CMPT 225 D100, Spring 2023

Final Exam - SOLUTIONS
April 18, 2023

Name_____

SFU ID: |__|__|__|__|__|__|__|__|__|

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. Duration of the exam is 180 minutes.
2. Write your full name and SFU ID **clearly**.
3. This is a closed book exam, no calculators, cell phones, or any other material.
4. The exam consists of four (4) problems, each worth 25 points
5. Write your answers in the provided space.
6. There is an extra page at the end of the exam. You may use it if needed.
7. You may write helper functions if needed
8. Explain all your answers.
9. Really, explain all your answers.

Good luck!

Problem 1 (Java syntax, Big-O notation) [25 points]

A. (5 pts) Consider the following function.

```
public static void do_something(int n) {
  ArrayList<String> ar = new ArrayList<String>();
  for (int i = 0; i < n; i++)
    ar.add(String.valueOf(i)); // adds the strings "0", "1", "2",...
  for (int i = 0; i < n; i++)
    ar.remove(0); // remove the element in index 0
}
```

What is the running time of this function? Use Θ notation to express the running time.

Answer: The first loop runs in $\Theta(n)$ time total. The only subtle point here is that the array inside ArrayList is being resized. However the resizing multiplies the length of the array by some constant factor (1.5 or 2), which still makes the total running time $\Theta(n)$.

In second loop, when the array is of size k, the running time of remove(0) is $\Theta(k)$. This means that the total running time is $\Theta(1+2+3+...(n-1)) = \Theta(n^2)$.

B. (5 pts) Prove that the running time of the function foo(n) is $O(n^{\log_3(5)\approx 1.465})$.

```
public static int foo(int n) {
    if (n>=4)
        return foo(n/2)+foo(n/3)+foo(n/4);
    return 1;
}
```

[Bonus: 5 pts] Prove that the running time is $O(n^c)$ for some $c<\log_3(5)$.

Answer: Denote the running time of foo(n) by T(n).
Then $T(n) = T(n/2) + T(n/3) + T(n/4) + O(1)$
[open the recursion of T(n/2)]
$= [T(n/4) + T(n/6) + T(n/8)] + T(n/3) + T(n/4) + O(1)$
$< 5*T(n/3) + O(1)$
Using Master Theorem we get $T(n)<O(n^{\log_3(5)})$

We can slightly improve it by opening the recursion for T(n/2) and T(n/3):
Then $T(n) = T(n/2) + T(n/3) + T(n/4) + O(1)$
$= [T(n/4) + T(n/6) + T(n/8)] + [T(n/6) + T(n/9) + T(n/12)] + T(n/4) + O(1)$
$< 7*T(n/4) + O(1)$
Using Master Theorem we get $T(n)<O(n^{\log_4(7)})$.

It turns out that $\log_4(7) < \log_3(5)$

C. (5 pts) Consider the class IntString.

```
public class IntString {
  private int n;
  private String s;

  @Override
  public int hashCode() {
    if (s.length()>0)
      return (n*s.hashCode()) % 1000;
    else
      return n % 1000;
  }

}
```

Write the method findHashCollison() that gets an instance of the class, and returns another instance that has the same hashCode.
You may assume the class has the standard constructors/getters/setters.

```
public static IntString findHashCollision(IntString is) {

// We could simply return new IntString(is.getN()+1000, is.getS());
// It will work for almost all inputs, except when is.getN() is very close
to Integer.MAX_VALUE.
// so we fix this small issue:

    if (is.getN()>0)
      return new IntString(is.getN()-1000, is.getS());
    else
      return new IntString(is.getN()+1000, is.getS());

}
```

D. (10 pts) List 5 public methods of the class java.util.LinkedList? Briefly explain their running time.

Answer: Linkedist implements doubly linked list inside. Therefore, adding to head and tail and removing to head and tail is done in O(1) time.

- size() - O(1) because LinkedList remembers its size in a variable of foo(n) by T(n).
- addFirst() - O(1) time
- removeFirst() - O(1) time
- addLast() - O(1) time
- removeLast() - O(1) time
- get(int i) - O(n) time in worst case, because we need to traverse the list

Problem 2 (Binary Trees) [25 points]

In this problem use the following definition of Binary Tree. You may assume the classes have the standard getters and setters.

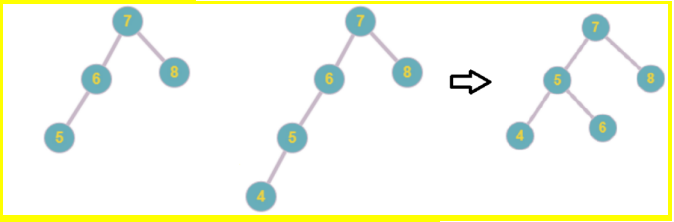
```
public class BTNode {
    private int data;
    private BTNode leftChild;
    private BTNode rightChild;
    private BTNode parent;
}

public class BinaryTree {
    private BTNode root;
}
```

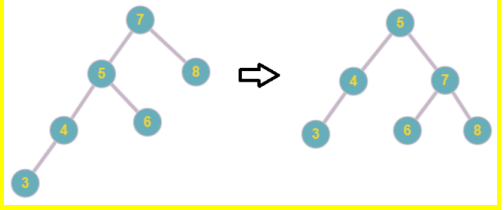
A. (10 pts) Draw the AVL tree obtained by inserting the following sequence of numbers: 8,7,6,5,4,3,2,1,0. Draw some intermediate trees to show your work.

Answer: add(8), add(7). add(6) needs a rotation:

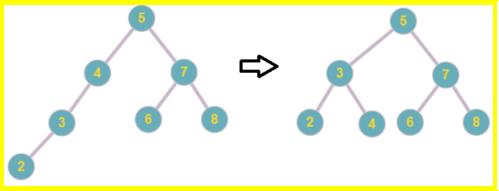
add(5). add(4) needs a rotation:



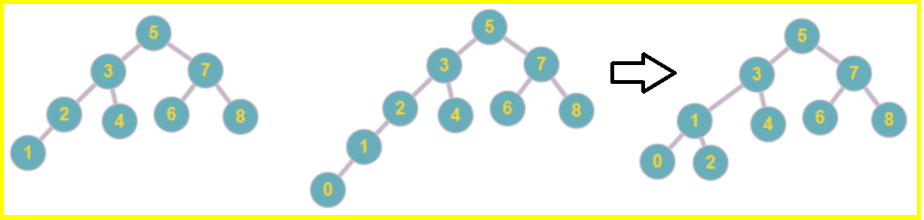
add(3) requires a rotation:



add(2) requires a rotation:



add(1), add(0):



B. (15 pts) Write a method that gets a Binary Tree and checks if it is a Binary Search Tree, i.e., every node is >= vertices in the left subtree, and <= vertices in the right subtree. The running time must be O(size of tree).

Explain the idea of the algorithm and the running time.

Answer: There are two ways to do it

1. Compute the inorder traversal of the tree and check that it is sorted.
2. Recursively for the left subtree and the right subtree do
   a. Compute min value, max value
   b. Check if the subtree is a BST
3. Compare root value to max of left subtree and min of right subtree

Below we'll do the second way, because the first one is pretty straightforward.

- We'll need a helper class to hold for each subtree: min, max and boolean isBST.
- And we'll need a helper function to obtain this info for the children of the root.

```
public static boolean isBST(BinaryTree tree) {
  if (tree.getRoot() == null)
    return true;
  return isBST(tree.getRoot()).getIsBST(); // call helper function
}

static class MinMaxBool {
  int min;
  int max;
  boolean isBST;
}

private static MinMaxBool isBST(BTNode root) {
      if (root == null) // stopping condition
            return null;
      int min = root.data; // returned min
      int max = root.data; // returned max
      boolean isBST = true; // returned boolean
      if (root.getLeftChild() != null) {
            MinMaxBool left = isBST(root.getLeftChild()); // recursion
            isBST = isBST && left.getIsBST()
                              && root.getData() >= left.getMax();
            min = Math.min(min, left.getMin());
            max = Math.max(max, left.getMax());
      }
      if (root.getRightChild() != null) {
            MinMaxBool right = isBST(root.getRightChild()); // recursion
            isBST = isBST && right.getIsBST()
                              && root.getData() <= right.getMin();
            min = Math.min(min, right.getMin());
            max = Math.max(max, right.getMax());
      }
      return new MinMaxBool(min, max, isBST);
}
```

Problem 3 (Quick sort and Selection algorithm) [25 points]

A. (12 pts) Write a function that gets an array of ints and a number k, and rearranges the array so that all numbers <=k appear in the array before all numbers >k.
The function needs to run in time O(ar.length), and use only O(1) additional memory.
Explain the idea of the algorithm.

Answer: we declare two pointers: leftPtr=0 and rightPtr=ar.length-1.
leftPtr moves to the right and only wants to see numbers <=k.
rightPtr moves to the left only wants to see numbers >k.
If ar[leftPtr]>k and ar[rightPtr]<=k, they swap their values.

```
public static void rearrange(int[] ar, int k) {
        int leftPtr = 0;
        int rightPtr = ar.length - 1;
        while (leftPtr < rightPtr) {
            while (ar[leftPtr] <= k)
                leftPtr++;
            while (ar[rightPtr] > k)
                rightPtr--;
            if (leftPtr<rightPtr)
                swap(ar, leftPtr, rightPtr);
        }
    }
```

The function swap is defined as follows:

```
    private static void swap(int[] ar, int i, int j) {
        int tmp = ar[i];
        ar[i] = ar[j];
        ar[j] = tmp;
    }
```

B. (6 pts) Show the execution of the Selection Algorithm that searches for the median of the array A = [9,16,5,10,6,12,0,6,5,4,6,9,8,1,2]. State explicitly all recursive calls.

Answer: the size of A is n=15, and we are looking for the k'th smallest number for k=8.
Partition A into [9,16,5,10,6] [12,0,6,5,4] [6,9,8,1,2].
The medians are 9,5,6. The median of these three numbers is M=6.
Next we rearrange A so that all numbers<=M are on the left, and all numbers>M are on the right.
[9,16,5,10,6,12,0,6,5,4,6,9,8,1,2] ⇒ swap(9,2)
[2,16,5,10,6,12,0,6,5,4,6,9,8,1,9] ⇒ swap(16,1)
[2,1,5,10,6,12,0,6,5,4,6,9,8,16,9] ⇒ swap(10,6)
[2,1,5,6,6,12,0,6,5,4,10,9,8,16,9] ⇒ swap(12,4)
[2,1,5,6,6,4,0,6,5,12,10,9,8,16,9] … done
There are 9 numbers <=M.

Therefore we recursively search for the k'th smallest number in [2,1,5,6,6,4,0,6,5].
Again, partition into 5-tuples: [2,1,5,6,6][4,0,6,5].
The first median in 5 the second median is 5. We set M=5, and rearrange:
[2,1,5,6,6,4,0,6,5]⇒ swap(6,5)
[2,1,5,5,6,4,0,6,6]⇒ swap(6,0)
[2,1,5,5,0,4,6,6,6] … done
6 numbers are smaller than M and 3 numbers are greater than M.
Therefore we apply recursion on the large numbers [6,6,6] searching for the (k-6)th smallest.
The answer is 6.

C. (7 pts) Consider the variant of Selection Algorithm, where we partition the input array into n/9 tuples each of size 9 (instead of n/5 tuples of size 5 we saw in class).
What will be the running time of the algorithm? Write the recursive formula for the running time, and then compute the closed formula.

Answer: Given an input of length n, we have 2 recursive calls:

- We recursively find the median of the n/9 medians of the 9-tuples and find M.
- Then, we get rid of at least (5/18)n elements, because there are at least n/18 tuples with medians <=M, and at least 5 in each is <=M. And same for numbers >=M.
- Hence we make a recursive call on (13/18)n elements.

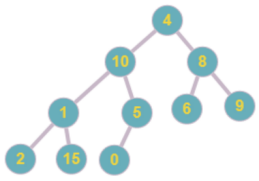
Therefore if we denote the running time by T(n), then $T(n) = T(n/9)+T(13n/18) + O(n)$.
Writing out the recursion tree to compute T(n), we will get
$T(n) = Cn + Cn*(1/9+13/18) + Cn*(1/9+13/18)^2 + Cn*(1/9+13/18)^3 + \ldots$
Since $1/9+13/18 = 15/18<1$, the geometric series $1+15/18+(15/18)^2+(15/18)^3+\ldots$ converges.
Therefore, T(n) = O(n)

Problem 4 (MinHeap) [25 points]

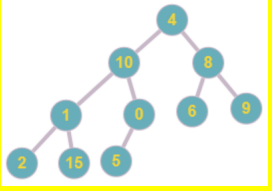
A. (7 pts) Apply buildMinHeap on the array A=[4,10,8,1,5,6,9,2,15,0]. Show some intermediate steps by drawing the corresponding tree and the array.
Draw the final result as a tree and as an array.

Answer: We start with

A=[4,10,8,1,5,6,9,2,15,0]:

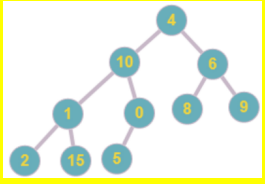


Heapify(5) swaps 5 and 0

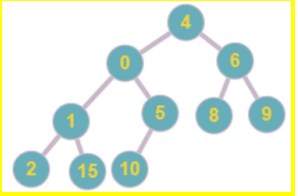


Heapify(1) has no effect.

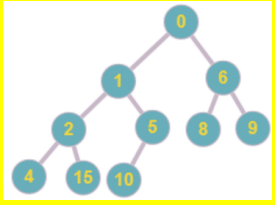
Heapify(8) swaps 8 and 6:



Heapify(10) swaps 10 with 0 and then 10 with 5:

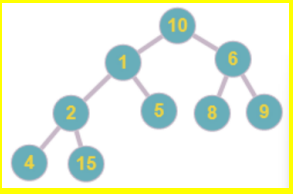


Finally we apply heapify(4)



B. (3 pts) Apply removeMin() from the heap obtained in part A. Draw the result as a tree and as an array.

Answer: move 10 to the root and then apply heapify()



C. (15 pts) Write a function that gets a MinHeap of n ints given as an array, and a parameter k (0<k<=n), and returns the k smallest numbers in the heap.
The heap itself remains unchanged.

- For 5 points write a function whose running time is $O(2^k)$
- For 10 points write a function whose running time is $O(k^2)$
- For 15 points write a function whose running time is O(k log(k))

Before writing code, explain the idea of the algorithm. You may assume you have the standard functions of the class MinHeap: getMin(), removeMin(), getLeftChild(), getRightChild(), getParent(), buildHeap(), heapify()...

Answer: A trivial solution: copy the array, sort, and return the first k numbers. Or copy the heap, and apply removeMin() k times. But these are too expensive.

Since the running time cannot depend on n, we are not allowed to perform any of the standard operations on the given heap. Instead, the idea is the following:

- The minimum of the heap the root, which is in heap[0]
- The second minimum is one of the two children of the root: either heap[1] or heap[2]
- Let's say for concreteness that the second minimum is heap[2].
- Then, the next smallest is either heap[1] or one of the children of heap[2]: heap[5,6]

As a first attempt we can read all vertices in the first k layers (there are at most 2^k of them), and find the k smallest numbers. Therefore, the total time will be $O(k*2^k)$ time.
For O(k log(k)) we do as follows: build a solution of smallest numbers so far, and store their children in the given heap in a priority queue PQ. In each iteration we move the min from PQ to our solution, and add its children from heap to the PQ.

```
public static Collection<Integer> getMinK(int[] heap, int k) {
  ArrayList<Integer> ret = new ArrayList<Integer>();
  PriorityQueue<IndexValue> pq = new PriorityQueue<IndexValue>();
  // pq holds children of ret
  pq.add(new IndexValue(0, heap[0])); // add the root of the heap to h
  while (ret.size() < k) {
    IndexValue iv = pq.poll(); // get the next minimum
    int i = iv.ind;
    ret.add(heap[i]); // add the next minimum from the solution
    if (2*i+1 < heap.length) // add to pq the left child of heap[i]
      pq.add(new IndexValue(2*i+1, heap[2*i+1]));
    if (2*i+2 < heap.length) // add to pq the right child of heap[i]
      pq.add(new IndexValue(2*i+2, heap[2*i+2]));
  }
  return ret;
}
// IndexValue is a helper class that stores pairs (i,heap[i])
public static class IndexValue {
  int ind;
  int val;
  int compareTo(Object other) { // compare objects by value
    return this.val - ((IndexValue) other).val;
  }
}
```

Extra page

Master Theorem

Let $T(n) = a\,T(n/b) + f(n)$, $T(1) = O(1)$

Define $c = \log_b(a)$

- If $f(n) = O(n^d)$ for $d<c$, then $T(n) = \Theta(n^c)$
- If $f(n) = \Omega(n^d)$ for $d>c$, then $T(n) = \Theta(n^d)$
- If $f(n) = \Theta(n^c)$, then $T(n) = \Theta(n^c \log(n))$