**ASSIGNMENT REPORT**

**TU856 / 1**

***ALGORITHM DESIGN AND PROBLEM SOLVING***

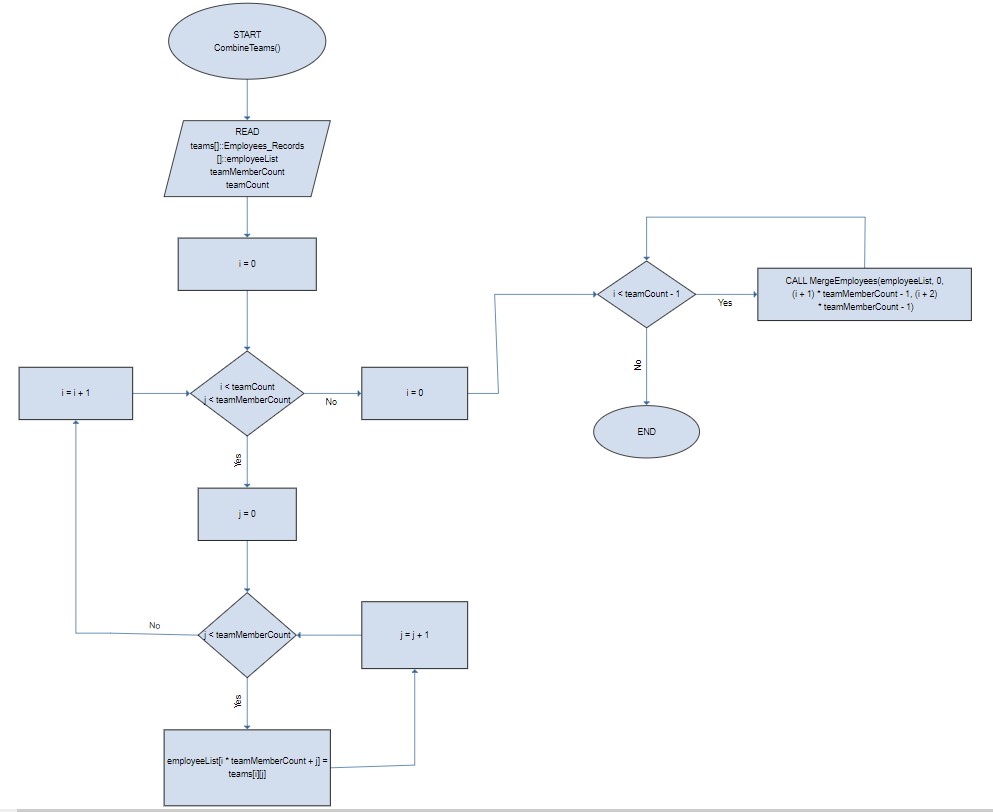
22/04/2022

By Paulina Czarnota C21365726

# DELIVERABLES

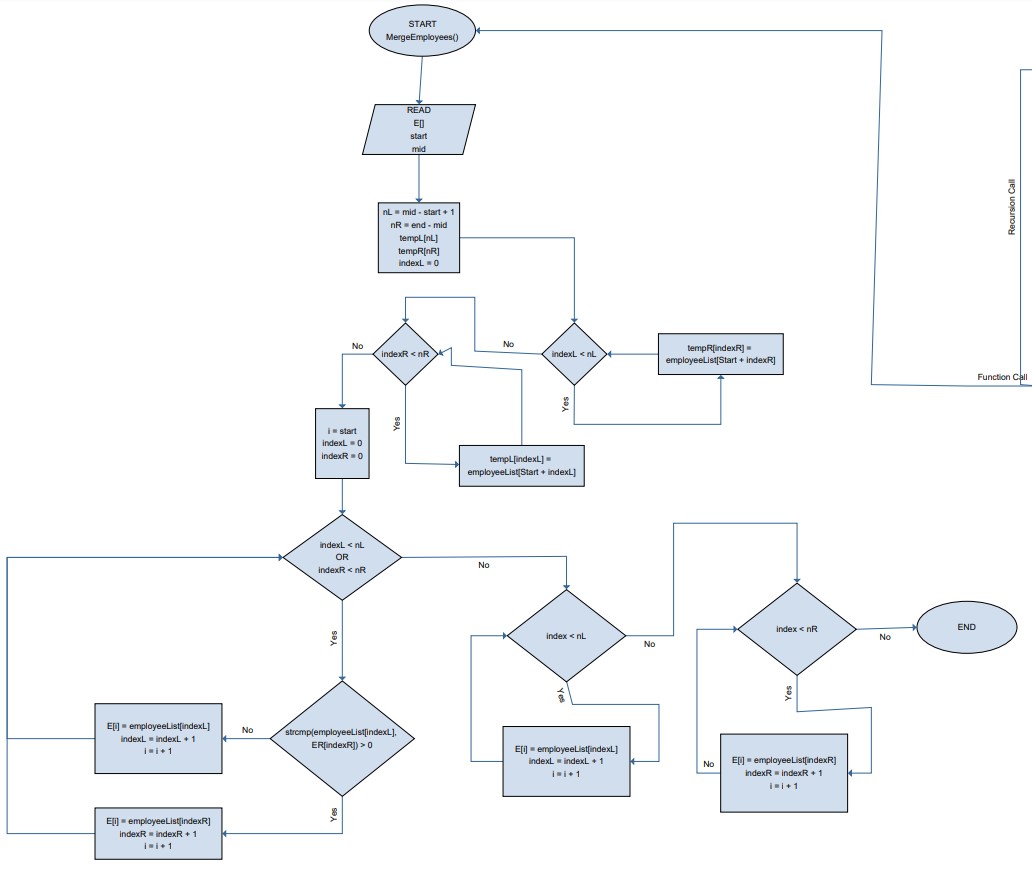
1. ***A flow chart for the combine and sort process of (a)***

**Flowchart for the combine sorted teams:**

****

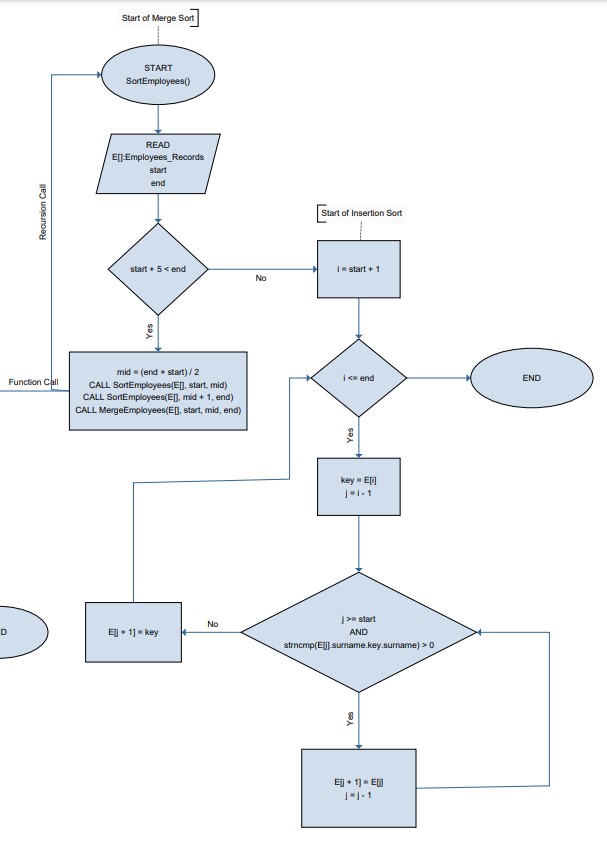
**Flowchart for the sort process:**

**Part 1**

******

**Flowchart for the sort process:**

**Part 2**

******

1. ***Pseudo-code for employees certified to work on all lines (b)***

/\*

Pseudocode for function FullCerts

This pseudocode prints the full name of all employees certified to work on all three lines

\*/

// Struct Templates

STRUCT Employees\_Records{employeeList}

// Prints a list of employees ceritified for all three lines

// Params: Employees\_Records[]::Struct Employee{employeeeID, surname, firstName, workLine}

START FUNCTION FullCerts(Employees\_Records[] AS STRUCT employeeList

// For each certification in FullCerts, prints the employee's name

PRINT "Searching for emoployees with full certifcations"

FOR i = 0; i < Number\_Certs

IF (employeeList + i) workLine EQUALS 3

PRINT (employeeList + i) -> employeeID, (employeeList + i) -> surname, (employeeList + i) -> firstName, (employeeList + i) -> workLine

END IF

END FOR

END FUNCTION

1. ***Pseudo-code for the surname search in (c)***

/\*

Pseudocode for function SearchSurname

This pseudocode searches for the employee’s surname

\*/

// Struct Templates

STRUCT Employees\_Records{employeeList, String key, int start, int end}

START FUNCTION GetEmployeeBySurname(employeeList[], int count)

DECLARE surname

DECLARE index

// Prompt user for surname

DO

PRINT "Enter surname: "

ASSIGN surname TO INPUT

IF SURNAME EQUALS ""

PRINT "Surname cannot be blank"

END IF

WHILE SURNAME EQUALS ""

// Search for employee

ASSIGN index TO CALL SearchSurname(employeeList, surname, 0, count)

IF index EQUALS -1

PRINT "No employees found with the surname "" "

ELSE

PRINT "Employee Found: "

PRINT Employee details at employeeList[index]

END FUNCTION

// Prints person matching the surname given

// Uses binary search to find the person

// Employee array must be sorted by surname

// Params: employees[]::Struct Employee{employeeID, firstName, surname, line}

// key

// start

// end

// Returns: index of employee matching the surname else -1 if not found

START FUNCTION SearchSurname(employeeList[], key, start, end)

IF end GREATER-OR-EQUAL start

// Recursive case

DECLARE mid TO (start + end) / 2

IF employeeList[mid].surname EQUALS key

RETURN mid

END IF

IF employeeList[mid].surname GREATER key

RETURN CALL SearchSurname(employeeList, key, start, mid - 1)

ELSE

RETURN CALL SearchSurname(employeeList, key, mid + 1, end)

END IF

END IF

// Base case

RETURN -1

END FUNCTION

1. ***C code for (a) - (c)***

*/\**

*TU856 / 1 - ALGORITHM DESIGN AND PROBLEM SOLVING: CMPU1001 ASSIGNMENT*

*Author: Paulina Czarnota C21365726*

*Submission Date: 22/04/2022*

*Program Description: This C program consists of 4 teams of pharmaceutical company employees with their team membership details and line certifications.*

*This program is menu-driven and provides 4 following functions:*

*1. Sorts each team by surname using merge sort then combines the 4 teams of employees into one array. Displays each team separately. Sorts each team with merge sort. Merges the 4 teams into one array. Displays full list of employees.*

*2. Prints a list of employees certified for all 3 lines.*

*3. Searches employees by surname.*

*4. Exits the program.*

*The data in the program is structured with 4 arrays which represent 4 teams of 6 employees. Each employee contains details such as employee id, first name, surname and work line. A separate array contains the line certifications of each employee. The line certifications consist of employee id and line certification.*

*Data consists of:*

*Employee ID, First Name, Surname, Line*

*Team 1:*

*8, Ryuhei, Matsuda, 1*

*3, Hiroshi, Abe, 1*

*7, Toma, Ikuta, 3*

*1, Namie, Amuro, 2*

*4, Hikaru, Utada, 3*

*19, Maaya, Sakamoto, 2*

*Team 2:*

*14, Nanase, Aikawa, 1*

*23, Tatsuya, Fujiwara, 1*

*5, Hiroyuki, Sanada, 1*

*11, Yui, Asaka, 1*

*10, Saori, Atsumi, 2*

*6, Takuya, Kimura, 0*

*Team 3:*

*20, Kanata, Hongo, 1*

*2, Mika, Nakashima, 0*

*22, Yoko, Ono, 1*

*18, Kiyoshi, Atsumi, 1*

*24, Yusaku, Matsuda, 2*

*15, Kyoko, Fukada, 0*

*Team 4:*

*17, Miyoshi, Umeki, 3*

*16, Ryusei, Yokohama, 2*

*12, Jun, Matsumoto, 3*

*13, Kento, Yamazaki, 1*

*9, Yui, Aragaki, 1*

*21, Yuko, Aoki, 2*

*Certifications:*

*Employee ID, Earned Certification ID*

*Employees:*

*8, 3*

*3, 7*

*7, 4*

*1, 5*

*4, 5*

*19, 5*

*14, 3*

*23, 7*

*5, 0*

*11, 1*

*10, 4*

*6, 3*

*20, 2*

*2, 2*

*22, 5*

*18, 2*

*24, 7*

*15, 2*

*17, 2*

*16, 7*

*12, 4*

*13, 0*

*9, 6*

*21, 1*

*\*/*

*// The libraries*

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

*// Symbolic names*

#define Number\_Teams 4 *// Number of teams*

#define Number\_Team\_Members 6 *// Number of employees in each team*

#define Number\_Certs 24 *// Number of certifications*

*// Character and structure templates*

typedef **char** String[30];

typedef **struct** Employees\_Records

{

**int** employeeID;

    String firstName;

    String surname;

**int** workLine;

} Employees\_Records;

typedef **struct** Certifications

{

**int** employeeID;

**int** certID;

} Certifications;

*// The function signatures*

**void** EmployeeList(**struct** Employees\_Records \*employeeList, **int** size);

**void** SortEmployees(**struct** Employees\_Records \*employeeList, **int** start, **int** end);

**void** MergeEmployees(**struct** Employees\_Records \*employeeList, **int** start, **int** mid, **int** end);

**void** CombineTeams(**struct** Employees\_Records \*teams**[]**, **struct** Employees\_Records \*employeeList, **int** numberTeamMember, **int** numberTeam);

**int** SearchSurname(**struct** Employees\_Records \*employeeList, String key, **int** start, **int** end);

*// The option functions*

**void** ProcessTeams(**struct** Employees\_Records \*teams**[]**, **struct** Employees\_Records \*employeeList, **int** numberTeamMember, **int** numberTeam);

**void** FullCerts(**struct** Employees\_Records \*employeeList);

**void** GetEmployeeBySurname(**struct** Employees\_Records \*employeeList, **int** count);

*// The global varables: g\_*

*// Team 1 with 6 employees*

**struct** Employees\_Records g\_Team1[Number\_Team\_Members] =

{

    {8, "Ryuhei", "Matsuda", 1}, {3, "Hiroshi", "Abe", 1}, {7, "Toma", "Ikuta", 3}, {1, "Namie", "Amuro", 2}, {4, "Hikaru", "Utada", 3}, {19, "Maaya", "Sakamoto", 2}

};

*// Team 2 with 6 employees*

**struct** Employees\_Records g\_Team2[Number\_Team\_Members] =

{

    {14, "Nanase", "Aikawa", 1}, {23, "Tatsuya", "Fujiwara", 1}, {5, "Hiroyuki", "Sanada", 1}, {11, "Yui", "Asaka", 1}, {10, "Saori", "Atsumi", 2}, {6, "Takuya", "Kimura", 0}

};

*// Team 3 with 6 employees*

**struct** Employees\_Records g\_Team3[Number\_Team\_Members] =

{

    {20, "Kanata", "Hongo", 1}, {2, "Mika", "Nakashima", 0}, {22, "Yoko", "Ono", 1}, {18, "Kiyoshi", "Atsumi", 1}, {24, "Yusaku", "Matsuda", 2}, {15, "Kyoko", "Fukada", 0}

};

*// Team 4 with 6 employees*

**struct** Employees\_Records g\_Team4[Number\_Team\_Members] =

{

    {17, "Miyoshi", "Umeki", 3}, {16, "Ryusei", "Yokohama", 2}, {12, "Jun", "Matsumoto", 3}, {13, "Kento", "Yamazaki", 1}, {9, "Yui", "Aragaki", 1}, {21, "Yuko", "Aoki", 2}

};

*// The certifications*

**struct** Certifications g\_Certifications**[]** =

{

    {8, 3}, {3, 7}, {7, 4}, {1, 5}, {4, 5}, {19, 5}, {14, 3}, {23, 7}, {5, 0}, {11, 1}, {10, 4}, {6, 3}, {20, 2}, {2, 2}, {22, 5}, {18, 2}, {24, 7}, {15, 2}, {17, 2}, {16, 7}, {12, 4}, {13, 0}, {9, 6}, {21, 1}

};

**struct** Employees\_Records g\_TeamsCombined[Number\_Teams \* Number\_Team\_Members];

**int** main()

{

*// The main variables*

**int** userInput;

**int** sortSurname = 0;

**int** employeeCount = Number\_Team\_Members \* Number\_Teams;

*// The program name*

    printf("\n\n\t\t The Employee Certification Program \n");

*// Loop - it stops when user enters option 4*

    while (userInput != 4)

    {

*// The main menu*

        printf("\n \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

        printf("\n\n Main Menu \n\n");

        printf("(1) Sort the teams by surname and combine the teams \n");

        printf("(2) Display a list of the employees certified to work on all three lines \n");

        printf("(3) Search for a specific employee by surname \n");

        printf("(4) Exit program \n");

        printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");

        printf("\n Please select one of the above options from the menu \n");

        printf("\n > ");

        userInput = 0;

        scanf("%d", &userInput);

        while (getchar() != '\n');

*// Switch on user input*

        switch (userInput)

        {

            case 1:

            {

                printf("Sorting the teams by surname and then combining the teams...\n");

*// Array of teams*

                Employees\_Records \*teams[Number\_Teams] = {g\_Team1, g\_Team2, g\_Team3, g\_Team4};

*// Print, sort, combine the teams and then print combined teams*

                ProcessTeams(teams, g\_TeamsCombined, Number\_Team\_Members, Number\_Teams);

                sortSurname = 1;

                break;

            }

            case 2:

            {

*// Check if teams have been sorted, option 1 must be selected first*

                if (sortSurname == 0)

                {

                    printf("Please select option 1 first.\n");

                    break;

                }

                printf("List of employees with certification to work on all three lines...\n");

*// Print all employees with all certifications*

                FullCerts(g\_TeamsCombined);

                break;

            }

            case 3:

            {

*// Check if teams have been sorted, option 1 must be selected first*

                if(sortSurname == 0)

                {

                    printf("Please select option 1 first.\n");

                    break;

                }

                printf("Search for the employee's surname...\n\n");

                printf("ALERT! Please enter the surname with upper case! \n");

*// Search for employee by surname*

                GetEmployeeBySurname(g\_TeamsCombined, employeeCount);

                break;

            }

            case 4:

            {

*// Exit program*

                printf("\nExiting program...\n");

                break;

            }

            default:

            {

*// Invalid input*

                printf("\n\aInvalid input. Please try again.\n");

                break;

            }

        }

    }

}

*// Utility functions*

*// Prints details of each employees in an array*

*// Params: employeeList[] - list of employees to print*

*// count - number of employees in array*

**void** EmployeeList(**struct** Employees\_Records \*employeeList, **int** size)

{

*// Prints employee list*

    printf("%-11s  %-12s  %-12s  %-4s\n", "Employee ID", "First Name", "Surname", "Work Line");

    for (**int** i = 0; i < size; i++)

    {

        printf("%11d  %-12s  %-12s  %4d\n", employeeList[i].employeeID, employeeList[i].firstName, employeeList[i].surname, employeeList[i].workLine);

    }

    printf("\n");

}

*// Sorts employees in an array using modified merge sort*

*// Base case is reached when array size is less than 5 then insertion sort is used*

*// Params: employeeList[] - array of employees to sort*

*// start - starting index of array*

*// end - ending index of array*

**void** SortEmployees(**struct** Employees\_Records \*employeeList, **int** start, **int** end)

{

*// Base case when start and end are the same*

    if (start + 5 < end)

    {

*// Recursive case*

**int** mid = start + (end - start) / 2;

        SortEmployees(employeeList, start, mid);

        SortEmployees(employeeList, mid + 1, end);

        MergeEmployees(employeeList, start, mid, end);

    }

    else

    {

*// Base case*

*// Insertion sort*

        for (**int** i = start + 1; i <= end; i++)

        {

            Employees\_Records key = employeeList[i];

**int** j = i - 1;

            while (j >= start && strcmp(employeeList[j].surname, key.surname) > 0)

            {

                employeeList[j + 1] = employeeList[j];

                j--;

            }

            employeeList[j + 1] = key;

        }

    }

}

*// Merges two sorted subarray into one sorted array*

*// Params: employeeList[] - array of employees to sort*

*// start - starting index of first subarray*

*// mid - ending index of first subarray*

*// end - ending index of second subarray*

**void** MergeEmployees(**struct** Employees\_Records \*employeeList, **int** start, **int** mid, **int** end)

{

*// Create two temp arrays*

**int** nL = mid - start + 1;

**int** nR = end - mid;

    Employees\_Records \*tempL = (Employees\_Records \*)malloc(sizeof(Employees\_Records) \* nL);

    Employees\_Records \*tempR = (Employees\_Records \*)malloc(sizeof(Employees\_Records) \* nR);

*// Copy elements from original array to temp arrays*

    for (**int** i = 0; i < nL; i++)

    {

        tempL[i] = employeeList[start + i];

    }

    for (**int** i = 0; i < nR; i++)

    {

        tempR[i] = employeeList[mid + 1 + i];

    }

*// Merge temp arrays into original array*

**int** i = start, j = 0, k = 0;

    while (j < nL && k < nR)

    {

        if (strcmp(tempL[j].surname, tempR[k].surname) < 0)

        {

            employeeList[i++] = tempL[j++];

        }

        else

        {

            employeeList[i++] = tempR[k++];

        }

    }

*// Copy remaining elements from temp arrays*

    while (j < nL)

    {

        employeeList[i++] = tempL[j++];

    }

    while (k < nR)

    {

        employeeList[i++] = tempR[k++];

    }

}

*// Combine sorted teams into one sorted array*

*// Params: \*teams[] - array of teams to combine*

*// employeeList[] - array of employees*

*// teamMemberCount - number of members in each team*

*// teamCount - number of teams*

**void** CombineTeams(**struct** Employees\_Records \*teams**[]**, **struct** Employees\_Records \*employeeList, **int** teamMemberCount, **int** teamCount)

{

**int** combinedSize = teamCount \* teamMemberCount;

*// Concatenate teams into one array*

    for (**int** i = 0; i < teamCount; i++)

    {

        for (**int** j = 0; j < teamMemberCount; j++)

        {

            employeeList[i \* teamMemberCount + j] = teams[i][j];

        }

    }

*// Merge employeeList*

    for (**int** i = 0; i < teamCount - 1; i++)

    {

        MergeEmployees(employeeList, 0, (i + 1) \* teamMemberCount - 1, (i + 2) \* teamMemberCount - 1);

    }

}

*// Search employee list by surname*

*// Params: employeeList[] - array of employees to search*

*// key - surname to search for*

*// start - starting index of array*

*// end - ending index of array*

*// Return: index of employee if found, -1 if not found*

**int** SearchSurname(**struct** Employees\_Records \*employeeList, String key, **int** start, **int** end)

{

    if (end >= start)

    {

**int** mid = start + (end - start) / 2;

        if (strcmp(employeeList[mid].surname, key) == 0) return mid;

        if (strcmp(employeeList[mid].surname, key) > 0)

        {

            return SearchSurname(employeeList, key, start, mid - 1);

        }

        else

        {

            return SearchSurname(employeeList, key, mid + 1, end);

        }

    }

    return -1;

}

*// The option functions*

**void** ProcessTeams(**struct** Employees\_Records \*teams**[]**, **struct** Employees\_Records \*employeeList, **int** teamMemberCount, **int** teamCount)

{

*// Displays each team separately*

    printf("\nDisplaying teams...\n");

    for (**int** i = 0; i < teamCount; i++)

    {

        printf("Team %d:\n", i + 1);

        EmployeeList(teams[i], teamMemberCount);

    }

*// Sorts each team using merge sort*

    printf("\nSorting teams...\n");

    for (**int** i = 0; i < teamCount; i++)

    {

        SortEmployees(teams[i], 0, teamMemberCount - 1);

    }

*// Displays each team separately*

    for (**int** i = 0; i < teamCount; i++)

    {

        printf("Team %d:\n", i + 1);

        EmployeeList(teams[i], teamMemberCount);

    }

*// Combines teams to employeeList and displays*

    printf("\nCombining teams...\n");

    CombineTeams(teams, employeeList, teamMemberCount, teamCount);

    printf("\nCombined teams list:\n");

    EmployeeList(employeeList, teamCount \* teamMemberCount);

}

*// Displays the list of the employees certified to work on all three lines*

**void** FullCerts(**struct** Employees\_Records \*employeeList)

{

*// Searches for full certifications*

    printf("\nSearching for employees with full certifications...\n");

    for (**int** i = 0; i < Number\_Certs; i++)

    {

*// For each full cert, it finds the employee linked to it and prints their details*

        if ((employeeList + i) -> workLine == 3)

        {

              printf("\n%d. { %d , %s , %s , %d }", i , (employeeList + i) -> employeeID, (employeeList + i)->surname , (employeeList + i) -> firstName, (employeeList + i) -> workLine);

        }

    }

   printf("\n");

}

*// Searches for a specific employee by surname*

**void** GetEmployeeBySurname(**struct** Employees\_Records \*employeeList, **int** count)

{

*// Prompt user for surname*

    String surname;

**int** index;

    do

    {

        printf("\nEnter surname (max 29): \n\n> ");

        scanf("%29s", surname);

        if (strcmp(surname, "") == 0)

        {

            printf("\nSurname cannot be blank.\n");

        }

    } while (strcmp(surname, "") == 0);

    index = SearchSurname(employeeList, surname, 0 , count - 1);

    if (index == -1)

    {

        printf("\nNo employees found with the surname '%s'.\n", surname);

    }

    else

    {

        printf("\n\n EMPLOYEE FOUND \n\n");

        printf("------------------ \n\n");

        printf("Employee ID:  %d\n\n", employeeList[index].employeeID);

        printf("First Name:  %s\n\n", employeeList[index].firstName);

        printf("Surname:  %s\n\n", employeeList[index].surname);

        printf("Work Line:  %d\n\n", employeeList[index].workLine);

    }

    printf("\n");

}

# HOW REQUIREMENTS WERE MET

* ***How did you use merge sort:***

Merge sort is a sorting algorithm based on the divide-and-conquer paradigm. It is an useful algorithm for sorting linked lists. Merge sort is a stable sort which means that the same element in an array maintain their original positions concerning each other.

I used merge sort in the ‘SortEmployees’, ‘MergeEmployees’ and ‘CombineTeams’ functions in my c program.

The ‘SortEmployees’ function sorts employees in an array using a modified merge sort. The base case is reached when the array size is less than 5. In this merge sort, I divided the array recursively into two halves, until each sub-array contains a single element, and then I had to merge the sub-array in a way that it results into a sorted array.

The ‘MergeEmployees' function merges two sorted sub-arrays into one, wherein it assumes that the first temp array[mid – start + 1 ] and second temp array[end - mid] are sorted. It then copies elements from the original array to temp arrays, then merges temp arrays into the original array and finally it copies the remaining elements from temp arrays.

In the ‘CombineTeams’ function the sorted teams are combined into one sorted array. Then the teams are concatenated into one array.

The overall time complexity of the merge sort is O(nLogn). It is more efficient as it is in the worst case also the runtime is O(nlogn).

* ***How did you decrease the number of leaves in the execution tree of merge sort using another sorting algorithm:***

I decreased the number of leaves in the execution three of merge sort by using the insertion sort of the base case in the ‘SortEmployees’ function.

The merge sort is an external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting. Therefore by using the insertion sort in the program, one element from the input elements is consumed in each iteration to find its correct position i.e., the position to which it belongs in the sorted array.

* ***How did you increase the efficiency of that sorting algorithm previously mentioned:***

Insertion sort has a fast best-case running time and is a good sorting algorithm to use when the input list is mostly sorted by the merge sort. n this case insertion sort has a linear running time which is O(n). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. Therefore, in this case, the binary insertion sort is used as it improves the efficiency when the array has a low number of elements thanks to the merge sort.

While insertion sort’s worst-case complexity is O(n^2), it is the most efficient one because it’s adaptive for data sets that are already substantially sorted making it have a possible time complexity of O(n + d), where d is the number of inversions. This makes it more efficient than sorts because it gives the algorithm a potential best case of O(n).

* ***The time complexity of each algorithm:***

The time complexity for employees certified to work on all lines (b) algorithm is O(n). O(n) represents the complexity of the ‘FullCerts’ function that increases linearly and in direct proportion to the number of inputs.

The time complexity for the employee's surname search (c) algorithm is O(1). O(1) represents the ‘GetEmployeeBySurname’ function that always takes the same take regardless of input size.