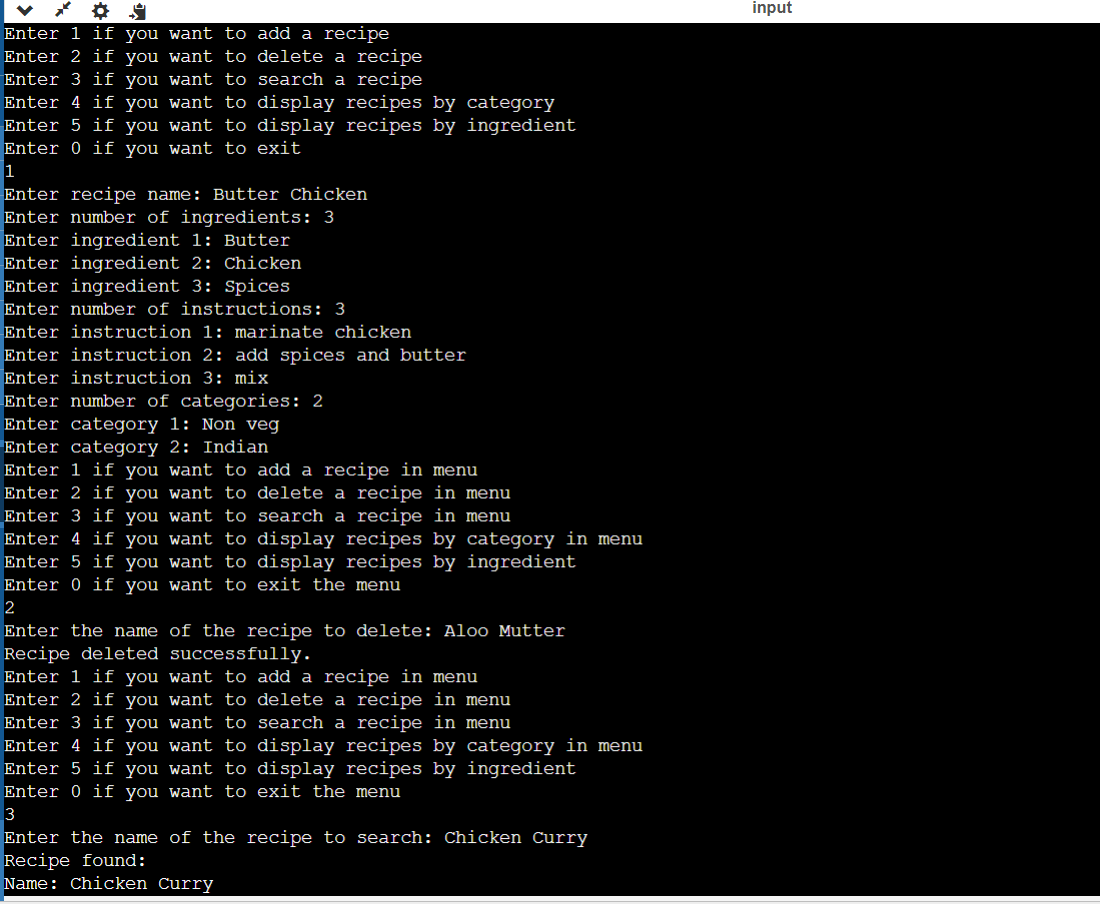
**The data structures used in our project are:**

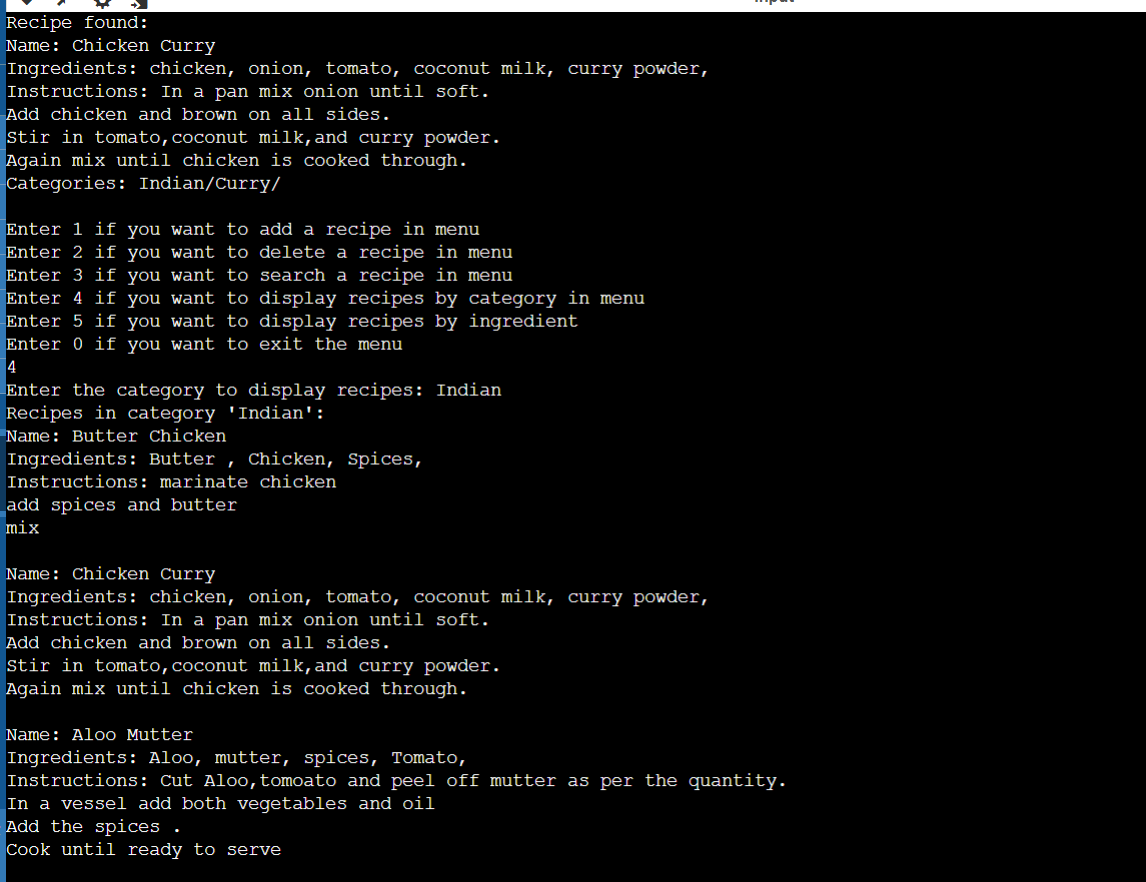
Linked list: To store the categorized recipes, you can use a linked list data structure. Each node in the linked list can represent a category, and it can have a pointer to the head of another linked list containing the recipes belonging to that category.

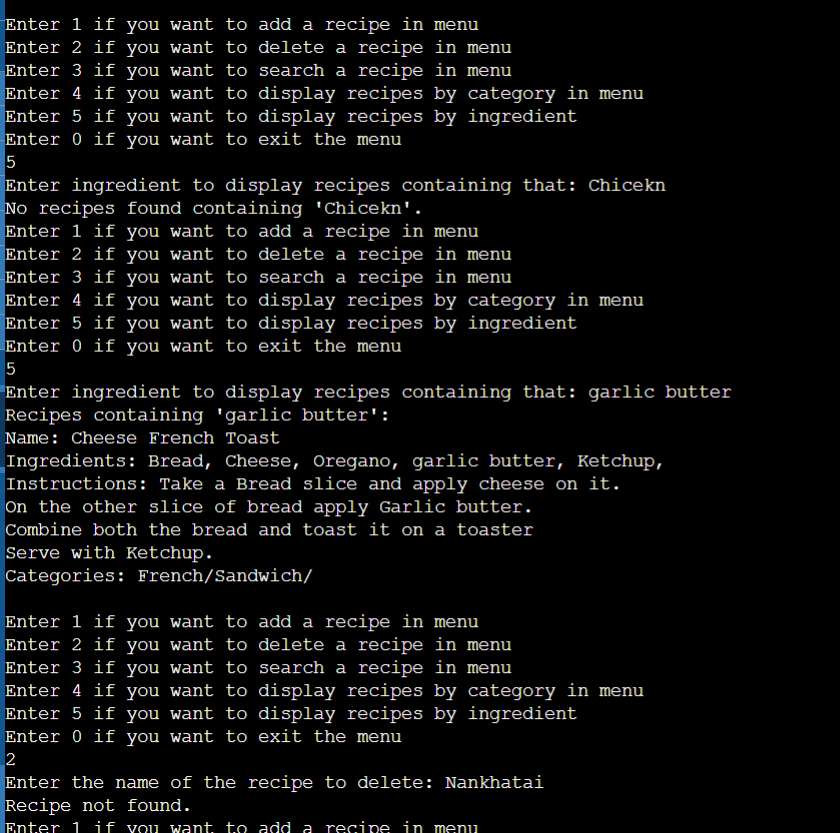
Array of classes: We have used an array of structures to store the recipe information. Each recipe can be represented as a class containing fields like recipe name, ingredients(array of string), instructions (an array of strings), and categories(an array of strings).

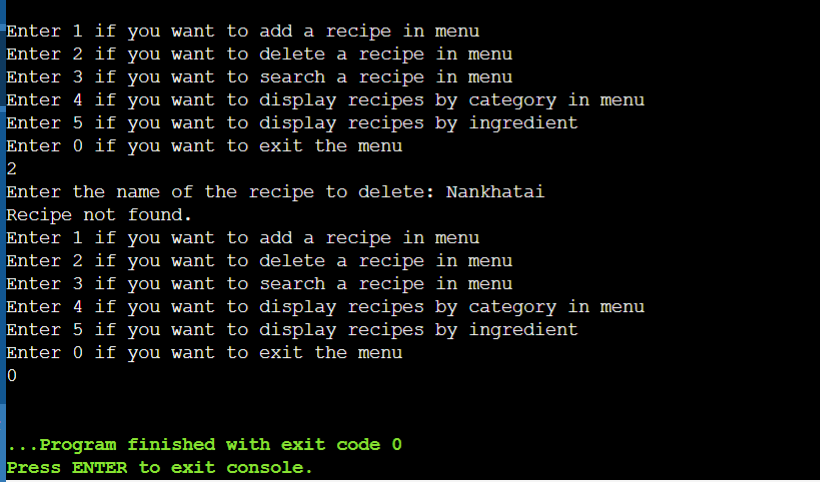
Hash table: We have implemented a search for recipes by ingredients. Each ingredient can be a key in the hashtable and the value can be a linked list of recipes containing that ingredients

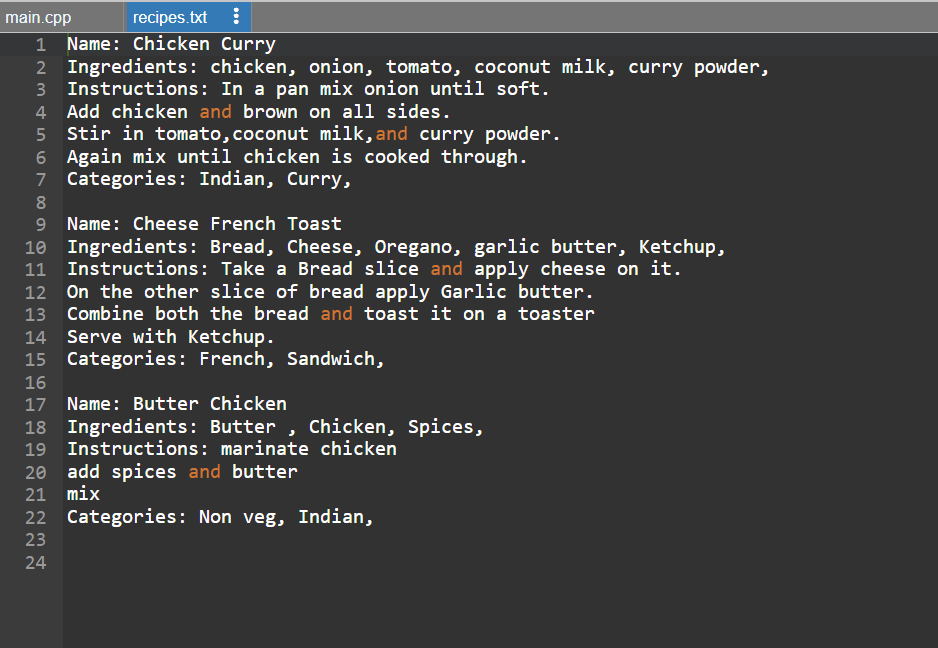
**Screenshots of output**











**Why we used the following data structures:**

Arrays: A set number of recipes (Recipe recipes[100]) can be stored in arrays. Arrays are useful when the number of recipes is known and restricted because they offer constant time access to elements. Anyways, because arrays have a fixed size, they might not be appropriate for dynamic resizing or situations in which the number of recipes can increase.

Linked Lists: Ingredients (IngredientNode) and categories (CategoryNode) are managed using linked lists. For insertion and deletion at the start or finish of the list, linked lists offer dynamic memory allocation and insertion/deletion operations with constant time complexity. They are therefore appropriate for handling collections whose element count may fluctuate regularly.

Hash Table: To effectively map ingredients to recipes, a hash table (ingrndtshashtable) is used. Hash tables are useful for searching recipes by ingredient because they offer constant time average case performance for insertion, deletion, and lookup operations. Rather than searching through every recipe one by one, the code could find recipes that contain a given ingredient more quickly by using a hash table.

We implemented the structures which we were comfortable and learned about those which we weren’t comfortable but familiar.

**Time Complexity & Space complexity of Function:**

**int hashingrdnt(const char\* ingredient)**

Time complexity: O(n)

Where n is the the string length Space complexity: O(1)

**void addRecipe(const Recipe& newRecipe) Time complexity O(n\*(m+p))**

where m is the length of Linked list starting from categories.

p is the number of ingredients. Space complexity: O(p+n)

n is number of categories

**void saverecipesinfile(const char\* filename)**

Time complexity O(n\*(m+p+k) ) n=number of recipes m=number of ingredients p=number of instructions k=number of categories

Space complexity:

**void findrecipebyname(const char\* name)**

Time complexity O(n) n is the number of recipes

Space complexity 0(1)

**void searchrecipebyuser()**

Time complexity 0(n) where n is number of recipes

Space complexity 0(1)

**void searchrecipesbyingredient(const char\* ingredient)**

O(m+k•(r•i))

m is the length of ingredient

k is the number of unique ingredients hashed to same index

r is the average number of recipes associated with each ingredient node i is the average number of ingredients in each recipe

Space Complexity: O(1)

**void addrecipeuser()**

Time complexity:O(n1+n2\*(m+p+q)) n1 is the length of recipe name

n2 is the average length of each ingredient m is the number of ingredients

p is the number of instructions q is the number of categories Space complexity: O(1)

**void deIeteRecipe(const char\* name)**

Time complexity: 0(n)

n is the number of recipes Space complexity: O(1)

**void deleterecipebyuser()**

Time complexity:O(n) where n is the number of recipes

**void displayrecipebycat(const char\* category)**

Time complexity: O(n) where n is the number of categories

Space complexity: O(1)