|  |  |
| --- | --- |
| **ALG** 3.2. | *Write your solutions in this sheet below the problem statement. Mark clearly your solutions by corresponding letters A, B, C, D and separate visually each solution from the other ones by empty space or by a line. Use the other side of the sheet or ask for an additional blank sheet if necessary.***eALG** *Each problem 1. – 4. is worth 0 – 4 points, each answer to a particular question A, B, C, D contributes at most 1 point to the total.* |

1. AVL tree

A. Insert the keys given below, one by one, into an originally empty AVL tree T. Determine the total number of single and double rotations which will be performed during this process. Explain your reasoning. Draw the resulting tree T.

Keys: 28 1 18 15 5 9 26 29 22 6

There will be 4 double rotations, namely after insertion of 18, 5, 9, 26. No single rotations. The resulting tree:

 \_\_\_\_\_\_\_\_\_15\_\_\_\_\_\_

 \_\_\_\_5\_\_\_\_ \_\_\_\_26\_\_\_

 1 \_\_9 18\_\_ 28\_\_

 6 22 29

B. From tree T created in problem A., remove, using Delete operation, the key in the root of T. Then again, for the second time, remove the key in the root using Delete operation. Draw T after each of these two Delete operation.

Suppose we replace the deleted key in the root by the smallest key in the right subtree of the root.

 \_\_\_\_\_\_\_\_\_18\_\_\_\_\_\_

 \_\_\_\_5\_\_\_\_ \_\_\_\_26\_\_\_

 1 \_\_9 22 28\_\_

 6 29

 \_\_\_\_\_\_\_\_\_22\_\_\_\_\_\_

 \_\_\_\_5\_\_\_\_ \_\_\_\_28\_\_\_

 1 \_\_9 26 29

 6

L rotation was applied in 26 after moving 22 to the root.

C. When a key is removed from an AVL tree, more than one single or one double rotations may be applied in the process of one Delete operation. Explain why this effect occurs and draw an example in which two single rotations are applied in one Delete operation in an AVL tree.

The deleted node may be in a relatively short branch, which may become even shorter after the first rotation caused by removal of the node. After that, compared to other branches, this shortened branch may be again too short. Example

 \_\_\_\_\_\_\_\_\_[A]\_\_\_\_\_\_\_\_

 \_\_\_\_\_\_[O]\_\_\_\_\_ \_\_\_[B]\_\_\_

 \_\_\_[O]\_\_\_ \_\_[O] [X] [C]\_\_\_

 \_\_\_[O] [O] [O] [O]

 [O]

Delete [X]. This forces L rotation in [B], and [C] becomes the root of the right subtree of [A]. Now the height of the left subtree of [A] is still 3, but the height of the right subtree of A is only 1. Another rotation, namely R rotation in A is needed to fix the unbalance.

D. AVL tree T contains *2n* keys. There are exactly *n* keys in T which value is even and exactly *n* keys which value is odd. We delete all odd valued keys from T by repeated application of Delete operation. Determine the maximum number of rotations which will be performed in T during this process. Express the result using O/ Θ / Ω notation and explain your calculations.

We calculate a relatively easy upper bound for the number of rotations. There will be *n* deletions. Each deletion happens in a tree which number of nodes is in range [*n*+1, 2*n*]. The number of rotations in one delete may be, in the worst case, proportional to the depth of the tree, which is at most ⎣log 2*n*⎦. The number of all rotations would not exceed *n* \* ⎣log 2*n*⎦. The exact number is difficult to calculate, after some number of rotations the tree becomes favourably balanced, so that the number of rotations in next few delete operations is expected to get smaller and without more complex calculations it is not obvious how it will influence the formula *n* \* ⎣log 2*n*⎦. Therefore, we can conclude, the number of rotations will be in

the class O(*n* log(*n*)).

 2.

**eALG**

We are given a set M of 3 items. The weights of items are 2, 4, and 7 kg. The values of the items, in the same order, are 30, 50 and 40 value units. Also, we are given a container which capacity is 18 kg. We have to fill the container with the items, the total weight of the items in the container cannot exceed the container capacity. Any item in the container may appear more times. We have to maximize the total value of items in the container.

A. Write a formula which defines the relation between the values in a dynamic programming table used for a solution of the given problem. Also, define the values in the first row and the first column of the table, if necessary. Describe the meaning of particular symbols in the formula.

T[c] = max( T[c−weight[item1]] + value[item1] , T[c−weight[item2]] + value[item2], T[c−weight[item3]] + value[item3] ),

if (c−weight[item\_i]) is negative, assume T[c−weight[item\_i]] + value[item\_i] = 0.

B. Draw a dynamic programming table and apply the formula produced in A. to fill the entire table with values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 0 | 0 | 30 | 30 | 60 | 60 | 90 | 90 | 120 | 120 | 150 | 150 | 180 | 180 | 210 | 210 | 240 | 240 | 270 |

C. The presented problem can be interpreted as a search for an optimal path in a directed acyclic graph. Explain what is represented by the nodes and the edges of the graph and explain how edge weights (values) are determined.

The nodes are (correspond to) all partial capacities [0, 1 …, 18]. Each edge exists between two partial capacities which difference is equal to the weight of some item and which value is equal to the value of that item.

D. Suppose that in a unlimited knapsack problem, N items are given, the weight and the cost (value, price) of any item are expressed by a positive three digit integer. The capacity of the knapsack is expressed as an integer occupying exactly *k* bits. Determine the asymptotic complexity of the solution of this instance of the knapsack problem. The asymptotic complexity will depend on the values of N and *k*. Explain your reasoning.

Filling each cell in the table is proportional to number N of items. The capacity is expressed with *k* bits, therefore its maximum value can be 2*k*−1. Thus, there would be about 2*k* table cells to fill, which in total yields a number proportional

To N \* 2*k* .

3.

**ALG B**

Graph G, given at the picture, contains 13 nodes labelled 1, 2, 3, …, 13. Depth-first search (DFS) is applied on G. It respects also respects an additional rule: Each time the algorithm decides where to expand the search to a next node (=open another node), it chooses that neighbour of a current node which label is the smallest among all available neighbours.

A. The search starts in node 13. Write the order in which the nodes will be open (=discovered) during the search.

13 8 7 2 1 3 4 5 6 9 10 11 12

B. The search starts in node 13. Write the order in which the nodes will be closed during the search.

3 4 5 6 1 2 9 10 11 12 7 8 13

C. The search produces a so-called search tree. The depth of this tree may vary depending on the choice of the start node of the search. Choose the start node so that the depth of the search tree is minimum possible. The search respects the additional rule above. Draw the chosen search tree.

 7

 / \

 2 8

 / \

 \_\_\_\_ 1 13 \_\_\_\_\_

 | / | \ / | \ |

 2 3 4 5 9 10 11 12

D. Suppose a general undirected graph G with *n* nodes is subjected to Depth-first search algorithm. The nodes of the graph are labelled 1, 2, 3, …, *n*. Suppose also, that the time spent on opening and closing a node is proportional to the value of the label of that node. Determine the asymptotic complexity of the search. Explain how did you derived the complexity.

The time spent on processing node with label k is proportional to 2\*k (open and close), thus, totalling all times, we arrive to the sum

const\*(1+1) + const\*(2+2) + const\*(3+3) + … + const\*(*n*+*n*) = const\*(2 + 4 + 6 + … + 2*n*) = const \* *n*\*(*n*+1),

using the formula for summation of arithmetic progression.

So, the total time of the algorithm is const \* *n*\*(*n*+1) + (anotherconstant)\*(*n* + *m*), where *m* is the number of edges in the graph. This is because the time (anotherconstant)\*(*n* + *m*) is spent in *any* DFS algorithm, which has to traverse all *n* nodes and all *m* edges.

In any graph, it holds *m* ≤ (*n* −1) \**n* /2. Thus, we can state:

 (anotherconstant)\*(*n* + *m*) ≤ (anotherconstant)\*(*n* + (*n* −1)\**n*/2) = (anotherconstant)\* (*n* +1) \**n* / 2.

Thus, the value of the second term (anotherconstant)\*(*n* + *m*) is either proportional or even smaller than the value of the first term const \* *n*\*(*n*+1). Therefore, even in the worst case, when there ara many edges in the graph, the running time will depend primarily on the value of the first term. With this property in mind, we can simplify the original expression

 const \* *n*\*(*n*+1) + (anotherconstant)\*(*n* + *m*),

 to mere

 (yetanotherconstant) \* *n*\*(*n*+1).

In words, the complexity of the search will be proportional to the value *n*\*(*n*+1), or even more simply, to the value of *n*2.

4.

**eALG**

Radix sort is applied on an input list P of strings. List P contains all possible strings of length 10 which are composed of characters A, B, C, D. No other characters, spaces, etc. appear in any of the string in P. All strings in P are pairwise different.

The strings in P are originally stored in lexicographically descending order, that is, the first string is DDDDDDDDDD, the next string is DDDDDDDDDC, and so on, the last string in P is AAAAAAAAAA.

The output of the radix sort is a list sorted in increasing lexicographical order.

Radix sort proceeds in steps, in each step it distributes all values, which are being sorted, into separate lists which contents are then used in the subsequent step.

A. Write down the first and last item in each separate list (not the input list) at the end of the first step of Radix sort.

ListA :[ DDDDDDDDD**A**, …, AAAAAAAAA**A** ]

ListB :[ DDDDDDDDD**B**, …, AAAAAAAAA**B** ]

ListC :[ DDDDDDDDD**C**, …, AAAAAAAAA**C** ]

ListD :[ DDDDDDDDD**D**, …, AAAAAAAAA**D** ]

B. Write down the first and last item in each separate list (not the input list) at the end of the penultimate step of Radix sort. The penultimate step is the step which is performed immediately before the very last step of Radix sort.

ListA :[ D**A**AAAAAAAA, …, A**A**DDDDDDDD ]

ListB :[ D**B**AAAAAAAA, …, A**B**DDDDDDDD ]

ListC :[ D**C**AAAAAAAA, …, A**C**DDDDDDDD ]

ListD :[ D**D**AAAAAAAA, …, A**D**DDDDDDDD ]

(Characters at current active position are printed in bold.)

C. In another variant of the same problem, Radix sort is processing an input list of strings where the length of the each string is 20. All other parameters of the problem are unchanged. Suppose that the solution of the original problem takes time T1. The variant of the problem is run in the same SW/HW environment and its solution takes time T2. Determine the factor by which T2 is bigger than T1. Explain your reasoning.

Each character in the list is processed (=checked, used for sorting) only once. Thus, the complexity, or the time of a solution, is proportional to the length of the list multiplied by the length of each string. In the first case it is const\*10\*410, in the second case it is const\*20\*420. The factor is then equal to 2\*410, or, expressed in base 2, 221 = 2097152.

D. Radix sort is known to be a stable sort, which means that it does not change relative order of items with the same value during the sorting process. Explain why Radix sort is stable.

In each step of the sort, the relative position of two identical items is not changed when they are transferred to the new list.

Because that happens in each step, the relative position of these two identical items in the final (output) list is the same as it was in the input list.