**Mini Project :**

|  |  |
| --- | --- |
| **Topic:** | Student attendance management system |
| **Group Details** | Roll\_No Name Sign  38 Noel Joe  31 Shawn Louis  28 Kunal Chaudhary |
| **Mapping With COs:** |  |
| **Objective**: |  |
| **Outcome:** | **CSL305.6-**Identify data structuring strategies that are appropriate to a given contextual problem and able to design, develop, test and debug in C language considering appropriate algorithm. |
| **Instructions:** | 1. Divide and Conquer ( Decompose the problem in modules) 2. Knowledge and understanding    1. **Identifying the best suited data structure for solving the sub problems with justification**    2. **Define algorithms for various identified functions**. 3. Thinking and Inquiry    1. **Implement the modules**    2. **Test cases to test correctness of the solution** |
| **Deliverlables:** | **Printouts submission:**   1. **hard copy of Report** 2. **softcopy of Report and PPT** 3. Students are required to run this program in C language on linux terminal 4. Take a screenshot of every output |
| **Conclusion:** | Justification of Choice of Data Structure |
| **References:** | DbitMoodle,geeksforgeeks,wikipedia |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DON BOSCO INSTITUTE OF TECHNOLOGY** | | | | |
| **Roll no: 31,28,38** | | | | |
| Project Title: | Attendance Management System | | | |
| Subject : | Data Structures | | | |
| Date : | 25-10-2019 | | | |
| Semester: | 3 | | | |
| Year: | 2nd Year | | | |
| **Grading Rubric for Mini Project** | | | | |
| **Parameters** | **0 point** | **3 point** | **5 point** | **Marks** |
| **Abstraction** | No Abstracti on | Problem Decomposed in Modules | A very cohesive divide and conquer strategy applied , well decomposed Modules |  |
|  | No clear | Appropriate data structure identified. | Appropriate data |  |
| **Knowledge and** | Understanding | structure identified |
| **Understanding** |  | with justification , |
|  |  | follows principle of |
|  |  | good software design |
| **Thinking and Inquiry** | No result presented  Poor naming of | Result  presented with no discussion. | Program divided modularly ( smaller and meaning full units) |  |
|  | functions and | Very few |  |
|  | variables | functions |  |
| **Quality of code/ script / program** | Code doesn’t compile, Syntax error. | Compile, crashes on running.  Doesn’t crash on  Erroneous input. | All functions implemented satisfactorily  Well demonstrated Program  Screen shoots of successful and unsuccessful output |  |
| **Viva** | Poor response | Moderately correct answers | Appropriate and satisfactory answers |  |
| **Total Marks(out of 25 )** | | | |  |
| **Signature of student** | | | |  |
| **Signature of faculty** | | | |  |

**DON BOSCO INSTITUTE OF TECHNOLOGY**

**DEPARTMENT OF COMPUTER ENGINEERING**

**CASE STUDY**

STUDENT

ATTENDANCE

MANAGEMENT

SYSTEM

Submitted by:-

NOEL JOE-38

KUNAL CHAUDHARY-28

SHAWN LOUIS-31

|  |  |
| --- | --- |
| **COURSE:** DATA STRUCTURE | **SEMESTER:**III |
| **COURSE CODE:** -CSC 303 | ASSESMENT:  Group PRESENTATION and Execution |
| **CASE STUDIES** | **ORAL:** 25 MARKS |
| **FACULTY INCHARGE:** MR. MIRZA ALI IMRAN | |

**Problem Statement:-**

To implement a student attendance management system.

Decomposition of the problem into modules:-

1. File management system
2. Sorting
3. Advanced Data Structures

Module 1.

File management System

In this module we look into the problem of data storing,that is storing data of the students.

We need to store this data,also we need to make changes and also delete the irrelevant data.

So by using the concept of file storage is best in this case.

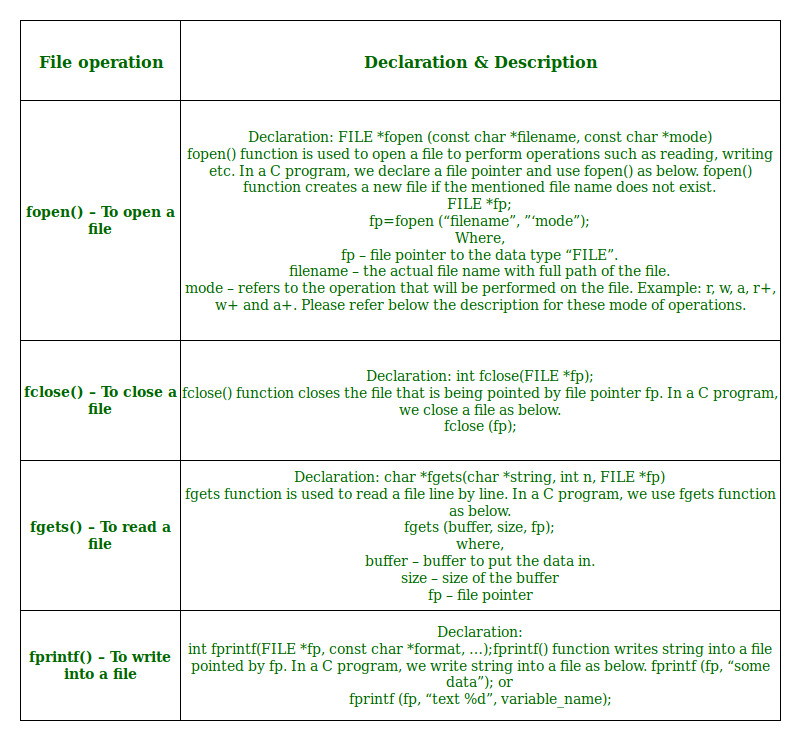
# File Handling in C

So far the operations using C program are done on a prompt / terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store the fetched information in a file. Different operations that can be performed on a file are:

1. Creation of a new file (**fopen with attributes as “a” or “a+” or “w” or “w++”)**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgetc**)
4. Writing to a file (**fprintf or fputs**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

The text in the brackets denotes the functions used for performing those operations.

**Functions in File**

**Operations:**

**Opening or creating file**  
For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

**File opening modes in C:**

* **“r” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
* **“w” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
* **“a” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
* **“r+” –** Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
* **“w+” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file.
* **“a+” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

As given above, if you want to perform operations on a binary file, then you have to append ‘b’ at the last. For example, instead of “w”, you have to use “wb”, instead of “a+” you have to use “a+b”. For performing the operations on the file, a special pointer called File pointer is used which is declared as

FILE \*filePointer;

So, the file can be opened as

filePointer = fopen(“fileName.txt”, “w”)

The second parameter can be changed to contain all the attributes listed in the above table.

* **Reading from a file –**  
  The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of printf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.

And the code snippet for reading a file is as:

FILE \* filePointer;

filePointer = fopen(“fileName.txt”, “r”);

fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);

* **Writing a file –**:

The file write operations can be perfomed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :

FILE \*filePointer ;

filePointer = fopen(“fileName.txt”, “w”);

fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);

* **Closing a file –**:  
  After every successful fie operations, you must always close a file. For closing a file, you have to use fclose function. The snippet for closing a file is given as :

FILE \*filePointer ;

filePointer= fopen(“fileName.txt”, “w”);

---------- Some file Operations -------

fclose(filePointer)

**CODE**:

-/\*File Declaration\*/

FILE \*fp, \*ft;

char another, choice;

char xfirst\_name[50], xlast\_name[50];

long int recsize;

fp=fopen("users.txt","rb+");

if (fp == NULL)

{

fp = fopen("users.txt","wb+");

if (fp==NULL)

{

puts("Cannot open file");

return 0;

}

}

recsize = sizeof(e);

/\*For adding Records\*/

system("cls");

cout << "Enter the First Name : ";

cin >> e.first\_name;

cout << "Enter the Last Name : ";

cin >> e.last\_name;

cout << "Enter the Course : ";

cin >> e.course;

cout << "Enter the Section : ";

cin >> e.section;

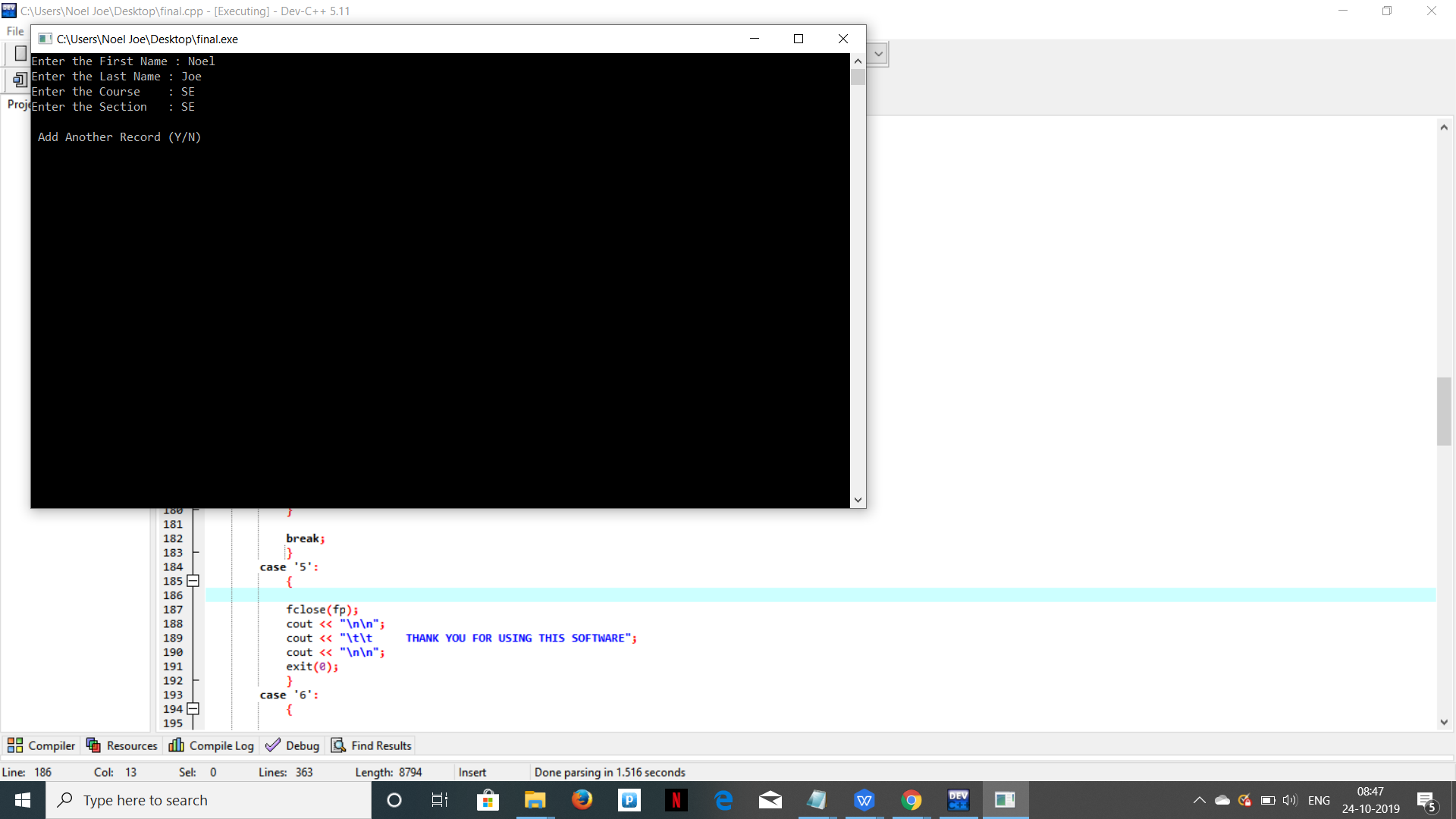
fwrite(&e,recsize,1,fp);

cout << "\n Add Another Record (Y/N) ";

fflush(stdin);

another = getchar();

**OUTPUT**



/\*For listing records\*/

while (fread(&e,recsize,1,fp) == 1)

{

cout << "\n";

cout <<"\n" << e.first\_name << setw(10) << e.last\_name;

cout << "\n";

cout <<"\n" <<e.course << setw(8) << e.section;

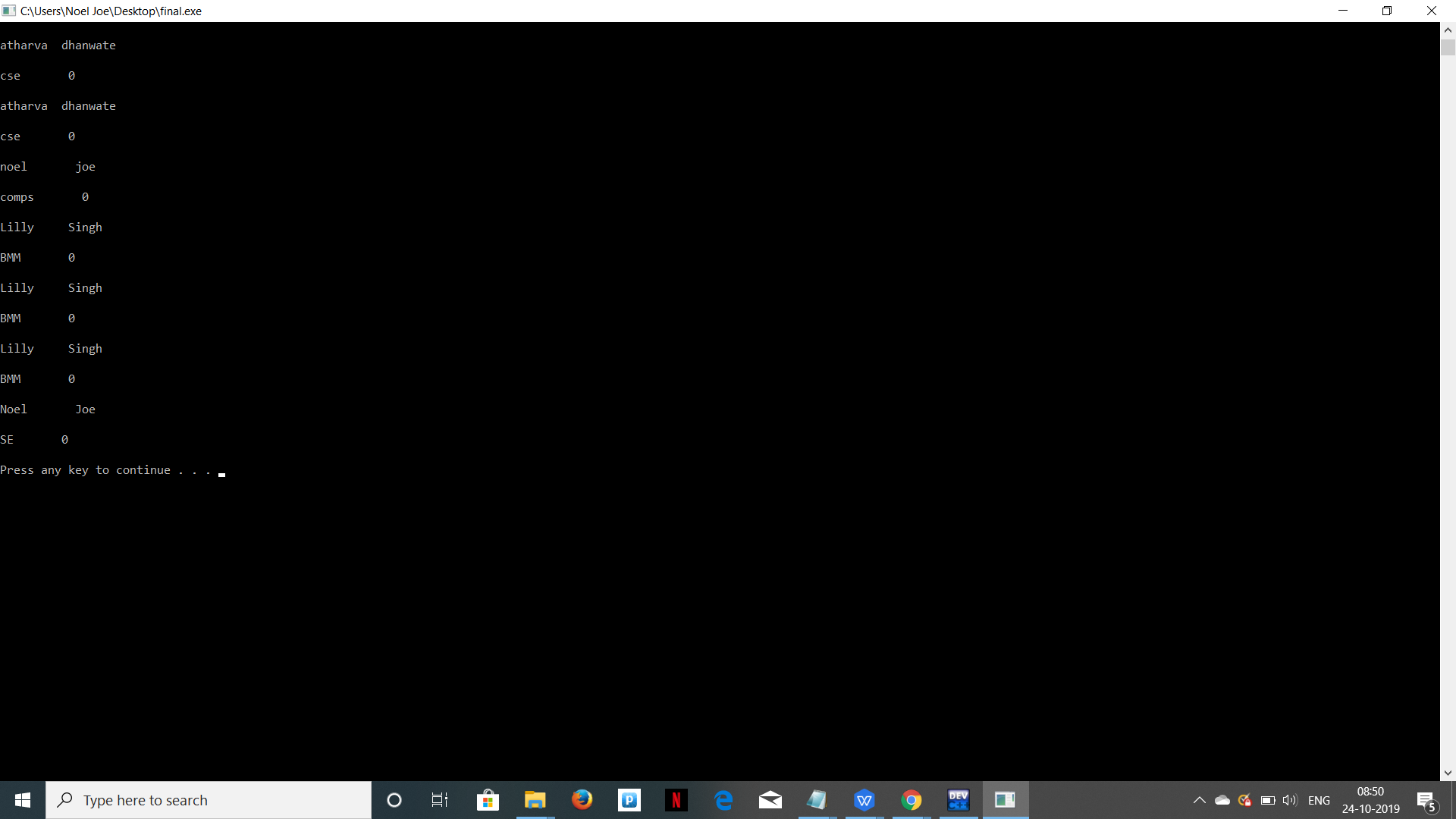
}

cout << "\n\n";

system("pause");

break;

**OUTPUT**



/\*For modifying records\*/

{

cout << "\n Enter the last name of the student : ";

cin >> xlast\_name;

rewind(fp);

while (fread(&e,recsize,1,fp) == 1)

{

if (strcmp(e.last\_name,xlast\_name) == 0)

{

cout << "Enter new the First Name : ";

cin >> e.first\_name;

cout << "Enter new the Last Name : ";

cin >> e.last\_name;

cout << "Enter new the Course : ";

cin >> e.course;

cout << "Enter new the Section : ";

cin >> e.section;

fseek(fp, - recsize, SEEK\_CUR);

fwrite(&e,recsize,1,fp);

break;

}

else

cout<<"record not found";

}

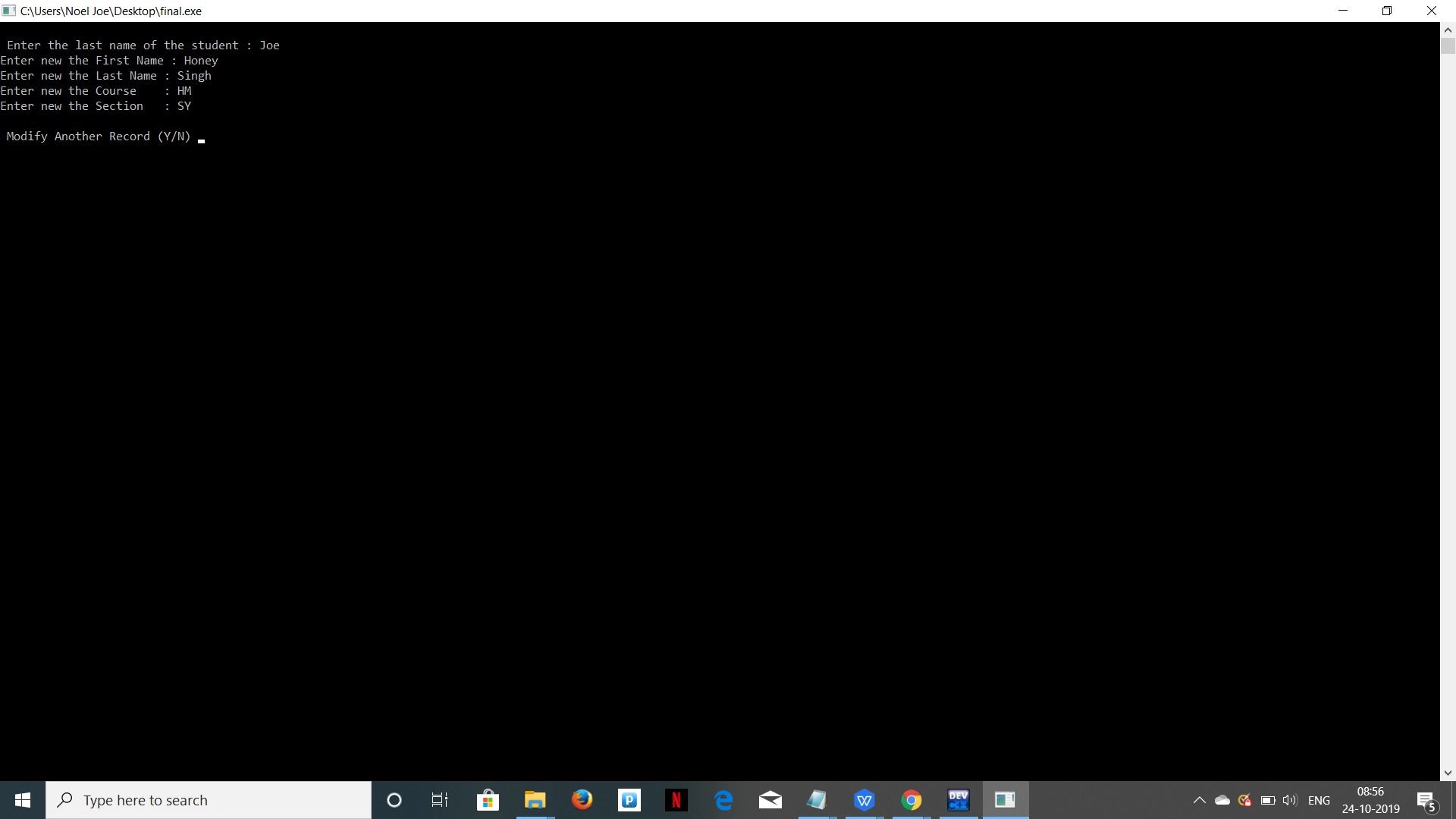
cout << "\n Modify Another Record (Y/N) ";

fflush(stdin);

another = getchar();

}

**OUTPUT**



/

\*For Deleting Records\*/

cout << "\n Enter the last name of the student to delete : ";

cin >> xlast\_name;

ft = fopen("temp.dat", "wb");

rewind(fp);

while (fread (&e, recsize,1,fp) == 1)

if (strcmp(e.last\_name,xlast\_name) != 0)

{

fwrite(&e,recsize,1,ft);

}

fclose(fp);

fclose(ft);

remove("users.txt");

rename("temp.dat","users.txt");

fp=fopen("users.txt","rb+");

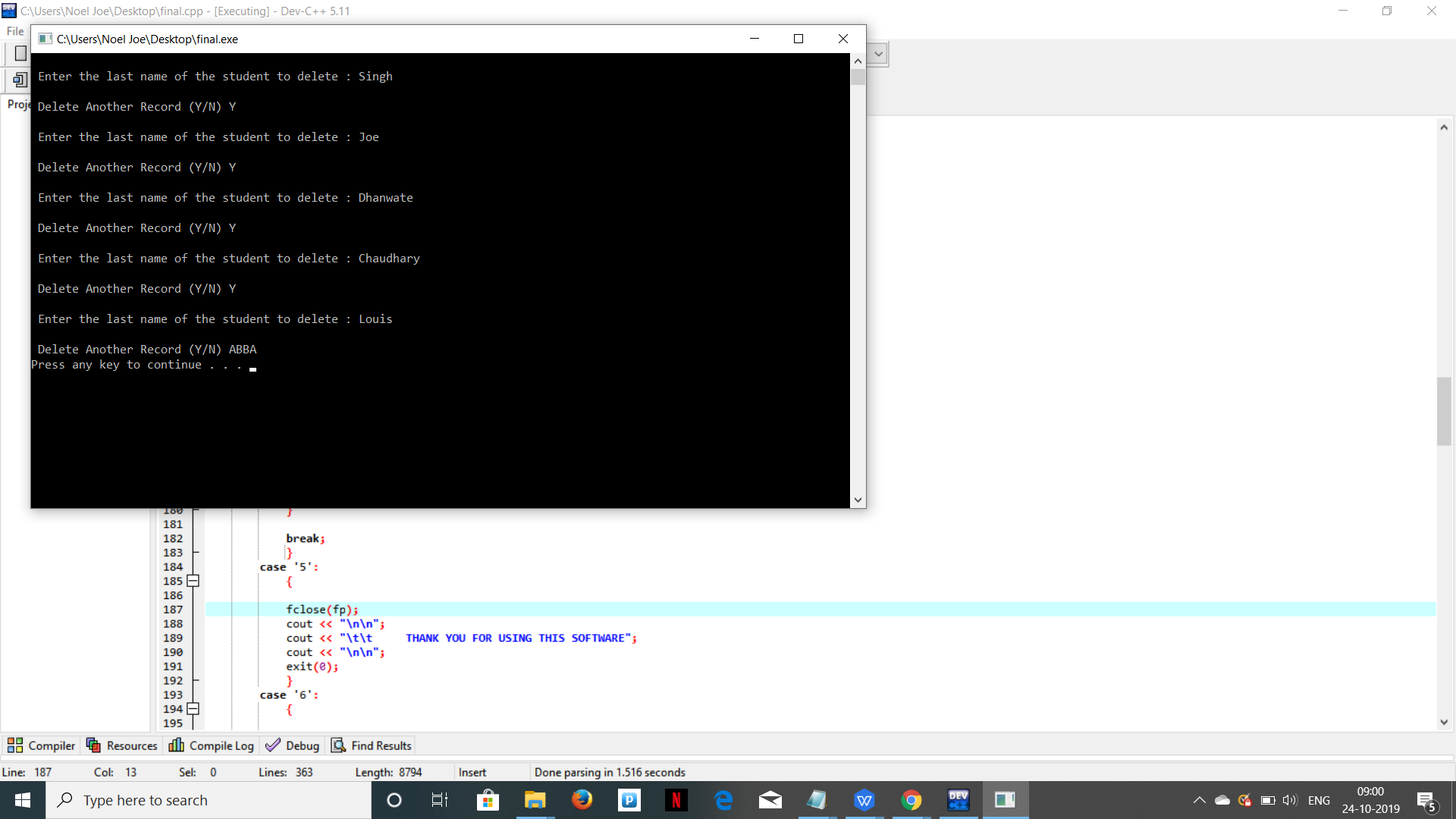
cout << "\n Delete Another Record (Y/N) ";

fflush(stdin);

another = getchar()

**OUTPUT**

;



Module 2:

Sorting

We had to include a sorting system in this project because after sorting it becomes very simple to search students or display lists of students.

Generally we use it to sort marks of the students.

In this project we have used Quick Sort

# QuickSort

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/" \t "https://www.geeksforgeeks.org/quick-sort/_blank), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



**Partition algorithm**

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element

for (j = low; j <= high- 1; j++)

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

}

**Analysis of QuickSort**  
Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + Rendered by QuickLaTeX.com(n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.  
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

**Worst Case:** The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + Rendered by QuickLaTeX.com(n)

which is equivalent to

T(n) = T(n-1) + Rendered by QuickLaTeX.com(n)

The solution of above recurrence is Rendered by QuickLaTeX.com(n2).

**Best Case:** The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

T(n) = 2T(n/2) + Rendered by QuickLaTeX.com(n)

The solution of above recurrence is Rendered by QuickLaTeX.com(nLogn). It can be solved using case 2 of [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem" \t "https://www.geeksforgeeks.org/quick-sort/_blank).

**Average Case:**  
To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/" \t "https://www.geeksforgeeks.org/quick-sort/_blank).  
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

T(n) = T(n/9) + T(9n/10) + Rendered by QuickLaTeX.com(n)

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/" \t "https://www.geeksforgeeks.org/quick-sort/_blank) and [Heap Sort](http://quiz.geeksforgeeks.org/heap-sort/" \t "https://www.geeksforgeeks.org/quick-sort/_blank), QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

**QuickSort is** [stable](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)**?**  
The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.  
 **QuickSort is**[In-place](https://www.geeksforgeeks.org/in-place-algorithm/)**?**  
As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the index.

CODE

void Quicksort(int \*marks,int start,int end)

{

if(start<end)

{

int P\_index=partition(marks,start,end);

Quicksort(marks,start,P\_index-1);

Quicksort(marks,P\_index+1,end);

}

}

int partition(int \*marks,int start,int end)

{

int pivot=marks[end];

int P\_index=start;

int i,t;

for(i=start;i<end;i++)

{

if(marks[i]<=pivot)

{

t=marks[i];

marks[i]=marks[P\_index];

marks[P\_index]=t;

P\_index++;

}

}

t=marks[end];

marks[end]=marks[P\_index];

marks[P\_index]=t;

return P\_index;

}

/\*Driver Code\*/

system("cls");

int a;

cout<<"Enter number of students: ";

cin>>a;

int marks[a];

cout<<"Enter the marks of student:\n";

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

{

cin>>marks[i];

}

Quicksort(marks,0,a-1);

{

cout<<("After quick sorting the array is\n");

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

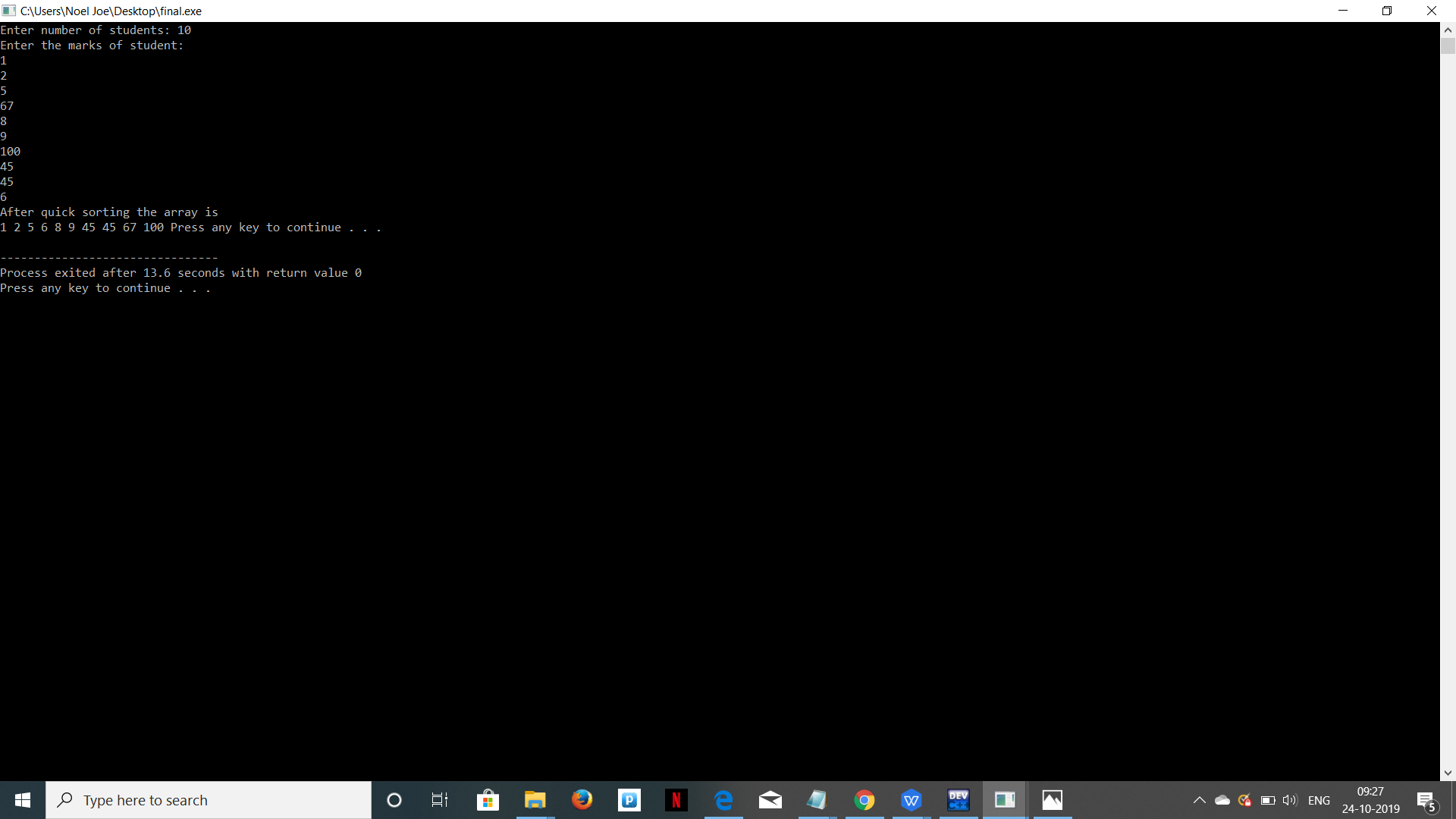
{

cout<<marks[i]<<" ";

}

}

**OUTPUT**



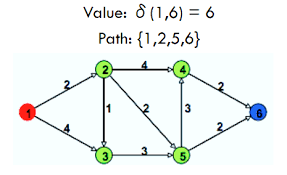
Module 3

Advanced Data structures

In this project we have decided to dive deep into Data Structures and Algorithms.

So we have implemented Priority Queue using Heaps.

And this is driven by the power of the Dijkstra’s shortest path Algorithm which is a greedy and very efficient algorithm.



|  |
| --- |
| **Dijkstra's algorithm** (or **Dijkstra's Shortest Path First algorithm**, **SPF algorithm**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm) for finding the [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem" \o "Shortest path problem) between [nodes](https://en.wikipedia.org/wiki/Vertex_(graph_theory)" \o "Vertex (graph theory)) in a [graph](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)" \o "Graph (abstract data type)), which may represent, for example, [road networks](https://en.wikipedia.org/wiki/Road_network" \o "Road network). It was conceived by [computer scientist](https://en.wikipedia.org/wiki/Computer_scientist" \o "Computer scientist) [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra" \o "Edsger W. Dijkstra) in 1956 and published three years later.  The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes,but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a [shortest-path tree](https://en.wikipedia.org/wiki/Shortest-path_tree" \o "Shortest-path tree).  For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithm is network [routing protocols](https://en.wikipedia.org/wiki/Routing_protocol" \o "Routing protocol), most notably [IS-IS](https://en.wikipedia.org/wiki/IS-IS" \o "IS-IS) (Intermediate System to Intermediate System) and Open Shortest Path First ([OSPF](https://en.wikipedia.org/wiki/OSPF" \o "OSPF)). It is also employed as a [subroutine](https://en.wikipedia.org/wiki/Subroutine" \o "Subroutine) in other algorithms such as [Johnson's](https://en.wikipedia.org/wiki/Johnson's_algorithm" \o "Johnson's algorithm).  The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the Generic Dijkstra shortest-path algorithm.[[6]](https://en.wikipedia.org/wiki/Dijkstra's_algorithm" \l "cite_note-Generic_Dijkstra-6)  Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. The original algorithm uses a [min-priority queue](https://en.wikipedia.org/wiki/Min-priority_queue" \o "Min-priority queue) and runs in [time](https://en.wikipedia.org/wiki/Time_complexity" \o "Time complexity) {\displaystyle O(|V|^{2})}IMG_256(where {\displaystyle |V|}IMG_257is the number of nodes). The idea of this algorithm is also given in [Leyzorek et al. 1957](https://en.wikipedia.org/wiki/Dijkstra's_algorithm" \l "CITEREFLeyzorekGrayJohnsonLadew1957). [Fredman & Tarjan 1984](https://en.wikipedia.org/wiki/Dijkstra's_algorithm" \l "CITEREFFredmanTarjan1984) propose using a [Fibonacci heap](https://en.wikipedia.org/wiki/Fibonacci_heap" \o "Fibonacci heap) min-priority queue to optimize the running time complexity to {\displaystyle O(|E|+|V|\log |V|)}IMG_258(where {\displaystyle |E|}IMG_259is the number of edges). This is [asymptotically](https://en.wikipedia.org/wiki/Asymptotic_computational_complexity" \o "Asymptotic computational complexity) the fastest known single-source [shortest-path algorithm](https://en.wikipedia.org/wiki/Shortest_path_problem" \o "Shortest path problem) for arbitrary [directed graphs](https://en.wikipedia.org/wiki/Directed_graph" \o "Directed graph) with unbounded non-negative weights. However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in [Specialized variants](https://en.wikipedia.org/wiki/Dijkstra's_algorithm" \l "Specialized_variants" \o "Dijkstra's algorithm).  Algorithm.  Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.   1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*. 2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.[[13]](https://en.wikipedia.org/wiki/Dijkstra's_algorithm" \l "cite_note-13) 3. For the current node, consider all of its unvisited neighbours and calculate their *tentative* distances through the current node. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbour *B* has length 2, then the distance to *B* through *A* will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept. 4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again. 5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished. 6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.   When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current").  Code:-  void min\_heapify(int \*b, int i, int n)  {  int j, temp;  temp = b[i];  j = 2 \* i;  while (j <= n)  {  if (j < n && b[j + 1] < b[j])  {  j = j + 1;  }  if (temp < b[j])  {  break;  }  else if (temp >= b[j])  {  b[j / 2] = b[j];  j = 2 \* j;  }}  b[j / 2] = temp;  return;  }  void build\_minheap(int \*b, int n)  {  int i;  for(i = n / 2; i >= 1; i--)  {  min\_heapify(b, i, n);  }  }  void addEdge(int am[][7], int src, int dest, int cost)  {  am[src][dest] = cost;  return;  }  void bell(int am[][7])  {  int i, j, k, c = 0, temp;  a[0].cost = 0;  a[0].from = 0;  a[0].value = 0;  for (i = 1; i < 7; i++)  {  a[i].from = 0;  a[i].cost = INFINITY;  a[i].value = 0;  }  while (c < 7)  {  int min = 999;  for (i = 0; i < 7; i++)  {  if (min > a[i].cost && a[i].value == 0)  {  min = a[i].cost;  }  else  {  continue;  }  }  for (i = 0; i < 7; i++)  {  if (min == a[i].cost && a[i].value == 0)  {  break;  }  else  {  continue;}}  temp = i;  for (k = 0; k < 7; k++)  {  if (am[temp][k] + a[temp].cost < a[k].cost)  {  a[k].cost = am[temp][k] + a[temp].cost;  a[k].from = temp;  }  else  {  continue;  }}  a[temp].value = 1;  c++;  }  cout<<"Cost"<<"\t"<<"Source Node"<<endl;  for (j = 0; j < 7; j++)  {  cout<<a[j].cost<<"\t"<<a[j].from<<endl;  }}  /\*Driver Code\*/  int n,am[7][7],c=0,i,j,cost;  for(int j;j<7;j++)  {  am[i][j]=INFINITY;    }  while (c<12)  {  cout<<"Enter the source,destination and cost of edge\n";  cin>>i>>j>>cost;  addEdge(am,i,j,cost);  c++;    bell(am);  }  **OUTPUT**    Case Study 2:-**Check if a linked list of strings forms a palindrome**  **(By reversing the list)** This method takes O(n) time and O(1) extra space. **1)** Get the middle of the linked list. **2)**Reverse the second half of the linked list. **3)** Check if the first half and second half are identical. **4)**Construct the original linked list by reversing the second half again and attaching it back to the first half  To divide the list in two halves, method 2 of [this](https://www.geeksforgeeks.org/write-a-c-function-to-print-the-middle-of-the-linked-list/" \t "https://www.geeksforgeeks.org/function-to-check-if-a-singly-linked-list-is-palindrome/_blank)post is used. When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don’t want the middle node as part of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable ‘midnode’.  Code:-  #include <stdbool.h>  #include <stdio.h>  #include <stdlib.h>    /\* Link list node \*/  **struct** Node {  **char** data;  **struct** Node\* next;  };    **void** reverse(**struct** Node\*\*);  **bool** compareLists(**struct** Node\*, **struct** Node\*);    /\* Function to check if given linked list is    palindrome or not \*/  **bool** isPalindrome(**struct** Node\* head)  {  **struct** Node \*slow\_ptr = head, \*fast\_ptr = head;  **struct** Node \*second\_half, \*prev\_of\_slow\_ptr = head;  **struct** Node\* midnode = NULL; // To handle odd size list  **bool** res = **true**; // initialize result    **if** (head != NULL && head->next != NULL) {          /\* Get the middle of the list. Move slow\_ptr by 1            and fast\_ptrr by 2, slow\_ptr will have the middle            node \*/  **while** (fast\_ptr != NULL && fast\_ptr->next != NULL) {              fast\_ptr = fast\_ptr->next->next;                /\*We need previous of the slow\_ptr for               linked lists  with odd elements \*/              prev\_of\_slow\_ptr = slow\_ptr;              slow\_ptr = slow\_ptr->next;          }            /\* fast\_ptr would become NULL when there are even elements in list.             And not NULL for odd elements. We need to skip the middle node             for odd case and store it somewhere so that we can restore the             original list\*/  **if** (fast\_ptr != NULL) {              midnode = slow\_ptr;              slow\_ptr = slow\_ptr->next;          }            // Now reverse the second half and compare it with first half          second\_half = slow\_ptr;          prev\_of\_slow\_ptr->next = NULL; // NULL terminate first half          reverse(&second\_half); // Reverse the second half          res = compareLists(head, second\_half); // compare            /\* Construct the original list back \*/          reverse(&second\_half); // Reverse the second half again            // If there was a mid node (odd size case) which          // was not part of either first half or second half.  **if** (midnode != NULL) {              prev\_of\_slow\_ptr->next = midnode;              midnode->next = second\_half;          }  **else**              prev\_of\_slow\_ptr->next = second\_half;      }  **return** res;  }    /\* Function to reverse the linked list  Note that this      function may change the head \*/  **void** reverse(**struct** Node\*\* head\_ref)  {  **struct** Node\* prev = NULL;  **struct** Node\* current = \*head\_ref;  **struct** Node\* next;  **while** (current != NULL) {          next = current->next;          current->next = prev;          prev = current;          current = next;      }      \*head\_ref = prev;  }    /\* Function to check if two input lists have same data\*/  **bool** compareLists(**struct** Node\* head1, **struct** Node\* head2)  {  **struct** Node\* temp1 = head1;  **struct** Node\* temp2 = head2;    **while** (temp1 && temp2) {  **if** (temp1->data == temp2->data) {              temp1 = temp1->next;              temp2 = temp2->next;          }  **else**  **return** 0;      }        /\* Both are empty reurn 1\*/  **if** (temp1 == NULL && temp2 == NULL)  **return** 1;        /\* Will reach here when one is NULL        and other is not \*/  **return** 0;  }    /\* Push a node to linked list. Note that this function    changes the head \*/  **void** push(**struct** Node\*\* head\_ref, **char** new\_data)  {      /\* allocate node \*/  **struct** Node\* new\_node = (**struct** Node\*)**malloc**(**sizeof**(**struct** Node));        /\* put in the data  \*/      new\_node->data = new\_data;        /\* link the old list off the new node \*/      new\_node->next = (\*head\_ref);        /\* move the head to pochar to the new node \*/      (\*head\_ref) = new\_node;  }    // A utility function to print a given linked list  **void** printList(**struct** node\* ptr)  {  **while** (ptr != NULL) {  **printf**("%c->", ptr->data);          ptr = ptr->next;      }  **printf**("NULL\n");  }    /\* Drier program to test above function\*/  **int** main()  {      /\* Start with the empty list \*/  **struct** Node\* head = NULL;  **char** str[] = "abacaba";  **int** i;    **for** (i = 0; str[i] != '\0'; i++) {          push(&head, str[i]);          printList(head);          isPalindrome(head) ? **printf**("Is Palindrome\n\n") : **printf**("Not Palindrome\n\n");      }    **return** 0;  } |

**Output:**

a->NULL

Palindrome

b->a->NULL

Not Palindrome

a->b->a->NULL

Is Palindrome

c->a->b->a->NULL

Not Palindrome

a->c->a->b->a->NULL

Not Palindrome

b->a->c->a->b->a->NULL

Not Palindrome