Introduction to Data Structures
Spring 2021 Final

Name:
Student ID:

1. **(20 pts, True or False) For each question, answer with "True" or "False". You don't need to explain your answer.**
   (a) If $T_1(N) = O(f(n))$ and $T_2(N) = O(f(n))$, then $T_1(N) = O(T_2(N))$
   (b) Every binary search tree on *n* nodes has height O(log *n*).
   (c) Insertion in a binary search tree is commutative.
   (d) In a binary search tree, it takes O(1) time on average to find the next largest element.
   (e) The breadth-first search uses the queue.
   (f) It is possible to have a comparison sort algorithm, which sorts 5 numbers that uses at most 6 comparisons in the worst case.
   (g) Suppose that a hash table, which resolves the collision by chaining, contains n items where its load factor α=1/(log n). Assuming a simple uniform hash function, the expected time to successfully search for an item takes O(1).
   (h) The worst-case performance of mergesort is O(n log n).
   (i) Dijkstra's algorithm is an example of a greedy algorithm.
   (j) Let *M* be a minimum spanning tree of *G*. For any pair of vertices *s* and *t*, the shortest path from *s* to *t* in *G* is the path from *s* to *t* in *M*.

2. **(20 pts, Asymptotic analysis)** For each pair of expressions (A, B) in the table below, where A is O, o, Ω, ω, or Θ of B. (e.g., A = O(B)), each empty cell (from a-1 to d-5) represents the asymptotic relationship between A and B. Assuming k >=1, ϵ > 0, and c > 1 are constants, fill all the 20 empty cells with either "true" or "false". For example, (a-1) is "true" if $\lg^k n = O(n^\epsilon)$ is true.

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	(a-1)	(a-2)	(a-3)	(a-4)	(a-5)
n^k	c^n	(b-1)	(b-2)	(b-3)	(b-4)	(b-5)
2^n	$2^{n/2}$	(c-1)	(c-2)	(c-3)	(c-4)	(c-5)
lg(n!)	$\lg(n^n)$	(d-1)	(d-2)	(d-3)	(d-4)	(d-5)

3. **(10 pts, Quicksort)** Proof that the runtime complexity of quicksort in the average case is O(*n* log*n*). Start your proof with the recurrence relation of T(*n*) (which represents the runtime of quicksort), and you should clearly show all necessary steps to reach the conclusion.

4. **(10 pts, Hash tables)** Suppose you are given two hash tables, each of which takes its own open addressing mechanism to resolve collisions. In the case of linear probing, the behavior follows

   `position(k, M, i) = (hash(k) + i) % M,`

   where `position(k, M, i)` returns the position within the base table with the size of `M` for the given key `k` at `i`-th trial (`i` starts from zero). In the case of quadratic probing, the behavior follows

   `position(k, M, i) = (hash(k) + 0.5*i + 0.5*i*i) % M.`

   Assume the initial table size is 16. Suppose you are inserting following keys in order:

   8, 3, 5, 21, ~~3~~, 10, 11

   For each hash table setup, show how each key will be placed within the table in the end.

   (a) (5 pts) h(x) = x % 16, linear probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

   (b) (5 pts) h(x) = x%16, quadratic probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

5. (30 pts, Augmenting data structures) Often, you will need to consider *augmenting* data structures to add helpful features. We'll practice data structure augmentation by adding a function called *RB-enumerate(n, a, b)* to the red-black tree. `RB-enumerate(n, a, b)` outputs the keys k, within a subtree rooted at n, such that $a \leq k \leq b$. For example, suppose there is an red-black tree with nodes 1, 2, …, 10. Then, `RB-enumerate(root, 3, 7)` will output 3, 4, …, 7. You can assume that there are *no redundant keys* in the tree.
(*Rule: Try your best to make sure your answer is written in valid C/C++ syntax. Yet, minor grammar errors are acceptable. You will get penalty if your answer doesn't look like C/C++ at all, or has significant flaws*)

(a) First, try to implement RB-enumerate(n, a, b) for a vanilla red-black tree. A vanilla red-black tree node has the following attributes:

```
enum class color : uint_8 {
    red,
    black,
};

struct rbnode {
    uint32_t key;
    color color_; // color::red or color::black
    struct rbnode* left;
    struct rbnode* right;
    struct rbnode* parent;
};
```

Implement RB-enumerate(n, a, b) using the above attributes (*You are not allowed to add fields to* `struct rbnode`.). For the sake of simplicity, make your RB-enumerate(n, a, b) just print (i.e., `std::cout << node.key`) all keys. (*Hint: Given a node, how can we find the node with the next smallest key?*)

```
// You can declare and implement helper functions if you want.
// You can add member functions to `struct rbnode` if you want.


void rb_enumerate(struct rbnode *n, uint32_t a, uint32_t b) {
    // FILL IN HERE
}
```

(b) Now, it's time to add an attribute to a vanilla red-black tree node, which helps us run RB-enumerate(n, a, b) in O(m + logn) time, where m is the number of outputs of RB-enumerate(n, a, b) and n is the number of nodes in the tree. Answer following three sub-questions.

(b)-①. Select an attribute that can improve the efficiency of RB-enumerate(n, a, b), and add that attribute to below red-black tree node definition. Also, explain what the attribute you selected should do (Hint: You may want to add only one new attribute).

```
struct rbnode {
    uint32_t key;
    color color_; // color::red or color::black
    struct rbnode* left;
    struct rbnode* right;
    struct rbnode* parent;
    // FILL IN HERE

};
```

(b)-②. Improve the above implementation with this new attribute.

```
// You can declare and implement helper functions if you want.


void rb_enumerate(struct rbnode *n, uint32_t a, uint32_t b) {
    // FILL IN HERE
}
```

(b)-③. Validate that the improved algorithm runs in O(m + logn) time (Just providing a high-level idea is good enough).

(c) After you add a new attribute to your data structure, you want to ensure that the rest operations (e.g., insert, rotate-right, rotate-left, etc.) do not break the semantics of the new attribute. Here are example implementations for insert, rotate-right and rotate-left for the vanilla (left-leaning) red-black trees.

```
rbnode *insert(rbnode *n, uint32_t key) {
  if (!n)
    n = new rbnode{key, color::red};

  if (key < n->key) {
    n->left = insert(n->left, key);
    n->left->parent = n;
  } else if (key > n->key) {
    n->right = insert(n->right, key);
    n->right->parent = n;
  } else return n; /* Anyway, ignore a redundant key */

  /* Fixup */
  if (is_red(n->right))
    n = rotate_left(n);

  if (is_red(n->left) && is_red(n->left->left))
    n = rotate_right(n);

  if (is_red(n->left) && is_red(n->right))
    n->flip_color();

  return n;
}
```

```
rbnode* rotate_right(rbnode* n) {
  if (!n->left)
    return n;

  auto x = n->left;

  n->left = x->right;
  x->right = n;
  x->color_ = x->right->color_;
  x->right->color_ = color::red;
  x->parent = x->right->parent;
  x->right->parent = x;
  if (x->right->left)
    x->right->left->parent = x->right;
  return x;
}
```

```
rbnode* rotate_left(rbnode* n) {
  if (!n->right)
    return n;

  auto x = n->right;

  n->right = x->left;
  x->left = n;
  x->color_ = x->left->color_;
  x->left->color_ = color::red;
  x->parent = x->left->parent;
  x->left->parent = x;
  if (x->left->right)
    x->left->right->parent = x->left;
  return x;
}
```

Answer following two sub-questions of (c):
(c)-①. Answer which function(s) should be updated among insert, rotate-right and rotate-left (Don't worry about deletion).
(c)-② Provide the updated implementation(s). Write down how you'd change the function(s). You are allowed to change the parameters of the function (e.g., the parameters) if you want.

```
// FILL IN HERE
```

6. (20 pts, AVL Tree) Suppose you want to add a new function `get_kth_smallest(struct avlnode *t, uint32_t k)` with the time complexity `O(log(n))`. Specifically, this function returns a pointer pointing to the node of the k-th smallest value from the AVL subtree, in which the given subtree is rooted at t**.** To do so, you add a new variable: **`size`**; in `struct avlnode`. Answer following two questions.

(a) Explain what information the variable **`size`** should contain.

(b) Complete the implementation of the function, `get_kth_smallest(struct avlnode *t, uint32_t k)`.

```
struct avlnode {
  uint32_t key;
  uint32_t height;
  struct avlnode *left;
  struct avlnode *right;

  uint32_t size;
};

struct avlnode *get_kth_smallest(struct avlnode *t, uint32_t k) {
  // FILL IN HERE

}
```

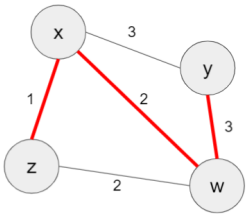
7. (10 pts, AVL Tree) Suppose there is a program which draws the AVL tree structure and you use that program to draw the AVL tree by inserting the following numbers: 9, 7, 2, 8, 6, 4, and 5 (in order) in the tree.

(a) Draw the final result of the AVL tree structure.
(b) Draw the final result of the AVL tree structure after you delete the number 8 from the tree.

8. (30 pts, Graph)

(a) (Topological sort) Prove that in a Directed Acyclic Graph (DAG), there cannot both be a path from ***v*** to ***w*** and a path from ***w*** to ***v,*** *where* ***v*** *and* ***w*** *are two different vertices.*

(b) (Minimum Spanning Tree) A **minimum bottleneck spanning tree (MBST)** in an undirected graph is a **spanning tree** in which the most expensive edge is as cheap as possible.



The above figure shows an example of MBST---i.e., edges (x, z), (x, w), and (y, w). Here, you can see that MBST is quite similar to MST. For the following two statements, write "True" if you think that the question is true. Write "False" if not. If you answered "True", proof it. Otherwise, provide a counterexample.

(b)-①. An MST is necessarily an MBST.
(b)-②. An MBST is necessarily an MST.

Hint: The cut property that we describe below may be helpful to answer this question. You can use the cut property without the proof. You don't need to use the cut property in your proof if you think it's not necessary.

Definition 1. Cut A **cut**(S, V-S) of an undirected graph G = (V, E) is a partition of V. Here, S and V denote a set of vertices, and V-S denotes a set of vertices where vertices in S are discarded from V. E denotes a set of edges.
Definition 2. A cross edge An edge (u, v) of an undirected graph G = (V, E) is called **a cross edge of the cut** if one of its

endpoints is in S and the other is in V-S.

Definition 3. Safe edge
Let G = (V, E) be a connected and undirected graph.
Let A be a subset of some minimum spanning tree.
An edge (u, v) is called a **safe edge** of A if A U {(u, v)} is also a subset of a minimum spanning tree.

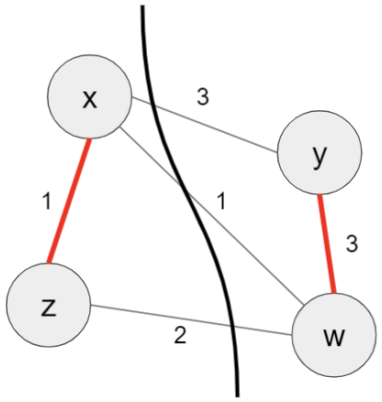
Definition 4. Respect
If no edge in A is a cross edge of the cut(S, V-S) where A is a set of edges, the cut(S, V-S) **respects** A.

Theorem 1. Cut property
Let G = (V, E) be a connected, weighted, and undirected graph.
Let A be a subset of E, which is included in some MST of G.
Let cut(S, V-S) be any cut of G that respects A.
Let (u, v) be the minimum weight edge and the cross edge of the cut(S, V-S).
Then (u, v) is a safe edge of A.

Example.
V = { w, x, y, z }
S = { x, z }
V - S = { y, w }
A = { (x, z), (y, w) }
cut(S, V-S)= { {x, z}, {y, w} }

In this example, you can observe that the cut(S, V-S) respects A because no edge in A is a cross edge of the cut(S, V-S). Moreover, (x, w) is the minimum weight edge and the cross edge of the cut(S, V-S).

Therefore, (x, w) is a safe edge of A by Theorem 1.



This is the last page.