Informatics II, Spring 2024, Exercise 5

Publication of exercise: March 18, 2024
Publication of solution: March 25, 2024
Exercise classes: April 8 - April 12, 2024

Learning Goal

- Understanding heaps and how algorithms can operate on them.
- Knowing how to implement quick sort and where this algorithm has difficulties.
- Being able to apply learned algorithms to unknown problems.

Task 1 [Medium]

1.1: In the lecture you learned what the criteria for a heap are. Use this knowledge to write the function `controlMaxHeap(int array[], int currentIndex, int length)` in C. The goal of this function is to return -1 if the array fulfills all the criteria that a max heap must fulfill. Otherwise the index of an element not fulfilling the property should be returned.

Examples:

- $[5, 5, 3, 2, 1, 3]$ should return -1, since no property is violated.
- $[5, 5, 3, 2, 1, 4]$ should return 2, because 4 can't be a child of 3. Since 2 is the index of the element with value 3 the number 2 should be returned.
- $[5, 5, 3, 2, 1, 6]$ should return 2, even if there is a conflict between the number 5 at index 0 and the number 6 at index 5, the direct violation of a property is the same as in the previous example.

1.2: Your task is to implement the function `heapify(int array[], int currentIndex, int length)` in C. The function should restore the heap property for the desired index and propagate downwards if necessary.

Example:

- When calling heapify with the array [4, 10, 3, 5, 1] of length 5 and the index 0, the function should rearrange the array to [10, 5, 3, 4, 1].

1.3: Suppose we have an array in which exactly one element does not fulfill the heap property. Write the function `fixHeap(int array[], int length)` in C which ensures that the array is a heap again. Use the algorithms from the previous sub tasks to find the element that causes difficulties and fix it.

Task 2 [Medium]

2.1: In the lecture you learned about the hoare partition. Write the function `hoarePartition(int array[], int leftIndex, int rightIndex)` in the manner of the learned algorithm. Also use the rightmost element as the pivot element.

Example:

- $[2, 6, 4, 1, 5, 3]$ should return 2 and the array should be [2, 3, 1, 4, 5, 6].

2.2: Use the previously programmed partition to implement the function `quickSort(int array[], int leftIndex, int rightIndex)`.

2.3: Modify your quick sort program so that you can determine how often the innermost loops of the hoare partition have been run. Add up the iterations of both loops across all function calls. To do this, you can use a global variable, add an argument to the function with a pointer to the counter (if you already know pointers) or come up with a method of your own. Run this modified algorithm with the following two arrays.

- $[6, 8, 4, 5, 3, 7, 2, 1, 0, 9]$
- $[1, 9, 0, 6, 3, 8, 7, 2, 4, 5]$

What do you notice regarding the counter and how do you explain what you observe?

2.4: Try to modify the array from the last task by rearranging the elements such that the count of the innermost loops is as high as possible. For which input does the quick sort take the longest?

2.5 Bonus: Do you have any ideas on how to reduce the risk of such a worst-case scenario?

Task 3 [Hard]

The median is the middle value in a set of numbers when they are arranged in numerical order, or the average of the two middle values if there is an even number of elements.

Write a C function `findMedian(int array[], int n)` that takes an array of integers and its length as parameters and returns the median of the array as a float. Your function should handle both even and odd-length arrays. However, you can assume that no number appears twice in the array.

Example:

- For the array [5, 2, 8, 3, 7], the function should return 5.0
- For the array [5, 2, 8, 3, 7, 4], the function should return 4.5

Note: The easiest way to solve this task is to sort the array first and then access the center. However, you can also find the median faster. For this it helps to consider whether the whole array actually has to be sorted. Try to find such a solution that reduces unnecessary sorting effort.