DATA STRUCTURES AND ALGORITHMS FINAL PROJECT CEP

Submitted to:
Sir. Khalid Butt

Submitted by:

- Abdullah Hassaan Ahmad 2019-EE-113 (section C)
- Syed Mouaaz Farrukh 2019-EE-116 (Section D)
- Filza Batool 2019-EE-154 (Section D)
- Faiq Irfan 2019-EE-112 (Section C)

DSA LAB PROJECT:

Problem statement:

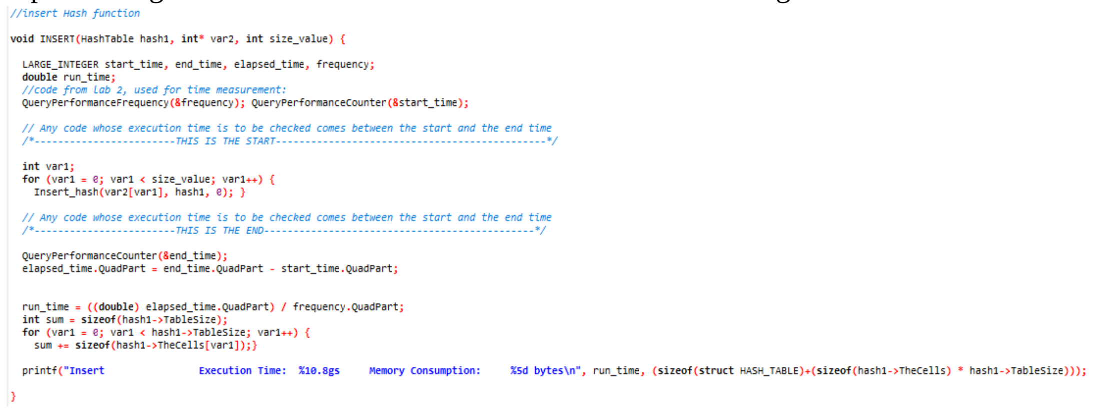
Given project requires to Insert, Sort, Find and Delete required values in a given data (provided by instructor) for enteries no. 10,100,1000,10000,100000 and 1000000 and the datatypes we have to experiment are

- Hashtables,
- Arrays
- Linked Lists
- Trees.

1) HASH_TABLES:

Insert function:

- Input coming from read file will be inserted in the hash table using this function.



```
//insert Hash function

void INSERT(HashTable hash1, int* var2, int size_value) {

  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
  //code from Lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/

  int var1;
  for (var1 = 0; var1 < size_value; var1++) {
    Insert_hash(var2[var1], hash1, 0); }

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;


  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;
  int sum = sizeof(hash1->TableSize);
  for (var1 = 0; var1 < hash1->TableSize; var1++) {
    sum += sizeof(hash1->TheCells[var1]);}

  printf("Insert               Execution Time:  %10.8gs     Memory Consumption:     %5d bytes\n", run_time, (sizeof(struct HASH_TABLE)+(sizeof(hash1->TheCells) * hash1->TableSize)));

}
```

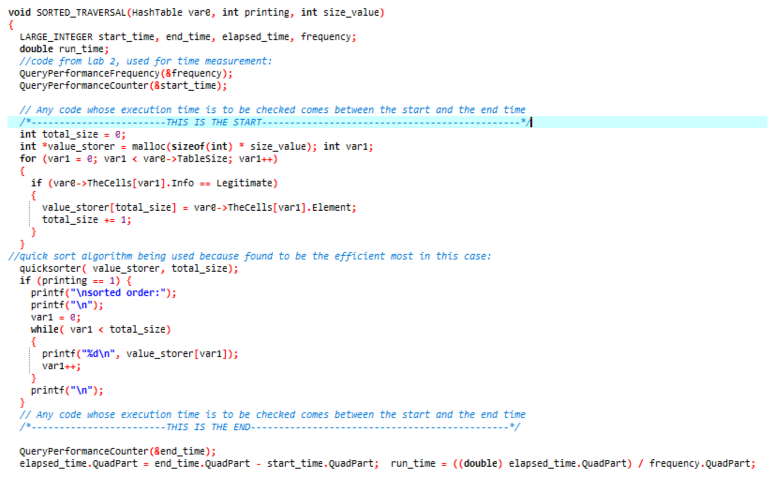
- In this function, a inserting function is called within function which was provided in the labs.

Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(1) (which is the case for most general case for insert functions). You have to insert anything at a desired place in list and so O(1) is the best approximation made.
-

SORTED_TRAVERSAL:

- Now that we have a hash table, we have to sort it.
- Here, we have used Quicksort algorithm as shown in figure (as an example):
- When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.



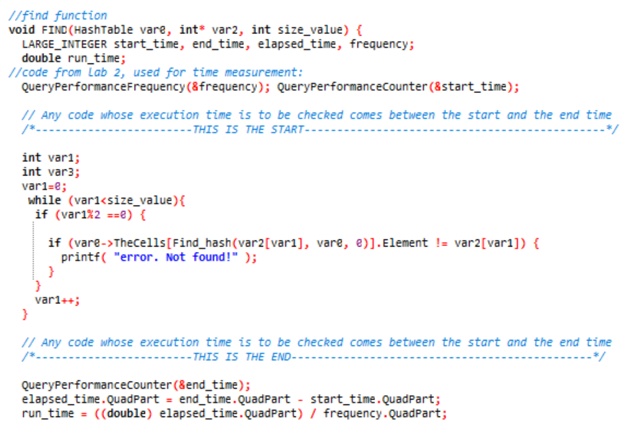
```
void SORTED_TRAVERSAL(HashTable var0, int printing, int size_value)
{
  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
  //code from Lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/
  int total_size = 0;
  int *value_storer = malloc(sizeof(int) * size_value); int var1;
  for (var1 = 0; var1 < var0->TableSize; var1++)
  {
    if (var0->TheCells[var1].Info == Legitimate)
    {
      value_storer[total_size] = var0->TheCells[var1].Element;
      total_size += 1;
    }
  }
//quick sort algorithm being used because found to be the efficient most in this case:
  quicksorter( value_storer, total_size);
  if (printing == 1) {
    printf("\nsorted order:");
    printf("\n");
    var1 = 0;
    while( var1 < total_size)
    {
      printf("%d\n", value_storer[var1]);
      var1++;
    }
    printf("\n");
  }
  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;
```

Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(Nlog*N) forquick sort specially.

❖ **FIND function:**

- Now that we have a list, we have to find some element in it.
- Unlike the case for linked lists, this function finds the point or position where value is located instead of traversing the whole table.



```
//find function
void FIND(HashTable var0, int* var2, int size_value) {
  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
//code from lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/

  int var1;
  int var3;
  var1=0;
   while (var1<size_value){
    if (var1%2 ==0) {

      if (var0->TheCells[Find_hash(var2[var1], var0, 0)].Element != var2[var1]) {
        printf( "error. Not found!" );
      }
    }
    var1++;
  }

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;
  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;
```

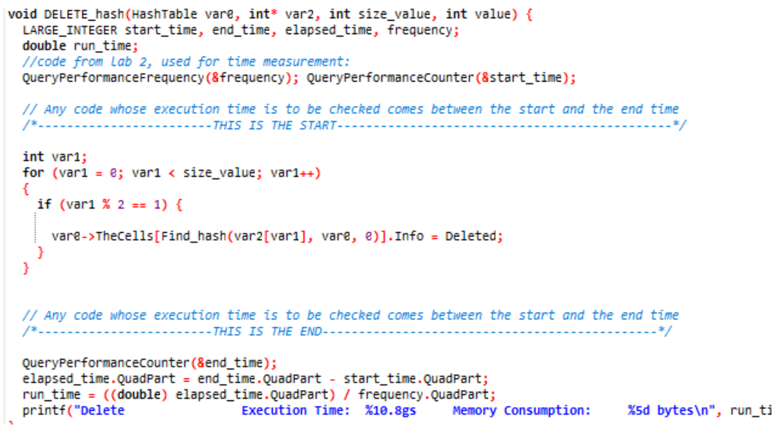
Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(1).

❖

❖ **DELETE_Hash function:**

- Now that we have a hash table, and we have to delete some element in it.
- We do not need to go through the whole table to find the desired value to be deleted.
- This function finds the point or position where value is located instead of traversing the whole table and deletes the value.



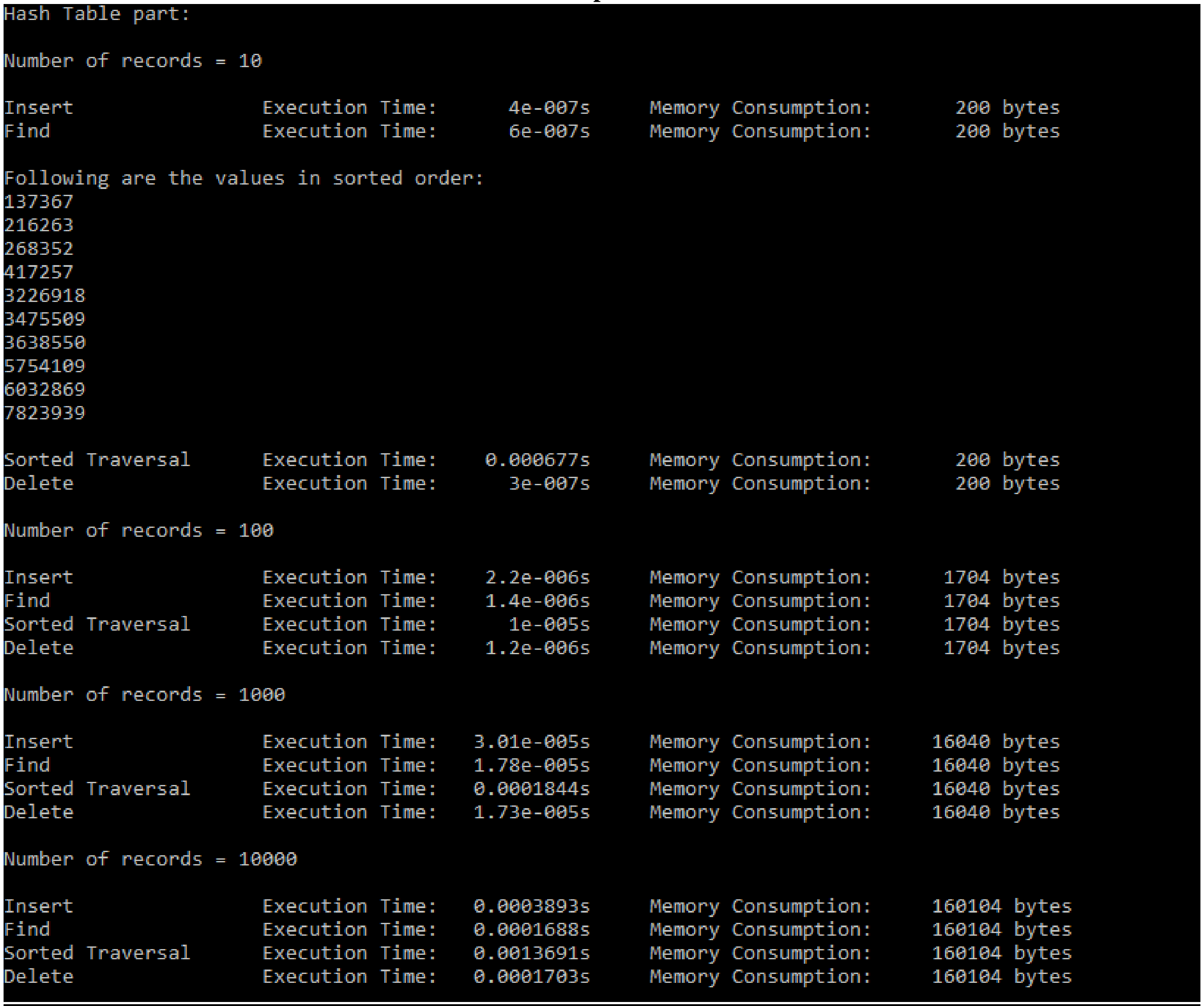
```
void DELETE_hash(HashTable var0, int* var2, int size_value, int value) {
  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
  //code from lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/

  int var1;
  for (var1 = 0; var1 < size_value; var1++)
  {
    if (var1 % 2 == 1) {

      var0->TheCells[Find_hash(var2[var1], var0, 0)].Info = Deleted;
    }
  }


  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;
  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;
  printf("Delete                Execution Time:  %10.8gs     Memory Consumption:     %5d bytes\n", run_ti
```

Time complexity:

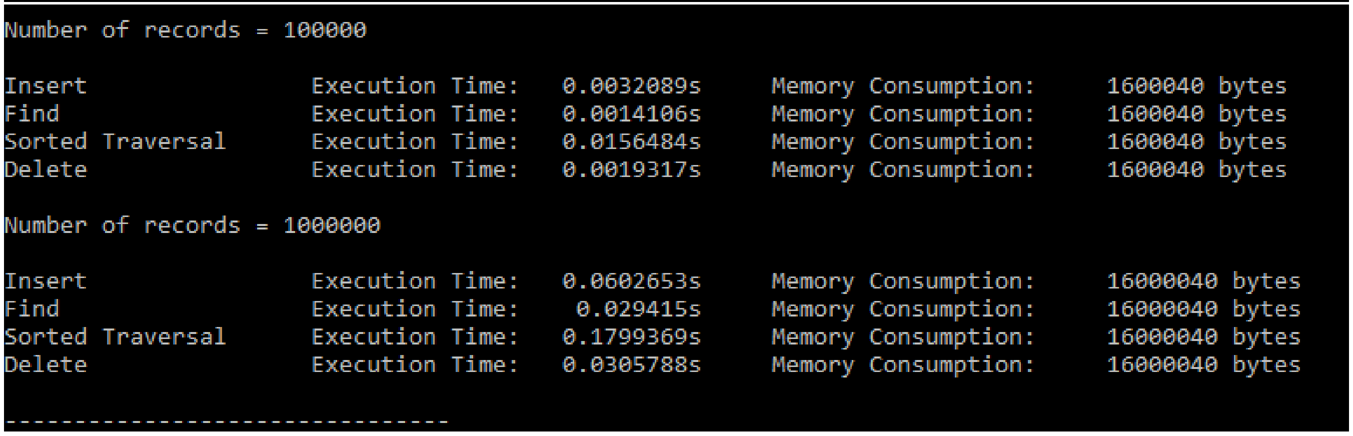
- Query performance code from Lab 2 has been used for time taken measurement.

In general, time complexity for this code is O(1). Because there is no need to traverse the whole table.

- **Output :**



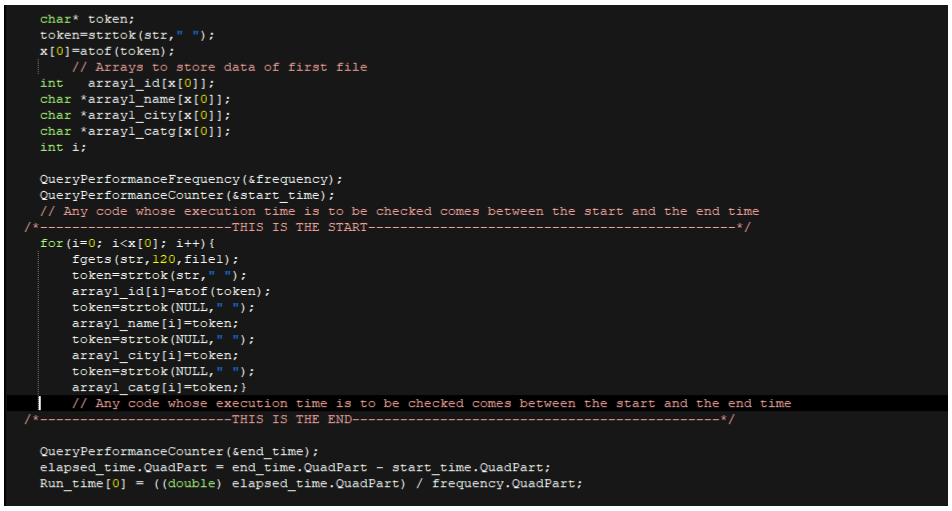
```
Hash Table part:

Number of records = 10

Insert                 Execution Time:      4e-007s     Memory Consumption:       200 bytes
Find                   Execution Time:      6e-007s     Memory Consumption:       200 bytes

Following are the values in sorted order:
137367
216263
268352
417257
3226918
3475509
3638550
5754109
6032869
7823939

Sorted Traversal       Execution Time:    0.000677s     Memory Consumption:       200 bytes
Delete                 Execution Time:      3e-007s     Memory Consumption:       200 bytes

Number of records = 100

Insert                 Execution Time:    2.2e-006s     Memory Consumption:      1704 bytes
Find                   Execution Time:    1.4e-006s     Memory Consumption:      1704 bytes
Sorted Traversal       Execution Time:      1e-005s     Memory Consumption:      1704 bytes
Delete                 Execution Time:    1.2e-006s     Memory Consumption:      1704 bytes

Number of records = 1000

Insert                 Execution Time:   3.01e-005s     Memory Consumption:     16040 bytes
Find                   Execution Time:   1.78e-005s     Memory Consumption:     16040 bytes
Sorted Traversal       Execution Time:   0.0001844s     Memory Consumption:     16040 bytes
Delete                 Execution Time:   1.73e-005s     Memory Consumption:     16040 bytes

Number of records = 10000

Insert                 Execution Time:   0.0003893s     Memory Consumption:     160104 bytes
Find                   Execution Time:   0.0001688s     Memory Consumption:     160104 bytes
Sorted Traversal       Execution Time:   0.0013691s     Memory Consumption:     160104 bytes
Delete                 Execution Time:   0.0001703s     Memory Consumption:     160104 bytes
```



```
Number of records = 100000

Insert                 Execution Time:   0.0032089s     Memory Consumption:     1600040 bytes
Find                   Execution Time:   0.0014106s     Memory Consumption:     1600040 bytes
Sorted Traversal       Execution Time:   0.0156484s     Memory Consumption:     1600040 bytes
Delete                 Execution Time:   0.0019317s     Memory Consumption:     1600040 bytes

Number of records = 1000000

Insert                 Execution Time:   0.0602653s     Memory Consumption:     16000040 bytes
Find                   Execution Time:    0.029415s     Memory Consumption:     16000040 bytes
Sorted Traversal       Execution Time:   0.1799369s     Memory Consumption:     16000040 bytes
Delete                 Execution Time:   0.0305788s     Memory Consumption:     16000040 bytes

---------------------------------
```

2) Array

- **Insert:**
  - Input is received through the ***fgets*** function from the given data files and then it will be inserted in the array using the following code.
  - We have initialize four arrays containing ID, name, city and category. Using for loop and the ***fgets*** function in it we separate the info.

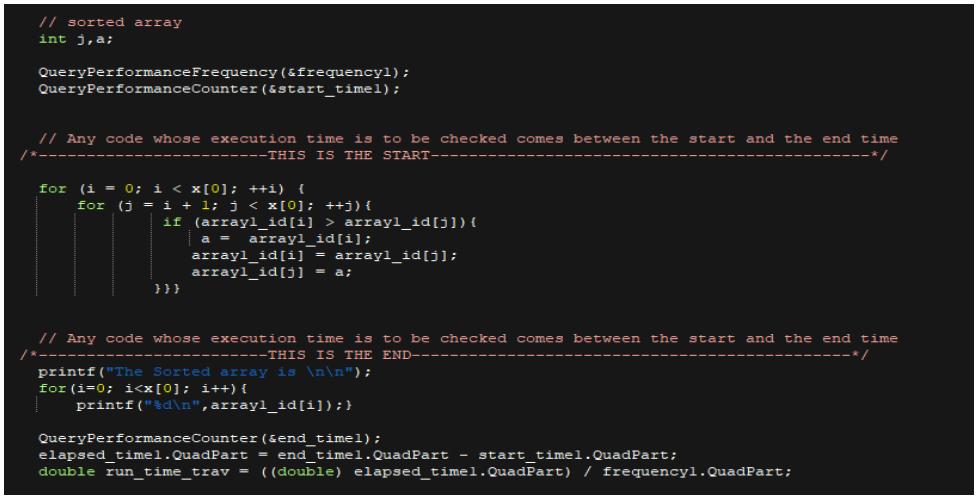


```
  char* token;
  token=strtok(str," ");
  x[0]=atof(token);
      // Arrays to store data of first file
  int   array1_id[x[0]];
  char *array1_name[x[0]];
  char *array1_city[x[0]];
  char *array1_catg[x[0]];
  int i;

  QueryPerformanceFrequency(&frequency);
  QueryPerformanceCounter(&start_time);
  // Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE START---------------------------------------------*/
  for(i=0; i<x[0]; i++){
      fgets(str,120,file1);
      token=strtok(str," ");
      array1_id[i]=atof(token);
      token=strtok(NULL," ");
      array1_name[i]=token;
      token=strtok(NULL," ");
      array1_city[i]=token;
      token=strtok(NULL," ");
      array1_catg[i]=token;}
      // Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE END-------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;
  Run_time[0] = ((double) elapsed_time.QuadPart) / frequency.QuadPart;
```

Time complexity:

- In general, Time complexity of insert is O (n) (which is the case for most general case for insert functions).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

- **SORTED TRAVERSAL:**
  - Sorted traversal gives the array after sorting it out through the following code.
  - We take the array containing the Id's and then that is sorted out by using for loop.

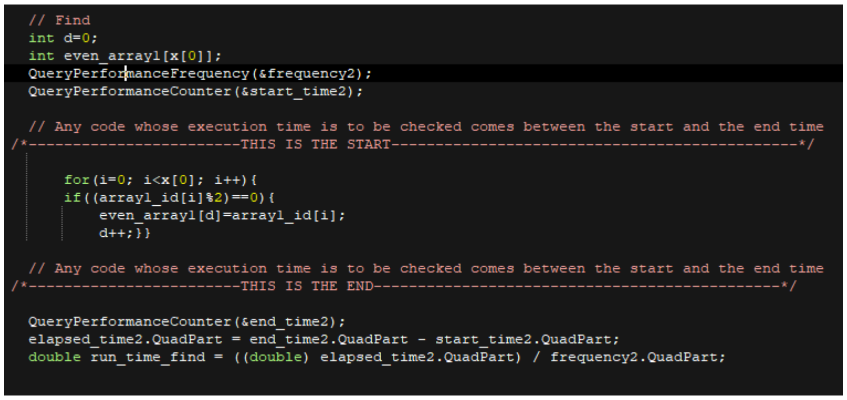


```
  // sorted array
  int j,a;

  QueryPerformanceFrequency(&frequency1);
  QueryPerformanceCounter(&start_time1);


  // Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE START---------------------------------------------*/

  for (i = 0; i < x[0]; ++i) {
      for (j = i + 1; j < x[0]; ++j){
                if (arrayl_id[i] > arrayl_id[j]){
                    a =  arrayl_id[i];
                  arrayl_id[i] = arrayl_id[j];
                  arrayl_id[j] = a;
              }}}


  // Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE END---------------------------------------------*/
  printf("The Sorted array is \n\n");
  for(i=0; i<x[0]; i++){
      printf("%d\n",arrayl_id[i]);}

  QueryPerformanceCounter(&end_time1);
  elapsed_time1.QuadPart = end_time1.QuadPart - start_time1.QuadPart;
  double run_time_trav = ((double) elapsed_time1.QuadPart) / frequency1.QuadPart;
```

Time complexity:

- In general, Time complexity of insert is O (nlogn) (which is the case for most general case for sort functions).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

- **FIND:**The array is traverse in which you read data from file and for every record with even index in this array, the record is found in the array using a for loop as shown in the code below.



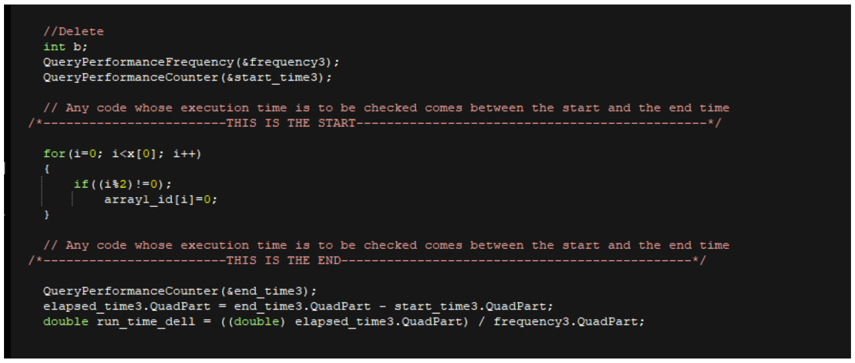
```
// Find
int d=0;
int even_array1[x[0]];
QueryPerformanceFrequency(&frequency2);
QueryPerformanceCounter(&start_time2);

// Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE START---------------------------------------------*/

    for(i=0; i<x[0]; i++){
    if((array1_id[i]%2)==0){
        even_array1[d]=array1_id[i];
        d++;}}

// Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE END---------------------------------------------*/

QueryPerformanceCounter(&end_time2);
elapsed_time2.QuadPart = end_time2.QuadPart - start_time2.QuadPart;
double run_time_find = ((double) elapsed_time2.QuadPart) / frequency2.QuadPart;
```

Time complexity:

- In general, Time complexity of insert is O (1) (which is the case for most general case for find functions in array).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

❖ **<u>DELETE</u>:**

- The array is traverse in which you read data from file and for every record with odd index in this array, the record is deleted in the array using for loop as shown in the code below.

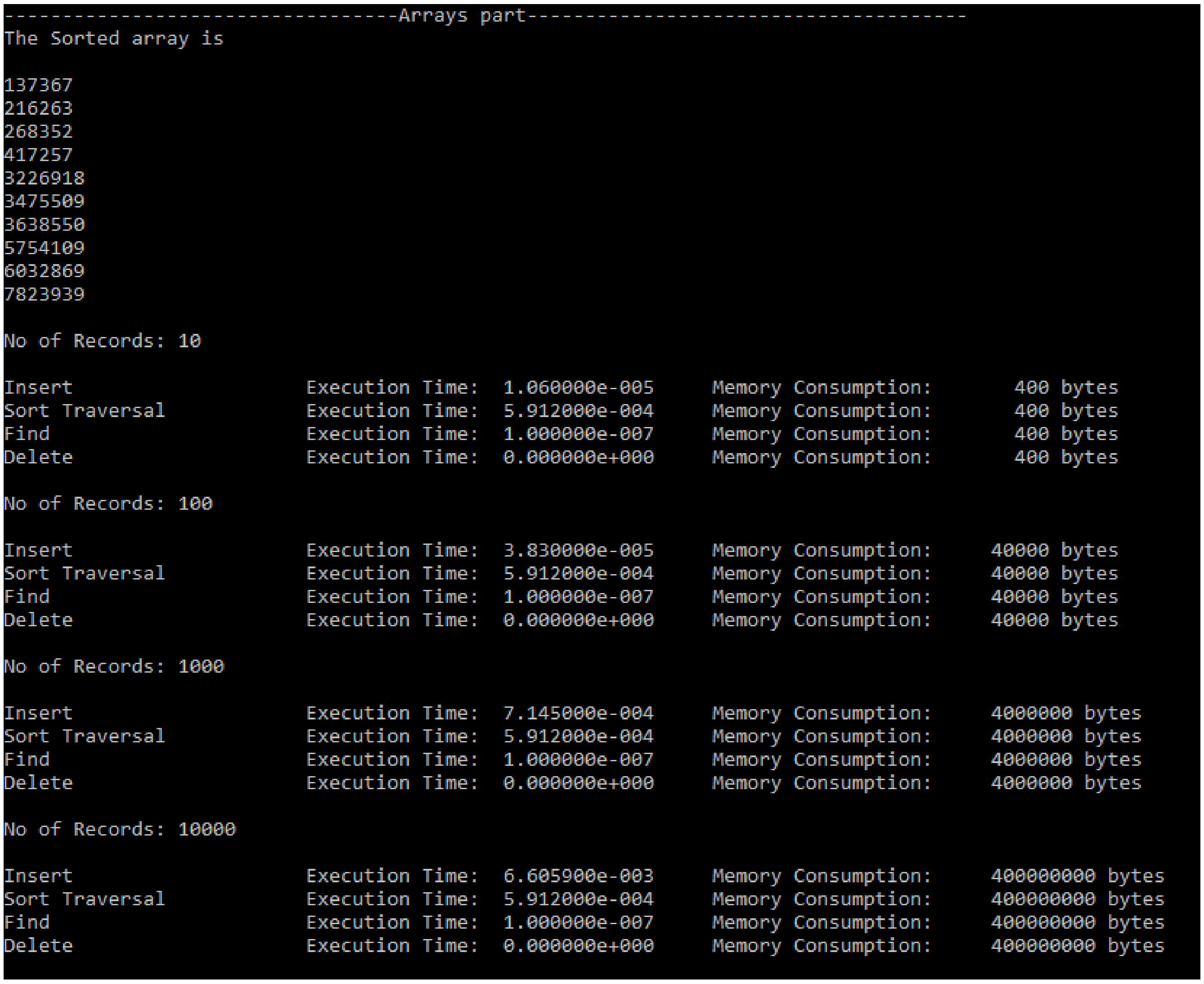


```
//Delete
int b;
QueryPerformanceFrequency(&frequency3);
QueryPerformanceCounter(&start_time3);

// Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE START-----------------------------------------------*/

for(i=0; i<x[0]; i++)
{
    if((i%2)!=0);
        array1_id[i]=0;
}

// Any code whose execution time is to be checked comes between the start and the end time
/*------------------------THIS IS THE END-----------------------------------------------*/

QueryPerformanceCounter(&end_time3);
elapsed_time3.QuadPart = end_time3.QuadPart - start_time3.QuadPart;
double run_time_dell = ((double) elapsed_time3.QuadPart) / frequency3.QuadPart;
```

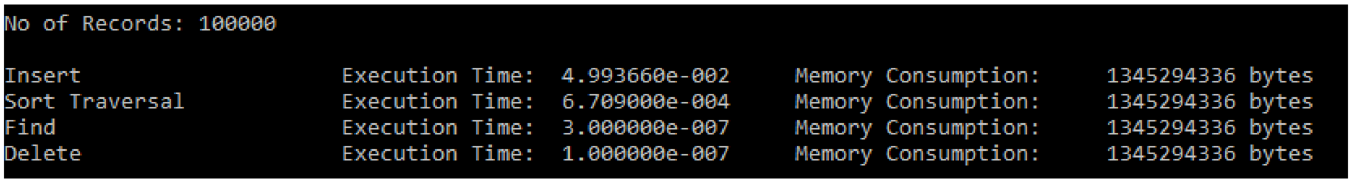
Time complexity:

- In general, Time complexity of insert is O (1) (which is the case for most general case for delete functions in array).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

Output :



```
----------------------------------Arrays part--------------------------------------
The Sorted array is

137367
216263
268352
417257
3226918
3475509
3638550
5754109
6032869
7823939

No of Records: 10

Insert                    Execution Time:  1.060000e-005     Memory Consumption:       400 bytes
Sort Traversal            Execution Time:  5.912000e-004     Memory Consumption:       400 bytes
Find                      Execution Time:  1.000000e-007     Memory Consumption:       400 bytes
Delete                    Execution Time:  0.000000e+000     Memory Consumption:       400 bytes

No of Records: 100

Insert                    Execution Time:  3.830000e-005     Memory Consumption:     40000 bytes
Sort Traversal            Execution Time:  5.912000e-004     Memory Consumption:     40000 bytes
Find                      Execution Time:  1.000000e-007     Memory Consumption:     40000 bytes
Delete                    Execution Time:  0.000000e+000     Memory Consumption:     40000 bytes

No of Records: 1000

Insert                    Execution Time:  7.145000e-004     Memory Consumption:     4000000 bytes
Sort Traversal            Execution Time:  5.912000e-004     Memory Consumption:     4000000 bytes
Find                      Execution Time:  1.000000e-007     Memory Consumption:     4000000 bytes
Delete                    Execution Time:  0.000000e+000     Memory Consumption:     4000000 bytes

No of Records: 10000

Insert                    Execution Time:  6.605900e-003     Memory Consumption:     400000000 bytes
Sort Traversal            Execution Time:  5.912000e-004     Memory Consumption:     400000000 bytes
Find                      Execution Time:  1.000000e-007     Memory Consumption:     400000000 bytes
Delete                    Execution Time:  0.000000e+000     Memory Consumption:     400000000 bytes
```



```
No of Records: 100000

Insert                    Execution Time:  4.993660e-002     Memory Consumption:     1345294336 bytes
Sort Traversal            Execution Time:  6.709000e-004     Memory Consumption:     1345294336 bytes
Find                      Execution Time:  3.000000e-007     Memory Consumption:     1345294336 bytes
Delete                    Execution Time:  1.000000e-007     Memory Consumption:     1345294336 bytes
```

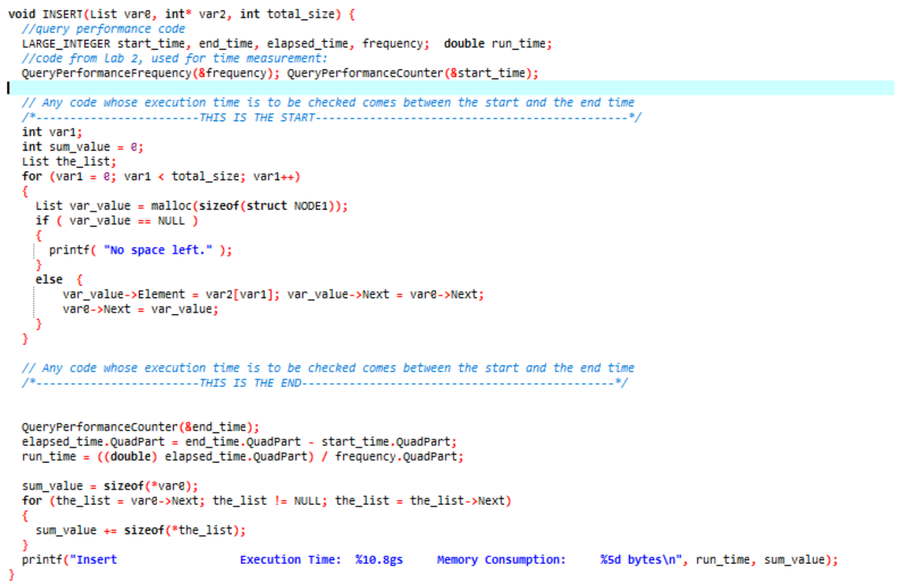
3) LINKED LISTS:

❖ **Insert function:**

- Input coming from file will be inserted in the linked list using this function.
- We will initialize a list ‘temp’ and one extra node to accomodate the pointing of the node next to head node.

Time complexity:

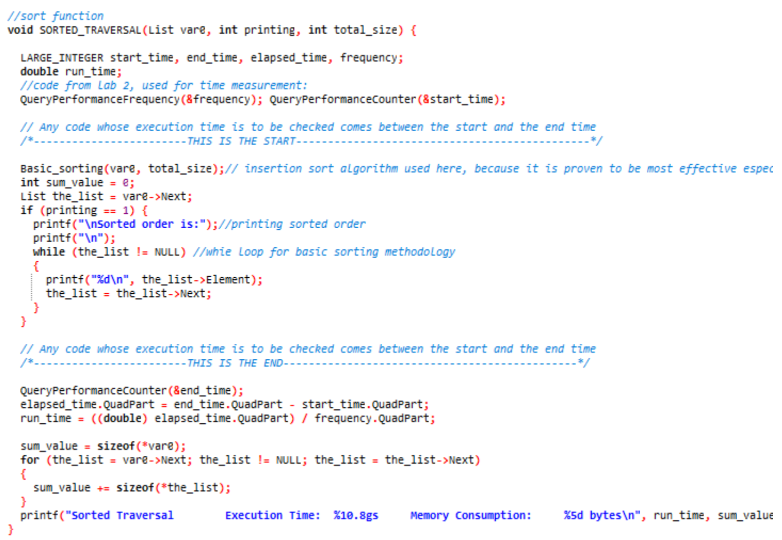
- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(1) (which is the case for most general case for insert functions). You have to insert anything at a desired place in list and so O(1) is the best approximation made.



```
void INSERT(List var0, int* var2, int total_size) {
  //query performance code
  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;  double run_time;
  //code from Lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/
  int var1;
  int sum_value = 0;
  List the_list;
  for (var1 = 0; var1 < total_size; var1++)
  {
    List var_value = malloc(sizeof(struct NODE1));
    if ( var_value == NULL )
    {
      printf( "No space left." );
    }
    else  {
        var_value->Element = var2[var1]; var_value->Next = var0->Next;
        var0->Next = var_value;
    }
  }

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/


  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;
  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;

  sum_value = sizeof(*var0);
  for (the_list = var0->Next; the_list != NULL; the_list = the_list->Next)
  {
    sum_value += sizeof(*the_list);
  }
  printf("Insert                 Execution Time:  %10.8gs     Memory Consumption:     %5d bytes\n", run_time, sum_value);
}
```

❖ **SORTED_TRAVERSAL function:**

- Now that we have a list, we have to sort it.
- Here, we have used insertion sort algorithm as shown in figure (as an example):

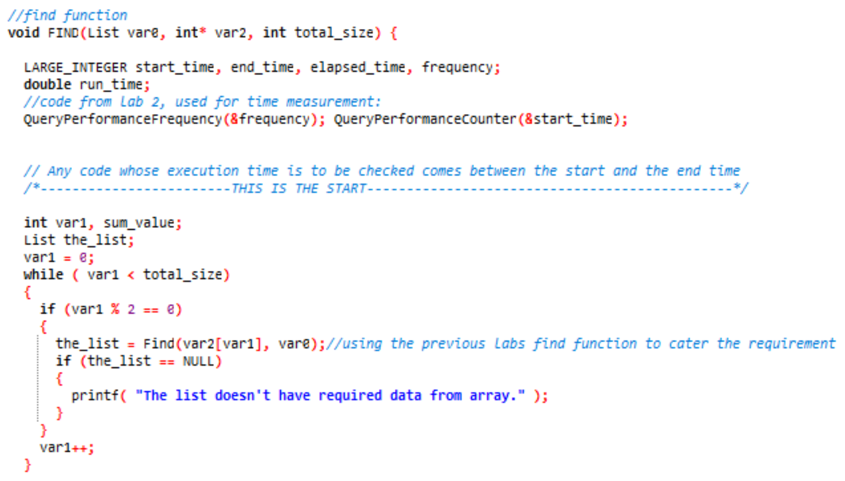


```
//sort function
void SORTED_TRAVERSAL(List var0, int printing, int total_size) {

  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
  //code from Lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/

  Basic_sorting(var0, total_size);// insertion sort algorithm used here, because it is proven to be most effective espec
  int sum_value = 0;
  List the_list = var0->Next;
  if (printing == 1) {
    printf("\nSorted order is:");//printing sorted order
    printf("\n");
    while (the_list != NULL) //whie loop for basic sorting methodology
    {
      printf("%d\n", the_list->Element);
      the_list = the_list->Next;
    }
  }

  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE END---------------------------------------------*/

  QueryPerformanceCounter(&end_time);
  elapsed_time.QuadPart = end_time.QuadPart - start_time.QuadPart;
  run_time = ((double) elapsed_time.QuadPart) / frequency.QuadPart;

  sum_value = sizeof(*var0);
  for (the_list = var0->Next; the_list != NULL; the_list = the_list->Next)
  {
    sum_value += sizeof(*the_list);
  }
  printf("Sorted Traversal       Execution Time:  %10.8gs     Memory Consumption:     %5d bytes\n", run_time, sum_value
}
```

- Insertion sort checks adjacent numbers to sort the whole list.

Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(N^2) because we have to check and compare two values N times.



```
//find function
void FIND(List var0, int* var2, int total_size) {

  LARGE_INTEGER start_time, end_time, elapsed_time, frequency;
  double run_time;
  //code from Lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);


  // Any code whose execution time is to be checked comes between the start and the end time
  /*-----------------------THIS IS THE START---------------------------------------------*/

  int var1, sum_value;
  List the_list;
  var1 = 0;
  while ( var1 < total_size)
  {
    if (var1 % 2 == 0)
    {
      the_list = Find(var2[var1], var0);//using the previous labs find function to cater the requirement
      if (the_list == NULL)
      {
        printf( "The list doesn't have required data from array." );
      }
    }
    var1++;
  }
```

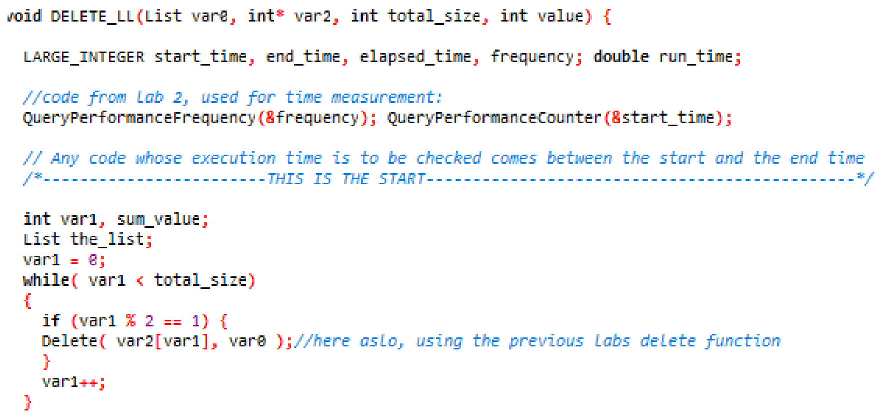
❖ **FIND function:**

- Now that we have a list, we have to find some element in it.
- We need to go through the whole linked list, to find the desired value.

Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(N). This is the worst case because element could be found at the 1st element-check too.

- **DELETE_LL function:**
  - Now that we have a list, and we have to delete some element in it.
  - We need to go through the whole linked list, to find the desired value to be deleted.
  - Next we perform the standard deleting algorithm used in linked list (the bypassing of nodes) as shown below:

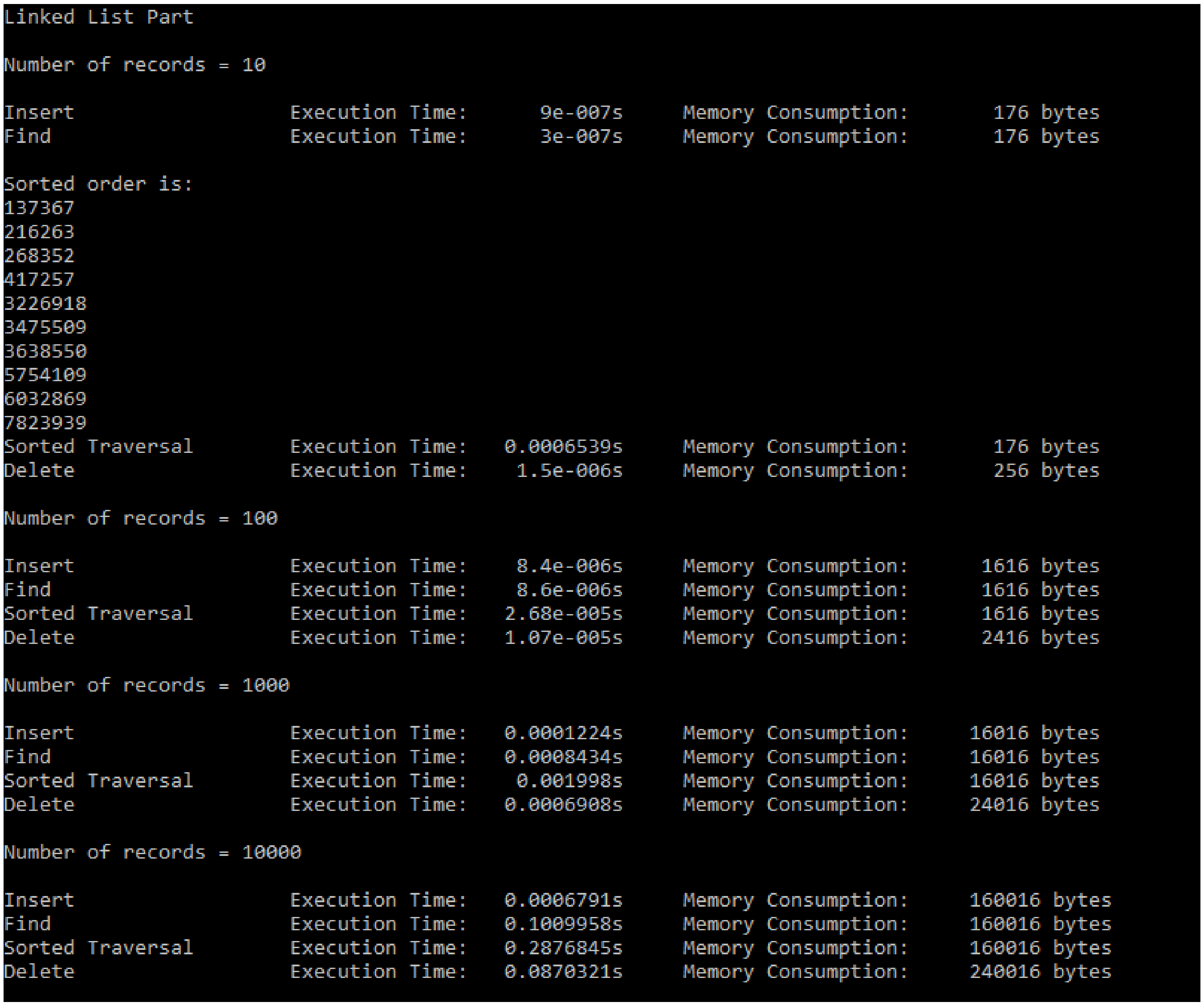


```
void DELETE_LL(List var0, int* var2, int total_size, int value) {

  LARGE_INTEGER start_time, end_time, elapsed_time, frequency; double run_time;

  //code from lab 2, used for time measurement:
  QueryPerformanceFrequency(&frequency); QueryPerformanceCounter(&start_time);

  // Any code whose execution time is to be checked comes between the start and the end time
  /*------------------------THIS IS THE START---------------------------------------------*/

  int var1, sum_value;
  List the_list;
  var1 = 0;
  while( var1 < total_size)
  {
    if (var1 % 2 == 1) {
    Delete( var2[var1], var0 );//here aslo, using the previous labs delete function
    }
    var1++;
  }
```

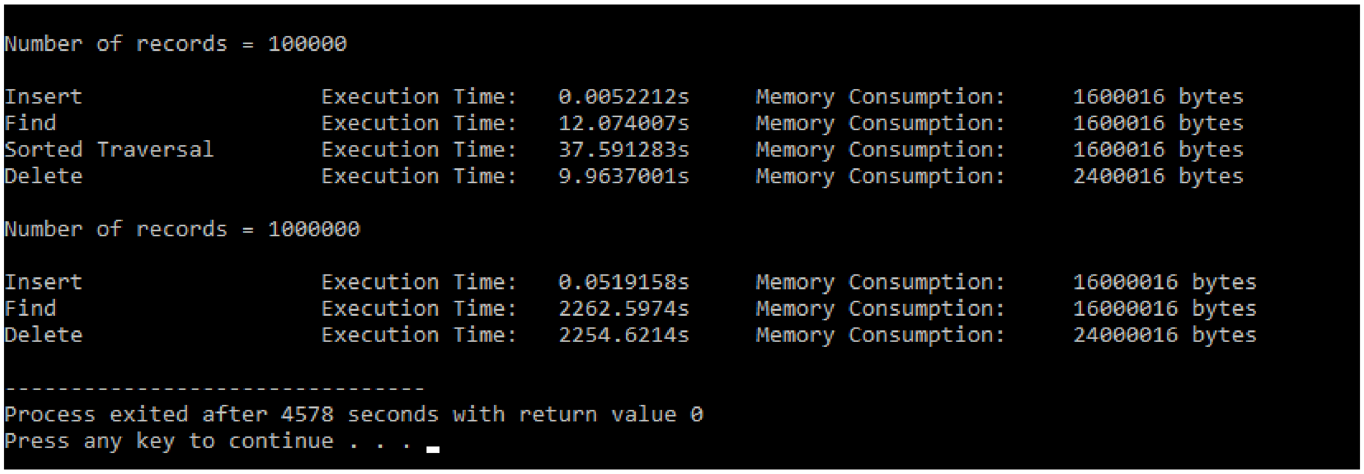
Time complexity:

- Query performance code from Lab 2 has been used for time taken measurement.
- In general, time complexity for this code is O(N). Because first, element to be deleted has to be found(which can extend for N-checks) and then deleted which is constant O(1).

Output :



```
Linked List Part

Number of records = 10

Insert                  Execution Time:      9e-007s     Memory Consumption:       176 bytes
Find                    Execution Time:      3e-007s     Memory Consumption:       176 bytes

Sorted order is:
137367
216263
268352
417257
3226918
3475509
3638550
5754109
6032869
7823939
Sorted Traversal        Execution Time:   0.0006539s     Memory Consumption:       176 bytes
Delete                  Execution Time:    1.5e-006s     Memory Consumption:       256 bytes

Number of records = 100

Insert                  Execution Time:    8.4e-006s     Memory Consumption:      1616 bytes
Find                    Execution Time:    8.6e-006s     Memory Consumption:      1616 bytes
Sorted Traversal        Execution Time:   2.68e-005s     Memory Consumption:      1616 bytes
Delete                  Execution Time:   1.07e-005s     Memory Consumption:      2416 bytes

Number of records = 1000

Insert                  Execution Time:   0.0001224s     Memory Consumption:     16016 bytes
Find                    Execution Time:   0.0008434s     Memory Consumption:     16016 bytes
Sorted Traversal        Execution Time:    0.001998s     Memory Consumption:     16016 bytes
Delete                  Execution Time:   0.0006908s     Memory Consumption:     24016 bytes

Number of records = 10000

Insert                  Execution Time:   0.0006791s     Memory Consumption:     160016 bytes
Find                    Execution Time:   0.1009958s     Memory Consumption:     160016 bytes
Sorted Traversal        Execution Time:   0.2876845s     Memory Consumption:     160016 bytes
Delete                  Execution Time:   0.0870321s     Memory Consumption:     240016 bytes
```



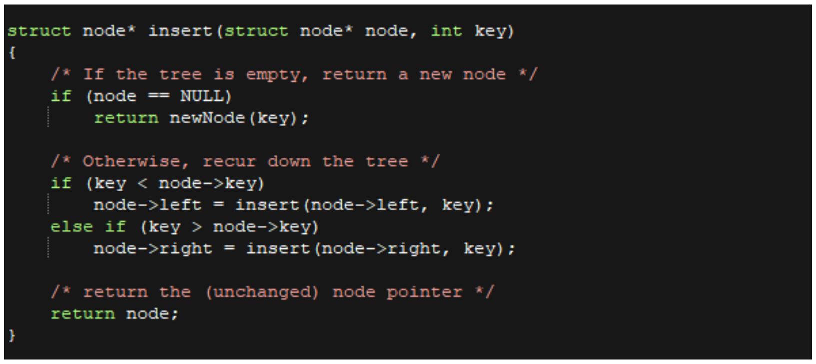
```
Number of records = 100000

Insert                  Execution Time:   0.0052212s     Memory Consumption:     1600016 bytes
Find                    Execution Time:   12.074007s     Memory Consumption:     1600016 bytes
Sorted Traversal        Execution Time:   37.591283s     Memory Consumption:     1600016 bytes
Delete                  Execution Time:   9.9637001s     Memory Consumption:     2400016 bytes

Number of records = 1000000

Insert                  Execution Time:   0.0519158s     Memory Consumption:     16000016 bytes
Find                    Execution Time:   2262.5974s     Memory Consumption:     16000016 bytes
Delete                  Execution Time:   2254.6214s     Memory Consumption:     24000016 bytes

--------------------------------
Process exited after 4578 seconds with return value 0
Press any key to continue . . .
```

*sorted traversal for 1000000 entries is ignored.

4) Trees

❖ **Insert:**

- Input is received through the ***fgets*** function from the given data files and then it will be inserted in the array.
- Then the insert function using the trees (binary search tree), the keys are used to find the position and the elements are inserted through the following code.

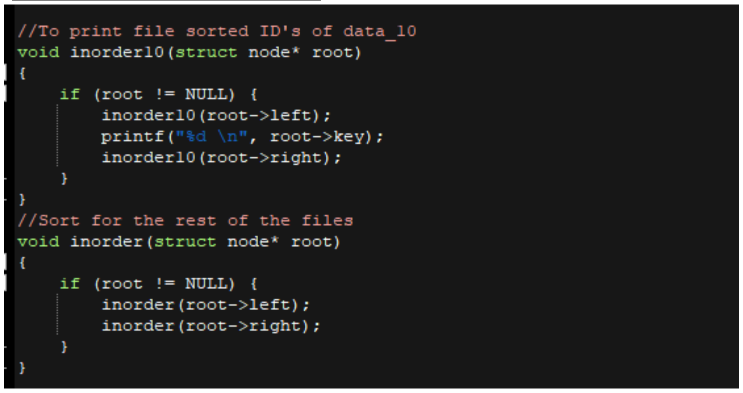


```
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

Time complexity:

- In general, Time complexity of insert is O (logN) (which is the case for most general case for insert functions).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

❖ **SORTED TRAVERSAL:**



```
//To print file sorted ID's of data_10
void inorder10(struct node* root)
{
    if (root != NULL) {
        inorder10(root->left);
        printf("%d \n", root->key);
        inorder10(root->right);
    }
}
//Sort for the rest of the files
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        inorder(root->right);
    }
}
```

- Trees sort uses binary search tree for sorting. Binary search tree is created from the arrays containing data and afterwards in-order traversal is performed to get input in sorted order.

Time complexity:

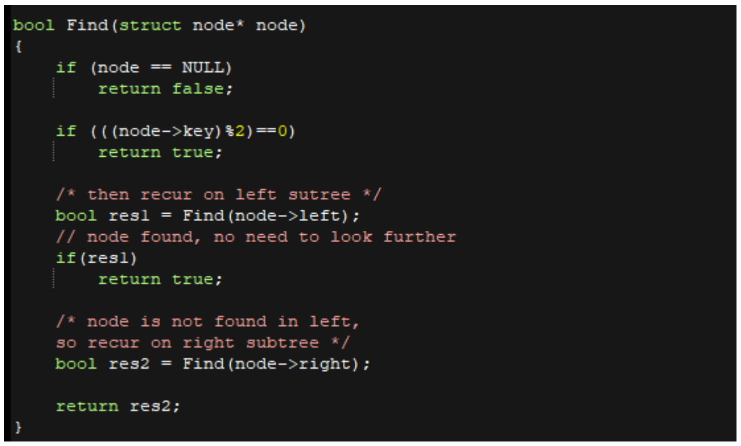
- In general, Time complexity of insert is O (N) (which is the case for most general case for sort functions).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

❖ **<u>FIND</u>:**

- Here the array is traversed in which you read data from file and for every record with even index in this array, the record is found in the binary search tree using the key.

  Time complexity:

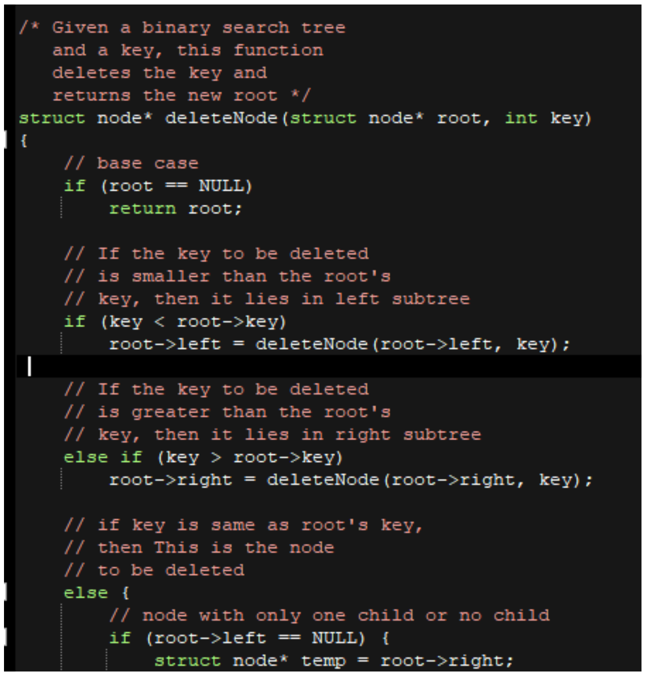
  - In general, Time complexity of insert is O (N) (which is the case for most general case for find functions in array).
  - To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.



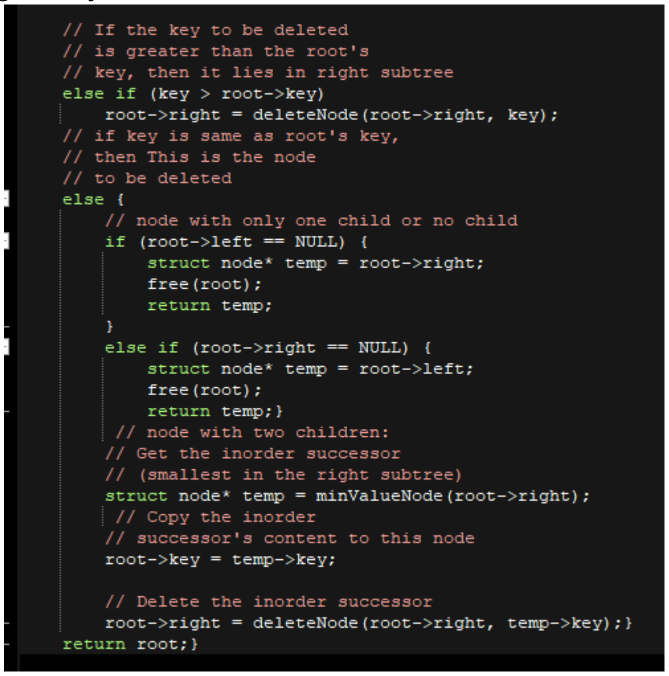
```
bool Find(struct node* node)
{
    if (node == NULL)
        return false;

    if (((node->key)%2)==0)
        return true;

    /* then recur on left sutree */
    bool res1 = Find(node->left);
    // node found, no need to look further
    if(res1)
        return true;

    /* node is not found in left,
    so recur on right subtree */
    bool res2 = Find(node->right);

    return res2;
}
```

❖ **<u>DELETE</u>:**

- The array is traverse in which you read data from file and for every record with odd index in the Binary search tree, the record is deleted in the BST using the key as shown in the code below.



```
/* Given a binary search tree
   and a key, this function
   deletes the key and
   returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL)
        return root;

    // If the key to be deleted
    // is smaller than the root's
    // key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted
    // is greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key,
    // then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (root->left == NULL) {
            struct node* temp = root->right;
```

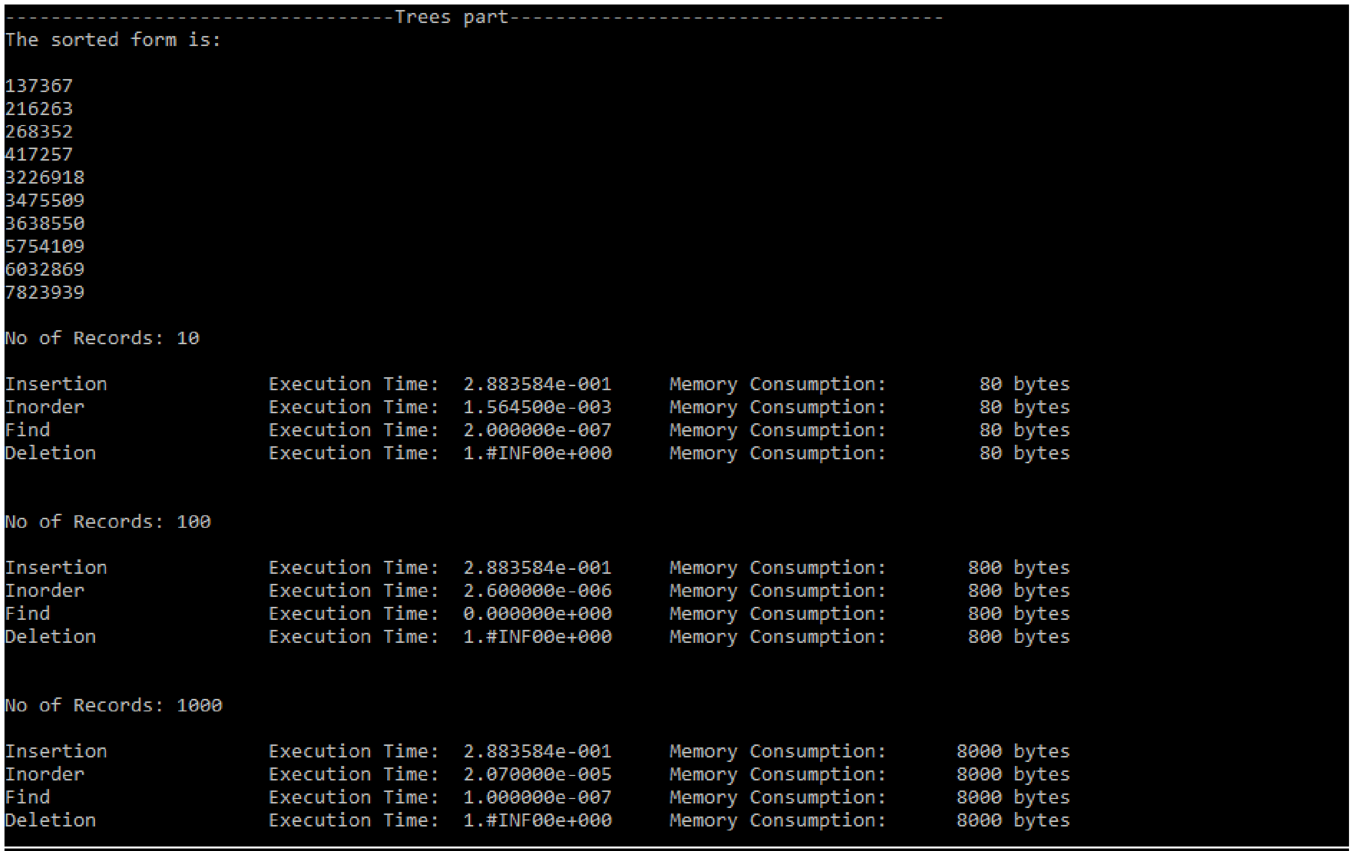


```
    // If the key to be deleted
    // is greater than the root's
    // key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    // if key is same as root's key,
    // then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;}
         // node with two children:
        // Get the inorder successor
        // (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);
         // Copy the inorder
        // successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);}
    return root;}
```

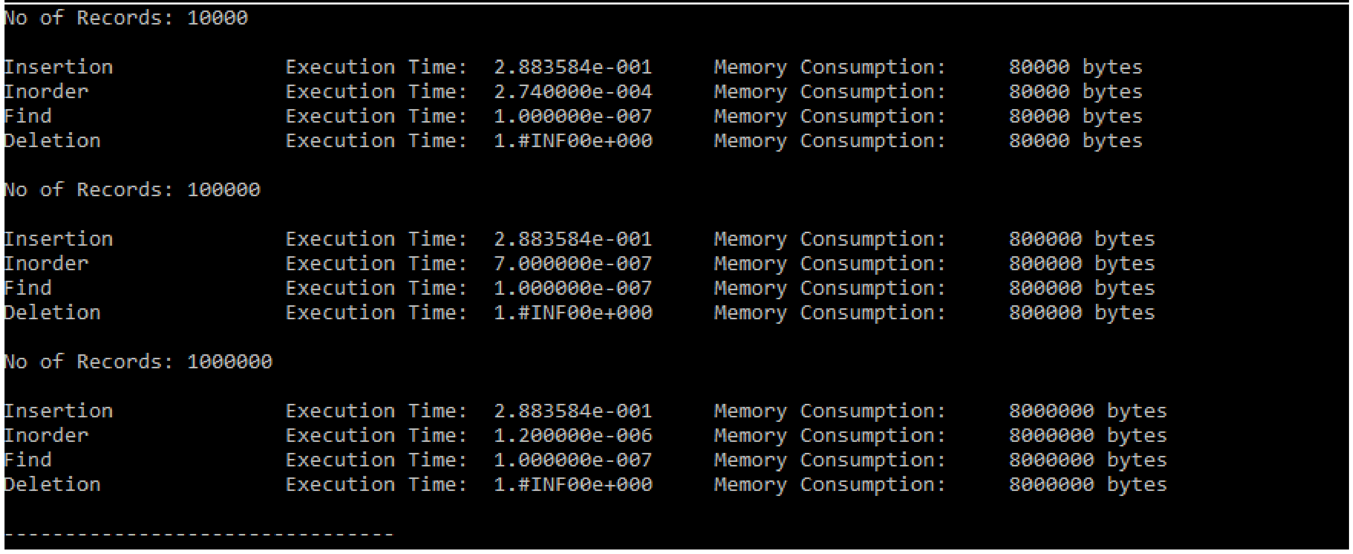
Time complexity:

- In general, Time complexity of insert is O (logN) (which is the case for most general case for delete functions in array).
- To calculate accurate time of the code Query performance counter is used from Lab 2 which is the most accurate method is used.

Output:



```
----------------------------------Trees part--------------------------------------
The sorted form is:

137367
216263
268352
417257
3226918
3475509
3638550
5754109
6032869
7823939

No of Records: 10

Insertion              Execution Time:  2.883584e-001     Memory Consumption:        80 bytes
Inorder                Execution Time:  1.564500e-003     Memory Consumption:        80 bytes
Find                   Execution Time:  2.000000e-007     Memory Consumption:        80 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:        80 bytes


No of Records: 100

Insertion              Execution Time:  2.883584e-001     Memory Consumption:       800 bytes
Inorder                Execution Time:  2.600000e-006     Memory Consumption:       800 bytes
Find                   Execution Time:  0.000000e+000     Memory Consumption:       800 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:       800 bytes


No of Records: 1000

Insertion              Execution Time:  2.883584e-001     Memory Consumption:      8000 bytes
Inorder                Execution Time:  2.070000e-005     Memory Consumption:      8000 bytes
Find                   Execution Time:  1.000000e-007     Memory Consumption:      8000 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:      8000 bytes
```



```
No of Records: 10000

Insertion              Execution Time:  2.883584e-001     Memory Consumption:     80000 bytes
Inorder                Execution Time:  2.740000e-004     Memory Consumption:     80000 bytes
Find                   Execution Time:  1.000000e-007     Memory Consumption:     80000 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:     80000 bytes

No of Records: 100000

Insertion              Execution Time:  2.883584e-001     Memory Consumption:     800000 bytes
Inorder                Execution Time:  7.000000e-007     Memory Consumption:     800000 bytes
Find                   Execution Time:  1.000000e-007     Memory Consumption:     800000 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:     800000 bytes

No of Records: 1000000

Insertion              Execution Time:  2.883584e-001     Memory Consumption:     8000000 bytes
Inorder                Execution Time:  1.200000e-006     Memory Consumption:     8000000 bytes
Find                   Execution Time:  1.000000e-007     Memory Consumption:     8000000 bytes
Deletion               Execution Time:  1.#INF00e+000     Memory Consumption:     8000000 bytes

--------------------------------
```

➢ Results Table:

Entries 10:

	Insert		*Find*		*Sort*		*Delete*	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*4.00E-07*	*200*	*6.00E-07*	*200*	*6.77E-03*	*200*	*3.00E-07*	*200*
Arrays	*1.60E-05*	*400*	*1.00E-07*	*400*	*5.91E-04*	*400*	*0.00E-00*	*400*
Trees	*2.88E-01*	*80*	*1.56E-03*	*80*	*2.00E-07*	*80*	*1.00E-00*	*80*
Linked List	*9.00E-07*	*176*	*3.00E-07*	*176*	*6.53E-04*	*176*	*1.50E-06*	*256*

Entries 100:

	Insert		*Find*		*Sort*		*Delete*	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*2.20E-06*	*1704*	*1.40E-06*	*1704*	*1.00E-05*	*1704*	*1.20E-06*	*1704*
Arrays	*3.83E-05*	*40000*	*1.00E-07*	*40000*	*5.91E-04*	*40000*	*0.00E-00*	*40000*
Trees	*1.88E-01*	*800*	*0.00E-00*	*800*	*2.60E-06*	*800*	*1.00E-00*	*800*
Linked List	*8.40E-06*	*1616*	*8.60E-06*	*1616*	*2.68E-05*	*1616*	*1.07E-05*	*2416*

Entries 1000:

	Insert		*Find*		*Sort*		*Delete*	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*3.01E-05*	*16040*	*1.78E-05*	*16040*	*1.84E-04*	*16040*	*1.73E-05*	*16040*
Arrays	*7.41E-04*	*4000000*	*1.00E-07*	*4000000*	*5.91E-04*	*4000000*	*0.00E-00*	*4000000*
Trees	*2.88E-01*	*8000*	*1.00E-07*	*8000*	*2.07E-07*	*8000*	*1.00E-00*	*8000*
Linked List	*1.22E-04*	*16016*	*8.43E-04*	*16016*	*1.99E-03*	*16016*	*6.90E-04*	*24016*

Enteries 10000:

	Insert		***Find***		***Sort***		***Delete***	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*3.89E-04*	*160104*	*1.68E-04*	*160104*	*1.38E-03*	*160104*	*1.70E-04*	*160104*
Arrays	*6.60E-03*	*40000000000*	*1.00E-07*	*40000000000*	*5.91E-04*	*40000000000*	*0.00E-00*	*40000000000*
Trees	*2.88E-01*	*80000*	*1.00E-07*	*80000*	*2.74E-04*	*80000*	*1.00E-00*	*80000*
Linked List	*6.79E-04*	*160016*	*1.00E-01*	*160016*	*2.80E-01*	*160016*	*8.70E-02*	*240016*

Enteries 100000:

	Insert		***Find***		***Sort***		***Delete***	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*3.20E-003*	*1600040*	*1.41E-03*	*1600040*	*1.56E-02*	*1600040*	*1.93E-03*	*1600040*
Arrays	*4.99366E-003*	*1345294336*	*3E-007*	*1345294336*	*6.709E-04*	*1345294336*	*1.00E-007*	*1345294336*
Trees	*2.88E-01*	*800000*	*1.00E-07*	*800000*	*7.00E-07*	*800000*	*1.00E-00*	*800000*
Linked List	*0.00522*	*1600016*	*12.074*	*1600016*	*37.5912*	*1600016*	*9.9637001*	*2400016*

Enteries 1000000:

	Insert		***Find***		***Sort***		***Delete***	
	Time (sec)	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***	***Time (sec)***	***Memory (bytes)***
Hash	*6.02E-02*	*16000040*	*2.94E-02*	*16000040*	*1.79E-01*	*16000040*	*3.05E-02*	*16000040*
Arrays	-	-	-	-	-	-	-	-
Trees	*2.88E-01*	*8000000*	*1.00E-07*	*8000000*	*1.20E-06*	*8000000*	*1.00E-00*	*8000000*
Linked List	*0.0519*	*1600016*	*2262.5974*	*1600016*	-	-	*2254.6214*	*1600016*

**arrays output for 1000000 entries was quite massive and was not compatible with the PC.*

Conclusion:

- Above we, see that trees are the best for sorting procedure.
- Also, hash tables are good at inserting.
- Arrays have taken lesser time in finding and deletion.
- Also, hash tables are good at deletion.

I would like to prefer hash tables or trees and not arrays because at more higher values they take tremendous running time and memory space also.