G. H. RAISONI COLLEGE OF ENGG., NAGPUR

(An Autonomous Institute under UGC Act 1956)

Department of Artificial Intelligence

Date: 27/07/2020

Practical Subject: Data Structures and Algorithms
Session: 2020-21

Student Details:

Roll Number	63
Name	Vishal Narnaware
Semester	3
Section	A
Branch	Artificial Intelligence

Practical Details: Practical Number- 3

Practical Aim	Design, develop and implement a menu driven program in C++ for implementing the following sorting methods to arrange a list of integers in ascending order: a) Insertion sort b) Merge sort c) Quick sort d) Heap sort
Theory	***Insertion Sort:*** Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort. ***Merge Sort:*** In Merge sort the idea is to split the unsorted list into smaller groups until there is only one element in a group. Then, group two elements in the sorted order and gradually build the size of the group. Every time the merging

	happens, the elements in the groups must be compared one by one and combined into a single list in the sorted order. This process continues till all the elements are merged and sorted. ***Quick Sort:*** Like Merge Sort, 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. 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. ***Heap Sort:*** Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
Procedure	***Insertion Sort:*** To sort an array of size n in ascending order: 1. Iterate from arr[1] to arr[n] over the array. 2. Compare the current element (key) to its predecessor. 3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element. ***Merge Sort:*** 1. Check if right > left 2. Find the middle point to divide the array into two halves 3. Call mergeSort for first half 4. Call mergeSort for second half 5. Merge the two halves sorted in step 3 and 4 ***Quick Sort:*** 1. We start from the leftmost element and keep track of index of smaller (or equal to) elements as i. 2. While traversing, if we find a smaller element, we swap current element with arr[i]. 3. Otherwise we ignore current element.

	Heap Sort: 1. Build a max heap from the input data. 2. At this point, the largest item is stored at the root of the heap. 3. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree. 4. Repeat above steps while size of heap is greater than 1.
Algorithm	***Insertion Sort:*** Step 1: START Step 2: Consider the first element to be sorted and the rest to be unsorted Step 3: Compare with the second element, if the second element $<$ the first element, insert the element in the correct position of the sorted portion else, leave it as it is Step 4: Repeat 2 and 3 until all elements are sorted. Step 5: STOP ***Merge Sort:*** Step 1: START Step 2: Split the unsorted list into groups recursively until there is one element per group Step 3: Compare each of the elements and then group them Step 4: Repeat step 3 until the whole list is merged and sorted in the process Step 5: STOP ***Quick Sort:*** Step 1: START Step 2: Choose the highest index value as pivot Step 3: Take two variables to point left and right of the list excluding pivot Step 4: Left points to the low index and right points to the high Step 5: While value at left is less than pivot, move right Step 6: While value at right is greater than pivot, move left Step 7: If both step 5 and step 6 does not match swap left and right Step 8: If left $\geq$ right, the point where they met is new pivot Step 9: STOP

Heap Sort:

Step 1: START

Step 2: Construct a Binary Tree with given list of Elements.

Step 3: Transform the Binary Tree into Min Heap.

Step 4: Delete the root element from Min Heap using Heapify method.

Step 5: Put the deleted element into the Sorted list.

Step 6: Repeat the same until Min Heap becomes empty.

Step 7: STOP

Program

practical3.cpp

```cpp
#include <iostream>

using namespace std;

void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void insertion(int arr[], int siz)
{
    int temp, i, j;
    for(i=1; i<siz; i++)    {
        temp = arr[i];
        j = i;
        while(j>0 && temp<arr[j-1]) {
            arr[j] = arr[j-1];
            j -= 1;
        }
        arr[j] = temp;
    }
}

```

```
void merge(int A[] , int start, int mid, int end)
{
    int p=start, q=mid+1;
    int Arr[end-start+1], k=0;

    for(int i=start; i<=end; i++)   {
        if(p > mid)
            Arr[k++] = A[q++];
        else if(q > end)
            Arr[k++] = A[p++];
        else if(A[p] < A[q])
            Arr[k++] = A[p++];
        else
            Arr[k++] = A[q++];
    }
    for(int p=0; p<k; p++) {
        A[start++] = Arr[p];
  }
}
void mergeSort(int A[], int start, int end)
{
    if(start < end) {
    int mid = (start+end)/2 ;
    mergeSort(A, start, mid);
    mergeSort(A, mid+1, end);
    merge(A, start, mid, end);
   }
}

int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)   {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)  {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```cpp
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i)   {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapSort(int arr[], int n)
{
    for (int i = n/2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i=n-1; i>0; i--)   {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

void display(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

int main()
{
    int arr[100], n, ch;
    char ext;
    cout << "\n\tAuthor: Vishal Narnaware. (A - 63)";
    do {
        cout << "\n Enter the size of Array: ";
        cin >> n;
        cout << "\n Enter the Array: ";
        for(int i=0; i<n; i++)  {
            cin >> arr[i];
        }
```

```cpp
        cout << "\n\n -----MAIN_MENU-----";
        cout << "\n 1. Insertion Sort";
        cout << "\n 2. Merge Sort";
        cout << "\n 3. Quick Sort";
        cout << "\n 4. Heap Sort";
        cout << "\n Enter your choice: ";
        cin >> ch;
        switch(ch)  {
            case 1: insertion(arr, n);
                    break;
            case 2: mergeSort(arr, 0, n-1);
                    break;
            case 3: quickSort(arr, 0, n-1);
                    break;
            case 4: heapSort(arr, n);
                    break;
            case 5:
            default: cout << "\n Err!!! Wrong Choice";
                    break;
        }
        cout << "\n Sorted array is: ";
        display(arr, n);
        cout << "\n Do you want to enter again?(y/n): ";
        cin >> ext;
    } while(ext != 'n');
}
```

Output

Insertion Sort:

```
C:\Users\bagde\Desktop\Vishal\DSA prac>g++ -o out.exe practical3.cpp

C:\Users\bagde\Desktop\Vishal\DSA prac>out.exe

        Author: Vishal Narnaware. (A - 63)
 Enter the size of Array: 5

 Enter the Array: 9 3 5 2 8


 -----MAIN_MENU-----
 1. Insertion Sort
 2. Merge Sort
 3. Quick Sort
 4. Heap Sort
 Enter your choice: 1

 Sorted array is: 2 3 5 8 9

 Do you want to enter again?(y/n): y
```

Merge Sort:

```
Enter the size of Array: 5

Enter the Array: 9 3 5 2 8


-----MAIN_MENU-----
1. Insertion Sort
2. Merge Sort
3. Quick Sort
4. Heap Sort
Enter your choice: 2

Sorted array is: 2 3 5 8 9

Do you want to enter again?(y/n): y
```

Quick Sort:

```
Enter the size of Array: 5

Enter the Array: 9 3 5 2 8


-----MAIN_MENU-----
1. Insertion Sort
2. Merge Sort
3. Quick Sort
4. Heap Sort
Enter your choice: 3

Sorted array is: 2 3 5 8 9

Do you want to enter again?(y/n): y
```

Heap Sort:

```
Enter the size of Array: 5

Enter the Array: 9 3 5 2 8


-----MAIN_MENU-----
1. Insertion Sort
2. Merge Sort
3. Quick Sort
4. Heap Sort
Enter your choice: 4

Sorted array is: 2 3 5 8 9

Do you want to enter again?(y/n): n

C:\Users\bagde\Desktop\Vishal\DSA prac>
```

Conclusion	Hence, successfully designed and implemented Sorting Algorithms in C++ and analyzed their Time Complexities.