The University of Western Ontario

Computer Science CS1027b

Final Examination - FEB GYM - 10:00am-1:00pm - Sunday, April 10th, 2016

Last Name:	
Given Names:	
Student Number:	

PLEASE CIRCLE ONE

Section I	Section II
John Barron	James Hughes
Tuesday 11:30am-1:30pm NCB 113	Tuesday 3:30pm-5:30pm NCB113
Thursday 11:30am-12:30pm NCB	Thursday 3:30pm-4:30pm NCB113

DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO!

Instructions

- Fill in your name and student number above immediately. Please use the last name and given names the university has for you (as on your student card).
- 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.
- Relevant Java 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, phones, laptops and any other electronic aids are not allowed!**

Mark Summary

1	2	3	4	5	6	7	8	9	10	total
/20	/12	/16	/12	/20	/20	/35	/10	/15	/15	/175

Problem 1: true/false (20 marks)

Choose **one** answer for each question.

1.	A binary search tree must always be implemented with a queue.	true	false
2.	A queue is an example of a LIFO structure.	true	false
3.	A binary tree is a LIFO structure.	true	false
4.	Exceptions cannot use inheritance.	true	false
5.	*Doubly linked list* is another word for a *binary tree*.	true	false
6.	`thing1.equals(thing2)` basically means the same thing as `thing1 == thing2`.	true	false
7.	There are three things required for recursion: a base case, the recursive call, and the induction call.	true	false
8.	Recursive algorithms are always better than iterative.	true	false
9.	Iterative algorithms are always better than recursive.	true	false
10.	In a binary tree, each node (with the exception of leaf nodes) must have exactly two children.	true	false
11.	`preOrder`, `inOrder`, and `postOrder` are examples of tree traversals that only work on binary *search* trees.	true	false
12.	The complexity of traversing over n nodes in a tree with a level order traversal is always $O(n^2)$.	true	false
13.	*Balancing* a binary search tree means to make sure the sum of all nodes on the left hand side is equal to the sum of all nodes on the right hand side.	true	false
14.	Given an element and an indexed list, the process of inserting that element is always $O(1)$ when using a *linked* data structure.	true	false
15.	`Selection sort` is a better algorithm than `insertion sort`.	true	false
16.	When inserting into an ordered list, you use `insertion sort`.	true	false
17.	`Quicksort` is better than `selection sort` because it uses a queue.	true	false
18.	By convention, the height of an empty tree is 0.	true	false
19.	The length of a path in a tree is the number of *nodes* in the path.	true	false
20.	Trees are linear data structures.	true	false

Problem 2 (12 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 added to an UnorderedArrayList of size n.
   Answer: $O(n)$
   expandCapacity have have to used to increase the array size

2. (2 marks) An inOrder traversal of a linked tree of of size n is performed.
   Answer: $O(n)$

3. (2 marks) We remove the smallest element from a binary tree of size n.
   Answer: $O(n)$

4. (3 marks) We execute the following code segment

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

   Answer: $O(n)$

5. (3 marks) We execute the following code segment

   ```
   static int func(int n){
     if (n <= 0)  return 1;
       else return (n+func(n/2));
   }
   ```

   Answer: $O(\log_2(n))$

Problem 3 (16 marks)

1. (2 marks) In what situation does *Bubble Sort* have it's best case performance of $O(n)$?

   The elements are already sorted

2. (2 marks) In what case(s) (Best, Worst, and Average) does *Selection Sort* perform $O(n^2)$

   All: Best, Worst, and Average.

3. (2 marks) What is the worst case time complexity of *Quicksort*?

   $O(n^2)$

4. (2 marks) What is the best case time complexity of *Quicksort*?

   $O(nlog(n))$

5. (2 marks) What is the best case performance of a search in a *binary search tree*?

   $O(1)$

6. (6 marks) Assuming we have the $O(n)$ function `isSorted(someArray)` which returns `true` if `someArray` is sorted, and `false` otherwise, what is the best and worse case performance of the following function written in pseudocode?

   ```
   someSort (int[] toSort){
     while (!isSorted(toSort))
       randomlyShuffle(toSort)
   }
   ```

   Best: $O(n)$, **Worst:** Who knows (it's unbounded)! (any answer suggesting they understand this will be fine)

Problem 4 (12 marks)

For this question you will write *two* methods.

First, `inOrder(BinaryTreeNode<T> node, ArrayUnorderedList<T> tempList)`, a recursive method which takes a reference to a binary tree and puts the *inorder* ordering of its elements into the unordered list (also passed as a parameter).

Second, write a method `iteratorInOrder()` which uses the `inOrder` method and ultimately returns an inorder iterator for the binary tree. Assume the `iterator()` method is defined in the `ArrayUnorderedList` class.

Note that all these methods are in a single class and they all can access the private variables of the class.

```
puublic void inorder (BinaryTreeNode<T> node, ArrayUnorderedList<T> tempList) {
  if (node != null){
    inorder (node.left, tempList);
    tempList.addToRear(node.element);
    inorder (node.right, tempList);
  }
}

public Iterator<T> iteratorInOrder(){
  ArrayUnorderedList<T> tempList = new ArrayUnorderedList<T>();
  inorder (root, tempList);
  return tempList.iterator();
}
```

Problem 5 (20 marks)

Assume that all methods of the LinkedList and LinkedOrderedList have been implemented correctly. Also, assume the `toString()` method will put the contents of the list onto a single line separated by a space. Assume the list is maintained in ascending order.

Consider the following small program:

```
1. public class Something{
2.
3.   public static void main(String[] args){
4.     LinkedOrderedList<Integer> myList = new LinkedOrderedList<Integer>();
5.     System.out.println(myList);
6.     System.out.println(mystery(myList.head));
7.     myList.add(5);
8.     myList.add(4);
9.     System.out.println(myList);
10.    System.out.println(mystery(myList.head));
11.    myList.add(3);
12.    myList.add(2);
13.    System.out.println(myList);
14.    System.out.println(mystery(myList.head));
15.  }
16.
17.  public static <T> String mystery(LinearNode<T> head){
18.    if(head == null)
19.      return "";
20.    else
21.      return head.getElement().toString() + " " + mystery(head.getNext());
22.  }
23. }
```

These questions are about the code found on the previous page.

1. (2 marks) What would be printed at line 9?

   `Answer: 4 5`

2. (2 marks) What would be printed at line 10?

   `Answer: 4 5`

3. (2 marks) What would be printed at line 13?

   `Answer: 2 3 4 5`

4. (2 marks) What would be printed at line 14?

   `Answer: 2 3 4 5`

5. (4 marks) What does the mystery method do?

   `Answer: It basically does toString(); it prints out the contents of the collection.`

6. (4 marks) If the size of the list is n, and we are only considering this instance (where the generic type is `Integer`), what is the time complexity of invoking the mystery method (in big O notation)?

   Answer: $O(n)$

7. (4 marks) If the size of the list is n, and the generic type is another collection of size roughly equal to n, what is the time complexity of invoking the mystery method (in big O notation)?

   Answer: $O(n^2)$

Problem 6 (20 marks)

(6a) (6 marks) Consider *heap* sorting in arrays. Write 3 Java methods to compute the parent and left and right child indices given index `index`. Parameter `n` is the number of nodes in the tree, `NOT_FOUND` is a constant set to -1 (an invalid index).

```
public static int getParentIndex(int index) {
  int parent = (index-1)/2);
  if(parent >= 0)
    return((int) (index-1)/2);
  else
    return(NOT_FOUND)
}

public static int getLeftChildIndex(int index,int n) {
  int childIndex=2*index+1;
  if(childIndex>=n) childIndex=NOT_FOUND;
  return(childIndex);
}

public static int getRightChildIndex(int index,int n) {
  int childIndex=2*index+2;
  if(childIndex>=n) childIndex=NOT_FOUND;
  return(NOT_FOUND);
}
```

(6b) (7 marks) Given these three methods, consider writing one step of the `bubbleDown` method for heaps. Given a node at `index`, give the Java code segment that potentially moves that node **down** 1 level in the heap. Remember `heap[index]` contains the heap integer value at that index. You can use a helper method `swapHeap(index1,index2)`, which swaps the heap elements indexed by `index1` and `index2` if you need to. `n` is the current number of nodes in the heap and the heap is organized so that to root of the tree is the maximum node in the tree.

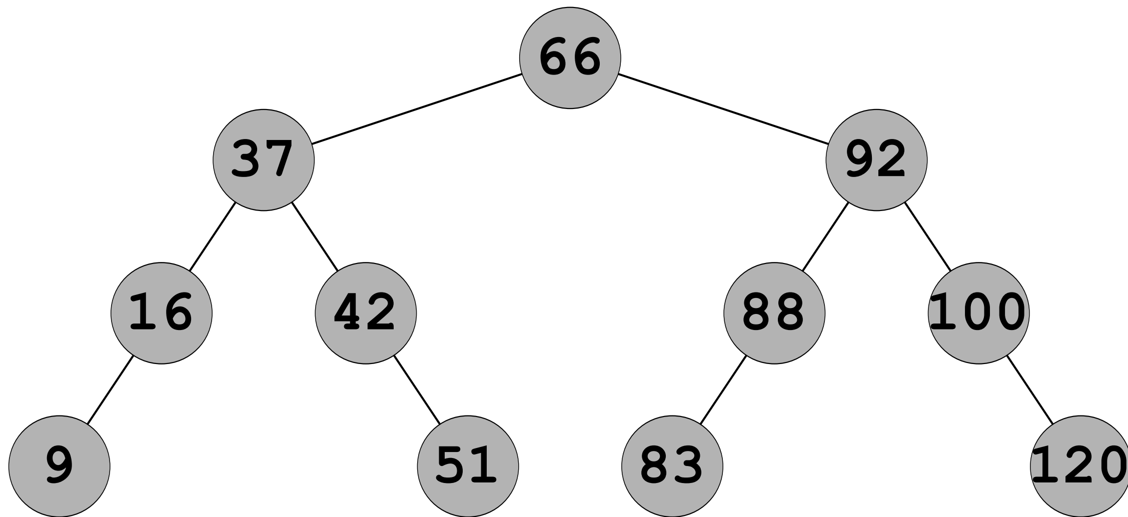
```
% one step in bubbleDown
int leftChildIndex=getLeftChildIndex(index,n)
int rightChildIndex=getRightChildIndex(index,n));
// if no left child then no right child either
if(leftChildIndex==NOT_FOUND) childIndex=NOT_FOUND
// wouldn't be here unless leftChindIndex is not NOT_FOUND
else if(rightChildIndex==NOT_FOUND) childIndex=leftChildIndex;
// both left and right child defined - which has the largest value?
else if(heap(leftChildIndex) > heap(rightChildIndex) childIndex=leftChildIndex;
else childIndex=rightChildIndex;

if(childIndex!=NOT_FOUND)
  if(heap(index) < heap(childIndex)) swapHeap(index,childIndex)
```

(6c) (7 marks) Given these three methods, consider writing one step of the `bubbleUp` method for heaps. Given a node at `index`, give the Java code segment that potentially moves that node **up** 1 level in the heap. Remember `heap[index]` contains the heap integer value at that index. You can use a helper method `swapHeap(index1,index2)`, which swaps the heap elements indexed by `index1` and `index2` if you need to. Again `n` is the current number of nodes in the heap and the heap is organized so that to root of the tree is the maximum node in the tree.

```
// one step in bubbleUp
int parentIndex=getParentIndex(index);
if (parentIndex!=index)
  if(heap(index) > heap(parentIndex)) swapHeap(index,parentIndex)
```

Problem 7 (35 marks)

Consider the following binary tree:



(7a) (5 marks) Give the **inorder** traversal of this tree:

9 16 37 42 51 66 83 88 92 100 120

(7b) (5 marks) Give the **preorder** traversal of this tree:

66 37 16 9 42 51 92 88 83 100 120

(7c) (5 marks) Give the **postorder** traversal of this tree:

9 16 51 42 37 83 88 120 100 92 66

(7d) (5 marks) Give the **level order** traversal of this tree:

66 37 92 16 42 88 100 9 51 83 120

In the methods below always use the BinaryTreeNode defined at the end of this exam and not the BinaryTreeNode use on this year's assignment 4 (although they are very similar). The BinaryTreeNode used here has getter and setter methods for getting and setting the element data, and the left and right children.

(7e) (5 marks) Consider the mystery traversal below. Use the BinaryTreeNode defined at the end of this exam and not the BinaryTreeNode use on this year's assignment 4. The BinaryTreeNode used here has getter and setter methods for getting and setting the element data, and the left and right children.

```
public static void mysteryA(BinaryTreeNode<String> node) {
  if(node==null) return;
  if((node.getLeft()!=null && node.getLeft().getLeft()!=null) && node.getRight()==null)
    System.out.format("%3s ",node.getElement());
  mysteryA(node.getLeft());
  mysteryA(node.getRight());
}
```

What is printed by this method for the binary tree on the previous page?

nothing is printed

(7f) (5 marks) Consider the mystery traversal below.

```
public static void mysteryB(BinaryTreeNode<String> node) {
  if(node==null) return;
  if((node.getRight()!=null && node.getRight().getRight()!=null) && node.getLeft()==null)
    System.out.format("%3s ",node.getElement());
  mysteryB(node.getLeft());
  mysteryB(node.getRight());
}
```

What is printed by this method for the binary tree on the previous page?

nothing is printed again

(7g) (5 marks) Consider the mystery traversal below.

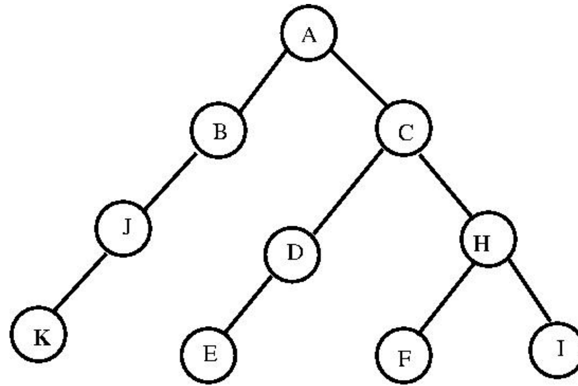
```
public static void mysteryC(BinaryTreeNode<String> node) {
  if(node==null) return;
  if((node.getLeft()!=null && node.getLeft().getLeft()==null) ||
     (node.getRight()!=null && node.getRight().getRight()==null))
       System.out.format("%3s ",node.getElement());
  mysteryC(node.getLeft());
  mysteryC(node.getRight());
}
```

What is printed by this method for the binary tree on the previous page?

16 42 88 100

Problem 8 (10 marks)

Write a recursive integer method `countSingleParents` that returns the numbers of nodes on the tree that are single parents (a node is single if it has 1 child node only). The BinaryTreeNode definition is given in the appendix. It has getter or setter methods for getting and setting the element data and the left and right children. If we run this method for the binary tree below the value 3 is returned (nodes B, J and D).




```
public static int countSingleParents(BinaryTreeNode<String> root) {
int count=0;
if(root==null) return count; // not a single parent
else
   {
   // if the left and right subtrees are null do nothing
   // Is the left node a single parent
   if(root.getLeft()!=null && root.getRight()==null)
      count=1+countSingleParents(root.getLeft());
   // Is the right parent a single node
   else if(root.getRight()!=null && root.getLeft()==null)
      count+=1+countSingleParents(root.getRight());
   // If the root is not a single parenet but has 2 children - check them
   else if(root.getLeft()!=null && root.getRight()!=null)
      count=countSingleParents(root.getLeft())+countSingleParents(root.getRight());
   }
return count;
}
```

Problem 9 (15 marks)

Consider 2 lists, `list1` and `list2`. Write a boolean method, `commonThreeOnly()` that returns `true` or `false` depending on whether the 2 lists have exactly 3 common strings or not. Lists that are less than 3 in length obviously fail this test. Duplicates count: we are searching for exactly 3 pairs of strings in the lists. See the ListADT at the end of the exam for its methods,

```
public static boolean commonThreeOnly(List<String> list1,List<String> list2) {
  Iterator it1=list1.iterator();
  Iterator it2=list2.iterator();
  int list1ct=0,list2ct=0;

  if(list1.size() < 3 || list2.size() < 3) return(false);

  while(it1.hasNext())
    if(list2.contains(it1.next())) list2ct=list2ct+1;
  while(it2.hasNext())
    if(list1.contains(it2.next())) list1ct=list1ct+1;
  System.out.println("list1_count:" + list1ct);
  System.out.println("list2_count:" + list2ct);
  return(list1ct==3 && list2ct==3);
}
```

Problem 10 (15 marks)

(10) This question concerns assignment 4. Consider level order traversal of a binary tree used by method `firstWidth()`, which finds the width of a binary tree as the maximum number of nodes at any level in the tree. Thus one must conduct a level order tree traversal to compute this. If we run this method on the binary tree in Problem 7 we get 4 returned. Write the code for this method:

```
public static int findWidth(BinaryTreeNode<String> root) {
  int maxWidth=0;
  LinkedQueue<BinaryTreeNode<String>> queue=new LinkedQueue<BinaryTreeNode<String>>();
  int numberNodesOnLevel=0;
  if(root==null) return 0;
  queue.enqueue(root);

  // We just keep the nodes on one level of the binary tree
  // on the queue at any one time
  while(!queue.isEmpty())
  {
    numberNodesOnLevel=queue.size();
    // if the number of nodes on the current level
    // is greated than the current values in maxWidth
    // update maxWidth to the new value
    if(numberNodesOnLevel>maxWidth)
    {
      maxWidth=numberNodesOnLevel;
    }
    while(numberNodesOnLevel>0)
    {
      BinaryTreeNode<String> node=queue.dequeue();
      if(node.getLeft()!=null) queue.enqueue(node.getLeft());
      if(node.getRight()!=null) queue.enqueue(node.getRight());
      numberNodesOnLevel--;
    }
  }
  return maxWidth;
}
```

Interfaces and Classes

```
public interface StackADT<T>{
// Adds one element to the top of this stack.
public void push (T element);

// Removes and returns the top element from this stack.
public T pop();

// Returns without removing the top element of this stack.
public T peek();

// Returns true if this stack contains no elements.
public boolean isEmpty();

// Returns the number of elements in this stack.
public int size();

// Returns a string representation of this stack.
public String toString();
}

public interface QueueADT<T>{
// Adds one element to the rear of this queue.
public void enqueue (T element);

// Removes and returns the element at the front of this queue.
public T dequeue();

// Returns without removing the element at the front of this queue.
public T first();

// Returns true if this queue contains no elements.
public boolean isEmpty();

// Returns the number of elements in this queue.
public int size();

// Returns a string representation of this queue
Public String toString();
}
```

```
public interface ListADT<T> extends Iterable<T>{
// Removes and returns the first element from this list.
public T removeFirst();

// Removes and returns the last element from this list.
public T removeLast();

// Removes and returns the specified element from this list.
public T remove(T element);

// Returns a reference to the first element in this list.
public T first();

// Returns a reference to the last element in this list.
public T last();

// Returns true if this list contains the specified target element.
public boolean contains(T target);

// Returns true if this list contains no elements.
public boolean isEmpty();

// Returns the number of elements in this list.
public int size();

// Returns an iterator for the elements in this list.
public Iterator<T> iterator();

// Returns a string representation of this list.
public String toString();
}

public interface OrderedListADT<T> extends ListADT<T>
{
/**
* Adds the specified element to this list at the proper location
*
* @param element  the element to be added to this list
*/
public void add (T element);
}
```

```
public interface UnorderedListADT<T> extends ListADT<T>
{
//  Adds the specified element to the front of this list
public void addToFront (T element);

//  Adds the specified element to the rear of this list
public void addToRear (T element);

//  Adds the specified element after the specified target
public void addAfter (T element, T target);
}
```

```
public interface Iterator<T>{
// Returns true if the iterator has more elements.
boolean hasNext();

// Returns the next element in the iterator.
T next();
}
```

```
public class BinaryTreeNode<T>{
protected T element;
protected BinaryTreeNode<T> left, right;

// the getters you may need
public T getElement();
public BinaryTreeNode<T> getLeft();
public BinaryTreeNode<T> getRight();
// the setters you may need
public T setElement();
public BinaryTreeNode<T> setLeft();
public BinaryTreeNode<T> setRight();
}
```

Rough work 1/4

Rough work 2/4

Rough work 3/4

Rough work 4/4