1.

(a)

We can represent a d-ary heap in a 1-dimensional array as follows. The root resides in A[1], its d children reside in order in A[2] through A[d + 1, their children reside in order in A[d + 2] through A[d2 + d + 1], and so on. The following two procedures map a node with index i to its parent and to its j-th child (for 1 ≤ j ≤ d), respectively.

D−ARY−PARENT( i )

r e t u r n b(i − 2)/d + 1c

D−ARY−CHILD( i , j )

r e t u r n d(i + 1) + j + 1

To convince yourself that these procedures really work, verify that

D−ARY−PARENET(D−ARY−CHILD( i , j ) ) = i ,

for any 1 ≤ j ≤ d. Notice that the binary heap procedures are a special case of

the above procedures when d = 2.

(b)

Since each node has d children, the height of a d-ary heap with n nodes is Θ(logd n) = Θ(lg d/ lg n).

(c)

The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for d-ary heaps too, assuming that HEAP-INCREASE-KEY works for d-ary heaps. The worst-case running time is still Θ(h), where h is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a d-ary heap, this is Θ(logd n) = Θ(lg d/ lg n).

(d)

Deletemax removes the value in the root (which is the maximum value) and places the last element of the heap - name it a - to the side. Then the procedure finds the child of the root with the maximum value and compares this value to the value of a. If a is bigger then it is placed in the root, and the heap has been restored. Otherwise, the maximum child value moves up to the root, and the algorithm is applied recursively using this child as the new root.

O(d logd n)

2.

The idea is to do a preorder traversal of the give Binary heap. While doing preorder traversal, if the value of a node is greater than the given value x, we return to the previous recursive call. Because all children nodes in a min heap are greater than the parent node. Otherwise we print current node and recur for its children.

Here is a sketch of a recursive algorithm: start from the root of the heap. If the value of the root is smaller than X then print this value and call the procedure recursively once for its left child and once for its right child . If the value of a node is bigger or equal than X then the procedure stops without printing that value.

The complexity of this algorithm is O(N), where N is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than X, so the procedure has to call each node of the heap.

3.

(a)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| keys | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 61 | 15 |  | 22 |  |  | 27 |  |  | 10 | 19 |
|  |  |  |  | 33 |  |  | 71 |  |  | 54 |  |
|  |  |  |  | 44 |  |  | 5 |  |  |  |  |

(b)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| keys | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 61 | 15 | 54 | 22 | 33 | 44 | 27 | 71 | 5 | 10 | 19 |

(c)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| keys | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 61 | 15 | 5 | 22 | 33 | 71 | 27 | 54 | 44 | 10 | 19 |

https://www.youtube.com/watch?v=AYcsTOeFVas&ab\_channel=Jenny%27slecturesCS%2FITNET%26JRF