CMPT 225 D200, Fall 2026

Midterm Exam - SOLUTIONS
October 27, 2026

Name_____

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

Problem 1	
Problem 2	
Problem 3	
Problem 4	
TOTAL	

Instructions:

1. The duration of the exam is 100 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. Explain all your answers.
8. Really, explain all your answers.

Good luck!

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

Consider the following class. Assume it has a proper constructor and all setters/getters.

```
class Book {
    String title;
    String author;
    int year;
    // Constructors,setters,getters -- assume the strings are not null
    @Override
    public boolean equals(Object other) {
      if (this==other)
        return true;
      if (other==null || !(other instanceof Book))
        return false;

      Book book = (Book) other;
      return year == book.year &&
             title.equals(book.title) &&
             author.equals(book.author);
    }
}
```

A. (10 points) What will be the output of the following code? Explain

```
Book b1 = new Book("Dune", "Frank Herbert", 1965);
Book b2 = new Book("Dune", "Frank Herbert", 1965);
Book b3 = b1;

if (b1==b3)
   System.out.println("b1==b3 is true");
if (b2==b3)
   System.out.println("b2==b3 is true");
if (b2.equals(b3))
   System.out.println("b2.equals(b3) is true");
```

Answer:

- the first if is true because b1 and b3 point to the same object
- second is false, because b2 points to a different object
- Third if is true because the have the same data inside (same author, same title, same year)

The output is:

b1==b3 is true
b2.equals(b3) is true

B. (5 points each) Apply the Master Theorem to the following recursive definition, and express the running time as a closed formula.

- $T(n) = 2\,T(n/2) + \Theta(n^2)$, $T(1) = O(1)$

  Answer: Using Master Theorem we have
  $T(n) = a\,T(n/b) + f(n)$ with $(a=2, b=2)$ and $f(n) = \Theta(n^2)$.

  We get $c = \log_b(a) = 1$.

  Since $f(n) = \Theta(n^2)$ has exponent $2 > c$, the closed formula is $T(n) = \Theta(n^2)$.

- $T(n) = 4\,T(n/2) + \Theta(n)$, $T(1) = O(1)$

  Answer: Using Master Theorem we have
  $T(n) = a\,T(n/b) + f(n)$ with $(a=4, b=2)$ and $f(n) = \Theta(n)$.

  We get $c = \log_b(a) = 2$.

  Since $f(n) = \Theta(n)$ has exponent $1 < c$, the closed formula is $T(n) = \Theta(n^c) = \Theta(n^2)$.

C. (5 points) Write a function that gets a parameter n, and its running time on the input n can be expressed as $T(n) = 2T(n/2) + O(n)$ with $T(1) = O(1)$.

```
void foo(int n) {
   if (n > 0) {
     for (int  i = 0; i < n; i++)    // O(n) time
       System.out.println(i);
     foo(n/2);  // two recursive calls on n/2
     foo(n/2);
   }
}
```

Problem 2 (Linked Lists) [25 points]

Implement a class `LinkedList` that supports the following operations with running time O(1) for each operation.

```
public class LinkedList<T> {

   public LinkedList() // constructor creates an empty list

   public void addFirst(T item) // adds item to the front of the list

   public T getFirst() // returns the first item in the list

   public T removeFirst() // removes the first item and returns it

   public void addLast(T item) // adds item to the end of the list

   public T getLast() // returns the last item in the list

   public T removeLast() // removes the last item and returns it

   public int size() // returns the number of elements in the list

   public boolean isEmpty() // checks if the list is empty

}
```

Before writing code, explain your answer.

We define a class Node, and implement doubly linked list

```
private class Node<T> {
   private T data; // add a constructor that sets data
   private BTNode<T> next;
   private BTNode<T> prev;
}

private Node<T> head;  // defining data fields
private Node<T> tail;
private int size;

public LinkedList() {
  size = 0;
}

public int size() {
  return size;
}

public boolean isEmpty() {
  return size==0;
}
```

```java
public void addFirst(T item) {
  Node<T> newNode = new Node(item); // assume Node has a constructor
  if (head==null) // list is empty
    head = tail = newNode;
  else {
    newNode.next = head;
    head = newNode;
  }
  size++;
}

public T getFirst() {
  if (size == 0)    throw new NoSuchelementException("list is empty");
  return head.data;
}

public T removeFirst() {
  if (size == 0)    throw new NoSuchelementException("list is empty");
  T ret = head.data;
  head = head.next;
  size--;
  if (size == 0)
    head=tail=null;  // removed last element
  return ret;
}

public void addLast(T item)  {
  Node<T> newNode = new Node(item); // assume Node has a constructor
  if (head==null) // list is empty
    head = tail = newNode;
  else {
    newNode.prev = tail;
    tail = newNode;
  }
  size++;
}

public T getLast() {
  if (size == 0)    throw new NoSuchelementException("list is empty");
  return tail.data;
}

public T removeLast() { // if size==0 throw NoSuchElementException
  if (size == 0)    throw new NoSuchelementException("list is empty");
  T ret = tail.data;
  tail = tail.prev;
  size--;
  if (size == 0)
    head=tail=null;  // removed last element
  return ret;
}
```

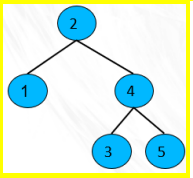
Problem 3 (Binary Trees) [25 points]

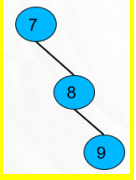
In this problem use the following definition of Binary Tree. You may assume the classes have the standard getters/setters.

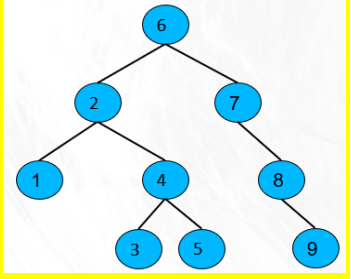
```
public class BTNode<T> {
      private T data;
      private BTNode<T> leftChild;
      private BTNode<T> rightChild;
      private BTNode<T> parent;
   }

public class BinaryTree<T> {
      private BTNode<T> root;
   }
```

A. (7 points) Draw a binary tree whose inOrder sequence is [1,2,3,4,5,6,7,8,9] and preOrder is [6,2,1,4,3,5,7,8,9].

Answer: the root is 6 because preOrder starts with 6.
On the left we have inOrder = [1,2,3,4,5] and preOrder [2,1,4,3,5]
On the right we have inOrder = [7,8,9] and preOrder [7,8,9]

- The left child of the root is 2
- On the left of 2 we have only 1
- On the right of 2 we have inOrder = [3,4,5] and preOrder [4,3,5]
- The left subtree is 

- The right child of the root is 7 and the right subtree is 

Solution: The tree is 

B. (18 points) Implement the class `PreOrderIterator` for the class `BinaryTree`. Your class needs to implement the following methods:

```
public PreOrderIterator(BinaryTree<T> tree)  // a constructor
public boolean hasNext()  // checks if the traversal has more vertices
public T next()  // returns the next item in the pre order
```

All operations must run in O(1) time in the worst-case.
Explain the idea first. Specify all private data fields and the algorithm.

```
public class PreOrderIterator<T> implements Iterator<T> {
```

Idea: we think of the implementation of preorder traversal using a stack.

- The constructor simply creates a new stack and adds the root to it
- Each iteration of the while loop corresponds to a call to next() method.
- the stopping condition stack is empty corresponds to hasNext()

```
private Stack<BTNode<T>> stack;

public PreOrderIterator(BinaryTree<T> tree) {a
    stack = new Stack<>();
    if (tree.getRoot() != null)
        stack.push(tree.getRoot());
}

@Override
public boolean hasNext() {as
    return !stack.empty();
}

@Override
public T next() {
    if (stack.empty())
        throw new NoSuchElementException("no more elements");
    BTNode<T> current = stack.pop();

    if (current.getRightChild() != null)
        stack.push(current.getRightChild());
    if (current.getLeftChild() != null)
        stack.push(current.getLeftChild());

    return current.getData();
}

}
```

Problem 4 (Binary Search Trees) [25 points]

A. (5 points) Consider the class BinarySearchTree, defined as follows.

```
public class BinarySearchTree<T extends Comparable<T>> {
  private BTNode<T> root;
};
```

Use the definition of `BTNode<T>` from Problem 3.

Write the method max() for the class BinarySearchTree that returns the maximal element in the tree. If the tree is empty, the method returns **`null`**.

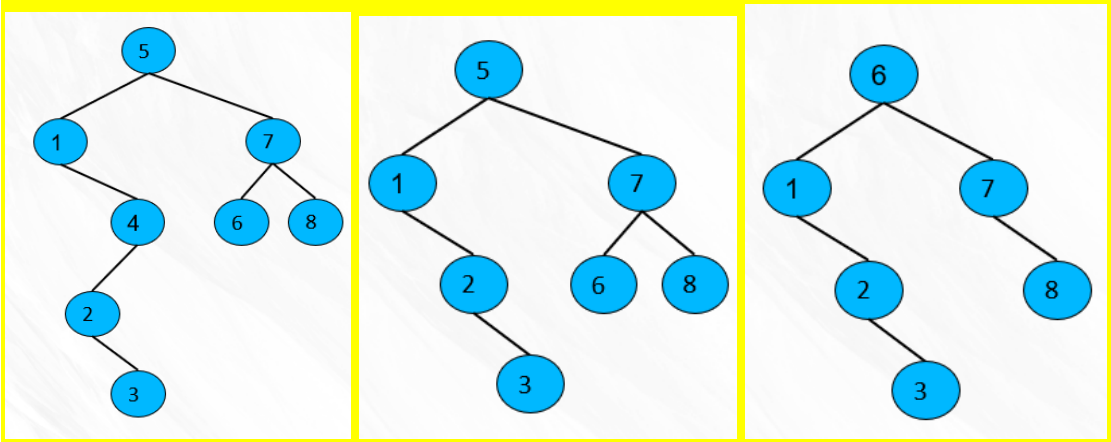
```
public T max() {
```

Idea: start from the root and go right

```
   BTNode<T> current = getRoot();

   while (current.getRightCild() != null)
       current = current.getRightCild()

   return current;


}
```

B. (5 points) Start with an empty tree

- Insert the elements [5, 1, 4, 7, 6, 8, 2, 3], and draw the obtained tree
- Remove 4 and draw the obtained tree
- Remove 5 and draw the obtained tree



C. (15 points) Write a function that gets an array with a post order traversal of a Binary Search Tree, and reconstructs the tree. The function returns the root of the tree. Use the definition of `BTNode<T>` from Problem 3.
Explain your answer.

```
public static <T extends Comparable<T>>
                        BTNode<T> postOrder2BST(T[] postOrder)
{
      return helper(postOrder, 0, postOrder.length - 1);

}


// helper function that gets the boundaries of our subarray
// the last element in the subarray is the root
// recursively create the left subtree and the right subtree
// and hang them under the root
private static <T extends Comparable<T>> BTNode<T>
                      helper(T[] arr, start, int end){


   // root is arr[end] -- that's easy
   BTNode<T> root = new BTNode<T>(arr[end]);

   // if start==end, simply return the root

   if (start < end) {

       // find first i, where postOrder[i] >= root
       // note that i will always stop with i<=end
       // because arr[end].compareTo(arr[end]) is zero
       int i = start;
       while (arr[i].compareTo(arr[end]) < 0)
           i++;

       // recursively reconstruct left subtree
       if (i > start)
           root.setLeftChild(helper(arr, start, i-1));

       // recursively reconstruct right subtree
       if (i < end)
           root.setRightChild(helper(arr, i, end-1));
   }

   return root;
}
```

Extra page

Master Theorem

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

Define $c = \log_b(a)$

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