Kelvin Lihandy

2702268183

Boogle

The libraries above are used for these functions:

1. stdio.h – standard input output
2. stdlib.h – malloc(), system(), exit()
3. stdbool.h – boolean variable support
4. string.h – strlen(), strcpy(), strchr(), strtok()
5. ctype.h – tolower()

To make a trie we use a struct that contains a pointer to struct array, boolean variable, and a character array. The struct trie array *child* are used with 26 indexes to represent each character in the alphabet (0 for a, 1 for b, 2 for c, …). This way, the struct are the character. The booleand *end* are used to determine if the word inserted has reached its end. The character array *descripion* are used to store a description for slang word as explained in the case study. Lastly, we initialize a *root* node as NULL to tell that there is no trie created for the current session.

The *menu()* function is used to display a menu for the user to choose from as explained in the case study. To store an input we first declare a variable, in this function as charactervariable *choice*. The decision to use a character is based of ASCII table and input validation. To make sure the user inputs 1 to 5 (number of options), we need a character variable to keep asking for input as long as the input character ASCII is higher that ‘5’ or lower than ‘1’ or if the inputted value is longer than 1 character. In addition to that, we use *[^\n]* to get the inputted value until *enter* key is pressed (if we use *c* than it would yield an error wher the user inputs more than one character or use spaces in between. Lastly we return those that passed the validation (1 to 5 as character).

The *createNode()* function is used to create a node for trie with *malloc()*, assign the node’s *end* as false as it is unknown whether this node is the end of word, loops 26 times (for each alphabet or index) to set each *child* struct to NULL to tell that the trie has not yet created for this session. Lastly we return the created trie node to really create it.

The *alphabetValid()* function is used to check if every character in the inputted word is an alphabet to validate if it’s an valid word. This function asks for 1 parameter, an array of characters to be checked *word[]*. First initialize the *alphabet* to true assuming *word[]* consists of alphabets only. The function loops until the last character in *word[]* to check the ASCII values of each alphabet and checks if that character is not placed between a to z (lower that ‘a’ or higher than ‘z’) and A to Z (lower than ‘A’ or higher than ‘Z’). if the character satisfies those constraints then we interrupt the loop and set *alphabet* to false (not all alphabet. After the loop is finished or interrupted, return the *alphabet* value.

The *escape()* function is used for asking the user to continue with another operation. The *\n* output is just for decoration. Similarly with *menu()* function, we need a character array variable *enter* to store the inputted characters. What the user inputs does not matter as in the end, the enter key is pressed which completes the requirements stated in the second output. The *getchar()* function is used to catch unwanted ‘\n’ from pressing the *enter* key.

The *inputSlang()* function is used to insert the word into the created trie. This function asks for 3 parameters, a pointer for root trie node *root*, an array of characters to be inputted *word[]*, and the description of that character array *desc[]*. The first step is to know what is the length of *word[]* using *strlen()* and stores the value to an integer variable *length*. The next step is to use *root* to assign it to another trie struct variable *current* that represents what the current node is. Then loop *length* times and check for each loop if the node for each character in *word[]* has been created before or not. If not, then create the node by using *createNode()* function above (the index in *child* node corresponds with the current *word[]* character – ‘a’ (‘a’-‘a’ equals 0, ‘b’-‘a’ equals 1, …)). After each check, move *current* to the node of that character as the new *current*. This way, a specific string of trie can be traversed and checked. When the loop ends, all characters from *word[]* has been created of traversed. As the last character node is the current node, we update the *end* to true to mark the end of that word and assign the description with *desc[]* with *strcpy()* because a character array can’t be simply assigned to another variable.

The function *releaseSlang()* is used to store data that needs to be inputted using *inputSlang()* function. This function asks for a parameter, a pointer for root trie node *root*. First we need to declare some character array variables and an integer. Then we need to ask the user for to input one of those character array, *newSlang[]* and validate until the input is longer than one word using *strlen()*, no occurrence of space (‘ ‘) using *strchr()*, and all character in input are alphabets using *alphabetValid()* function above. After all of those requirements are satisfied, the loop will be exited. The next step would be changing all occurrences of uppercase character with its lowercase counterparts to make it simpler by regarding uppercase and lowercase as the same thing.

As the case study includes a scenario where the user inputted the same word and the system updates the new description, this function needs to know whether the current inputted word is in the trie yet by using a similar method with *inputSlang().* So, assume the word is in the trie by assigning *found* to true. What differs from that function’s method is when a node for the next character is not found (it points to NULL), set *found* to false (for the case where the new word branches from the existing word or continues from it) and exit the validation loop. Otherwise continue traversing until the end of *newSlang[]* (*length* times). The logic for this case is that when the system checks for the first extension character, the node for that character has not been created yet as the existing word does not reach that side of extension (new word “abcd”, existing word “ab”, “abc”, “abce”).

After that, we ask the user for the second character array *description[]* and validate it. This method of validation uses *strtok()* to divide *description[]* to count how many words inputted using spaces (1 space means 2 words, 2 spaces means 3 words, …). Variable *descriptionCopy[]* is used as a substitute to *description[]* to prevent it get divided from *strtok()* and causing only 1 word left. To store the count, we use the declared integer *parts* that is set back to 0 to refresh the counter every failed validation loop (parts still less or equal to 2). Then place *root* from parameter, *newSlang[]*, and *description[]* into *inputSlang()* function’s call. In the selection between new release and update, current->end == false serves as the case where the new word is in an existing word (new word is “ab”, existing word is “abc”). The logic is based on when the new word reaches the end of loop, the found would be true as that node has been created before, but because it is the end of word, *end* would be false as the new word has not been inserted and triggers new release output. In the case when the user inputter an existing word, the loop would not set found to false and the *end* would be true so output update message. Lastly *escape()* function is used as shown in case study document.

The function *searchSlang()* is used to search a word from created trie. This function asks for a parameter, a pointer for struct trie *root*. The function starts with validating if the current session’s trie holds any data by looping the *child* indexes until there is one that does not point to NULL (a trie connected *root* exists) therefore there is at least one data inserted in trie. If there is no data then output no data message.

If a data exists then the system asks the user for a word that needs to be searched and validates in the same way as *releaseSlang()* above when inputting a word to be inserted into trie.

The next step would be using the same method again in *releaseSlang()* when validating whether inputted word exists in the trie or not. This method is used because if inputted word doesn’t exist, then that word is a new word that can be inserted using *releaseSlang()* so we show no word message. Similarly, if that word already existed, that word’s description can be updated in *releaseSlang()* so we show the word and its description. Lastly, *escape()* function is used again for the same reason as before.

The *printSlang()* function is used to show every word in trie by doiing it recursively. This function asks for 4 parameters, a pointer for the current trie node *current*, an array of characters to store the word to be showed *word[]*, an integer that represents the current depth of specific trie branch (for the system to know where to put the new character inside the array that forms a word) *index*, and an integer pointer so the word’s number can be updated when one word is completed. This function outputs the word lexicographically as it iterates from left to right (“abcd” would be iterated first than “abda”) because of how it is created and iterated. First the function loops for 26 times (26 alphabets) then it checks if the current node is the end of word. If it is, then we assign the last index of the word as a NULL terminator so the system knows it is a string and can be outputted without error. After it increase the number by one using the pointer. Whether or not that *if* operation is executed, the program would run a loop to find another character and adds that character to *word[]* whose placing controlled by *index*. After it, the program calls this function again with some modifications notably, passing the node where that character is found as *current* to traverse down the trie. When the new character is found, the program does not stop and keeps looping until it cant’t to find every single instance of word.

The *printPrefix()* function is used to show all words that starts with a specific word (prefix). This function asks for 3 parameters, a pointer for the current trie node *current*, an array of characters to store the word to be passed *word[]*, and an array of characters *prefix[]*. First we create an integer *skipIndex* to know when the loop would stop. In the loop we first check if there is a node with the characters in *prefix[]*. If there is not, output no prefix message and exit the function. If there is, copy each character in *prefix[]* to *word[]* while traversing down the trie. After the loop ends, we declare counter variable *count* to 1 to be used in *printSlang()* that we call next using the current node as root, *word[]* that contains *prefix[]*, index from *skipIndex*, and pass count. This logic can be explained using subtree. When showing all words that starts with a prefix, we just have to ignore anything that’s not rooted from the last character of the prefix while still starting traversing from the first character. By this, we can start the traversal from the last character as *root* and by passing *word[]* and *skipIndex[]* we can continue all the words from *word[skipIndex]*.

the *viewPrefix()* function is used to store data that need to be passed to *printPrefix()* function. This function asks for a parameter, root trie node root. First we do data validation just like *searchSlang()*. Then validate inputted prefix if that prefix consists by just alphabet and no spaces. After that declare a character array *word[]* to be used to call *printPrefix()*. Lastly call *escape()* like before.

The *viewSlang()* function is used for data validation and storing variables that are passed to *printSlang()* function. This function asks for a parameter, root trie node *root*. First we do data validation just like *searchSlang()*. Then we output decorative message and pass character array *word[]* and counter *count* to *printSlang()* with index value of 0 as the *word[]* is still empty so it needs to start from the first index 0. Lastly we call *escape()* like before.

The *main()* function is used as a driver for the whole program. First we initialize root node to be used for all functions above. Then we create a do-while loop to constantly do operations with the trie. After that call *menu()* and reduce its output by the ASCII value of 0 to convert from character to integer that can be used to pick the operations. Before picking we clear the console with *system(“cls”)* for decorative purposes. We do this again after finishing an operation. If the user picks 5, we display a closing message and exits the program using *exit(0)*.

**INPUT WORD**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, tampilan

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, tampilan

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**SEARCH WORD**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**SEARCH PREFIX**

**Sebuah gambar berisi cuplikan layar, teks, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi cuplikan layar, teks, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi cuplikan layar, teks, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi cuplikan layar, teks, software, Software multimedia

Deskripsi dibuat secara otomatis**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**VIEW WORD**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**

**EXIT**

**Sebuah gambar berisi teks, cuplikan layar, software, Software multimedia

Deskripsi dibuat secara otomatis**