Khuyen Le Thi Minh s5128 – ASD 5

TASK DESCRIPTION:

D-heap is a heap-like structure where every node can have at most d successors. Other properties of standard heap structure are sustained (i.e. tree shape, labels' partial order). Find a method which allows to represent, in an effective way, the d-heap in an array. With this approach design a DelMin algorithm for the d-heap. Prove the algorithm correctness and estimate its complexity.

ALGORITHM:

We will represent D-heap in an array (where the index number start from 0 and the last index with n elements array would be "n-1"), where we can find the position of element in the arrays by the following formulas:

- *The parent position for node with position "i" can be calculate by formula: $P(i) = (i-1)/4$*
- *The position i-position child (where $i \in d$ and $i > 0$) of parent P(j) can be calculate with formula: $j * d + i$*

For the algorithm, we assume that D-heap is type min and it has finite number of nodes.

```
//We assume that the D-heap's node can have the most d-amount children. The algorithm should take as the input the finite array (we assume that it's an integer array but not necessary), the size of the array and maximum number of successors that a node can have.
delMin(int[] inputArray, int arraySize, int d)
{
//if array has only 1 element, we just return the message informing about it
        if (arraySize == 1) then
        {
                Print message 'The array has only 1 element and it will be remove';
        }
//we will start out algorithm if the D-Heap has at least 2 nodes – the base condition.
        else if (arraySize > 1) then
        {
//first, we will put the value at the last element in the array to the first element.
                inputArray [0] = inputArray[arraySize – 1];
//since we remove the minimum element at the root, so the size of the array should be decrease by 1.
                arraySize = arraySize – 1;
//we are calling the moveDown function to check if our new root is the smallest element, if not then we should switch its place with its child that has the least value at the label.
                moveDown(inputArray, arraySize, d, 0);
        }fi;
}

//The algorithm to check if parent node is smaller than every of its child, if not then we switch its place with its child that has the smallest value at label. The algorithm take as an input the finite array, the size of that array, number indicate how many children the node can have as maximum and an index of the parent node.
moveDown(int[] inputArray, int arraySize, int d, int x)
{
 // The base condition to start the algorithm, we check if the node has any children, if not then the node is a leaf and the algorithm won't start.
        if (x*d + 1 < arraySize) then
        {
// We declare the value 'min' to store the index of the child of inputArray[x] that has the least value at its label and we point it by default to the first child of the parent node.
                int min = x*d + 1;
// We assume that the node would have a finite number of children (that is at least 2) and it won't be too big, so the easiest way to find the minimum value would be a linear search. We start the loop from the parent node's second child and each time we moving to next child until we reach the last child of the node. The loop will stop once we reach the last child.
                for(int i = 2; i <= d; i = i+1) do
                {
// The base condition to start the statement. We check if the current child that we take under consideration does exist.
                        if (x*d + i < arraySize) then
                        {
// We check if the number at the label of the child that we assume has minimum value at its label is bigger than the child that we are taking under consideration.
                                if (inputArray[min] > inputArray[x*d + i]) then
                                {
// If the answer is yes then we put the index of the child that we're taking under consideration as an index of the child that has the minimum value at the label.
```

```
                        min = x*d + i;
                    }fi;
                }fi;
            }od;
// Now we're checking if the value at our parent node is bigger than the value at the child node that contain the minimum value at its label.
            if (inputArray[x] > inputArray[min]) then
            {
// If the answer is yes then first we create the variable `temp` to temporary store the value at our parent node.
                int temp = inputArray[x];
// We put the value at the child that contain the minimum value to the parent node.
                inputArray[x] = inputArray[min];
// We put the value of parent node that we stored at temp to the child that contain the minimum value at its label.
                inputArray[min] = temp;
// We call for recursive function moveDown to move our node down further. The function now will take as an input our original array, the size of the array, number of children and the new index of the node (now our value is stored at index 'min' instead of 'x'). The recursive function will be call until our value has been put in the right place.
                moveDown(inputArray, arraySize, d, min)
            }fi;

        }fi;
```

Justification of an algorithm's correctness:

The specification of the algorithm delMin becomes a pair <WP, WK>, where WP = (The min D-heap that is heap like structure with at most d successors that is represented in the finite array. The D-heap keep the standard properties of the heap) and WK = (The new D-heap represented in an array form that the first – minimum element of the array had been removed).

Stop condition of the algorithm – for delMin algorithm we have base condition where the main algorithm will start only if the D-heap has at least 2 elements. If it has only 1 element we would only print the message to inform about it, if the D-heap is empty then we won't start our algorithm.

For moveDown algorithm, it will start only if the parent node is not a leaf. After that we have a loop that go through all the children of the parent node. Since our WP stated that the node has at most d number of children then the loop will stop once we reach the last child.

Partial correctness of the algorithm – for the initial condition WP = (The min D-heap that is heap like structure with at most d successors that is represented in the finite array. The D-heap keep the standard properties of the heap) and WK = (The new D-heap represented in an array form that the first – minimum element of the array had been removed).
In the algorithm, first we will remove the minimum element of the D-heap by moving the element at the last index to first one, at the same time we would decreasing the size of the array by 1. After that we would move the element at the root down until it reach the leaf (when it has no children) or all its children contains a bigger element.

Example:
The D-heap that each node can have at most 4 successors that is represented in an array:
inputArray = {6, 15, 7, 12, 19, 16, 18, 13, 14, 16, 17, 18}
arraySize = 12 and the d = 4.

```
delMin(int[] inputArray, int arraySize, int d)
{
        if (arraySize == 1) then
        {
                Print message 'The array has only 1 element and it will be remove';
        }
//Since arraySize = 12, we start the algorithm
        else if (arraySize > 1) then
        {
//inputArray[0] = 18, now we have new array: inputArray = {18, 15, 7, 12, 19, 16, 18, 13, 14, 16, 17, 18}
                inputArray [0] = inputArray[arraySize – 1];
//arraySize = 12-1 = 11, so we will consider the inputArray without last element:  inputArray = {18, 15, 7, 12, 19, 16, 18, 13, 14, 16, 17, }
                arraySize = arraySize – 1;
//we start to call function to move down the node at index 0 with the arraySize =11
        moveDown(inputArray, arraySize, d, 0);
    }fi;
}

moveDown(int[] inputArray, int arraySize, int d, int x)
{
  // 1) We check if the node has children: x=0, d=4, arraySize = 11 → 0*4+1 < 11 since (1<11), we start the calculation
// 2) We check if the node has children: x=2, d=4, arraySize = 11 → 2*4+1 < 11 since (9<11), we start the calculation
// 3) We check if the node has children: x=9, d=4, arraySize = 11 → 9*4+1 < 11 since (37>11), we stop the algorithm.
    if (x*d + 1 < arraySize) then
    {
//1) min = 0*4+1=1
//2) min = 2*4+1=9
        int min = x*d + 1;
// the loop to go through the second child till the fourth child of the node. Each time we'll go to next child.
        for(int i = 2; i <= d; i = i+1) do
        {
//1*) we check if the second child exist: x=0, d=4, i=2 arraySize = 11 → 0*4+2 < 11 since (2<11), we start the calculation.
//1**) we check if the third child exist: x=0, d=4, i=4 arraySize = 11 → 0*4+3 < 11 since (3<11), we start the calculation.
//1***) we check if the fourth child exist: x=0, d=4, i=4 arraySize = 11 → 0*4+4 < 11 since (4<11), we start the calculation.
//2*) we check if the second child exist: x=2, d=4, i=2 arraySize = 11 → 2*4+2 < 11 since (10<11), we start the calculation.
//2**) we check if the third child exist: x=2, d=4, i=4 arraySize = 11 → 2*4+3 < 11 since (11 is not smaller than 11), the third child does not exit, we exit the loop.

            if (x*d + i < arraySize) then
            {
// 1*)we check if the value at index min is bigger than the value at the second child: inputArray[1]=15, inputArray[2]=7, since 15 > 7 then min = 0*4+2 = 2
// 1**)we check if the value at index min is bigger than the value at the third child: inputArray[2]=7, inputArray[3]=12, since 7<12 we do nothing.
// 1***)we check if the value at index min is bigger than the value at the fourthchild: inputArray[2]=7, inputArray[4]=19, since 7<19 we do nothing.
// 2*)we check if the value at index min is bigger than the value at the second child: inputArray[9]=16, inputArray[10]=17, since 16<17 then we do nothing,
                if (inputArray[min] > inputArray[x*d + i]) then
                {
                    min = x*d + i;
                }fi;
            }fi;
        }od;
//1) we check if the value at parent is bigger than value at minimum child: inputArray[0]=18, inputArray[2]=7
//2) we check if the value at parent is bigger than value at minimum child: inputArray[2]=18, inputArray[9]=16
        if (inputArray[x] > inputArray[min]) then
        {
//1) since18>7 we create temp = 18
//2) since18>16 we create temp = 18
            int temp = inputArray[x];
//1) inputArray[0]=7
//2) inputArray[2]=16
            inputArray[x] = inputArray[min];
//1) inputArray[2]=18, we have new array: inputArray = {7, 15, 18, 12, 19, 16, 18, 13, 14, 16, 17, }
//2) inputArray[9]=18, we have new array: inputArray = {7, 15, 16, 12, 19, 16, 18, 13, 14, 18, 17, }
            inputArray[min] = temp;
//1) we are calling the recursive function that will take inputArray, arraySIze = 11 and min = 2. We are starting the algorithm from beginning.
//2) we are calling the recursive function that will take inputArray, arraySIze = 11 and min = 9. We are starting the algorithm from beginning.
            moveDown(inputArray, arraySize, d, min)
            }fi;

        }fi;

  //The D-heap represented in an array after delMin:  inputArray = {7, 15, 16, 12, 19, 16, 18, 13, 14, 18, 17, }.
```

Because the delMin algorithm is partially correct in terms of specification <WP,WK> and we have shown the stop condition for this algorithm, it is also absolutely correct with respect to the considered specification.

Estimation of algorithm's complexity:

Time complexity – since the algorithm delMin dominant operation is assigning value and calling for function moveDown (that has inside it recursive function) we would first have considered the cost of the operations inside moveDown algorithm without the recursive function.
The dominant operation inside algorithm moveDown is comparison and assignment, that would normal cost a constant time, however we have one loop that go through the children elements of node so it would go 'd-1' time so the cost of time complexity for the moveDown algorithm without counting the recursive function would be O(d-1).
As we moving the element down the tree, which each level the node can be swap with only one another node at lower level, so at the worst case we would have to call for recursive function the times equals to height of the D-heap tree which we can calculate with formula: $\log_d n$
So we have T(n) = $(\mathrm{d}-1) * \log_d n$ *so the average time complexity for the delMin algorithm is* $\theta((\mathrm{d}-1) * \log_d n)$ *and the time complexity in worst case is* $O((\mathrm{d}-1) * \log_d n)$ *(!the reason why (d-1) is for example in the case of normal heap when parent can has max 2 children the cost of time complexity would be O(log(n)), instead of O(2log(n))).*

Space complexity – the delMin algorithm only uses 2 auxiliary variables that are independently of the value of the argument n, therefore S should be S(n) = 1 so we have $O(1)$.