CS 4120: Homework 2

Adam Camilli (aocamilli@wpi.edu)

April 9, 2018

Problem 1

Describe a $O(n)$ time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S. (15 points)

1. Find median. This takes $O(n)$ time in the worst case using the famed median-of-medians selection algorithm.
2. Create an associative array B<key,value> whose keys are equal to the members of S. Let each of these keys i have an associated value x equal to the absolute value of the difference of i from the median (creation of this array is again $O(n)$):

B

i_1	$\|i_i - m\|$
i_2	$\|i_2 - m\|$
...	...

3. Let t be the k-th statistic selection of B's values. That is, let t be equal to the k-th smallest element of x, or the differences of each of S's elements from the median of S. This calculation is once again $O(n)$ in the worst case.
4. Finally, locate the k keys of B with the smallest distance from the median, i.e. the k keys that are less than or equal to t. This loop takes $O(n)$ time in the worst case:

```
for(i=1;i<=n;i++)
  if B.getValue(i) <= s
    print(B.getKey(i));
```

These keys are the answers.

Problem 2

Write a complete pseudocode to find the predecessor of a node in a binary search tree. (10 points)

Intuitively, a method to compute the predecessor can be described in two steps:

1. If the node has a right subtree, return its max element.
2. Otherwise, return the highest ancestor (i.e. the one that is farthest down in tree) of the node, whose right child is also an ancestor of the node (this right child can be the node itself).

A complete pseudocode to accomplish this is as follows:

```
predecessor (BTree tree, Node target):
    if (tree.right != NULL):
        return treeMax(tree.right);
    elif (tree.root == target):
        return NULL;

    Node r = tree.root;
    while (r.left != target && r != target):
        if (r.right == NULL):
            return NULL;
        r = r.right:

    return r;
```

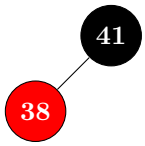
Problem 3

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree. Show all steps. (15 points)

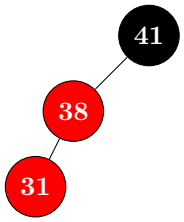
(on next page)

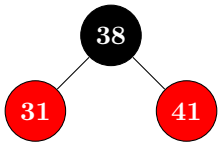
Insert 41:

Insert 38:

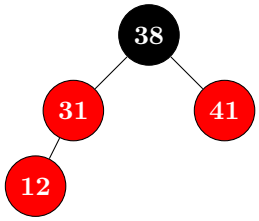


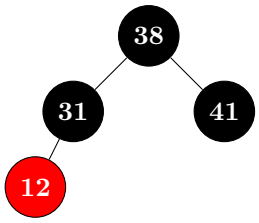
Insert 31: $\longrightarrow$ Case 3: Balance around 38 and recolor 38 black as root:



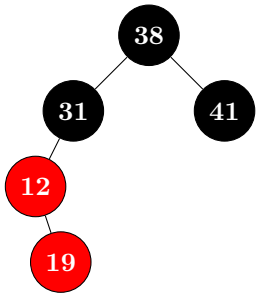


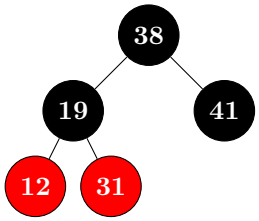
Insert 12: $\longrightarrow$ Case 1: Recolor 31 and 41. Case 0: Recolor root.



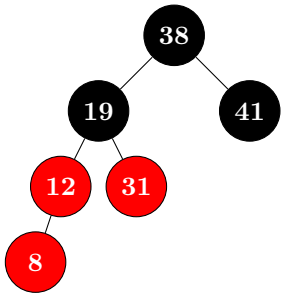


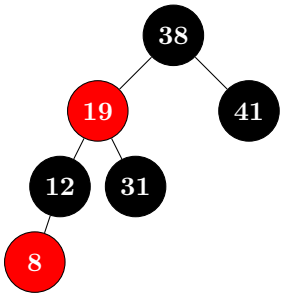
Insert 19: $\longrightarrow$ Case 2: Rebalance subtree around 19. Recolor 19 and 31.





Insert 8: $\longrightarrow$ Case 1: Recolor 19, 12, and 31.





Problem 4

Consider a connected weighted graph where the edge weights are all distinct. Show that for every cycle of the graph, the edge of maximum weight on the cycle does not belong to any minimum spanning tree of the graph. (15 points)

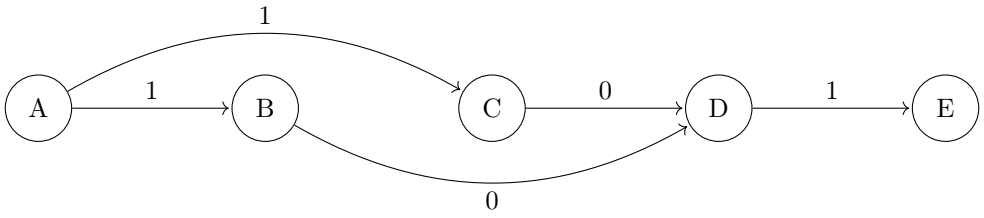
Assume the contrary:

1. Let edge F belong to a minimum spanning tree T and have the largest weight in a cycle C.
2. If F is deleted, T will be broken into two different subtrees, one containing `predecessor(`F`)` and the other `successor(`F`)`.
3. Let S be one of these subtrees.
4. Since F belonged to a cycle, there must be an edge E in C that has exactly one endpoint in S.Let T^* be a new spanning tree that contains E.
5. Since `weight(`E`) < weight(`F`)`, `cost(`T^*`) < cost(`T`)`. This produces a contradiction: T was not the minimum spanning tree. ■

Problem 5

A certain professor thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order. (15 points)

Consider the following directed graph with source vertex A and target vertex E:



There are two shortest paths from A to E:

$$ABDE \text{ (3)}$$

$$ACDE \text{ (3)}$$

If the first path is found, the order of Dijkstra's algorithm will be as follows:

$$(A, B) \rightarrow (A, C) \rightarrow (B, D) \rightarrow (D, E) \rightarrow (C, D)$$

This permutation is a contradiction, since edge (C, E) appears before edge (D, E), and should be relaxed first according to the professor's claim. ■

Problem 6

In all the shortest paths algorithms we've learned in class we break ties arbitrarily. Discuss how to modify these algorithms such that, if there are several different paths of the same length, then the one with the minimum number of edges will be chosen. (15 points)

The general answer is simply to keep track of the number of edges for each path when applying the algorithm. A specific example for Dijkstra's algorithm:

1. For any vertex v, let `numEdges(`v`)` be equal to the shortest possible number of edges on a shortest path from s to v P_v.
2. While traversing a graph, let v be the current vertex we have found, and w the next vertex on the adjacency list. Let C(current, adjacent) = the weight of going to an adjacent vertex from the current one, and let D(node) equal the result of Dijkstra's algorithm on a node. We update `numEdges` as follows:
   (a) If $D(v) + C(v, w) = D(w)$, change `numEdges(`w`)` to `numEdges(`v`)` $+ 1$ if `numEdges(`u`)` $+1 <$ `numEdges(`v`)`
   (b) If If $D(v) + C(v, w) < D(w)$, change `numEdges(`w`)` to `numEdges(`v`)` $+ 1$.

Problem 7

In a directed graph a set of paths is edge-disjoint if their edge sets are disjoint, i.e. no two paths share an edge (although they may share vertices). Given a directed graph $G = (V, E)$ with two distinguished vertices s and t, give an efficient algorithm to find the maximum number of edge-disjoint $s - t$ (directed) paths in G. (Hint: Use a flow network.). (15 points)

We can reduce this problem to the maximum flow problem from optimization theory:

1. Let s and t be the source and sink, respectively, in a flow network.
2. Run the Ford-Fuklerson algorithm to find the maximum flow from s (source) to t(sink):
   (a) Let initial flow equal 0.
   (b) While there is an augmenting path from source to sink, add it to a list `flow[]`
   (c) Return `flow[]`
3. This result is equal to the maximum number of $s - t$ edge-disjoint directed paths.