CS 477/677 Analysis of Algorithms

Fall 2023

Homework 5

Due date: October 24, 2023

1. (U&G-required) [20 points] Suppose that an array that needs to be sorted contains only integers in the range 0 to *k* and that there is no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of a new array B.

```
Counting-sort(A,k)
      Let C[0…k] be a new array
      For i=0 to k
            C[i]=0
      For j=1 to A.length
            C[A[j]]+
      cInd = k
      aInd = A.length
      While aInd >= 1
            If c[cInd] == 0:
                  cInd--
            Else:
                  c[cInd] --
                  A[aInd] = cInd
                  aInd --
```

2. (U&G-required) [20 points]

Write pseudocode for an algorithm that determines if an array A[1..*n*] is a heap. Determine the time efficiency of the algorithm.

```
Func isHeap(A, n)
// A is array of keys
// n is num keys
For i from 1 to n/2
       If n >= 2i && A[i] < A[2i]
              Return false
       If n >= 2i + 1 && A[i] < A[2i + 1]
              Return false
Return true
```

This algorithm has O(n) runtime efficiency. The runtime is deterministic since the runtime does not depend on the data set. The algorithm will always execute all the way through the range. The algorithm has a for loop that runs once and only calls primitive operations inside the for loop.

3. (U&G-required) [20 points]

a) [10 points] The RB-INSERT algorithm sets the color of the newly inserted node z to red. If instead z's color were set to black, then property 4 of a red-black tree would not be violated. Why not set z's color to black?

It is important to set the color of z to red instead of to black because if the inserted node is black, then more intensive difficult computations are required. Coloring the new node black possibly violates the property that the tree branches black heights must be the same. Fixing this violation is much more intensive and involves checking the whole tree at every step.

b) [10 points] Argue that in a red-black tree, a red node cannot have exactly one non-NIL child.

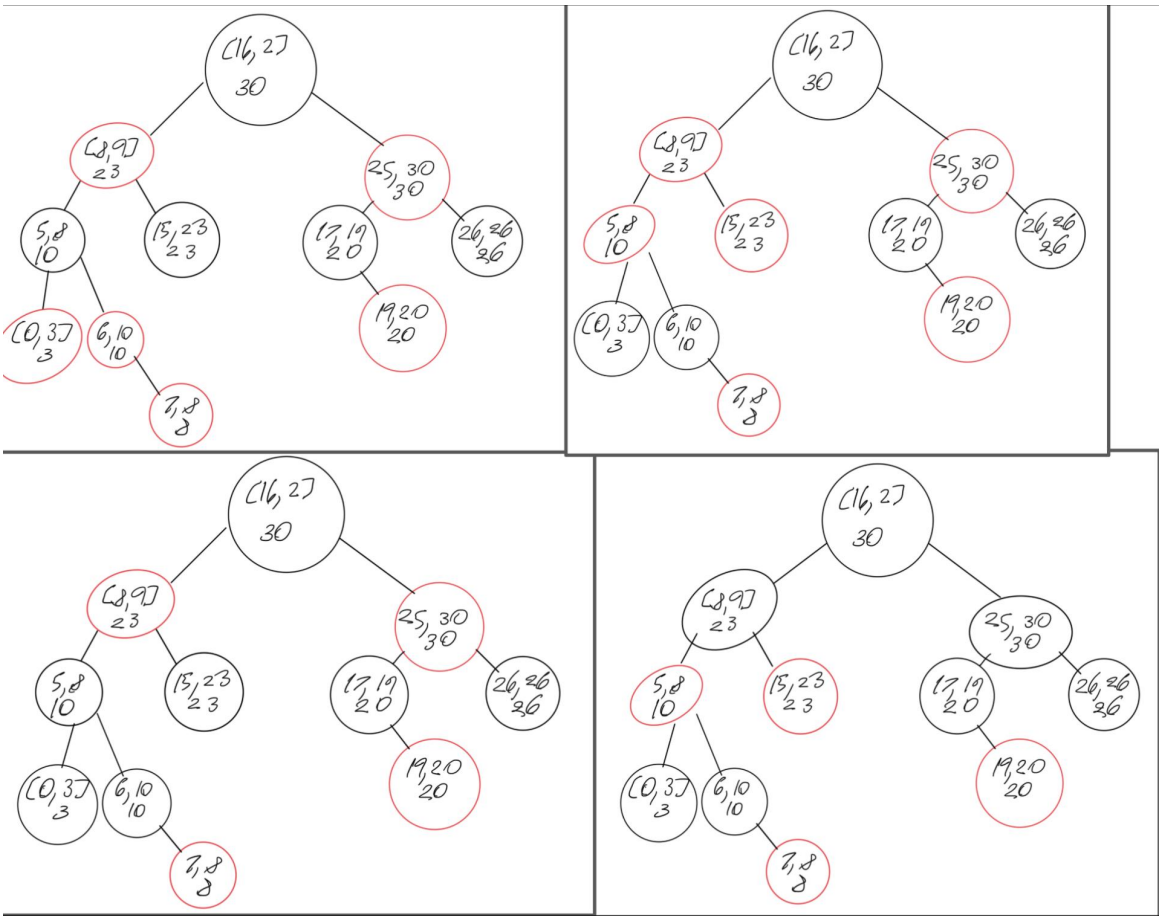
A red node cannot have one non nil red node because of the properties of red black trees therefore, This question can be rephrased to state "Show a red node always has either 2 nil children or 2 internal children". We know this statement to be true because if a red node had one black non nil child and one black nil child the two branches from the red node would have different black heights. The non nil branch would have a black height of 2 while the nill branch would have a black height of 1 with respect to the red node in question.

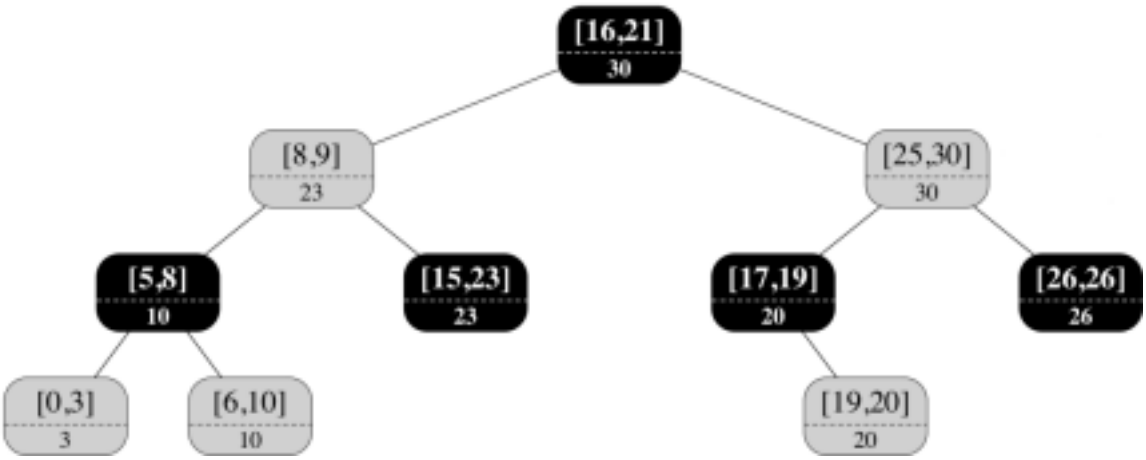
4. (U & G-required) [20 points]

(a) [10 points] Show how INTERVAL-SEARCH(T, i) operates on the tree T shown in the figure below, with `i = [11, 16]`.

Interval search will first set x equal to the root of tree or [16,21]. Since this is not nill, or overlap with i, the while loop is ran. The left subtree of [16,21] is not nil, and the max of the root (30) is greater than the low of i (16), x becomes the root of the left subtree so x is now [8,9]. Since this is not nill, or overlap with i, the while loop is ran. The left subtree of [8,9] is not nill, but the max of the left subtree is less than the low of [11,16], so x becomes the root of the right subtree. X is now[15,23]. [15,23] does overlap i so the interval, so the function returns the interval.

(b) [10 points] Show the tree that results after inserting interval `i = [7, 8]` into the tree T shown in the figure below. Make sure to restore any red-black tree properties that may be affected during the insert.





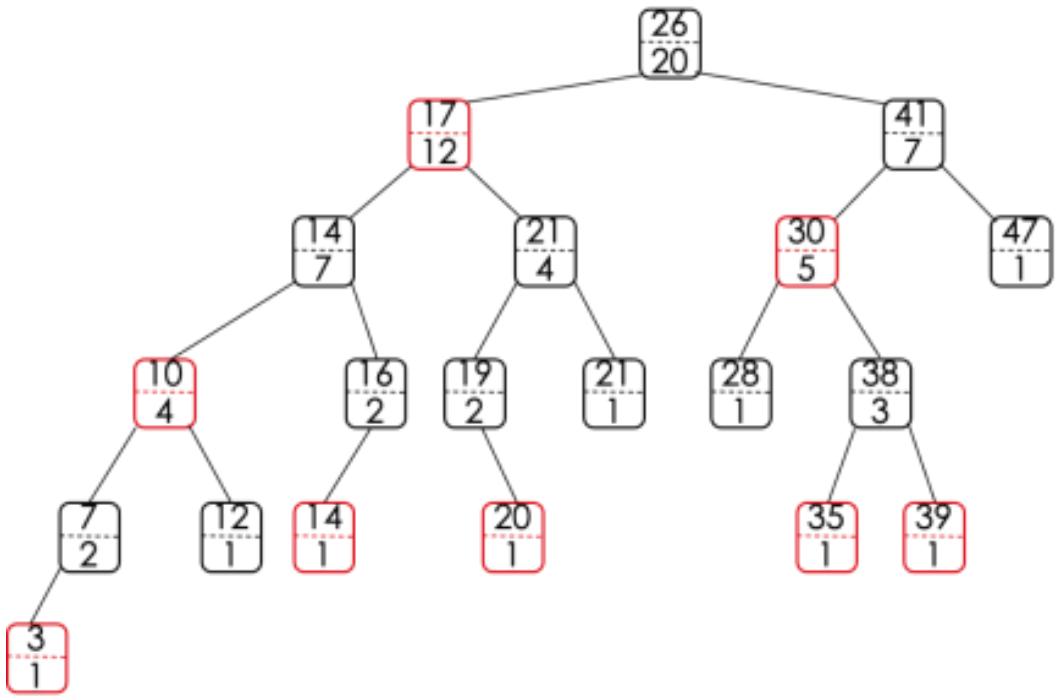
5. (U & G-required) [20 points]

(a) [10 points] Show how OS-SELECT (*T.root*, 18) operates on the red-black tree shown in the figure below.

OS-Select(T.root,18) is used to determine the 18th smallest element in the red black tree. On the first call r is equal to size(left) + 1 or 13. Since i 18 is greater than 12 we explore the right tree with a new i value of i-r or 18 - 13 so 5. We are now looking for the 5th smallest element of the right subtree of the root. On the second call, r is equal to size(left) + 1 or 6. Since i is less than r (5<6) we explore the left subtree. On the third call, r is size(left) +1 or 2. Since i is greater than r(5 > 2) we explore the right subtree. We are now looking for the 3rd smallest element of this subtree (root of 38). On the fourth call r is size(left) + 1 or 2. Since i is greater than r(3 >2) we explore the right subtree. We are now looking for the 1st smallest element of this subtree(root of 39) On the fifth call r is size(left) + 1 = 1. Since r = i, we return this value all the way up the tree. Therefore, the result of the call is 39.

(b) [10 points] Show how OS-RANK(T, x) operates on the red-black tree shown in the figure below and the node x with $x.key = 21$.

In this function call of OS-Rank(T,21), we start with the node with key 21. Initially, r is 1 since the size of the left tree of 21 is 0. We also set y equal to 21. 21 is not the root so we process one iteration of the while loop. 21 is the right sub child of its parent so we increment r by the size of the left sub child of 21s parent and one. Therefore r is now 4 (1 + 2 + 1). We are now on the node which also has key 21 but is slightly higher. 21 is not the root so we process one iteration of the while loop. 21 is the right sub child of its parent so we increment r by the size of the left sub child of 21s parent and one. Therefore, r is now 12 (7 + 4 + 1). We are now on the node with the key of 17. 17 is not the root so we process one iteration of the while loop. 17 is not the right sub child so move up the tree. We are now on the node with the key of 26. Since 26 is the root node, we return r which is 12.



6. (G-required) [20 points]

What is the largest possible number of internal nodes in a red-black tree with black height k? What is the smallest possible number?

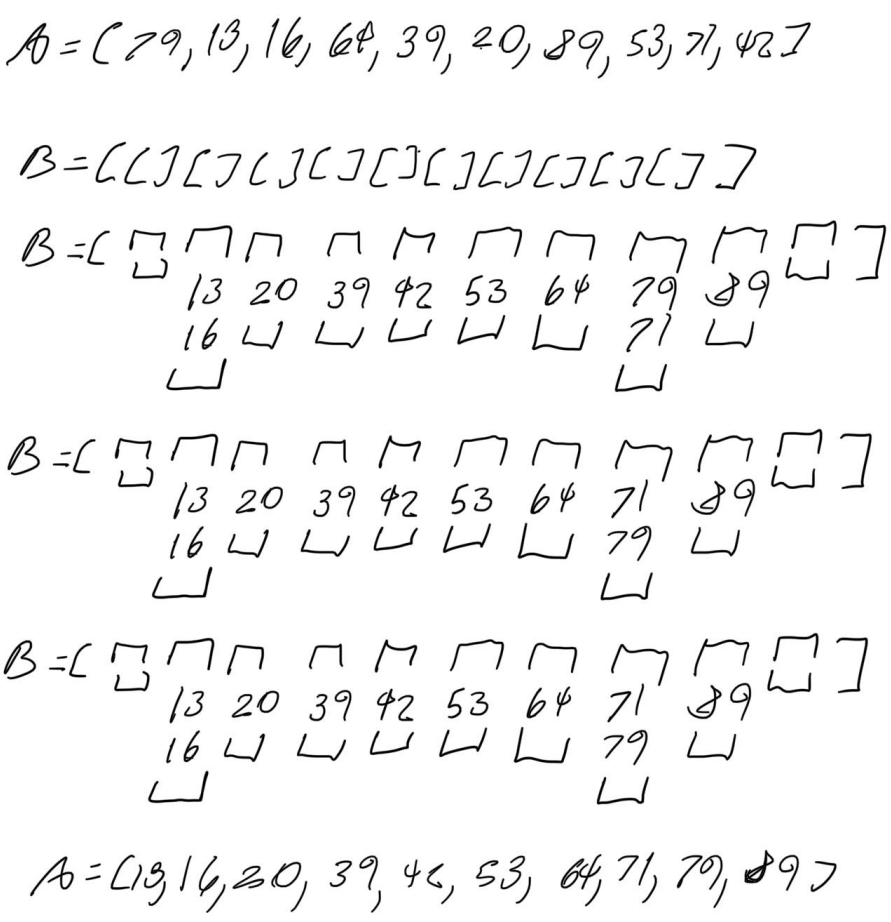
Extra credit:

7. [20 points]

a) [10 points] Illustrate the operation of BUILD-MAX-HEAP on the array A = [5, 3, 17, 10, 84, 19, 6, 22, 9].

See Attached Work Below

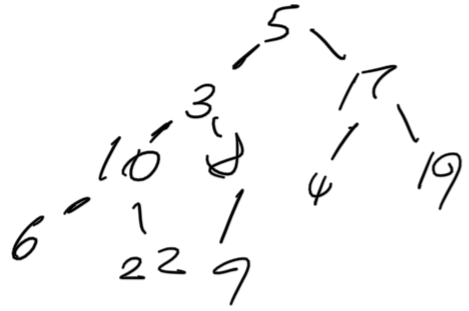
b) [10 points] Illustrate the operation of BUCKET-SORT on the array A = [.79, .13, .16, .64, .39, .20, .89, .53, .71, .42].



1 2 3 4 5 6 7 8 9 10

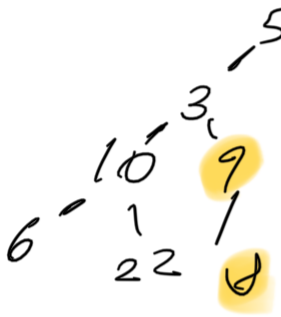
Array = [5, 3, 17, 10, 8, 4, 19, 6, 22, 9]

Original tree



call max heapify on ind (5 → 1)

max heapify (Array, 5)



1 2 3 4 5 6 7 8 9 10

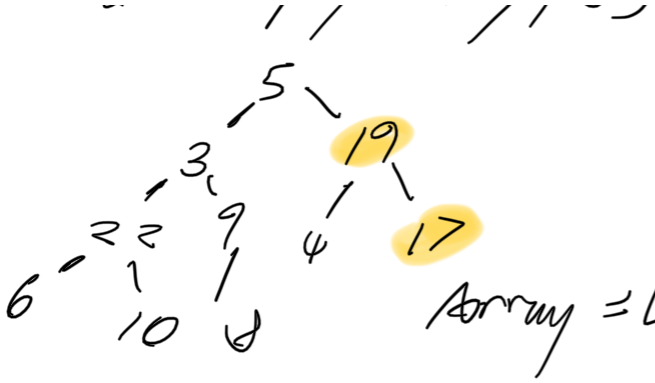
Array = [5, 3, 17, 10, 9, 4, 19, 6, 22, 8]

max heapify (Array, 4)

1 2 3 4 5 6 7 8 9 10

Array = [5, 3, 17, 22, 9, 4, 19, 6, 10, 8]

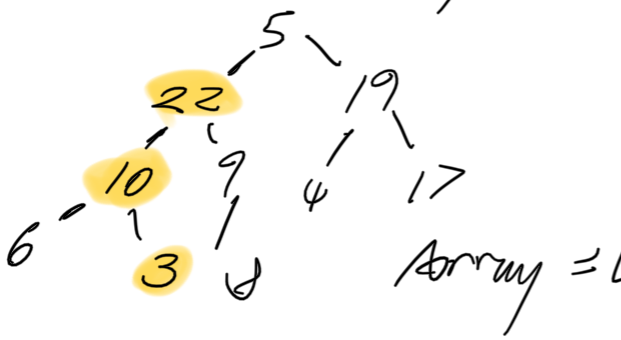
max heapify (Array, 3)



1	2	3	4	5	6	7	8	9	10
5	3	19	22	9	4	17	6	10	8

Array = [5, 3, 19, 22, 9, 4, 17, 6, 10, 8]

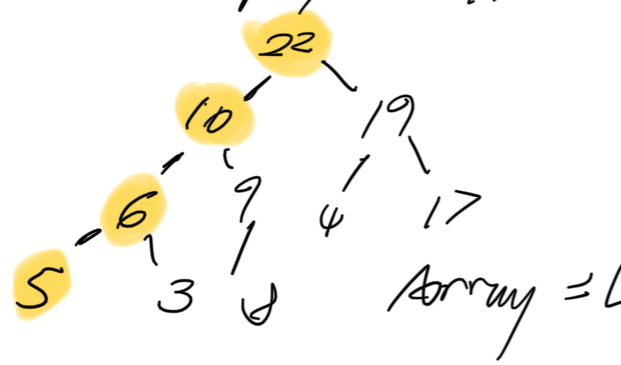
max heapify (Array, 2)



1	2	3	4	5	6	7	8	9	10
5	22	19	10	9	4	17	6	3	8

Array = [5, 22, 19, 10, 9, 4, 17, 6, 3, 8]

max heapify (Array, 1)



1	2	3	4	5	6	7	8	9	10
22	10	19	6	9	4	17	5	3	8

Array = [22, 10, 19, 6, 9, 4, 17, 5, 3, 8]