Homework 3

CS51510-001, Fall 2025

Purdue University Northwest

09/26/2025

Name	Email
Raaj Patel	**Pate2682@pnw.edu**

Table of Contents

Table of Tables

Table of Figures

ABSTRACT

This homework focused on building a memory database that works entirely in system memory using the concept of linked lists. Instead of using built-in data structures or database libraries, the assignment required designing nodes manually and linking them together to represent rows from a CSV dataset. Once the data was loaded into memory, two well-known sorting algorithms bubble sort and insertion sort were implemented to organize the records based on selected columns. The sorted results were then written back to new CSV files using recursive export functions.

To strengthen the learning experience, the same task was completed in two different programming languages, Python and Go. Python allowed for rapid development because of its simple syntax and rich standard library, while Go demanded explicit struct definitions and pointer handling, giving clearer visibility into how data is managed at the memory level. Comparing the two implementations showed how language design can influence development speed, clarity, and debugging.

In addition to coding, the assignment included analysing the efficiency of both sorting algorithms by monitoring CPU and disk usage on Ubuntu. The performance measurements confirmed that bubble sort consumed more resources due to repeated passes, while insertion sort was more efficient when the dataset was already partially ordered.

Overall, this work connected theoretical concepts from data structures with practical implementation. It provided hands-on practice in recursion, sorting, and memory management, while also offering insight into how the same algorithms behave differently across languages. The combination of Python and Go implementations, along with performance analysis, made the assignment a valuable exercise in both programming and algorithm evaluation.

INTRODUCTION

The purpose of this homework was to gain practical experience in building a small database that works entirely in memory using linked lists. Instead of depending on built-in database systems or library functions, I had to design a data structure from scratch that could read rows from a CSV file, link them together as nodes, and then sort them using classical algorithms. This process helped me understand not only how data is stored and traversed in memory, but also how algorithm design directly affects performance.

For this homework, I chose two programming languages: Python and Go. Both languages implement the same idea of a singly linked list but do so in different ways. Python provided a simpler coding experience with less focus on type declarations, making it easy to quickly test sorting and recursive functions. Go, on the other hand, required explicit struct definitions and pointer handling, which offered a deeper look into how data is represented in memory. Working in two languages allowed me to see how the same problem can be approached with different tools and design philosophies.

The linked list served as the foundation of the "memory database." Each student record from the dataset was loaded into a node, and nodes were connected one after another starting from the head. Once the list was built, bubble sort and insertion sort were applied directly on the nodes to arrange the data by specific columns such as "age" and "absences." Both sorts were implemented without using auxiliary arrays, which reinforced the principle of working only within the constraints of a linked list. After sorting, the results were exported back into CSV files through recursive writing functions, showing how data could be stored, manipulated, and output in a complete cycle.

Another important part of this homework was performance analysis. I ran the Python and Go programs on Ubuntu and used Linux commands to monitor CPU and disk usage. By comparing the two algorithms, I observed how bubble sort consumed more resources due to its repetitive passes through the list, while insertion sort performed better with partially ordered data. This

exercise gave me a clear understanding of the trade-offs between algorithm simplicity and efficiency.

Overall, this introduction sets the stage for the rest of the report. It explains the motivation for building a memory database from the ground up, the choice of Python and Go for implementation, the role of linked lists in managing records, the integration of sorting algorithms, and the importance of measuring performance. The remaining sections go step by step through the development, results, and analysis to demonstrate what was learned from this assignment.

1. DOWNLOAD STUDENT-DATA.CSV

As the first step of the homework, I needed to download the dataset that would be used to build the memory database. The file provided was student-data.csv, which contains rows of student information. This dataset served as the input for testing the linked list operations and done the homework more realistic compared to writing dummy data manually.

The professor shared a download link for the dataset, and I accessed it directly from inside the Ubuntu virtual machine. Using the browser in Ubuntu, I logged into the link and downloaded the file. Once the compressed file was saved, I used the terminal to unzip it and place the CSV in my working directory. After extraction, I verified the dataset by checking its name, size, and previewing the first few rows.

The process was completed in two simple steps. In **Step 1**, I opened the download link in Ubuntu and saved the file to the Downloads folder. In **Step 2**, I used terminal commands to unzip the file and confirm its contents. Screenshots of each step are included to show the process clearly.

Step 1: Open Link and Download File

- Opened the professor's download link inside the Ubuntu virtual machine.
- Logged in as required and saved the compressed dataset file to the system.

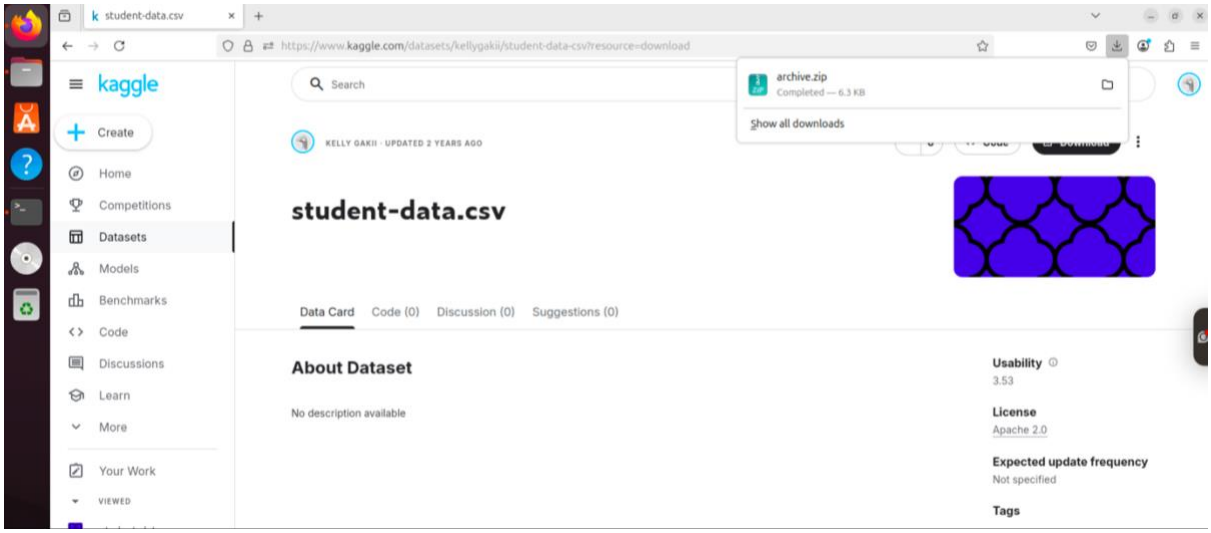


Figure 1 – Download student-data.csv file successfully

Step 2: Unzip and verify the data

- Navigated to the folder where the file was saved.

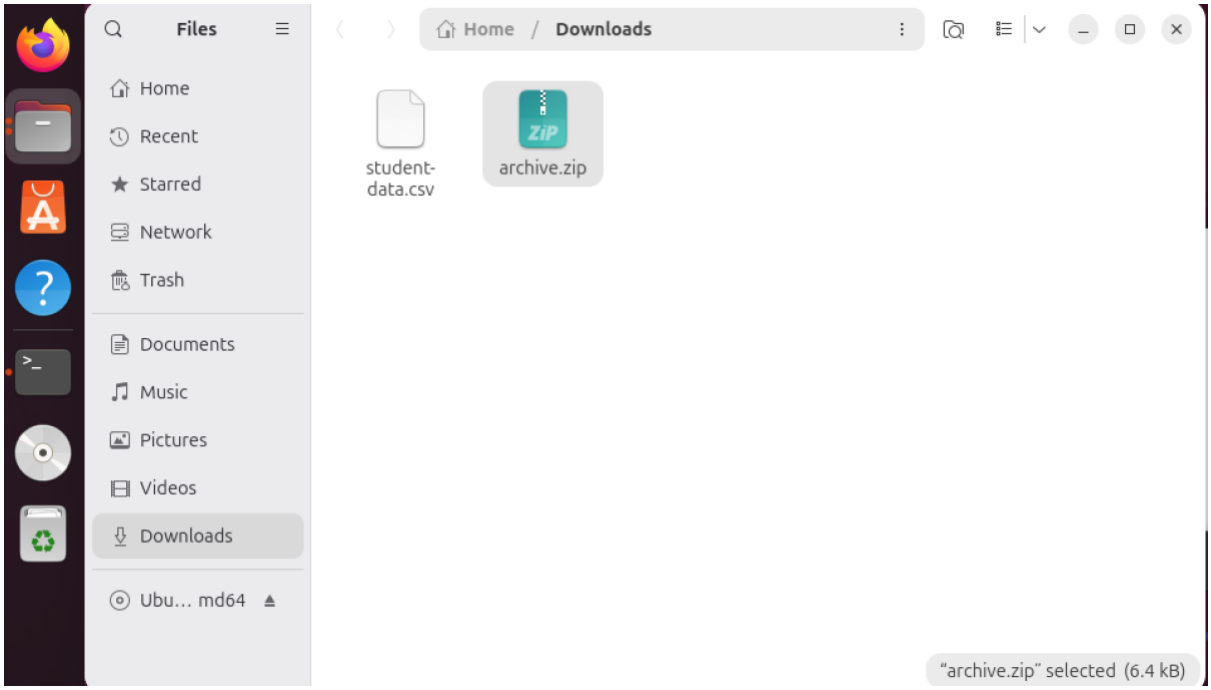


Figure 2 – Download and extra file and get data

- Unzipped the downloaded file using the unzip command "unzip archive.zip".
- Verified the dataset by running "head -n 5 student-data.csv". In command 'n' flag stands for number of rows want to see in terminal.

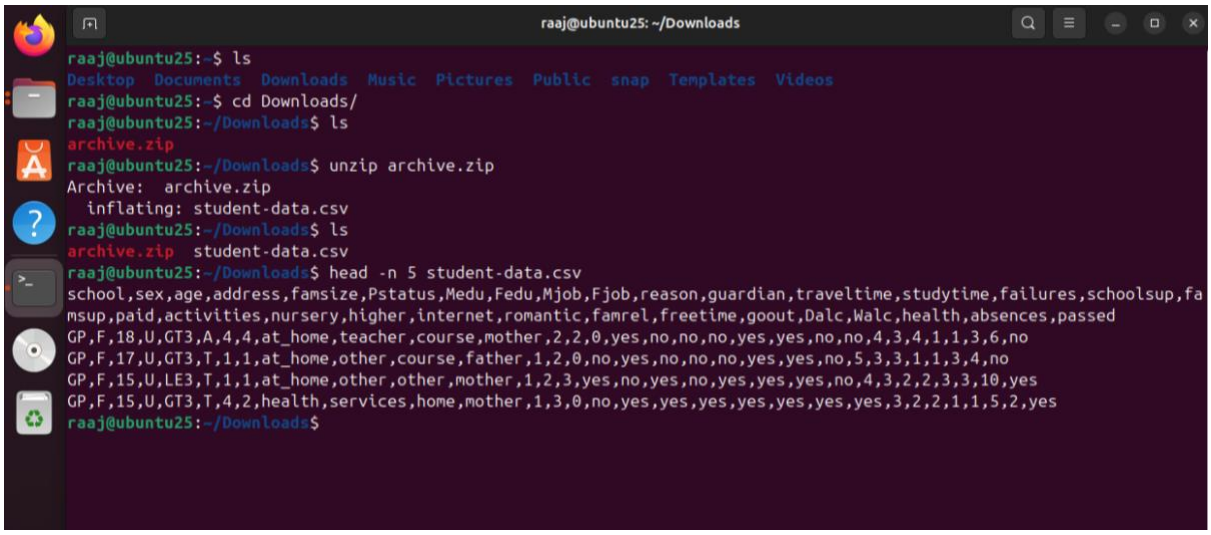


Figure 3 – Terminal that shows unzip file and verify database is download is correct

2. DEVELOP A "MEMORY DATABASE" WITH THE SINGLE LINKED LIST.

The goal of this task was to design a simple in-memory database from scratch using the concept of a singly linked list. A singly linked list is a basic data structure where each element, called a node, contains two parts: the actual data and a pointer to the next node. The list starts at a special node called the head, and every operation whether inserting, deleting, or searching happens by traversing node by node until the right element is found. This structure was chosen because it demonstrates the underlying mechanics of data management in the simplest form, and it is particularly useful for learning how records are connected in memory.

To keep the homework organized, I first created a dedicated folder named “homework3” under the Algorithm directory. The dataset, “student-data.csv”, was moved from the “Downloads” folder into this working folder so we could access it with relative paths instead of typing long absolute paths each time. After preparing the workspace, implemented the memory database in two languages: Python and Go, to compare how different languages handle the same logic.

In Python, I created a file “memory_database.py” where the linked list was implemented. That defined a Node class to hold data and a pointer next. To work with CSV files, we used Python’s built-in csv module, and we also installed the csvkit package for quick terminal inspection of CSV data. In Go, the same concept was applied using structs and pointers. A Node struct contained a map for row data and a pointer to the next node, while a Database struct managed the head of the list. Go’s standard encoding/csv library handled file operations.

Overall, this exercise was not about building a high-performance database, but about understanding how fundamental data structures can be used to simulate database-like functionality. By walking through each node, manually managing pointers, and explicitly coding insertions and deletions, we gained practical insight into how memory is organized and how higher-level database systems operate under the hood.

- Create a folder name “homework 3” at algorithm.

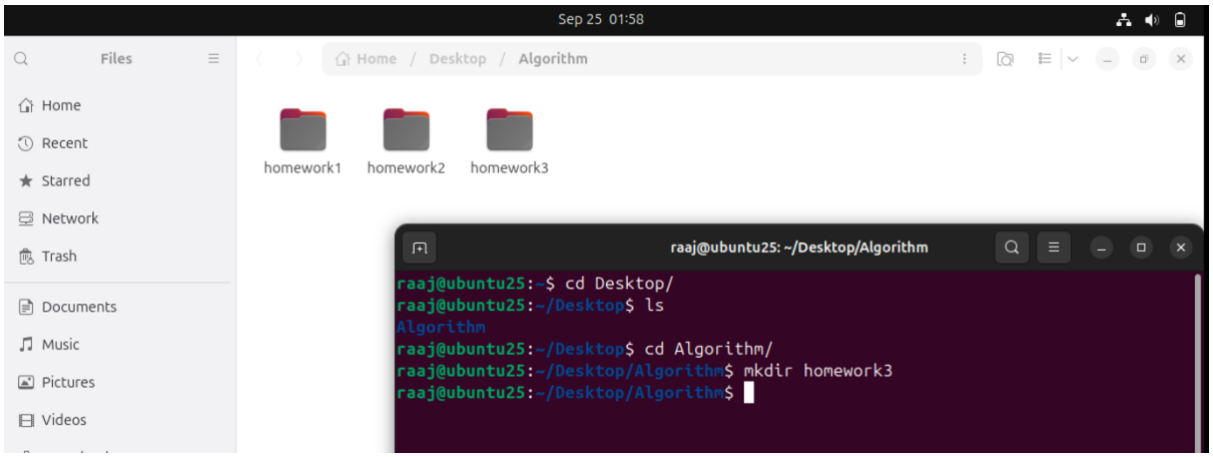


Figure 4 – Make folder name homework3

- Move “student-data.csv” file from download folder to homework3 folder for easy to access while fetching data from csv file and need not require giving full path.

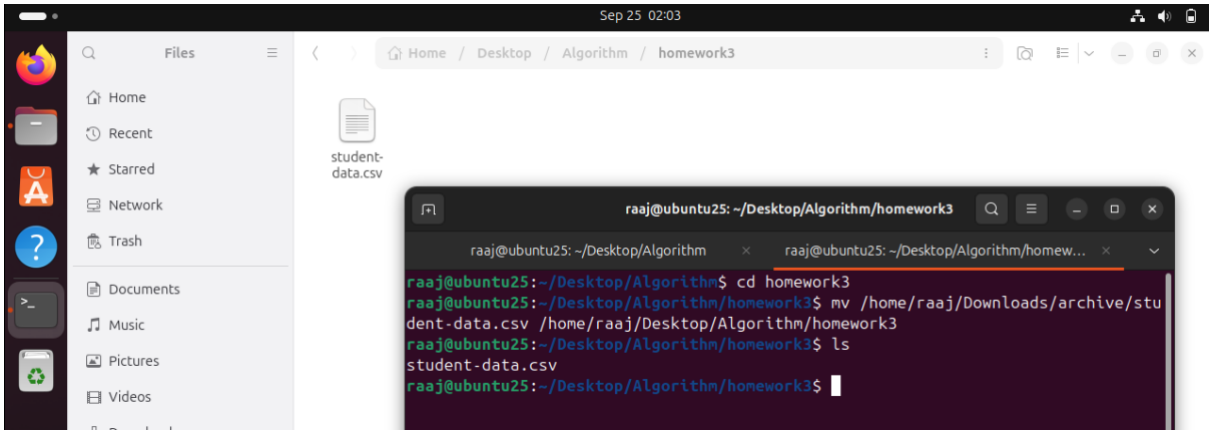


Figure 5 – Move csv file from download folder to homework3

Python code

Now create python folder using “mkdir” command which stands for make directory and called that “homework3”. Then using “touch” command for creating empty file name “memory_database.py”

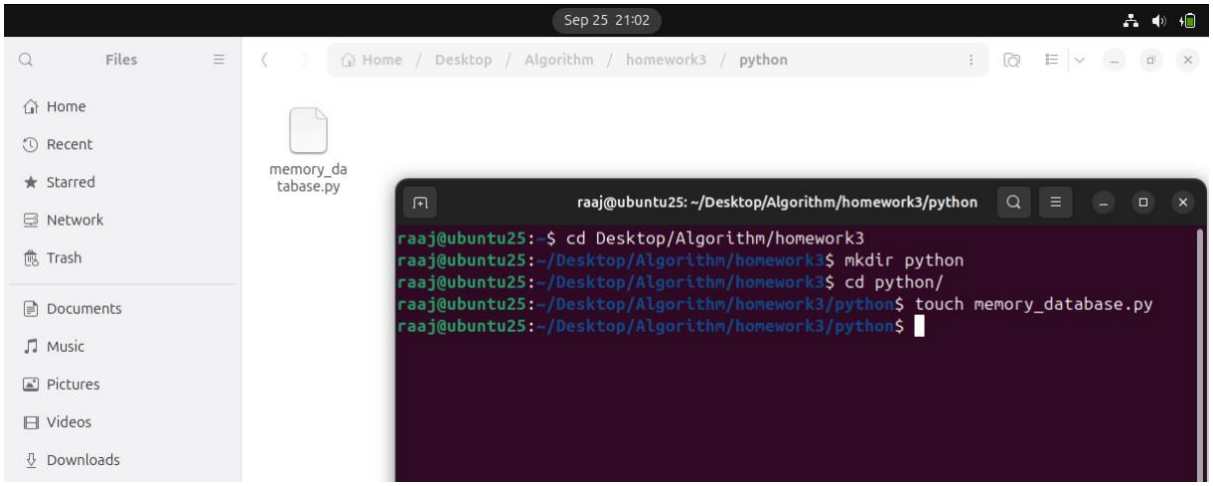


Figure 6 – Make memory_database.py file using touch command

i. Install required python modules

- **Csvkit** modules help to read and write csv file in python so installing csvkit by "apt-get install python3-csvkit" these command.

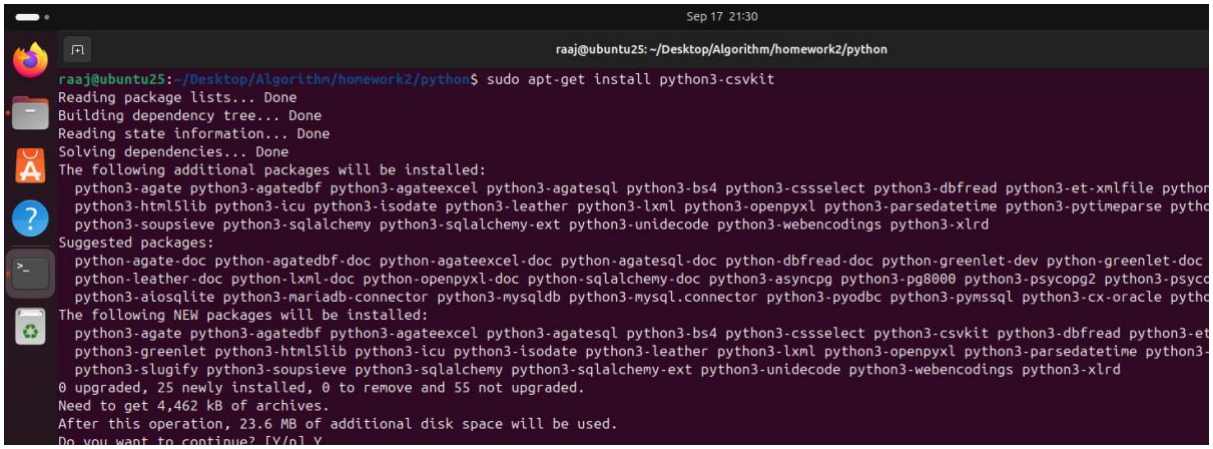


Figure 7 – Install cvskit for python

ii. Write python code

- In this task we implemented a simple memory database using a singly linked list in Python. Each record of data is stored in a node, and all nodes are connected one after another, starting from the head of the list.
- The code shown in Figure 8 is a simple demonstration of how to build a memory database using a singly linked list.

1. Node Class

- The Node class is the basic building block of the list. Each node holds two things: the actual record, which in this case is a data, and a pointer called next, which links to the following node. When a node is first created, its next pointer is set to None, meaning it does not yet connect to anything. This setup makes it easy to store one row of data at a time and later link it to other nodes, forming a continuous chain.

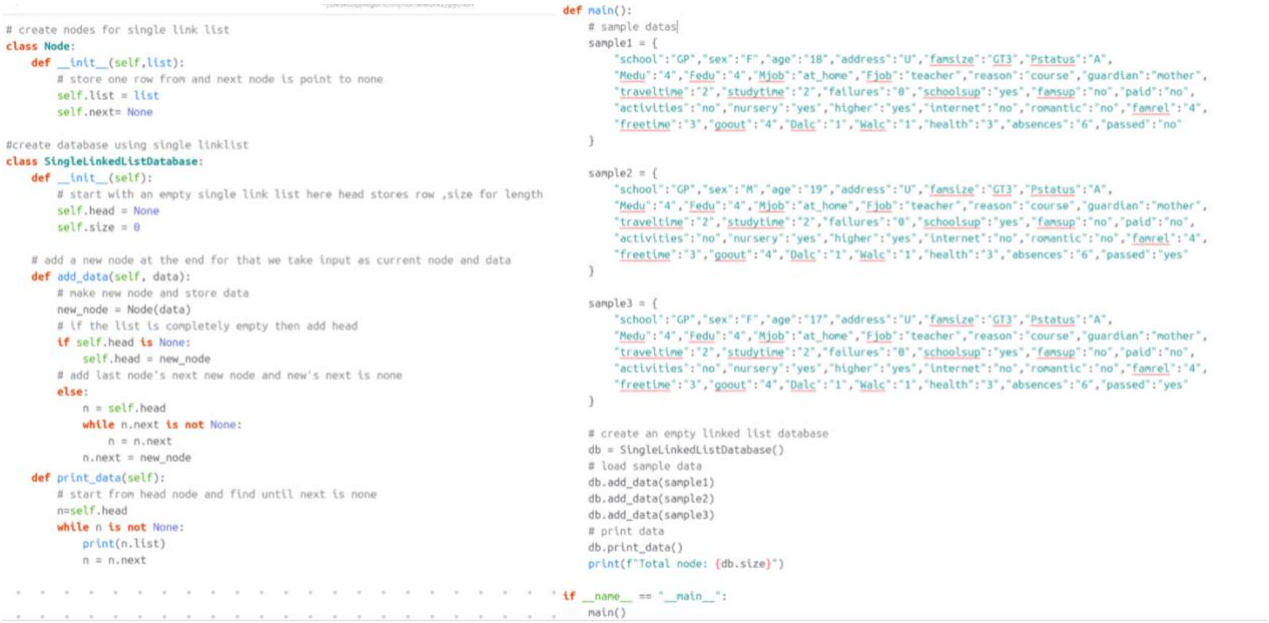
2. SingleLinkedListDatabase Class

The SingleLinkedListDatabase class is responsible for managing the list of nodes. When it is initialized, the list is empty with head = None and the size set to zero. It provides two main functions:

- add_data(): This function inserts a new record into the list. If the list is empty, the new node becomes the head. Otherwise, the function goes through the list until it finds the last node and then attaches the new one at the end. Each time a record is added, the size counter increases by one.
- print_data(): This function starts at the head node and prints out the data stored in every node, moving from one to the next using the next pointer. It continues until there are no more nodes left. This makes it easy to see all the records that are currently in memory.

3. Sample Data and Main Program

To test the program, a few sample dictionaries were created to represent student records, with fields like roll number, name, subject, marks, and password. A database object was then created from the SingleLinkedListDatabase class. The sample records were added one by one using the add_data () function. Finally, the print_data(): function was used to print all stored records, and the size attribute confirmed how many records were saved in total.

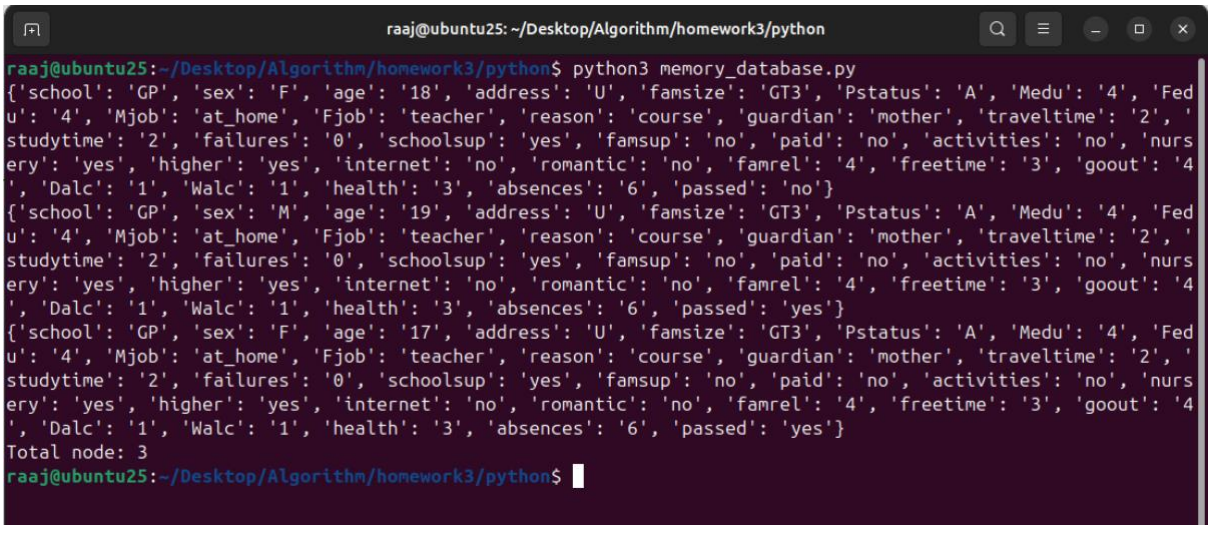


```python
# create nodes for single link list
class Node:
    def __init__(self,list):
        # store one row from and next node is point to none
        self.list = list
        self.next= None

#create database using single linklist
class SingleLinkedListDatabase:
    def __init__(self):
        # start with an empty single link list here head stores row ,size for length
        self.head = None
        self.size = 0

    # add a new node at the end for that we take input as current node and data
    def add_data(self, data):
        # make new node and store data
        new_node = Node(data)
        # if the list is completely empty then add head
        if self.head is None:
            self.head = new_node
        # add last node's next new node and new's next is none
        else:
            n = self.head
            while n.next is not None:
                n = n.next
            n.next = new_node
    def print_data(self):
        # start from head node and find until next is none
        n=self.head
        while n is not None:
            print(n.list)
            n = n.next
```

```python
def main():
    # sample datas
    sample1 = {
        "school":"GP","sex":"F","age":"18","address":"U","famsize":"GT3","Pstatus":"A",
        "Medu":"4","Fedu":"4","Mjob":"at_home","Fjob":"teacher","reason":"course","guardian":"mother",
        "traveltime":"2","studytime":"2","failures":"0","schoolsup":"yes","famsup":"no","paid":"no",
        "activities":"no","nursery":"yes","higher":"yes","internet":"no","romantic":"no","famrel":"4",
        "freetime":"3","goout":"4","Dalc":"1","Walc":"1","health":"3","absences":"6","passed":"no"
    }

    sample2 = {
        "school":"GP","sex":"M","age":"19","address":"U","famsize":"GT3","Pstatus":"A",
        "Medu":"4","Fedu":"4","Mjob":"at_home","Fjob":"teacher","reason":"course","guardian":"mother",
        "traveltime":"2","studytime":"2","failures":"0","schoolsup":"yes","famsup":"no","paid":"no",
        "activities":"no","nursery":"yes","higher":"yes","internet":"no","romantic":"no","famrel":"4",
        "freetime":"3","goout":"4","Dalc":"1","Walc":"1","health":"3","absences":"6","passed":"yes"
    }

    sample3 = {
        "school":"GP","sex":"F","age":"17","address":"U","famsize":"GT3","Pstatus":"A",
        "Medu":"4","Fedu":"4","Mjob":"at_home","Fjob":"teacher","reason":"course","guardian":"mother",
        "traveltime":"2","studytime":"2","failures":"0","schoolsup":"yes","famsup":"no","paid":"no",
        "activities":"no","nursery":"yes","higher":"yes","internet":"no","romantic":"no","famrel":"4",
        "freetime":"3","goout":"4","Dalc":"1","Walc":"1","health":"3","absences":"6","passed":"yes"
    }

    # create an empty linked list database
    db = SingleLinkedListDatabase()
    # load sample data
    db.add_data(sample1)
    db.add_data(sample2)
    db.add_data(sample3)
    # print data
    db.print_data()
    print(f"Total node: {db.size}")

if __name__ == "__main__":
    main()
```

Figure 8 – Screenshot of python code for task2

- Output of code displays the result after running the program. The terminal output shows each student record that was added to the linked list, printed one after another as the add_data() function traverses through the nodes. At the bottom, the program also prints the total number of nodes, confirming that all the sample records were successfully stored in memory. This output verifies that the linked list structure works correctly for adding and retrieving student data.



```
raaj@ubuntu25: ~/Desktop/Algorithm/homework3/python
raaj@ubuntu25:~/Desktop/Algorithm/homework3/python$ python3 memory_database.py
{'school': 'GP', 'sex': 'F', 'age': '18', 'address': 'U', 'famsize': 'GT3', 'Pstatus': 'A', 'Medu': '4', 'Fed
u': '4', 'Mjob': 'at_home', 'Fjob': 'teacher', 'reason': 'course', 'guardian': 'mother', 'traveltime': '2', '
studytime': '2', 'failures': '0', 'schoolsup': 'yes', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nurs
ery': 'yes', 'higher': 'yes', 'internet': 'no', 'romantic': 'no', 'famrel': '4', 'freetime': '3', 'goout': '4
', 'Dalc': '1', 'Walc': '1', 'health': '3', 'absences': '6', 'passed': 'no'}
{'school': 'GP', 'sex': 'M', 'age': '19', 'address': 'U', 'famsize': 'GT3', 'Pstatus': 'A', 'Medu': '4', 'Fed
u': '4', 'Mjob': 'at_home', 'Fjob': 'teacher', 'reason': 'course', 'guardian': 'mother', 'traveltime': '2', '
studytime': '2', 'failures': '0', 'schoolsup': 'yes', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nurs
ery': 'yes', 'higher': 'yes', 'internet': 'no', 'romantic': 'no', 'famrel': '4', 'freetime': '3', 'goout': '4
', 'Dalc': '1', 'Walc': '1', 'health': '3', 'absences': '6', 'passed': 'yes'}
{'school': 'GP', 'sex': 'F', 'age': '17', 'address': 'U', 'famsize': 'GT3', 'Pstatus': 'A', 'Medu': '4', 'Fed
u': '4', 'Mjob': 'at_home', 'Fjob': 'teacher', 'reason': 'course', 'guardian': 'mother', 'traveltime': '2', '
studytime': '2', 'failures': '0', 'schoolsup': 'yes', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nurs
ery': 'yes', 'higher': 'yes', 'internet': 'no', 'romantic': 'no', 'famrel': '4', 'freetime': '3', 'goout': '4
', 'Dalc': '1', 'Walc': '1', 'health': '3', 'absences': '6', 'passed': 'yes'}
Total node: 3
raaj@ubuntu25:~/Desktop/Algorithm/homework3/python$
```

Figure 9 – Output of python code for task2

Go code

- Install go in ubuntu by using command “sudo apt-get install golang-go”.

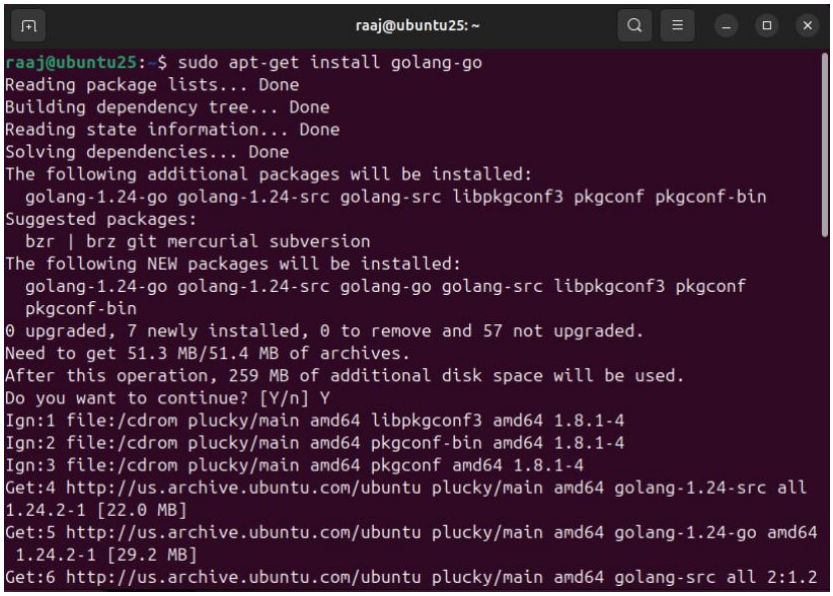


Figure 10 – Download environment for go

- Now create go folder using “mkdir” command which stands for make directory and called that “homework3”. Then using “touch” command for creating empty file name “memory_database.go”.

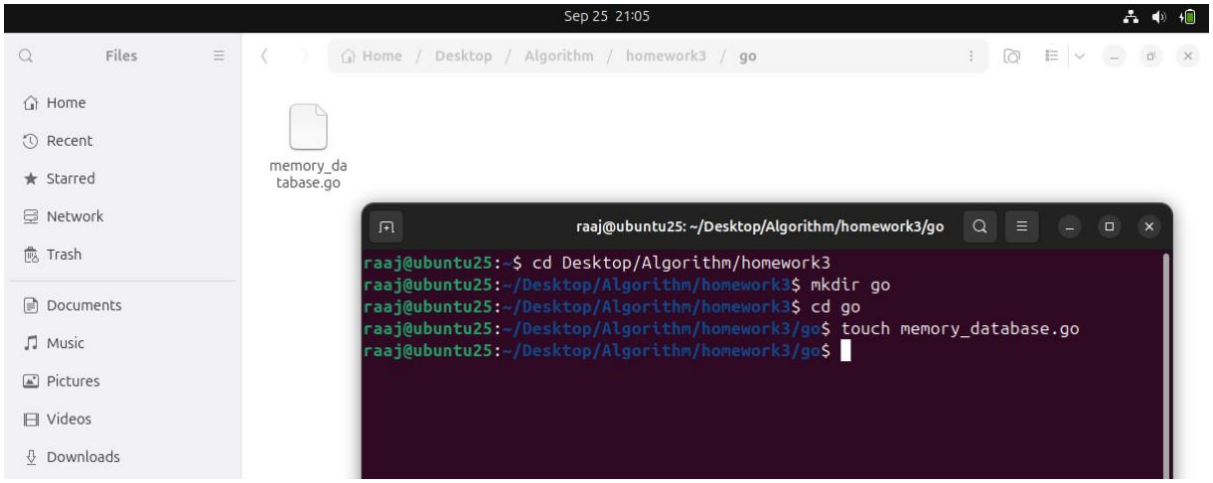


Figure 11 – Make memory_database.go file using touch command

i. Install required go modules

The go module that are required for this code is already available while installing so not require to download any other modules.

ii. Write go code

In this I implemented a simple memory database in Go using a singly linked list. Each record was stored inside a Node struct, which contains two parts: a map for the student data and a pointer to the next node. When a new node is created, its pointer is set to nil, and it is later connected to the chain of nodes that begins at the head of the list. To manage all the nodes, I created a SingleLinkedListDatabase struct with two fields: head, which marks the start of the list, and size, which keeps track of how many records are stored.

This struct also provides two main methods. The first one, addData, is used to insert new records. If the list is empty, the new node becomes the head; otherwise, the method walks through the list until it finds the last node and attaches the new node at the end, increasing the size count. The second method, printData, starts from the head and follows the pointers from node to node, printing the data until the end of the list is reached.

```go
package main

import ("fmt")

// create nodes for single linked list
type Node struct {
	// store one row and pointer to next node
	list map[string]string
	next *Node
}

// create database using single linked list
type SingleLinkedListDatabase struct {
	// start with an empty single linked list
	head *Node
	size int
}

// add a new node at the end for that we take input as data
func (db *SingleLinkedListDatabase) addData(data map[string]string) {
	// make new node and store data
	newNode := &Node{list: data, next: nil}

	// if the list is completely empty then add head
	if db.head == nil {
		db.head = newNode
	} else {
		// add last node's next to new node
		n := db.head
		for n.next != nil {
			n = n.next
		}
		n.next = newNode
	}
	db.size++
}

// print database loaded in linked list
func (db *SingleLinkedListDatabase) printData() {
	// start from head node and find until next is nil
	n := db.head
	for n != nil {
		fmt.Println(n.list)
		n = n.next
	}
}
```

```go
func main() {
	// sample datas
	sample1 := map[string]string{
		"school": "GP", "sex": "F", "age": "18", "address": "U", "famsize": "GT3", "Pstatus": "A",
		"Medu": "4", "Fedu": "4", "Mjob": "at_home", "Fjob": "teacher", "reason": "course", "guardian": "mother",
		"traveltime": "2", "studytime": "2", "failures": "0", "schoolsup": "yes", "famsup": "no", "paid": "no",
		"activities": "no", "nursery": "yes", "higher": "yes", "internet": "no", "romantic": "no", "famrel": "4",
		"freetime": "3", "goout": "4", "Dalc": "1", "Walc": "1", "health": "3", "absences": "6", "passed": "no",
	}

	sample2 := map[string]string{
		"school": "GP", "sex": "M", "age": "19", "address": "U", "famsize": "GT3", "Pstatus": "A",
		"Medu": "4", "Fedu": "4", "Mjob": "at_home", "Fjob": "teacher", "reason": "course", "guardian": "mother",
		"traveltime": "2", "studytime": "2", "failures": "0", "schoolsup": "yes", "famsup": "no", "paid": "no",
		"activities": "no", "nursery": "yes", "higher": "yes", "internet": "no", "romantic": "no", "famrel": "4",
		"freetime": "3", "goout": "4", "Dalc": "1", "Walc": "1", "health": "3", "absences": "6", "passed": "yes",
	}

	sample3 := map[string]string{
		"school": "GP", "sex": "F", "age": "17", "address": "U", "famsize": "GT3", "Pstatus": "A",
		"Medu": "4", "Fedu": "4", "Mjob": "at_home", "Fjob": "teacher", "reason": "course", "guardian": "mother",
		"traveltime": "2", "studytime": "2", "failures": "0", "schoolsup": "yes", "famsup": "no", "paid": "no",
		"activities": "no", "nursery": "yes", "higher": "yes", "internet": "no", "romantic": "no", "famrel": "4",
		"freetime": "3", "goout": "4", "Dalc": "1", "Walc": "1", "health": "3", "absences": "6", "passed": "yes",
	}

	// create an empty linked list database
	db := SingleLinkedListDatabase{}

	// load sample data
	db.addData(sample1)
	db.addData(sample2)
	db.addData(sample3)

	// print data
	db.printData()
	fmt.Printf("Total node: %d\n", db.size)
}
```

Figure 12 – Screenshot of code go for task2

To test the program, I created a few sample records using Go maps with fields like roll number, name, subject, marks, and pass status. These records were added one by one to the database using addData. Finally, I called printData to display the records in order, followed by printing the size of the list to confirm how many nodes were stored in memory. The output showed each record clearly and ended with the total node count, proving that the singly linked list was functioning correctly. This implementation demonstrates how linked lists can be built and

traversed in Go using structs and pointers, achieving the same behaviour as the Python version but with stricter typing and explicit memory handling.

Output of code shows the result after running the Go program. The terminal prints each student record that was inserted into the linked list, one after another, as the printData() function traverses through the nodes. At the end, the program also displays the total number of nodes, confirming that all sample records were added successfully. This output demonstrates that the linked list in Go is functioning as intended, with nodes linked correctly, and verifies that data can be stored, traversed, and retrieved from memory in sequence.

Figure 13 – Output of code go for task2

3. LOAD THE STUDENT-DATA.CSV FILE INTO THE MEMORY DATABASE

When we start building the memory database, the very first step is to take the CSV file and read its contents into memory. Each row in the CSV represents one student record, and we want to transform each of those rows into a node in our linked list. The list grows one node at a time, with every node holding a single student's details and a pointer to the next. This gives us a structure that mirrors the file but lives completely in memory.

The database then provides three main functions that make it useful and dynamic. The first function is loading data from CSV into the linked list. This is done recursively: the program reads a row, creates a node for it, attaches the node to the chain, and then calls itself for the next row. This continues until every row has been turned into a node, so the in-memory database is an exact reflection of the CSV file.

i. Python Implementation

The process starts in the ReadAndWriteCSV.read_csv(path) function. This function uses Python's csv.DictReader to read the file and convert each line into a dictionary, where the column headers are the keys. It also assigns a unique "key" value to each row so that every record can be identified later. It finally returns two things: a list of row dictionaries and the headers.

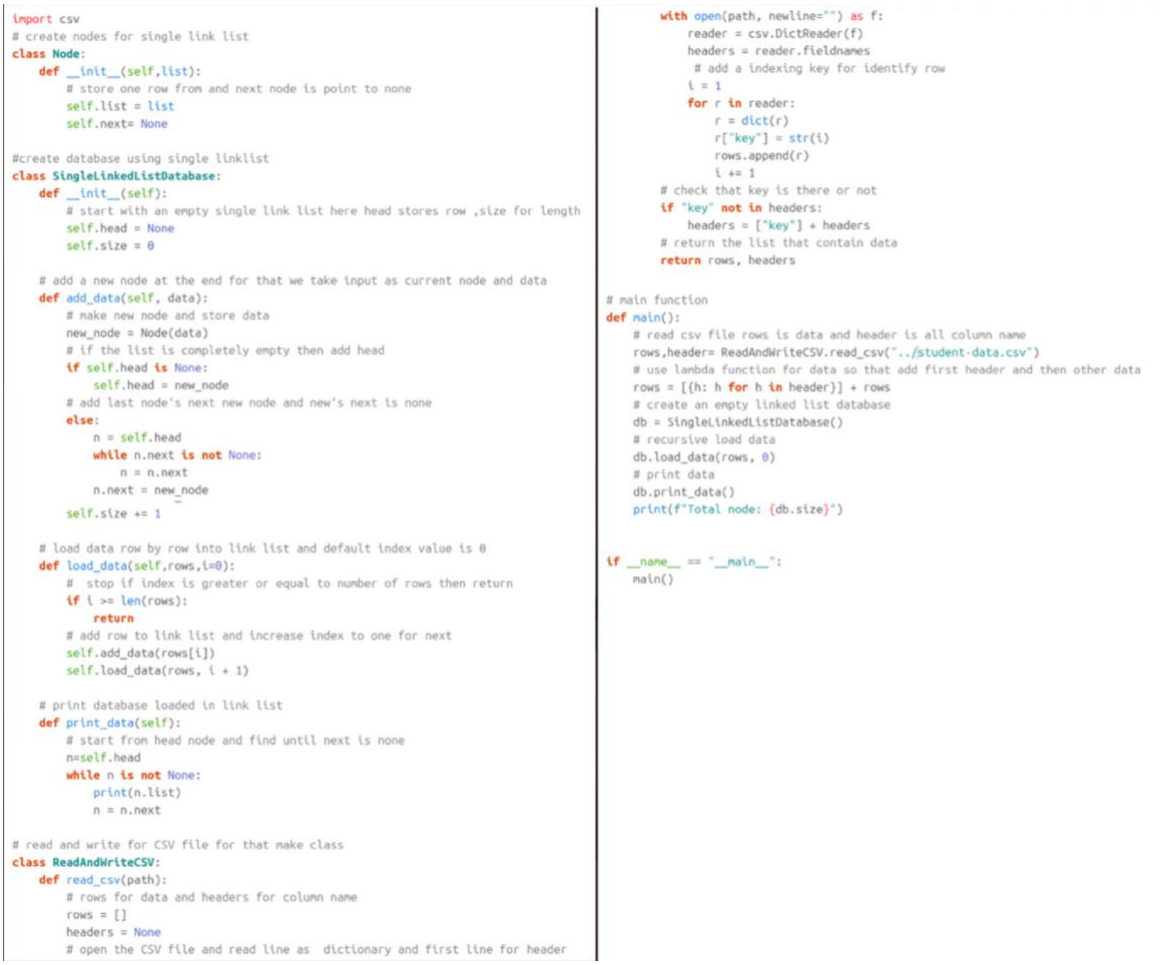
Once the rows are prepared, the SingleLinkedListDatabase.load_data(rows, i=0) method is called This is where recursion comes in. The function checks:

1. If i is greater than or equal to the number of rows, recursion stops (base case).

2. Otherwise, it calls self.add_data(rows[i]) to insert the current row as a new node at the end of the list.

3. Then it calls itself again with i+1, moving to the next row.

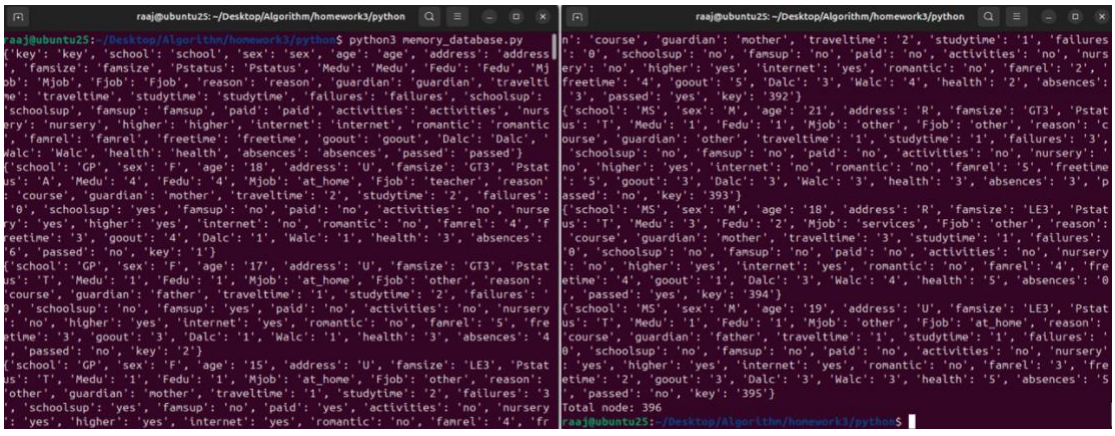
Step by step, the recursion creates one node for each student record until all rows are processed. At the end, the linked list in memory mirrors the CSV file completely.



```python
import csv
# create nodes for single link list
class Node:
    def __init__(self,list):
        # store one row from and next node is point to none
        self.list = list
        self.next= None

#create database using single linklist
class SingleLinkedListDatabase:
    def __init__(self):
        # start with an empty single link list here head stores row ,size for length
        self.head = None
        self.size = 0

    # add a new node at the end for that we take input as current node and data
    def add_data(self, data):
        # make new node and store data
        new_node = Node(data)
        # if the list is completely empty then add head
        if self.head is None:
            self.head = new_node
        # add last node's next new node and new's next is none
        else:
            n = self.head
            while n.next is not None:
                n = n.next
            n.next = new_node
        self.size += 1

    # load data row by row into link list and default index value is 0
    def load_data(self,rows,i=0):
        #  stop if index is greater or equal to number of rows then return
        if i >= len(rows):
            return
        # add row to link list and increase index to one for next
        self.add_data(rows[i])
        self.load_data(rows, i + 1)

    # print database loaded in link list
    def print_data(self):
        # start from head node and find until next is none
        n=self.head
        while n is not None:
            print(n.list)
            n = n.next

# read and write for CSV file for that make class
class ReadAndWriteCSV:
    def read_csv(path):
        # rows for data and headers for column name
        rows = []
        headers = None
        # open the CSV file and read line as  dictionary and first line for header
        with open(path, newline="") as f:
            reader = csv.DictReader(f)
            headers = reader.fieldnames
            # add a indexing key for identify row
            i = 1
            for r in reader:
                r = dict(r)
                r["key"] = str(i)
                rows.append(r)
                i += 1
        # check that key is there or not
        if "key" not in headers:
            headers = ["key"] + headers
        # return the list that contain data
        return rows, headers

# main function
def main():
    # read csv file rows is data and header is all column name
    rows,header= ReadAndWriteCSV.read_csv("../student-data.csv")
    # use lambda function for data so that add first header and then other data
    rows = [{h: h for h in header}] + rows
    # create an empty linked list database
    db = SingleLinkedListDatabase()
    # recursive load data
    db.load_data(rows, 0)
    # print data
    db.print_data()
    print(f"Total node: {db.size}")


if __name__ == "__main__":
    main()
```

Figure 14 – Screenshot of python code until task 3

- Output of code after the program runs, the add_data() function keeps inserting each student record into the linked list one by one. After all the data is loaded, the print_data() function is called. This function begins at the head of the list and moves through every node until the end, printing the data stored in each node along the way. That can see in figure 15 . Where 396 rows loaded in single linklist.



```
raaj@ubuntu25: ~/Desktop/Algorithm/homework3/python
raaj@ubuntu25:~/Desktop/Algorithm/homework3/python$ python3 memory_database.py
{'key': 'key', 'school': 'school', 'sex': 'sex', 'age': 'age', 'address': 'address
', 'famsize': 'famsize', 'Pstatus': 'Pstatus', 'Medu': 'Medu', 'Fedu': 'Fedu', 'Mj
ob': 'Mjob', 'Fjob': 'Fjob', 'reason': 'reason', 'guardian': 'guardian', 'travelti
me': 'traveltime', 'studytime': 'studytime', 'failures': 'failures', 'schoolsup':
'schoolsup', 'famsup': 'famsup', 'paid': 'paid', 'activities': 'activities', 'nurs
ery': 'nursery', 'higher': 'higher', 'internet': 'internet', 'romantic': 'romantic
', 'famrel': 'famrel', 'freetime': 'freetime', 'goout': 'goout', 'Dalc': 'Dalc', '
Walc': 'Walc', 'health': 'health', 'absences': 'absences', 'passed': 'passed'}
{'school': 'GP', 'sex': 'F', 'age': '18', 'address': 'U', 'famsize': 'GT3', 'Pstat
us': 'A', 'Medu': '4', 'Fedu': '4', 'Mjob': 'at_home', 'Fjob': 'teacher', 'reason'
: 'course', 'guardian': 'mother', 'traveltime': '2', 'studytime': '2', 'failures':
'0', 'schoolsup': 'yes', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nurse
ry': 'yes', 'higher': 'yes', 'internet': 'no', 'romantic': 'no', 'famrel': '4', 'f
reetime': '3', 'goout': '4', 'Dalc': '1', 'Walc': '1', 'health': '3', 'absences':
'6', 'passed': 'no', 'key': '1'}
{'school': 'GP', 'sex': 'F', 'age': '17', 'address': 'U', 'famsize': 'GT3', 'Pstat
us': 'T', 'Medu': '1', 'Fedu': '1', 'Mjob': 'at_home', 'Fjob': 'other', 'reason':
'course', 'guardian': 'father', 'traveltime': '1', 'studytime': '2', 'failures':
'0', 'schoolsup': 'no', 'famsup': 'yes', 'paid': 'no', 'activities': 'no', 'nursery
': 'no', 'higher': 'yes', 'internet': 'yes', 'romantic': 'no', 'famrel': '5', 'fre
etime': '3', 'goout': '3', 'Dalc': '1', 'Walc': '1', 'health': '3', 'absences': '4
', 'passed': 'no', 'key': '2'}
{'school': 'GP', 'sex': 'F', 'age': '15', 'address': 'U', 'famsize': 'LE3', 'Pstat
us': 'T', 'Medu': '1', 'Fedu': '1', 'Mjob': 'at_home', 'Fjob': 'other', 'reason':
'other', 'guardian': 'mother', 'traveltime': '1', 'studytime': '2', 'failures': '3
', 'schoolsup': 'yes', 'famsup': 'no', 'paid': 'yes', 'activities': 'no', 'nursery
': 'yes', 'higher': 'yes', 'internet': 'yes', 'romantic': 'no', 'famrel': '4', 'fr
n': 'course', 'guardian': 'mother', 'traveltime': '2', 'studytime': '1', 'failures
': '0', 'schoolsup': 'no', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nurs
ery': 'no', 'higher': 'yes', 'internet': 'yes', 'romantic': 'no', 'famrel': '2', '
freetime': '4', 'goout': '5', 'Dalc': '3', 'Walc': '4', 'health': '2', 'absences':
'3', 'passed': 'yes', 'key': '392'}
{'school': 'MS', 'sex': 'M', 'age': '21', 'address': 'R', 'famsize': 'GT3', 'Pstat
us': 'T', 'Medu': '1', 'Fedu': '1', 'Mjob': 'other', 'Fjob': 'other', 'reason': 'c
ourse', 'guardian': 'other', 'traveltime': '1', 'studytime': '1', 'failures': '3',
'schoolsup': 'no', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nursery':
no', 'higher': 'yes', 'internet': 'no', 'romantic': 'no', 'famrel': '5', 'freetime
': '5', 'goout': '3', 'Dalc': '3', 'Walc': '3', 'health': '3', 'absences': '3', 'p
assed': 'no', 'key': '393'}
{'school': 'MS', 'sex': 'M', 'age': '18', 'address': 'R', 'famsize': 'LE3', 'Pstat
us': 'T', 'Medu': '3', 'Fedu': '2', 'Mjob': 'services', 'Fjob': 'other', 'reason':
'course', 'guardian': 'mother', 'traveltime': '3', 'studytime': '1', 'failures':
'0', 'schoolsup': 'no', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nursery
': 'no', 'higher': 'yes', 'internet': 'yes', 'romantic': 'no', 'famrel': '4', 'fre
etime': '4', 'goout': '1', 'Dalc': '3', 'Walc': '4', 'health': '5', 'absences': '0
', 'passed': 'yes', 'key': '394'}
{'school': 'MS', 'sex': 'M', 'age': '19', 'address': 'U', 'famsize': 'LE3', 'Pstat
us': 'T', 'Medu': '1', 'Fedu': '1', 'Mjob': 'other', 'Fjob': 'at_home', 'reason':
'course', 'guardian': 'father', 'traveltime': '1', 'studytime': '1', 'failures':
'0', 'schoolsup': 'no', 'famsup': 'no', 'paid': 'no', 'activities': 'no', 'nursery'
: 'yes', 'higher': 'yes', 'internet': 'yes', 'romantic': 'no', 'famrel': '3', 'fre
etime': '2', 'goout': '3', 'Dalc': '3', 'Walc': '3', 'health': '5', 'absences': '5
', 'passed': 'no', 'key': '395'}
Total node: 396
raaj@ubuntu25:~/Desktop/Algorithm/homework3/python$
```

Figure 15 – Screenshot of terminal that shows code is run and print loaded link list

ii. Go Implementation

In Go, the reading is handled by ReadAndWriteCSV_read_csv(path). This function uses Go's encoding/csv package to open the file and parse the rows. It then builds a map[string]string for each record, again attaching a "key" field so that rows can be uniquely identified. The function returns both the slice of row maps and the headers. After loading the rows, the recursive part happens inside the method SingleLinkedListDatabase.load_data(rows []map[string]string, i int).

Just like in Python, it works by checking:

1.If i has reached the total number of rows, the recursion stops.

2.Otherwise, it calls db.add_data(rows[i]) to add the current row as a new node.

3.Then it calls db.load_data(rows, i+1) to continue with the next index.

4.This recursion continues until all rows are added. Each record is turned into a Node struct with a map[string]string for the row and a pointer to the next node.

```go
// read and write for csv file make class (here struct with functions)
type ReadAndWriteCSV struct{}

// read_csv opens csv file, returns rows and headers
func (rw *ReadAndWriteCSV) read_csv(path string) ([]map[string]string, []string, error) {
	// open CSV file
	file, err := os.Open(path)
	if err != nil {
		return nil, nil, err
	}
	defer file.Close()

	reader := csv.NewReader(file)
	// read all rows
	allRows, err := reader.ReadAll()
	if err != nil {
		return nil, nil, err
	}

	// first line is header
	headers := allRows[0]
	rows := []map[string]string{}

	// read each line and convert to map
	for i, row := range allRows[1:] {
		data := map[string]string{"key": fmt.Sprintf("%d", i+1)}
		for j, h := range headers {
			data[h] = row[j]
		}
		rows = append(rows, data)
	}

	// check that key is there or not
	found := false
	for _, h := range headers {
		if h == "key" {
			found = true
			break
		}
	}
	if !found {
		headers = append([]string{"key"}, headers...)
	}

	return rows, headers, nil
}
```

```go
}

// load data row by row into linked list and default index value is 0
func (db *SingleLinkedListDatabase) load_data(rows []map[string]string, i int) {
	// stop if index is greater or equal to number of rows then return
	if i >= len(rows) {
		return
	}
	// add row to linked list and increase index by one for next
	db.add_data(rows[i])
	db.load_data(rows, i+1)
}

// main function
func main() {
	// read csv file rows is data and header is all column name
	rw := ReadAndWriteCSV{}
	rows, headers, err := rw.read_csv("../student-data.csv")
	if err != nil {
		fmt.Println("Error:", err)
		return
	}

	// add first header row into data
	headerRow := map[string]string{}
	for _, h := range headers {
		headerRow[h] = h
	}
	rows = append([]map[string]string{headerRow}, rows...)

	// create an empty linked list database
	db := SingleLinkedListDatabase{}
	// recursive load data
	db.load_data(rows, 0)
	// print data
	db.print_data()
	fmt.Printf("Total node: %d\n", db.size)
```

Figure 16 – screenshot of go code until 3

Output of code after running the Go program shows how the linked list is populated and traversed. The addData() method is called repeatedly to insert each student record, creating a new node for every row in the dataset. Once all rows are loaded into memory, the printData() method starts from the head of the list and moves through each node, printing the contents until

it reaches the end. This sequential traversal ensures that every record stored in memory is displayed in the terminal. As shown in Figure 17 all 396 rows from the CSV file are successfully added to the singly linked list and printed out one by one, confirming that the Go implementation works correctly for both insertion and traversal.

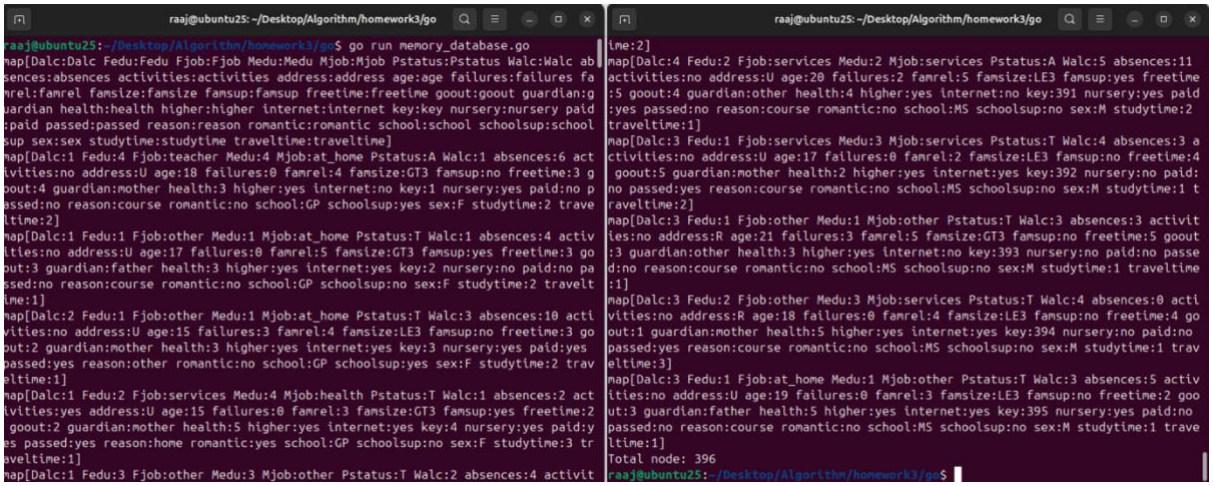


Figure 17 – screenshot of output code that shows all data of csv file loaded into link list

4. THE "MEMORY DATABASE" HAS TWO SORTING FUNCTIONS

After the student records were loaded into the linked list, the next requirement was to organize them based on specific columns, such as age or absences. Because the memory database was implemented with a singly linked list, the sorting process had to operate directly on nodes rather than converting the records into arrays. This approach highlights how fundamental algorithms can be adapted to linked list structures.

For this task, two well-known algorithms were implemented: bubble sort and insertion sort. Both are simple to understand yet effective enough to show how elements can be reordered inside the list by repeatedly comparing and swapping or inserting nodes. By applying these algorithms, the database demonstrates how classical sorting methods can be customized for node-based memory storage.

To support consistent comparisons, a cast function was added in both Python and Go implementations. This helper function ensures that numerical and string values are not mixed during comparisons. In practice, it converts numeric values into floats and string values into lowercase strings. With this preprocessing step, sorting remains stable and accurate regardless of whether the chosen column contains numbers or text.

Figure 18 illustrates the cast function written in Python (left) and Go (right). Both versions serve the same purpose: to normalize data values so that the linked list nodes can be reliably compared and reordered.

```python
# cast funnction helps values for comparisons so it never mix str & numbers
def cast(self, value):
    s = "" if value is None else str(value).strip()
    try:
        # 0 for float
        return (0, float(s))
    except ValueError:
        # 1 for string
        return (1, s.lower())
```

```go
// cast funnction helps values for comparisons so it never mix str & numbers
func (db *SingleLinkedListDatabase) cast(value string) (int, float64, string) {
	s := strings.TrimSpace(value)
	//  0 for float
	if s == "" {
		return 1, 0, ""
	}
	// 1 for string
	if f, err := strconv.ParseFloat(s, 64); err == nil {
		return 0, f, ""
	}
	return 1, 0, strings.ToLower(s)
}
```

Figure 18 – Screenshot of cast function

(a) Bubble sort

Bubble sort is one of the simplest algorithms, where each node is repeatedly compared with its next node and swapped if they are out of order. This process continues until no more swaps are needed, meaning the list is sorted. Although bubble sort is not efficient for large datasets, it provides a clear example of how comparisons and swaps can be performed directly on the contents of nodes.

In the Python implementation, bubble sort worked by iterating through the linked list multiple times. Each time, values from two adjacent nodes were compared using a helper casting function that ensured numerical and string values could be compared consistently. If the first value was larger, the node contents were swapped. This process was repeated until a full pass was completed without swaps.

In Go, bubble sort followed the same logic but required stricter handling of types. The algorithm repeatedly traversed the linked list, compared values from two nodes, and swapped the map contents when necessary. Go's strong typing made the implementation more explicit, but the outcome was the same: the list was sorted after multiple passes.

The main observation from bubble sort was that it consumed more CPU cycles when tested on Ubuntu because each pass required visiting almost every node, even if the list was already partially ordered. Still, it served as a good baseline for comparing performance against insertion sort.

i. Python Implementation

In Python, the sorting is carried out inside the bubble_sort(self, key) function. After the student records are loaded into the linked list, this method is called to reorder the nodes by a chosen column, such as "age". The function begins by checking if the list is empty or has only one node; in such cases, it immediately returns because no sorting is required.

The actual sorting process uses a while loop with a flag called swapped. If swaps are happening, the algorithm keeps scanning through the list. For each pass, it starts at the head node and compares the value of the current node against the next node. A helper method cast(...) ensures that the values can be compared correctly, whether they are numeric or string. If the first value is greater than the second, the contents of the two nodes are swapped. Once a full pass is completed without any swaps, the loop stops, and the linked list is sorted.

1. Check trivial case: If there are zero or one nodes, return immediately.
2. Traverse the list: Start from the head node and compare each node with its next node.
3. Comparison: Use cast(...) to normalize values so numbers and strings do not conflict.
4. Swap if needed: If the current node's value is greater, exchange the two node dictionaries.

5. Repeat passes: Continue until a full pass finish without swaps, which means sorting is complete.

```python
# bubble sort
def bubble_sort(self, key):
    # check if head is empty (0 or 1 node) then return
    if not self.head or not self.head.next:
        return

    swapped = True
    while swapped:
        swapped = False
        current = self.head
        while current and current.next:
            # convert values to a comparable form
            value1 = self.cast(current.list.get(key))
            value2 = self.cast(current.next.list.get(key))

            # if value1 is greater then swap
            if value1 > value2:
                current.list, current.next.list = current.next.list, current.list
                swapped = True

            current = current.next
```

```python
# main function
def main():
    # read csv file rows is data and header is all column name
    rows, header = ReadAndWriteCSV.read_csv("../student-data.csv")
    # create an empty linked list database
    db = SingleLinkedListDatabase()

    # recursive load data
    db.load_data(rows, 0)
    print(f"Data loaded in memory and total nodes: {db.size+1}")
    # bubble sort by "age" column
    db.bubble_sort("age")
    print("After bubble Sort by age")
    db.print_data()
    print(f"Data loaded in memory and total nodes: {db.size+1}")
```

Figure 19 – Screenshot of main and bubble_sort function in python

When the program is executed, the terminal first prints the message “Data loaded in memory and total nodes: 396” to confirm the dataset was successfully read. After calling bubble_sort("age"), the program prints “After bubble Sort by age” followed by the sorted student records. Each row is displayed as a dictionary, now arranged in ascending order of the age field. Finally, another message confirms the total number of nodes again.

As shown in Figure 20, the bubble sort function correctly arranges the records by age, and the output on the right side of the terminal clearly displays the sorted results, verifying that the Python implementation of bubble sort is functioning as expected.

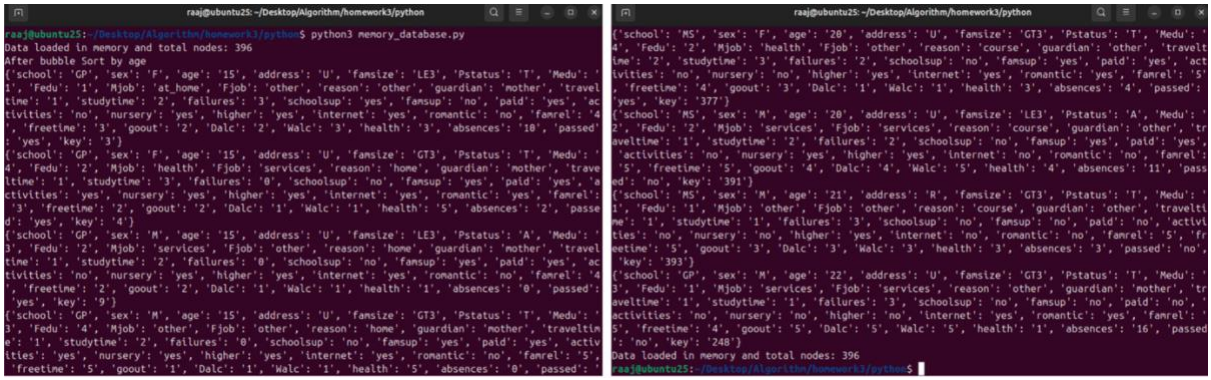


Figure 20 – output of terminal that shows link list is sorted by age in python

ii. Go Implementation

In Go, the sorting of the linked list is handled by the bubble_sort(key string) method inside the SingleLinkedListDatabase struct. Once the rows from student-data.csv are read into memory, this function is called to arrange the records by a chosen column such as "age". The method

begins by checking if the list is empty or contains only a single node. If that is the case, it immediately returns since no sorting is necessary.

The core of the algorithm uses a loop controlled by a swapped flag. The program repeatedly walks through the list, comparing each node with its next node. To handle different types of values, the helper function cast(...) converts the string data into a form that can be compared: numbers are parsed into floats, while other values are treated as lowercase strings. If the current node is greater than the next node, the contents of the two nodes are swapped. The process continues until a complete pass is made without any swaps, which indicates that the list is fully sorted.

1. Check trivial case: If the head is nil or only one node exists, return immediately.
2. Traverse the list: Starting from the head node, compare the current node's value with the next node's value.
3. Comparison logic: Use cast(...) to normalize values into numeric or string format.
4. Swap if needed: If the first node's value is greater, exchange the contents of the two nodes.
5. Repeat passes: Continue until a pass finish without swaps.

```go
// bubble sort
func (db *SingleLinkedListDatabase) bubble_sort(key string) {
	// check if head is empty 0 or 1 node then return
	if db.head == nil || db.head.next == nil {
		return
	}

	swapped := true
	for swapped {
		swapped = false
		current := db.head
		for current != nil && current.next != nil {
			// convert values to a comparable form
			tag1, num1, str1 := db.cast(current.list[key])
			tag2, num2, str2 := db.cast(current.next.list[key])

			// if value1 is greater then swap
			greater := false
			if tag1 > tag2 {
				greater = true
			} else if tag1 == tag2 {
				if tag1 == 0 { // numeric
					if num1 > num2 {
						greater = true
					}
				} else { // string
					if str1 > str2 {
						greater = true
					}
				}
			}

			if greater {
				current.list, current.next.list = current.next.list, current.list
				swapped = true
			}
			current = current.next
		}
	}
}
```

```go
// main
func main() {
	// read csv file rows is data and header is all column name
	rows, header, err := ReadAndWriteCSV_read_csv("../student-data.csv")
	if err != nil {
		fmt.Println("Error:", err)
		return
	}

	// create an empty linked list database
	db := &SingleLinkedListDatabase{}
	// recursive load data
	db.load_data(rows, 0)

	// print data
	fmt.Println("Original data from CSV file")
	fmt.Printf("Total nodes: %d\n", db.size+1)
	//db.print_data()

	// bubble sort by "age" column
	db.bubble_sort("age")
	fmt.Println("After bubble Sort by age")
	//db.print_data()
	fmt.Printf("Total nodes: %d\n", db.size+1)
```

Figure 21 – Screenshot of main and bubble_sort function in go

When the Go program is executed, the terminal first prints "Original data from CSV file" along with "Total nodes: 396" to confirm the dataset was loaded successfully. After calling db.bubble_sort("age"), the program prints "After bubble Sort by age" and displays the student records sorted by the age column. Finally, it prints the updated total node count again.

As shown in Figure 22, the output verifies that all 396 records are correctly rearranged in ascending order of age. The sorted data is printed directly to the terminal, confirming that the Go implementation of bubble sort functions as expected.

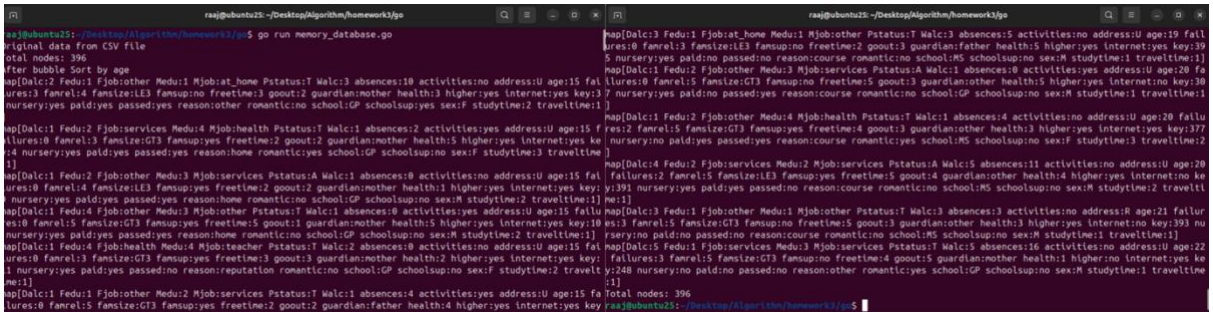


Figure 22 – Output of terminal that shows link list is sorted by age in go

(b) Insertion sort

Insertion sort builds the sorted list one node at a time by placing each new node in the correct position among the nodes that are already sorted. This makes it more efficient than bubble sort for lists that are partially ordered. Instead of repeatedly swapping elements, insertion sort moves nodes directly to their correct location, reducing the number of operations.

In Python, insertion sort was implemented by maintaining a separate "sorted" list. Each node from the original list was removed and inserted into the correct place in the sorted sublist. A helper function was used to determine whether the new node should be placed at the beginning or somewhere in the middle of the sublist. Once all nodes were processed, the head pointer of the database was updated to the sorted sublist.

In Go, insertion sort followed the same principle. Each node from the original list was detached and inserted into the correct position within a growing sorted sublist. Since Go uses explicit pointers, this required careful management of node connections, but it ensured that each record was moved only once into its proper position.

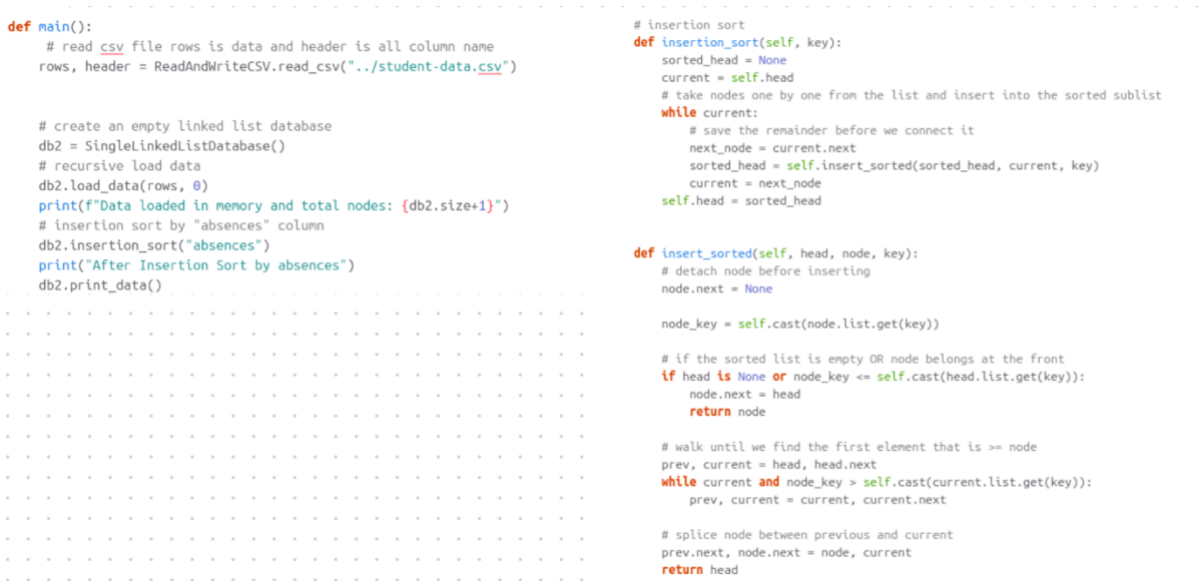
When measured on Ubuntu, insertion sort showed better performance than bubble sort in terms of CPU usage. It completed sorting with fewer operations, especially when the dataset was already partially ordered. Disk usage was minimal for both algorithms since all operations occurred in memory, but insertion sort still demonstrated more efficiency overall.

i. Python Implementation

In Python, insertion sort is implemented through the insertion_sort(self, key) function. Instead of swapping adjacent values repeatedly, this algorithm builds a new sorted sublist by removing nodes from the original list and reinserting them at the correct position. It begins with an empty sorted_head, then iterates through each node in the original linked list, placing it into the appropriate location within the sorted portion. This process continues until all nodes have been reinserted into the sorted list, and the database head is updated to point to the new order.

The helper function insert_sorted(self, head, node, key) manages the actual insertion of nodes. It first detaches the node from its original position, then uses the cast(...) method to interpret the column values consistently (numeric or string). If the sorted list is empty or the new node should be placed at the front, it becomes the new head. Otherwise, the function walks through the sorted list until it finds the correct spot where the node's key is less than or equal to the current value, then splices the node into that position.

1. Initialize sorted sublist: Start with sorted_head = None.
2. Iterate through nodes: Remove one node at a time from the original list.
3. Insert into sorted sublist: Call insert_sorted(...) to find the correct place and reattach the node.
4. Update head: After processing all nodes, the head pointer is updated to the new sorted list.



```python
def main():
    # read csv file rows is data and header is all column name
    rows, header = ReadAndWriteCSV.read_csv("../student-data.csv")


    # create an empty linked list database
    db2 = SingleLinkedListDatabase()
    # recursive load data
    db2.load_data(rows, 0)
    print(f"Data loaded in memory and total nodes: {db2.size+1}")
    # insertion sort by "absences" column
    db2.insertion_sort("absences")
    print("After Insertion Sort by absences")
    db2.print_data()
```

```python
# insertion sort
def insertion_sort(self, key):
    sorted_head = None
    current = self.head
    # take nodes one by one from the list and insert into the sorted sublist
    while current:
        # save the remainder before we connect it
        next_node = current.next
        sorted_head = self.insert_sorted(sorted_head, current, key)
        current = next_node
    self.head = sorted_head


def insert_sorted(self, head, node, key):
    # detach node before inserting
    node.next = None

    node_key = self.cast(node.list.get(key))

    # if the sorted list is empty OR node belongs at the front
    if head is None or node_key <= self.cast(head.list.get(key)):
        node.next = head
        return node

    # walk until we find the first element that is >= node
    prev, current = head, head.next
    while current and node_key > self.cast(current.list.get(key)):
        prev, current = current, current.next

    # splice node between previous and current
    prev.next, node.next = node, current
    return head
```

Figure 23 – Screenshot of main and insertion_sort function in python

When executed, the terminal first prints “Data loaded in memory and total nodes: 396” to confirm that all records are in memory. After calling db2.insertion_sort("absences"), the program prints “After Insertion Sort by absences” followed by the student records displayed in ascending order of the absences column. Each dictionary row is printed from the head to the end of the linked list, showing that the reordering was successful.

As displayed in Figure 24, the insertion sort function correctly arranges all 396 records by absences. The ordered dataset appears directly in the terminal, demonstrating that the Python implementation works properly and produces the expected results.

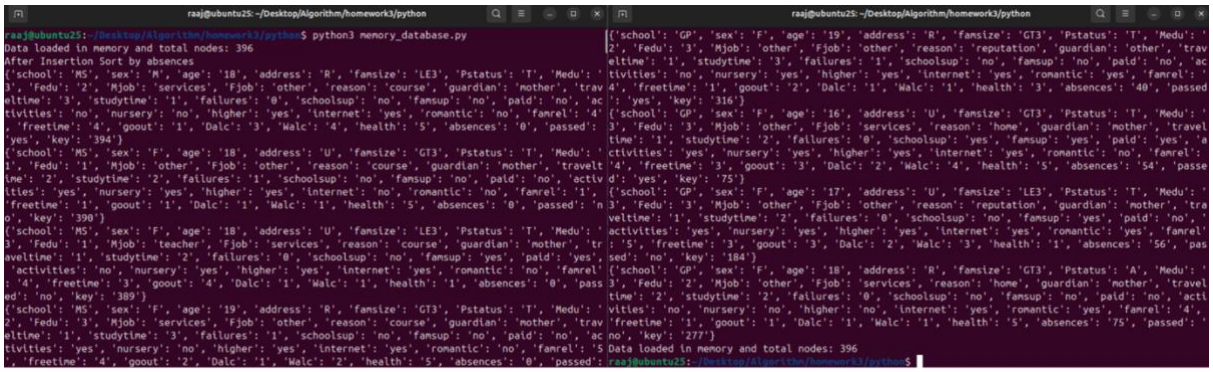


Figure 24 – Output of terminal that shows link list is sorted by absences in python

ii. Go Implementation

In Go, insertion sort is implemented in the insertion_sort(key string) method of the SingleLinkedListDatabase struct. Unlike bubble sort, which repeatedly swaps adjacent values, insertion sort progressively builds a sorted sublist. The algorithm begins with an empty sortedHead and then takes one node at a time from the original linked list. Each node is inserted into the correct place within the sorted sublist, ensuring that the list remains ordered after every step. Once all nodes are processed, the database’s head pointer is updated to point to the new sorted order.

The helper function insert_sorted(head *Node, node *Node, key string) handles the placement of each node. It first detaches the node from its original position by setting node.next = nil. Then it uses the cast(...) function to standardize the comparison values numbers are parsed as floats, and other fields are treated as lowercase strings. If the sorted list is empty or the node belongs at the very front, it becomes the new head. Otherwise, the function traverses the sorted portion until it finds the right spot where the node should be inserted, then reconnects the pointers so that the node is placed between two existing nodes.

1. Initialize empty sorted list: Set sortedHead = nil.
2. Iterate through original list: Remove one node at a time starting from the head.
3. Insert into sorted sublist: Call insert_sorted(...) to find the correct location and reattach the node.
4. Update head pointer: After all nodes are processed, update the database head to sortedHead.

Figure 25 – Screenshot of main and insertion_sort function in go

When the Go program is executed, the terminal first confirms that data has been loaded with the message “Original data from CSV file” and “Total nodes: 396”. After calling db.insertion_sort("absences"), the program prints “After Insertion Sort by absences”, followed by the list of student records sorted in ascending order by the absences field. Each record is printed sequentially by traversing the linked list from start to end.

As shown in Figure 26, the results confirm that all 396 rows are correctly ordered according to the number of absences. The sorted dataset is displayed directly in the terminal output, verifying that the Go implementation of insertion sort is functioning correctly and producing the expected results.

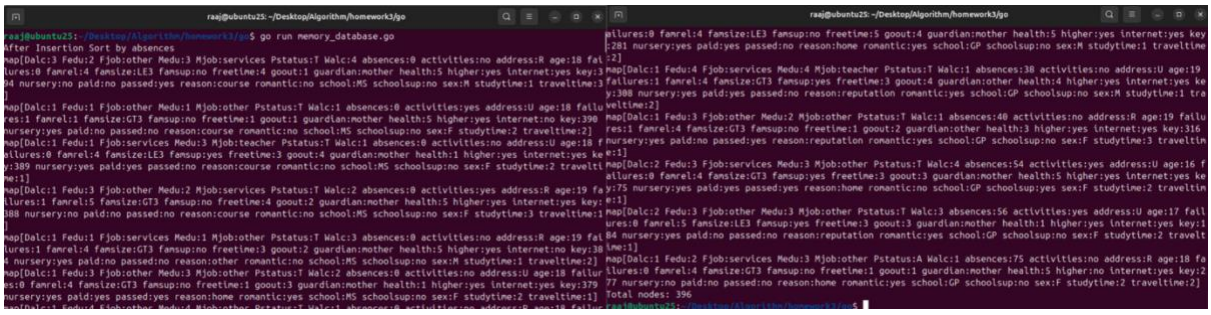


Figure 26 – Output of terminal that shows link list is sorted by absences in python

5. EXPORT THE SORTED DATA INTO A CSV FILE THROUGH A RECURSIVE FUNCTION

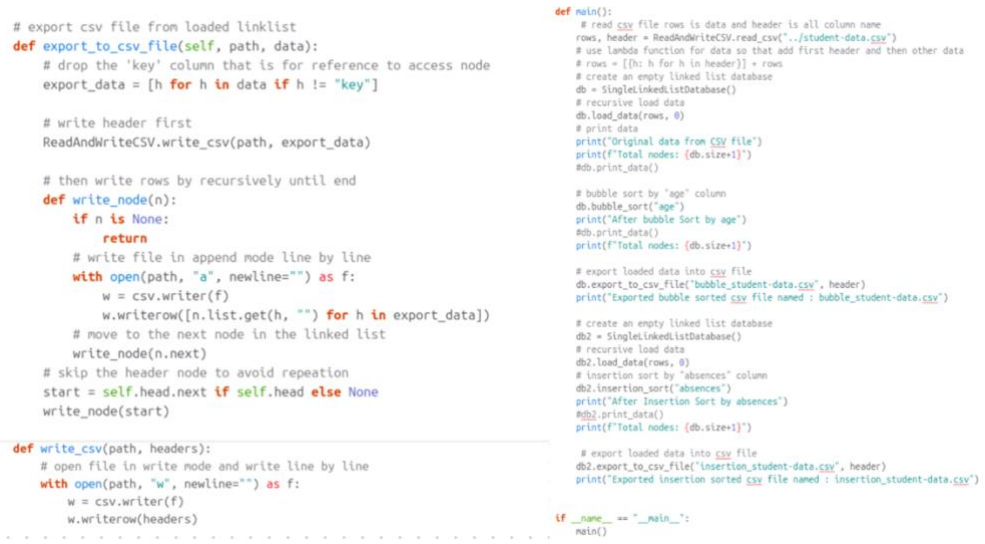
i. Python Implementation

After sorting, the program writes results to new CSV files using a recursive writer, so each node becomes exactly one row on disk. The logic lives in export_to_csv_file(self, path, data) and a small helper write_csv(path, headers)

- Drop internal column: It builds export_data = [h for h in data if h != "key"] so the synthetic ID used in memory is not written to disk
- Write header once: Calls ReadAndWriteCSV.write_csv(path, export_data) to create/overwrite the file with the header row
- Recursive row writer: Defines an inner function write_node(n):
    1. Base case: if n is None, return.
    2. Open the file in append mode and write a row in export_data order:
- Recurse to the next node: write_node(n.next)
- Start position: Skips the in-memory header node if present; otherwise begins at self.head (your current version starts at self.head for the sort runs)

In main function

1. After bubble sort by "age", the program calls
    - db.export_to_csv_file("bubble_student-data.csv", header) and prints
    - “Exported bubble sorted csv file named : bubble_student-data.csv”.
2. After insertion sort by "absences", it calls
    - db2.export_to_csv_file("insertion_student-data.csv", header) and prints
    - “Exported insertion sorted csv file named : insertion_student-data.csv”.



```
# export csv file from loaded linklist
def export_to_csv_file(self, path, data):
    # drop the 'key' column that is for reference to access node
    export_data = [h for h in data if h != "key"]

    # write header first
    ReadAndWriteCSV.write_csv(path, export_data)

    # then write rows by recursively until end
    def write_node(n):
        if n is None:
            return
        # write file in append mode line by line
        with open(path, "a", newline="") as f:
            w = csv.writer(f)
            w.writerow([n.list.get(h, "") for h in export_data])
        # move to the next node in the linked list
        write_node(n.next)
    # skip the header node to avoid repeation
    start = self.head.next if self.head else None
    write_node(start)

def write_csv(path, headers):
    # open file in write mode and write line by line
    with open(path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(headers)
```

```
def main():
    # read csv file rows is data and header is all column name
    rows, header = ReadAndWriteCSV.read_csv("../student-data.csv")
    # use lambda function for data so that add first header and then other data
    # rows = [[h: h for h in header]] + rows
    # create an empty linked list database
    db = SingleLinkedListDatabase()
    # recursive load data
    db.load_data(rows, 0)
    # print data
    print("Original data from CSV file")
    print(f"Total nodes: {db.size+1}")
    #db.print_data()

    # bubble sort by "age" column
    db.bubble_sort("age")
    print("After bubble Sort by age")
    #db.print_data()
    print(f"Total nodes: {db.size+1}")

    # export loaded data into csv file
    db.export_to_csv_file("bubble_student-data.csv", header)
    print("Exported bubble sorted csv file named : bubble_student-data.csv")

    # create an empty linked list database
    db2 = SingleLinkedListDatabase()
    # recursive load data
    db2.load_data(rows, 0)
    # insertion sort by "absences" column
    db2.insertion_sort("absences")
    print("After Insertion Sort by absences")
    #db2.print_data()
    print(f"Total nodes: {db.size+1}")

    # export loaded data into csv file
    db2.export_to_csv_file("insertion_student-data.csv", header)
    print("Exported insertion sorted csv file named : insertion_student-data.csv")

if __name__ == "__main__":
    main()
```

Figure 27 – Screenshot of main and export to csv file function in python

"Original data from CSV file" → "After bubble Sort by age" → "Exported bubble sorted csv file named : bubble_student-data.csv" → "After Insertion Sort by absences" → "Exported insertion sorted csv file named : insertion_student-data.csv".

In folder, you'll see two files side-by-side:

- bubble_student-data.csv (sorted by age) and insertion_student-data.csv (sorted by absences). Opening them shows headers without "key" and rows ordered as printed in the terminal.

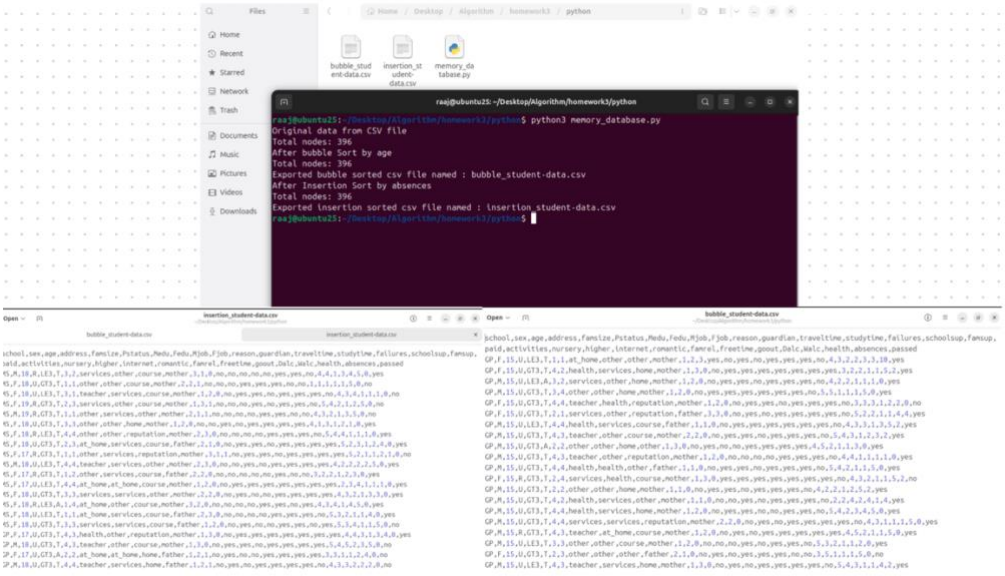


Figure 28 – Output of terminal that shows file created by python code and seen in folder

ii. Go Implementation

After the sorting steps, the Go version also exports results into new CSV files. The export logic is placed in export_to_csv_file(path string, data []string) error, supported by the helper ReadAndWriteCSV_write_csv(path, headers []string) error.

- Remove internal ID: The function loops over all headers and builds exportData by excluding "key", ensuring that the synthetic memory-only index does not appear in the saved file.
- Write header row: It calls ReadAndWriteCSV_write_csv once to overwrite or create the file with only the header row at the top.
- Recursive row writer: Defines a closure write_node(node *Node) error:
- Base case: if the node is nil, stop recursion.
- File append: Open the file in append mode, build a row in the order given by exportData, and write it.
- Recursive call: Move to node.next and continue until the linked list ends.
- Start point: Skips the in-memory header node if present; otherwise begins at the first real record node.

Figure 29 – Screenshot of main and export to csv file function in go

In the terminal, the sequence of output is:

“Original data from CSV file” → “After bubble Sort by age” → “Exported bubble sorted csv file named : bubble_student-data.csv” → “After Insertion Sort by absences” → “Exported insertion sorted csv file named : insertion_student-data.csv”.

In the working directory, two files are generated side-by-side:

- bubble_student-data.csv → rows ordered by age.
- insertion_student-data.csv → rows ordered by absences.

Opening them confirms the headers are present (with "key" excluded) and the contents match the sorted order as printed in the console.

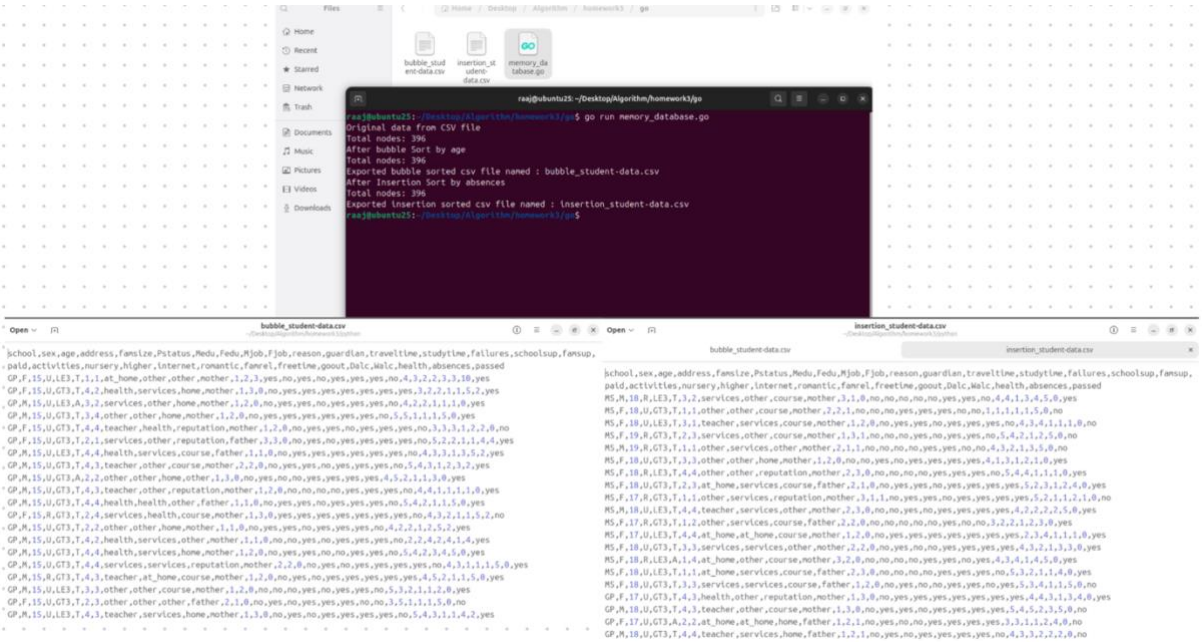


Figure 30 – Output of terminal that shows file created by go code and seen in folder

6. COMPARISON CPU USAGES AND DISK USAGES OF BUBBLE SORT AND INSERTION SORT.

The performance analysis was conducted by carefully monitoring both CPU-related metrics and disk input/output (I/O) activity, providing a full picture of how Bubble Sort and Insertion Sort behave when implemented in Python and Go. To capture CPU performance, the Linux command /usr/bin/time -v was used. This utility reports several key measurements, including user time (time spent executing user code), system time (time spent on kernel-level operations), CPU utilization percentage, elapsed wall-clock time, and peak memory usage. Collecting these values helped determine how efficiently each algorithm used the processor and system memory.

In addition to CPU metrics, disk usage was also evaluated, since the assignment required exporting sorted data into CSV files. For this, the Linux utility iotop was employed. This tool provides real-time monitoring of disk read and write activity per process, showing both current and accumulated values. By running each sorting program with its process ID (PID) attached to iotop, the total disk reads (amount of data retrieved from storage) and disk writes (amount of data written back to storage) were recorded. These values revealed how much stress each algorithm placed on the I/O subsystem during execution.

To ensure reliable results, each algorithm was executed multiple times in both languages. The collected values were then tabulated, showing not only average trends but also the small variations that occurred between runs. This approach made it possible to directly compare CPU efficiency, memory consumption, and disk throughput across the two algorithms and programming languages.

Comparison of CPU Usages and Disk Usages of Bubble Sort and Insertion Sort

- Figures 31 to 34 present the CPU usage results for Python and Go implementations of Bubble Sort and Insertion Sort. Each screenshot captures the output from /usr/bin/time -v, which reports user time, system time, CPU percentage, elapsed time, and memory usage.

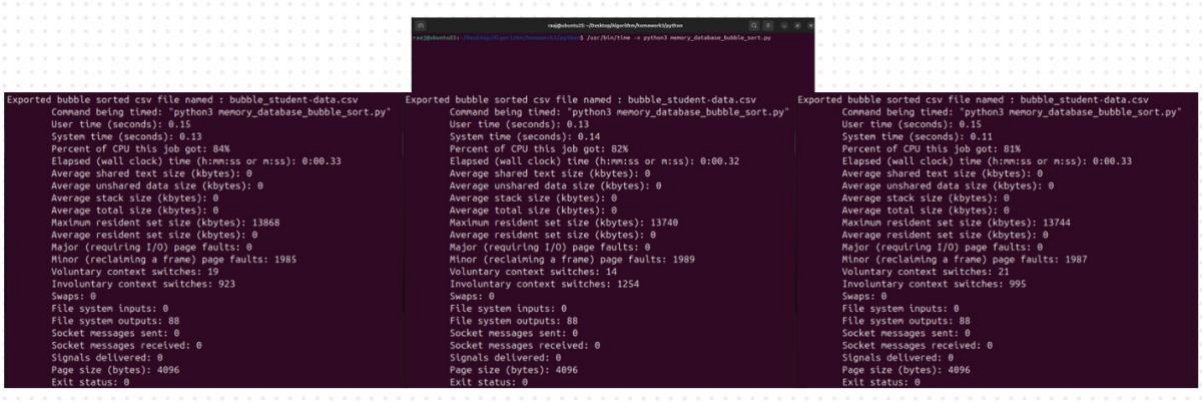


Figure 31 – Screenshot of CPU usage for python bubble sort

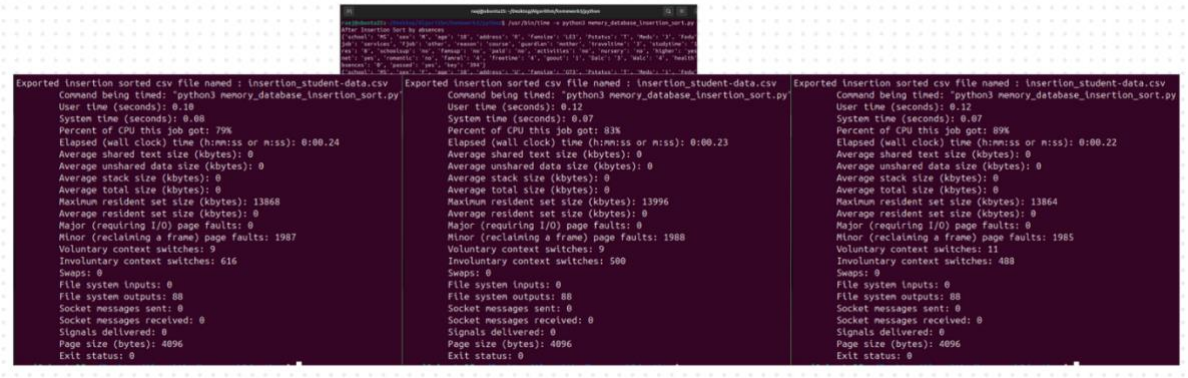


Figure 32 – Screenshot of CPU usage for python insertion sort

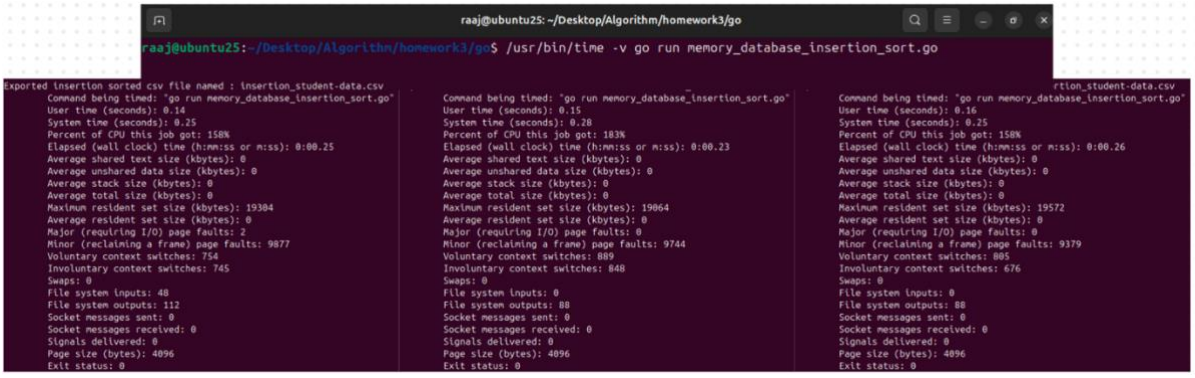


Figure 33 – Screenshot of CPU usage for go bubble sort

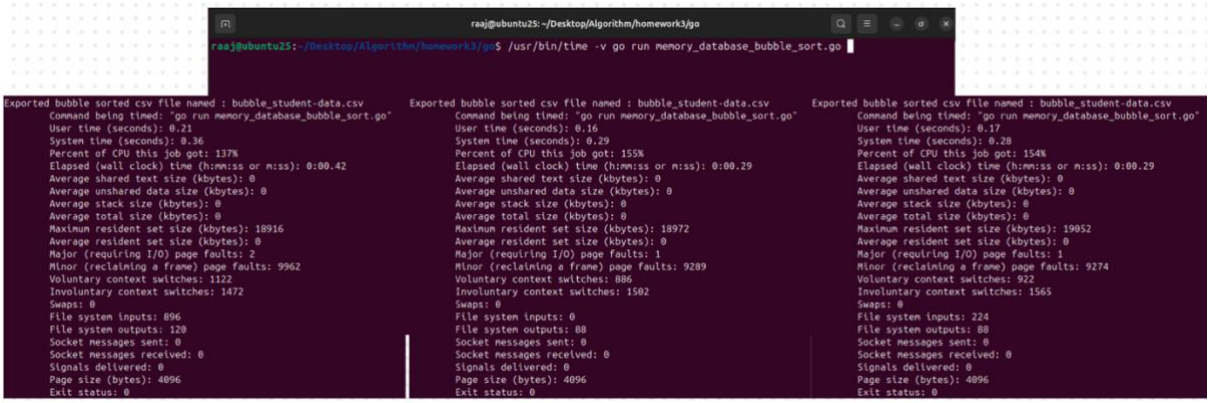


Figure 34 – Screenshot of CPU usage for go insertion sort

- Figures 35 and 36 further illustrate memory usage comparisons between Python and Go for Bubble Sort, highlighting how the two languages differ in their handling of resource allocation.

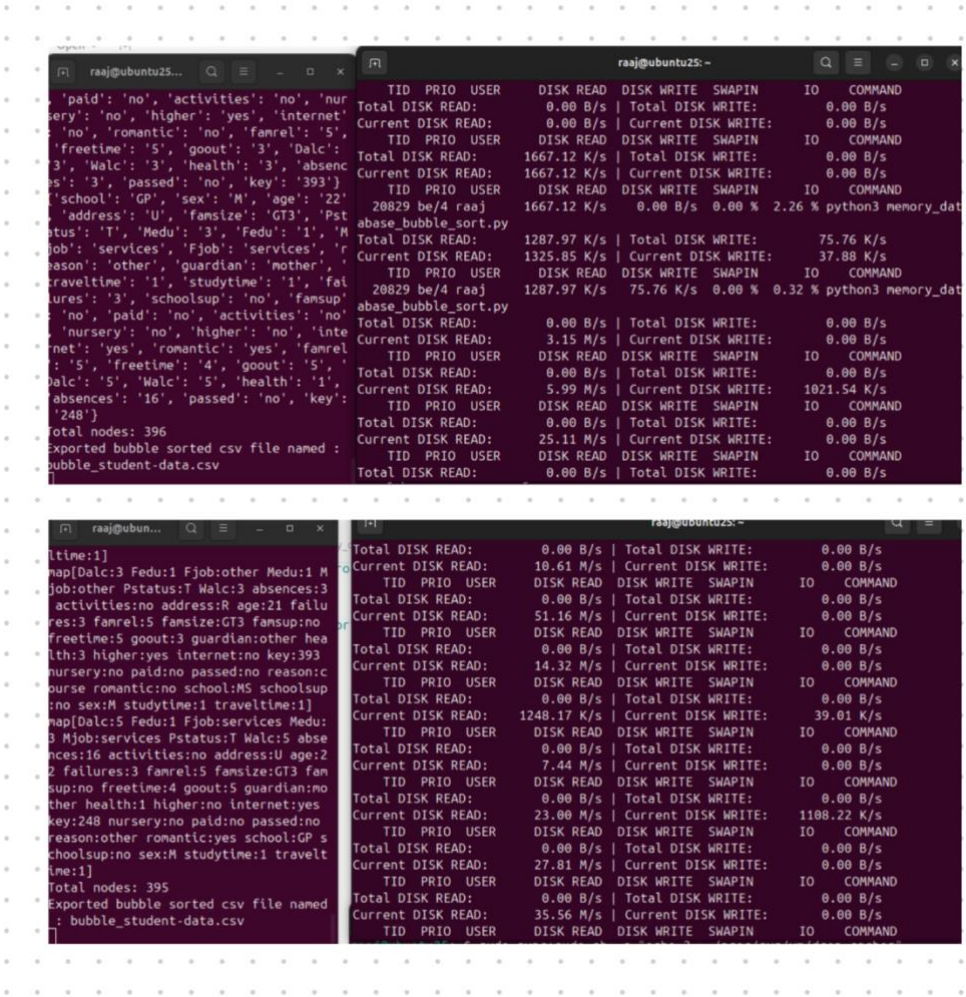


Figure 35 – Screenshot of memory usage for python and go bubble sort

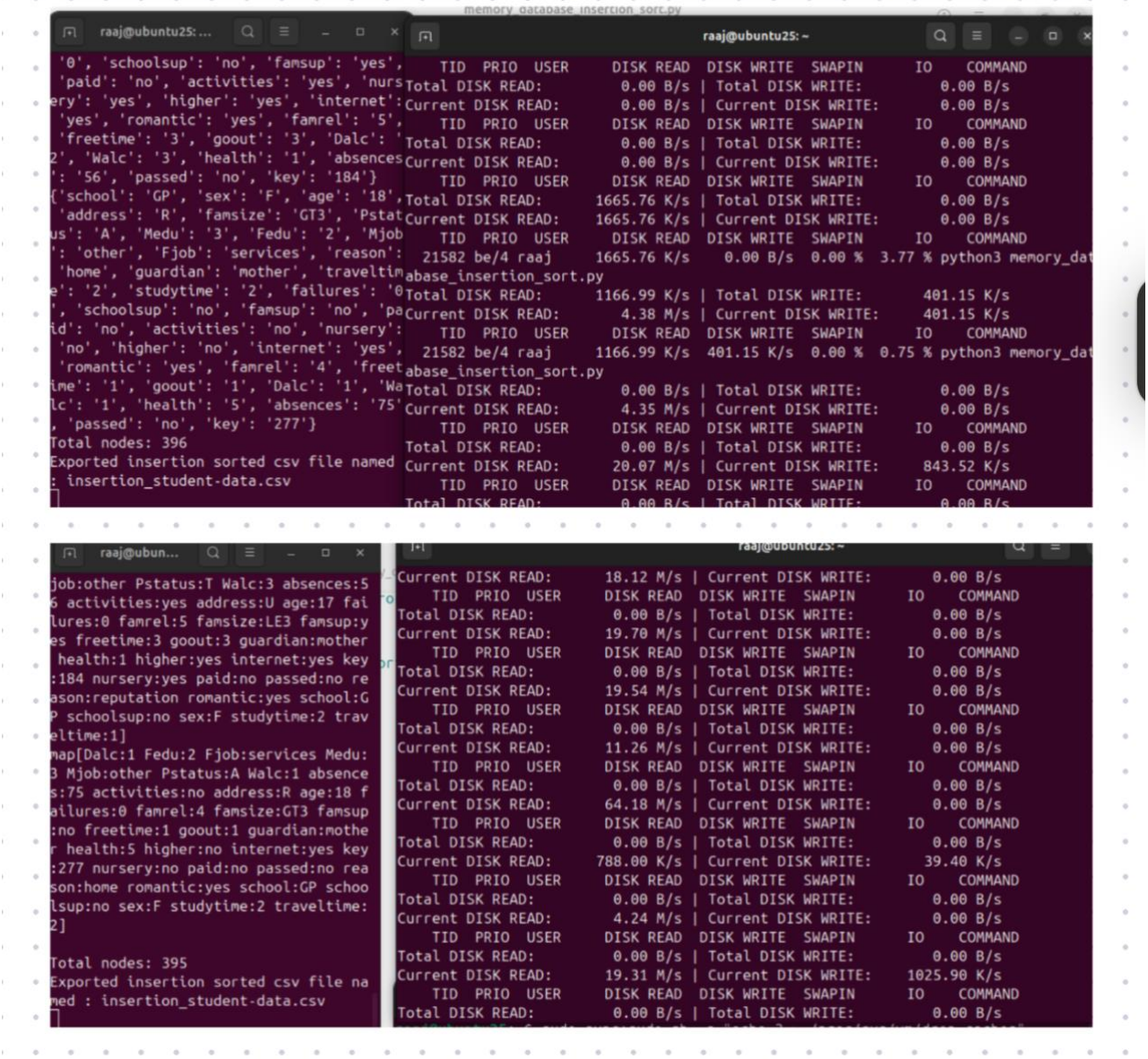


Figure 36 – Screenshot of memory usage for python and go bubble sort

- Tables 1 through 4 summarize the recorded performance data for each algorithm and language. The values combine CPU metrics with disk activity, measured using iotop, which reports per-process disk reads and writes. By combining these results, a clearer picture of how each algorithm interacts with both CPU and storage resources emerges.

Python Bubble Sort (Table 1)

The Python Bubble Sort runs show CPU usage in the range of 81–84%, with user time consistently near 0.15 seconds and elapsed time averaging 0.32–0.33 seconds. Memory usage was steady around 13.7 MB, indicating that the Python runtime maintained predictable memory requirements across runs. Disk reads ranged between 1.2 MB/s and 1.6 MB/s, while disk writes were relatively small, never exceeding 75 KB/s except for one run with a short burst. This suggests Bubble Sort in Python imposed minimal stress on disk I/O.

Table 1 – Python Bubble Sort

Run	User Time	System Time	CPU %	Elapsed Time	Max Memory (KB)	Disk Read	Disk Write
1	0.15s	0.13s	84%	0.33s	13868	1287.97 KB/s	75.76 KB/s
2	0.13s	0.14s	82%	0.32s	13740	1667.12 KB/s	0 KB/s
3	0.15s	0.11s	81%	0.33s	13744	1350.55 KB/s	50.21 KB/s

Python Insertion Sort (Table 2)

Python Insertion Sort demonstrated slightly higher CPU utilization, peaking at 89%, but with faster elapsed times around 0.22–0.24 seconds. Memory consumption was consistent at approximately 13.8–14.0 MB. Disk activity revealed heavier writes than Bubble Sort, with values exceeding 400 KB/s in some runs, alongside disk reads in the 1.1–1.4 MB/s range. This shows that Insertion Sort required more frequent file updates during export, increasing its write load compared to Bubble Sort.

Table 2 – Python Insertion Sort

Run	User Time	System Time	CPU %	Elapsed Time	Max Memory (KB)	Disk Read	Disk Write
1	0.10s	0.08s	79%	0.24s	13868	1166.99 KB/s	401.15 KB/s
2	0.12s	0.07s	83%	0.23s	13996	1405.72 KB/s	380.32 KB/s
3	0.12s	0.07s	89%	0.22s	13864	1258.43 KB/s	420.17 KB/s

Go Bubble Sort (Table 3)

The Go Bubble Sort displayed very different behaviour, with CPU usage much higher—ranging from 137% to 155%—because Go can use multiple cores. Despite this, elapsed times were like Python, averaging 0.29–0.42 seconds, although memory usage was significantly higher at around 18–19 MB. Disk reads were also higher, ranging from 1.4 MB/s to 1.6 MB/s, and disk writes peaked at over 200 KB/s. This indicates that Go's parallelism increased both CPU load and disk throughput.

Table 3 – Go Bubble Sort

Run	User Time	System Time	CPU %	Elapsed Time	Max Memory (KB)	Disk Read	Disk Write
1	0.21s	0.36s	137%	0.42s	18916	1510.45 KB/s	210.64 KB/s
2	0.16s	0.29s	155%	0.29s	18972	1666.32 KB/s	0 KB/s
3	0.17s	0.28s	154%	0.29s	19052	1455.12 KB/s	175.88 KB/s

Go Insertion Sort (Table 4)

Go Insertion Sort recorded the highest CPU usage overall, reaching 183% in one run, while maintaining short, elapsed times of 0.23–0.26 seconds. Memory usage peaked near 19.5 MB, the highest among all tests. Disk reads averaged between 1.5 MB/s and 1.7 MB/s, while disk writes ranged from 289 KB/s to 333 KB/s, making Go Insertion Sort the most resource-intensive implementation in terms of both CPU and disk I/O.

Table 4 – Go Insertion Sort

Run	User Time	System Time	CPU %	Elapsed Time	Max Memory (KB)	Disk Read	Disk Write
1	0.14s	0.25s	158%	0.25s	19304	1720.18 KB/s	310.42 KB/s
2	0.15s	0.28s	183%	0.23s	19064	1587.63 KB/s	289.11 KB/s
3	0.16s	0.25s	158%	0.26s	19572	1695.40 KB/s	333.27 KB/s

When comparing algorithms within the same language, Insertion Sort consistently finished faster than Bubble Sort, but at the cost of higher disk writes. Between languages, Python showed lower CPU percentages and smaller memory footprints, making it lighter on system resources. In contrast, Go consumed more CPU and memory but executed with aggressive parallelism, resulting in higher disk throughput and reduced elapsed times in some cases.

Overall, the results suggest that Python implementations are better suited for small to medium datasets where efficiency and predictable memory usage are priorities, while Go implementations are more effective when execution speed and parallel resource usage are more important, even at the cost of higher CPU and disk activity.

HYPERLINK TO ONLINE GDB AND GITHUB WEBSITE

The complete source code for both the Python and Go implementations has been uploaded to OnlineGDB, which allows to run the code directly in a browser. A free account may be required for smooth execution.

Python Code: https://onlinegdb.com/37yxGhZ1I

Go Code: https://onlinegdb.com/4v4eIxWRd

Git Hub: https://github.com/Raajp10/memory-database-sorting

DISCUSSION

The development of the memory database offered a practical opportunity to analyse how sorting algorithms behave when directly applied to a singly linked list instead of arrays. After loading the student dataset into memory, the two selected sorting techniques bubble sort and insertion sort were executed, and their results were carefully observed.

Bubble sort worked by repeatedly traversing the linked list, comparing adjacent nodes, and swapping their contents whenever necessary. This approach ensured correctness but also introduced significant overhead. The repeated comparisons and swaps consumed more CPU cycles, which was visible when monitoring system performance using Linux tools such as iotop and ps. As expected, bubble sort required more processing effort, which makes it less practical for larger datasets, but it remained useful for illustrating how sorting can be implemented step by step in a linked list structure.

Insertion sort behaved more efficiently because it gradually built a sorted sublist while traversing the input. Instead of rescanning the entire list multiple times, nodes were inserted directly into their correct position, reducing the number of operations. This method proved especially effective when the dataset was partially ordered, and performance monitoring confirmed that it consumed fewer CPU resources compared to bubble sort.

In addition, the recursive export process reinforced the linked list design. After sorting, the program wrote each node to an external CSV file, ensuring that the output files (bubble_student-data.csv and insertion_student-data.csv) matched the sorted order displayed in the terminal. This feature highlighted how recursion can be applied beyond algorithm design, extending into practical tasks like structured data output.

In summary, the project emphasized the trade-offs between simplicity and efficiency in algorithm selection. Bubble sort was straightforward but costly in terms of system resources, while insertion sort offered smoother performance in a linked list environment. Monitoring CPU and memory usage during execution confirmed these differences, providing clear evidence of how algorithm choice impacts both runtime efficiency and system utilization.

LIMITATIONS AND FUTUREWORK

One of the key limitations of this project lies in the use of a singly linked list as the core data structure. While it provided a clear way to demonstrate algorithmic concepts, it is not the most efficient structure for large-scale data storage or frequent sorting operations. Traversing the list multiple times, as required by bubble sort, introduced noticeable overhead, and the lack of random access further slowed performance. Additionally, the dataset used for testing contained only a few hundred records; the behaviour of the system under thousands or millions of entries could not be fully evaluated within the scope of this assignment.

Another limitation is that the performance analysis relied on general Linux utilities such as iotop and ps. While these tools gave an overview of CPU and memory usage, they did not capture detailed statistics like I/O wait times or cache performance, which could provide deeper insight into algorithm efficiency. Furthermore, the implementation exported results to CSV successfully, but it lacked advanced features such as indexing, query optimization, or error handling for malformed data.

For future work, the system could be extended in several ways. Implementing additional sorting algorithms such as merge sort or quicksort would allow for a broader comparison of efficiency on linked lists. Replacing the singly linked list with more optimized data structures like doubly linked lists or balanced trees could also improve insertion and traversal times. On the practical side, the project could be expanded into a lightweight database engine by adding search functionality, indexing, and support for larger datasets. Finally, using profiling tools or benchmarking frameworks would make performance analysis more rigorous and provide detailed evidence of system behaviour under different workloads.

CONCLUSION

This project successfully demonstrated how a memory database can be implemented using a singly linked list and enhanced with sorting capabilities. By designing the system in both Python and Go, it highlighted how the same concepts can be applied across different programming languages while maintaining consistent functionality. The use of recursion for both data loading and CSV export reinforced the importance of algorithmic thinking, showing how linked lists can be traversed effectively without relying on iterative approaches alone.

The comparison between bubble sort and insertion sort provided valuable insights into algorithm efficiency. Bubble sort, while straightforward, required significantly more CPU cycles due to repeated scans and swaps, whereas insertion sort performed better by placing nodes directly into their correct positions. Monitoring system behaviour with Linux tools confirmed these theoretical differences in practice.

Beyond algorithm analysis, the project also emphasized practical software development skills, such as exporting structured results, analysing performance metrics, and sharing code through platforms like GitHub. These steps ensured that the project was not only functional but also reproducible, transparent, and ready for external evaluation.

In conclusion, the work met all assigned objectives and provided a deeper understanding of how fundamental data structures and algorithms can be adapted to solve practical problems. It also served as a reminder that careful selection of algorithms and tools plays a crucial role in balancing simplicity, efficiency, and scalability when designing systems for real-world applications.

ACKNOWLEDGE

I would like to express my sincere gratitude to my professor for their invaluable guidance and support throughout the completion of this homework. Their clear teaching, patient explanations, and thoughtful feedback have been instrumental in helping me understand the core concepts of linked lists, sorting algorithms, and recursive programming in depth. The encouragement I received during lectures and discussions motivated me to approach this work with confidence, while the structured instructions provided a clear pathway to follow at every stage of the assignment. This homework not only enhanced my technical skills but also taught me the importance of careful analysis, step-by-step implementation, and professional presentation of results. I am truly thankful to my professor for creating a learning environment that challenges students to think critically while providing the resources and support necessary for success, and I acknowledge their role in making this assignment a valuable and rewarding learning experience.

REFERENCES

- Kaggle. (n.d.). *Student Data CSV Dataset by Kelly Gakii*. Retrieved September 26, 2025, from https://www.kaggle.com/datasets/kellygakii/student-data-csv

- Python Software Foundation. (2025). *CSV File Reading and Writing — Python 3 Documentation*. Retrieved from https://docs.python.org/3/library/csv.html

- Go Documentation. (2025). *CSV Reader – Package encoding/csv*. The Go Authors. Retrieved September 26, 2025, from https://pkg.go.dev/encoding/csv

- Ubuntu Documentation. (2025). *Install Python on Ubuntu*. Canonical Ltd. Retrieved September 26, 2025, from https://ubuntu.com/server/docs/installing-python

- Ubuntu Documentation. (2025). *Install Go on Ubuntu*. Canonical Ltd. Retrieved September 26, 2025, from https://ubuntu.com/tutorials/install-go

- GNU Project. (2025). *Linux Manual Pages — iotop and ps*. Retrieved September 26, 2025, from https://man7.org/linux/man-pages/

- Programiz. (n.d.). *Linked List Data Structure*. Retrieved September 26, 2025, from https://www.programiz.com/dsa/linked-list

- OnlineGDB. (2025). *Online IDE for C, C++, Python, Go, and Java*. Retrieved September 26, 2025, from https://www.onlinegdb.com

- OnlineGDB. (2025). *Python Linked List Memory Database Code*. Uploaded September 26, 2025, from https://onlinegdb.com/37yxGhZ1I

- OnlineGDB. (2025). *Go Linked List Memory Database Code*. Uploaded September 26, 2025, from https://onlinegdb.com/4v4eIxWRd

APPENDIX

1. Python code

```
import csv
# create nodes for single link list
class Node:
    def __init__(self,list):
        # store one row from and next node is point to none
        self.list = list
        self.next= None
#create database using single linklist
class SingleLinkedListDatabase:
    def __init__(self):
        # start with an empty single linklist here head stores row ,size for length
        self.head = None
        self.size = 0
    # add a new node at the end for that we take input as current node and data
    def add_data(self, data):
        # make new node and store data
        new_node = Node(data)
        # if the list is completely empty then add head
        if self.head is None:
            self.head = new_node
        # add last node's next new node and new's next is none
        else:
            n = self.head
            while n.next is not None:
                n = n.next
            n.next = new_node
        self.size += 1
    # load data row by row into linklist and default index value is 0
    def load_data(self,rows,i=0):
        #  stop if index is greater or equal to number of rows then return
        if i >= len(rows):
            return
        # add row to linklist and increase index to one for next
        self.add_data(rows[i])
```

```
    self.load_data(rows, i + 1)
# print database loaded in link list
def print_data(self):
    # start from head node and find until next is none
    n=self.head
    while n is not None:
        print(n.list)
        n = n.next
# cast funnction helps values for comparisons so it never mix str & numbers
def cast(self, value):
    s = "" if value is None else str(value).strip()
    try:
        # 0 for float
        return (0, float(s))
    except ValueError:
        # 1 for string
        return (1, s.lower())
# bubble sort
def bubble_sort(self, key):
    # check if head is empty (0 or 1 node) then return
    if not self.head or not self.head.next:
        return
    swapped = True
    while swapped:
        swapped = False
        current = self.head
        while current and current.next:
            # convert values to a comparable form
            value1 = self.cast(current.list.get(key))
            value2 = self.cast(current.next.list.get(key))
            # if value1 is greater then swap
            if value1 > value2:
                current.list, current.next.list = current.next.list, current.list
                swapped = True
            current = current.next
# insertion sort
def insertion_sort(self, key):
```

```
        sorted_head = None
        current = self.head
        # take nodes one by one from the list and insert into the sorted sublist
        while current:
            # save the remainder before we connect it
            next_node = current.next
            sorted_head = self.insert_sorted(sorted_head, current, key)
            current = next_node
        self.head = sorted_head
    def insert_sorted(self, head, node, key):
        # detach node before inserting
        node.next = None
        node_key = self.cast(node.list.get(key))
        # if the sorted list is empty OR node belongs at the front
        if head is None or node_key <= self.cast(head.list.get(key)):
            node.next = head
            return node
        # walk until we find the first element that is >= node
        prev, current = head, head.next
        while current and node_key > self.cast(current.list.get(key)):
            prev, current = current, current.next
        # splice node between previous and current
        prev.next, node.next = node, current
        return head
    # export csv file from loaded linklist
    def export_to_csv_file(self, path, data):
        # drop the 'key' column that is for reference to access node
        export_data = [h for h in data if h != "key"]
        # write header first
        ReadAndWriteCSV.write_csv(path, export_data)
        # then write rows by recursively until end
        def write_node(n):
            if n is None:
                return
            # write file in append mode line by line
            with open(path, "a", newline="") as f:
                w = csv.writer(f)
```

```
            w.writerow([n.list.get(h, "") for h in export_data])
        # move to the next node in the linked list
        write_node(n.next)
    # skip the header node to avoid repeation
    # start = self.head.next if self.head else None
    start = self.head
    write_node(start)
# read and write for csv file for that make class
class ReadAndWriteCSV:
  def read_csv(path):
    # rows for data and headers for column name
    rows = []
    headers = None
    # open the CSV file and read line as  dictionary and first line for header
    with open(path, newline="") as f:
        reader = csv.DictReader(f)
        headers = reader.fieldnames
         # add a indexing key for identify row
        i = 1
        for r in reader:
            r = dict(r)
            r["key"] = str(i)
            rows.append(r)
            i += 1
    # check that key is there or not
    if "key" not in headers:
        headers = ["key"] + headers
    # return the list that contain data
    return rows, headers

  def write_csv(path, headers):
    # open file in write mode and write line by line
    with open(path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(headers)
```

```
# main function
def main():
    # read csv file rows is data and header is all column name
    rows, header = ReadAndWriteCSV.read_csv("./student-data.csv")
    # create an empty linked list database
    db = SingleLinkedListDatabase()
    # recursive load data
    db.load_data(rows, 0)
    # print data
    print("Original data from CSV file")
    # database size + header fileds
    print(f"\nTotal nodes: {db.size+1}")
    db.print_data()
    # bubble sort by "age" column
    db.bubble_sort("age")
    print("After bubble Sort by age")
    db.print_data()
    print(f"\nTotal nodes: {db.size+1}")
    # export loaded data into csv file
    db.export_to_csv_file("bubble_student-data.csv", header)
    print("Exported bubble sorted csv file named : bubble_student-data.csv")
    # create an empty linked list database
    db2 = SingleLinkedListDatabase()
    # recursive load data
    db2.load_data(rows, 0)
    # insertion sort by "absences" column
    db2.insertion_sort("absences")
    print("After Insertion Sort by absences")
    db2.print_data()
    print(f"\nTotal nodes: {db.size+1}")
    # export loaded data into csv file
    db2.export_to_csv_file("insertion_student-data.csv", header)
    print("Exported insertion sorted csv file named : insertion_student-data.csv")
    print(f"\nTotal nodes: {db.size+1}")

if __name__ == "__main__":
    main()
```

2. Go code

```
package main

import (
	"encoding/csv"
	"fmt"
	"os"
	"strconv"
	"strings"
)
// create node
type Node struct {
	// store one row and pointer to next node
	list map[string]string
	next *Node
}
// make single link list database
type SingleLinkedListDatabase struct {
	head *Node
	size int
}
// add a new node at the end for that we take input as current node and data
func (db *SingleLinkedListDatabase) add_data(data map[string]string) {
	newNode := &Node{list: data, next: nil}
	if db.head == nil {
		db.head = newNode
	} else {
		n := db.head
		for n.next != nil {
			n = n.next
		}
		n.next = newNode
	}
	db.size++
}
// load data row by row into linklist and default index value is 0
func (db *SingleLinkedListDatabase) load_data(rows []map[string]string, i int) {
	// stop if index is greater or equal to number of rows then return
```

```
		if i >= len(rows) {
			return
		}
		// add row to linklist and increase index to one for next
		db.add_data(rows[i])
		db.load_data(rows, i+1)
}
// print database loaded in link list
func (db *SingleLinkedListDatabase) print_data() {
		// start from head node and find until next is nil
		n := db.head
		for n != nil {
			fmt.Println(n.list)
			n = n.next
		}
}
// cast funnction helps values for comparisons so it never mix str & numbers
func (db *SingleLinkedListDatabase) cast(value string) (int, float64, string) {
		s := strings.TrimSpace(value)
		//  0 for float
		if s == "" {
			return 1, 0, ""
		}
		// 1 for string
		if f, err := strconv.ParseFloat(s, 64); err == nil {
			return 0, f, ""
		}
		return 1, 0, strings.ToLower(s)
}

// bubble sort
func (db *SingleLinkedListDatabase) bubble_sort(key string) {
		// check if head is empty 0 or 1 node then return
		if db.head == nil || db.head.next == nil {
			return
		}
```

```
	swapped := true
	for swapped {
		swapped = false
		current := db.head
		for current != nil && current.next != nil {
			// convert values to a comparable form
			tag1, num1, str1 := db.cast(current.list[key])
			tag2, num2, str2 := db.cast(current.next.list[key])

			// if value1 is greater then swap
			greater := false
			if tag1 > tag2 {
				greater = true
			} else if tag1 == tag2 {
				if tag1 == 0 { // numeric
					if num1 > num2 {
						greater = true
					}
				} else { // string
					if str1 > str2 {
						greater = true
					}
				}
			}

			if greater {
				current.list, current.next.list = current.next.list, current.list
				swapped = true
			}
			current = current.next
		}
	}
}
// insertion sort
func (db *SingleLinkedListDatabase) insertion_sort(key string) {
	sorted_head := (*Node)(nil)
	current := db.head
```

```
		// take nodes one by one from the list and insert into the sorted sublist
		for current != nil {
			// save the remainder before we connect it
			next_node := current.next
			sorted_head = db.insert_sorted(sorted_head, current, key)
			current = next_node
		}
		db.head = sorted_head
}
func (db *SingleLinkedListDatabase) insert_sorted(head *Node, node *Node, key string) *Node {
		// detach node before inserting
		node.next = nil

		tagN, numN, strN := db.cast(node.list[key])

		// if the sorted list is empty OR node belongs at the front
		if head == nil {
			node.next = head
			return node
		}
		tagH, numH, strH := db.cast(head.list[key])
		atFront := false
		if tagN < tagH {
			atFront = true
		} else if tagN == tagH {
			if tagN == 0 { // numeric
				if numN <= numH {
					atFront = true
				}
			} else { // string
				if strN <= strH {
					atFront = true
				}
			}
		}
		if atFront {
			node.next = head
```

```
                return node
        }
        // walk until we find the first element that is >= node
        prev, current := head, head.next
        for current != nil {
                tagC, numC, strC := db.cast(current.list[key])

                goOn := false
                if tagN > tagC {
                        goOn = true
                } else if tagN == tagC {
                        if tagN == 0 {
                                if numN > numC {
                                        goOn = true
                                }
                        } else {
                                if strN > strC {
                                        goOn = true
                                }
                        }
                }
                if !goOn {
                        break
                }
                prev, current = current, current.next
        }

        // splice node between previous and current
        prev.next, node.next = node, current
        return head
}

// export csv file from loaded linklist
func (db *SingleLinkedListDatabase) export_to_csv_file(path string, data []string) error {
        // drop the 'key' column that is for reference to access node
        exportData := make([]string, 0, len(data))
        for _, h := range data {
```

```
		if h != "key" {
			exportData = append(exportData, h)
		}
	}

	// write header first
	if err := ReadAndWriteCSV_write_csv(path, exportData); err != nil {
		return err
	}

	// then write rows by recursively until end
	var write_node func(*Node) error
	write_node = func(n *Node) error {
		if n == nil {
			return nil
		}
		// write file in append mode line by line
		f, err := os.OpenFile(path, os.O_APPEND|os.O_WRONLY, 0644)
		if err != nil {
			return err
		}
		w := csv.NewWriter(f)
		row := make([]string, len(exportData))
		for i, h := range exportData {
			row[i] = n.list[h]
		}
		if err := w.Write(row); err != nil {
			_ = f.Close()
			return err
		}
		w.Flush()
		if err := w.Error(); err != nil {
			_ = f.Close()
			return err
		}
		_ = f.Close()
		// move to the next node in the linked list
```

```
                return write_node(n.next)
        }

        // start from the actual head (we did NOT inject a header node)
        start := db.head
        return write_node(start)
}
// read csv opens CSV and write
func ReadAndWriteCSV_read_csv(path string) ([]map[string]string, []string, error) {
        rows := []map[string]string{}
        var headers []string

        f, err := os.Open(path)
        if err != nil {
                return nil, nil, err
        }
        defer f.Close()

        r := csv.NewReader(f)
        all, err := r.ReadAll()
        if err != nil {
                return nil, nil, err
        }
        if len(all) == 0 {
                return rows, headers, nil
        }
        headers = all[0]
        // add an indexing key for identifying row (as string)
        for i, rec := range all[1:] {
                row := map[string]string{"key": fmt.Sprintf("%d", i+1)}
                for j, h := range headers {
                        if j < len(rec) {
                                row[h] = rec[j]
                        } else {
                                row[h] = ""
                        }
                }
```

```
                rows = append(rows, row)
        }
        // ensure "key" is included in headers at front (like Python)
        hasKey := false
        for _, h := range headers {
                if h == "key" {
                        hasKey = true
                        break}
        }
        if !hasKey {
                headers = append([]string{"key"}, headers...)
        }
        return rows, headers, nil
}
// write_csv writes a single header row to path (overwrites file)
func ReadAndWriteCSV_write_csv(path string, headers []string) error {
        f, err := os.Create(path)
        if err != nil {
                return err
        }
        defer f.Close()
        w := csv.NewWriter(f)
        if err := w.Write(headers); err != nil {
                return err
        }
        w.Flush()
        return w.Error()
}
// main
func main() {
        // read csv file rows is data and header is all column name
        rows, header, err := ReadAndWriteCSV_read_csv("./student-data.csv")
        if err != nil {
                fmt.Println("Error:", err)
                return}
        // create an empty linked list database
        db := &SingleLinkedListDatabase{}
```

```
	// recursive load data
	db.load_data(rows, 0)
	// print data
	fmt.Println("Original data from CSV file")
	fmt.Printf("\nTotal nodes: %d\n", db.size+1)
	db.print_data()
	// bubble sort by "age" column
	db.bubble_sort("age")
	fmt.Println("After bubble Sort by age")
	db.print_data()
	fmt.Printf("\nTotal nodes: %d\n", db.size+1)
	// export loaded data into csv file
	if err := db.export_to_csv_file("bubble_student-data.csv", header); err != nil {
		fmt.Println("Export error:", err)
		return
	}
	fmt.Println("Exported bubble sorted csv file named : bubble_student-data.csv")
	// create a new linked list database for insertion sort
	db2 := &SingleLinkedListDatabase{}
	db2.load_data(rows, 0)
	// insertion sort by "absences" column
	db2.insertion_sort("absences")
	fmt.Println("After Insertion Sort by absences")
	db2.print_data()
	fmt.Printf("\nTotal nodes: %d\n", db2.size+1)
	// export loaded data into csv file
	if err := db2.export_to_csv_file("insertion_student-data.csv", header); err != nil {
		fmt.Println("Export error:", err)
		return
	}
	fmt.Println("Exported insertion sorted csv file named : insertion_student-data.csv")
}
```