**/\*Searching(Linear Search, Binary Search) and Sorting(Insertion Sort, Bubble Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort) time of the following Algorithms on randomly generated array.\*/**

//#pragma once

#include <Windows.h>

#include <iostream>

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

#include<string.h>

//#include<vector>

#include<algorithm>

//#include<conio.h>

using namespace std;

enum Operations

{

LINEAR\_SEARCH=1,BINARY\_SEARCH,INSERTION\_SORT, BUBBLE\_SORT,MERGE\_SORT,SELECTION\_SORT,QUICK\_SORT,HEAP\_SORT,EXIT

};

class CHRTimer

{

protected:

LARGE\_INTEGER m\_liStart;

LARGE\_INTEGER m\_liStop;

public:

CHRTimer(void)

{

m\_liStart.QuadPart = m\_liStop.QuadPart = 0;

}

~CHRTimer(void)

{

}

// Starts the timer

void Start()

{

::QueryPerformanceCounter(&m\_liStart);

}

// Stops the timer

void Stop()

{

::QueryPerformanceCounter(&m\_liStop);

}

// Returns the counter at the last Start()

LONGLONG GetStartCounter()

{

return m\_liStart.QuadPart;

}

// Returns the counter at the last Stop()

LONGLONG GetStopCounter()

{

return m\_liStop.QuadPart;

}

// Returns the interval between the last Start() and Stop()

LONGLONG GetElapsed()

{

return (m\_liStop.QuadPart - m\_liStart.QuadPart);

}

// Returns the interval between the last Start() and Stop() in seconds

double GetElapsedAsSeconds()

{

LARGE\_INTEGER liFrequency;

::QueryPerformanceFrequency(&liFrequency);

return ((double)GetElapsed() / (double)liFrequency.QuadPart);

}

};

class Array

{

private:

int\* arr;

int size;

public:

Array(int s)

{

size = s;

arr = new int[size];

//cout << "Successfully called\n";

srand(time(0));

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

{

arr[i] = rand()%1000;

}

}

void print()

{

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

cout << arr[i] << "\t";

}

int linearSearch(int);

int binarySearch();

void insertionSort();

void bubbleSort();

void selectionSort();

void mergeSortCall();

void quickSortCall();

void heapSortCall();

~Array()

{

delete arr;

}

};

void swap(int &a, int &b)

{

int temp = a;

a = b;

b = temp;

}

int Array::binarySearch()

{

int no;

sort(arr, arr + size);

cout<<"\nSorted Array List:";

print();

cout<<"\nEnter the element to be Searched:";

cin>>no;

int left = 0, right = size;

while (left <= right)

{

int mid;

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

if (no == arr[mid])

return mid+1;

if (arr[mid] < no)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

int Array::linearSearch(int x)

{

int i;

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

if (arr[i] == x)

return i+1;

return -1;

}

void Array::insertionSort()

{

for (int i = 1; i < size; i++)

{

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key)

{

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

print();

}

void Array::selectionSort()

{

int min\_index;

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

{

min\_index=i;

for (int j = i + 1; j < size; j++)

{

if (arr[j]<arr[min\_index])

{

min\_index = j;

}

}

if(i != min\_index)

{

swap(arr[i], arr[min\_index]);

}

}

print();

}

void Array::bubbleSort()

{

bool swapped = false;

for (int i = 1; i < size; i++)

{

//swapped = false;

for (int j = 0; j < size-i; j++)

{

if (arr[j] > arr[j + 1])

{

swap(arr[j], arr[j + 1]);

swapped = true;

}

}

if (swapped == false)

break;

}

print();

}

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

//int \*L,\*R;

//L = new int[n1];

//R = new int[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

i = 0; j = 0; k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

int mid = l +(r-l) / 2;

mergeSort(arr, l, mid);

mergeSort(arr, mid + 1, r);

merge(arr, l, mid, r);

}

}

void Array::mergeSortCall()

{

mergeSort(arr,0,size-1);

cout << "\n";

print();

}

int partition(int arr[], int low, int high)

{

int pivot = arr[high];

int pindex = low ;

for (int j = low; j < high ; j++)

{

if (arr[j] < pivot)

{

swap(arr[pindex], arr[j]);

pindex++;

}

}

swap(arr[pindex], arr[high]);

return pindex;

}

void quickSort(int arr[], int low, int high)

{

int pindex;

if (low < high)

{

pindex = partition(arr, low, high);

quickSort(arr, low, pindex - 1);

quickSort(arr, pindex + 1 , high);

}

}

void Array::quickSortCall()

{

quickSort(arr, 0, size-1);

cout << "\n";

print();

}

// To heapify a subtree rooted with node i which is

// an index in arr[]. n is size of heap

void heapify(int arr[], int n, int i)

{

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

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

largest = l;

// If right child is larger than largest so far

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

largest = r;

// If largest is not root

if (largest != i)

{

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// main function to do heap sort

void heapSort(int arr[], int n)

{

//int n = size;

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// One by one extract an element from heap

for (int i = n - 1; i >= 0; i--)

{

// Move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

void Array::heapSortCall()

{

heapSort(arr, size);

cout<<"\n";

print();

}

int main()

{

int findNo, result,oper;

char c;

int flag = 1;

CHRTimer ch;

do

{

Array arr(50);

cout<<"------------------------------------------------------------------------------------------------------------";

cout<<"\n\nActual randomly generated array:....\n";

arr.print();

cout << "\n\nOPERATIONS:....\n1.LINEAR\_SEARCH \n2.BINARY\_SEARCH\n3.INSERTION\_SORT\n4.BUBBLE\_SORT\n5.MERGE\_SORT\n6.SELECTION\_SORT\n7.QUICK\_SORT\n8.HEAP SORT\n9.Exit\n\nENTER YOUR OPERATION:";

cin >> oper;

switch (oper)

{

case LINEAR\_SEARCH: cout << "\nEnter the element to be searched: ";

cin >> findNo;

ch.Start();

result = arr.linearSearch(findNo);

ch.Stop();

(result == -1) ? cout << "Element is not present in array": cout << "Element is present at index " << result;

cout<<"\nTime taken for Linear Search:"<<ch.GetElapsedAsSeconds();

break;

case BINARY\_SEARCH: //sort(arr,arr+size);

//arr.print();

//cout << "\nEnter the element to be searched: ";

//cin >> findNo;

ch.Start();

result = arr.binarySearch();

ch.Stop();

(result == -1)?cout<<"Element is not present in array": cout << "Element is present at index "<< result;

cout<<"\nTime taken for Binary Search:"<<ch.GetElapsedAsSeconds();

break;

case INSERTION\_SORT: ch.Start();

arr.insertionSort();

ch.Stop();

cout<<"\nTime taken for Insertion Sort:"<<ch.GetElapsedAsSeconds();

break;

case BUBBLE\_SORT: ch.Start();

arr.bubbleSort();

ch.Stop();

cout<<"\nTime taken for Bubble Sort:"<<ch.GetElapsedAsSeconds();

break;

case MERGE\_SORT: ch.Start();

arr.mergeSortCall();

ch.Stop();

cout<<"\nTime taken for Merge Sort:"<<ch.GetElapsedAsSeconds();

break;

case SELECTION\_SORT: ch.Start();

arr.selectionSort();

ch.Stop();

cout<<"\nTime Taken forSelection Sort:"<<ch.GetElapsedAsSeconds();

break;

case QUICK\_SORT: ch.Start();

arr.quickSortCall();

ch.Stop();

cout<<"\nTime Taken for Quick Sort :"<<ch.GetElapsedAsSeconds();

break;

case HEAP\_SORT: ch.Start();

arr.heapSortCall();

ch.Stop();

cout<<"\nTime Taken for Heap Sort:"<<ch.GetElapsedAsSeconds();

break;

case EXIT: cout<<"\n\n-------------EXIT-------------";

//exit();

flag = 0;

break;

default : cout<<"You Entered Wrong Opertion Number..... ";

break;

}

if(flag == 1)

{

cout<<"\n\nDo you want perform other opertion(y/n): ";

cin>>c;

}

}

while(oper != 9 && c != 'n' && c != 'N');

//getch();

return 0;

}

**OUTPUT**

