Western University
Department of Computer Science
Computer Science 1027b Final Exam
3 hours

PRINT YOUR NAME:

PRINT YOUR STUDENT NUMBER:

DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO!

Instructions

- Fill in your name and student number above immediately.
- You have **3 hours** to complete the exam.
- For multiple choice questions, circle your answers on this exam paper.
- For other questions, write your answers in the spaces provided in this exam paper.
- The marks for each individual question are given.
- Several interfaces are at the back of the exam.
- There are also pages for rough work at the back of the exam. You may detach them if you wish, but hand them in with the rest of the exam paper.
- Calculators are not allowed!

Mark summary

1	2	3	4	5	6	7	8	9	10	bonus	total
/20	/15	/15	/15	/15	/20	/12	/10	/10	/10	/10	/142

Problem 1: true/false (20 marks)

Choose **one** answer for each question.

1. A Java class always contains a constructor.

   true

2. All Java classes derive from `Object`

   true

3. An algorithm with running time $O(n)$ is faster than an algorithm with running time $O(n^2)$ for *any* value of n

   false $n=1$. equals.

4. For any interface you write, there must be at least one class that implements it

   false

5. Insertion in an `ArrayStack` with n elements is $O(n)$ in the worst case ✓

   true

6. Insertion in a `LinkedStack` with n elements is $O(n)$ in the worst case

   false

7. a `RuntimeException`, for instance an `ArrayIndexOfOutBoundsException`, must always be caught with a `try-catch`

   false

8. During a Java program execution, objects are created on the heap.

   true

9. During a Java program execution, call frames are stored in the program queue.

   false

10. Level-order traversal of a tree can be implemented using a queue.

    true

11. Suppose we use a heap to store objects of a class `Student` that represents Western students. It is possible to store all Western students in a heap of height 30 or less.

    true

12. In the worst case, `Quicksort` uses $O(n\log(n))$ operations to sort n entries

    false

13. The best case for `Quicksort` is when the pivot is the smallest element in the collection we want to sort

    false

14. There exists an algorithm that sorts n entries using $O(n \log(n))$ operations, even in the worst case
    true

15. In a binary search tree, the root always contains the smallest entry
    false

16. A binary search tree with n elements has height $O(\log(n))$
    false

17. Linear search in a sorted array of size n takes time $O(n)$ in the worst case
    true

18. Binary search in a sorted array of size n takes time $O(n)$ in the worst case
    false

19. Insertion in a heap with n elements can be done in time $O(\log(n))$
    true

20. Removing any item in a heap with n elements can be done in time $O(\log(n))$
    false

Problem 2 (15 marks)

In each of the following situations, use big-O notation to express the amount of work being done in terms of n.

1. (2 marks) An element is inserted in a `LinkedQueue` of size n

   Answer: $O(1)$

2. (2 marks) An element is removed from a `LinkedQueue` of size n

   Answer: $O(1)$

3. (2 marks) An element is inserted in a `BinarySearchTree` of size n (worst case)

   Answer: $O(n)$

4. (2 marks) We execute the following code segment

   ```
   for (int i = 1; i < n; i+=2)
     for (int j = 1; j < n; j+=3)
       System.out.println(i+j);
   ```

   Answer: $O(n^2)$

5. (2 marks) We execute the following code segment

   ```
   for (int i = 1; i < n; i*=2)
     for (int j = 1; j < n; j++)
       System.out.println(i+j);
   ```

   Answer: $O(n \log(n))$

6. (2 marks) We call `func(n)`, with `func` given here:

   ```
   static int func(int n){
     if (n <= 0)
       return 1;
     else
       return n*n*func(n-1);
   }
   ```

   Answer: $O(n)$

7. (3 marks) We call the `inorder()` method to construct an iterator from a binary search tree with n elements.

   Answer: $O(n)$

Problem 3 (15 marks)

In all following cases, write a code segment whose running time is as required

1. (2 marks) $O(1)$

   Answer: `System.out.println(1);`

2. (2 marks) $O(n)$, using a loop

   Answer:

```
for (int i = 0; i < n; i++)
  System.out.println(i);
```

3. (2 marks) $O(n)$, using a recursive method and no loop

   Answer:

```
static void func(int n){
  if (n <= 0)
    System.out.println(0);
  else{
    System.out.println(n);
    func(n-1);
  }
}
```

4. (3 marks) $O(n^2)$, using one or more loop(s)

   Answer:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    System.out.println(i+j);
```

5. (3 marks) $O(n^2)$, using one or more recursive method(s) and no loop. Hint: reuse the answer of question 3.

   Answer:

```
static void func2(int n){
  if (n <= 0)
    System.out.println(0);
  else{
    func(n)
    func2(n-1);
  }
}
```

6. (3 marks) $O(2^n)$ or more (but no infinite loop!).

   Answer:

```
int p = 1;
for (int i = 0; i < n; i++)
  p = 2*p;
for (int i = 0; i < p; i++)
   System.out.println(i);
```

   or

```
static void func3(int n){
  if (n <= 0)
    System.out.println(0);
  else{
    System.out.println(n);
    func3(n-1);
    func3(n-1);
  }
}
```

Problem 4 (15 marks)

Without using the variable `count`, rewrite the method `size()` for class `LinkedList` (10 marks), and give its complexity using the big-O notation (5 marks). Remember that a `LinkedList<T>` has two attributes of type `LinearNode<T>` called `head` and `tail`. For linear nodes, you may want to use `getNext()`. Do not destroy the list!

```
public int size(){
  int sz = 0;
  LinearNode<T> tmp = head;
  while (tmp != null){
    sz++;
    tmp = tmp.getNext();
  }
  return sz;
}
```

Problem 5 (15 marks)

1. (5 marks) Write a method `void transfer(LinkedStack<T> src, LinkedStack<T> dest)` that implements the following algorithm: while `src` is not empty, pop the element at the top of stack `src` and push it on stack `dest`. You may assume that neither `src` nor `dest` is the `null` reference.

```
void transfer(LinkedStack<T> src, LinkedStack<T> dest){
  while (!src.isEmpty())
    dest.push(src.pop());
}
```

2. Consider the following class (which uses the transfer method you wrote)

```
public class Mystery<T>{

  private LinkedStack<T> stack1;
  private LinkedStack<T> stack2;

  public Mystery(){
    stack1 = new LinkedStack<T>();
    stack2 = new LinkedStack<T>();
  }

  private void transfer(LinkedStack<T> src, LinkedStack<T> dest){
    ...
  }

  public void in(T element){
    stack1.push(element);
  }

  public T out() {
    if (stack2.isEmpty())
      transfer(stack1, stack2);
    return stack2.pop();
  }
}
```

Suppose now that we run the following test:

```
public class Test{
  public static void main(String[] args){
    Mystery<Integer> m = new Mystery<Integer>();
    m.in(new Integer(1));
    m.in(new Integer(2));
    System.out.println(m.out());
    m.in(new Integer(3));
    System.out.println(m.out());
    System.out.println(m.out());
  }
}
```

(5 marks) What do you see?
Answer:

```
1
2
3
```

(5 marks) What abstract data type is this mystery class implementing? (we do not ask you to prove it)
Answer: A queue

Problem 6 (20 marks)

Suppose you have two unordered lists of `String`. Write a `boolean` method that returns `true` if the two lists have the same contents but perhaps different orders. Both lists have iterators (we show you how to construct them); we give the `Iterator` interface at the end of this exam paper.

```
public static boolean compareLists(LinkedList<String> list1, LinkedList<String> list2) {
   Iterator<String> it1 = list1.iterator();
   Iterator<String> it2 = list2.iterator();
   // your code here

  boolean flag=false;

  if(list1.size() != list2.size()){
    System.out.println("Lists not the same sizes\n");
    return(false);
  }

  while(it1.hasNext()){
    String value1 = it1.next();
    while(it2.hasNext() && !flag) {
      String value2 = it2.next();
      if(value1.equals(value2))
        flag = true;
    }
    if(flag == false)
      return false;
    // Reset list2 to check against the next element of list 1
    else
      it2 = list2.iterator();
  }

  it1 = list1.iterator();
  while(it2.hasNext()) {
    String value2 = it2.next();
    while(it1.hasNext() && !flag){
      String value1 = it1.next();
      if(value1.equals(value2))
        flag = true;
  }
  if(flag == false)
     return false;
  // Reset list1 to check against the next element of list 2
  else
    it1 = list1.iterator();
  }
  return true;
}
```

Problem 7 (20 marks)

Consider searching for the largest integer in both a general binary tree and in a binary search tree.

(7a) (8 marks) Write a method to find the largest integer in a general binary tree of integers, given by a `BinaryTreeNode`. Remember that a `BinaryTreeNode` has the following fields: an `element` (an `Integer` in our case), and left and right child references, `left` and `right`, that should be accessed through getters `getElement()`, `getLeft()` and `getRight()`. To compare `Integer`'s, you can just use $<$, $>$, … or `Math.min` and `Math.max`. If the node is null, throw an exception.

```
static Integer findMaxInBinaryTree(BinaryTreeNode<Integer> node) {
  if(node == null)
    throw new EmptyCollectionException("binary tree");
  Integer val = node.getElement();
  if (node.getLeft() != null) {
    Integer valLeft = findMaxInBinaryTree(node.getLeft());
    if (valLeft > val)
      val = valLeft;
  }
  if (node.getRight() != null) {
    Integer valRight = findMaxInBinaryTree(node.getRight());
    if (valRight > val)
      val = valRight;
  }
  return val;
}
```

(7b) (2 marks) The worst case theoretical efficiency for this method is (circle one):

$O(n)$

(7c) (8 marks) Write a method to find the largest integer in a binary search tree of integers. The tree nodes are instances of `BinaryTreeNode<Integer>` as in (7a).

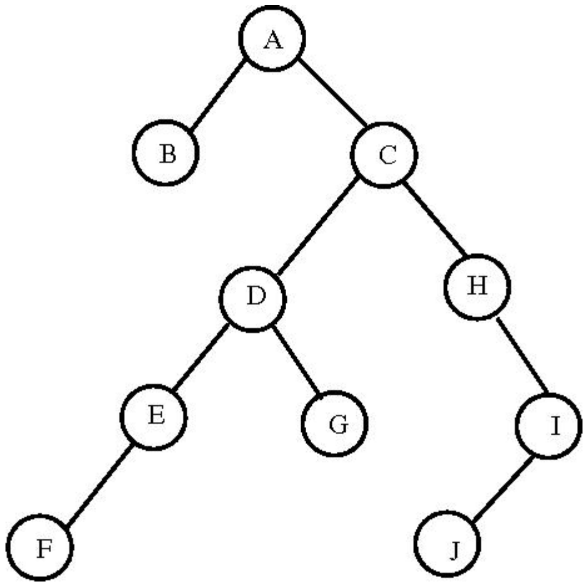
```
  static Integer findMaxInBinarySearchTree(BinaryTreeNode<Integer> node) {

    if(node == null)
      throw new EmptyCollectionException("binary tree");
    if (node.getRight() == null)
      return node.getElement();
    else
      return findMaxInBinarySearchTree(node.getRight());
}
```

(7b) (2 marks) For a tree with n elements, the worst case theoretical efficiency for this method is (circle one):

$O(n)$

Problem 8 (20 marks)

Consider the following binary tree:



(8a) (3%) Give the **inorder** traversal of this tree:

```
B A F E D G C H J I
```

(8b) (3%) Give the **preorder** traversal of this tree:

```
A B C D E F G H I J
```

(8c) (3%) Give the **postorder** traversal of this tree:

```
B F E G D J I H C A
```

(8d) (3%) Give the **level order** traversal of this tree:

```
A B C D H E G I F J
```

(8e) (8%) Consider the mystery traversal:

```
 A C H I
```

Write the recursive binary tree traversal method to compute this (the method should just print the node's contents; you do not have to put them in a list).

```
static void mysteryOrder(BinaryTreeNode<Character> node) {
  if(node == null)
    return;
  System.out.println(node.getElement();
  mysteryOrder(node.getRight());
}
```

Problem 9 (15 marks)

Write a recursive method that finds the number of single parents in a binary tree (a single parent has exactly one child; the other is null). The count for a tree with no single parents, or for the empty tree, is 0.

```
static int findNumberSingleParents(BinaryTreeNode<Integer> node) {
  if(node == null)
    return 0;
  if((node.getLeft() == null && node.getRight)() != null) ||
     (node.getLeft() != null && node.getRight() == null))
     return 1 + findNumberSingleParents(node.getLeft())
              + findNumberSingleParents(node.getRight());
  else
     return findNumberSingleParents(node.getLeft())
            + findNumberSingleParents(node.getRight());
}
```

Problem 10 (15 marks)

Write a recursive method that, given two binary tree nodes of `Integer`'s, returns `true` if they are structurally identical, that is, if they have the same parent/child structure. The values at the nodes can be different.

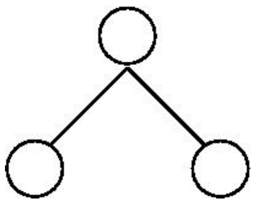
```
static boolean sameTreeStructure(BinaryTreeNode<Integer> a,
                                 BinaryTreeNode<Integer> b) {
  if(a == null && b == null)
    return true;
  if((a == null && b != null) || (a != null && b == null))
    return false;
  return(sameTreeStructure(a.getLeft(), b.getLeft()) &&
         sameTreeStructure(a.getRight(), b.getRight()));
}
```

Bonus problem (10 marks)

Consider the following methods, written in the class `LinkedBinaryTree`.

```
public double mystery(){
  if (isEmpty())
    throw new EmptyCollectionException("binary tree");
  return mysteryRec(root, 0) / (double)size();
}

public int mysteryRec(BinaryTreeNode<T> node, int i){
  if (node == null)
    return 0;
  return i + mysteryRec(node.getLeft(), i+1) + mysteryRec(node.getRight(), i+1);
}
```

(5 marks) Suppose you call `mystery` on the following tree (with strings stored at the nodes). What do you get? (the program returns a `double`, but you may give us a fraction instead)



Answer: 2/3

(5 marks) What is this mystery method computing?
Answer: the average height of the nodes in the tree.