SP Exercise 1a – Report

Eva Kupcova

2479253k

My solution to the Assessed Exercise 1a Website monitoring is fully working. I admit, there are few memory issues which could be improved but I have not registered any memory leaks.

I used the array-based implementation of a stack for the implementation of the iterator. The size of array structure in C cannot be changed and that leads to unnecessary use of memory in this case. As we don't know the size of the input file, I implemented the array with the size of (maximum long value/size of the TLDNode). This size is definitely not needed, as the maximum size we need is the count associated with TLDList. Even this size would probably be an overestimate as the count represents everything added to the list, not just the number of Nodes with unique domains. Therefore, we could assume the count of the list is the worst-case space complexity for the iterator. However, I must have used more space as the program does not know the count before the implementation of the array.

Another possible implementation would be a stack based on the linked list principle. This way would allocate new memory space only when a new node is added and would free it immediately after deletion.

In the **date.c** file, I only implemented the functions given by the **date.h** file. The ***date_create*** function creates a new Date object if it was entered in the correct format. However, it only checks if the slashes ('/' symbols) are correct. This implementation does not check if the month and days are within their limits, which could be improved. Date stores the day, month and year values as integers. The ***date_duplicate*** function creates a new Date object with the same attributes as the parameter. The ***date_compare*** function compares two given Date objects based on year, month and finally day. Lastly, the ***date_destroy*** function frees the memory space allocated to a given Date object.

In the **tldlist.c** file, I implemented the functions given by the **tldlist.h** file and a few helper functions. The ***tldlist_create*** functions create a new List object with the initial count 0. List stores its root, the count of all additions, and begin and end date restrictions. Its allocated memory space could be freed by the ***tldlist_destroy*** function. This list is implemented through a binary search tree, which stores the domains based on their alphabetical order. This will help reduce the time complexity of accessing a certain node or adding a new node to a list. However, it does not help in the worst case where the input would be already in alphabetical order and the program would therefore need to traverse the whole tree to add a new Node.

The ***tldnode_tldname*** and ***tldnode_count*** functions return the list associated with the given node and its count respectively. The ***tldlist_add*** function adds a new Node object with a given hostname to a given list. Firstly, it checks whether the date is within the date constraints of the list. Next, it separates the domain from the given hostname and increases the count if it is already present in the list. If it is not, it creates a new Node object and stores it in a tree based on alphabetical order. We can access the count of all additions through the ***tldlist_count*** function.

As mentioned above, the list iterator is implemented through an array-based stack data structure. It traverses the list (in a form of a binary search tree) from the root to the left-most child, which creates a list in alphabetical order. Returning to the root, if a Node has a right child (if there exists a domain that is alphabetically larger than given Node but smaller than its parent), it traverses also through its children. This iteration is used recursively. The ***tldlist_iter_create*** function creates an iterator and its allocated memory space can be freed by the ***tldlist_iter_destroy*** function. The ***tldlist_iter_next*** function accesses the next domain in the list but also starts the recursive traversal on its children.

Moreover, I implemented the helper functions associated with the array-based stack and node initialization.

To sum up, I am overall satisfied with the quality of my solution. The code works as it should and provides correct output. There is space for minor improvements, such as changing the array-based stack implementation to linked list to reduce the space complexity. Additionally, the checks for correct input of the Date object could be improved.