



using System;

namespace LogicalPrograms

{

class Program

{

static public void MergeMethod(int[] numbers, int left, int mid, int right)

{

int[] temp = new int[25];

int i, left\_end, num\_elements, tmp\_pos;

left\_end = (mid - 1);

tmp\_pos = left;

num\_elements = (right - left + 1);

while ((left <= left\_end) && (mid <= right))

{

if (numbers[left] <= numbers[mid])

temp[tmp\_pos++] = numbers[left++];

else

temp[tmp\_pos++] = numbers[mid++];

}

while (left <= left\_end)

temp[tmp\_pos++] = numbers[left++];

while (mid <= right)

temp[tmp\_pos++] = numbers[mid++];

for (i = 0; i < num\_elements; i++)

{

numbers[right] = temp[right];

right--;

}

}

static public void SortMethod(int[] numbers, int left, int right)

{

int mid;

if (right > left)

{

mid = (right + left) / 2;

SortMethod(numbers, left, mid);

SortMethod(numbers, (mid + 1), right);

MergeMethod(numbers, left, (mid + 1), right);

}

}

static void Main(string[] args)

{

int[] numbers = { 38, 27, 43, 3, 9, 82, 10 };

int len = numbers.Length;

Console.WriteLine("Before Merge Sort:");

foreach(int item in numbers)

{

Console.Write(item + " ");

}

Console.WriteLine();

Console.WriteLine("After Merge Sort");

SortMethod(numbers, 0, len - 1);

foreach (int item in numbers)

{

Console.Write(item + " ");

}

Console.Read();

}

}

}

Quick Sort

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Quick\_Sort

{

class Program

{

// Method to perform Quick Sort on an array

private static void Quick\_Sort(int[] arr, int left, int right)

{

// Check if there are elements to sort

if (left < right)

{

// Find the pivot index

int pivot = Partition(arr, left, right);

// Recursively sort elements on the left and right of the pivot

if (pivot > 1) {

Quick\_Sort(arr, left, pivot - 1);

}

if (pivot + 1 < right) {

Quick\_Sort(arr, pivot + 1, right);

}

}

}

// Method to partition the array

private static int Partition(int[] arr, int left, int right)

{

// Select the pivot element

int pivot = arr[left];

// Continue until left and right pointers meet

while (true)

{

// Move left pointer until a value greater than or equal to pivot is found

while (arr[left] < pivot)

{

left++;

}

// Move right pointer until a value less than or equal to pivot is found

while (arr[right] > pivot)

{

right--;

}

// If left pointer is still smaller than right pointer, swap elements

if (left < right)

{

if (arr[left] == arr[right]) return right;

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

}

else

{

// Return the right pointer indicating the partitioning position

return right;

}

}

}

static void Main(string[] args)

{

int[] arr = new int[] { 2, 5, -4, 11, 0, 18, 22, 67, 51, 6 };

Console.WriteLine("Original array : ");

foreach (var item in arr)

{

Console.Write(" " + item);

}

Console.WriteLine();

// Call Quick Sort to sort the array

Quick\_Sort(arr, 0, arr.Length-1);

Console.WriteLine();

Console.WriteLine("Sorted array : ");

foreach (var item in arr)

{

Console.Write(" " + item);

}

Console.WriteLine();

}

}

}

--

Heap

using System;

namespace Heap\_sort

{

public class MainClass

{

public static void Main(string[] args)

{

int[] mykeys = new int[] { 2, 5, -4, 11, 0, 18, 22, 67, 51, 6 };

// Uncomment and use the following array types for testing:

// double[] mykeys = new double[] {2.22, 0.5, 2.7, -1.0, 11.2};

// string[] mykeys = new string[] {"Red", "White", "Black", "Green", "Orange"};

Console.WriteLine("\nOriginal Array Elements :");

printArray(mykeys); // Display the original array elements

heapSort(mykeys); // Sort the array using Heap Sort

Console.WriteLine("\n\nSorted Array Elements :");

printArray(mykeys); // Display the sorted array elements

Console.WriteLine("\n");

}

// Method to perform Heap Sort on an array of type T, which implements IComparable

private static void heapSort<T>(T[] array) where T : IComparable<T>

{

int heapSize = array.Length;

buildMaxHeap(array); // Build the max heap

// Sorting by extracting elements from the heap

for (int i = heapSize - 1; i >= 1; i--)

{

swap(array, i, 0); // Swap the first and last elements

heapSize--;

sink(array, heapSize, 0); // Restore the heap property

}

}

// Method to build the max heap

private static void buildMaxHeap<T>(T[] array) where T : IComparable<T>

{

int heapSize = array.Length;

// Iterate over half the array elements and sink each one

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

{

sink(array, heapSize, i); // Sink the element to maintain the heap property

}

}

// Method to maintain the heap property

private static void sink<T>(T[] array, int heapSize, int toSinkPos) where T : IComparable<T>

{

if (getLeftKidPos(toSinkPos) >= heapSize)

{

// No left kid => no kid at all

return;

}

int largestKidPos;

bool leftIsLargest;

// Determine the largest kid

if (getRightKidPos(toSinkPos) >= heapSize || array[getRightKidPos(toSinkPos)].CompareTo(array[getLeftKidPos(toSinkPos)]) < 0)

{

largestKidPos = getLeftKidPos(toSinkPos);

leftIsLargest = true;

}

else

{

largestKidPos = getRightKidPos(toSinkPos);

leftIsLargest = false;

}

// Swap the elements if necessary and sink recursively

if (array[largestKidPos].CompareTo(array[toSinkPos]) > 0)

{

swap(array, toSinkPos, largestKidPos);

if (leftIsLargest)

{

sink(array, heapSize, getLeftKidPos(toSinkPos));

}

else

{

sink(array, heapSize, getRightKidPos(toSinkPos));

}

}

}

// Method to swap two elements in the array

private static void swap<T>(T[] array, int pos0, int pos1)

{

T tmpVal = array[pos0];

array[pos0] = array[pos1];

array[pos1] = tmpVal;

}

// Method to get the index of the left child of a parent node

private static int getLeftKidPos(int parentPos)

{

return (2 \* (parentPos + 1)) - 1;

}

// Method to get the index of the right child of a parent node

private static int getRightKidPos(int parentPos)

{

return 2 \* (parentPos + 1);

}

// Method to print the array elements

private static void printArray<T>(T[] array)

{

foreach (T t in array)

{

Console.Write(' ' + t.ToString() + ' ');

}

}

}

}