# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 08 : Merge and Quick Sort Algorithms**

**Date: 23th November, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# 

# Lab 08 : Merge and Quick Sort Algorithms

**Introduction**

In this lab, you will implement merge and quick sort algorithms that have already been discussed in the class.

**Objectives**

Objective of this lab is to implement Merge & Quick Sort.

**Tools/Software Requirement**

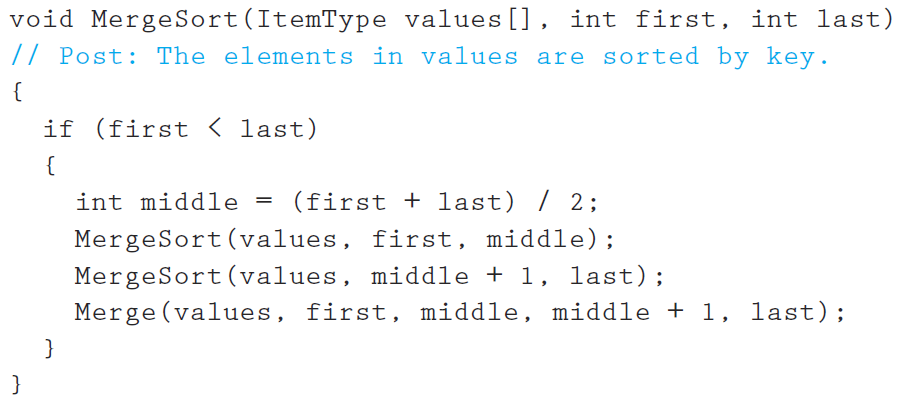
Visual Studio C++

**Description**

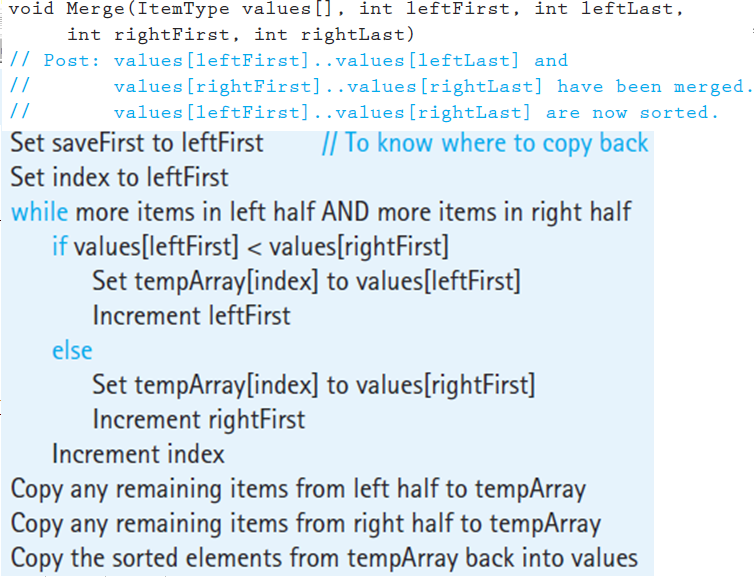
You will implement a version of the pseudo codes of merge and quick sort algorithms given in the book "Algorithms and Data Structures using C++" by Nell Dale.

**Merge Sort:**

Its best and the worst case time complexities are of order nlogn. Unlike insertion sort, it is not an in-place sorting algorithm as it requires a temporary array of size n to sort values.. The pseudo code for merge sort is shown below:



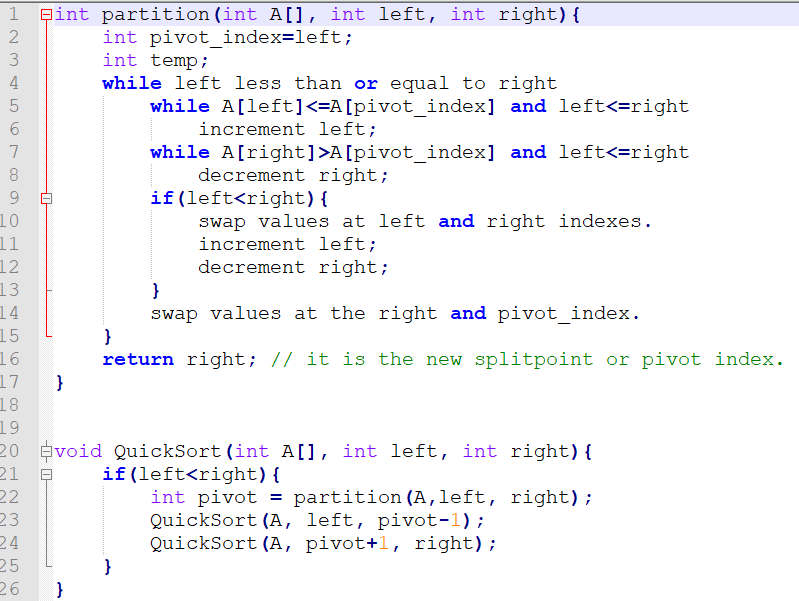
Make sure you print the updated array after a call to the Merge function terminates.

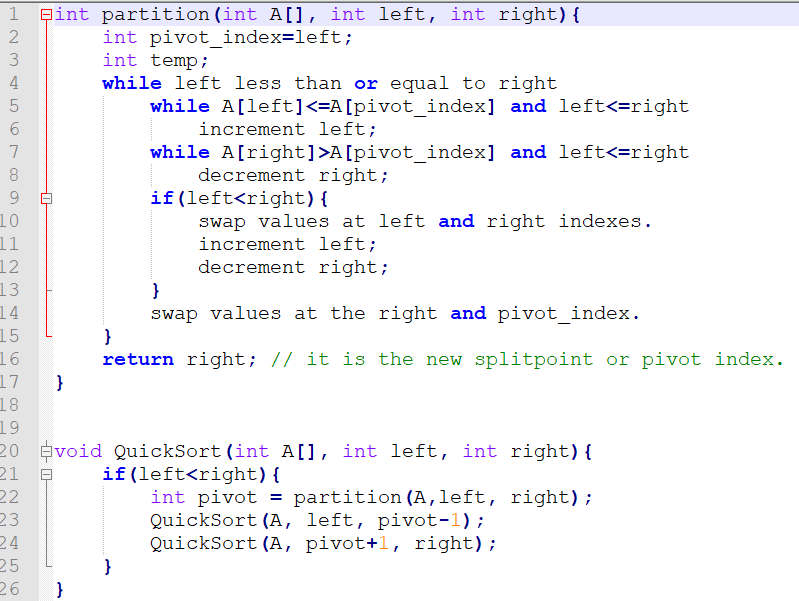
****

**Quick Sort:**It is also a divide-conquer based 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.

Always pick first element as pivot.

* Always pick last element as pivot (implemented below)
* Pick a random element as pivot.
* 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. Once, done quicksort the left partition and quicksort the right partition.  
  
****

****

**Lab Tasks**

1. You will run the algorithms on data that we used in the class to verify the results.

Code:

// C++ program for Merge Sort

#include <iostream>

using namespace std;

int arr\_size = 6;

// Function to print an array

void printArray(int Arry[], int size)

{

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

cout << Arry[i] << " ";

}

// Merges two subarrays of array[].

// First subarray is arr[begin..mid]

// Second subarray is arr[mid+1..end]

void merge(int array[], int left, int mid, int right)

{

//cout << "A" << endl;

int subArrayOne = mid - left + 1;

int subArrayTwo = right - mid;

// Create temp arrays

int\* leftArray = new int[subArrayOne];

int\* rightArray = new int[subArrayTwo];

// Copy data to temp arrays leftArray[] and rightArray[]

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

leftArray[i] = array[left + i];

for (int j = 0; j < subArrayTwo; j++)

rightArray[j] = array[mid + 1 + j];

int indexOfArrayOne = 0, // Initial index of first sub-array

indexOfArrayTwo = 0; // Initial index of second sub-array

int index = left; // Initial index of merged array

// Merge the temp arrays back into array[left..right]

while (indexOfArrayOne < subArrayOne && indexOfArrayTwo < subArrayTwo)

{

if (leftArray[indexOfArrayOne] <= rightArray[indexOfArrayTwo]) {

array[index] = leftArray[indexOfArrayOne];

indexOfArrayOne++;

}

else {

array[index] = rightArray[indexOfArrayTwo];

indexOfArrayTwo++;

}

index++;

}

// Copy the remaining elements of

// left[], if there are any

while (indexOfArrayOne < subArrayOne) {

array[index] = leftArray[indexOfArrayOne];

indexOfArrayOne++;

index++;

}

// Copy the remaining elements of

// right[], if there are any

while (indexOfArrayTwo < subArrayTwo) {

array[index] = rightArray[indexOfArrayTwo];

indexOfArrayTwo++;

index++;

}

printArray(array, arr\_size); cout << endl;

// cout << "A" << endl;

}

// begin is for left index and end is

// right index of the sub-array

// of arr to be sorted \*/

void mergeSort(int array[], int first, int last)

{

// cout << "A" << endl;

if (first >= last)

return; // Returns recursively

int mid = first + (last - first) / 2;

mergeSort(array, first, mid);

mergeSort(array, mid + 1, last);

merge(array, first, mid, last);

// cout << "A" << endl;

}

int main()

{

int arr[] = { 12, 11, 13, 5, 6, 7 };

cout << "Given array is \n";

printArray(arr, arr\_size); cout << endl;

mergeSort(arr, 0, arr\_size - 1);

cout << endl << "Sorted array is \n";

printArray(arr, arr\_size);

cout << endl;

cout << endl;

return 0;

}

Merge Sort:

Text

Description automatically generated

Code:

#include <iostream>

using namespace std;

void swap(int\* t1, int\* t2)

{

int t = \*t1;

\*t1 = \*t2;

\*t2 = t;

}

int partition(int arr[], int left, int right)

{

int pivot = arr[right]; // pivot

int i = (left - 1); // Index of smaller element and indicates the right position of pivot found

for (int j = left; j <= right - 1; j++)

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[right]);

return (i + 1);

}

/\* The main function that implements QuickSort

arr[] --> Array to be sorted,

left --> Starting index,

right --> Ending index \*/

void quickSort(int arr[], int left, int right)

{

if (left < right)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

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

// Separately sort elements before

// partition and after partition

quickSort(arr, left, pivot - 1);

quickSort(arr, pivot + 1, right);

}

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

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

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

cout << endl;

}

int main()

{

int arr[] = { 10, 7, 8, 9, 1, 5 };

printArray(arr, 6);

int n = 6;

quickSort(arr, 0, n - 1);

cout << "Sorted array: \n";

printArray(arr, n);

return 0;

}

Quick Sort:

Text

Description automatically generated

1. Compare how many partitions happened in each case.

Answer: In merge sort for 6 elements the number of partitions was 6 times. Each half was partitioned 3 times until it reached individual elements.

1. To understand which calls to the recursive Merge sort and the merge functions are made, include print statements in the first line and before the closing parenthesis of the merge sort function. Do the same for the merge function as well. Moreover, print the updated array-list after a call to the merge() function in the merge sort function.

Answer: If we count the number of As in the screenshot, there are 16 As for 6 members in the array. Therefore, the general formula for n array elements will be 3n-2 for all positive n values.

Text

Description automatically generated with low confidence

1. To understand which calls to the recursive Quicksort and the Partition functions are made, include print statements in the first line and before the closing parenthesis of the Quicksort function. Do the same for the partition function as well. Print the updated array after line 22 i.e., when a new split has been created after sorting the pivot value.

Answer: when we look at the number of As, we come to know that there 13 calls of both functions in total, when n is equal to 6.

Text

Description automatically generated

1. Choose a different pivot value and see if number of partitions decreases. The new pivot value can be chosen as the median of the first, middle and last elements of the array.

Answer: if we choose the first and the last elements of the array as pivots the number of calls of both functions is less than the number of calls when the pivot is chosen other than the first and the last elements. When pivot is chosen in middle, more function calls are required.