CS25100 Homework 2: Spring 2017

Due Monday, April 17, 2017, before 11:59 PM. Please edit directly this document to insert your answers. You can use any remaining slip days for his homework. Submit your answers on Vocareum.

1. True/False Questions (18 pts)

1. _F__ The reverse postorder of a digraph's reverse is the same as the postorder of the digraph.
2. _F__ Adding a constant to every edge weight does not change the solution to the single-source shortest-paths problem.
3. _T__ An optimization problem is a good candidate for dynamic programming if the best overall solution can be defined in terms of optimal solutions to subproblems, which are not independent.
4. _T__ If we modify the Kosaraju algorithm to run first depth-first search in the digraph *G* (instead of the reverse digraph G^R) and the second depth-first search in G^R (instead of *G*), the algorithm will find the strong components.
5. _T__ If you insert keys in increasing order into a red-black BST, the tree height is monotonically increasing.
6. _T__ A good hash function should be deterministic, i.e., equal keys produce the same hash value.
7. _T__ In the situation where all keys hash to the same index, using hashing with linear probing will result in *O(n)* search time for a random key.
8. _F__ Hashing is preferable to BSTs if you need support for ordered symbol table operations.
9. _T__ In an adjacency list representation of an undirected graph, *v* is in *w*'s list if and only if *w* is in *v*'s list.
10. _F__ Every directed, acyclic graph has a unique topological ordering.
11. _F__ Preorder traversal is used to topologically sort a directed acyclic graph.
12. __T_ MSD string sort is a good choice of sorting algorithm for random strings, since it examines $N \log_R N$ characters on average (where *R* is the size of the alphabet).
13. __T_ The shape of a TST is independent of the order of key insertion and deletion, thus there is a unique TST for any given set of keys.
14. _F__ In a priority queue implemented with heaps, *N* insertions and *N* removeMin operations take *O(N log N).*

15. __T_ An array sorted in decreasing order is a max-oriented heap.

16. __T_ If a symbol table will not have many insert operations, an ordered array implementation is sufficient.

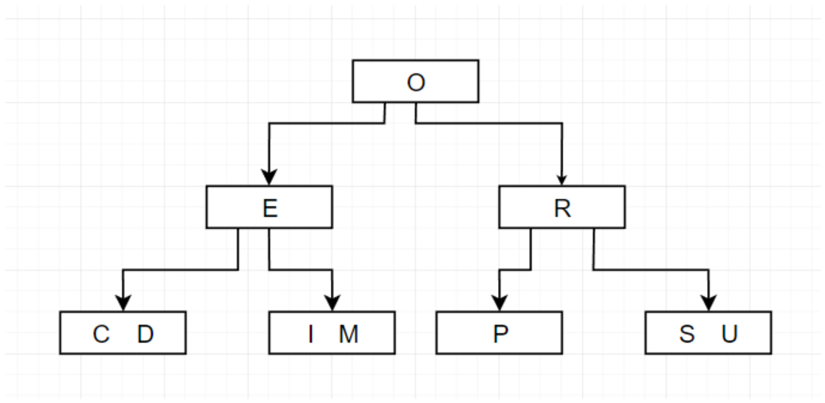
17. __F_ The floor operation returns the smallest key in a symbol table that is greater than or equal to a given key.

18. __F_ The root node in a tree is always an internal node.

2. Questions on Tracing the Operation of Algorithms (30 pts)

1. (4 pts) Draw the 2-3 tree that results when you insert the following keys (in order) into an initially empty tree:

   P U R D U E C O M P S C I



2. (5 pts) Give the contents of the hash table that results when you insert the following keys into an initially empty table of *M* = *5* lists, using separate chaining with unordered lists. Use the hash function *11k mod M* to transform the k-th letter of the alphabet into a table index, *e.g.*, *hash(I) = hash(9) = 99 % 5 = 4*. Use the conventions from Chapter 3.4 (new key-value pairs are inserted at the beginning of the list).

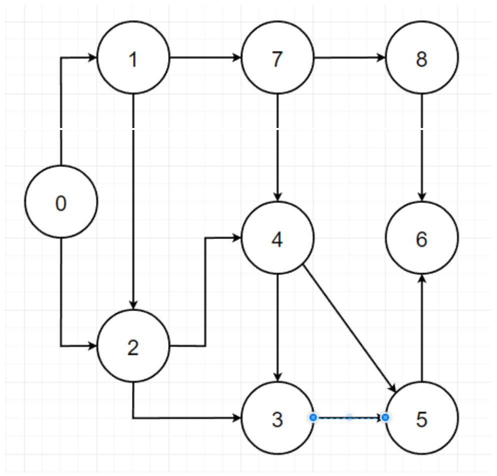
   M I T C H D A N I E L S

LL #	**Head -> …**
LL0	(E, 9) -> (T, 2)
LL1	(A, 6)
LL2	(L, 10)
LL3	(H, 4) -> (C, 3) -> (M, 6)
LL4	(S, 11) -> (N, 7) -> (D, 5) -> (I, 8)

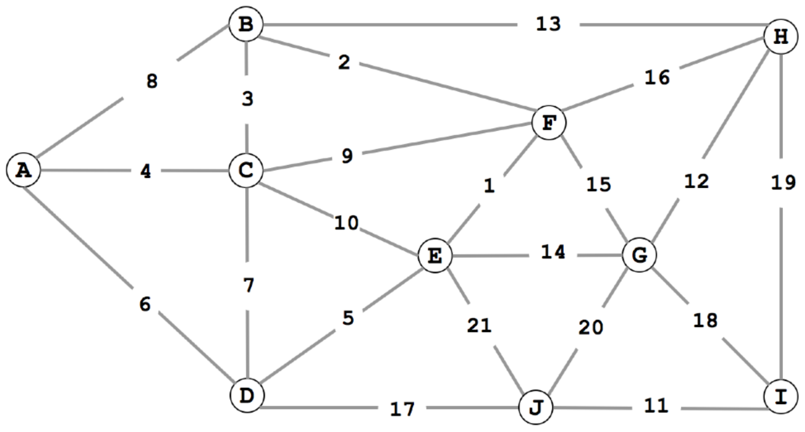
3. (4 pts) List the vertices in the order in which they are visited (for the first time) in DFS for the following undirected graph, starting from vertex 0. For simplicity, assume that the Graph implementation **always iterates through the neighbors of a vertex in increasing order**. The graph contains the following edges:

0-1 1-2 1-7 2-0 2-4 3-2 3-4 4-5 4-6 4-7 5-3 5-6 7-8 8-6

DFS Order: 0, 1, 2, 3, 4, 5, 6, 8, 7



4. (7 pts) Consider the following weighted graph with 10 vertices and 21 edges. Note that the edge weights are distinct integers between 1 and 21. Since all edge weights are distinct, identify each edge by its weight (instead of its endpoints).



(a) List the sequence of edges in the MST in the order that Kruskal's algorithm includes them (starting with 1).

Sequence of edges in the order that Kruskal's Algorithm includes:

1, 2, 3, 4, 5, 11, 12, 13, 17

NOT INCLUDED:

6, 7, 8, 9, 10, 14, 15, 16**,** 18, 19, 20 ,21

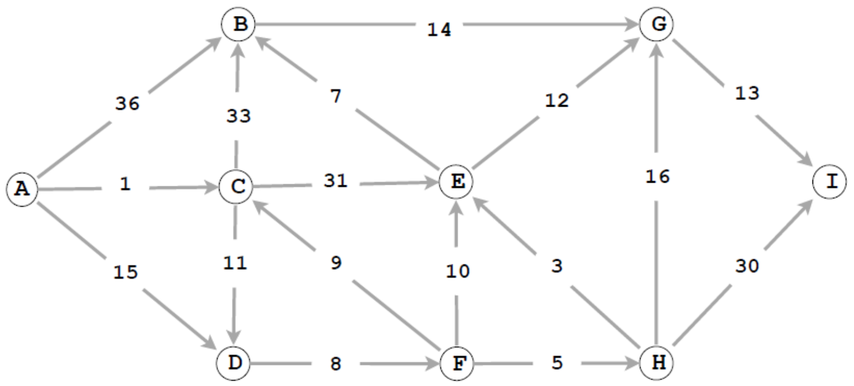
(b) List the sequence of edges in the MST in the order that Prim's algorithm includes them. Start Prim's algorithm from vertex A.

Prim's algorithm visits the least weighted path available whilst not connecting an edge to an already existing vertex.

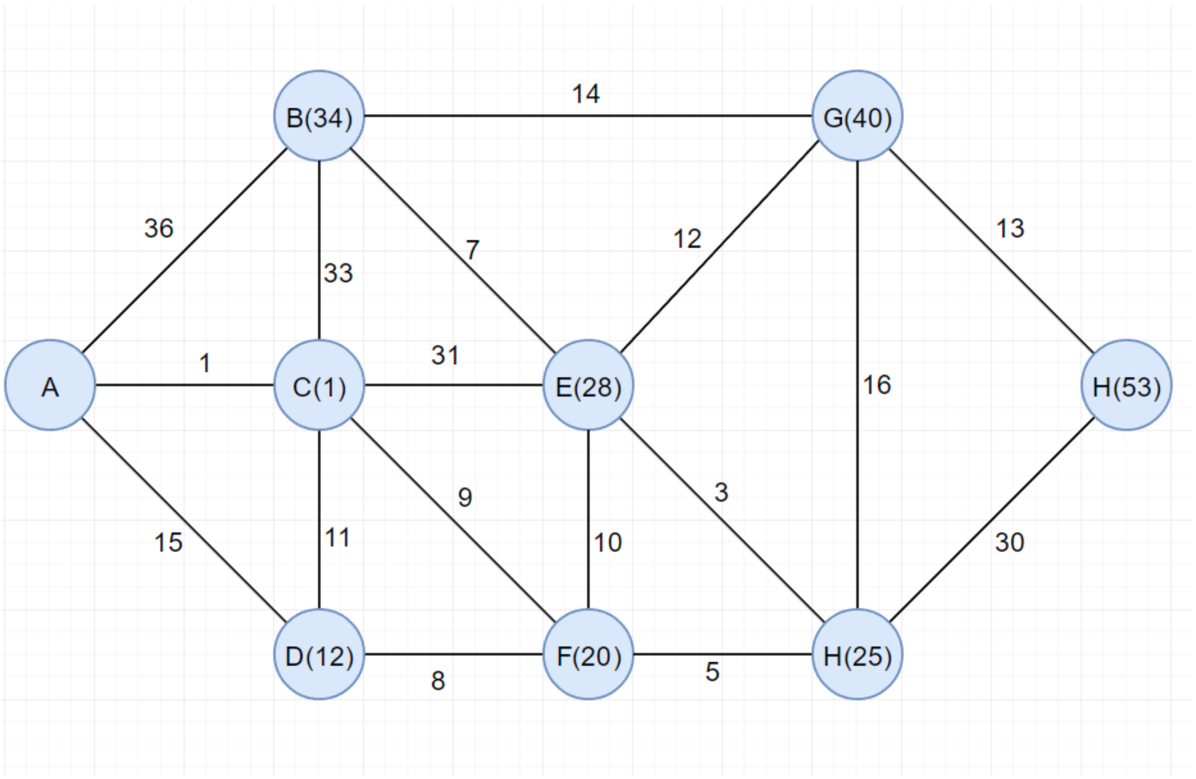
Starting at A the order is:

4, 3, 2, 1, 5, 13, 12, 17, 11

5. (5 pts) Consider the following weighted digraph and consider how Dijkstra's algorithm will proceed starting from vertex A. List the vertices in the order in which the vertices are dequeued (for the first time) from the priority queue and give the length of the shortest path from A to each vertex.



Vertex	A	C	_D_	_F_	_H_	_E_	_B_	_G_	_I_
Distance	0	1	_12_	_20_	_25_	_28_	_34_	_40_	_53_



6. (5 pts) Sort the 12 names below using LSD string sort. Show the result (by listing the 12 full words) at each of the four stages of the sort:

John, Jane, Alex, Eric, Will, Nick, Jada, Jake, Nish, Luke, Yuan, Emma

STAGE 1	**STAGE 2**	**STAGE 3**	**STAGE 4**
Jada	Yuan	Jada	Alex
Emma	Nick	Jake	Emma
Eric	Jada	Jane	Eric
Jane	Alex	Nick	Jada
Jake	John	Will	Jake
Luke	Eric	Nish	Jane
Nish	Jake	Alex	John
Nick	Luke	Emma	Luke
Will	Will	John	Nick
John	Emma	Eric	Nish
Yuan	Jane	Yuan	Will
Alex	Nish	Luke	Juan

3. The Right Data Structure (8 pts)

Indicate, for each of the problems below, the best data structure from the following options: binary search tree, hash table, linked list, heap. Provide a brief justification for each answer.

1. Find the kth smallest element.

   The best data structure from the list in order to find the kth smallest element would be a (minimum oriented) heap. Most of the other options to choose from would require a O(n) runtime as they require sorting, or traversing the whole list in order to simply find the smallest element. For a min heap structure, the data is placed with the smallest value as the root with its next smallest as the left child and so on. With this algorithm brought about by the min heap we can use a simple sweep method to find the kth smallest element efficiently.

2. Find the last element inserted.

   The best data structure for finding the last element inserted would be a linked list. This is because you can implement a stack which will always have the last element inserted at the top, thus resulting in an O(1) or constant runtime and the best method of the available options.

3. Find the first element inserted

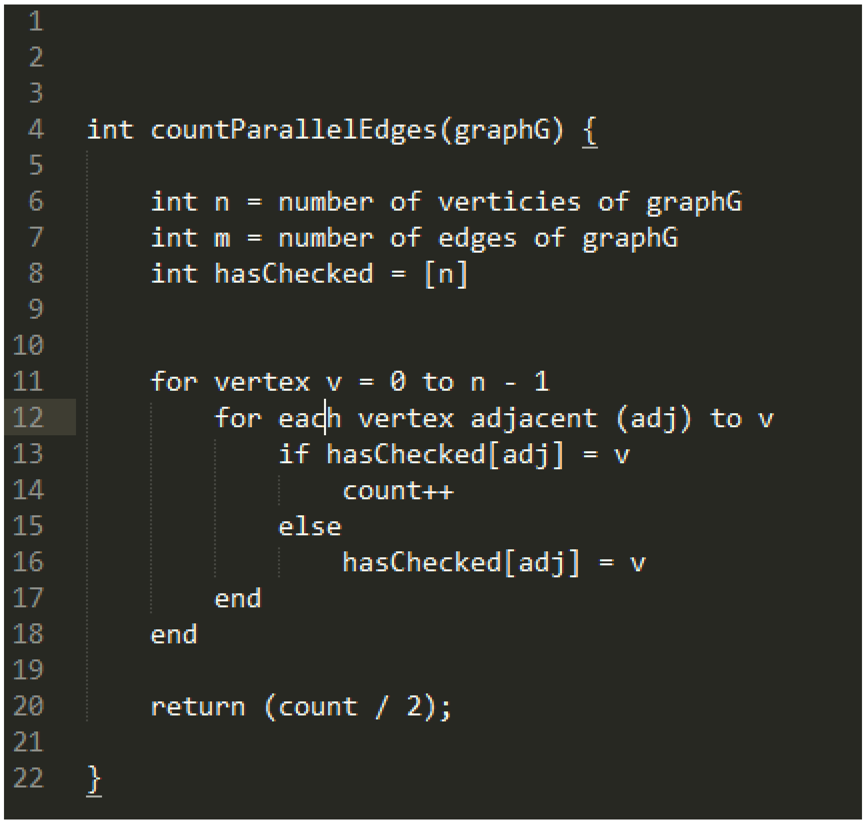
   The best data structure to find the first element inserted would, again, be a linked list. Similar to how you can implement a stack, you can instead implement a queue here (where first in is first out) thus giving us an O(1) or constant time method of finding the first element inserted.

4. Guarantee constant time access to any element.

   The best data structure for guarantee constant time access to any element would be a hash table. The hash table data structure guarantees constant time accesses while the other data structures om the list cannot in this case.

4. Design/Programming Questions (34 pts)

1. (7 pts) Give the pseudocode or Java code for a linear-time algorithm to count the parallel edges in an undirected graph



```
int countParallelEdges(graphG) {

    int n = number of verticies of graphG
    int m = number of edges of graphG
    int hasChecked = [n]


    for vertex v = 0 to n - 1
        for each vertex adjacent (adj) to v
            if hasChecked[adj] = v
                count++
            else
                hasChecked[adj] = v
        end
    end

    return (count / 2);

}
```

Here we have a time complexity of O(n + m) since each vertex and edge is encountered once, thus producing the requested linear runtime. My idea here was to create an array called hasChecked that we can use to reference the previously accessed adjacent nodes. By doing so each time we can repeatedly check and count all parallel edges of the graph. However, this method accesses each parallel vertices twice, so at the end we must divide our count by 2 in order to return the proper amount of parallel edges.

2. (7 pts) Given an MST for an edge-weighted graph G and a new edge e, describe how to find an MST of the new graph in time proportional to V .

When adding an edge to an edge-weighted graph, a cycle is created. In order to find the new MST you must delete the edge with the maximum weight on the cycle that was just created by adding edge ‘e’. This will take an estimated running time equal to or less than V

3. (10 pts) Design a data type that supports:

- insert in logarithmic time,
- find the median in constant time,
- remove the median in logarithmic time.

Give pseudocode of your algorithms. Discuss the complexities of the three methods. Your answer will be graded on correctness, efficiency, clarity, and succinctness.

We can accomplish this data structure by combining two priority queues (one min queue and one max queue), a previously studied data structure in this class, and build on top of their already matching running times in order to make this specific data type. We apply the minimum priority queue to the 'right' half (the numbers to the right of the median) and a maximum priority queue to the 'left' half (the numbers to the left of the median). We can then retrieve our median from the left half as it will always contain our median as the maximum.

NOTE: CODE IS ON THE NEXT PAGE, EXPLANATIONS BELOW!

Constructor: For the constructor I simply initialized three variables, the two priority queues (as described above: right half is minPQ and left half is maxPQ). I also initialize a counter integer to keep track of the total number of elements in the data structure.

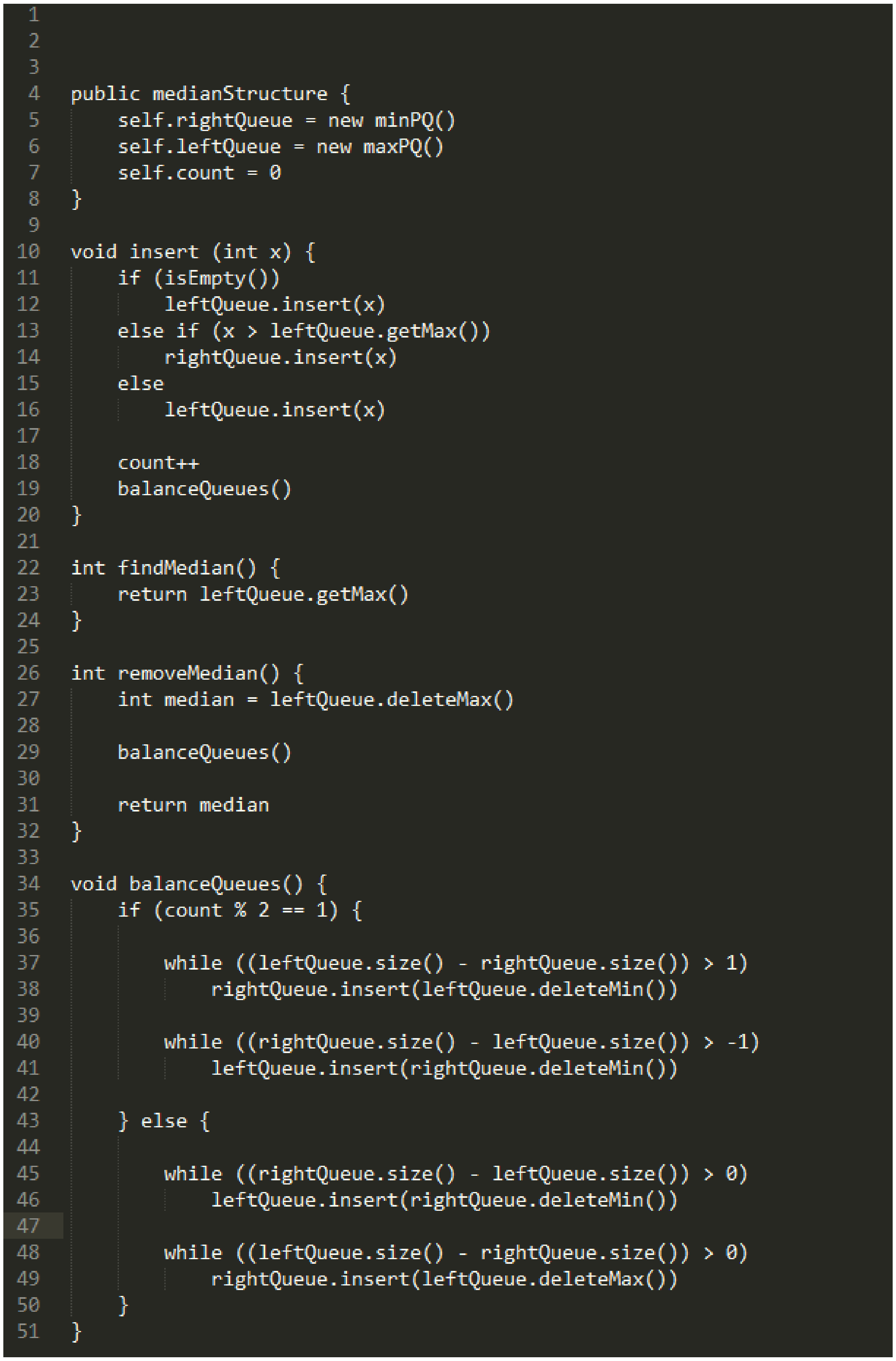
Insert: After comparing the input integer 'x' to the median (the maximum value in the leftPQ) we insert it into the right if it is greater or left it is equal to or less than. By doing this insertion method we are taking advantage of the Priority Queue's already built in insertion. My implementation combined with the stabled PQ makes for a O(logn) runtime. Also, by using this methodology to insert we can see that the median will ALWAYS be the max of the left queue, which gives way to how we can access our median in constant time! After the insertion we finish up by incrementing the count and balancing the queues (IMPORTANT TO MAINTAIN BALANCE OTHERWISE THE DATA STRUCTURE DOESN'T WORK!).

getMedian: As I mentioned in the previous section, the way that I insert into the data structure and built the two PQs enablers me to simply make a call to get the max of the left queue and that is the median. This is constant time (O(1)).

removeMedian: Again here we simply take advantage of the pre-built PQ functions to delete the max. I then balance the queues using my helper method and return the now-deleted median.

balanceQueues: The balancing is very important to this data structure as maintaining the balance and order truly enable us to access the getMedian in constant time. The goal is to make sure that our left PQ has our median in it and is at least 1 larger or the same size as the right PQ. We use a conditional to check which case it is (which PQ is larger) and then use while loops to delete switch the min/max of the left/right queue and insert it into the opposing queue. By maintaining this balance we are able to access the max value of the left queue and be confident it is our median.

Pseudocode:



```
public medianStructure {
    self.rightQueue = new minPQ()
    self.leftQueue = new maxPQ()
    self.count = 0
}

void insert (int x) {
    if (isEmpty())
        leftQueue.insert(x)
    else if (x > leftQueue.getMax())
        rightQueue.insert(x)
    else
        leftQueue.insert(x)

    count++
    balanceQueues()
}

int findMedian() {
    return leftQueue.getMax()
}

int removeMedian() {
    int median = leftQueue.deleteMax()

    balanceQueues()

    return median
}

void balanceQueues() {
    if (count % 2 == 1) {

        while ((leftQueue.size() - rightQueue.size()) > 1)
            rightQueue.insert(leftQueue.deleteMin())

        while ((rightQueue.size() - leftQueue.size()) > -1)
            leftQueue.insert(rightQueue.deleteMin())

    } else {

        while ((rightQueue.size() - leftQueue.size()) > 0)
            leftQueue.insert(rightQueue.deleteMin())

        while ((leftQueue.size() - rightQueue.size()) > 0)
            rightQueue.insert(leftQueue.deleteMax())
    }
}
```

4. (10 pts) The 1D nearest neighbor data structure has the following API:

- constructor: create an empty data structure.
- insert(x): insert the real number x into the data structure.
- query(y): return the real number in the data structure that is closest to y (or null if no such number).

Design a data structure that performs each operation in logarithmic time in the worst- case. Your answer will be graded on correctness, efficiency, clarity, and succinctness. You may use any of the data structures discussed in the course provided you clearly specify it.

For these specifics we can simply use a previously reviewed data structure that is now quite familiar to us: **the red-black binary search tree.**

Constructor:

Create an empty red-black BST object using the default constructor with no arguments.

Insert(x):

Insert the real number x into the red-black BST created via the constructor above utilizing the default function redBlackBST#put(Key, Value) with the Value = real number x

Query(y):

This is where we start to build on top of the Red-Black BST's already built in functions. If the data structure is empty, simply return NULL. Now we compute two variables to store our floor(y) and ceiling(y). We then compare each of these to 'x' to find which is closest and return that (closer) value which is found within the tree