UNIVERSITY OF OTAGO EXAMINATIONS 2025

SCHOOL OF COMPUTING

COSC 201

Algorithms and Data Structures

Semester One

(TIME ALLOWED: 3 HOURS)

This examination paper comprises 10 pages

Candidates should answer questions as follows:

Candidates should answer **All** questions

Marks are shown thus: (2)

The total number of marks for this exam is 75

The following material is provided:

Nil

Use of calculators:

Any model of calculator provided that is battery powered, silent, truly portable and free of communication capabilities

Candidates are permitted copies of:

No additional material

Other Instructions:

Nil

TURN OVER

1. Union-find

The purpose of the union-find data structure is to maintain a dynamic set partition, i.e., a partition of the elements $\{0, 1, 2, ..., n-1\}$ into groups, that supports:

- `union(x,y)` which joins the groups containing x and y (if they are not already in the same group), and
- `find(x)` which determines a representative element of the group that x belongs to. The values of `find(x)` and `find(y)` must be the same whenever x and y belong to the same group (and only then).

(a) Suppose that a union-find instance has been initialised with $n = 10$. The following calls to union are made: `union(2, 5)`, `union(7, 4)`, `union(6, 7)`, `union(5,8)`. All of the following refer to the state of the union- find instance at this point.

(i) What is the value of `find(0)`? (1)

(ii) How many different values of x are there for which `find(x)` and `find(4)` have the same value? (1)

(b) In our first union-find implementation (UF1) an array, `reps`, of representatives was maintained in such a way that the value of `find(x)` was just the element `reps[x]`.

(i) In UF1, what is the cost of a union operation? Justify your answer. (2)

(c) In UF3, `find(x)` is computed using local representatives. When `union(x,y)` is run, the local representative is decided based on the *rank* of `find(x)` and `find(y)` with the chosen representative being the one with the largest rank, or `find(y)` in the case of equal ranks. The rank of an item is just the length of the longest chain of local representatives that reach it. Suppose that a union-find instance has been initialised with $n = 10$. The following shows the state of the `reps` array and `rank` array after `union(2,3)`:

	0	1	2	3	4	5	6	7	8	9
reps	0	1	3	3	4	5	6	7	8	9
rank	0	0	0	1	0	0	0	0	0	0

Show the state of the `reps` array and the `rank` array after each of the following union operations:

(i) `union(4,2)` (2)

(ii) `union(8,9)` (2)

(iii) `union(8,2)` (2)

2. Induction and algorithm analysis.

(a) Consider the following code:

```
public long fib(int n) {
  return fibPair(1,1,n);
}

private long fibPair(long a, long b, int n) {
  if (n == 0) return a;
  return fibPair(b, a+b, n-1);
}
```

The time taken to execute a call to `fib(n)` can be written as:

$$T(0) = C$$
$$T(n) = D + T(n-1)$$

for some constants C and D. Using the substitution method, derive a non-recurrent expression for $T(n)$. *Note:* a recurrent expression has $T(.)$ on both sides of the equal sign as in the expression above. A non-recurrent expression will have $T(n)$ on the left hand-side only and no $T(n-1)$ anywhere. (5)

(b) The definition of $f(n) = O(g(n))$ is:
$f(n) = O(g(n))$ *if there is some constant* $A > 0$ *such that for all sufficiently large n,*

$$f(n) \leqslant A \times g(n).$$

Prove, by induction, that $2n+1 = O(2^n)$. (5)

 TURN OVER

3. Sorting.

Code for merging two sorted arrays (as used in MergeSort) is:

```
1  public static int[] merge(int[] a, int[] b) {
2
3    int[] m = new int[a.length + b.length];
4    int ai = 0, bi = 0, mi = 0;
5
6    while (ai < a.length && bi < b.length) {
7      if (a[ai] <= b[bi]) {
8        m[mi++] = a[ai++];
9      } else {
10       m[mi++] = b[bi++];
11     }
12   }
13
14   while (ai < a.length) {
15     m[mi++] = a[ai++];
16   }
17   while (bi < b.length) {
18     m[mi++] = b[bi++];
19   }
20
21   return m;
22 }
```

What is the time complexity for running `merge`? Justify your answer by detailing the time cost for each line of code. (5)

4. Heaps.

Suppose that we have an array-based implementation `Heap.java` that represents a max-heap of `Integer` elements. That is, it contains a data-field `heap` of type `Integer` and the following conditions hold:

- The element at index 0 is the biggest number in the heap.
- The children (if any) of the element at index `i` are at indices `2*i+1` and `2*i+2` and are less than or equal to the element at index `i`.

We consider positions 0 through $n-1$ of the array to be *occupied* if the heap contains n elements (the remaining positions, if any, are free space).

The `add` operation places a new element at the first unoccupied index and then adjusts its position relative to its parent (and grandparent, etc.) until the conditions are restored.

The `remove` operation removes (and eventually returns) the element at position 0, by replacing it with the element at the last occupied index and then adjusting its position relative to its children (and grandchildren, etc.) until the conditions are restored.

(a) List (in increasing order of their index) the occupied positions of the array `heap` after the following sequence of operations. (3)

```
Heap h = new Heap();
h.add(8);
h.add(2);
h.add(10);
h.add(12);
h.add(3);
```

(b) What is printed by the following? (1)

```
Heap h = new Heap();
int [] as = new int[] {1, 2, 12, 1, 0, 3};
for(int a : as) h.add(a);
System.out.println(h.remove());
```

(c) List (in increasing order of their index) the occupied positions of the array `heap` after the following sequence of operations. (3)

```
Heap h = new Heap();
int[] as = new int[] {25, 2, 1, 17, 5, 9};
for(int a : as) h.add(a);
h.remove(); h.remove();
```

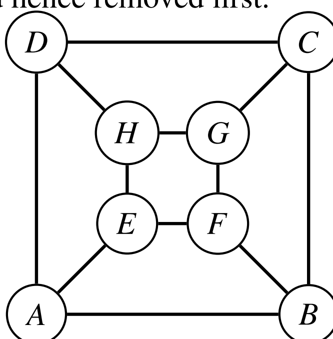
(d) How would you implement a *priority queue* data structure using a heap? In a priority queue, each element consists of a *value* and an associated *priority* and we add pairs consisting of a value and a priority, but when we remove an element all we want back is the value of the element of greatest priority. (3)

5. Graphs

(a) Pseudocode for breadth-first and depth-first traversal from a given vertex v in a graph is given below. We will assume that the vertices have distinct `String` labels and that the list of neighbours of a vertex is returned in alphabetical order.

Breadth-first	**Depth-first**
$q.add(v)$, $v.seen \leftarrow$ **true**	$s.push(v)$, $v.seen \leftarrow$ **true**
while q is not empty **do**	**while** s is not empty **do**
$w \leftarrow q.remove()$	$w \leftarrow s.pop()$
for n in $w.neighours$ **do**	**for** n in $w.neighours$ **do**
if $n.seen =$ **false then**	**if** $n.seen =$ **false then**
$q.add(n)$	$s.push(n)$
$n.seen \leftarrow$ **true**	$n.seen \leftarrow$ **true**
end if	**end if**
end for	**end for**
end while	**end while**

Note that the only difference is the use of a queue (q) in the breadth-first case and a stack (s) in the depth-first case. For clarity about the order: in a three vertex graph with vertices A, B and C and edges AB and AC only, the breadth-first traversal from A in visiting order is ABC while the depth-first traversal is ACB because C is added to the stack after B and hence removed first.



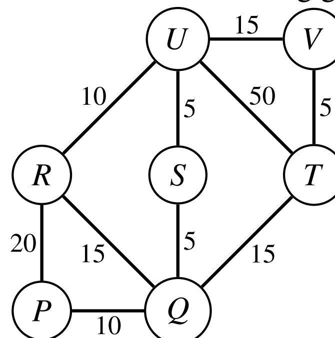
(i) List the breadth-first traversal of the graph above, from vertex C. (2)

(ii) List the depth-first traversal of the graph above, from vertex E. (2)

(b) Dijkstra's algorithm generates a shortest-path tree given a starting node. An implementation of Dijsktra's algorithm is as follows:

```
public void dijkstra(String start) {
        PQueue<Vertex> pq = new PQueue();
        for (Vertex v : graph.getVertices().values()) {
             distance.put(v, Double.MAX_VALUE);
             predecessor.put(v, null);
        }
        Vertex startVertex = graph.getVertices().get(start);
        distance.put(startVertex, 0.0);
        pq.add(startVertex, 0.0);
        while (!pq.isEmpty()) {
             Vertex v = pq.remove();
             for (Edge e : graph.getEdges(v) ) {
                  double alt = distance.get(v) + e.weight;
                  if (alt < distance.get(e.v2)) {
                       distance.put(e.v2, alt);
                       predecessor.put(e.v2, e);
                       pq.add(e.v2, alt);
                  }
             }
        }
   }
```

The following questions relate to the following graph:



(i) Give the shortest-path tree as computed by Dijkstra's algorithm for the above graph starting at node *S*. (3)

(ii) Prim's algorithm is identical to Dijsktra's algorithm except that the distance used is the distance of a single graph edge rather than an accumulated distance. Give the minimum spanning tree for the above graph starting at node *S*. (3)

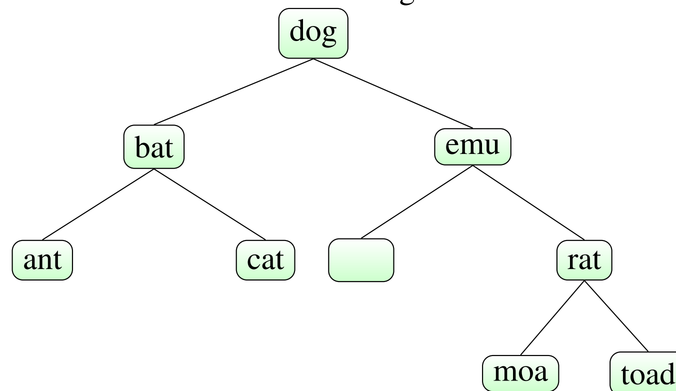
6. Binary Search Trees

In a *binary search tree* (BST) of strings (for the sake of example) each node contains a string and has at most two children (also nodes), called *left child* and *right child*. The strings stored in the left child or any of its descendants are less than the string stored at the node, and the strings stored in the right child or any of its descendants are greater than the string stored at the node. Binary search trees support `add`, `remove` and `find` as fundamental operations.

(a) The height of a binary tree is the maximum length of all the paths from the root to a leaf. A binary tree with a single node has height 0. Prove, by induction, that that a binary tree of height h, has at most $2^{h+1} - 1$ nodes. You may use the observation (without proof) that the maximum number of leaves in a tree of height h is 2^h. (5)

(b) In *self-balancing* trees, the longest path from the root to a leaf is less than or equal to twice the shortest path. Assume that a self-balanced tree has n nodes. What is the worst case cost for searching in such a tree in terms of n? You do not need to prove this answer, but you should justify it. (3)

(c) Give a preorder traversal of the following tree. (1)



 TURN OVER

7. Hashing

A collection of various `String` objects in Java are shown below along with their hash-codes:

"ant"	96743
"cat"	98263
"dog"	99644
"hog"	103487
"kea"	106055
"moa"	108287
"rat"	112677
"yak"	119395

Suppose that we constructed a hash set of these elements using an underlying array of 10 buckets, choosing the index by taking the remainder of the hashcode modulo 10, and added the elements in alphabetical order.

(a) Would any collisions occur when we add these elements to the array? If so, what is the first addition that causes a collision? (1)

(b) Show the contents of the underlying buckets if we use chaining to deal with collisions. (1)

(c) Show the contents of the underlying buckets if we use linear probing. (1)

(d) Suppose that we are using linear probing, that we remove "dog", replacing it by a tombstone, and then add "yyy" with hashcode 120153. At what index would it be stored? (1)

A set data type simply contains the items put into it. A map data type contains a mapping from a key to a value. A `HashSet` is a set implemented as a hash table, whereas a `TreeSet` is a set implemented as a binary search tree. `HashMap` and `TreeMap` are similarly defined, but for maps instead of sets. In each of the situations below, which of the Java library classes `HashSet`, `TreeSet`, `HashMap`, or `TreeMap` would be the most appropriate representation of the data? Include a brief justification (of a sentence or two in each case).

(a) The intended application is for a vocabulary learning flash card application that allows users to lookup the meaning of words as well as testing them on the meaning of words. (2)

(b) The intended application is for an inventory in an online shop. Shoppers want to search for items, check if they are in stock, and get a description and cost of the item. Shop owners want to print a list of all items in their shop in alphabetical order with the stock numbers. (2)

(c) The intended application is for access to a building using an ID card. The card reader will read the user's ID and grant access if it is on the list of allowable IDs. (2)

8. Greedy Algorithms and Dynamic Programming

(a) Huffman coding is an example of a greedy algorithm for computing prefix codes that can be used to do lossless compression. It works in a bottom-up greedy fashion. Assume we have the following letter frequencies:

	a	b	c	d	e	f
frequency	45	13	12	16	9	5

(i) Show the Huffman code tree with individual letters in the leaves of the tree, frequencies in the nodes, and 0 or 1 on the branches. (3)

(ii) Encode the following string "badcab". (2)

(b) Let $X = [x_1, x_2, \ldots, x_m]$ and $Y = [y_1, y_2, \ldots, y_n]$ be two strings. Further, let $Z = [z_1, z_2, \ldots, z_k]$ be a longest common subsequence (LCS) of X and Y. Then the following statements hold:

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- If $x_m \neq y_n$ then either:
  - Z is an LCS of X_{m-1} and Y; or
  - Z is an LCS of X_m and Y_{n-1}.

Note that the elements of a subsequence must be in order but need not be adjacent. So the LCS for "abc" and "adec" is "ac".

(i) Give a naive recursive algorithm for solving the LCS problem. (4)

(ii) Briefly explain how to make the naive recursive algorithm efficient. (2)

END

(COSC 201)

PLEASE DO NOT

TURN OVER YOUR

EXAMINATION

PAPER UNTIL

INSTRUCTED TO

DO SO

Please check your desk and pockets for unauthorised items including cell phones