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. _F__ 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. (CHECK AGAIN)
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. (check later)
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. _F_ 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. _T_ 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.

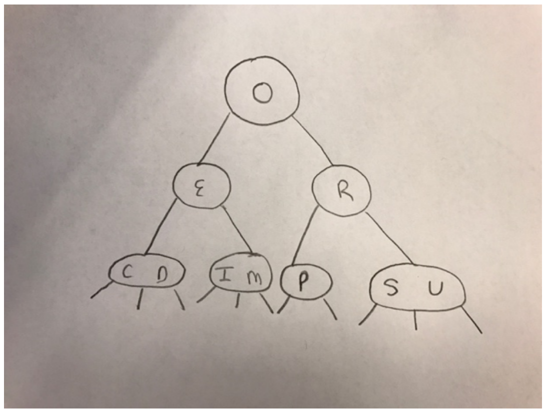
17. _F__ The floor operation returns the smallest key in a symbol table that is greater than or equal to a given key.

18. _T__ The root node in a tree is always an internal node. (Should be true but there is an exception, so idk.)

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

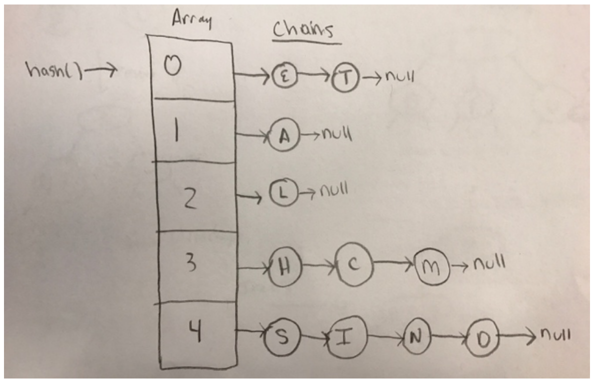
1. (4 pts) Draw the 2-3 tree that results when you insert the following keys (in order) into an initially empty tree:

   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

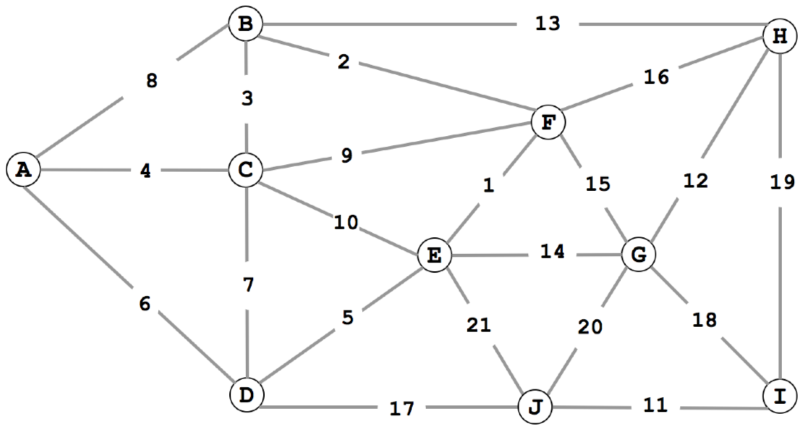


3. (4 pts) List the vertices in the order in which they are visited (for the first time) in DFS for the following undirected graph, starting from vertex 0. For simplicity, assume that the Graph implementation **always iterates through the neighbors of a vertex in increasing order**. The graph contains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



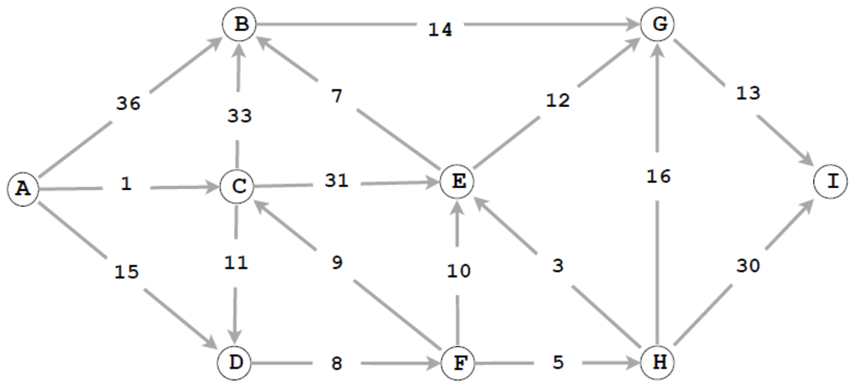
(a) List the sequence of edges in the MST in the order that Kruskal's algorithm includes them (starting with 1).

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

(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.

```
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
```

Step One:	Step Two:	Step Three:	Step Four:
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	Yuan

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 k^{th} smallest element.

   Heap. A min heap would require k remove() operations to retrieve the kth smallest element.

2. Find the last element inserted.

   Linked List. This is the only data structure option that preserves the order in which elements are inserted, and this would be marked as the tail of the LL.

3. Find the first element inserted

   Linked List. This data structure preserves the order in which the elements are inserted, and this would be the head of the list.

4. Guarantee constant time access to any element.

   Hash table. By the Uniform Hashing Assumption, this is inherently true for hash tables.

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.

```
public class Graph {
    private static final String NEWLINE =
System.getProperty("line.separator");

    private final int V;
    private int E;
    private Bag<Integer>[] adj;
    private int p_lines;
    private Boolean[][] neighbors = new Booolean[V][V];

    /**
     * Initializes an empty graph with {@code V} vertices and 0
edges.
     * param V the number of vertices
     *
     * @param  V number of vertices
     * @throws IllegalArgumentException if {@code V < 0}
     */
    public Graph(int V) {
        if (V < 0) throw new IllegalArgumentException("Number of
vertices must be nonnegative");
        this.V = V;
        this.E = 0;
        this.p_lines = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }
public void addEdge(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        E++;

        If(this.neighbors[v][w] == true){
                This.p_number++;
        }
        else{
        this.p_number += 2;
        this.neighbors[v][w] = true;
        this.neighbors[w][v] = true;
        }
        adj[v].add(w);
        adj[w].add(v);
    }
public int p_number_count(){
        return this.p_number;
}
```

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.

By adding edge e to the minimum spanning tree, we create a unique cycle in the graph. In order to find the MST of the new tree, we delete the maximum weighted edge of the cycle, which can be achieved in time proportional to 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, and clarity.

   This data type involves the use of two heaps, one minimum and one maximum. A global variable 'int median' will exist and be initialized for the purpose of the algorithm.

```
Insert(int n){
    If( n < median){ max_heap.insert(n); //logN time for heap insertion}
    Else { min_heap.insert(n); } //logN time


    Resize();                //this method is included to rebalance the two heaps if the
                             //size of diff of the heaps exceeds 1
    median = find_median(); //reset global variable using find_median method
}


int find_median(){
    //if max and min are equal, return smaller of the two
    if(max_heap.size() == min_heap.size()) { return max_heap.max() }
    //return max val of vals < the original median
    else{ return max_heap.max();}
}


int remove_median(){
    return max_heap.removeMax(); //heap deletion is logN time
```

}

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.

The following data structure is a modification to a Red-Black Binary Search Tree, in which the query function acts as a floor search for the given search term. The methods are implemented as follows:

```
Public class NearestNeighbor{

   RBBST tree;

   Public NearestNeighbor(){

         tree = new RBBST(); //the constructor function initializes

         //an empty red black binary search tree

   }


   public void  insert(real x){


         this.tree.insert(x); //RBBST has logN insertion

   }
```

```
      public real query(real y){

            if(this.tree.contains(y)){ return y;}

            if(this.tree.isEmpty()){return 0;}

             return recQuery(y, this.tree.root, Integer.MAX_VALUE);

      }



      private recQuery(real y, RBBST_Node current, RBBST_Node closest,
real difference){

                  if(current == null) { return closest.value;}

                  else if(current.value < y){

                        dif = Abs(y - current.value);

                        if(dif < difference){

                        return recQuery(y, current.right, current, dif);

                        }

                        else{ return recQuery(y, current.right, closest,
difference);}

                  }

                  else{

                        dif = Abs( y - current.value);

                        if(dif < difference){
```

```
                    return recQuery(y, current.left, current, dif);

                    }

                    else{ return recQuery(y, current.left, closest, 
difference);}

                    }

     }



}
```