Introduction to Data Structures
Fall 2020 Final

Name:
Student ID:

1. (10 pts, Design) Throughout the semester, we have covered various data structures, where each of which has its own advantage. For a specific application, pick one data structure which would suit the best, out of five data structures that we listed below. For each application, you should shortly justify your answer as well (two sentences at max per application).

List of data structures: (i) Linked list, (ii) Queue, (iii) Binary search tree, (iv) AVL tree, and (v) A hashed table,

(a) (2.5 pts) Application: You are asked to develop a large dictionary, where the input is the word and the output is the definition of the word.

(b) (2.5 pts) Application: You are asked to develop a waiting lounge, implementing first-come, first-served services. In this service, the input is a sequence of persons, and the output is the person to be served next.

(c) (2.5 pts) You are asked to develop an index for the large text document, where the input is the word and the output is true (if the word is used in the text document) or false (if the word is not used). Note that you are asked to optimize for the time initially constructing the index, not for the time taken to query.

(d) (2.5 pts) Application: You are asked to develop a tab completion feature for you editor (i.e., if you type “tab” on an incomplete word, it automatically completes the word based on the words present in a large number of currently editing documents in your editor), where the input is the incomplete word and the output is the suggested word to be completed. Note that the editing documents are assumed to be frequently updated.

2. (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:

48, 32, 16, 0, 1, 10

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	

3. (15 pts, asymptotic complexity) Answer following questions. For true/false questions, you don't need to provide the reasoning.

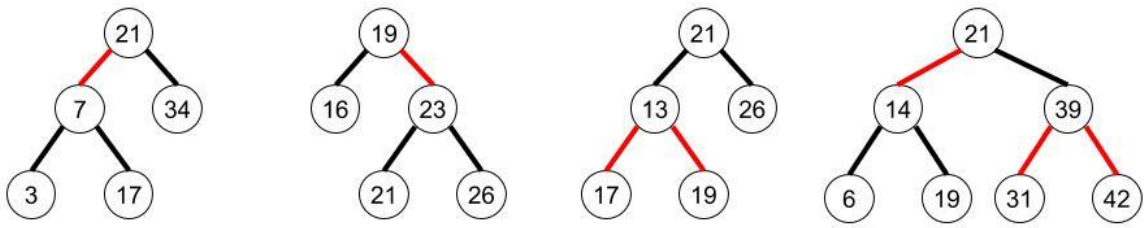
(a) (3 pts) True/False: If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

(b) (3 pts) True/False: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $h(n) = \Omega(f(n))$

(c) (3 pts) True/False: If $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = g(n)$

(d) (6 pts) Show the average depth of a node in a perfect binary tree. You should show how it is derived, meaning that your answer should begin with the sum of the depths and the total number of nodes. Your answer should be represented with the Big Theta notation in the end.

4. (10 pts, Red-black tree) Choose *all* valid *left-leaning red-black trees* among four trees, (a) through (d). Recall that the in-order traversal of a left-leaning red-black tree must output a sequence of values sorted in an *ascending order* (e.g., {1, 2, 3, 4, 5}).



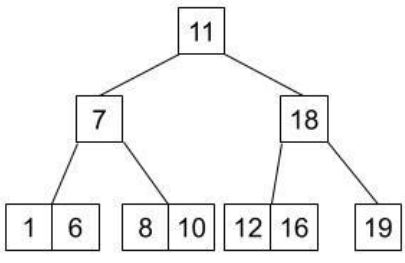
(a) (b) (c) (d)

5. (10 pts, B-tree) The `insert` operation for a B-tree takes as input a value (say an integer), and stores it to the B-tree, while preserving B-tree invariants including:

- Each non-root node has [B-1, 2*B-1] keys
- An in-order traversal of a B-tree should be sorted in an ascending order
- All leaf nodes have the same depth

Given a simplified pseudo-code of the `insert` operation, a B-tree (B = 2) and a sequence of `insert` operations below, draw the B-tree after each `insert` operation. You have to draw five B-trees for your answer (i.e., after `insert(2)`, after `insert(20)`, and so on). Note that you do not have to consider duplicate keys.

```
function insert(node=root, key, value): // Pseudo code for top-down insertion
  if isLeaf(node) && !isRoot(node):
    insertAtLeaf(node, key, value)  // Find a proper position in the leaf
                                    // node and insert the key and value.
                                    // Requires that `node` be NOT full.
    return

  // NOTE: At this point, if the root node is full, it must be split,
  // and the middle key is promoted to the new root node. However,
  // we omit this step to keep it simple.

  edge = findEdge(node, key)  // Find the edge where a new key has to
                              // be inserted.
  if isFull(edge):
    splitChild(node, edge) // Split a full node. The `edge` must be full (i.e.,
                           // (has (2*B - 1) keys). The `edge` will be split into
                           // two nodes with B keys, and the middle key in the
                           // `edge` will be moved to the `node`.
    edge = findEdge(node, key) // A new key is added to `node`. Find the proper
                               // child edge, again

  return insert(edge, key) // Recurse
```



(a)

1. insert(2)
2. insert(20)
3. insert(13)
4. insert(3)
5. insert(4)

(b)

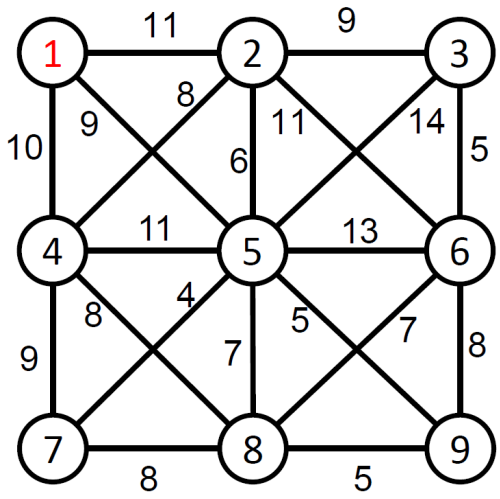
6. (15 pts, Sorting) Assume that you're trying to implement Heapsort by using the max heap. Complete the following pseudo code by filling up five empty lines. (For simplicity, the array index starts with "1".)

- function **`Heapsort`** sorts an array in ascending order by using max heap.
- function **`build_max_heap`** converts the unordered array into a max heap
- function **`max_heapify`** arranges node i and it's subtrees to satisfy the heap property.

```
// array[i] - a value at the node (element) i
// LEFT(i) - an index of left child node (i.e., 2i)
// RIGHT(i) - an index of right child node (i.e., 2i + 1)
// swap(a,b) - swap two values, a and b
// array.length - the total number of elements in the array
// array.heapsize - the number of elements in the heap stored within the array

function Heapsort(array):
    build_max_heap(array)
    for i = array.length downto 2
        (1)________
        array.heapsize = array.heapsize - 1
        (2)________

function build_max_heap(array):
    // use all elements in array to build max-heap
    array.heapsize = array.length
    for i = floor(array.length / 2) downto 1
        max_heapify(array, i)

function max_heapify(array, i):
    L = LEFT(i)
    R = RIGHT(i)
    if L <= array.heapsize  and (3)________
        largest = L
    else
        largest = i
    if R <= array.heapsize  and (4)________
        largest = R
    if largest != i
        swap(array[i], array[largest])
        (5)_________
```

7. (25 pts, Graph Algorithms) Answer the following questions.

a) (5 pts) Draw the minimum spanning tree of the below figure using Prim's algorithm. Suppose that the root is the vertex 1.



b) (5 pts) Here is a pseudo code of Prim's algorithm (We omit the initialization step for simplicity):

```
for _ in range(|V|): // until all the vertices are reached
  // visit the min distance vertex
  v = find_and_remove_min(table)
  // update the distance if needed
  for j in get_adj_vertices(v):
    if (graph.get_dist(v, j) < table.get_dist(j)):
      table.set_dist(j) = graph.get_dist(v, j) // NOTE: set_dist updates
                                               // the distance of a SINGLE
                                               // vertex.
```

Given the above pseudo code, express the time complexity of Prim's algorithm using the following four terms with Big-O notations: |V|, |E|, T(`find_and_remove_min`), and T(`set_dist`). Note that |V| denotes the number of vertices, |E| denotes the number of edges, T(`find_and_remove_min`) denotes the time taken to perform `find_and_remove_min()`, and T(`set_dist`) denotes the time taken to perform `set_dist()`.

c) (5 pts) You can implement the table (of vertices) that supports above operations (i.e., `find_min_remove_min()` and `set_dist()`) using various data structures. In particular, suppose you maintain two different types of data structures to implement the Prim's algorithm: (1) array, (2) AVL tree. The below table summarizes the time complexity of each operation for these data structures:

Data structure \ Operation	T(`find_and_remove_min`)	T(`set_dist`)	The time complexity of Prim's Algorithm
(1) Array	Fill in here	Fill in here	Fill in here
(2) Red-black tree	Fill in here	Fill in here	Fill in here

First, fill the above table and briefly explain your answer. Note that we assume the graph is presented using the adjacency list. You are NOT allowed to augment the data structures, meaning that you should stick to the basic definitions of these data structures.

d) (5 pts) If you maintain the Fibonacci heap, Prim's algorithm can be further optimized. In this case, you will need to implement a feature, which updates the key of the existing in the Fibonacci heap. How would you implement this feature? Shortly describe your ideas in two aspects: (1) how to find the element to be updated; and (2) how to update the key value in the Fibonacci heap.

e) (5 pts) Implementing Dijkstra's algorithm is quite similar to Prim's algorithm. Complete the following pseudo code of Dijkstra's algorithm (i.e., two lines, (i) and (ii))

```
// Pseudo code of Dijkstra's algorithm
for _ in range(|V|): // until visiting all vertices
  // visit the min distance vertex
  v = find_and_remove_min(table)
  // update the shortest path if needed
  for j in get_adj_vertices(v):
    if ( (i) ____________________________ ) :
      (ii) ____________________________
```

8. (15 pts, Heap) Assume that you have a max heap ordered binary heap taking non-duplicate, unsigned integer keys (i.e., non-negative unique integer keys). This binary heap is implemented with the binary tree structure, and supports the basic heap operations:

- `insert`
- `extract_max`
- `is_empty`
- `size`

Our goal is to convert our max heap ordered binary heap into a min heap ordered binary heap without using any additional memory spaces (i.e., O(1) additional memory allocation is fine).

(a) (10 pts) How would you achieve such a goal? Describe your idea. You don't need to construct a concrete algorithm, but you should clearly describe your idea.

Hint: All elements (i.e., $x_1, x_2, \ldots, x_n$) in the max heap satisfy the following property:

$-1 * \max(x_1, x_2, \ldots, x_n) <= \min(x_1, x_2, \ldots, x_n)$

(b) (5 pts) The assumption in the question had a constraint that the max heap only takes non-negative unique integer keys. Can you relax the constraint such that the max heap takes any unique integer (i.e., unique signed integer keys)? Answer "true" if you think this is possible, and show how this can be done. Answer "false", if you think this is not possible, and justify why this is not possible.

This is the last page.