Com S 228
Spring 2014
Final Exam

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: ____

ISU NetID (username): ____

Closed book/notes, no electronic devices, no headphones. Time limit **120 minutes**. Partial credit may be given for partially correct solutions.

- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

Please *peel off* pages **17-20** from your exam sheets for scratch purpose.

If you have questions, please ask!

Question	Points	Your Score
1	17	
2	14	
3	12	
4	12	
5	17	
6	14	
7	14	
Total	100	

1. (17 pts) These questions ask for the ***worst-case*** execution times of various methods or code snippets. Express your answers in big-O notation.

   a) (2 pts) What is the ***worst-case*** execution time of the **`peek()`** method on an *n*-element heap?

   b) (2 pts) What is the ***worst-case*** execution time of the **`add()`** method on an *n*-element heap?

   c) (2 pts) What is the ***worst-case*** execution time of the **`remove()`** method on an *n*-element heap?

   d) (2 pts) What is the ***worst-case*** execution time of the **`heapify()`** method on an *n*-element array?

   e) (3 pts) Suppose **`LinkedQueue<E>`** is a linked list implementation of queues, where all operations take constant time. **`Stack<E>`** is Java's implementation of stacks, where all operations take constant time. What is the ***worst-case*** execution time of the following code snippet in terms of *n*, the number of elements in **`qu`**?

```
public static <E> void reverseQueue(LinkedQueue<E> qu)
{
   if ( qu == null || qu.isEmpty() )
      return;
   Stack<E> tmp = new Stack<E>();
   while ( ! qu.isEmpty() )
      tmp.push( qu.dequeue() );
   while ( ! tmp.isEmpty() )
      qu.enqueue( tmp.pop() );
}
```

f) (3 pts) Suppose **c** is a **BSTSet** of Strings; that is, **c** is a set of **String**s stored in a binary search tree. What is the ***worst-case*** execution time of the following code snippet in terms of *n*, the number of **String**s in **c**?

```
for (String s : c)
{
   System.out.println(s);
}
```

g) (3 pts) Suppose that **G** is an undirected graph stored using an adjacency list and that **v** is a vertex in **G**. What is the ***worst-case*** time for executing depth-first search on **G**, starting at **v**, in terms of *n* and *m*, the number of vertices and edges of **G**, respectively?

2. (14 pts) Infix and postfix expressions.

a) (6 pts) Convert the following infix expression into postfix.

```
a ^ (b – 2 ^ (c + d))^2 – a * ((b – c / (4 – 5 / (c + d))) * b)
```

Postfix:

b) (6 pts) Convert the following postfix expression into infix.

```
a c a c a b + * + * + a b % c % d % -
```

Infix:

c) (2 pts) Consider the input **`String`** object **`I`** of a ***correct*** infix expression with *n* operators and operands in total. In big-O notation please give the running times of the following operations.

- In ______________________ time we can convert **`I`** into a postfix string **P**.
- In ______________________ time we can evaluate **P**.

3. (12 pts) Stacks and Queues

a) (3 pts) Given the following contents of an ***array implementation*** of a stack:

	0	1	2	3	4	5
stack	13	15	3	4		
				top		

Show the contents of **`stack`** and the location of **`top`** after the following sequence of operations:

```
stack.pop();
stack.pop();
stack.push(8);
stack.push(9);
stack.pop();
stack.push(11);
```

	0	1	2	3	4	5
stack						

b) (3 pts) Given the following contents of a ***circular array*** implementation of a queue:

	0	1	2	3	4	5
queue	20	30	10			
	f		b			

Show the contents of **queue** and the locations of **f** (front) and **b** (back) after the following sequence of operations:

```
queue.dequeue();
queue.enqueue(3);
queue.enqueue(4);
queue.enqueue(5);
queue.dequeue();
queue.enqueue(6);
```

	0	1	2	3	4	5
queue						

c) (6 pts) Write a static generic method called **`reverseStack`** that takes as a parameter an object named **`stk`** of type **`Stack<E>`**, where **E** is a type parameter, and returns **`void`**. If **`stk`** is null or empty, the method does nothing. Otherwise, the method ***reverses the order of element*s** in the **`Stack<E>`** object. You may use ***at most one*** additional **`LinkedQueue`** object to complete this task. You are ***not allowed*** to use any other types of objects, arrays, or even primitive types.

Only these three methods can be called from the **`Stack<E>`** object:

- `boolean` **`isEmpty();`** `// Tests if this stack is empty.`
- `E` **`pop();`** `// Removes and returns the top object in this stack.`
- `void` **`push(E item);`** `// Pushes an item onto the top of this stack.`

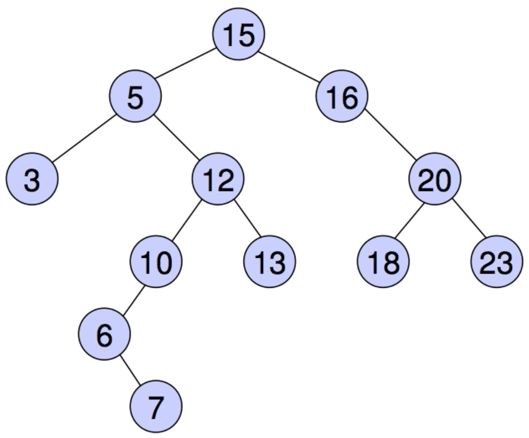
Only these three methods can be called from the **`LinkedQueue<E>`** object:

- `boolean` **`isEmpty();`** `// Tests if this queue is empty.`
- `E` **`dequeue();`** `// Removes and returns the front object of this queue.`
- `void` **`enqueue(E item);`** `// Pushes an item onto this queue.`

```
public static <E> void reverseStack(Stack<E> stk)
{
      if ( stk == null || stk.isEmpty() ) return;
      LinkedQueue<E> tmp = new LinkedQueue<E>();
      // TODO
















}
```

4. (12 pts) Tree traversal

a) (6 pts) Write the outputs from the pre-order, in-order, post-order traversals of the following binary tree.



Pre-order (2 pts):

In-order (2 pts):

Post-order (2 pts):

b) (6 pts) Write a public method **levelOrderTraversal()** that conducts a ***level-order traversal*** of a child-sibling tree, saves the level-order sequence of the tree in a **String** object, and returns the **String** object. The data items in the sequence should be separated by ***one or more*** spaces. Use the **toString()** method of a data item to generate its **String** form. (***Hint***: you may want to use a **LinkedQueue** and some of its methods listed in Problem 3c) for the level order traversal.)

```
public class Tree<E>
{
   private Node root;      // refers to the root node of a tree.
   private class Node
   {
         public E data;
         public Node parent;
         public Node firstChild;
         pubilc Node nextSibling;
         public Node( E item, Node prnt, Node fChild, Node sibling )
         {
               if ( item == null) throw new NullPointerException();
               data = item;
               parent = prnt;
               firstChild = fChild;
               nextSibling = sibling;
         }
   }


   public String levelOrderTraversal()
   {
         // TODO













   }
}
```

5. (17 pts) You are given the following array of 10 elements:

9	12	6	1	4	3	5	10	9	2

a) (3 pts) Draw the ***complete*** binary tree **T** generated from a left-to-right scan of the array elements, and by creating nodes for them top-down, starting at depth 0, and left to right at each depth.

b) (6 pts) Build a ***min*** binary heap by calling the **`heapify()`** method on the same array, starting at the last element that is a ***non-leaf node*** in the tree **T** above. Write out the contents of the array after the call. (For partial credit, you may want to draw on the next page intermediate tree configurations during the execution of **`heapify()`**.)

c) (8 pts) Perform the heapsort algorithm on the following array.

5	1	2	8

Sort the array ***in place*** with its elements in the ***increasing order***. Recall that heapsort starts by building a ***max*** heap. Write out the array contents after ***every swap*** (including any during the initial heap construction) by filling a blank array to the right of an update prompt. (Note that there may be more or fewer than the nine updates shown.)

Update 1:

Update 2:

Update 3:

Update 4:

Update 5:

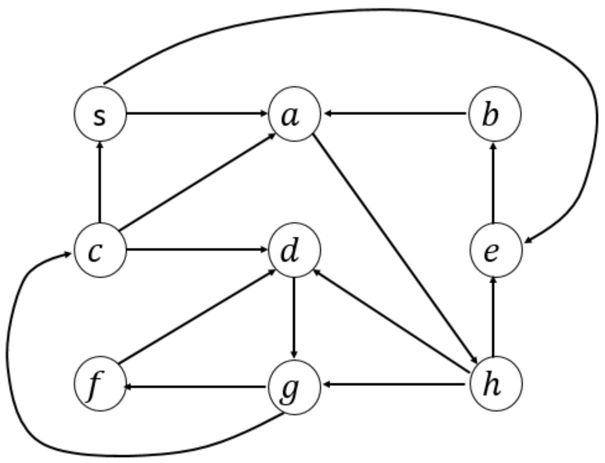
Update 6:

Update 7:

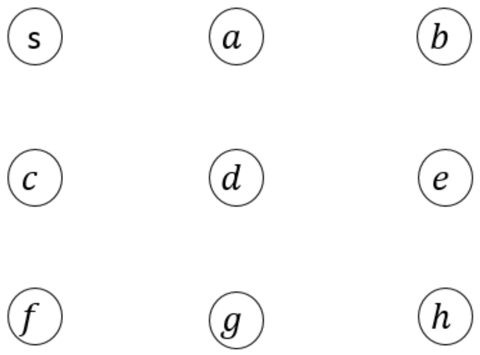
Update 8:

Update 9:

6. (14 pts) Consider the simple directed graph below with nine vertices.

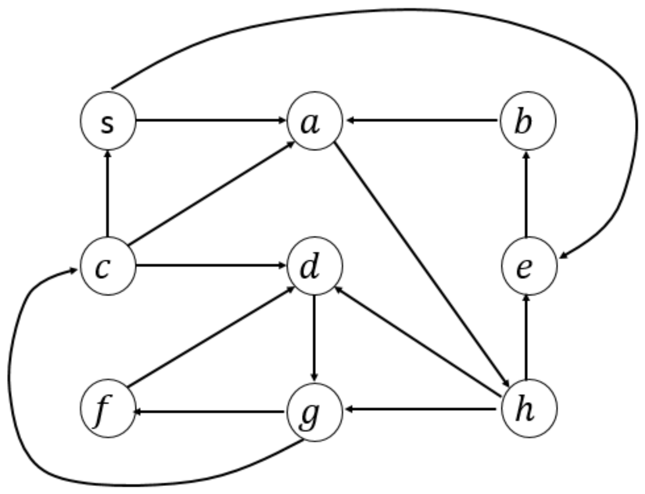


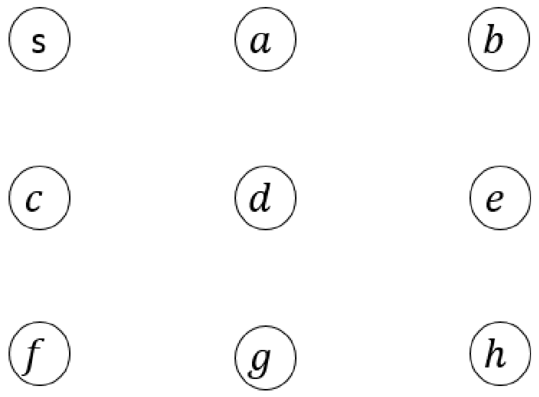
a) (7 pts) Perform a breadth-first search (BFS) starting at the vertex ***s***. Always choose a vertex in the ***alphabetical order*** in case of a tie. Draw the BFS tree by adding edges to the vertices below. Label each edge by the order in which it is added to the tree. For instance, the edge coming out of ***s*** to the next visited vertex (i.e., the first visited vertex that is not ***s***) is labeled 1, and so on. Also fill in the table the distance dist(V) of every vertex V from s. Assume that every edge has length 1.



vertex V	s	a	b	c	d	e	f	g	h
dist(V)									

b) (4 pts) Perform a depth-first search (DFS) starting at the vertex ***s***. Again, break a tie according to the ***alphabetical order***. In the below draw the DFS tree by adding edges to the vertices. As in a), label each edge by the order in which it is added to the tree, starting with 1. (For scratch convenience, the same graph is copied over first.)





c) (2 pts) Is the graph strongly connected? ____________ Yes ___________ No

d) (1 pt) Is the graph weakly connected? ____________ Yes ___________ No

7 (14 pts) Write a public method named **`removeRoot()`** for the binary search tree (**`BST`**) class below. The method deletes the root from the tree and returns the item stored there. It also ensures that the modified tree at the completion of the method is still a binary search tree by doing the following:

- The predecessor of the deleted root in the original tree will become the new root after the deletion.
- In case the root does not have a predecessor (i.e., it has no left subtree), its right child will become the new root.

All should be done via link updates. You are ***not allowed*** to copy the data stored at the predecessor to the root. You are ***not allowed*** to use any known method from the binary search tree class.

Note that for any node in the tree, its **`data`** item must be greater than all the **`data`** items stored in its left subtree, and less than all the **`data`** items in its right subtree.

```
public class BST<E extends Comparable<? super E>> extends AbstractSet<E>
{

  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;
    public Node right;
    public Node parent;
    public E data;

    public Node(E key, Node parent)
    {
      this.data = key;
      this.parent = parent;
    }
  }



  // other methods
  // ...


  public E removeRoot() throws IllegalStateException
  {

        E data;

        if (size == 0)
              throw new IllegalStateException("No root removal on an empty tree");

        data = root.data;
```

```
// BST has only one node. (1 pt)
if (size == 1)
{
    // TODO






}

// BST has two or more nodes but no left subtree.  (2 pts)
else if (root.left == null)
{
    // TODO











}

// BST has two or more nodes and a left subtree.
// find the predecessor of the root and promote it to the new root.
else
{
      Node cur = root.left;
      Node prnt = root;


      // find the predecessor of the root.
      while (cur.right != null)
      {
            prnt = cur;
            cur = cur.right;
      }



      // left child of root has a right subtree.
      if (prnt != root)
      {
            // update links related to the predecessor's subtree(s). (3 pts)
            // TODO
```

```
                    // set up the new root's left subtree.  (2 pts)
                    // TODO








                }

                // set up the new root's right subtree.  (3 pts)
                // TODO









                // set up the new root.  (2 pts)
                // TODO









            }


            // other updates if needed (1 pt)
            // TODO






            return data;
        }


        // other methods and iteration implementation
        // ...
    }
```

(this page is for scratch only)

(scratch only)

(scratch only)

(scratch only)