İZMİR UNIVERSITY OF ECONOMICS

## Faculty of Engineering

# CE 221 - Data Structures and Algorithms

# Final Exam

## January 18th, 2022, Tuesday, 14:30

| | | | |
|---|---|---|---|
| **First Name:** | | **Last Name:** | |
| **Student Number:** | | **Signature:** | |
| **Lecturer's Name** | | | |

## Instructions (Read them carefully!)

- According to Higher Education Council's (YÖK) Student Discipline Regulation, the consequence of cheating or an attempt to cheat is 6 to 12 months expulsion. Having been done intentionally or accidentally would not change the punitive consequences of academic dishonesty. This is your responsibility.
- Show all your work for each question in the specified spaces on the exam sheets. Make sure your solutions are neat and clearly marked. You may use the blank areas on the exam pages for scratch work. Please **do not** use any additional scratch paper. **No** calculators, notes, books, cell phones, laptops, etc. are permitted to be used in this exam.
- *Simplicity and clarity of solutions do count.* You may get as few as **zero** points for a problem if your solution is far more complicated than necessary, or if we cannot understand your solution.

## **Duration**: 90 minutes

| **Problem** | **1** | **2** | **3** | **4** | **Total** |
|---|---|---|---|---|---|
| **Maximum Points** | 25 | 25 | 25 | 25 | 100 |
| **Your Score** | | | | | |

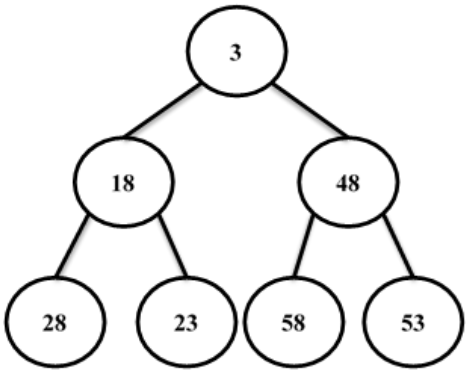
**Good Luck**

## Questions

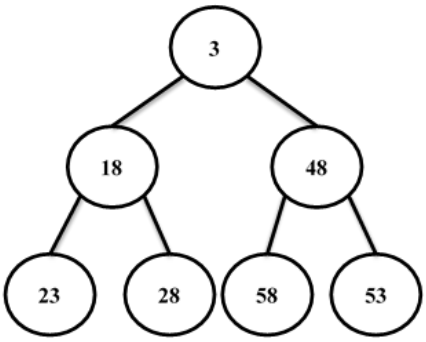
**Problem 1) A Binary Heap is a computer scientist's best friend! [25 pts.]**
You have the following sequence of items specified by their keys as:

**28, 23, 48, 3, 18, 58, 53**

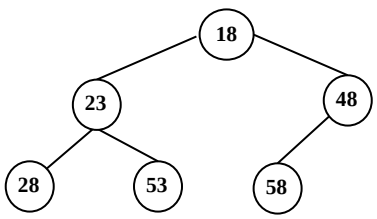
**1.1.** Show the result of constructing a binary heap (min-heap) by inserting the items one at a time in exactly the order given above, into an initially empty binary heap (i.e., insert 28, insert 23, insert 48, and so on). **(10 pts.)**
**Solution:**




**1.2.** Show the result of building a binary heap (min-heap) with the same input given above, this time by using the linear-time algorithm discussed in class to build a binary heap (*buildHeap*). **(10 pts.)**
**Solution:**




**1.3.** Show the result of performing a `deleteMin` operation in the resulting binary heap obtained in (1.1) above. **(5 pts.)**
**Solution:**




**after a `deleteMin` to 1.1**

**Problem 2) To hash or not to hash! [25 pts.]**
Consider inserting keys

**12, 8, 11, 9, 18**

into a hash table of size "7" with positions numbered 0 through 6.

Show the contents of the hash table after inserting all the keys in the specified order (i.e., first insert 12, then insert 8, insert 11 and so on) **if** you use linear probing with **hash(x) = x mod 7**,

**2.1.** without taking the load factor of the hash table into account. (Hint: Insert the keys by simply using the given hash function without rehashing.) **(15 pts.)**
**Solution:**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | *8* | *9* | | *11* | *12* | 18 |

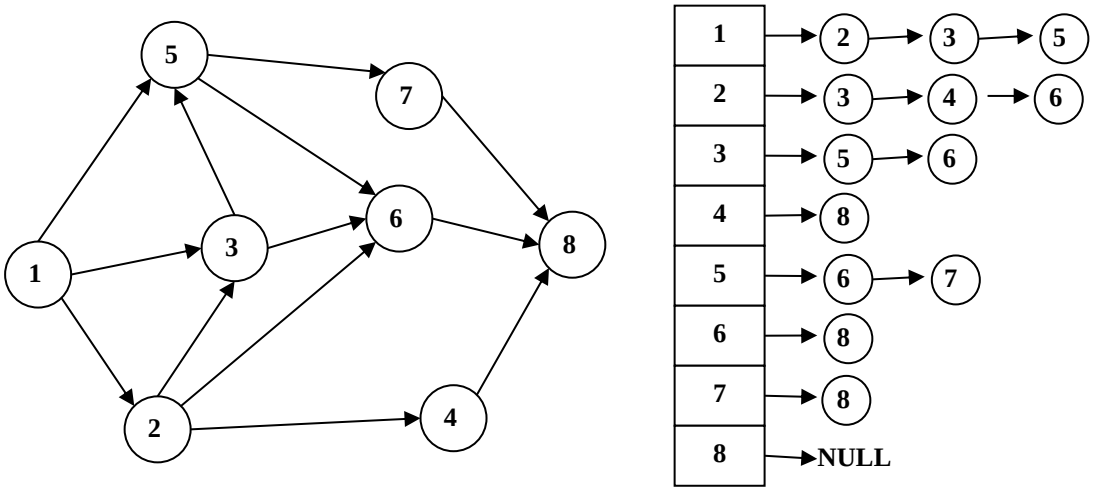
**2.2.** by taking the load factor of the hash table into account this time. You should also specify the new size of the hash table. (Hint: With linear probing, please recall that the load factor is monitored to be less than 1/2, and otherwise a rehashing to a new table is performed (in line with the source code discussed in class) to accordingly adjust the size of the new table along with an update to the hash function.) **(10 pts.)**
**Solution:**

Smallest prime > 2*old table size is **17**. When 9 gets inserted, the load factor becomes 4/7 > 1/2, and a rehash of 8, 9, 11, and 12 already in the old table is performed in this order into the new table using the new hash function. The insertion of 18 into the new table finally follows.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *18* | | | | | | | *8* | *9* | | *11* | *12* | | | | |

**Problem 3) Yet another one: Topological Sort! [25 pts.]**

In this question, you are asked to topologically sort the vertices in the directed acyclic graph (DAG) given on the left below. We also show the adjacency list representation of this DAG on the right below. You should note that the vertices adjacent to a particular vertex are stored in increasing order in this adjacency list. Please follow the instructions carefully for questions 3.1 and 3.2.




**3.1.** Find a topological ordering by using a queue data structure in the linear time (i.e. $O(|V|+|E|)$) algorithm discussed in class. Make sure to examine the vertices adjacent to a node in the order they appear on the corresponding singly linked lists. Note that since you need to use a queue data structure, you should process the vertices with in-degree zero in a FIFO (First-In First-Out) order. **(15 pts.)**

**Solution:**

We arrive at the following ordering by using a queue: **1, 2, 3, 4, 5, 6, 7, 8**

**3.2.** Find a topological ordering by using a stack data structure instead of a queue this time. Make sure to examine the vertices adjacent to a node in the order they appear on the corresponding singly linked lists. Note again that since you need to use a stack data structure, you should process the vertices with in-degree zero in a LIFO (Last-In First-Out) order. **(10 pts.)**

**Solution:**

The topological order produced when a stack is used is: **1, 2, 4, 3, 5, 7, 6, 8**

**Problem 4) Quicksort is not the quickest! [25 pts.]**

You have an array ***A*** with positions 0 through 12 holding the elements

10, 14, 7, 12, 6, 3, 1, 5, 7, 15, 18, 2, 20

You will call the method shown below by ***quicksort (A, 0, 12)*** examined in class. Show the contents of the array after exactly one partitioning, right before the recursive calls are made. *swapReferences* is used to swap the elements at the positions given by its last two parameters. Make sure that the partitioning performed follows precisely the instructions in the following code. Please assume that the CUTOFF is set to a value such as 6 to let at least a single partitioning to be performed.

```
private static <AnyType extends Comparable<? super AnyType>>
void quicksort( AnyType [ ] a, int left, int right )
{
    if( left + CUTOFF <= right )
    {
        AnyType pivot = median3( a, left, right );

            // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j )
                swapReferences( a, i, j );
            else
                break;
        }

        swapReferences( a, i, right - 1 );   // Restore pivot

        quicksort( a, left, i - 1 );    // Sort small elements
        quicksort( a, i + 1, right );   // Sort large elements
    }
    else  // Do an insertion sort on the subarray
        insertionSort( a, left, right );
}
private static <AnyType extends Comparable<? super AnyType>>
AnyType median3( AnyType [ ] a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, center );
    if( a[ right ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, right );
    if( a[ right ].compareTo( a[ center ] ) < 0 )
        swapReferences( a, center, right );

        // Place pivot at position right - 1
    swapReferences( a, center, right - 1 );
    return a[ right - 1 ];
}
```

**Solution:**

The initial ordering in the array is 10, 14, 7, 12, 6, 3, 1, 5, 7, 15, 18, 2, 20. We have 13 elements. The centermost position is (0+12) / 2 = 6. The median-of-three pivot is specified with respect to the elements at the first, last and the center positions where 10, 1, and 20 sit. Once sorted, these three are 1, 10, 20 with 10 in the middle and hence the pivot. First, an in place sort is performed and followed by taking the pivot out of the way via a swap with the element at the next-to-last position. We give the individual steps in the figure below:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| initially | 10 | 14 | 7 | 12 | 6 | 3 | 1 | 5 | 7 | 15 | 18 | 2 | 20 |
| | | | | | | | | | | | | | |
| in-place sort | 1 | 14 | 7 | 12 | 6 | 3 | 10 | 5 | 7 | 15 | 18 | 2 | 20 |
| | | | | | | | pivot | | | | | next-to-last | |
| get the pivot out of the way | 1 | 14 | 7 | 12 | 6 | 3 | 2 | 5 | 7 | 15 | 18 | 10 | 20 |
| start | | i | | | | | | | | | j | | |
| advance i and j | | i | | | | | | | j | | | | |
| swap | 1 | 7 | 7 | 12 | 6 | 3 | 2 | 5 | 14 | 15 | 18 | 10 | 20 |
| advance | | | | i | | | | j | | | | | |
| swap | 1 | 7 | 7 | 5 | 6 | 3 | 2 | 12 | 14 | 15 | 18 | 10 | 20 |
| i and j cross | | | | | | | j | i | | | | | |
| **swap i and right-1** | **1** | **7** | **7** | **5** | **6** | **3** | **2** | **10** | **14** | **15** | **18** | **12** | **20** |

**Good Luck**