C21348423 Adrian Capacite

Assignment report

Algorithm and Design and Problem

Contents

[Deliverables 2](#_Toc101536721)

[Data structure 2](#_Toc101536722)

[a) Flowchart of combine and sort process 2](#_Toc101536723)

[Function: ProcessTeams(): 2](#_Toc101536724)

[Function: SortEmployees(): 4](#_Toc101536725)

[Function: CombineTeams(): 5](#_Toc101536726)

[b) Pseudocode for employees certified to work on all lines 6](#_Toc101536727)

[c) Pseudocode for searching by surnames 7](#_Toc101536728)

[Code for each flowchart and pseudocode above in C 8](#_Toc101536729)

[a) Combine and sort process code 8](#_Toc101536730)

[b) Certified to work on all lines code 12](#_Toc101536731)

[c) Searching by surnames code 13](#_Toc101536732)

[Source code 14](#_Toc101536733)

# Deliverables

## Data structure

Employee:

* employeeID
* firstName
* surname
* line

Certification

* employeeID
* earnedCertID

## a) Flowchart of combine and sort process

Time complexity analysis:

O(N Log N)

### Function: ProcessTeams():

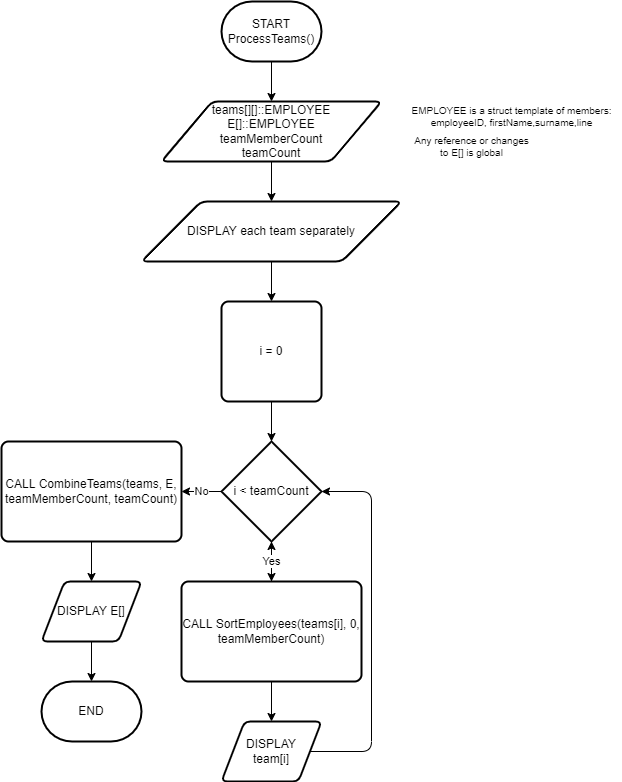
Time Complexity: O(n log n) + O(n) -> n

Merge sort: O(n log n)

Combine teams: O(n)

There were two efficiency requirements to meet:  
The first requirements were to improve the efficiency of the merge sort by reducing the leaves within the call stack. This is done by stopping the recursive splitting of the array when the sub array size is less than 5. When the array size reaches 5, a modified insertion sort is used to sort the array. Then the algorithm merges the sub arrays back up to the full array.

The second requirement were to improve the efficiency of the elementary sort which was used to improve the merge sort. The elementary sort we used was insertion sort and in insertion sort, it searches for a spot in the sorted array to insert the key value to be placed where it is smaller than the value above it. This is done using insertion sort where it searches for the spot where the value is supposed to be and returns the index of it. Then the indexes higher than the index the key is supposed to be is shifted up to make room to insert the key.

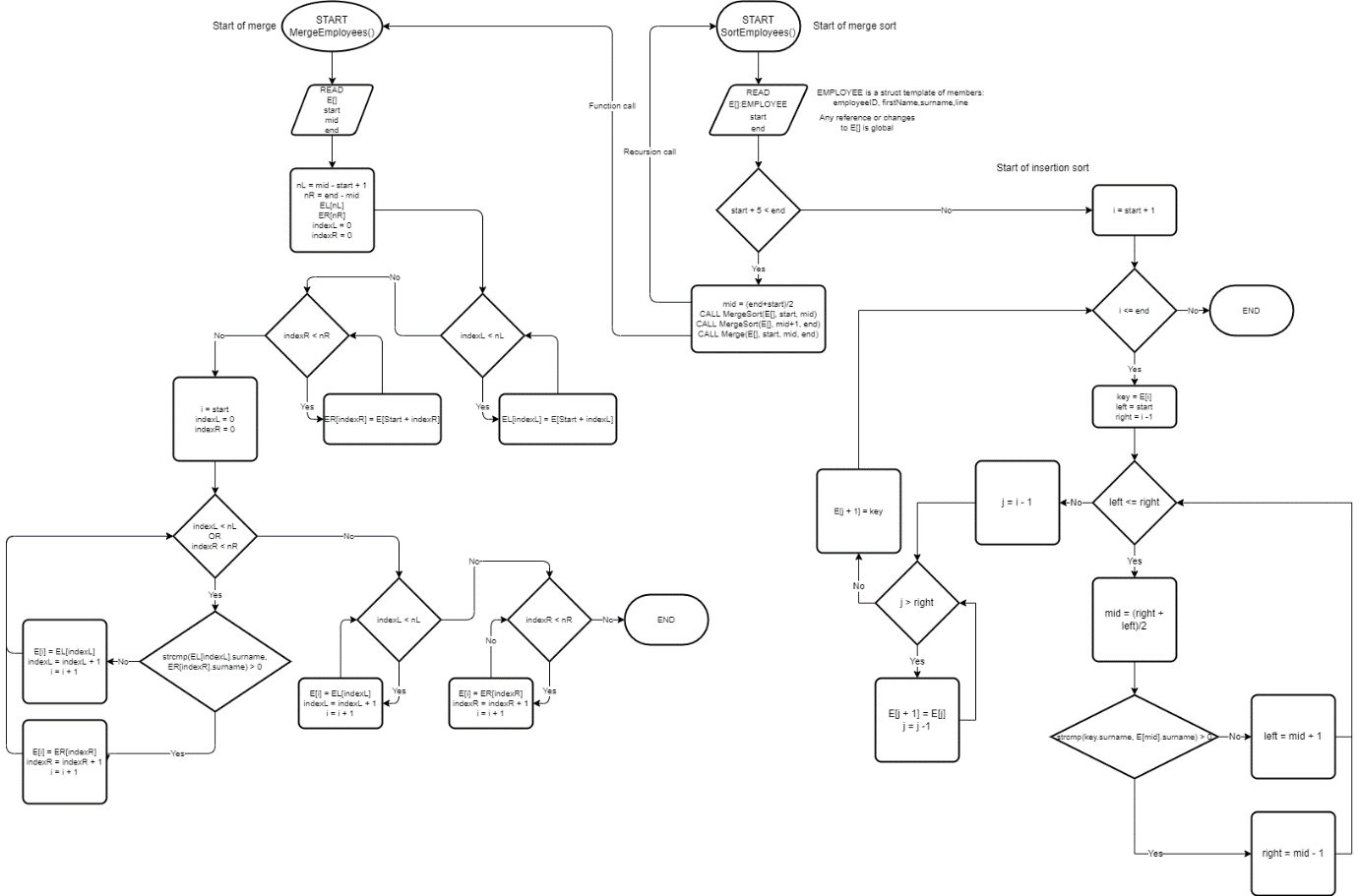


### Function: SortEmployees():

Time Complexity: O(Log n) x O(n) -> O(N Log N)

At the divide part: O(Log n)

At conquer part nested within divide: O(n)



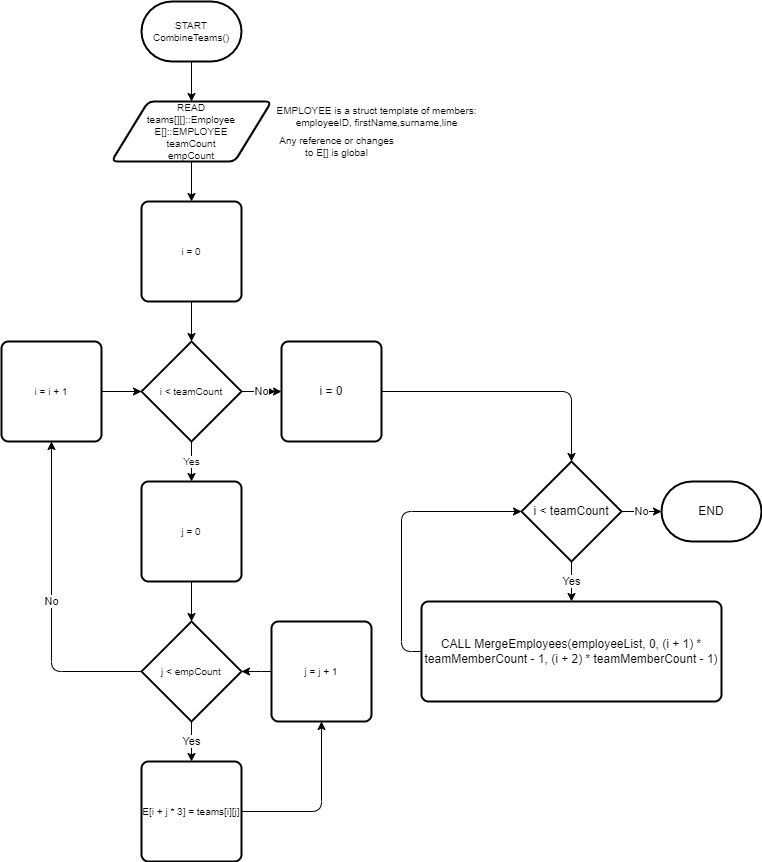
### Function: CombineTeams():

Time complexity: O(n)

There is two for loops however, no nested for loops

MergeEmployees() function is O(n) however the loop it is contained in does not fully nest the whole data.

Merging the employees is done by individually adding each team to the full list of employees while keeping the list sorted by surnames.



## b) Pseudocode for employees certified to work on all lines

Time complexity: O(n^2)

Depth of 2 nested for loop gives us O(n^2). First loop loops through all certs, second loops through each employee.

To find an employee the function uses linear search to find a certification for all 3 lines, if it encounters a certification for all 3 lines then it uses linear search to find the associated employeeID to the certification in the employee list.

// Struct Templates:

STRUCT EMPLOYEE{employeeID, firstName, surname, line}

STRUCT CERTIFICATION{employeeID, certificationID}

// Searches certifications for full certs where earnedCertID has a value of 7 and prints the employee's name

// Params: employeeList[]::Struct EMPLOYEE{employeeeID, firstName, surname, line} - array of employees

//         certList[]::Struct CERTIFICATION{employeeID, certificationID} - array of certifications to search

//         employeeCount - number of employees

//         certCount - number of certifications

FUNCTION PrintFullCerts(employeeList[] AS STRUCT Employees, certList[] AS STRUCT Certifications, employeeCount, certCount)

    // Search for full certs

    PRINT "Employee ID, First Name, Surname, Line" // Row headers

    FOR i FROM 0 TO certCount

        IF certList[i].certificationID EQUALS 7

            // Search for employee with matching employeeID and print details

            FOR j FROM 0 TO employeeCount

                IF employeeList[j].employeeID EQUALS certList[i].employeeID

                    PRINT employee details at index[]

                    BREAK

                ELSE IF j EQUALS employeeCount - 1

                        PRINT "N/A"

                END IF

            END FOR

        END IF

    END FOR

END FUNCTION

## c) Pseudocode for searching by surnames

Time complexity: O(Log N)

Binary search splits array in half to find the solution which gives us O(Log N)

The search employee by surname algorithm is a binary search algorithm which searches an array sorted by surname, the algorithm returns the index of where the employee is which then can be used to print the employee, otherwise it tells the user there is no such employee.

// Struct Templates

STRUCT EMPLOYEE{employeeID, firstName, surname, line}

// Prompts user for surname to search then searches employee list for employees with that surname using binary search and prints the employee's details

// Params: employeeList[]::Struct EMPLOYEE{employeeeID, firstName, surname, line} - array of employees

//         employeeCount - number of employees

FUNCTION GetEmployeeBySurname(employeeList[], count)

    DECLARE surname

    DECLARE index

    // Prompt user for surname

    DO

        PROMPT "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 SearchEmployeeBySurname(employeeList, count, surname)

    IF index EQUALS -1

        PRINT "Employee not found"

    ELSE

        PRINT "Employee found: "

        PRINT Employee details at employeeList[index]

END IF

END

// Uses binary search to find the person

// Employee array must be sorted by surname

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

//         key - surname to search for

//         start - start index of array

//         end - end index of array

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

FUNCTION SearchBySurname(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 SearchBySurname(employeeList, key, start, mid - 1)

        ELSE

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

        END IF

    END IF

    // Base case

    RETURN -1

END FUNCTION

## Code for each flowchart and pseudocode above in C

### a) Combine and sort process code

function: ProcessTeams()

// Option 1: a) Sorts and combines teams

// Prints individual teams then sorts the teams individually

// Then combines the teams into one sorted array

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

//         employeeList[] - array of employees which will contain the combined sorted teams

//         teamMemberCount - number of members in each team

//         teamCount - number of teams

void ProcessTeams(EMPLOYEE \*teams[], EMPLOYEE \*employeeList, int teamMemberCount, int teamCount)

{

    // Display each team separately

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

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

    {

        printf("Team %d:\n", i + 1);

        PrintEmployeeList(teams[i], teamMemberCount);

    }

    PromptContinue();

    PrintSoftDivider();

    // Sort each team using merge sort

    printf("\nSorting teams...\n");

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

    {

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

    }

    // Display each team separately

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

    {

        printf("Team %d:\n", i + 1);

        PrintEmployeeList(teams[i], teamMemberCount);

    }

    PromptContinue();

    PrintSoftDivider();

    // Combine teams to employeeList and display

    printf("\nCombining teams...\n");

    CombineTeams(teams, employeeList, teamMemberCount, teamCount);

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

    PrintEmployeeList(employeeList, teamCount \* teamMemberCount);

}

Function: SortEmployees()

// Modified merge sort with insertion srot sorts employees in a array by surname

// 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(EMPLOYEE \*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++)

        {

                        EMPLOYEE key = employeeList[i];

            // Binary search can be used here to find the correct index for insertion to reduce comparisons

            // incremental binary search

            int left = start;

            int right = i - 1;

            int j;

            while (left <= right)

            {

                int mid = left + (right - left) / 2;

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

                {

                    right = mid - 1;

                }

                else

                {

                    left = mid + 1;

                }

            }

            // Shift elements to the right to make room for insertion

            for (j = i - 1; j > right; j--)

            {

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

            }

            employeeList[j + 1] = key;

        }

    }

}

Function: MergeEmployees()

// 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(EMPLOYEE \*employeeList, int start, int mid, int end)

{

    // Create two temp arrays

    int nL = mid - start + 1;

    int nR = end - mid;

    EMPLOYEE \*tempL = (EMPLOYEE \*)malloc(sizeof(EMPLOYEE) \* nL);

    EMPLOYEE \*tempR = (EMPLOYEE \*)malloc(sizeof(EMPLOYEE) \* 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++];

    }

    // Free temp arrays

    free(tempL);

    free(tempR);

}

Function: CombineEmployees()

// 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(EMPLOYEE \*teams[], EMPLOYEE \*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);

    }

}

### b) Certified to work on all lines code

Function: PrintFullCerts()

// Option 2: b) Searches certifications for full certs where earnedCertID has a value of 7 and prints the employee's name

// Params: employeeList[] - array of employees

//         certList[] - array of certifications to search

//         employeeCount - number of employees

//         certCount - number of certifications

void PrintFullCerts(EMPLOYEE employeeList[], CERTIFICATION certList[], int employeeCount, int certCount)

{

    // Search for full certs

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

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

    for (int i = 0; i < certCount; i++)

    {

        // For each full cert, find the employee linked to it and print their details

        if (certList[i].earnedCertID == 7)

        {

            for (int j = 0; j < employeeCount; j++)

            {

                if (employeeList[j].employeeID == certList[i].employeeID)

                {

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

                    break;

                }

                else if (j == employeeCount - 1)

                {

                    printf("%11d  %-12s  %-12s\n", certList[i].employeeID, "Unknown", "Unknown");

                }

            }

        }

    }

    printf("\n");

}

### c) Searching by surnames code

Function: getEmployeeBySurname()

// Option 3: c) Prompts user for surname to search then searches employee list for employees with that surname using binary search and prints the employee's details

// Params: employeeList[] - array of employees

//         employeeCount - number of employees

void GetEmployeeBySurname(EMPLOYEE employeeList[], int count)

{

    // Prompt user for surname

    STRING30 surname;

    int index;

    do

    {

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

        scanf("%29s", surname);

        if (strcmp(surname, "") == 0)

        {

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

        }

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

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

    if (index == -1)

    {

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

    }

    else

    {

        printf("\nEmployee found:\n");

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

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

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

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

    }

    printf("\n");

}

Function: SearchBySurname()

// 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

// Returns: index of employee if found, -1 if not found

int SearchBySurname(EMPLOYEE \*employeeList, STRING30 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 SearchBySurname(employeeList, key, start, mid - 1);

        }

        else

        {

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

        }

    }

    return -1;

}

# Source code

/\*

    Algorithms and Design and Problems Assignment - assignment

    Program contains details of 4 teams of employees and their details and line certifications

    Data structure:

        4 arrays represent 4 teams of 4 employees::

            Each employee contains details: employee id, first name, surname and line

        A separate array contains the line certifications of each employee

            The line certifications consists of: employee id, line certification

                The line certification has 3 bits:

                    Bit 0 (+1): Line 1

                    Bit 1 (+2): Line 2

                    Bit 2 (+4): Line 3

    Functions

        (a). Sorts each team by surname using merge sort then combines the 4 teams of employees into one array

            Display each team separately

            Sort each team with merge sort

            Merge the 4 teams into one array

            Display full list of employees

        (b). Prints a list of employees certified for all 3 lines

            Create list of full certs (earnedCertID = 7)

        (c). Search employees by surname

    Data:

        Teams:

            Cols:

                employeeID,firstName,surname,line

            Team 1:

                8,Hanae,Mejia,1

                3,Evan,Underwood,1

                7,Jenette,David,3

                1,Quin,Knight,2

                4,Gannon,Mueller,3

                19,Merrill,Gilmore,2

            Team 2:

                14,Shellie,Soto,1

                23,Carson,Ayala,1

                5,Orla,Wyatt,1

                11,Neville,Berg,1

                10,Macy,Cotton,2

                6,Emi,Stafford,0

            Team 3:

                20,Austin,William,1

                2,Dana,Stephenson,0

                22,Amery,Bridges,1

                18,Mollie,Hester,1

                24,Brent,Molina,2

                15,Bradley,Ortiz,0

            Team 4:

                17,Maia,Mccullough,3

                16,India,Calhoun,2

                12,Nathaniel,Solis,3

                13,Drake,Leach,1

                9,May,Bowers,1

                21,Maxine,Marquez,2

        Certifications:

            Cols:

                employeeID,earnedCertID

            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

    Author: Adrian Thomas Capacite

    Date: 13 / 04 / 2022

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define TEAM\_MEMBER\_COUNT 6     // Number of employees in each team

#define TEAM\_COUNT 4    // Number of teams

#define CERT\_COUNT 24   // Number of certs

#define PrintCentered(f, str, width) printf("%\*s" f, (int)((width - strlen(str)) / 2), "", str)

#define PrintDivider() printf("\n================================================================\n")

#define PrintSoftDivider() printf("\n----------------------------------------------------------------\n")

#define PromptContinue() printf("Press enter to continue..."); while(getchar() != '\n'); // Prompt user to continue

/\* Struct templates \*/

typedef char STRING30[30];

typedef struct EMPLOYEE

{

    int employeeID;

    STRING30 firstName;

    STRING30 surname;

    int line;

} EMPLOYEE;

typedef struct CERTIFICATION

{

    int employeeID;

    int earnedCertID;

} CERTIFICATION;

/\* Function prototypes \*/

// Utility functions

void PrintEmployeeList(EMPLOYEE [], int);

void SortEmployees(EMPLOYEE [], int, int);

void MergeEmployees(EMPLOYEE [], int, int, int);

void CombineTeams(EMPLOYEE \*[], EMPLOYEE [], int, int);

int SearchBySurname(EMPLOYEE [], STRING30, int, int);

// Option functions

void ProcessTeams(EMPLOYEE \*[], EMPLOYEE [], int, int);

void PrintFullCerts(EMPLOYEE [], CERTIFICATION [], int, int);

void GetEmployeeBySurname(EMPLOYEE [], int);

/\* Global varables prefix: g\_ \*/

// 4 Teams of 6 employees

EMPLOYEE g\_team1[TEAM\_MEMBER\_COUNT] = {

    {8, "Hanae", "Mejia", 1},

    {3, "Evan", "Underwood", 1},

    {7, "Jenette", "David", 3},

    {1, "Quin", "Knight", 2},

    {4, "Gannon", "Mueller", 3},

    {19, "Merrill", "Gilmore", 2}

};

EMPLOYEE g\_team2[TEAM\_MEMBER\_COUNT] = {

    {14, "Shellie", "Soto", 1},

    {23, "Carson", "Ayala", 1},

    {5, "Orla", "Wyatt", 1},

    {11, "Neville", "Berg", 1},

    {10, "Macy", "Cotton", 2},

    {6, "Emi", "Stafford", 0}

};

EMPLOYEE g\_team3[TEAM\_MEMBER\_COUNT] = {

    {20, "Austin", "William", 1},

    {2, "Dana", "Stephenson", 0},

    {22, "Amery", "Bridges", 1},

    {18, "Mollie", "Hester", 1},

    {24, "Brent", "Molina", 2},

    {15, "Bradley", "Ortiz", 0}

};

EMPLOYEE g\_team4[TEAM\_MEMBER\_COUNT] = {

    {17, "Maia", "Mccullough", 3},

    {16, "India", "Calhoun", 2},

    {12, "Nathaniel", "Solis", 3},

    {13, "Drake", "Leach", 1},

    {9, "May", "Bowers", 1},

    {21, "Maxine", "Marquez", 2}

};

// Certifications

CERTIFICATION g\_certifications[CERT\_COUNT] = {

    {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}

};

EMPLOYEE g\_teamsCombined[TEAM\_COUNT \* TEAM\_MEMBER\_COUNT];

int main()

{

    int userInput;

    int sortBySurname = 0;

    int employeeCount = TEAM\_MEMBER\_COUNT \* TEAM\_COUNT;

    // Program title

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

    PrintCentered("%s\n", "---- Employee Certification Program ----", 64);

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

    // Program loop - stops when user enters 4

    while (userInput != 4)

    {

        // Print menu prompt

        printf("\n");

        PrintDivider();

        PrintCentered("%s\n\n", "---- Main Menu ----", 64);

        printf(

            "(1) Sort Teams by Surname then combine teams\n"

            "(2) Print employees certified for all lines\n"

            "(3) Search for employee by surname\n"

            "(4) Exit\n"

            "> "

        );

        userInput = 0;

        scanf("%d", &userInput);

        while(getchar() != '\n');

        // Switch on user input

        switch (userInput)

        {

            case 1: // Part a)

            {

                printf("Running sort and combine routine...\n");

                PrintDivider();

                // Create array of teams

                EMPLOYEE \*teams[TEAM\_COUNT] = {g\_team1, g\_team2, g\_team3, g\_team4};

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

                ProcessTeams(teams, g\_teamsCombined, TEAM\_MEMBER\_COUNT, TEAM\_COUNT);

                sortBySurname = 1;

                PromptContinue();

                break;

            }

            case 2: // Part b)

            {

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

                if (sortBySurname == 0)

                {

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

                    PromptContinue();

                    break;

                }

                printf("Running full cert search routine...\n");

                PrintDivider();

                // Print all employees with all certifications

                PrintFullCerts(g\_teamsCombined, g\_certifications, employeeCount, CERT\_COUNT);

                PromptContinue();

                break;

            }

            case 3: // Part c)

            {

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

                if(sortBySurname == 0)

                {

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

                    PromptContinue();

                    break;

                }

                printf("Running search by surname routine...\n");

                PrintDivider();

                // Search for employee by surname

                GetEmployeeBySurname(g\_teamsCombined, employeeCount);

                PromptContinue();

                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 a array

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

//         count - number of employees in array

void PrintEmployeeList(EMPLOYEE employeeList[], int size)

{

    // Print employee list

    printf("%-11s  %-12s  %-12s  %-4s\n", "Employee ID", "First Name", "Surname", "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].line);

    }

    printf("\n");

}

// Modified merge sort with insertion srot sorts employees in a array by surname

// 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(EMPLOYEE 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++)

        {

            EMPLOYEE key = employeeList[i];

            // Binary search can be used here to find the correct index for insertion to reduce comparisons

            // incremental binary search

            int left = start;

            int right = i - 1;

            int j;

            while (left <= right)

            {

                int mid = left + (right - left) / 2;

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

                {

                    right = mid - 1;

                }

                else

                {

                    left = mid + 1;

                }

            }

            // Shift elements to the right to make room for insertion

            for (j = i - 1; j > right; j--)

            {

                employeeList[j + 1] = employeeList[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(EMPLOYEE employeeList[], int start, int mid, int end)

{

    // Create two temp arrays

    int nL = mid - start + 1;

    int nR = end - mid;

    EMPLOYEE \*tempL = (EMPLOYEE \*)malloc(sizeof(EMPLOYEE) \* nL);

    EMPLOYEE \*tempR = (EMPLOYEE \*)malloc(sizeof(EMPLOYEE) \* 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++];

    }

    // Free temp arrays

    free(tempL);

    free(tempR);

}

// 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(EMPLOYEE \*teams[], EMPLOYEE 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

// Returns: index of employee if found, -1 if not found

int SearchBySurname(EMPLOYEE employeeList[], STRING30 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 SearchBySurname(employeeList, key, start, mid - 1);

        }

        else

        {

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

        }

    }

    return -1;

}

// Option functions

// Option 1: a) Sorts and combines teams

// Prints individual teams then sorts the teams individually

// Then combines the teams into one sorted array

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

//         employeeList[] - array of employees which will contain the combined sorted teams

//         teamMemberCount - number of members in each team

//         teamCount - number of teams

void ProcessTeams(EMPLOYEE \*teams[], EMPLOYEE employeeList[], int teamMemberCount, int teamCount)

{

    // Display each team separately

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

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

    {

        printf("Team %d:\n", i + 1);

        PrintEmployeeList(teams[i], teamMemberCount);

    }

    PromptContinue();

    PrintSoftDivider();

    // Sort each team using merge sort

    printf("\nSorting teams...\n");

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

    {

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

    }

    // Display each team separately

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

    {

        printf("Team %d:\n", i + 1);

        PrintEmployeeList(teams[i], teamMemberCount);

    }

    PromptContinue();

    PrintSoftDivider();

    // Combine teams to employeeList and display

    printf("\nCombining teams...\n");

    CombineTeams(teams, employeeList, teamMemberCount, teamCount);

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

    PrintEmployeeList(employeeList, teamCount \* teamMemberCount);

}

// Option 2: b) Searches certifications for full certs where earnedCertID has a value of 7 and prints the employee's name

// Params: employeeList[] - array of employees

//         certList[] - array of certifications to search

//         employeeCount - number of employees

//         certCount - number of certifications

void PrintFullCerts(EMPLOYEE employeeList[], CERTIFICATION certList[], int employeeCount, int certCount)

{

    // Search for full certs

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

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

    for (int i = 0; i < certCount; i++)

    {

        // For each full cert, find the employee linked to it and print their details

        if (certList[i].earnedCertID == 7)

        {

            for (int j = 0; j < employeeCount; j++)

            {

                if (employeeList[j].employeeID == certList[i].employeeID)

                {

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

                    break;

                }

                else if (j == employeeCount - 1)

                {

                    printf("%11d  %-12s  %-12s\n", certList[i].employeeID, "Unknown", "Unknown");

                }

            }

        }

    }

    printf("\n");

}

// Option 3: c) Prompts user for surname to search then searches employee list for employees with that surname using binary search and prints the employee's details

// Params: employeeList[] - array of employees

//         employeeCount - number of employees

void GetEmployeeBySurname(EMPLOYEE employeeList[], int count)

{

    // Prompt user for surname

    STRING30 surname;

    int index;

    do

    {

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

        scanf("%29s", surname);

        if (strcmp(surname, "") == 0)

        {

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

        }

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

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

    if (index == -1)

    {

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

    }

    else

    {

        printf("\nEmployee found:\n");

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

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

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

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

    }

    printf("\n");

}