CMPSC 122 Lab 18 Report （revised on Apr 30, 2014）

Code

// Programmer: Yizhou Wang

// Section: 2

// Lab: 18

// Date: Apr. 23, 2014

// Description: Implementation of merge sort

#include <iostream>

using namespace std;

int const MAX\_SIZE = 50; //the maximal physical size of all lists in this lab

void MERGE(int L[], int l\_size, int R[], int r\_size, int A[])

//PRE: l\_size > 0, r\_size > 0, l\_size <= MAX\_SIZE, r\_size <= MAX\_SIZE,

// L[0..l\_size-1] is sorted in ascending order,

// R[0..r\_size-1] is sorted in ascending order,

// A[] is allocated to a size at least l\_size+r\_size

//POST: A[0..(l\_size+r\_size-1)] contains all elements of L[0..l\_size-1] and R[0..r\_size-1] sorted in

// ascending order

{

//DATA DICTIONARY

int l\_ctr; //a counter of L[0..l\_size-1]

int r\_ctr; //a counter of R[0..r\_size-1]

int a\_size; //the logical size of A[]

l\_ctr = 0;

r\_ctr = 0;

a\_size = 0;

//when neither of l\_ctr and r\_ctr has surpassed the end of their respective list, keep comparing

// elements from the two lists

while (l\_ctr < l\_size && r\_ctr < r\_size)

{

if (L[l\_ctr] <= R[r\_ctr]) //if L[l\_ctr] is smaller than or equal to R[r\_ctr],

// copy L[l\_ctr] to A[a\_size]

{

A[a\_size] = L[l\_ctr];

l\_ctr ++;

}

else //if L[l\_ctr] is greater than R[r\_ctr], copy R[r\_ctr]

// to A[a\_size]

{

A[a\_size] = R[r\_ctr];

r\_ctr ++;

}

a\_size ++;

}

//when either of l\_ctr and r\_ctr has surpassed the end of their respective list, copy the rest

// elements of the other list to A[]

if (l\_ctr < l\_size) //if L[0..l\_size-1] has element(s) that haven't

// been copied to A[], copy them to A[]

{

for (int i=l\_ctr; i<l\_size; i++) //copy the elemnt(s) of L[l\_ctr..l\_size-1] to

// the end of A[]

{

A[a\_size] = L[i];

a\_size ++;

}

}

else //if R[0..l\_size-1] has element(s) that haven't

// been copied to A[], copy them to A[]

{

for (int i=r\_ctr; i<r\_size; i++) //copy the elemnt(s) of R[r\_ctr..r\_size-1] to

// the end of A[]

{

A[a\_size] = R[i];

a\_size ++;

}

}

}

void MERGE\_SORT(int A[], int p, int q)

//PRE: p >= 0, q >= 0, q-p+1 <= MAX\_SIZE, A[p..q] is initialized

//POST: A[p..q] contains all elements from A[p..q] at the time of invocation and is sorted

// ascendingly

{

//DATA DICTIONARY

int m; //a variable storing the midpoint of A[p..q]

int \* left; //a pointer pointing to the left subarray

int \* right; //a pointer pointing to the right subarray

if (p < q) //if there exists element in A[p..q]

{

m = (p+q)/2; //calculate the midpoint and asign it to m

left = new int [m-p+1]; //allocate the subarray pointed to by left

right = new int [q-(m+1)+1]; //allocate the subarray pointed to by right

for (int i=0; i < m-p+1; i++) //copy A[p..m] to left[0..m-p]

{

left[i] = A[p+i];

}

for (int i=0; i < q-(m+1)+1; i++) //copy A[m+1..q] to right[0..q-(m+1)]

{

right[i] = A[m+1+i];

}

MERGE\_SORT(left, 0, m);

MERGE\_SORT(right, 0, q-(m+1));

MERGE(left, m-p+1, right, q-(m+1)+1, A);

}

}

void traversal(int A[], int n)

//PRE: n > 0, n-1 <= MAX\_SIZE, A[0..n-1] is initialized

//POST: all the elements of A[0..n-1] are printed to the console in a sequential order

{

for (int i = 0; i < n; i ++) //traverse the array A[0..n-1] and print

// elements spaced by a whitespace

{

cout << A[i] << " ";

}

cout << endl;

}

int main()

{

//DATA DICTIONARY

int array1[8];

int array2[4];

int array3[15];

//initialization

array1[0] = 3;

array1[1] = 0;

array1[2] = 7;

array1[3] = 9;

array1[4] = 25;

array1[5] = 42;

array1[6] = 101;

array1[7] = 99;

array2[0] = 5;

array2[1] = 10;

array2[2] = 35;

array2[3] = 50;

array3[0] = 500;

array3[1] = 400;

array3[2] = 300;

array3[3] = 200;

array3[4] = 100;

//Test1

cout << "\*\*\*Test1: The original array1 (mixed order) is:" << endl;

traversal(array1, 8);

MERGE\_SORT(array1, 0, 7);

cout << "After merge sort, array1 turns into:" << endl;

traversal(array1, 8);

cout << endl;

//Test2

cout << "\*\*\*Test2: The original array2 (ascending order) is:" << endl;

traversal(array2, 4);

MERGE\_SORT(array2, 0, 3);

cout << "After merge sort, array2 turns into:" << endl;

traversal(array2, 4);

cout << endl;

//Test3

cout << "\*\*\*Test3: The original array3 (descending order) is:" << endl;

traversal(array3, 5);

MERGE\_SORT(array3, 0, 4);

cout << "After merge sort, array3 turns into:" << endl;

traversal(array3, 5);

cout << endl;

return 0;

}

Sample Runs

\*\*\*Test1: The original array1 (mixed order) is:

3 0 7 9 25 42 101 99

After merge sort, array1 turns into:

0 3 7 9 25 42 99 101

\*\*\*Test2: The original array2 (ascending order) is:

5 10 35 50

After merge sort, array2 turns into:

5 10 35 50

\*\*\*Test3: The original array3 (descending order) is:

500 400 300 200 100

After merge sort, array3 turns into:

100 200 300 400 500

Discussion

In this lab, the tricky parts include the usage of pointers to pass subarrays and the switch from 1-based array to 0-based array.