|  |  |  |
| --- | --- | --- |
| **ECS 60** | **Written Homework #2: Lists, Stacks, Queues, Recursion (10 points)** | **Winter 2016** |

Due Wednesday, January 20th, 4:00 pm in 2131 Kemper

1. (2 points) Let *A* be an array of size *n ≥* 6 containing integers from 1 to *n* – 5, inclusive with exactly five repeated. Describe an O(*n*) algorithm for finding the five integers in *A* that are repeated. (Goodrich, p. 150)

**Create another char array, Count, size n-1, and set all elements to zero, O(n). Loop through A, and for each element in A increment the element in Count that has index of the value of the element of A, i.e. Count[A[i]]++, O(n). If an element in Count becomes two, then you have found the repeated element, all in O(n).**

1. (2 points) Given a list *L* of *n* arbitrary integers, describe an O(*n*) time function for finding an integer that cannot be formed as the sum of two integers in *L*. (Goodrich, p. 266)

**Pass through the list to find the largest integer in O(*n*) time, call it *max.* Then 2 \* *max* + 1 is guaranteed to satisfy the requirement.**

1. (2 points) Describe how to implement a stack ADT using two queues. What is the running time of the push(), and pop() function in this case? (Goodrich p. 224)

**There are two approaches to this. Both will have only one queue that will contain elements at the beginning of each operation. One approach will have the non-empty queue stored in FIFO order based on the order of inserts into the stack. For this approach, S.push(object) is simply Q.enqueue(object), for O(1). For this approach, S.pop() involves transferring all items from one queue to other except the last one, and returning the last one, O(n). The other approach will have the non-empty queue stored in LIFO order based on the order of inserts into the stack. For that approach, S.pop() is simply Q.dequeue(), for O(1). For that approach, S.push(object) involves enqueuing the object in the empty queue, and then transferring the other queue into it, O(n).**

1. (2 points) Propose a data structure that supports the stack push and pop operations, and a third operation findMin, which returns the smallest element in the data structure, all in O(1) worst-case time. (Weiss, p. 119)

**Let *E* be our extended stack. We will implement *E* with two stacks. One stack, which we’ll call *S* , is used to keep track of the *push* and *pop* operations, and the other, M, keeps track of the minimum. To implement *E.push(x),* we perform *S.push(x).* If *x* is smaller than or equal to the top element in stack M, then we also perform *M.push(x).* To implement *E.pop(),* we perform *S.pop().* If *x* is equal to the top element in stack M, then we also *M.pop(). E.findMin()* is performed by examining the top of M. All these operations are clearly O(1).**

1. (2 points) If the recursive routine on page 59 of the text used to compute Fibonacci numbers is run for *N* = 50, is stack space likely to run out? Why or why not? (Weiss p. 119)

**Stack space will not run out because at most 49 calls will be stacked at any given time. This is because when processing each call to the fib() function, it will wait for the call to fib(n – 1) returns before calling fib(n – 2). Thus, the most stack calls occur when each successive function is calling fib(n – 1). However the running time is exponential, as shown in Chapter 2, and thus the routine will not terminate in a reasonable amount of time.**

Sources of questions:

Michael T. Goodrich, Roberto Tamassia, and David Mount, *Data Structures & Algorithms*, *Second Edition*, Hoboken, NJ, John Wiley & Sons, 2011.

Mark Weiss, *Data Structures and Algorithm Analysis in C++, Fourth Edition,* New York, NY, Pearson Education, 2014.