CHUA, KYLE MATTHEW C. S19B

Hash Table is the data structure of my program, within this structure, there are linked lists to store 2 or more words in 1 memory space in universe whenever there is a collision. For my search algorithms, I used hashing for keys without wild cards, while I used linear searching for words with wild cards.

For initializing the words in universe, I passed the address of the universe, which I have declared as struct listTag \*, as the parameter to my initialize function. Inside this function, I dynamically allocated 154459 struct listTag memory spaces. 154459 is the size because I read in a hashing guide that it is better to have a prime number as a hash table size to reduce collisions, and even though the size is smaller than the known required size, the size doesn’t really matter because I can store 2 or more words in 1 memory space of universe using linked lists. After malloc, I initialize all the allocated memory space from 0 to 154459 wherein ((\*universe)+i)->element gets “-1” symbolizing that the memory space doesn’t have any words occupying it yet, and ((\*universe)+i)->pNext = NULL.

Then, to store all the words from enable2k.txt, I created a while loop that will only terminate if there is no more words to scan from enable2k.txt. Within the loop, I will get the Hash address of a specific word by calling the function getHash and pass the word as parameter, the function will then return the address where the word will be stored in the structure and it will be temporarily stored as an int value in the int variable hashAdd. It will then check if ((\*universe)+hashAdd)->element is empty by comparing if ((\*universe)+hashAdd)->element and “-1” is equal, if true, the word will be stored in ((\*universe)+hashAdd)->element. Else if pNext is not NULL, we will position the temporary struct listTag \* variable pTemp to the last node in the linked list, after positioning, it dynamically allocates a struct listTag memory space and store it in pTemp->pNext, then store the word in pTemp->pNext->element and pTemp->pNext->pNext gets NULL. Else, the program will dynamically allocate a struct listTag space to ((\*universe)+hashAdd)->pNext and store the word to ((\*universe)+hashAdd)->pNext->element and initialize ((\*universe)+hashAdd)->pNext->pNext to NULL.

For my search function, I have used 2 methods, hashing if word doesn’t have any wildcards and linear if word has wildcards. Inside my search function, I first check if word contains any wildcard by checking if each element of word is ‘?’ or ‘\*’, if true, I increment questNum and astNum respectively, then store each element of word to Str30 variable key. After checking and storing the elements of word to key, I convert the string key to lower case by using the function strlwr from string.h. Then I check if quesNum is 0 and astNum is 0 meaning there are no wildcards in key and if true, I go directly to the address where key is stored by (universe+getHash(key)) and check if element is equal to key. If true then I create a node that will contain the key and storing it in pList by calling Create\_Sort\_Node, passing the empty list pList and key as parameters. Else if (universe+getHash(key))->element and key is not equal, then I check if pNext is not equal to NULL, I then check all the nodes of the linked list in (universe+getHash(key)), if I have found the word, I stop searching and call the function Create\_Sort\_Node to store key in a node and assign it to pList. If questNum or astNum is not equal to 0. I check if questNum is greater than 0, if true, then I linear search by universe+i where i starts from 0 and ends at 154458. pTemp gets universe+i, if pTemp->element is not “-1”, then I check every node of the linked list. I first check if the length of pTemp->element is equal to the string length of key, if false, I check the element in the next node, else if true, I check every element of ptemp->element and key by a for loop. I only check if key[j] is not ‘?’, if pTemp->element[j] and key[j] is not equal, then I end the loop by break;, else it will continue to check. If key[j] is ‘?’, I just ignore it and proceed to the next element. After this if j is equal to the string length of key, this means that I have checked every element of key which means it is true, I create and sort the node containing the word of pTemp->element by calling Create\_Sort\_Node, passing pList and pTemp->element as parameters, and assigning this to pList. And I proceed to the next word to check. If questNum is 0, then I check if astNum is greater than 0. Same for questNum the only difference is how I compare words. A while loop is used to search every element of key. Inside this loop, if astNum is still greater than 0, I only compare the element of pTemp->element and key if key[j] is not ‘\*’, else if key[j] is ‘\*’, I decrement astNum and proceed to the next element of key. If key[j] is not ‘\0’, k gets the string length of pTemp->element. Else if key[j+1] is not ‘\0’, I find the index where the element key[j] is found in pTemp->element[k], then I check if the number of remaining elements in pTemp->element is sufficient for the remaining elements of key by if((strlen(pTemp->element)-k < strlen(key)-j-astCtr)) if I should continue searching or to proceed to the next word. Now astNum is 0 meaning there are no more asteris. I just compare the remaining elements of pTemp->element and key. After this if j is equal to the string length of key and k is equal to the string length of pTemp->element, I do the exact same thing to store the word in a node and assign it to pList. After searching, I just output the result by calling function traverse with pList as a parameter.