DESIGN.pdf
Assignment 7
Despina Patronas

Trie implementation

using code.h defined macros (ALPHABET = 256) (STOP_CODE = 0) (EMPTY_CODE = 1) (START_CODE = 2)

Define struct Trienode

Trie struct:

Trienode children[ALPHABET] *//each trie node is initialized with 256 children aka: alphabet*

Code *// the data held within the trienode*

End Trie struct

Define creating the trie:

Creating a trie starts with allocating a trie node initially.
Start the trie node with a root which is a trie node created with EMPTY_CODE index

TrieNode trie_create (void)

Return trie_node_create(EMPTY_CODE)

End trie_create

Define creating trie_node function

The creation of the trie_node itself allocated memory for the trienode and set space to NULL
There is no allocation of the children, however children will each be set to NULL
set the code field as the index
*Setting the indexed child to NULL to keep the memory clean

TrieNode trie_node(index)

Allocate data for the newnode

Iterate from I: [0, ALPHABET]

newnode ->children[i] = NULL
Increment i

end iteration

newnode ->code = index

return newnode

end trie_node

Define Trie_step function

Trie node step will querie a node in the tree for the symbol.
If the symbol is found return the ptr to the trinode child which the symbol was found at.
If the symbol is not found, return null.
If the node has no children return null
*always check if parameter ptr exists first

```
TrieNode trie_step ( *node, sym)

    If (node->children[sym] == NULL )

        Return NULL

    Return node->children[sym]

End TrieNode trie_step
```

Define Free(ptr) Function

Function to free ptr
Also set mem to NULL

Define trie node destructor function

delete the node that was allocated and set the memory to null
using the Free() function

```
Trie_Node_delete ( *node )

    Free(node)

End Trie_node_delete
```

Define trie tree destructor function

Trie delete will clear all the children from the bottom up.
This is done recursively by calling trie_delete on the children of the node (while there is children call)
And setting all of the children null
*Setting the indexed child to NULL to keep the memory clean
*always check if parameter ptr exists first

*Trie_delete (*node)*

Iterate from i: [0, ALPHABET]

Trie_delete(node->children[i])

Node->children[i] = NULL

Increment i

End iteration

Trie_node_delete (node)

End trie_delete

Define trie reset function

Trie reset will retain the root while clearing the children from the bottom up
This is done by using the recursive trie_tree delete function on the children of the root
*Setting the indexed child to NULL to keep the memory clean
*always check if parameter ptr exists first

*Trie_reset(*root)*

Iterate from I: [0, ALPHABET]

Trie_delete (root->children[i])

Root->children[i] = NULL

Increment i

End iteration

End trie_reset

End Trie implementation

Start Word Implementation

using code.h defined macros (ALPHABET = 256) (STOP_CODE = 0) (EMPTY_CODE = 1) (START_CODE = 2)

Define struct Word

Word contains two fields

** syms* *//holds the symbols for the word (as a byte array)*

length *//holds the length of the words aka num of syms*

End struct word

Define word_creation

allocate space for the word and the symbol array
set the fields of the new word
word passed in = symbols which will be set individually into the symbol array
*always check if parameter ptr exists first

*word_create (*syms, length)*

Allocate memory for new_word

Allocate memory for new_word syms

Set new_word length and iteratively set the new_word syms at each index: [0,len)

Return new_word

End word_create

Define Word_append_sym function

make a new word
if word length is 0, then create a new word with just the symbol and length 1
otherwise word length is old word length +1 and syms + appended symbol
*always check if parameter ptr exists first

*Word *Word_append_sym (*old_word, * sym)*

Allocate memory for new_word

Set new_word->len = old_word->len + 1

Allocate memory for new_word syms

Iteratively set the new_word->syms using old_word->syms at each index: [0,oldword->len)

Add the final symbol at the end of new_word (the one passed into fnx)

Return the new_word

End word_append_sym

Define Word_delete function

free the word passed into the function including its fields.
Set the memory to NULL

*Void word_delete (*word)*

Free the word->syms

Set the word->syms to NULL

Free (word)

End word_delete

Define Wt_create function

word table is an array of words
to create the word table allocate memory for it using MAX_CODE
initialize wt index with a word to start. This word has a length of 0 and NULL symbols to start

*WordTable *Wt_create (void)*

Allocate wt

Set the wt[0] = word_create (NULL, 0)

Return wt

End wt_create

Define wt_reset

reset the word table by setting all the indexes to NULL
iterate through index: [START_CODE, MAX_CODE) incrementing index by 1

Define wt_delete

call wt_reset on wt to delete
free(wt)

End Word Implementation