ICT1002 P3 TEAM 15

C PROJECT REPORT

TAN JIA JUN CLARENCE (2102420)

PANG KA HO (2102047)

WANG QIXIAN (2101751)

zARIFHAZIQ B ZULQHIFFRI (2101668)

TAN SHIREEN (2102432)

2021

Contents

[**ICT 1002 P3 Team 15 C Project Report** 2](#_Toc89371862)

[Introduction 2](#_Toc89371863)

[Data structure implementation for knowledge base 2](#_Toc89371864)

[What is a hash table? 2](#_Toc89371865)

[Knowledge base data structure representation 3](#_Toc89371866)

[Why use a hash table? 4](#_Toc89371867)

[How the program works 5](#_Toc89371868)

[chat1002.h 6](#_Toc89371869)

[datastructure.h 6](#_Toc89371870)

[main.c 6](#_Toc89371871)

[chatbot.c 6](#_Toc89371872)

[knowledge.c 11](#_Toc89371873)

[Breakdown of contribution 12](#_Toc89371874)

# **ICT 1002 P3 Team 15 C Project Report**

ICT 1002 P3 Team 15: Tan Jia Jun Clarence (2102420), Pang Ka Ho (2102047), Wang Qixian (2101751), Zarifhaziq Bin Zulqhiffri (2101668), Tan Shireen (2102432)

[2102420@sit.singaporetech.edu.sg](mailto:2102420@sit.singaporetech.edu.sg), [2102047@sit.singaporetech.edu.sg](mailto:2102047@sit.singaporetech.edu.sg), [2101751@sit.singaporetech.edu.sg](mailto:2101751@sit.singaporetech.edu.sg), [2101668@sit.singaporetech.edu.sg](mailto:2101668@sit.singaporetech.edu.sg), [2102432@sit.singaporetech.edu.sg](mailto:2102432@sit.singaporetech.edu.sg)

# Introduction

This is a terminal chatbot that is able to respond to users' intent (questions/verbs/instructions) and also store/retrieve a database of knowledge in memory, referred to as the knowledge base, if the user requests for it. It is able to learn new answers by asking questions from the user and storing it in its knowledge base. The knowledge base also grows in size during runtime. Therefore, our team has to evaluate a suitable data structure to implement this knowledge base.

# Data structure implementation for knowledge base

As the chatbot only recognizes basic instructions and question intents, with each question intent (a section) mapping to a series of key-value pairs, we wanted to design the program such that when the chatbot is provided with some intent and entity, the program can immediately access the related intent and entity in its knowledge base, and respond with the description to the user, if such knowledge exists. If not, it will query the user, and insert the new knowledge into the relevant question intent. As the chatbot also needs the capability to load a knowledge base externally, we wanted a data structure that can provide fast lookup and insert operations. If there is an existing section, we want to add entries to the section, else create a new section. If there is already an existing entry with the same entity name, we want to replace the description instead.

We considered the application of different data structures and ultimately concluded that the data structure that best matches our desired implementation is a hash table.

## What is a hash table?

A hash table is a data structure that allows us to map some value, called a key, to some other value, called a value, using a technique called hashing. Hashing is a way of taking some key ‘x’ and mapping it to some integer in some fixed range. This is done by a function, which is referred to as the hash function. One property of the hash function is that it is deterministic. Given the same key ‘x’, the hash function will always output the same integer value in a given fixed range. As the hash function has to be deterministic in order for the hash table to operate properly, we require that the keys of our hash table be immutable to guarantee this behaviour.

A hash table is essentially an array. For our project’s specific implementation, our data structure is an array of linked lists, among other implementations. We get the benefit of quick lookup from arrays, while also getting the benefit of quick insertion from linked lists. The hash function is used as a way to index key-value pairs into an array, or the hash table. The key is passed as the input to a hash function, which then generates an integer. This integer is used as the index to address the specific array index where we can insert the key-value pair into.

## Knowledge base data structure representation

Figure 1 shows the representation of our sections hash table. In each section, they are identified by their “Key” field in a structure called a “section\_node”. Each section node is uniquely identified by its key. For each section node, it also contains the entries related to the section as the “Value” structure field.

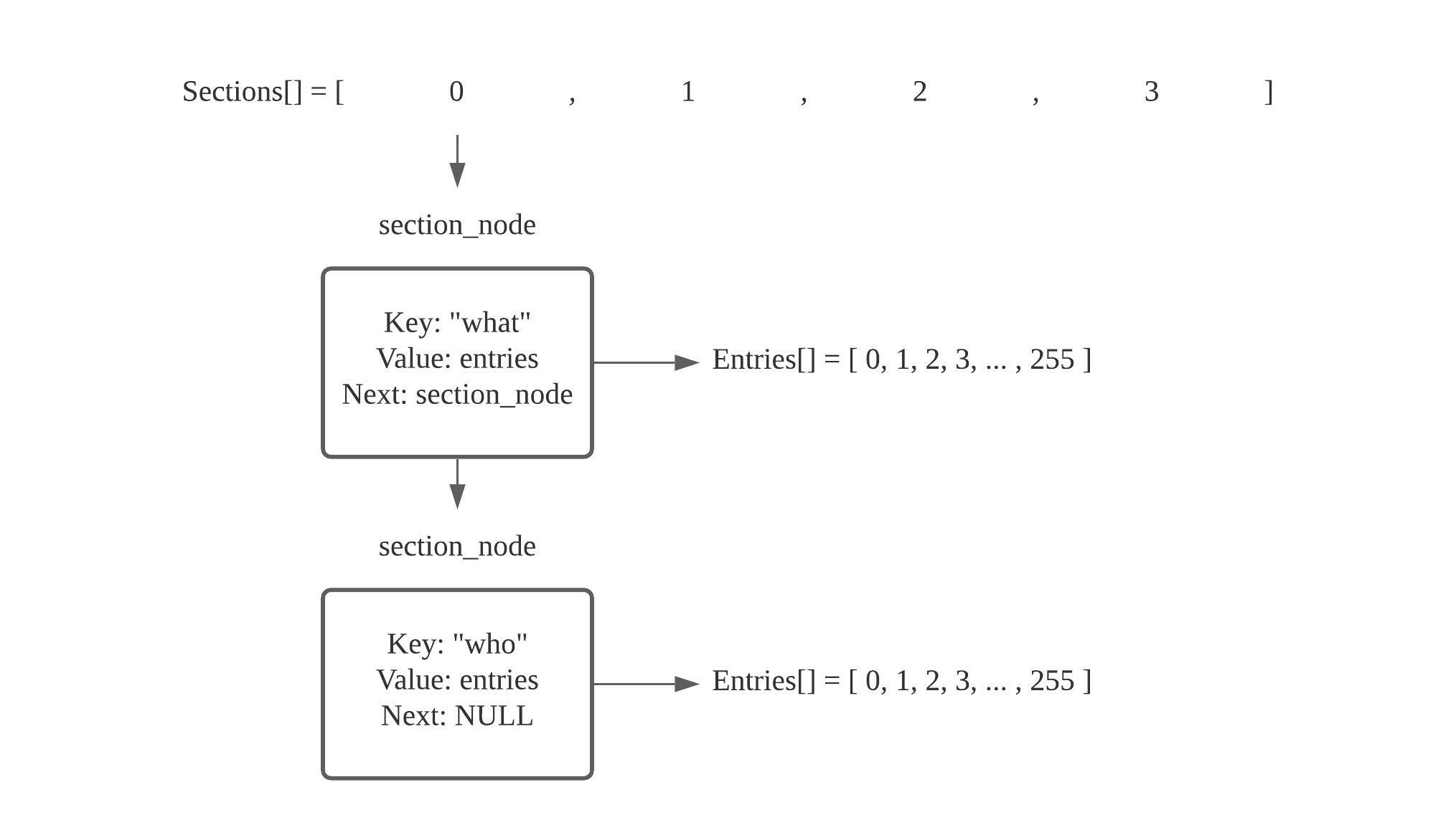


Fig. 1 Hash table representation of sections and entries per section

In Figure 2, we can see that the entries in a section is also a hash table, where the implementation is also an array of linked lists. Each entry node structure, referred to as “node” in Figure 2, contains a “Key” field which stores the entity, and the “Value” field which stores the description of the entity. Likewise, the “Key” field uniquely identifies the entry.

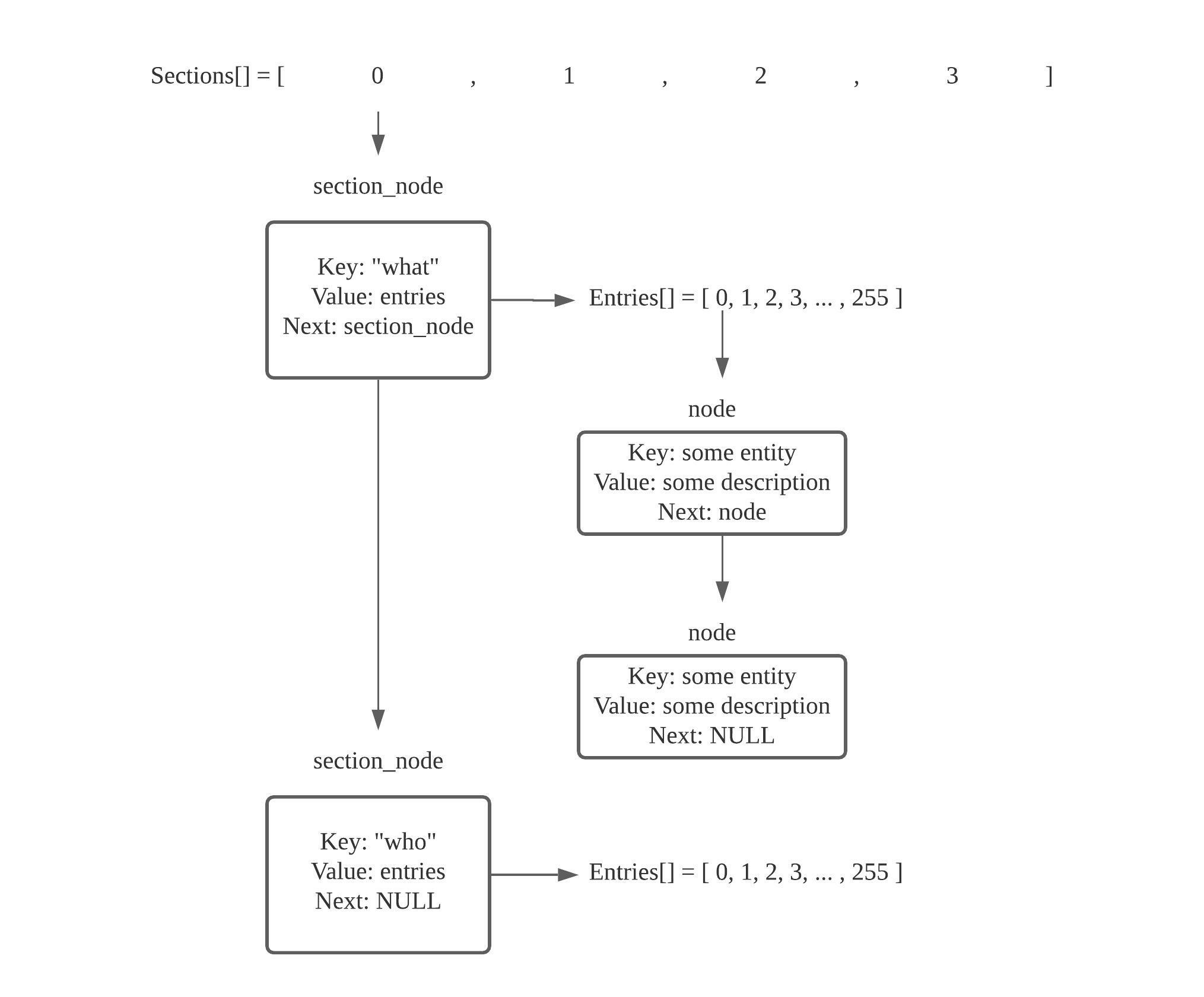


Fig. 2 Collision handling and entries hash table per section

As there is a possibility that two or more keys can hash to the same index, we resolve this collision by using separate chaining. Separate chaining deals with the collision by turning the index slot into a linked list. This can be seen in Figure 1 and 2 where each respective node has a pointer to the next node in the linked list; the end of the list is reached when the next pointer points to NULL.

## Why use a hash table?

The main reason why a hash table is chosen is because we wanted to mimic the chatbot behaviour as closely to its program specifications, where the chatbot can access the intent, and provide the entity description immediately.

In addition, with a hash table, we are able to have a constant time O(1) operation on our lookup and inserts, assuming we have a good uniform hash function and distribution. This means that we are able to provide very fast program execution regardless of the size of the knowledge base, up to a certain limit.

The limitation is the case where either the hash function distributes unequally, where in that case, the big O decays to the length of the linked list at the index slot, or when there are too many entries that the hash table becomes too full.

The string hash function that we used is djb2’s string hashing function. Discussion of how djb2’s hashing algorithm works is beyond the scope of this report and will not be discussed here. As djb2’s hashing algorithm is one of the better generic string hashing algorithms, the distribution will be fairly uniform.

In the other case when there are too many entries loaded, automatic resizing of the hash table needs to be done when the load factor reaches a certain threshold in order to preserve the constant time O(1) operation. Load factor is the number of entries divided by the total hash table size. This is, however, not implemented in the data structure due to time constraints and program complexity. This remains a possible improvement that can be made in the future.

The main disadvantage of the hash table is that it is memory intensive, so we are assuming that we have enough memory capable to support such a data structure. This is the trade-off between space and time, if we want the fastest speed possible. In addition, we are also assuming that ordering of the sections and entries are not needed, as one property of a generic hash table is that it does not preserve order of insertion.

# How the program works

The main execution of the program lies in the main.c source file. It has a do while loop that calls chatbot\_main(), which is the main execution of chatbot. chatbot\_main() controls the logic of the chatbot operations, where each operation is further divided into smaller functions that provide the logic for the chatbot.

These smaller functions provide the specific logic for the operation of the chatbot as well as handle any data structure and knowledge base operations.

When the chatbot program first executes, it will greet the user on the terminal. The user will be prompted for some input, and this is repeated until the user provides a recognized intent. Depending on the intent, different actions are performed by the chatbot. The recognized intents are “what”, “where”, “who”, “reset”, “load”, “save”, “exit”, “goodbye”, “quit” as well as “display”.

If it is a question asked, the chatbot reads from its knowledge base, and provides an answer accordingly, if it exists. Else, it will ask the user for the answer, this answer is then inserted into the knowledge base.

If it is an instruction provided, the chatbot will perform the action accordingly.

1. RESET - resets the chatbot and frees all memory allocated for the knowledge base.
2. LOAD - loads a provided knowledge base file, some error checking is provided (empty file names, unloadable files, unsupported file types, bad file contents are accounted for).
3. SAVE - saves the existing knowledge base file into the folder relative to the program execution folder.
4. EXIT / GOODBYE / QUIT - free all memory allocated for the knowledge base and terminates the program execution.
5. DISPLAY - displays the current chatbot knowledge in the knowledge base onto the terminal.

The do while loop keeps executing until an “exit” / “goodbye” intent is provided by the user before the program terminates.

We will briefly discuss each file and the functions in each file for the chatbot program.

## chat1002.h

This header file contains the macro definitions for input lengths and function return codes, as well as function prototypes provided in the skeleton code.

## datastructure.h

This header file contains the structure definition for the data structure implemented for the knowledge base to store the "intents", "entities" and "descriptions". The data structure implementation is a nested hash table. We have an outer hash table of "sections", it contains key-value pairs of a string as the key (section key), and the value as the hash table for the entries for the specified section (1 section key to 1 hash table). This inner hash table contains the entries for the specified section. This inner hash table contains key value pairs of a string as the key (the entity), and the value as the description for the entity, which is also a string.

This header file also contains the macro definitions used for setting the size of the hash table as well as function prototypes for the data structure operations. These operations included get and set functions, as well as helper functions such as displaying the hash table contents and unloading the data structure (freeing the allocated memory).

## main.c

This is the source file for the main execution of the chatbot program. It calls chatbot\_main() which is defined in chatbot.c.

## chatbot.c

This is the source file that contains the main execution of the chatbot. It contains the logic for the chatbot as well as the function declarations for the data structure operations. chatbot\_main() also initializes the data structure for use during its first call.

Below is a brief description of the list of functions in chatbot.c.

* chatbot\_main()
  + This is the main function that controls the chatbot, it branches to the specific action depending on the user input, and returns a function code to main.c to determine if the program should keep running.
* chatbot\_is\_exit()
  + This simply checks if the user provided an exit, goodbye or quit intent.
* chatbot\_do\_exit()
  + This calls unload\_section\_ht(), which frees all allocated memory for the knowledge base. It then modifies the chatbot response, and returns the function code to terminate the program.
* chatbot\_is\_display()
  + This simply checks if the user provided a display intent.
* chatbot\_do\_display()
  + This calls display\_section\_ht(), which prints the knowledge base contents to the terminal.
* chatbot\_is\_load()
  + This simply checks if the user provided a load intent.
* chatbot\_do\_load()
  + This function checks for proper file name input as well as file type. If it is, it will open the file for reading and pass the file as input to the knowledge\_read(). This function also supports space-separated file names.
* chatbot\_is\_question()
  + chatbot\_is\_question() has a fixed set of intents in an array which includes “what”, “where” and “who”. This function iterates through the array using the first word of the user’s input. If the first word of the user’s input matches an intent in the array, it will return 1. Else, it will return 0.
* chatbot\_do\_question()
  + chatbot\_do\_question() initialises the entity string and intent buffer and checks if the second word in the response is “are” or “is”. If it is, the function will concatenate the words after the second word into the entity string and copy the intent of the user into the intent buffer. This will also utilize the function knowledge\_get() to check if the entity exists for the user’s intent in the knowledge base. If it exists, it will output the response to the user based on the user’s response. However, if it does not exist, it will prompt the user for a new response to store in the knowledge base and check if the response is valid. The function will then utilize knowledge\_put() to pass the intent, entity and user’s response in the knowledge base. In the case of an empty knowledge base bot, upon the first question asked for each intent, the function will create a new section for that intent in the hash table which will then prompt the user to input a response so that the entity and response will be assigned to the new section.
* chatbot\_is\_reset()
  + chatbot\_is\_reset() utilizes the function compare\_token() to check if the intent is “reset” and return 0.
* chatbot\_do\_reset()
  + chatbot\_do\_reset() utilizes the function knowledge\_reset() to reset the chatbot and remove the entire knowledge base. It will then print out a success statement and return 0.
* chatbot\_is\_save()
  + chatbot\_is\_save() checks if the intent of the user was to save the file by making use of the compare\_token() which returns 0 if results are matched.
* chatbot\_do\_save()
  + chatbot\_do\_save() saves the knowledge input from the user, it will start by checking if there is valid section, such as “what”, “where” and “who”, and if the section exists, it will continue to check the elements of user’s input. It caters for several scenarios: if the second element of the user’s input is “as” or “to” (example: “save as filename.ini, save to filename.ini”), then the third element of user’s input will be copied to the filename variable. Otherwise, if the user enters a command without “as” or “to” (example: save filename.ini), then the second element of user’s input is copied to the filename variable. Lastly, the function opens an empty file with the filename mentioned by the user for writing, and utilize the knowledge\_write() function to write the knowledge base into the file.
* chatbot\_is\_smalltalk()
  + chatbot\_is\_smalltalk() searches for the intent listed in an array that belongs to any small talk words. Example: “Hello”, “Hi”, “Hey”, “Greetings”. It utilises the function compare\_token() to check if the intent consists of those words. If it does, it will return 1 and chatbot\_do\_smalltalk() will be called. If it does not content the words listed in the array, it will return 0.
* chatbot\_do\_smalltalk()
  + chatbot\_do\_smalltalk() contains an array of output that gets matched to each intent that was identified in chatbot\_is\_smalltalk(). It utilises compare\_token() again to identify the words and uses the snprintf() function to output it into the response buffer which would then be printed by the main loop as a reply to the user. If the user says “bye” or “goodbye”, the chatbot will return 1 and then this will exit the program. Otherwise, the chatbot will return 0 and continue prompting the user to chat.
* create\_entity\_ht()
  + This function simply allocates the memory for an entity hash table. It allocates enough memory for the entity hash table structure (the struct pointer) as well as enough memory for the entries (the array pointer) inside the structure. Memory allocation checking is done to ensure that there is still sufficient memory to be allocated. NULL is returned if memory allocated fails.
* section\_entity\_ht\_set(sections hash table, section key, entity key, entity description)
  + This function sets the entity hash table entry in the entity hash table for the related section node with the given section key, entity key and the entity description.   
      
    If it is a new entry (new entity key), new memory is allocated and a new entry is inserted into the entity hash table for the related section node.   
      
    If there is an existing entry with the same key, the entry’s value is updated instead. It returns true / false depending on the success of the operation.   
      
    In summary, the specific implementation is we first hash the section key to get the index for the section in the sections hash table. We access the section immediately and check for NULL, if it is NULL, the section does not exist, and a message is printed onto the terminal to inform the programmer that he is trying to set an entity key-value pair in a section that does not exist. A new section must be created first using section\_ht\_set(). Else if the section does exist, we will traverse the section nodes in the index (if there is a collision and it is chained, so a linked list is formed) to find the section node with the same section key as the section key provided in the function call.   
      
    If the section key matches, we will call entity\_ht\_set() to set the value in the entity hash table (the entries). Else we reached the end of the list at the index, and we inform the programmer once again.
* section\_entity\_ht\_get(sections hash table, section key, entity key)
  + This function gets the entity descriptionwith the given entity key in the related section (that is identified by the section key).   
      
    It first hashes the section key to get the index to address the section. Then it checks for NULL, if the section does not exist, a simple debug message is printed onto the terminal to inform the programmer that he is trying to get a value from a section that does not exist. Else, we traverse the linked list at the index, compare the section key, and if there is a match, call entity\_ht\_get() to get the value for the entity.   
      
    Else if the end of the list is reached, NULL is returned as there is a section entry at the index, but no matching section key.
* entity\_ht\_set(entity hash table, entity key, entity description value)
  + This is a helper function that sets the entity hash table entry with the given entity description key value pair. It is called in section\_entity\_ht\_set().   
      
    It first hashes the entity key to get the entity hash table index, address the index slot and check for NULL. If it is NULL, we simply insert the entry. Else, we traverse the existing index (which is a linked list) and compare the entity keys. If there is a key match, this means that we are updating the value for an entity, in this case, we allocate memory for the new value, copy the description over to the new value, free the memory for the old value, set the new value and return true. Memory allocation checking is done as well.   
      
    If the end of the linked list is reached, we simply insert it at the end of the list.
* create\_entity\_entry(entity key, entity value)
  + This is a helper function that simply allocates memory for an entry in the entity hash table. Memory allocation checking is done as well.
* entity\_ht\_get(entity hash table, entity key)
  + This is a helper function that gets the entity hash table entry with the given entity description key value pair.   
      
    It hashes the entity key, addresses the specific index slot and checks for NULL. If it is, that means there is no entry at the index, NULL is returned.   
      
    Else if there is an entry, we traverse the linked list at the index, and compare the entity key. If the entity key compares successfully, we return the description value for the entity. Else if the end of the linked list is reached, NULL is returned.
* display\_entity\_ht(entity hash table)
  + This is a helper function that simply iterates through the entity hash table and prints the key value pairs in the hash table onto the terminal.
* unload\_entity\_ht()
  + This is a helper function that simply iterates through the entity hash table and frees the allocated memory for each entity node (the entry pointer itself, the key and the value), the array pointer in the hash table structure, and the hash table structure pointer itself.
* section\_ht\_get(sections hash table, section key)
  + This function gets the entity hash table in the sections hash table with the given section key.   
      
    It hashes the section key to get the index to address the section node at the index, checks for NULL, and if it is, NULL is returned as there is no such section with the given section key.   
      
    Otherwise, a section is found, then we traverse the linked list at the index, compare the given section key with each of the section node and its key, and if the key compares successfully, the entity hash table (the entries) related to the section is returned. Else, if the end of the list is reached, there is an entry but no match key, therefore NULL is returned.
* section\_ht\_set(sections hash table, section key, new entity hash table)
  + This function sets the entity hash table in the sections hash table with the given section key and the new entity hash table.   
      
    It hashes the section key to get the index to address the section node at the index, checks for NULL, and if it is, we call create\_section\_entry() to allocate memory for the section node, and simply insert the new node at the index.   
      
    Else, if a section is found, we traverse the linked list at the index, check for key comparison, if a key compares successfully, we free the previous entity hash table, and replace it with the given new entity hash table.   
      
    Else, if we reach the end of the list, we simply create a new section node and insert it at the end of the list.
* create\_section\_entry()
  + This is a helper function that simply allocates memory for an entry in the sections hash table. Memory allocation checking is done as well.
* display\_section\_ht()
  + This is a helper function that simply iterates through the sections hash table and prints the key value pairs in the hash table onto the terminal. As the value is an entity hash table, display\_entity\_ht() is called for each section node.
* unload\_section\_ht()
  + This is a helper function that simply iterates through the sections hash table and frees the allocated memory for each section node (the section pointer itself, the key and the entity hash table). The entity hash table is freed using unload\_entity\_ht().
* hash()
  + This is the djb2’s string hash function. Credits goes to djb2 hash function from <http://www.cse.yorku.ca/~oz/hash.html>
* entity\_hash()
  + This is a helper function that simply calls hash() and modulo it by the maximum entity table size macro definition.
* section\_hash()
  + This is a helper function that simply calls hash() and modulo it by the maximum section table size macro definition.

## knowledge.c

* knowledge\_get()
  + knowledge\_get() utilises the function section\_ht\_get() to check if the intent exists in the hash table. If the section returns null, it will return a KB\_INVALID, which means that the intent is not listed in the sections. Then, it will try to search for the description value with the intent and the entity key. If it manages to find a valid entry, it will then copy the contents of the description and load it into the response buffer. This will return KB\_OK. Otherwise, if there is no match with the entity key, its will return KB\_NOTFOUND.
* knowledge\_put()
  + knowledge\_put() utilizes the function section\_ht\_get() to check if the section that is related to the intent exists in the hash table. If section retrieval fails, it will return an invalid statement. Else, the function section\_entity\_ht\_set() will insert the new response in the knowledge base or overwrite if the response already exists. Once the set operation is completed, it will return a success statement. However, if it does not fulfill either of the criterias, it will return a memory allocation failure statement.
* knowledge\_read()
  + knowledge\_read() reads a file line by line, looking for sections that start with ‘[‘ and ‘]’ brackets to detect a section. Checking for bad input is done here, such as disallowing empty sections. We also placed an additional check for only recognized intents, but this can be changed in the future as well. Initially, all sections are recognized but we restricted it to “what”, “where” and “who” as specified in the project requirements.   
      
    If a valid section has been detected, we first check if such a section exists in the sections hash table, if not, we create the new section using section\_ht\_set(). If it exists, we will scan for valid entries next, reading it line by line, until a new section has been detected. Blank lines are skipped, and entries are only valid and accepted if there is an ‘=’ (but empty ‘=’ is checked and not allowed), else we read the file character by character, building the entity string and description, then storing it in a buffer where we will call section\_ht\_entity\_set() to set the entry. It returns a boolean value, where we will increment a counter variable for the pairs, and return the counter when the end of file is reached.
* knowledge\_reset()
  + knowledge\_reset() utilizes the function unload\_section\_ht() to unload section hash table and free the allocated memory to make all pointers null. This function resets the entire knowledge base which removes all known entities from intents.
* knowledge\_write()
  + knowledge\_write() writes the knowledge base to a file. It will start by iterating through the sections hash table to check if the section exists. If it does, a pointer will be set to the sections’ index as the head of the list. We will traverse the list at the index, as long as the pointer has not reached the end of the list, the section key (e.g. what, where and who) will be added to the file. The same process will be repeated on the entity hash table, lastly, the entity key and the description value will be added to the corresponding section.

# Breakdown of contribution

**Everyone:**

1. Report writing
2. Bug fixing

**Pang Ka Ho:**

1. Data structure implementation
2. knowledge\_read()
3. chatbot\_is\_load()
4. chatbot\_do\_load()
5. Memory leak testing (only using Visual Studio 2019 CRT library)

**Zarif and Shireen:**

1. chatbot\_is\_question()
2. chatbot\_do\_question()
3. chatbot\_is\_reset()
4. chatbot\_do\_reset()
5. knowledge\_put()
6. knowledge\_reset()

**Clarence and Qixian:**

1. chatbot\_is\_smalltalk()
2. chatbot\_do\_smalltalk()
3. chatbot\_is\_save()
4. chatbot\_do\_save()
5. knowledge\_get()
6. knowledge\_write()