Assignment 2 - Wacky Trees

- Due Mar 17 by 11:59p.m.
- Points 100
- Submitting a file upload
- File Types c
- Available Feb 26 at 12a.m. - Mar 17 at 11:59p.m.

This assignment was locked Mar 17 at 11:59p.m..

Assignment 2 - Wacky Trees

Managing a large discount store poses its challenges. After several years in business, The Wacky Store is currently grappling with a storage dilemma as customer data is gradually overwhelming their storage drives. The students enrolled in CSCA48 have been tasked with addressing this issue by developing an encoding algorithm capable of representing text as concise numerical values.

Get the starter code here: a2.zip (https://q.utoronto.ca/courses/332080/files/30639063?wrap=1) (https://q.utoronto.ca/courses/332080/files/30639063/download?download_frd=1)

Notes:

- Any mention of ASCII refers to the standard ASCII table from decimal values 0 to 127. This assignment does not involve any extended codes.
- An ASCII table can be found at this link (https://www.asciitable.com/). This table shows the proper ASCII character vs. decimal values.
- In this assignment, `ASCII_CHARACTER_SET_SIZE` will always be `128`.
- You can make your own helper functions.
- You can make your own structs. However, please do so in **wackman.c** as it is the only file you are submitting.
- However, when submitting, do not submit a file with a `main()` function.
  - The main function is purposely kept in a separate file so you can modify it however you wish.
- Additional notes and clarifications will be posted on Piazza. Please ask questions and follow the forums for updates!
- Your file must compile with: `gcc -std=c99 -Wall -Werror -lm main.c -o main`
- The last message in `main()` `decode string = ?` was rendered on a **32-bit** runnable. You may need to include `-m32` in your `gcc` arguments to **force 32-bit compilation**.

Understanding the Data:

Assignment 2 has a strong focus on using both Linked Lists and Binary Trees to solve a common real world compression problem.

WackyLinkedNodes:

- Stores a reference to the root of a WackyTree.

WackyTreeNodes:

- If `left` and `right` are null, it is considered a leaf node. Only leaf nodes can have a `val` that is not `\0`.
- If `left` or `right` is non-null, the other must be non-null as well. One side cannot exist without the other.

Understanding the Starter Code:

The following lists the functions in the starter code as well as their respective documentation.

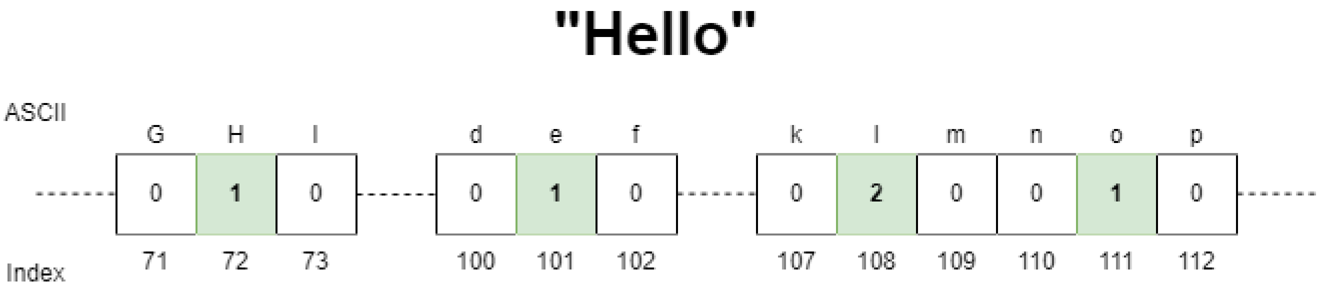
```
int sum_array_elements(int int_array[], int array_size)
```

Given an array of integers and its length, this function calculates the sum of all the elements in the array.

```
void compute_occurrence_array(int occurrence_array[ASCII_CHARACTER_SET_SIZE], char* string)
```

Given an integer array of size `ASCII_CHARACTER_SET_SIZE`, this function computes the number of occurrences of each ASCII character in the given string. The results are stored in the corresponding indices of the occurrence_array. The computation stops at and does not include the DELIMITER character.

The image below shows the resulting `occurrence_array` for the string `"Hello"`.



ASCII	G	H	I	d	e	f	k	l	m	n	o	p
	0	1	0	0	1	0	0	2	0	0	1	0
Index	71	72	73	100	101	102	107	108	109	110	111	112

```
int count_positive_occurrences(int occurrence_array[ASCII_CHARACTER_SET_SIZE])
```

Given an integer array of size `ASCII_CHARACTER_SET_SIZE`, this function computes and returns the number of characters that occur one or more times.

In the image shown above, the `occurrence_array` has `4` ASCII characters with a positive occurrence, as highlighted in green.

```
WackyLinkedNode* create_wacky_list(int occurrence_array[ASCII_CHARACTER_SET_SIZE])
```

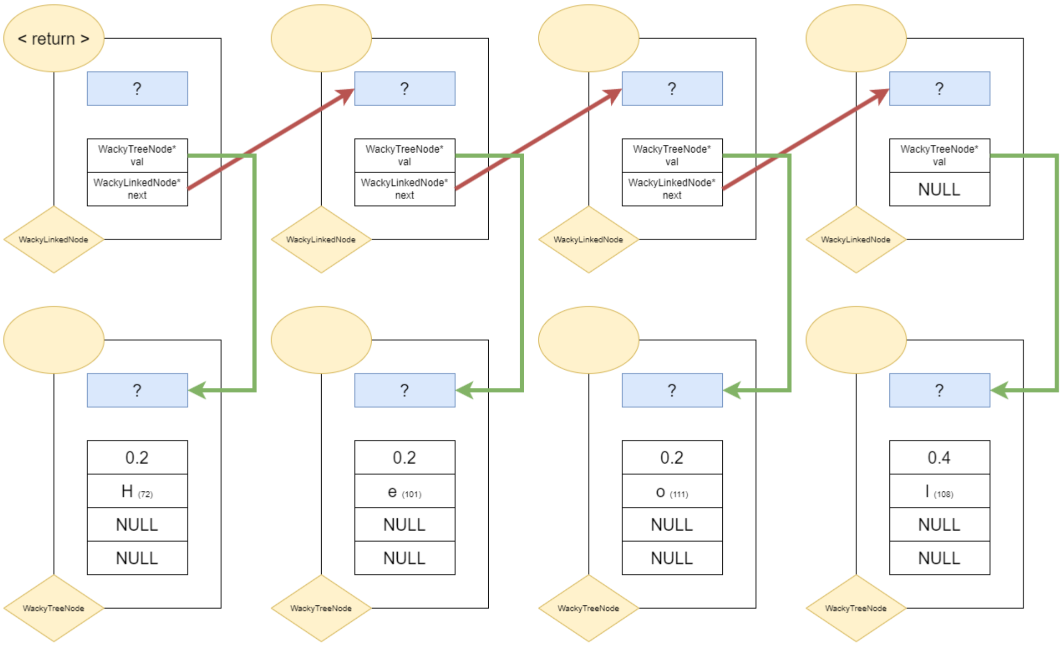
Given an integer array of size `ASCII_CHARACTER_SET_SIZE`, representing the number of occurrences of ASCII characters, this function creates and returns a sorted linked list of `WackyLinkedNodes`.

Each node in the list stores a tree with a single leaf node. The node will contain information about the probability of occurrence `weight` and the ASCII character itself `val`.

Notes:

- The memory of any required data types must be allocated manually.
- Compute the probability of occurrence `weight` for each ASCII character `α` as `occurrence_array[α] / SUM(occurrence_array)`.
- Exclude any elements with a `weight` of 0 from the linked list.
- Ensure that the linked list is sorted in ascending order from head to tail, first by the probability of occurrence `weight`, and in case of ties, by the ASCII character `val` **in ascending order**.

The image below shows the resulting linked list with the `occurrence_array` above.



```
WackyTreeNode* merge_wacky_list(WackyLinkedNode* linked_list)
```

Given a sorted linked list of `WackyTreeNodes`, where each node (initially) contains a tree with a single leaf node, this function generates a tree based on the following algorithm:

- **If the size of the linked list is 2 or more:**

  1. Remove 2 `WackyLinkedNodes` from the head.

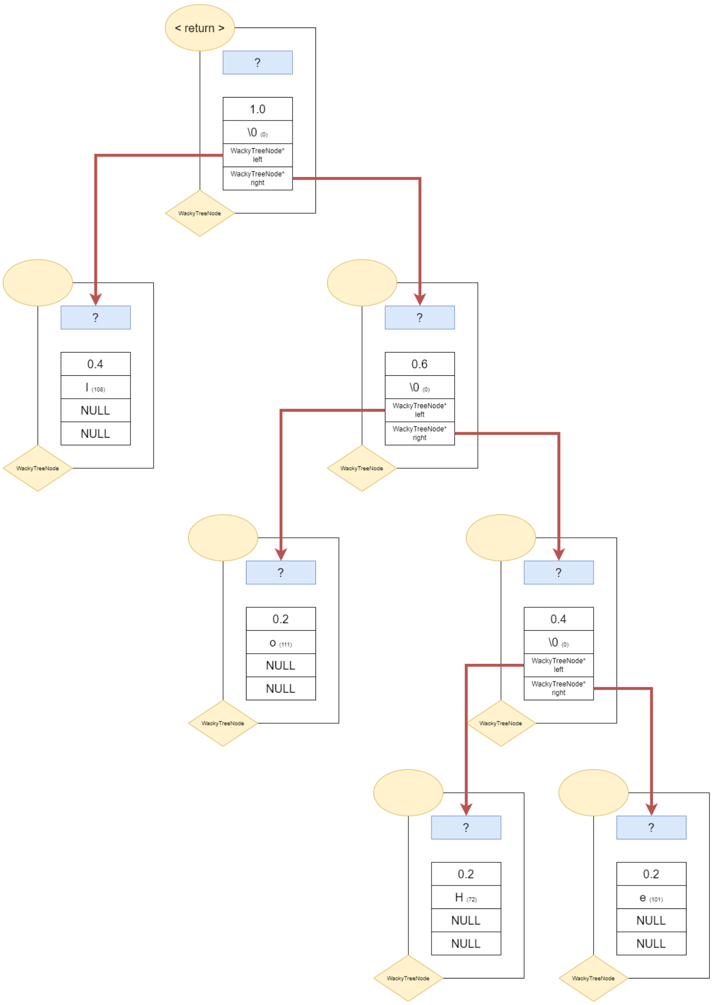
2. Create a new `WackyTreeNode` that joins the tree nodes inside the removed linked nodes. The first node `head` goes to the `left`, and the following node `head->next` goes to the `right`.
3. Create a new `WackyLinkedNode` and add the newly created `WackyTreeNode` back into the linked list. The linked list must remain in sorted order by the probability of occurrence `weight`. If the newly created node has the same weight as another node already in the list, add it in front of all existing similarily weighted nodes.
4. Repeat this algorithm until the size of the linked list is 1.

- **If the size of the linked list is 1:**
  1. Return the address of the tree held at the head of the linked list.
- **Otherwise:**
  1. Return `NULL`.

Notes:

- The memory of any `WackyLinkedNodes` must be freed by the end of this function.

The image below shows the resulting binary tree with the linked list above.



```
int get_height(WackyTreeNode* tree)
```

Given a tree, this function calculates and returns the height of the tree.

```
void get_wacky_code(WackyTreeNode* tree, char character, bool boolean_ar
ray[], int* array_size)
```

Given a WackyTree and a specific character, this function computes the traversal of the character based on its position in the tree. Movement to the **LEFT** is `false`, and movement to the **RIGHT** is `true`. The steps are written inside `boolean_array`, and the total number of steps is stored in `array_size`.

For example, to get to the character `H` using the Wacky tree shown above, we would need to step **RIGHT RIGHT LEFT**. This translates to `[true, true, false]` and an `array_size` of `3`.

```
char get_character(WackyTreeNode* tree, bool boolean_array[], int array_
size)
```

Given the root of a WackyTree, a boolean array, and the size of the array, this function traverses the tree. `false` indicates a movement to the **LEFT**, and `true` indicates a movement to the **RIGHT**. The function returns the character at the node reached after all the steps have been taken. If the node is not a leaf node, it returns the DELIMITER `\0` instead.

```
void free_tree(WackyTreeNode* tree)
```

Given a binary tree, this function frees the memory associated with the entire tree.

Final Remarks:

Use piazza and office hours wisely! Start early and work diligently. Assignments are more challenging than exercises, so you will need to work productively and not wait until the last minute.

Please submit the contents of your `wackman.c`. The file name does not matter.