Technological University Dublin

BSc. (Honours) Degree in Computer Science
Year 1

Assignment 2022-2023

Algorithm Design and Problem Solving [CMPU1001]

Lecturer Ciaran Kelly

Name: Aaron Baggot **Student Number: C22716399**

Table of Contents

Introduction

The purpose of this project is to provide an overview of the design and implementation of a code for a School of Computer Science. The code maintains data of students who are enrolled in various postgraduate programmes.

This report will provide an overview of the design of the code through the use of flowcharts, algorithms, and test cases. The assignment will also discuss the importance of maintaining data for students who are enrolled in postgraduate programmes and the challenges that come with managing the data. It will take you through the steps taken to ensure that the software was tested thoroughly. It will also explained thoroughly the measures taken to ensure that the design meets the requirements specified by the stakeholders.

This project portrays the design and implementation of various algorithms, flowcharts, and test cases that are necessary for managing and manipulating the data. It also includes exploration of the correct procedures for sorting and searching files, and identifies suitable data types for storing and manipulating the data.

You will find code in this project that informed my analysis and were not implemented in the final code.

The project aims to solve the problem of managing the data for each module, which can be challenging due to the varying number of students enrolled in each programme.

This project takes you on the journey of implementation analysis, frustrations in trying to achieve workable code for this assignment. It concludes with invaluable learning and tremendous joy.

Part 1

Design of Data Types

The code for this design was implemented using programming language *C* and code editor VS Code. Data types best suited for the manipulating and storing of data were explored extensively.

Firstly, four files needed to be created for the different modules including:

- DT265A part time higher diploma with admission number of 13 students,
- DT265C part time masters qualifier admission number of 9 students,
- DT265B full time masters qualifier with a capacity of 14 students and
- D8900 full time international masters qualifier admission number of 6 students

Defined constants for the length of various character arrays included: *'NAME', 'SURNAME', 'ID'* and *'MODULE'*. The main function of structures defining the number of elements named students which contained the *name, surname, ID and course* of each student initialised in the structure. Using *fopen()* the .txt files are opened and a for loop is used to write the details of each student to their allocated file while checking if the file was opened successfully and if not displaying an error message.

A switch statement is used to determine which menu option selected by the user. The first case sorts the list using the insert() function, and the second case prints the sorted list using the print_list() function. The third case searches for full-time students using the linear_search() function. The fourth case searches for a student by surname using the binary_search() function. The fifth case sets the value of exit to 0, which will cause the while loop to exit.

There are several functions to manipulate the list of students. The fill_struct() function is used to fill the array with data from text files. The print_list() function is used to print the list of students. The insertion_sort() function is a modified version of the insertion sort algorithm that can sort a subset of an array between two given indices.

The data types used in this program include character arrays *char[]* to store name, surname, and id of the students, integer *int* for iteration, and structure to hold the information of the

students. Additionally, a file pointer *FILE** is used to open and close the text file as illustrated in Image 1 below.

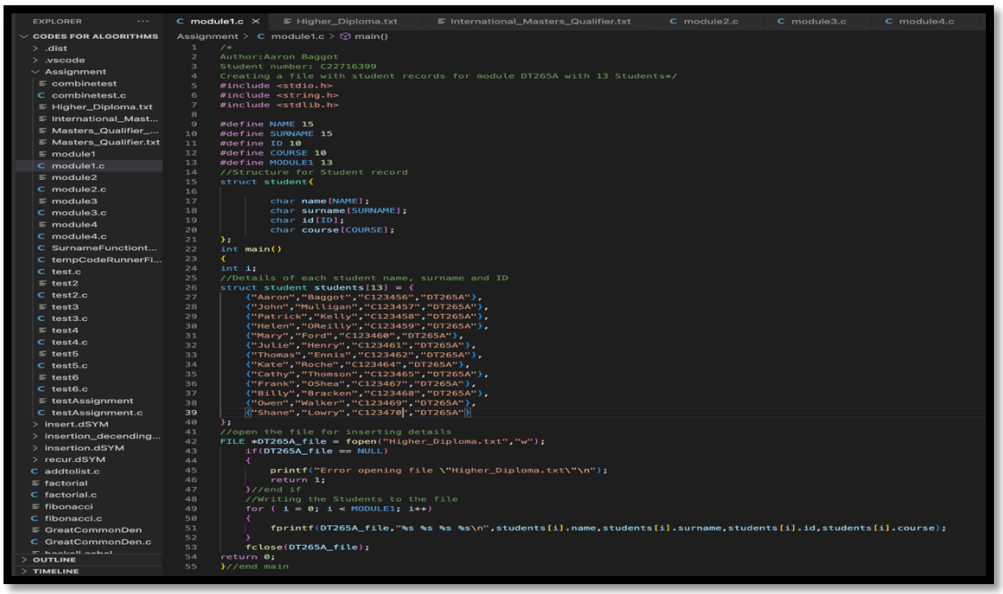


Image 1

The next step is to add the files, sorting the lists in order by surnames. The first sorting algorithm implemented is Bubble sort used as a test case as this is an algorithm being most familiar with.

The *'FILE'* data type provided by the standard library is used for opening and reading the four files *Higher Diploma, Master's Qualifier part-time, Master's Qualifier full-time, International Master's Qualifier*. Data type for representing characters 'char' was used to store the first name, surname, id and program type for each student. A pointer is used to point to the string using dynamically allocated pointers to *'char'* which stores the surnames of each student from the files. To keep track of the number of names read in from the files *'int'* (integers) was implemented. The malloc function is used which dynamically allocates a block of memory, the data is entered into the memory block displayed and freed (released) at the end when the memory block is no longer required. In the program it changes the size of an already dynamically allocated block of memory to allocate memory for the new pointer to *'char'*. String compare *'strcmp'* to compare the surnames of students during the bubble sort. (Collins, 2023). See Image 3 below for illustration.

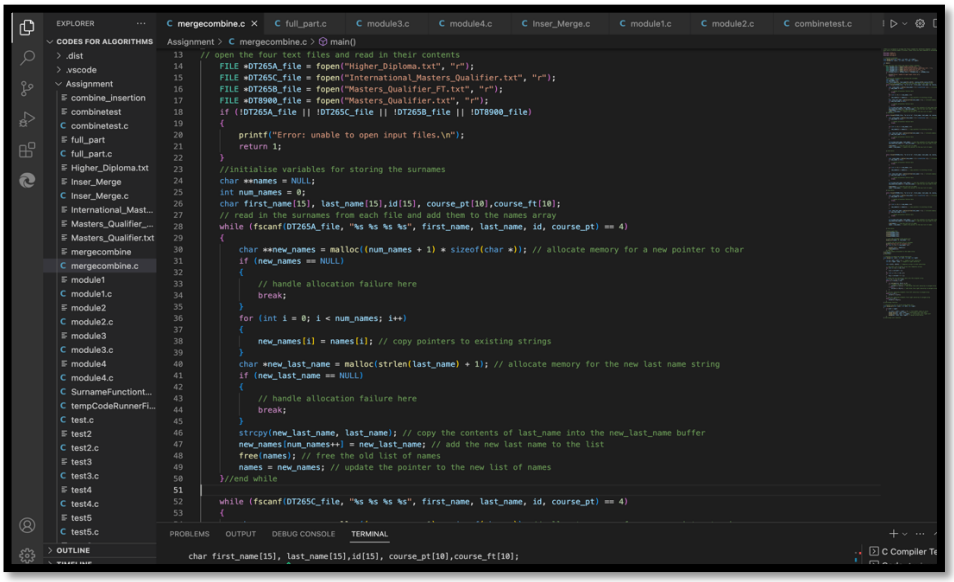


Image 2

Although bubble sort is a simple algorithm it is not efficient for sorting large lists. It has a time complexity of O(n^2) which means the number of operations in sorting the list increases quadratically with the number of elements in the list. If the list is in reverse order it will require n^2 comparisons and swaps to sort the list size n making this the worst case. Example of the code implemented and output of surnames in order can be seen in Image 3 & Image 4 below.

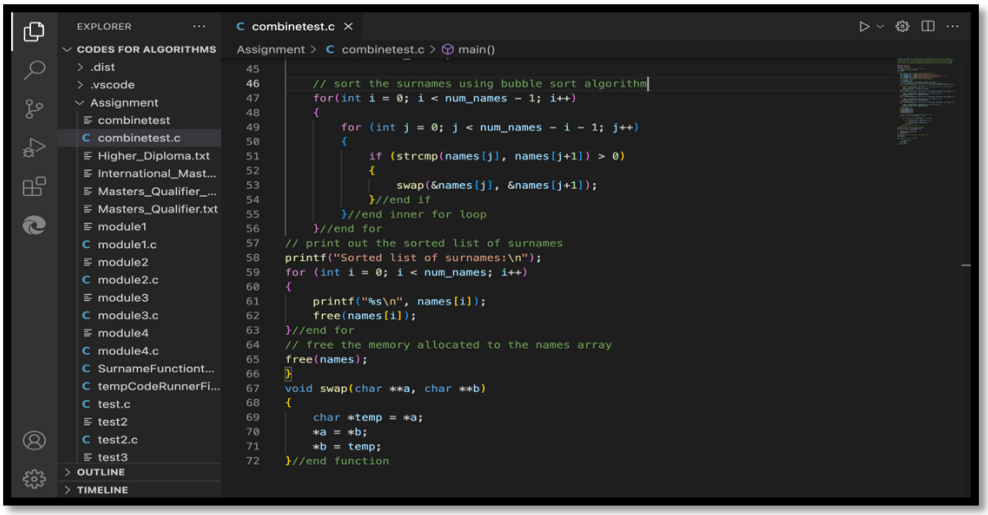


Image 3

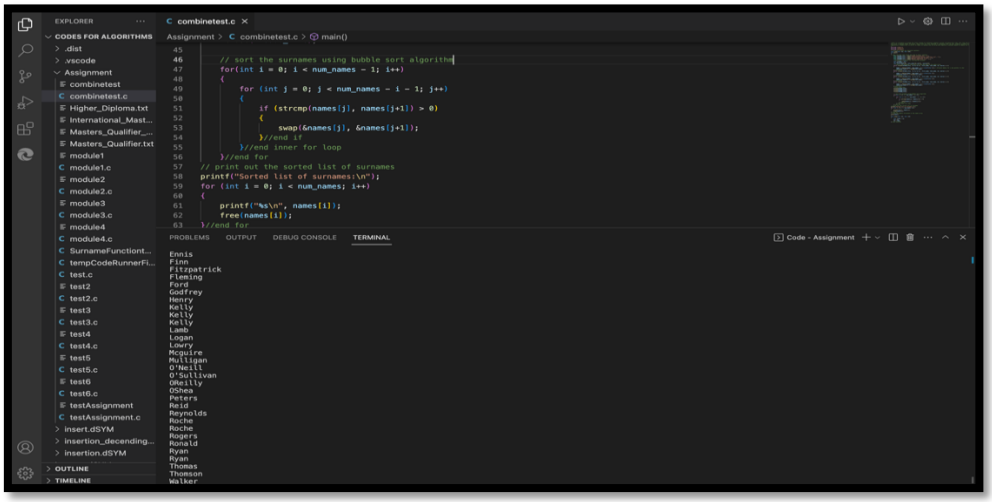


Image 4

Insertion sort (See Image 5) is another algorithm which is familiar through video demonstrations and detailed lecturer explanations. Insertion sort function works by iteratively inserting each element of the array into its correct position within a sorted subarray which starts at the beginning of the array and grows in size with each iteration. Two operations affect the run time of insertion sort, comparison operation in the inner loop and the exchange operation in the inner loop.

It is O(n^2) algorithm with the number of comparisons and the number of exchanges in the worst case. In the best case it O(n) and average case (n^2) /4 comparisons and (n^2)/8 exchanges still making it O(n^2) , however slightly better than selection sort. This makes insertion sort a really good choice for almost sorted files. Each element at index *'i'* the algorithm stores its value in a temporary variable *'key'* and starts from index 'i – 1' continuing backwards until finds an element less than or equal to 'key' shifting one place to the right making room for the 'key' to be inserted in the correct position within the sorted array. At the end of the inner loop the algorithm inserts 'key' into its correct position within a sorted subarray setting the value at 'j + 1'. The entire array is then sorted by surnames in ascending alphabetic order. The space complexity is O(1) as it uses a constant amount of additional memory for temporary variables (Kelly, 2023).

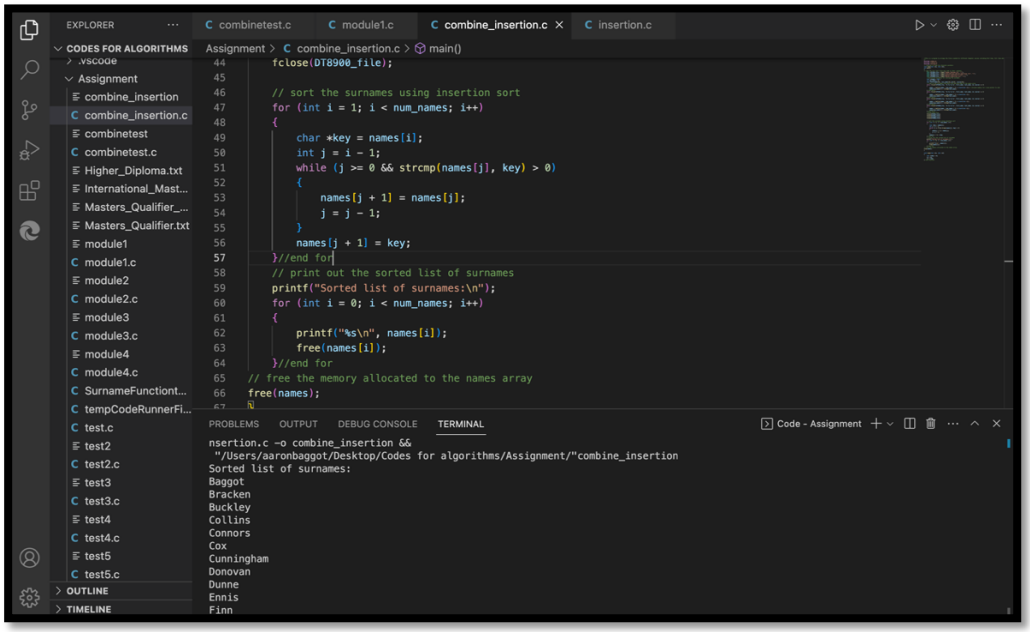


Image 5

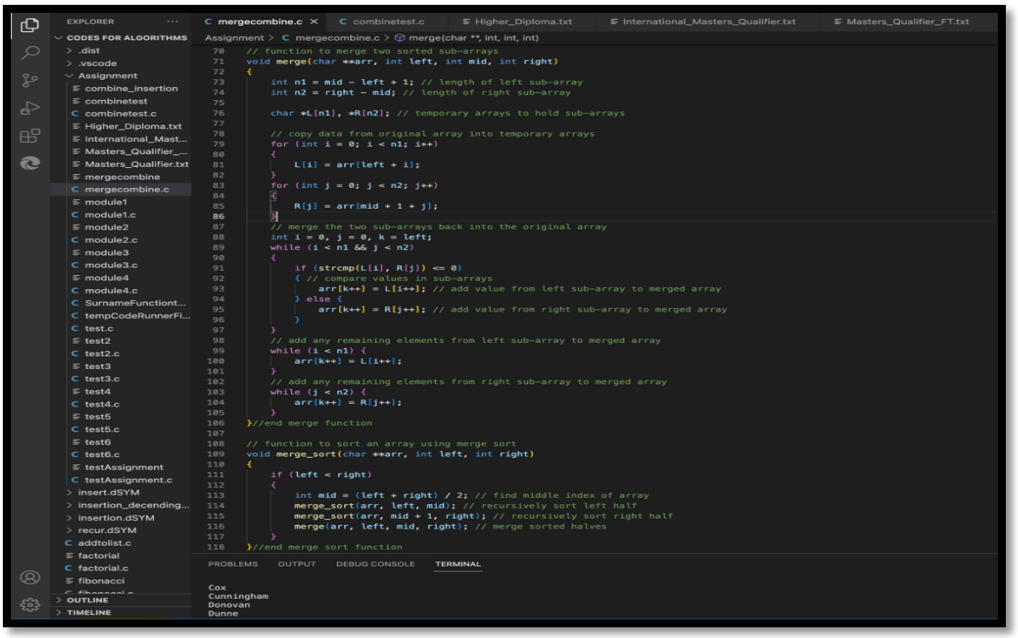


Image 6

Another algorithm used as test case for efficiency (See Image 6 above) is merge sort which has time complexity O(n), where n is the total number of elements in the two sub arrays merged. In the code used it uses '*i', 'j' and 'k'* to keep check of the position in the left, right and output arrays respectively. This means the index is incremented and decremented in a constant time for every comparison.

The code for opening the files is the same used in the bubble and insertion sorting algorithms initialising an array to store the surnames and reads in from each file adding them to the names array. Malloc allocates the memory for each string separately and copies the string into that memory using 'strcpy'. The '*merge_sort'* function is recursive that divides the array into halves sorting the left and right half. Then merging the two sorted halves into the original array in ascending order using the surnames. Dynamic memory allocation is used to allocate memory for the surnames array.

Combining insertion sort and merge sort algorithms to divide an array into two separate parts, a sorted and unsorted part. The algorithm picks values from the unsorted part and places them at the correct position in the sorted part. Basically, insertion sort is more efficient for small input sizes and runs faster when the elements are already sorted. This means there is no auxiliary space required. Merge is a divide and conquer algorithm recursively calling itself for the two halves. The advantage of this algorithm is time complexity of O(n(k+log(n/k)), n being the number of elements in the list. If k = 1 merge sort is better time complexity and if k = n insertion sort is better in space complexity. These functions are illustrated in Image 7 below. Using '*hybrid_sort'* if the size of the array is small enough use insertion otherwise merge sort using standard merging mechanism (Faizahafiz, 2022).

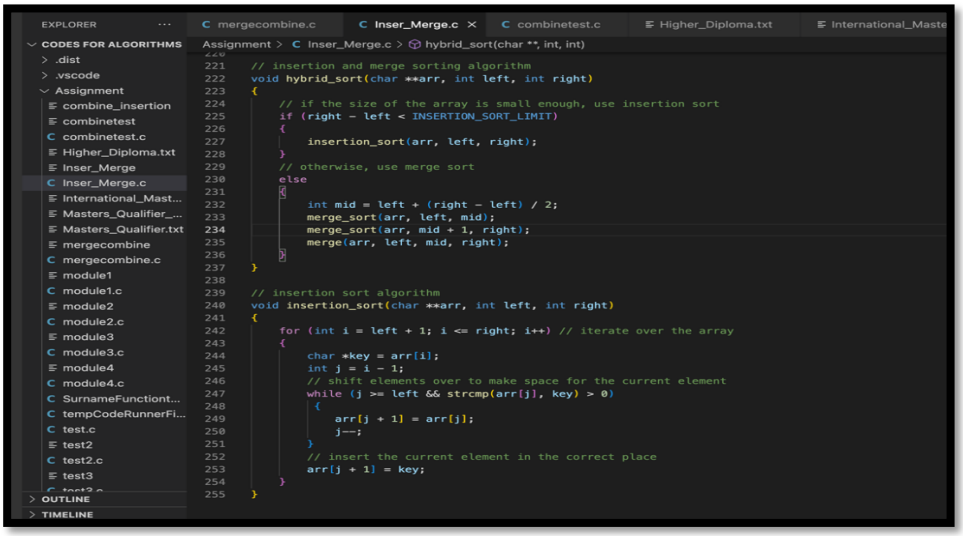


Image 7

Having tried several test cases changing the code to suit the requirements of the program. The program starts by declaring variables and opening files that contain information about the students. The DT_mix array is created to store the information from the files. A menu (See Image 8 below) is displayed for the user with various options such as sorting the list, displaying the list, searching for a student, and exiting the program. The user's input is read and processed using a switch-case statement. Select option 1 to sort the list, the *"insert()"* function is called to sort the list in ascending order by the student's surnames. Select option 2 to display the list, the *"print_list()"* function is called to display the list of students. The *"fflush()"* is a function used which returns 0 if the call is successful and the buffer is flushed, therefore if unsuccessful returns EOF (End of File). This is implemented for data in the buffer memory to get flushed when a new line character (Singh,2022).

The code *"size_array=sizeof(DT_combined)/sizeof(DT_combined[0])"* calculates the number of elements in an array called *"DT_combined"* by dividing the size of the array by the size of a single element in the array. This gives the total number of elements in the array and assigns it to a variable called *"size_array"*.

If the user selects option 3 to display full-time students, the *"linear_search()"* function is called to search the list for students whose course is *"Full_Time"* and display their information.

If the user selects the option 4 to search for a student surname, the program prompts the user to enter a surname to search for. If the list has not been sorted, the program displays a message asking the user to sort the list first. Otherwise, the program calls the *"binary_search()"* function to search the list for the specified surname and display the information of the student with that surname. Finally, if the user selects the option to exit, the program ends by displaying a message saying *"Goodbye"*.

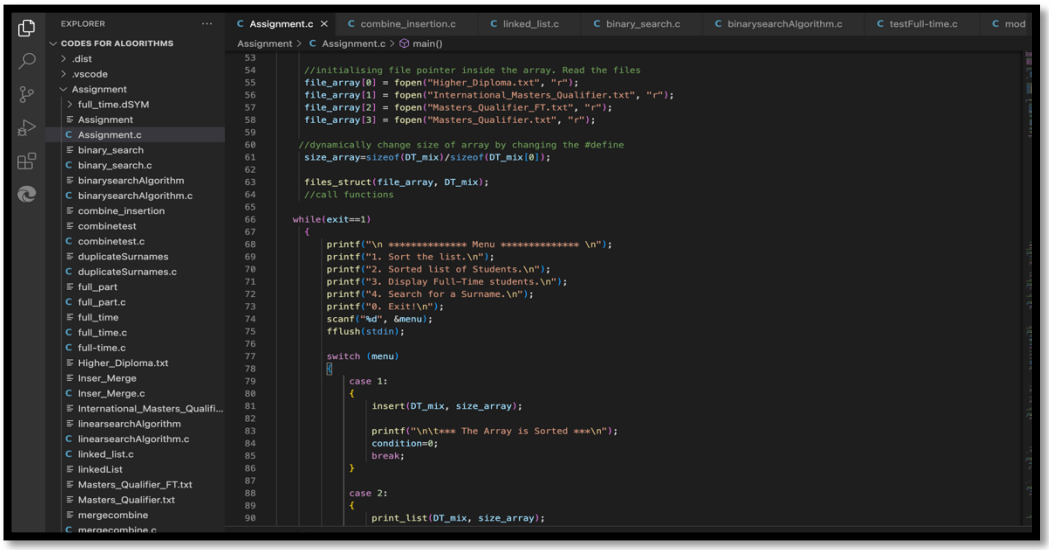


Image 8

Part 2

This section outlines the information in flowcharts and explain the big O notation. The combining list flowchart was implemented in the test cases.

Flowchart

The three flowcharts below are used to illustrate the steps taking to add students details, combine lists and set up a merge sort.

1. List for students

This flowchart illustrates the adding of students details their appropriate .txt file.

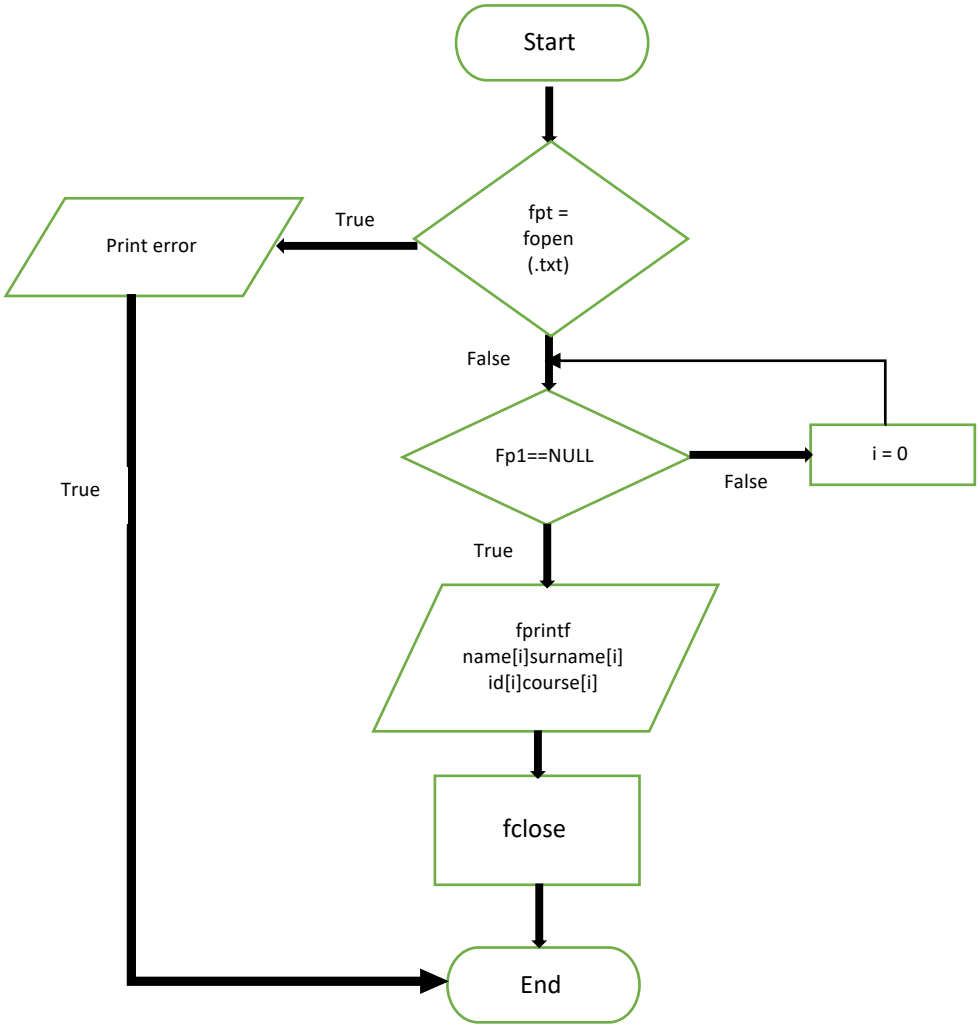
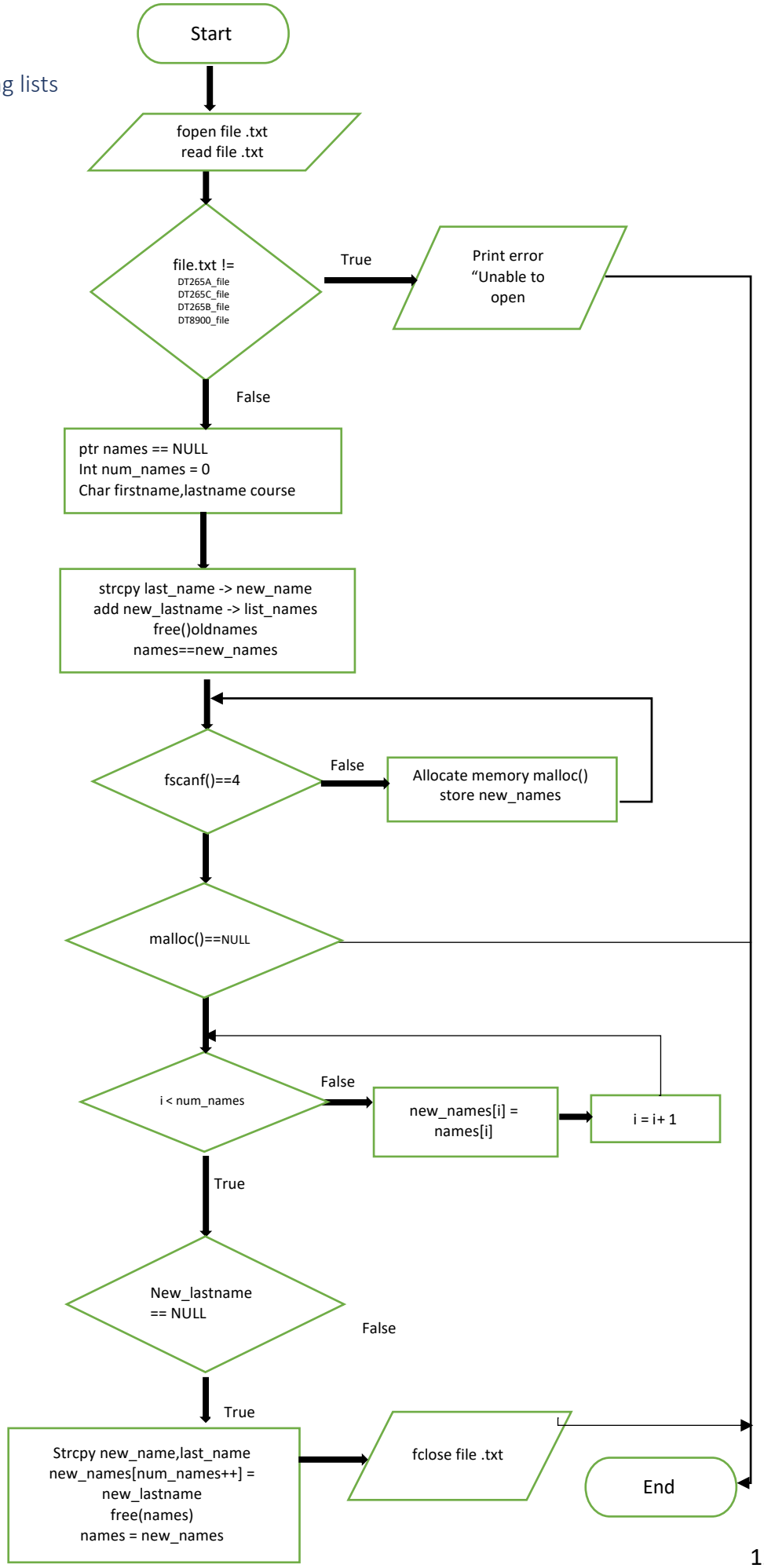


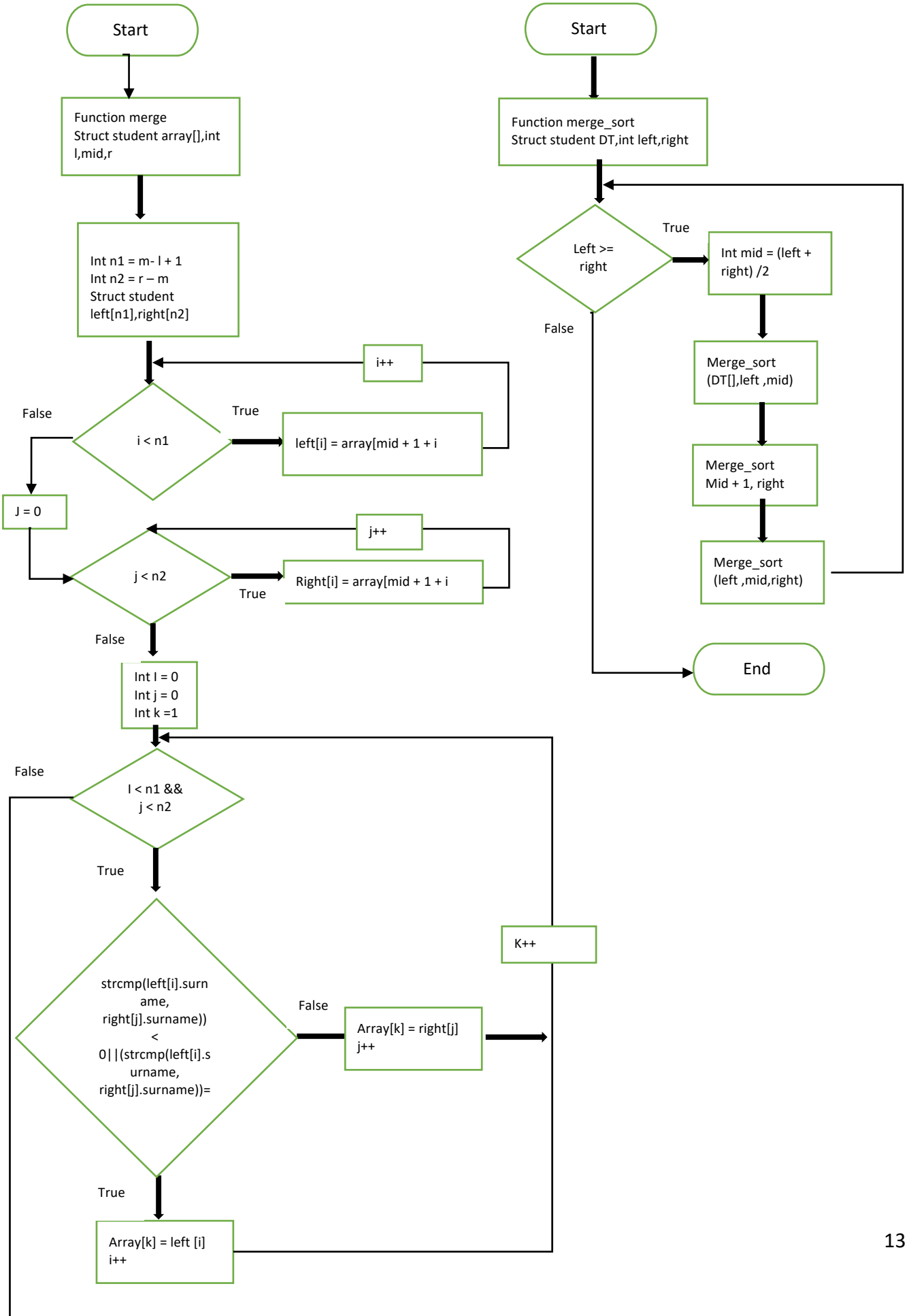
Diagram 1

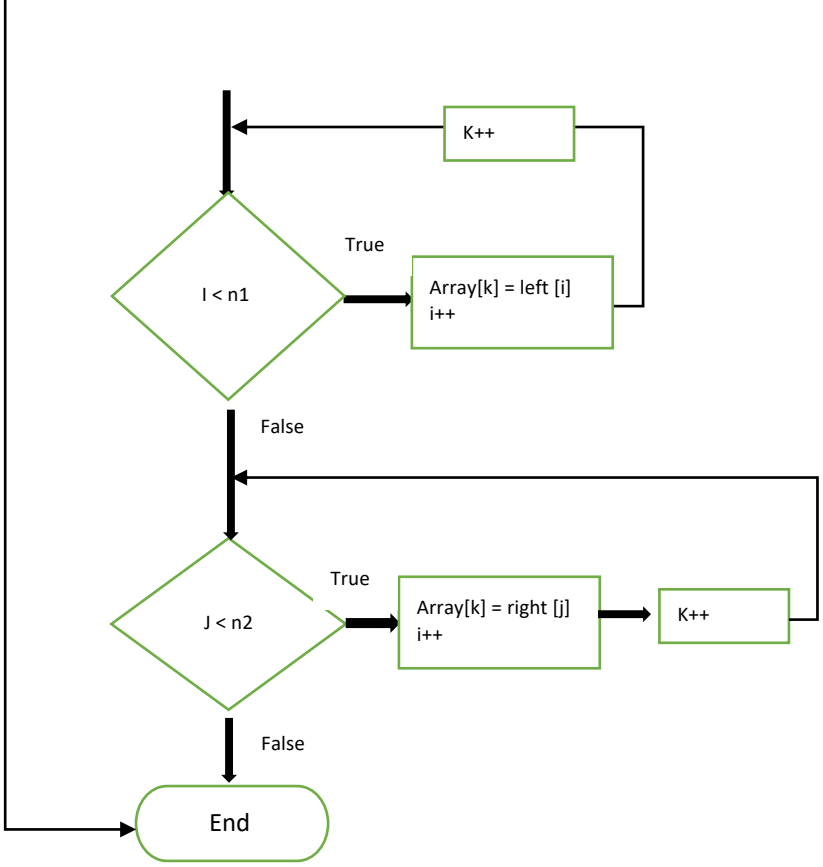
2. Combining lists



3. Merge sort

Merge sort algorithm implemented to search for a student's surname from the combined list.




K++
I < n1
True
Array[k] = left [i]
i++
False
J < n2
True
Array[k] = right [j]
i++
K++
False
End

The Big O

The big O notation is a way of expressing the time complexity of an algorithm in terms of the number of input values it processes. In this code, there are several loops that read in data from text files and store the surnames in an array. The time complexity of these loops depends on the number of surnames in each file.

The time complexity of the while loop that reads in the surnames from each file and adds them to the names array is O(n^2). This is because for each surname read in, the loop allocates memory for a new pointer to char and a new last name string, and then copies the pointers to the existing strings into the new_names array. The loop then copies the contents of last_name into the new_last_name buffer and adds the new last name to the list. Finally, the loop frees the old list of names and updates the pointer to the new list of names. All of these operations take O(n) time, where n is the number of surnames in each file. Since there are four files being read, the total time complexity of this section of the code is O(4n^2), which simplifies to O(n^2). Overall, the time complexity of this code is dominated by the O(n^2) loop that reads in the surnames from the text files, so the overall time complexity is O(n^2) (McConnell, 2008).

The time complexity of the merge sort algorithm used in the insert function is O(nlog(n)) which is considered to be an efficient sorting algorithm. The time complexity of the loop is O(n), which is linear. Therefore, the overall time complexity of the insert function is O(nlog(n)) + O(n) = O(n*log(n)).

Part 3

This section portrays how to search for full time students, the order of growth and assemble a linked list.

Algorithm search full time students

The code takes an array of student structs *"DT"*, the size of the array and character pointer *"student type"*. This function searches through the array for students whose course matches the *"student type"*. string, and then prints out the details of those students.

The function first declares a character array named course and initialises it with the string *"Full_Time"*. It then loops through the array of student structs, checking if the course member of each struct matches the *"student type"* string using the strcmp() function. If there is a match, it prints out the details of the student using the printf() statement.
This function searches for Full-Time students, since the course array is initialised to *"Full_Time"* (see Image 9 below).

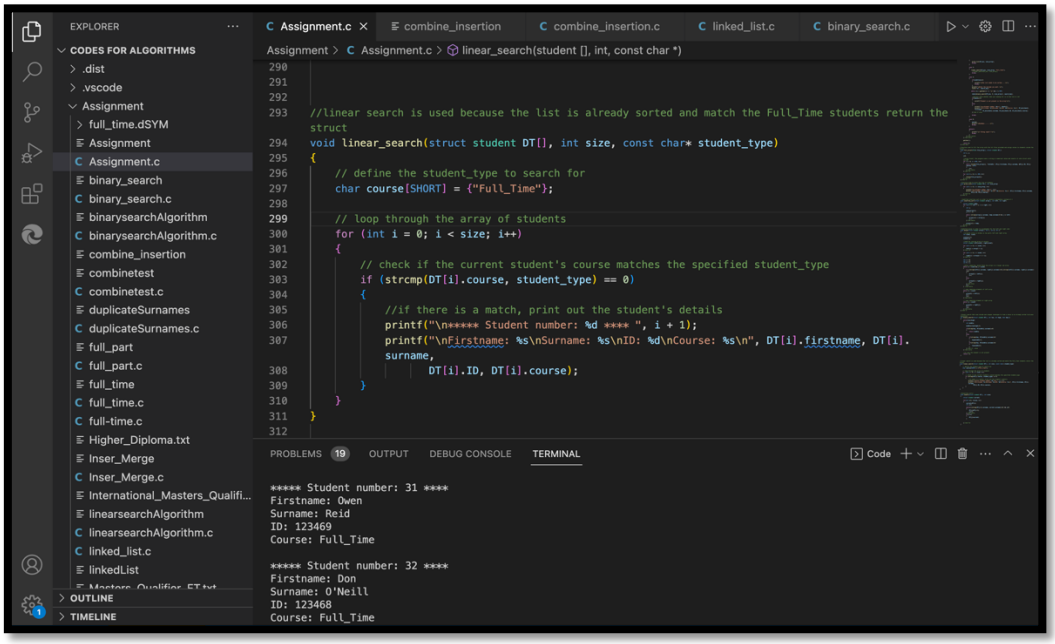


Image 9

```
Procedure List Full Time Students
        FOR each file DO
                WHILE fscanf(file, "%s %s %s %s", first_name, last_name, id, course_type)
        equals 4 DO
                        IF strcmp(course_type, "Full_Time") equals 0 THEN
                                // allocate memory for new array of names
                                char **new_names = malloc((num_names + 1) * sizeof(char *))
                                strcpy(new_last_name, last_name)
                                // Add new_last_name to the end of new_names array
                                        new_names[num_names++] = new_last_name
                         END IF
                END WHILE
        END FOR
        // Sort the names array in alphabetical order
                merge_sort(names, 0, num_names - 1);
        // Print out all full-time students
        FOR i = 0 to num_names - 1 do
                printf("%s\n", names[i]);
        END FOR
                // Free memory used by names array
        FOR i = 0 to num_names - 1 do
                free(names[i]);
        END FOR
                free(names);
END Procedure List Full Time Students
```

Modified version used in the program

```
START Procedure List Full Time Students
        char course = “Full_Time”
        FOR i from 0 TO size -1 DO
                IF DT[i].course equals student_type THEN
                        Print “Student Number” + i + 1
                        Print “Firstname” + DT[i].firstname
                        Print “Surname” + DT[i].surname
                        Print “ID” + DT[i].ID
                        Print “Course” + DT[i].course
                END IF
        END FOR
END Procedure List Full Time Students
```

Order of growth

The order of growth, or time complexity, of the linear search function is O(n), where n is the size of the input array DT. This is because the function loops through the array once, checking each element to see if its course matches the specified student_type. In the worst case, where none of the elements in the array match the student type, the function would have to iterate through all n elements. Therefore, the time taken by the function to execute is proportional to the size of the input array, which is n.

Although the input array is sorted, this doesn't affect the time complexity of the function, since it still has to loop through all elements of the array to check for matches. If the array was not sorted, the linear search would still be required and the time complexity would still be O(n).

Linked list

This section of the code is attempted in a test case but not implemented in the full program.
This C code reads student data from four files and creates a linked list of students who are enrolled in full-time courses. Then, it sorts this linked list of students by last name using merge sort algorithm and finally, it prints out the last names of all students in the linked list.

A struct called “student” is defined, a character array for the student's last name and a pointer to the next student in the linked list. Then, a typedef is created to define the “Student” type as a struct student pointer.
Function prototypes are declared for the merge sort and merge functions that will be used later in the code.
In the main function, four files are opened for reading, and their contents are read into the linked list of students who are enrolled in full-time courses.
For each file, the data is read using the fscanf function, and if the course type is “Full_Time”, a new student struct is created dynamically using the malloc function, the last name of the student is copied into it, and it is added to the linked list.
After reading all four files, the linked list is sorted by last name using the merge sort algorithm, and the sorted last names of all students in the linked list are printed.
The merge_sort function is called with the head of the linked list as an argument. It is a recursive function that divides the linked list into two halves and calls itself on each half until the base case of a single-element list is reached.
The merge function is called from within the merge_sort function and merges two sorted linked lists into a single sorted linked list. It takes pointers to the heads of the two linked lists as arguments and modifies the head of the first linked list to point to the merged linked list

Step by step instructions are outlined below on assembling a linked list, this information is also displayed in image 10:

- Open the four text files: DT265A_file, DT265C_file, DT265B_file, and DT8900_file.
- Create a linked list to store full time students, with head and tail pointers initially set to NULL.
- Loop through each file using a while loop and read in the data line by line using fscanf() function.
- For each line, check if the course_type is “Full_Time”.

- If the course_type is “Full_Time”, allocate memory for a new student using malloc() function, and copy the last_name to the new student's last_name field using strcpy() function.
- If the head is NULL, set the head and tail pointers to point to the new student.
- If the head is not NULL, set the tail pointer's next field to point to the new student, and then update the tail pointer to point to the new student.
- Repeat steps checking course type “Full_Time” for each file.
- Once all full time students are added to the linked list, sort the list by last name using the merge_sort() function.
- Print the full time students' last names in sorted order by traversing the linked list using a while loop and printing each student's last name using printf() function.
- Close all four files and return 0.

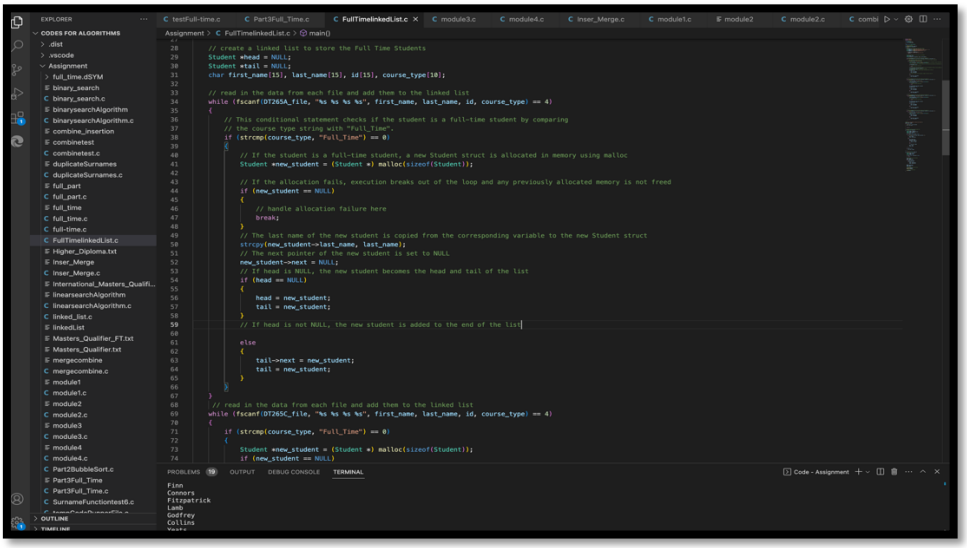


Image 10

Part 4

Linear Search and Binary Search are outlined in this section along with their pseudocodes.

Linear Search

There are two different types of search algorithms that are familiar through lecture notes and research that were implemented, however each has their own attributes and would depend on the requirements of the program. Linear search (see Image 11) implements a linear search algorithm that searches through an array of student structures to find the "*Full_Time"* students, whose course matches the specified student_type. It uses a loop to iterate through the array elements and check each element's course field. This has a worst case time complexity of O(n), n being the number of Surnames in the array.

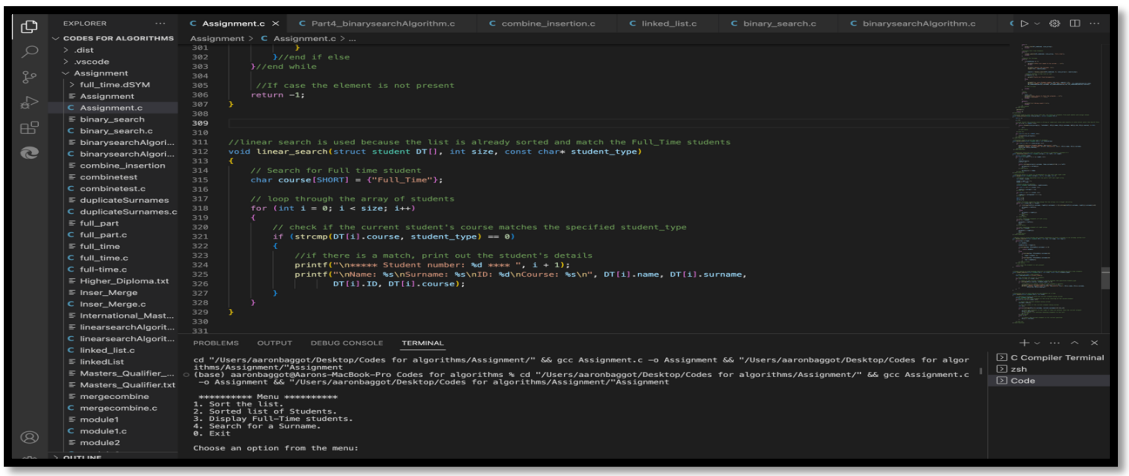


Image 11

The algorithm meets the requirements of finding *"Full_Time"* students by prompting the user from the menu option. The time complexity of the linear search algorithm is O(n), where n is the size of the array. The worst-case scenario occurs when the Full_Time students are located at the end of the array. In this case, the algorithm will have to check all n elements of the array. In the best-case scenario, the *"Full_Time"* students are located at the beginning of the array, and the algorithm will only need to check the first few elements. In the average case, the Full_Time students can be found anywhere in the array, and the algorithm will need to scan approximately half of the elements. The big O notation is O(n), which means that the time complexity of the algorithm increases linearly with the size of the input array.

Pseudocode Linear Search

Pseudocode used in the test case.

```
Procedure Linear Search Surname
                printf “Enter a Surname to search for”
                declare char search_name[15]
                read surname into search_name
                found = 0
        for i in the range [0, num_names]
                if surname at index i == search_name
                        print “found search_name at index I”
                        found = 1
                end if
        end for
                if found = 0
                        print “Could not find Surname search_name”
                end if
END Procedure Linear Search Surname (Mulani, 2022).
```

Modified pseudocode for Linear Search in the program.

```
Procedure Linear Search Surname
function linear_search(struct student DT[], int size, char student_type[])
   //set the course to search for as “Full_Time”
   char course[SHORT] = {“Full_Time”}
   // loop through the array of students
   FOR (i = 0; i < size; i++)
      // check if the current student's course matches the specified student_type
      IF (strcmp(DT[i].course, course) == 0)
         //if there is a match, print out the student's details
         print student's details
      END IF
   END FOR
END Procedure Linear Search Surname
```

Binary Search

Binary search is implemented with the modification of the code that finds a specific surname in a sorted array of student records (see Image 12). The algorithm works by dividing the search range in half and comparing the middle element with the key. If the key matches the middle element's surname, the index of the middle element is returned. the search range is updated based on whether the key is greater or smaller than the middle element's surname, and the process is repeated until the key is found or the search range is reduced to an empty set.

The binary search operation takes O(log(n)) time in the worst case and the algorithm performs it for n elements. Therefore in the code provided the overall time complexity of the algorithm is O(log n) n being the number of elements in the sorted list.

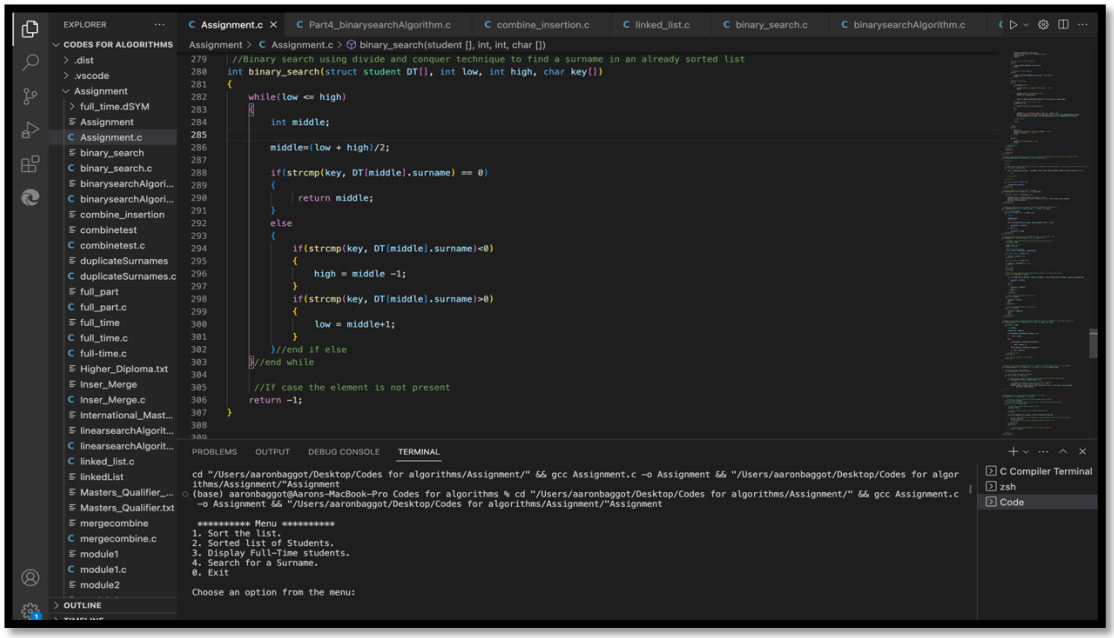


Image 12

Pseudocode Binary Search

```
ProcedureBinary Search Surname
    if (condition == 1)
       printf "The List needs to be sorted"
       break;
    end if

       printf "Enter the Surname you want"
       scanf search_key
       int c;
       while ((c = getchar()) != '\n' &&  c != EOF)  // clear input buffer
       int check = binary_search(DT_mix, 0, size_array-1, search_key)
    // perform check whether what you are looking for is in the list or not
    if (check == -1)
       printf "Could not find Surname"
       else
                        printf "Student number" check+1
                        printf  "Firstname: Surname: ID: Course: "
              DT_mix[check].firstname, DT_mix[check].surname
              DT_mix[check].ID, DT_mix[check].course
       end else
     end if
       int binary_search struct student DT[], int low, int high, char key[]

  while low <= high

    int middle = (low + high) / 2
    if (strcmp(key, DT[middle].surname) == 0)
       return middle;
        else

                if (strcmp(key, DT[middle].surname) < 0)
            high = middle - 1

                 else
              low = middle + 1
            end else
          end if
          end else
      end if
          // in case the element is not present
          return -1
  end while
```

Part 5

This section considers the testing of the project using Bubble Sort, Insertion Sort and Merge Sort algorithms.

Testing

Bubble Sort:

Bubble sort (see Image 13) is used as a test case as this is an algorithm which is easy to implement and very familiar with. Having tested the algorithm it was easier to modify and test various sorting algorithms to make the program more efficient and effective. This algorithm when tested sorted the Surnames in ascending order

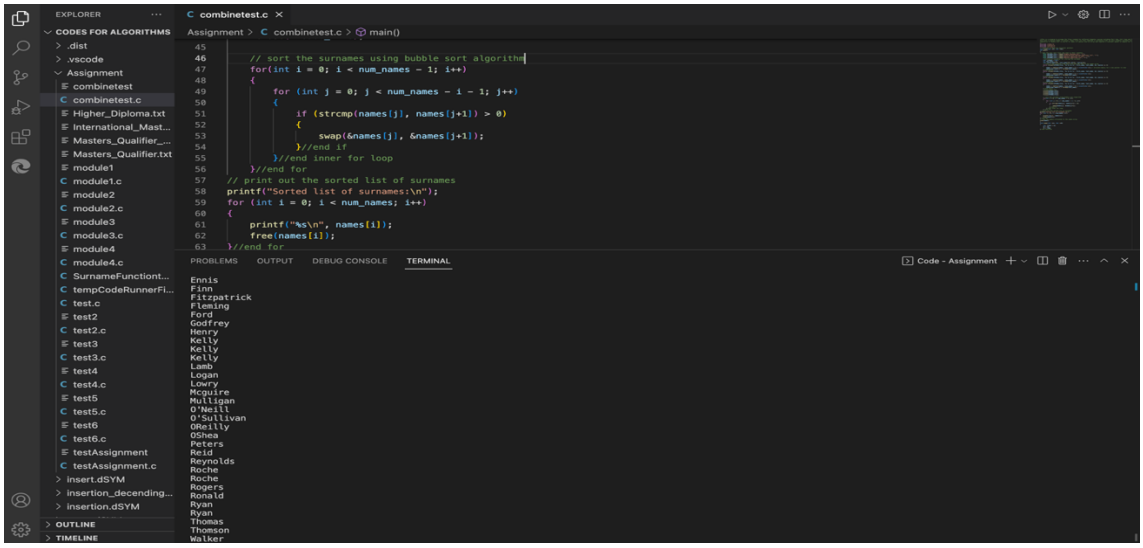


Image 13

Image 14 illustrates the implementation of Insertion sort as it is more efficient than Bubble sort.

Insertion Sort:

The insertion sort algorithm is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages such as its simplicity, low memory usage, and efficient for small data sets. The algorithm works by iterating over the array, and for each element, it moves the element to its correct position in the sorted array.

A test case for insertion sort:

An unsorted array of Surnames and after applying the insertion sort algorithm, the sorted list is in ascending order alphabetically.

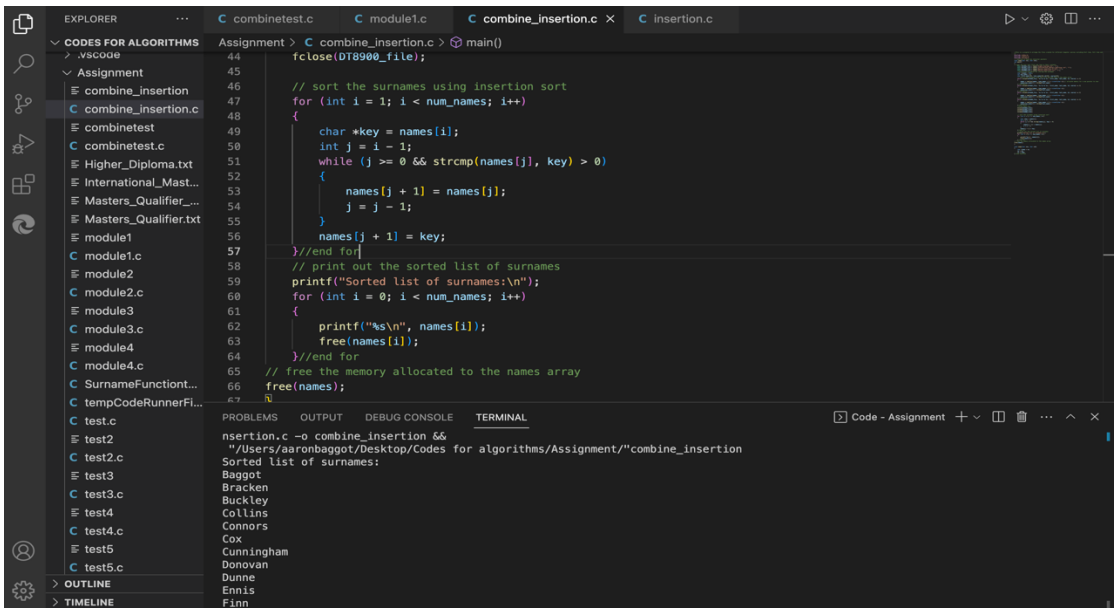
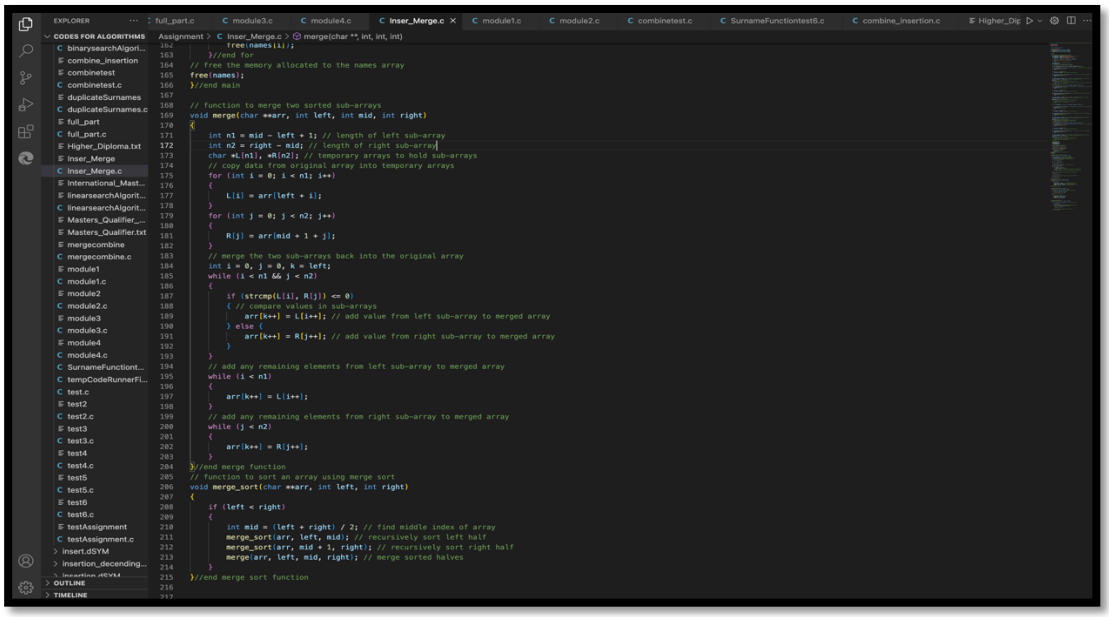


Image 14

Merge Sort:

The merge sort (see Image 15 below) algorithm is a sorting algorithm that divides the unsorted list into n sub lists, each containing one element, and then repeatedly merges sub lists to produce new sorted sub lists until there is only one sub list remaining. It is a divide-and-conquer algorithm that sorts two smaller sorted sub-arrays and merges them into a single, larger sorted sub-array.

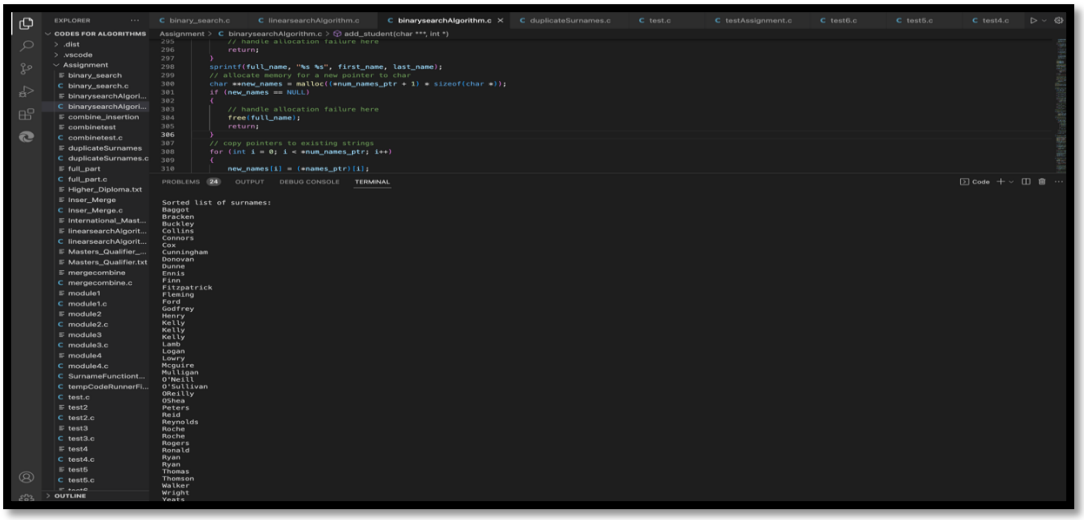


```
        free(names[i]);
    }//end for
// free the memory allocated to the names array
free(names);
}//end main

// function to merge two sorted sub-arrays
void merge(char **arr, int left, int mid, int right)
{
    int n1 = mid - left + 1; // length of left sub-array
    int n2 = right - mid; // length of right sub-array
    char *L[n1], *R[n2]; // temporary arrays to hold sub-arrays
    // copy data from original array into temporary arrays
    for (int i = 0; i < n1; i++)
    {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++)
    {
        R[j] = arr[mid + 1 + j];
    }
    // merge the two sub-arrays back into the original array
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (strcmp(L[i], R[j]) <= 0)
        { // compare values in sub-arrays
            arr[k++] = L[i++]; // add value from left sub-array to merged array
        } else {
            arr[k++] = R[j++]; // add value from right sub-array to merged array
        }
    }
    // add any remaining elements from left sub-array to merged array
    while (i < n1)
    {
        arr[k++] = L[i++];
    }
    // add any remaining elements from right sub-array to merged array
    while (j < n2)
    {
        arr[k++] = R[j++];
    }
}//end merge function
// function to sort an array using merge sort
void merge_sort(char **arr, int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2; // find middle index of array
        merge_sort(arr, left, mid); // recursively sort left half
        merge_sort(arr, mid + 1, right); // recursively sort right half
        merge(arr, left, mid, right); // merge sorted halves
    }
}//end merge sort function
```

Image 15

Image16 below demonstrates the output of the full list of students from each file combined in ascending order.



```
Sorted list of surnames:
Baggot
Bracken
Buckley
Collins
Connors
Cox
Cunningham
Donovan
Dunne
Ennis
Finn
Fitzpatrick
Fleming
Ford
Godfrey
Henry
Kelly
Kelly
Kelly
Lamb
Logan
Lowry
Mcguire
Mulligan
O'Neill
O'Sullivan
OReilly
OShea
Peters
Reid
Reynolds
Roche
Roche
Rogers
Ronald
Ryan
Ryan
Thomas
Thomson
Walker
Wright
Yeats
```

Image 16

The code (see image 17 below) was used for each module file defines a struct student that contains fields for a student's name, surname, ID, and course choice. It then creates an array of 6 student structs and initializes them with sample data.
surname, ID, and course of a single student. The fclose function is then called to close the file.

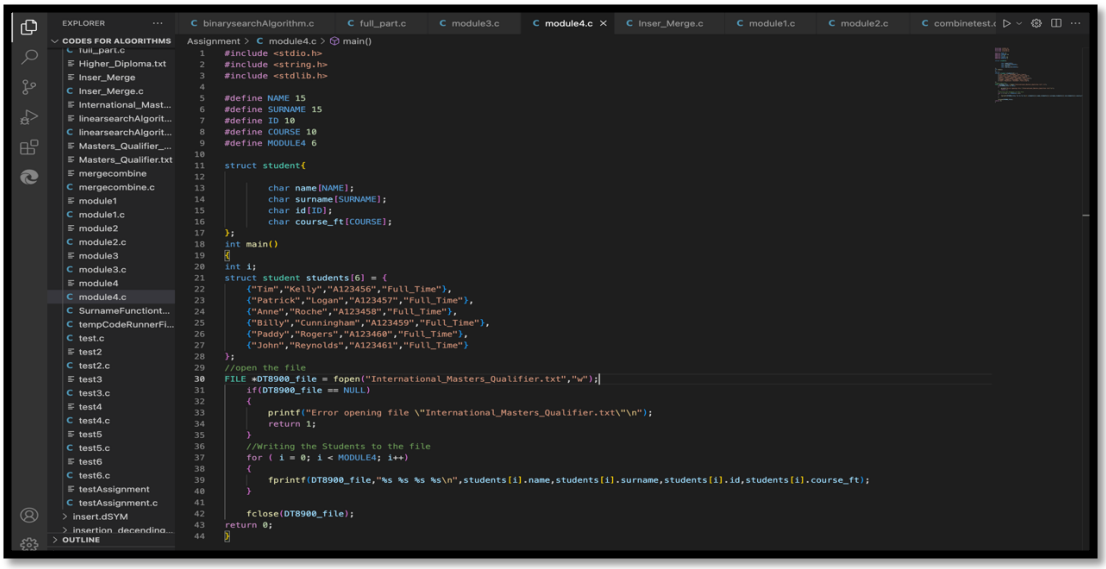


Image 17

The program then opens a file named "International_Masters_Qualifier.txt" in write mode using the fopen function. If the file fails to open, the program prints an error message and returns with an error code. The program then writes the data from the students array to the file using the fprintf function, which writes formatted data. Each line in the file contains the name,

Another test case (see Image 18 below) was to implement a function add_student which adds a new student to a list of students represented as an array of strings (char **). It takes two arguments: a pointer to a pointer to char (char ***names_ptr) and a pointer to an integer (int *num_names_ptr).

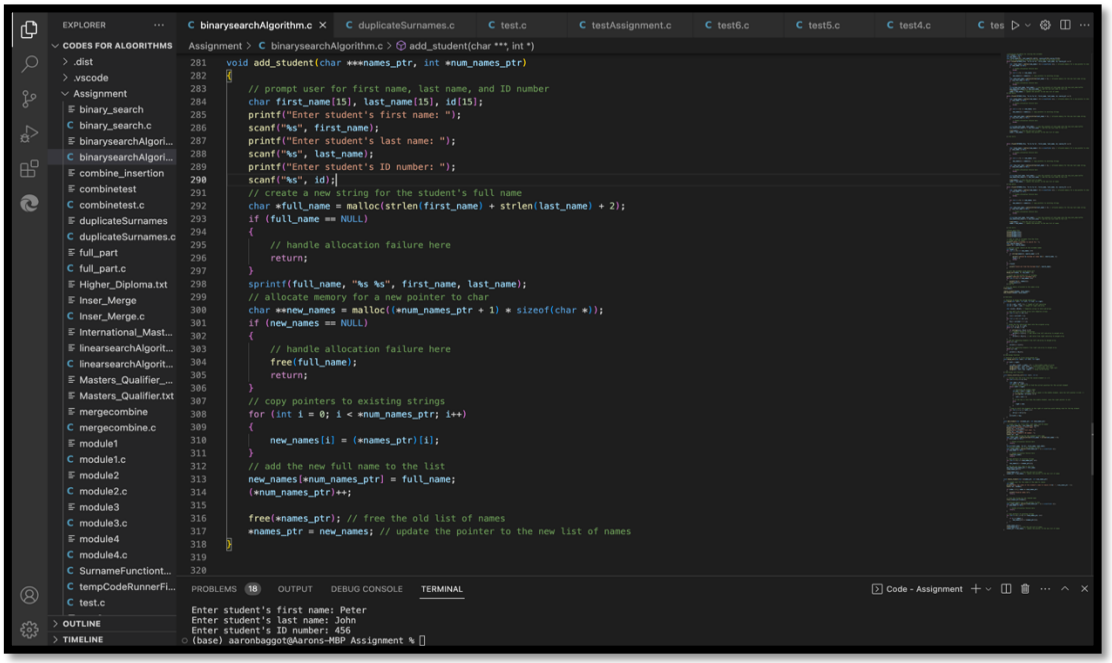


Image 18

Inside the function, the user is prompted to enter the first name, last name, and ID number of the new student using scanf. Then, a new string is created to hold the full name of the student using malloc to allocate memory. The strlen function is used to determine the length of the first and last names.

Next, a new array of strings is created using malloc to allocate memory for the new student's full name. The size of this new array is one greater than the original array, since a new student is being added. The for loop is used to copy all of the existing strings from the old array into the new array. The new student's full name is added to the end of the new array. The num_names_ptr variable is incremented to reflect the new size of the array.

Finally, the old array is freed using free and the pointer to the new array is assigned to the original names_ptr pointer so that it now points to the new array.

This following function (see Image 19 below) removes a student's name from an array of student names, given the index of the name to remove. It takes two arguments: a pointer to a pointer to char (which points to the array of names) and a pointer to an int (which holds the number of names in the array). First, the function prompts the user for the index of the name to remove and checks that the index is valid (i.e. within the bounds of the array). If the index

is not valid, the function prints an error message and returns. If the index is valid, the function frees the memory allocated for the string representing the removed student's name. It then allocates memory for a new array of pointers to char, with one fewer element than the original array.

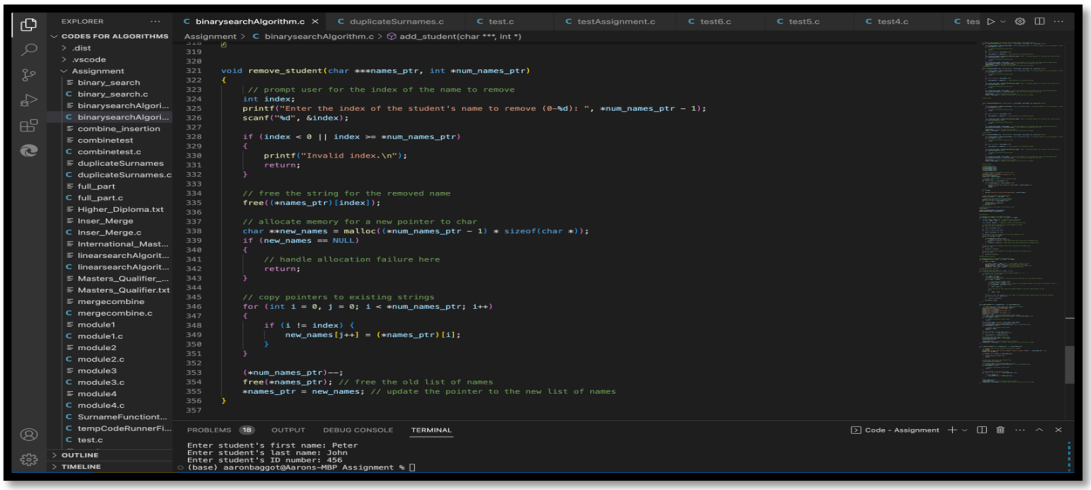


Image 19

Next, it copies all the pointers to existing strings from the original array to the new array, except for the pointer to the string that was removed. To do this, the function uses two index variables, i and j. The i variable iterates over the indices of the original array, and the j variable iterates over the indices of the new array. The j variable is only incremented when a pointer is added to the new array. After copying the pointers to existing strings, the function updates the pointer to the array of names to point to the new array, frees the memory allocated for the old array of names, and decrements the number of names in the array by one.

This test case was used to verify that the program correctly sorts the list of students by surname and displays the sorted list in a formatted table. See Image 20 below.

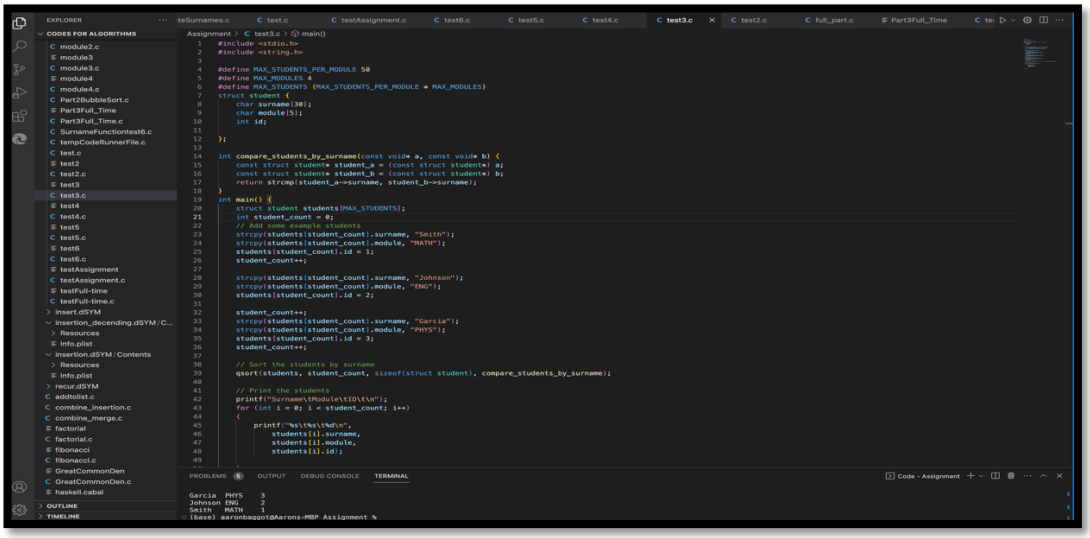


Image 20

It can also be used to verify that the program correctly limits the number of students and modules that can be managed.

This function (see image 21 below) takes in two parameters: an array of FILE pointers representing the text files to be read, and an array of struct student objects to be filled with the data from the text files. The function uses a nested loop structure to read in the data from each text file and fill the corresponding elements in the DT array. The outer loop iterates over each file in file array, while the inner loop uses the fscanf function to read in data from the current file until the end of the file (EOF) is reached.

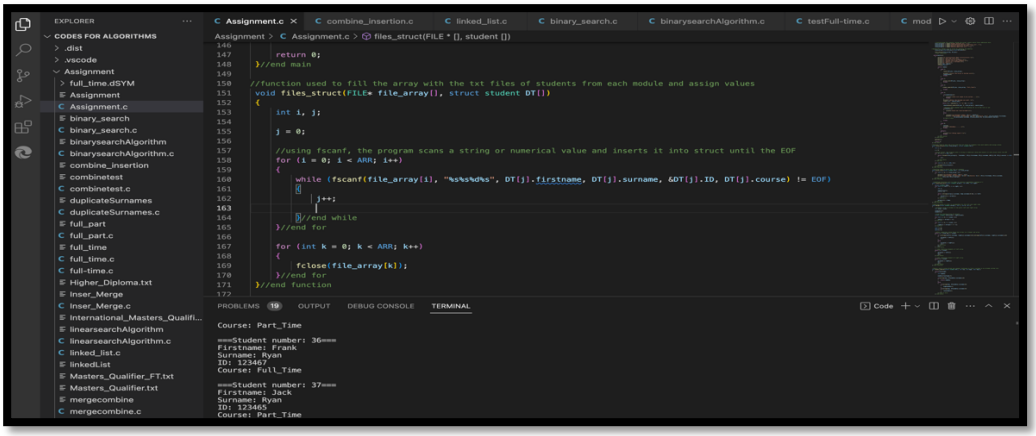


Image 21

The fscanf function reads in four values from each line of the text file, corresponding to the firstname, surname, ID, and course fields of each struct student object. The values are then

assigned to the corresponding elements in the DT array using the index variable j. The j variable is incremented after each set of values is read in, so that each set of values is assigned to a unique element in the DT array. After all of the values have been read in and assigned to the DT array, the function closes each file in file_array using the fclose function.

This function (see image 22 below) prints out the list of students stored in an array of struct student.
The function takes two arguments: the array of students (DT) and the size of the array (size_array). It uses a for loop to iterate over each student in the array and print out their information. For each student, the function prints out their student number (which is just their index in the array plus one), their first name, last name (surname), student ID, and course.

Image 22

Search for full time function (see Image 23 below) searches the "studentData" array for all full-time students. It simply loops through the array and prints the information of each full-time student.

Image 23

Binary search function performs a binary search on the *studentData* array to find a specific student record by surname. It takes three arguments: the surname to search for, the lower index of the search range, and the upper index of the search range. It recursively divides the search

range in half and compares the target surname to the middle element of the current range. If they match, the function returns the index of that element. If the target surname is less than the middle element, the function recursively searches the left half of the current range. If the target surname is greater than the middle element, the function recursively searches the right half of the current range. If the target surname is not found in the array, the function returns -1. See Image 24 below for this illustration.

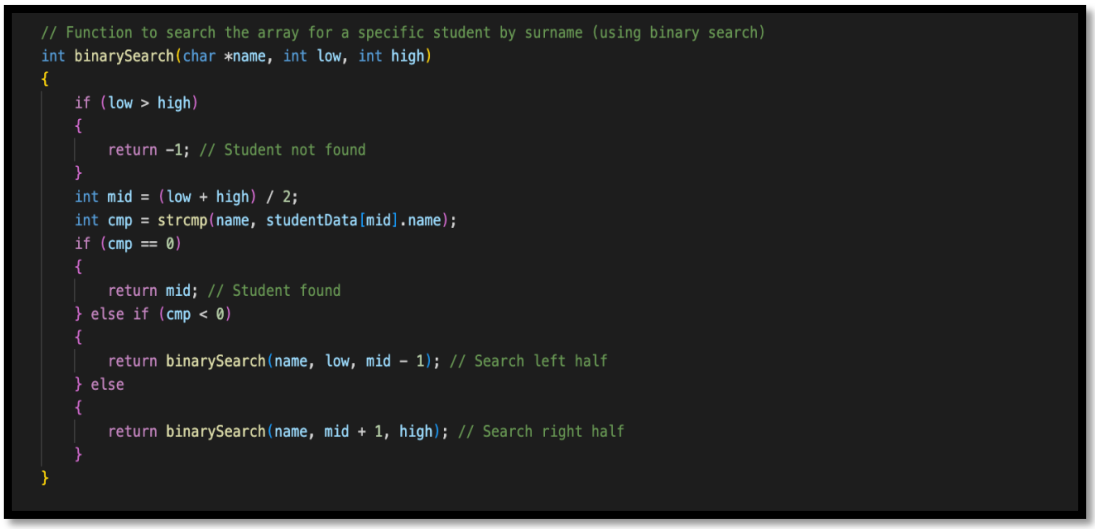


Image 24

Search student function (see Image 25 below) first calls *sortStudents()* to ensure that the studentData array is sorted by surname. Then it calls *binarySearch()* to search the array for the student record with the specified surname. If the record is found, the function prints the information of that student. If not, it prints an error message.

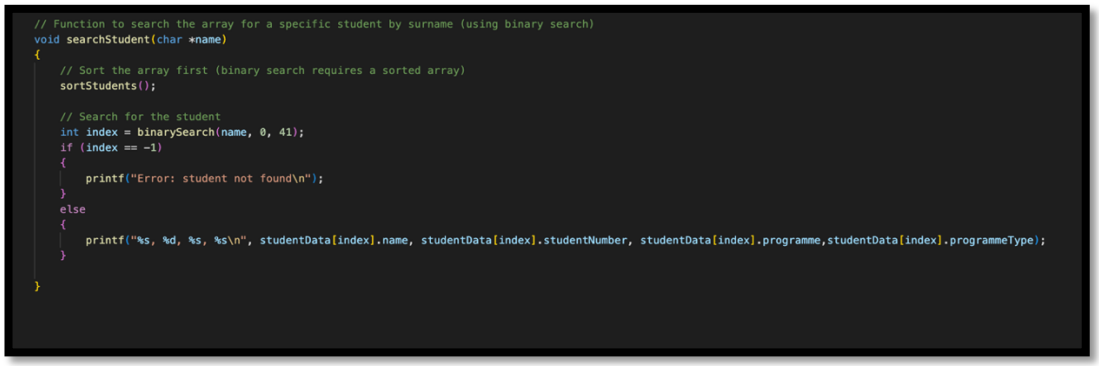


Image 25

Conclusion

In the assimilation of information for this assignment several factors came to light. C programming language is a powerful language for creating efficient and effective programs through design of various algorithms. It's design of data types in respect of their importance. The capacity of flowcharts depicting the program implemented helped to visualise the program's logic is enormous. In addition, following an in-dept analysis in relation to the two search algorithms, linear search and binary search, both commonly used in programming, it was concluded that binary search is more efficient than linear search for larger datasets.

This report also concludes that testing is extremely important step in the development of any software application. Various testing techniques were used, namely Bubble Sort, Insertion Sort, and Merge Sort. However, the discussion concluded that the implementation of Insertion Sort and Merge Sort, proved to be a most challenging task due to their big-time complexity. However, with careful research and attention to detail, these challenges can be overcome as demonstrated within the project.

A major conclusion of the assignment is the realisation that developing efficient algorithms is crucial to meeting the user's specifications. In today's world, where time is of the essence, software applications that perform optimally are highly sought after. Therefore, creating programs that are both efficient and effective are more likely to be a success in terms of design and user engagement.

Personal Gained

The learning for attained form researching content and developing this report proved to be considerable. Through the implementation of the various algorithms and data structures in this report, with the power of the C programming language in developing software applications became evident to me. By doing this assignment the knowledge attained will never be forgotten. The challenges faced and overcome in the implementation of some of the algorithms as discussed required careful research and demonstrated to me the importance of research and paying attention to detail, if I am to achieve optimal program performance.

Bibliography

Collins, M. (2023) “Notes 11 - Dynamic Memory Allocation (DMA).” Dublin: Technology University Dublin, 4 April. The realloc() function (Accessed: April 4, 2023).

Faizahafiz (2022) Sorting by combining insertion sort and merge sort algorithms, GeeksforGeeks. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/sorting-by-combining-insertion-sort-and-merge-sort-algorithms/ (Accessed: April 4, 2023).

Kelly, C. (2023) “Algorithm Design & Problem Solving: Sorting Tutorial.” Technology University Dublin: Technology University Dublin, 16 February.
Underlying data structure (array, linked list, etc.).
How comparison is carried out – upon the entire datum or upon parts of the datum (the key)?
To describe algorithms we need a language which is less formal than programming languages (implementation details are of no interest in algorithm analysis) easy to read for a human reader.

McConnell, J.J. (2008) Analysis of algorithms: An active learning approach. Boston: Jones and Bartlett Publishers (Accessed: April 6, 2023).

Mulani, S. (2022) Linear search algorithm and implementation in C, DigitalOcean. DigitalOcean. Available at: https://www.digitalocean.com/community/tutorials/linear-search-algorithm-c (Accessed: April 6, 2023).

Singh, P. (2022) Fflush() in C, Scaler Topics. Scaler Topics. Available at: https://www.scaler.com/topics/fflush-in-c/ (Accessed: April 10, 2023).

What is linked list (2023) GeeksforGeeks. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/what-is-linked-list/ (Accessed: April 12, 2023).