn't, then the first element in the linked list would be 0. A for loop is used to fill the linked list, and to make things easier, the data of the nodes all match the corresponding array elements. The linked search is called and timed, and then the result and execution time is outputted.

The insert node function creates a new node, passes in the new data, and makes it point to null, since there's not going to be a node after it. A while loop and a temporary node are used to traverse to the end, where the new node is then appended.

The recursive search checks if the leftmost element is less than or equal to the rightmost element. If so, it calculates the middle of the array, then checks if it is the target. This is why the elements are in increasing order. If the middle is the target, it returns true. If not, it recursively calls itself, altering the parameters so that the array will halve itself over and over until it either finds the target in the middle or doesn’t, in which case it returns false. The iterative search works very similarly, except it uses a while loop to halve the array until it finds the target in the middle (or doesn’t).

For the linked search, it uses a while loop to check if the target is within the linked list. Finding the middle of the linked list works differently than the array; toMid and toTail are created to find the middle. ToTail traverses to the end at twice the speed as toMid, so that when it reaches the end, toMid will have stopped at the middle. The node reached by toMid is then passed into mid, then the function finally checks if the middle is the target, and performs the same checks as the previous two binary searches. It changes its leftmost and rightmost value depending on if the middle is less than or greater than the target, and then loops until it gets an answer, which is either true it’s in the list, or false it is not.

Running this program multiple times shows that the iterative search is most definitely the fastest among all three, consistently taking less time than the other two functions whenever the program is executed. Second fastest is recursive, and then the slowest is the linked search by a very large margin.