CMPT225, Spring 2021

Final Exam

Name__Nathan Esau_____

SFU ID: |_3_|_0_|_1_|_1_|_9_|_7_|_5_|_6_|_8_|

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. You should write your solutions directly in this word file, and submit it to Coursys. Submitting a pdf is also ok.
2. Submit your solutions to Coursys before April 20, 23:59. No late submissions, no exceptions
3. Write your name and SFU ID on the top of this page.
4. This is an open book exam. You may use textbooks, calculators, wiki, stack overflow, geeksforgeeks, etc. If you do, specify the references in your solutions.
5. Discussions with other students are not allowed. Posting questions online asking for solutions is not allowed.
6. The exam consists of four (4) problems. Each problem is worth 25 points.
7. Write your answers in the provided space.
8. You may use all classes in standard Java, and everything we have learned.
9. Explain all your answers.
10. **Really, explain all your answers.**

Good luck!

Problem 1 [25 points]

A. (15 points) In this question you need to design a data structure that supports PriorityQueue with deletions. Specifically, you need to write the class PriorityQueueWithDeletions. As a motivation you may think of a priority queue for a printer that allows adding a document, removing a document in some order (e.g. shorter documents are printed first), or a user can cancel the job. Specifically the class needs to support the following operations:
The running time of each operation must be O(log(size of the queue)).

```
public class PriorityQueueWithDeletions<T extends Comparable<T>> {

        public PriorityQueueWithDeletions() - creates an empty priority queue

        public Ticket enqueue(T item) - adds a new element to the queue.
        Returns a ticket that can be used to remove your item from the queue.

        public T dequeue() - removes the oldest element from the queue
                             and returns it.

        public removeByTicket(Ticket t) - removes an element by ticket,
                                          and returns the removed element.

        public int size() - returns the number of elements in the queue

        public boolean isEmpty() - checks if the queue is empty

}
```

The running time of each operation must be O(log(n)), where n is the size of the priority queue.

Explain your answer in detail. Define the class Ticket and explain how you use it. Make sure Ticket does not allow the user to modify the queue.

Solution

The enqueue and dequeue requirements can be accomplished using a min heap. The min heap can be implemented using an array. The enqueue and dequeue methods will be O(log(n)).

The tricky requirement here is the removeByTicket in O(log(n)). This is possible if we want get our item in O(1) and adjust the heap in O(log(n)).

It is possible to get the item in O(1) if we keep an index hash map in our PriorityQueueWithDeletions class and update the index hash map as we swap elements around, delete elements and add new elements to our heap.

These would be the member variables of the PriorityQueueWithDeletions class.

- List<T> arr
- Map<T, Integer> indexMap;

These would be the member variables of the Ticket class.

- T value

To adjust the heap in O(log(n)) do the following:

- Mark item we are removing as “null”. This will be a temporary min value in heap.
- While the parent of our index is greater than the value at our index, swap these two values.

Example:

array = [3,7,5,10,15,8]
indexMap = {“3”: 0, “5”: 2, “7”: 1, “8”: 5, “10”: 3, “15”: 4}
ticket = “5”

We can instantly lookup 5 in our indexMap (index is 2). Then, we mark the item in array as null and remove item form indexMap.

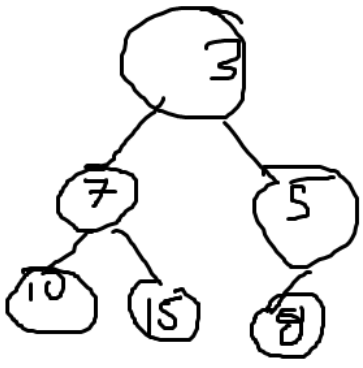
array = [3,7,null,10,15,8]
indexMap = {“3”: 0, “7”: 1, “8”: 5, “10”: 3, “15”: 4}

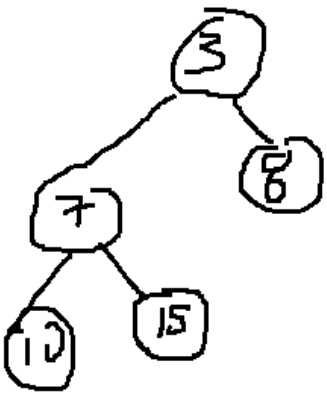
Then we swap the 3 and null to get

array = [null,7,3,10,15,8]

then we remove the min which is null using usual dequeue approach and get

array = [3,7,8,10,15]





Here is the full implementation.

```
public class Ticket<T extends Comparable<T>> {

    private T value;

    public Ticket(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

}

public class PriorityQueueWithDeletions<T extends Comparable<T>> {

    private List<T> arr;

    private Map<T, Integer> indexMap;

    private int getParent(int i) {
        return ((i+1)/2)-1;
    }

    private int getLC(int i) {
        return 2*i+1;
    }

    private int getRC(int i) {
        return 2*i+2;
    }

    private void swap(int i, int j) {
        T tmp = arr.get(i);
        arr.set(i, arr.get(j));
        arr.set(j, tmp);
        // update indexMap
        if (arr.get(i) != null) indexMap.put(arr.get(i), i);
        if (arr.get(j) != null) indexMap.put(arr.get(j), j);
    }

    private int minVertexOrChildren(int i) {
```

```
        int min = i;
        if (getLC(i) < arr.size() && arr.get(getLC(i)).compareTo(arr.get(min)) < 0)
            min = getLC(i);
        if (getRC(i) < arr.size() && arr.get(getRC(i)).compareTo(arr.get(min)) < 0)
            min = getRC(i);
        return min;
    }

    private void heapify(int i) {
        if (i < arr.size()) {
            int j = minVertexOrChildren(i);
            if (j != i) {
                swap(i,j);
                heapify(j);
            }
        }
    }

    // creates an empty priority queue
    public  PriorityQueueWithDeletions() {
        this.arr = new ArrayList<>();
        this.indexMap = new HashMap<>();
    }

    // adds a new element to the queue - returns ticket used to remove item
    public Ticket<T> enqueue(T item) {
        arr.add(item);
        // update indexMap
        indexMap.put(item, arr.size()-1);

        var i = arr.size() - 1;
        while (i > 0 && arr.get(i).compareTo(arr.get(getParent(i))) < 0) {
            swap(i, getParent(i));
            i = getParent(i);
        }
        return new Ticket<T>(item);
    }

    // removes the oldest element from the queue and returns it.
    public T dequeue() {
        T ret = arr.get(0);
        arr.set(0, arr.get(arr.size()-1));
        arr.remove(arr.size()-1);
        // update index Map
```

```
        indexMap.remove(ret);
        indexMap.put(arr.get(0), 0);

        if (size() > 0)
            heapify(0);
        return ret;
    }

    public Boolean greaterThan(T a, T b) {
        if (a == null) return false;
        if (b == null) return true;
        return a.compareTo(b) > 0;
    }

    // removes an element by ticket and returns the removed element
    public T removeByTicket(Ticket<T> t) {
        Integer index = indexMap.get(t.getValue());
        T value = arr.get(index);
        arr.set(index, null);  // treat as smallest possible value
        // update indexMap
        indexMap.remove(value);

        var i = index;
        while (i != 0 && greaterThan(arr.get(getParent(i)), arr.get(i))) {
            swap(i, getParent(i));
        }
        dequeue();
        return value;
    }

    // returns the number of elements in the queue
    public int size() {
        return arr.size();
    }

    // checks if the queue is empty
    public boolean isEmpty() {
        return arr.isEmpty();
    }

}
```

B. (10 pts) Write a data structure that supports all operations of a stack, and in addition supports getMax. Specifically, you need to write the class StackWithMax.
The running time of each operation must be O(1).

```
public class StackWithMax<T extends Comparable<T>> {

        public StackWithMax() - creates an empty stack

        public void push(T item) - adds a new element to the stack.

        public T pop() - removes the element from top and returns it.

        public T getMax() - returns the maximum in the stack (without
        modifying the stack).

        public int size() - returns the number of elements in the stack

        public boolean isEmpty() - checks if the stack is empty

}
```

The running time of each operation must be O(1).

Explain your answer in detail.

Solution

We can accomplish this using two stacks. First, I will give illustrative example. Suppose we have following input:

3, 15, 8, 12, 17

the stack will be like this (remove index 0 to get top of stack).

[17, 12, 8, 15, 3]
[17, 15, 15, 15, 3]

for first push, max is 3. For second push, max (3, 15) = 15. For third push, max (8, 15) = 15. And so on. And when we remove, we have all the previously computed max values.

Here is the full implementation.

```
public class StackWithMax<T extends Comparable<T>> {

    private List<T> s1;

    private List<T> s2;

    // creates an empty stack
```

```
    public StackWithMax() {
        this.s1 = new ArrayList<>();
        this.s2 = new ArrayList<>();
    }

    // adds a new element to the stack
    public void push(T item) {
        s1.add(0, item);
        T max = (!s2.isEmpty() && item.compareTo(s2.get(0)) < 0) ? s2.get(0) : item
;
        s2.add(0, max);
    }

    // removes the element from top and returns it
    public T pop() {
        s2.remove(0);
        return s1.remove(0);
    }

    // returns the maximum of the stack (without modifying stack)
    public T getMax() {
        return s2.get(0);
    }

    // returns the number of elements in the stack
    public int size() {
        return s1.size();
    }

    // check if the stack is empty
    public boolean isEmpty() {
        return s1.isEmpty();
    }

}
```

Problem 2 [25 points]

In this problem use the following definition of Binary Tree. You may assume the classes have the standard getters/setters.

```
public class BTNode<T> {
   private T data;
   private BTNode<T> leftChild;
```

```
    private BTNode<T> rightChild;
    private BTNode<T> parent;

}

public class BinaryTree<T> {
    private BTNode<T> root;

}
```

A. (7 pts) Write a method that gets a binary search tree and checks if it is an AVL tree. You may assume that the given tree is a binary search tree.
Explain your answer.

Solution

The AVL property is |height(v.leftSubtree) – height(v.rightSubtree)| for each vertex v of the tree. We can traverse the tree using pre-order (left, right, root) and check that the AVL property is satisfied.

The height of a given node will be max(left_height, right_height) + 1 with base case that height of a null node is 0.

The condition to check is abs(left_height – right_height) <= 1 for a valid AVL tree. Note that if abs(left_height – right_height) >= 2, it is not a valid AVL tree.

Also, the left and right subtrees must both be valid a AVL tree for the root to be a valid AVL tree.

Here is the full implementation.

```
public Boolean isAVLTreeHelper(NodePair rootP) {
        if (rootP.node == null) {
            rootP.height = 0;
            return true;
        }

        var leftP = new NodePair<T>(rootP.node.getLeftChild(), rootP.height);
        var rightP = new NodePair<T>(rootP.node.getRightChild(), rootP.height);
        var leftB = isAVLTreeHelper(leftP);  // true if left is AVL tree
        var rightB = isAVLTreeHelper(rightP);  // true if right is AVL tree

        rootP.height = 1 + Math.max(leftP.height, rightP.height);

        if (Math.abs(leftP.height - rightP.height) >= 2) {
```

```
            return false;
        }

        return leftB && rightB;
    }

    public Boolean isAVLTree(BTNode<T> root) {

        var rootP = new NodePair<T>(root, 0);
        return isAVLTreeHelper(rootP);
    }
```

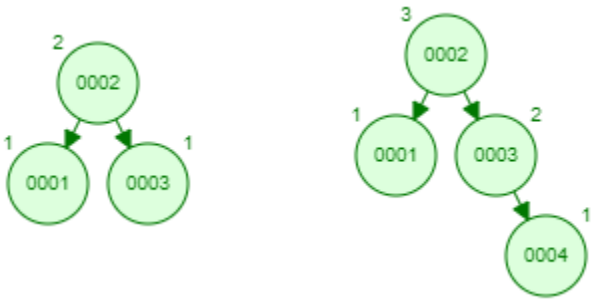
B. (18 pts) For each of the following statements decide if they are true or false. Explain your answers. A correct T/F answer without a correct explanation will give 1/5 points.

1. (2 pts) Insertion into an AVL tree always increases the number of leaves.

   Solution

   False.

   Consider constructing an AVL tree from the numbers 1, 2, 3 and then adding 4 to the AVL tree. The images for this was produced using our usual visualizer https://www.cs.usfca.edu/~galles/visualization/AVLtree.html.



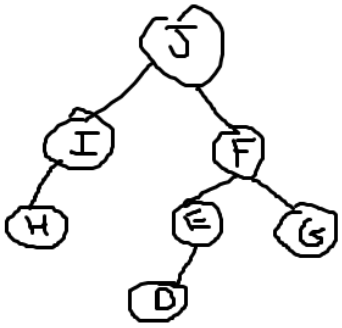
   Both of these trees have 2 leaves. So, we have added an element to the tree without increasing the number of leaves.

2. (4 pts) An AVL tree of height 3 has at least 7 nodes and at most 15 nodes

   Solution

True.

The smallest AVL tree of h =3 looks like this (7 nodes):



And the largest would have all levels filled which would be size = 1 + 2 + 4 + 8 = 15 nodes.

3. (6 pts) If inserting a node into an AVL tree requires rebalancing, then the height of the tree does not change.

   Solution

   True.

   With an optimal AVL implementation, we will fill each level before adding nodes to the levels below.

   Consider case where we have two nodes are adding a third node. On first level, there is one node, and now on second level there will be two nodes. The height of the tree doesn't change.

   Now consider case where we have three nodes and are adding a fourth node. On the first level, there is on node and on the second level there are two nodes. Both levels are full, so we need to create a new level with one node. The height of the tree increases.

   In fact, the height of the tree will only increase in the case when all the levels are full and we need to start a new level. In this situation, we never need to rebalance.

   Therefore, we never rebalance when height increases. And the following statement is also true: we only rebalance when the height stays the same.

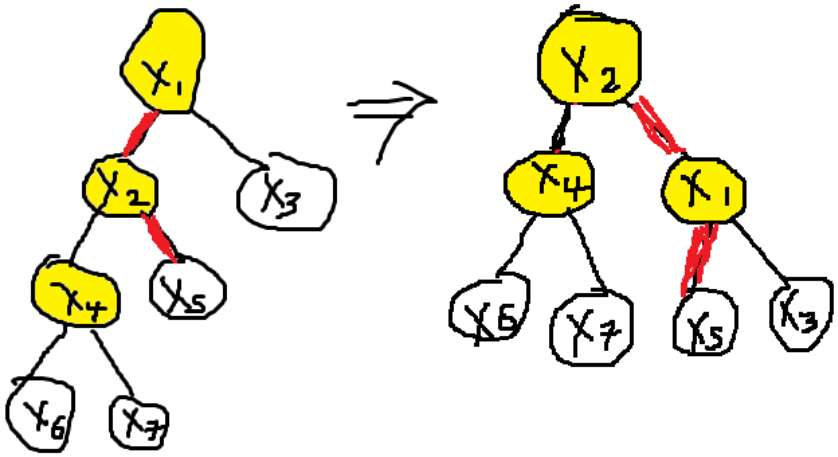
To summarize, when we are rebalancing, the height of the tree doesn't change. That's the whole point of rebalancing.

4. (6 pts) When removing a node from an AVL tree with n nodes, then at most 3 vertices need to be rebalanced.

   Solution

   True.

   A rotation consists of re-assigning left, right and parent of a few nodes. For the example below we are re-assigning left, right and parent for nodes connected to x1, x2 and x4.



   This is what allows the AVL algorithm to have its O(log(n)) time complexity. Finding the node to remove takes O(log(n)) and the rotation can be done in constant time. The rotation time doesn't increase as n increases – it is constant.

Problem 3 [25 points]

A. (20 pts) Write a method that gets a 2d grid given as an array of ints with 0s and 1s. The goal is to find a path from the top left corner of the matrix to the bottom right using the minimal number of steps when you are only allowed to step on the 1s of the array. Write an algorithm that solves this problem and prints the path. In the end of the algorithm the input needs to be in the same state as it was in the beginning.

   For example, for the input below, the path from grid[0][0] to grid[5][4] is marked in

red, and consists of 10 ones. Note that you need to find the shortest path. If there are several shortest paths, your algorithm needs to print only one of them.

```
int[][] board = {
        {1,1,1,1,1},
        {1,0,0,0,1},
        {1,0,1,1,1},
        {1,1,1,0,0},
        {1,1,1,1,1},
        {0,0,1,0,1}
};
```

Before writing the algorithm, explain your answer.

Solution

This is a maze problem. We can treat the 0's as walls. We can represent the board as a graph. For top left square (0, 0), the neighbors are the square immediately to the right (0, 1) and the square immediately below (1, 0). Squares which are outside the dimensions of the board or squares which are 0's are not neighbors. Here is the full graph for provided example.

(0, 0): [(1, 0), (0, 1)]
(0, 1): [(1, 1), (0, 0), (0, 2)]
(0, 2): [(1, 2), (0, 1), (0, 3)]
(0, 3): [(1, 3), (0, 2), (0, 4)]
(0, 4): [(1, 4), (0, 3)]
(1, 0): [(0, 0), (2, 0), (1, 1)]
(1, 1): []
(1, 2): []
(1, 3): []
(1, 4): [(0, 4), (2, 4), (1, 3)]
(2, 0): [(1, 0), (3, 0), (2, 1)]
(2, 1): []
(2, 2): [(1, 2), (3, 2), (2, 1), (2, 3)]
(2, 3): [(1, 3), (3, 3), (2, 2), (2, 4)]
(2, 4): [(1, 4), (3, 4), (2, 3)]
(3, 0): [(2, 0), (4, 0), (3, 1)]
(3, 1): [(2, 1), (4, 1), (3, 0), (3, 2)]
(3, 2): [(2, 2), (4, 2), (3, 1), (3, 3)]
(3, 3): []
(3, 4): []
(4, 0): [(3, 0), (5, 0), (4, 1)]
(4, 1): [(3, 1), (5, 1), (4, 0), (4, 2)]

(4, 2): [(3, 2), (5, 2), (4, 1), (4, 3)]
(4, 3): [(3, 3), (5, 3), (4, 2), (4, 4)]
(4, 4): [(3, 4), (5, 4), (4, 3)]
(5, 0): []
(5, 1): []
(5, 2): [(4, 2), (5, 1), (5, 3)]
(5, 3): []
(5, 4): [(4, 4), (5, 3)]

Once we have the graph, we can apply BFS(graph, src, dest) where src is (0,0), i.e. the top left corner and dest is (5,4), i.e. the bottom right corner. For the BFS algorithm, store the initial node in a queue. Mark that node as visited. Mark distance for that node as 0. Then while we are not at the destination, pop the current node, add the neighbors we haven't visited yet to the queue (if the current path to them is optimal) and mark them as visited. Keep track of the parent of these neighbors.

Once we have reached the destination, traverse backwards through the parents to get the path.

In this example a shortest path is
[(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (4, 2), (4, 3), (4, 4), (5, 4)]

For the algorithm, there are three parts:

- build_graph: builds the graph (vertices and adjacency list for each vertex). Basically, we are figuring out the neighbors for each vertex.
- shortest_path: construct the path from dest to src using parent info.
- solve: perform BFS from src to dest and return shortest_path once path found.

Here is the full implementation.

```
private static Map<String, List<Integer[]>> build_graph(int[][] grid) {

    var graph = new HashMap<String, List<Integer[]>>();

    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            var nb = new ArrayList<Integer[]>();
            if (i-1 >= 0 && grid[i][j] == 1) nb.add(new Integer[]{i-1, j});
            if (i+1 < grid.length && grid[i][j] == 1) nb.add(new Integer[]{i+1, j});
            if (j-1 >= 0 && grid[i][j] == 1) nb.add(new Integer[]{i, j-1});
            if (j+1 < grid[i].length && grid[i][j] == 1) nb.add(new Integer[]{i, j+1});
            graph.put(String.format("%d-%d", i, j), nb);
        }
```

```
        }

        return graph;
    }

    private static List<Integer[]> shortest_path(Map<String, Integer[]> prev,
            Integer[] src, Integer[] dest) {

        List<Integer[]> path = new ArrayList<>();
        Integer[] u = dest;
        while (!u.equals(src)) {
            path.add(0, u);
            u = prev.get(String.format("%d-%d", u[0], u[1]));
        }
        path.add(0, src);
        return path;
    }

    private static List<Integer[]> solve(Map<String, List<Integer[]>> graph,
        Integer[] src, Integer[] dest) {

        String destk = String.format("%d-%d", dest[0], dest[1]);
        Map<String, Integer> dist = new HashMap<String, Integer>();
        Map<String, Integer[]> prev = new HashMap<String, Integer[]>();

        for (Entry<String, List<Integer[]>> entry : graph.entrySet()) {
            dist.put(entry.getKey(), Integer.MAX_VALUE);
            prev.put(entry.getKey(), null);
        }

        List<Map.Entry<Integer, Integer[]>> q = new ArrayList<>();
        q.add(new AbstractMap.SimpleEntry<>(0, src));

        while (!q.isEmpty()) {

            Entry<Integer, Integer[]> entry = q.remove(0);
            Integer distu = entry.getKey();
            Integer[] u = entry.getValue();
            String uk = String.format("%d-%d", u[0], u[1]);

            if (uk.equals(destk)) {
                return shortest_path(prev, src, u);
            }
```

```
            var nb = graph.get(uk);
            for (Integer[] v : nb) {
                Integer alt = distu + 1;
                String vk = String.format("%d-%d", v[0], v[1]);
                if (alt < dist.get(vk)) {
                    dist.put(vk, alt);
                    prev.put(vk, u);
                    q.add(new AbstractMap.SimpleEntry<>(alt, v));
                }
            }
        }

        return Arrays.asList();
    }

    public static void printPath(int[][] grid) {
        var graph = build_graph(grid);
        var src = new Integer[]{0, 0};
        var dest = new Integer[]{grid.length-1, grid[0].length-1};
        var path = solve(graph, src, dest);
        path.forEach(p -> System.out.printf("%d,%d%n", p[0], p[1]));
    }
```

Output for provided example:

0,0
1,0
2,0
3,0
4,0
4,1
4,2
4,3
4,4
5,4

B. (5 pts) What is the running time of your algorithm when the input is an nxn array? Explain your answer.

Solution

BFS involves visiting every node once. There are potentially N^2 nodes. The time complexity is O(N^2). Note these are at most 4 neighbors for each node. This doesn't

increase with N. This is why the neighbors do not affect the time complexity, it is still O(N^2).

Some more details:

- Inserting and retrieving a distance from a HashMap is O(1).
- Storing and retrieving a parent from a HashMap is O(1).
- Building the graph takes O(N^2) time.
- Traversing the graph takes O(N^2) time.

Problem 4 [25 points]

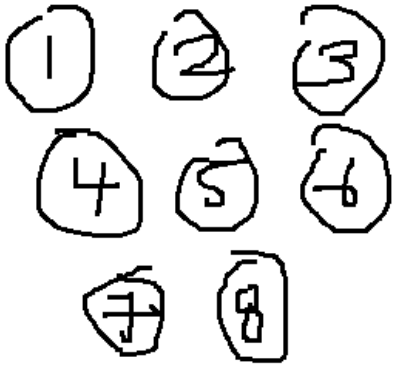
A. (8 pts) Draw the representation of union-find data structure after the following sequence of operations. Draw some intermediate steps.
   - For i=1...8
     makeset(i)
   - union(1,2)
   - union(3,4)
   - union(3,5)
   - union(1,6)
   - union(1,7)
   - union(8,1)

Solution

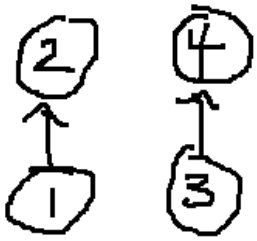
Assume we are using following implementation for union-find data structure.

```
public static Node makeSet(Object data) {
    return new Node(data);
}
public static Node find(Node node) {
    if (node.getParent() == null) return node;
    else return find(node.getParent());
}
public static <T> void union(Node u, Node v) {
    Node ru = find(u);
    Node rv = find(v);
    if (ru.getRank() > rv.getRank()) rv.setParent(ru);
    else if (ru.getRank() < rv.getRank()) ru.setParent(rv);
    else {  // ru.getRank() == rv.getRank()
        ru.setParent(rv);
        rv.setRank(rv.getRank()+1);
    }
}
```

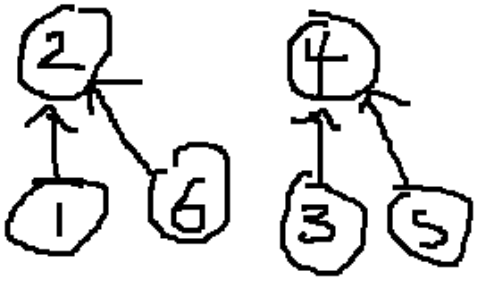
makeSet just creates a node, so after makeset(1), … , makeset(8) we have:



after union(1,2) and union(3, 4) we have (nodes 5-8 not shown, no change).

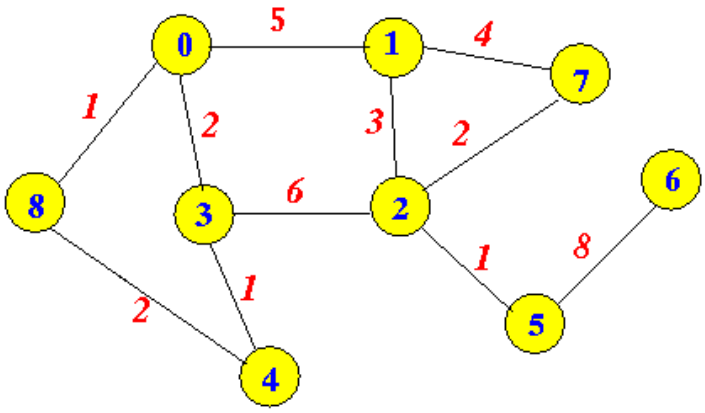


after union(3,5) and union (1,6) we have:

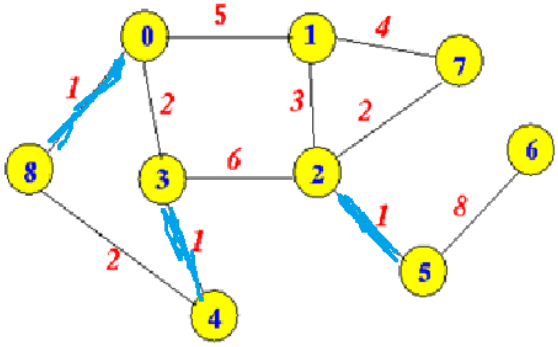


finally, after union (1, 7) and union (8, 1) we have:

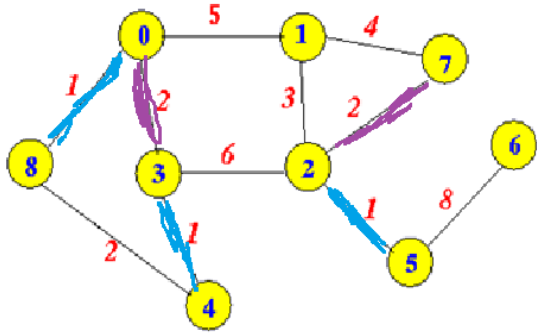
B. (7 pts) Show the execution of Kruskal's algorithm on the following graph.



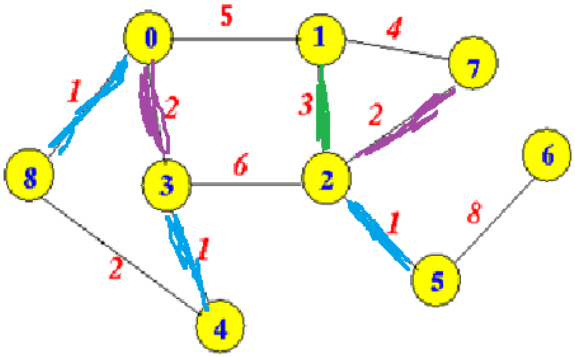
Explore edges with distance 1 first. After this, we have visited edges (0, 8), (3,4) and (2,5).



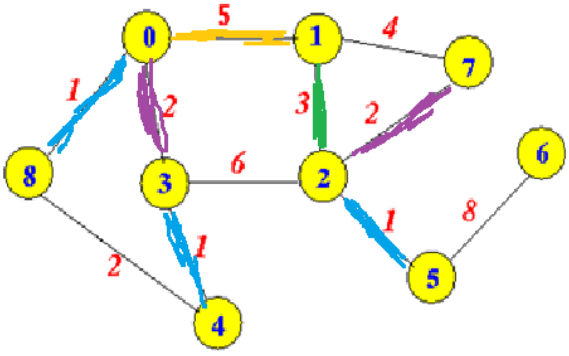
Next explore edges with distance 2. Note: we shouldn't visit (8,4) because is would form a cycle. After this we have visited edges (0, 8), (3,4), (2,5), (0,3) and (2,7).



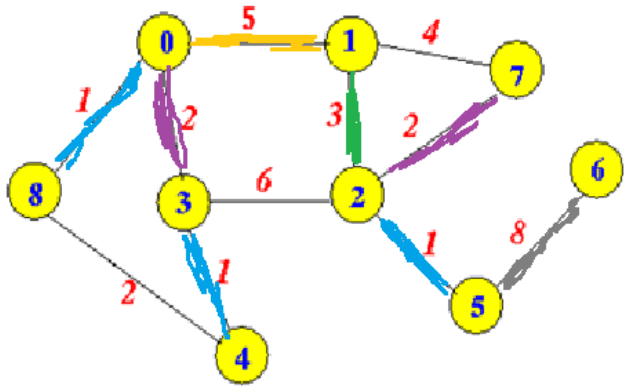
Next explore edges with distance 3. Note, for distance 4 there is nothing to do since we don't want to explore (1,7) since that would form a cycle. At this point we have visited edges (0, 8), (3,4), (2,5), (0, 3), (2, 7) and (1,2).



Then we explorer edges with distance 5. After this we have explored (0, 8), (3,4), (2,5), (0, 3), (2, 7), (1,2) and (0, 1).



For distance 6, there is nothing to do. We shouldn't explore (3, 2) since it would form a cycle. After this we have explored (0, 8), (3,4), (2,5), (0, 3), (2, 7), (1,2), (0, 1) and (5, 6).



The minimum spanning distance is sum of the 8 distances shown below.

(0, 8): 1
(3,4): 1
(2,5): 1
(0, 3) : 2
(2, 7): 2
(1, 2): 3
(0, 1): 5
(5, 6): 8

Which equals to 23.

C. (10 pts) Write a function that gets an array A of length n of int, and an integer k, such that $0 \le k \le 2n/\log_2(n)$ The function needs to permute the elements of A so that the first k elements of A are the smallest numbers sorted in the non-decreasing order. The **running time** must be **O(n)** and **extra space used** should be **O(1)**.

For example, on input A =[3,1,8,2,6,11,4,12,5,7,10,9] and k=6 the resulting array A needs to have [**1,2,3,4,5,6**] as its first six elements, and the rest can be any permutation of the remaining elements. For example [**1,2,3,4,5,6**,12,7,9,8,10,11] The order of the elements after in the last n-k positions is not important.

Explain your idea before writing the code.

Solution

Here are the steps for the solution.

- Find kth smallest number using quickselect. Average time O(n) and space O(1).
- Move k smallest numbers to first k indices. Time O(n) and space O(1). Done during quickselect step.
- Sort first k indices using quicksort. Time O(n) and space O(1).

Proof that sort first k indices takes O(n) time.

Use the fact that quicksort of first k indices takes average time O(k*log(k)).

Next, let k = 2n/log(n) which is the max value of k.

O(k*log(k))
= O (2n/log(n) * log(2n/log(n)))
= O(n*(log(2n) - log(log(n)))/log(n)))
= O(n * (log(n) + log(2) - log(log(n))) / log(n))
= O(n)

Pseudo code for quicksort and quickselect is below. Pseudocode for quicksort was obtained from https://en.wikipedia.org/wiki/Quicksort. Note that quickselect uses same partition function, but the logic in quickselect is a little different.

```
# NOTE: pivot = arr[r]
def partition(arr, l, r):
    x, i = arr[r], l
    for j in range(l, r):
        if arr[j] <= x:
            i, arr[i], arr[j] = arr[j], arr[i], i+1
    arr[i], arr[r] = arr[r], arr[i]
    return i

# for reference, compare quickselect to quicksort
def quicksort(arr, l, r):
    if lo < hi:
        p = partition(arr, l, r)
        quicksort(arr, l, p-1)
        quicksort(arr, p+1, r)

def quickselect(arr, k, l, r):
    if k > 0 and k <= r – l + 1:
        p = partition(arr, l, r)
        if i-l == k-1: return arr[p]
```

elif index-l > k-1: return quickselect(arr, k, l, p-1)
else: return quickselect(arr, k-p+l-1, p+1, r)

Example
arr = [3,1,5,2,8,12,4]
k = 4
kthSmallest = quickselect(arr, k, 0, 6)
quicksort(arr, 0, k)

Here the kthSmallest = 4.
After quickselect, arr = [3,1,2,4,8,12,5]
After quicksort, arr = [1,2,3,4,8,12,5]

Now all that is left is to implement above approach in Java. Here is the full implementation.

```java
import java.util.Arrays;

public class Solution {
    public static Integer partition(int[] arr, int l, int r) {
        int x = arr[r];
        int i = l;
        for (int j = l; j < r; j++) {
            if (arr[j] <= x) {
                int tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i += 1;
            }
        }
        int tmp = arr[i];
        arr[i] = arr[r];
        arr[r] = tmp;
        return i;
    }

    // return pivot = kthSmallest, move values less than pivot before pivot
    public static Integer quickselect(int[] arr, int k, int l, int r) {
        if (k > 0 && k <= r - l + 1) {
            int p = partition(arr, l, r);
            if (p - l == k - 1)
                return arr[p];
            else if (p - l > k - 1)
                return quickselect(arr, k, l, p - 1);
```

```
            else
                return quickselect(arr, k-p+l-1, p+1, r);
        }
        return -1;
    }

    // sort values between l and r
    public static void quicksort(int[] arr, int l, int r) {
        if (l < r) {
            int p = partition(arr, l, r);
            quicksort(arr, l, p - 1);
            quicksort(arr, p + 1, r);
        }
    }

    public static void main(String[] args) {
        int[] arr = {3,1,5,2,8,12,4};
        int k = 4;
        int kthSmallest = quickselect(arr, k, 0, arr.length - 1);
        quicksort(arr, 0, k);
        System.out.println(kthSmallest);
        System.out.println(Arrays.toString(arr));
    }
}
```