Assignment 3: Priority Queues, Binary Search Trees, Balanced Search Trees

In this assignment, you are forbidden from using standard APIs that would otherwise implement **data structures or sort functions or etc**. Any data structure used must be programmed on your own and submitted with the assignment. **Do not use java.util.Collections or the C++ STL or etc. unless otherwise stated.**

You will push all of your solutions to the Assignment 2 repository through the Github Classroom. Organize your repository in such a way that it contains:

Root – Assignment 3 directory

- Problem 1 directory
  - Problem 1 code…
  - Problem 1 PDF…
- Problem 2 directory
  - Problem 2 code…
  - Problem 2 PDF…
- …etc. for all numbered problems in this assignment set.

Note: We have removed the requirement for a main file.

Note: Do not include binaries / executables / bin folders / etc. Only upload the .java / .cpp / .py / .h / etc only.

Note: You are graded on directory structure.

There is no restriction on the programming language that you use.

For the problems, you will be implementing your code as an API through which the graders will make function calls.

Include a README.md file for the GitHub repository that has your Name as the title and a description of what is inside the repository, generally.

!! INVITATION LINK: https://classroom.github.com/a/2a759zfD !!

For anyone new to Github, we encourage you to use **Github Desktop** as a simple GUI for interacting: https://help.github.com/desktop/guides/getting-started-with-github-desktop/

For people who want to be more advanced and up to industry standards, you can use the Git command line:

https://git-scm.com/downloads

A very barebones introduction to the command line (very readable, I recommend it):

http://rogerdudler.github.io/git-guide/

1. **Priority Queues** – In the slides and lecture, we talked about binary heaps, a data structure where the only constant order to maintain is that the parent is either larger (max-heap) or smaller (min-heap) than its two children. On Slide 24 of Priority Queues, they list two other data structures: the *d-ary heap* and the *Fibonacci heap*. In this problem, we want to implement the max *d-ary heap*.
   A *d-ary heap* is a heap data structure that allows for more than two children per parent node., with the rule that the parent node must always be larger than the child node. You will implement this using arrays internally, storing only integers inside. The functions are:
   a. DaryHeap(int d); -- Constructor to create the DaryHeap, where 'd' is the number of children allowed.
   b. Void insert(int k); -- Adds an integer 'k' into the DaryHeap in proper place.
      a. Note: you will need to properly balance the heap in the event of the child being larger than the parent. As such, implement this method:
         i. swim(int k); -- Promotes a child further up the heap structure until it is in proper order. Typically done by swapping with the parent.
   c. Int delMax(); -- Swap the largest value in the heap with the most recent child, and then delete that largest value from the heap. **Return the largest value to the user**.
      a. Note: you will need to properly balance the heap after deletion. Implement the method:
         i. Sink(int k) – Demotes a parent node down the heap structure through swaps until it is in final order. Make sure that the heap structure is preserved once the method is completed (max heap condition).
   d. Int[] sortedArray daryHeapsort(); -- Returns the contents of the heap in sorted order implementing heapsort.
   e. Inside your heap, begin with an array of constant size '10'. When the values inserted into the heap exceed size 10, implement this method:
      a. doubleArray(); -- Double the internal array inside the heapsort in the event that it becomes too large. Do this by instantiating an array of size 2N and copying the values over.
   f. Include a PDF, Problem1.PDF, where you give a short explanation as to what the Big-O time complexity of the delMax(); and daryHeapsort(); functions are ***in the worst case*. Please include leading coefficients, if any. If there is a constant, say 3*N, please show that. If the constant is a variable, list the variable in your O() function. Look at the heapsort on slide 39 for an example.** Your proof is up to you, detailed mathematics is not required. Please highlight in your code where the time complexity of the operations occur from. Reference slides 24, 38, 39 for ideas on what this might be.

   For this problem, please refer to slides 15-20 of Priority Queues to see heap implementation. Primarily, focus on slide 16 to figure out how to structure the array for a heap of 'd' children as opposed to just two children. For heapsort, see slides 30-34.

For submission, include only code that implements this data structure. We do not need a main file or main function.

2. **Balanced Binary Search Trees** - In lecture, we discussed a Binary Search Tree as a data structure created by assigning a value smaller than the parent to the left and a value larger than the parent to the right. To improve performance of the BST for regular operations, we discussed 2-3 Trees and Red-Black Trees as a way of rebalancing the tree to maintain a balanced tree height. 2-3 trees did this by handling different cases for temporary 4-nodes. Red-black trees did this by rotating the subtrees for different temporary 4-node cases. In this problem, we will balance an existing BST without *self-balancing*, which is when the insert/delete operations restructure the tree.
    a. Begin by implementing a BST for integers. The underlying structure is a linked list. You need these methods:
        i. BST(); -- Constructor
        ii. void put (int) – Inserts a value into the BST.
        iii. Void put(int[] a) – Inserts an entire array 'a' into the BST.
        iv. int search(int key) – Searches for a value in the BST. If found, return the key. If not found, return 0 and print 'Value not found!' to the console. Additionally, print out the total number of comparisons needed (whether the key is found or not).
        v. int returnSize(); -- returns the total number of nodes in the BST.

    See Slides 3-10 on Binary Search Trees to see how a BST might be structured with these methods. The implementation is effectively the same as shown.
    b. Now, we will attempt to balance the tree after the values have been inserted. Implement these methods:
        i. int[] a sortedTree(); -- Outputs the tree in sorted order of an array.
        ii. BST balanceTreeOne(); -- Balances the tree by creating a new BST and inserting the values of the sorted array **in a way such that the tree height is balanced.** Linear insertion of the sorted values would be the worst possible height.
    c. During Red-Black Trees, we discussed the concept of 'rotations'. Rotations of a node involve pushing a node to the left or right and reconnecting the parent/children nodes. Please see Slides 26-29 of the Balanced Search Tree lectures to see what occurs in rotations. Implement these methods for your BST:
        i. Node rotateRight(Node h) – Rotates a given node h to the right and returns the new parent node after rotation. See the Slides for detail.
            1. *Note! You need to update the code here to cancel the function if there are no left subchildren!*
        ii. Node rotateLeft(Node h) – Rotates a given node h to the left and returns the new parent node after rotation. See the slides for detail.
            1. *Note! You need to update the code here to cancel the function is there are no right subchildren!*
        iii. **void** transformToList(); -- Performs a series of right rotations on the BST until the BST is just a linked list oriented to the right. The logic is like so:

1. Begin at the root, and check if there are left subchildren. If there are, rotate the root to the **right**.
2. If the root *still* has left children after rotating, continue rotating **right** until there are no more left children at the root.
3. Move to the right. Check if this parent node has left children. If so, **rotateRight** on the parent. Continue doing so until there are no more left children.
4. Repeat all the way down until there are no more nodes that have left children. The tree is now a right-leaning linked list.

d. We will now attempt to balance the BST without creating new memory, as we did in Part b. Implement these methods:
   i. void balanceTreeTwo(); -- Restructures the BST. It will do this by following this logic:
      1. First, restructure the tree by calling transformToList(); to turn it into a right-leaning linked list.
      2. Compute the parameter $M = (N + 1) - 2^{\lfloor \log_2 N \rfloor}$.
         a. N is the total number of nodes, $\lfloor \log_2 N \rfloor$ means to compute the floor of $\log_2 N$.
      3. From the root of the tree (which is now a right-leaning linked list), perform left rotations on every odd node until M odd nodes have been rotated.
         a. Example: Imagine a list 0->1->2->3->4. If $M = 2$, then we perform rotateLeft() on the '0' node and the '2' node. That's exactly M odd nodes.
      4. Compute the parameter $K = \lfloor \log_2 N \rfloor - 1$.
      5. Compute this loop:
         a. While $K > 1$, rotate every odd node to the left.
         b. Decrement K by 1 when finishing a set of rotations.
         c. When $K == 1$, only rotate the parent node and complete the algorithm.
      6. Congratulations, the tree is now balanced!

e. Include a PDF, Problem2.PDF. Include these parts:
   i. How does the algorithm of balanceTreeTwo(); balance the tree -- why does this work? Explain it for all steps.
   ii. What is the total <u>time complexity</u> of the balanceTreeOne(); function in the average case? Consider how this function worked. balanceTreeOne(); first calls sortedTree(); then creates a new BST out of that. The cost of this should be $O(BalanceTreeOne) = O(sortedTree) + O\big(put(int[a])\big)$. You will fill this in for the actual values.
   iii. What is the <u>space complexity</u> of the balanceTreeOne(); function?
   iv. What is the total <u>time complexity</u> of the balanceTreeTwo(); function? Perform a similar analysis to I, where you sum up the cost of the internal operations.
   v. What is the <u>space complexity</u> of balanceTreeTwo();?