

Midterm Exam

Name: _____

SS Number: _____

Instructions: Answer all questions. In the case of multiple choice questions, only the *best* answer will be accepted.

1. (24) Let $V = \{v_1, v_2, \dots, v_n\}$ be a set of nodes. Call an unordered pair $\langle v_i, v_j \rangle$, $v_i, v_j \in V$, an edge. A set N of edges is called a *network*. For every $v \in V$, the set $N_v = \{\langle v, w \rangle \in N\}$ is called the *neighborhood* of v . We say that a subset $P \subset N$ of edges is a *path* between two nodes v_i and v_j , $i \neq j$, if there is some permutation π of vertices such that

$$P = \{\langle v_i, v_{\pi_1} \rangle, \langle v_{\pi_1}, v_{\pi_2} \rangle, \langle v_{\pi_2}, v_{\pi_3} \rangle, \dots, \langle v_{\pi_{k-1}}, v_{\pi_k} \rangle, \langle v_{\pi_k}, v_j \rangle\}$$

for some positive k and $i \neq \pi_p$ and $j \neq \pi_p$ for any $1 \leq p \leq k$. The cardinality of P is the number of edges in P . The edge $\langle v_i, v_j \rangle$ is a path between v_i and v_j of cardinality 1. Write pseudocode for finding a minimum cardinality path between two given vertices v_i and v_j . Write the pseudocode using Mathematical structures such as *Set* and *List*. For sets, use \emptyset to denote an empty set, use the symbol \setminus to denote the operation of removing elements from a set and use the symbol \cup to denote the operation of adding elements to a set. For example:

```
Set S =  $\emptyset$ ;           // Define and initialize set S to be empty
Set A = {a,b,c,d,e};   // Define and initialize set A to {a,b,c,d,e}
S = A \ {a,b};        // Assign set {c,d,e} to S
S = A  $\cup$  {f,g};      // Assign set {a,b,c,d,e,f,g} to S
```

For lists use += and + to denote adding one or more elements to a list and -- to denote removing an element from a list. For example:

```
Vertex x = v1, Vertex y = v2; // Define and initialize Vertices
List L;                          // Define a list L, initially empty
L += x;                            // List L is [v1]
List Q = L + y                      // List Q is [v1, v2]
Vertex w = Q--;                     // Vertex w is v1 and Q is now [v2]
```

Explain which data structures would best be used to implement various portions of your pseudocode. Then explain the complexity of the operations in your pseudocode if implemented as you suggest.

```

// Assume origin is  $v_1$  and destination is  $v_n$ 
Set  $V = v_1, v_2, \dots, v_n$ ; // a set of cities
 $\forall v \in V$  let  $N_v$  be a List of neighbors of  $v$ ;
 $\forall v \in V$  let  $F_v = \emptyset$  // Empty from lists;
List  $q = \emptyset$  // a list of city objects
Set  $M = \emptyset$  // set of marked cities

 $M = M \cup \{v_1\}$ ;
 $q += v_1$ ;
while ( $q \neq \emptyset$ ) do the following:
    Vertex  $v = q--$ ;
     $\forall w \in N_v$  do the following:
        if ( $w == v_n$ ) then output  $F_v + v$ ;
        if ( $w \notin M$ ) then do the following:
             $M = M \cup \{w\}$ ;
             $F_w = F_v + v$ ;
             $q += w$ ;
output "no route";

```

Use an array of `bool` to implement M : indices will be vertex numbers so finding a vertex and marking it will take $O(1)$ time. Use a `Queue` object (implemented with linked lists) for q because the List q is used with additions made to the rear and removals from the front: hence all `Queue` operations are $O(1)$. Use `Queue` objects for all N_v because each neighbor is considered one time only (when it is unmarked - each is marked when it is considered the first time). All operations then are $O(1)$. Implement F_v as a link back to another Vertex object with $F_w = F_v + v$ meaning set the link of Vertex w to point to Vertex v . These operations are $O(1)$. Worst case complexity of the whole algorithm is then proportional to the number of edges in the network.

2. (20) A radix sort is to be performed on the following numbers: 734, 374, 437, 347, 473, 743, 437. Fill in the table below showing the placement of these numbers in *ordered* lists and when collected from the lists after each run through. The order shown in the lists is of *crucial* importance.

start:

734	374	437	347	473	743	437
-----	-----	-----	-----	-----	-----	-----

lists	473	743					
	734	374					
	437	347	437				

collect:

473	743	734	374	437	347	437
-----	-----	-----	-----	-----	-----	-----

lists	734	437	437				
	743	347					
	473	374					

collect:

734	437	437	743	347	473	374
-----	-----	-----	-----	-----	-----	-----

lists	347	374					
	437	437	473				
	734	743					

collect:

347	374	437	437	473	734	743
-----	-----	-----	-----	-----	-----	-----

lists							

collect:

--	--	--	--	--	--	--

lists							

collect:

--	--	--	--	--	--	--

3. (20) Let \odot denote a binary operation on general trees. The action of $t_1 \odot t_2$ is to make the root of tree t_1 a child of tree t_2 . Given a forest F of n trees, each tree containing just a root, the \odot operator is applied many times to trees of F until F consists of a single tree of n nodes. Note: think of the operation $t_1 \odot t_2$ as replacing both t_1 and t_2 in F with a tree that is the size of t_1 plus the size of t_2 . In other words, larger trees are made from smaller trees but all trees are always in F .
- What is the minimum number of applications of \odot and why?
 - What is the maximum height of the final tree and why?
 - What can be changed to considerably reduce the maximum height?
 - What is the maximum height of the final tree with the change and why?

(a) Each application of \odot reduces the number of sets in the partition by 1. Hence $n - 1$ is the minimum number of applications needed to reduce the partition from n trees to 1 tree.

(b) The final tree cannot be higher than n . Such a tree is merely a line of vertices all leading to the root and can be constructed from repeated applications of \odot by making sure one of the operands is a single vertex tree and then by making the root of the other tree a child of the root of the single vertex tree.

(c) Instead of the root of a higher tree being made a child of the root of a shorter tree, as in the previous item, make the root of the smaller tree a child of the root of the larger tree.

(d) We can show the maximum height of the final tree is $O(\log(n))$. In fact, we can show that any \odot operation on two trees of size s_1 and s_2 , with heights no greater than $\log_2(s_1) + 1$ and $\log_2(s_2) + 1$, respectively, will result in a tree of height no greater than $\log_2(s_1 + s_2) + 1$ if the root of the smaller is made a child of the root of the larger.

By induction on the height of trees. By convention, trees t_1 and t_2 of sizes s_1 and s_2 are \odot ed to get a tree of height h .

Base step: Tree height $h = 1$. Since size is always at least 1, $\log_2(s) + 1 \geq 1 = h$ and the hypothesis holds.

Induction step: Suppose the hypothesis is true for all trees of height less than h . Let t be a tree of height $h > 1$ with fewest vertices. Then t must have been obtained from $t_1 \odot t_2$ where t_1 , say, has height $h - 1$ and t_1 has no more vertices than t_2 . By the induction hypothesis, $\log_2(s_1) + 1 \geq h - 1$ so $s_1 \geq 2^{h-2}$. Since t_2 has at least as many vertices as t_1 , $s_2 \geq s_1 \geq 2^{h-2}$ as well. Hence $s_1 + s_2 \geq 2^{h-1}$. rewriting gives $\log_2(s_1 + s_2) \geq h - 1$ or $\log_2(s_1 + s_2) + 1 \geq h$ so the hypothesis holds on the induction step.

4. (6) What is $\log^*(\log_2(n)) + 1$? Treat n as a tower of 2's.

We will say $n = 2^{2^{2^{\dots}}}$. Then

$$\begin{aligned}\log^*(\log_2(n)) + 1 &= \log^*(\log_2(2^{2^{2^{\dots}}})) + 1 = \log^*(2^{2^{\dots}} \log_2(2)) + 1 \\ &= \log^*(2^{2^{\dots}}) + 1 = \log^*(2^{2^{2^{\dots}}}) = \log^*(n)\end{aligned}$$

5. (12) A set P of unique elements is partitioned into a collection of subsets so that every element is in a subset but no element is in more than one subset. Let a and b be elements and suppose $a \in A \subseteq P$ and $b \in B \subseteq P$. Define binary operator \oplus on two elements such that $a \oplus b$ replaces A and B with $A \cup B$. We wish to perform many \oplus operations. Identify a good data structure for this purpose and say why it is good. That is, show the worst case complexity of \oplus using your data structure.

Use inverted trees. Write a function f that maps an element to a number in such a way that every number from 1 to $|P|$ is output for some input element. This can generally be done so that $f(x)$ can be computed in $O(1)$ time, for any $x \in P$. Define a **Vertex** class with local state consisting of a pointer **parent** to an object of the **Vertex** class and a number called **length** which will be used to record the length of the longest path from any **Vertex** object to **this Vertex** object following **parent** pointers. Define an array of pointers to **Vertex** objects which will be used to locate **Vertex** objects of specific elements of P : the i^{th} element of the array will always point to a **Vertex** object representing element x such that $f(x) = i$. Initialize the **parent** field of all **Vertex** objects to **NULL** and the **length** field to 0. A **Vertex** object with a **NULL parent** field will be the representative (root of the inverted tree) for all **Vertex** objects that can reach it through **parent** links. Finding **Vertices** of two elements of P consists of applying f twice to compute the indices of elements in the array that contain pointers to the associated **Vertex** objects. This (1st phase of find) has $O(1)$ complexity from the above. The complexity of identifying sets containing those elements (2nd phase of find) is a function of the maximum length of the linked list through **parent** fields to the set representative. By the answer to question 3 this can be done in $O(\log(n))$ time if, when unioning, the root of the inverted tree with the shorter longest path is made the child of the other root. The unioning can be done in $O(1)$ time since it consists of setting a single pointer and replace a **length** with the sum of two **lengths**. If path compression is used, the find complexity becomes $\log^*(n)$ which we will not show.

6. (3) Identify five properties of a Red-Black tree
- (a) Each vertex is either red or black
 - (b) Every vertex has at most two children
 - (c) The number of black vertices on any path from root to leaf is the same
 - (d) There are never more than two reds in a row on any path from the root
 - (e) Numbers are associated with every vertex and the number of the left child of a vertex is less than the number of the vertex and the number of the right child of a vertex is greater than the number of the vertex.

7. (3) State why the height of a Red-Black tree must be $O(\log(n))$ where n is the number of nodes in the tree.

Recall all paths from root to leaf have the same number of black vertices. Since no two consecutive reds exist on a path from root to leaf, the vertices of a longest path would alternate in color between red and black. Such a path would have as many black vertices as red. Hence, the longest path (alternating colors) can be no longer than twice the shortest (only black). The shortest path is no greater than $\log_2(n)$ since that would require more than n vertices in the tree, a contradiction. Then the longest is no greater than $2 \log_2(n) = O(\log(n))$.

8. (6) We want to implement multitasking on a computer with a single processor. In other words, we want to create the illusion of processing several jobs simultaneously by letting the processor compute for a time (called a time-slice) on one job, then switch to another job and compute for another time-slice, and so on, until all jobs are finished. At the completion of a time-slice for any job J , an earliest resume-time for J is always (re)computed (and never decreases). At the beginning of a time-slice, the processor always chooses to process the job with lowest resume time. Let a *node* be a data element containing all information about a suspended process including information needed to resume that process. The data structure we should use to hold nodes is a
(a) Stack (b) Queue (c) Bipolar List (d) Binary Tree (e) Tree **(f) Heap**

9. (6) Consider the following code

```
Object *p = new Object(1);
Cell *h = new Cell(p, new Cell(p, new Cell(p, h)));
h->setObject(new Object(2));
h->getNext()->setObject(new Object(3));
```

Which of the following is true?

- (a) p points to NULL.
(b) p points to Object number 1.
(c) p points to Object number 2.
(d) p points to Object number 3.
(e) The above is illegal in C++.