Câu hỏi 1

Chính xác

Điểm 1,00 của 1,00

Cờ câu hỏi

Implement static methods **Merge** and **MergeSort** in class Sorting to sort an array in ascending order. The Merge method has already been defined a call to method printArray so you do not have to call this method again to print your array.

```
#ifndef SORTING_H
#define SORTING_H
#include <iostream>
using namespace std;
template <class T>
class Sorting {
public:
    /* Function to print an array */
    static void printArray(T *start, T *end)
    {
        long size = end - start + 1;
        for (int i = 0; i < size - 1; i++)
            cout << start[i] << ", ";
        cout << start[size - 1];
        cout << endl;
    }

    static void merge(T* left, T* middle, T* right){
        /*TODO*/
        Sorting::printArray(left, right);
    }
    static void mergeSort(T* start, T* end) {
       /*TODO*/
    }
};
#endif /* SORTING_H */
```

For example:

Test	Result

`int arr[] = {0,2,4,3,1,4};`
`Sorting<int>::mergeSort(&arr[0], &arr[5]);`
	`0, 2`
	`0, 2, 4`
	`1, 3`
	`1, 3, 4`
	`0, 1, 2, 3, 4, 4`

`int arr[] = {1};` `Sorting<int>::mergeSort(&arr[0], &arr[0]);`	

Câu hỏi **2**

Chính xác

Điểm 1,00 của 1,00

Cờ câu hỏi

The best way to sort a singly linked list given the head pointer is probably using merge sort.

Both Merge sort and Insertion sort can be used for linked lists. The slow random-access performance of a linked list makes other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible. Since worst case time complexity of Merge Sort is O(nLogn) and Insertion sort is O(n^2), merge sort is preferred.

Additionally, Merge Sort for linked list only requires a small constant amount of auxiliary storage.

To gain a deeper understanding about Merge sort on linked lists, let's implement **mergeLists** and **mergeSortList** function below

Constraints:

0 <= list.length <= 10^4
0 <= node.val <= 10^6

Use the nodes in the original list and don't modify ListNode's val attribute.

```
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int _val = 0, ListNode* _next = nullptr) : val(_val), next(_next) { }
};

// Merge two sorted lists
ListNode* mergeSortList(ListNode* head);

// Sort an unsorted list given its head pointer
ListNode* mergeSortList(ListNode* head);
```

For example:

Test	Input	Result

int arr1[] = {1, 3, 5, 7, 9};
int arr2[] = {2, 4, 6, 8};
unordered_map<ListNode*, int> nodeAddr;
ListNode* a = init(arr1, sizeof(arr1) / 4, nodeAddr);
ListNode* b = init(arr2, sizeof(arr2) / 4, nodeAddr);
ListNode* merged = mergeLists(a, b);
try {
    printList(merged, nodeAddr);
}
catch(char const* err) {
    cout << err << '\n';
		1 2 3 4 5 6 7 8 9

Test	Input	Result

int arr1[] = {1, 3, 5, 7, 9};
int arr2[] = {2, 4, 6, 8};
unordered_map<ListNode*, int> nodeAddr;
ListNode* a = init(arr1, sizeof(arr1) / 4, nodeAddr);
ListNode* b = init(arr2, sizeof(arr2) / 4, nodeAddr);
ListNode* merged = mergeLists(a, b);
try {
    printList(merged, nodeAddr);
}
catch(char const* err) {
    cout << err << '\n';
}
freeMem(merged);
		1 2 3 4 5 6 7 8 9

int size;
cin >> size;
int* array = new int[size];
for(int i = 0; i < size; i++) cin >> array[i];
unordered_map<ListNode*, int> nodeAddr;
ListNode* head = init(array, size, nodeAddr);
ListNode* sorted = mergeSortList(head);
try {
    printList(sorted, nodeAddr);
}
catch(char const* err) {
    cout << err << '\n';
}
freeMem(sorted);
delete[] array;
	9
	9 3 8 2 1 6 7 4 5
		1 2 3 4 5 6 7 8 9

Answer: (penalty regime: 0 %)

Reset answer