```
/**
* This is our code! Its goal is to sort a list of values of dtatatype E as a Tre
e sort
* CS 312 - Assignment 6
* @author(s) Mari Sisco, in collaboration with Emma Smith and Aidan Shaughnessy
*/

import java.util.List;
import java.util.Queue;
import java.util.LinkedList;
import java.lang.Comparable;


public class Tree<E extends Comparable<E>>
{
    protected List<E> list = new LinkedList<E>();
    protected Boolean debug;

    /*
    * purpose: initializes an empty tree
    * input:   none
    * result:  a Tree with nothing in it
    */
    public Tree()
    {
    list = null;
    debug = false;
    }

    /*
    * purpose: constructs a tree from a List of generic items
    * input:   the list and a debug flag
    * result:  a Tree with this.list = list. If debug == true print status infor
mation
    */
    public Tree(List<E> list, Boolean debug)
    {
    this.list = list;
    this.debug = debug;
    }

    /*
    * purpose: sort tree
    * input:   just the tree and the list
    * result:  the sorted list
    */
    public List<E> sort()
    {
        Queue<Node> queue = new LinkedList<Node>();
    List<E> output = new LinkedList<E>();

    for(E item : list)
    {
        queue.offer(new Node(item, null, null));
        }

        while (queue.size() > 1)
        {
            Node n1 = queue.poll();
            Node n2 = queue.poll();

        if (debug == true)
        {
                System.out.println("\npairing " + n1 + " and " + n2);
        }

        queue.offer(promote(n1,n2));
        if (debug == true)
        {
```



```
        for(Node node : queue)
        {
            System.out.println(node);
        }
        System.out.println("\n");
        }
        }

    if(queue.size() == 1)
    {
        output = atRoot(queue.peek());
    }

        return output;
    }


    /*
    * purpose: combine two Nodes into one while promoting the smallest Node
    * input:   nodes to be combined, n1 and n2
    * result:  promoted node with n1 and n2
    */
    protected Node promote(Node n1, Node n2)
    {
    int value = n1.compareTo(n2);

    if (value >= 0)
    {
        Node temp = n1;
        n1 = n2;
        n2 = temp;
    }

    if (n1.leftIsNull())
    {
        n1 = new Node(n1.data, n2, n1.right);
    }
    else if (n1.rightIsNull())
    {
        n1 = new Node(n1.data, n1.left, n2);
    }
    else if (n1.noneAreNull())
    {
        n1 = new Node(n1.data, promote(n1.left, n1.right), n2);
    }
    else // n1.bothAreNull
    {
        n1 = new Node(n1.data, n2, null);
    }

    //System.out.println(n1);
    return n1;
    }


    /*
    * purpose: At root, output root value and promote the two children
    * input:   node n
    * result:  List<E> of the sorted data
    */
    public List<E> atRoot(Node n)
    {
        if(debug == true)
    {
        System.out.println("Building up output list: ");
    }

    List<E> outputList = new LinkedList<E>();
    Node node = n;
```

```
  while(n.noneAreNull())
  {
    outputList.add(n.data);

    if (debug == true)
    {
          System.out.println(outputList);
    }

    n = promote(n.left, n.right);
  }

  if (n.leftIsNull() | n.rightIsNull())
  {
    if(n.leftIsNull())
        {
          while(n.bothAreNull() == false)
      {
        outputList.add(n.data);

        if (debug == true)
              System.out.println(outputList);

        n = n.right;
      }
    }
    else if(n.rightIsNull())
    {
      while(n.bothAreNull() == false)
      {
        outputList.add(n.data);

        if (debug == true)
        {
      System.out.println(outputList);
            }

        n = n.left;
      }
        }
  }
  if(n.bothAreNull())
      {
    outputList.add(n.data);
    if(debug == true)
    {
      System.out.println(outputList);
    }
  }

  //showing list if its size is less than 15
  if(debug == true || outputList.size() < 15)
  {
    System.out.println("ouput: " + outputList + "\n");
  }

    return outputList;
  }

  /*
  * purpose: create a human readable representation of a Tree
  * input:   none
  * result:  a String representation of the tree
  */
  @Override
  public String toString()
  {
  String s = "";
```



```
  for(E item : list)
  {
      s += item + " ";
  }
  return s;
  }


  // this is the beginning of Node class
  protected class Node
  {
      protected E data;
      protected Node left;
      protected Node right;

  /*
      * purpose: instantialize a Node
      * input:   none
      * result:  an empty node
      */
      public Node()
      {
      data = null;
          left = null;
          right = null;
      }

      /*
      * purpose: construct a Node
      * input:   E data, Node left, Node right
      * result:  this node
      */
  public Node(E data, Node left, Node right)
  {
      this.data = data;
      this.left = left;
      this.right = right;
  }

  /*
      * purpose: tell whether both children are null
      * input:   none
      * result:  true if both are null
      */
      public boolean bothAreNull()
      {
          return (left == null && right == null);
      }

  /*
      * purpose: tell whether left is null and right is not null
      * input:   none
      * result:  true if left is null and right is not null
      */
  public boolean leftIsNull()
  {
      return (left == null && right != null);
  }

  /*
      * purpose: tell whether left is not null and right is null
      * input:   none
      * result:  true if left is not null and right is null
      */
  public boolean rightIsNull()
  {
      return (left != null && right == null);
  }
```

```
    /*
    * purpose: tell whether neither children are null
    * input: none
    * result: true if both are null
    */
    public boolean noneAreNull()
    {
        return (left != null && right != null);
    }

    /*
    * purpose: compare data of two nodes
    * input: none
    * result: >0 if node > otherGuy, <0 if node < otherGuy, = 0 if node = otherG
uy
    */
    public int compareTo(Node otherGuy)
    {
        return data.compareTo(otherGuy.data);
    }


    /*
    * purpose: create a human readable representation of a node
    * input: none
    * result: a string represntation of the node
    */
        @Override
        public String toString()
        {
        if (left == null && right != null)
        {
        return data + "(null," + right.toString() + ")";
        }
        else if (right == null && left != null)
        {
        return data + "(" + left.toString() + ",null)";
        }
        else if (right == null && left == null)
        {
        return data + "(null,null)";
        }
        else
        {
            return data + "(" + left.toString() + "," + right.toString() + ")";
            }
    }

    } // this is the end of Node class

}
```



```
/**
* This is our code! Its goal is to interact with the user; it is the command-lin
e interface
* CS 312 - Assignment 6
* @author(s) Mari Sisco and Dr. David Binkley, in collaboration with Emma Smith
and Aidan Shaughnessy
*/

import java.util.Random;
import java.util.List;
import java.util.LinkedList;

public class Driver
{
  private String [] args;
  protected Boolean debug;
  protected Boolean timeit;

  /*
   * purpose: construct Driver class
   * input:   CLI arguments as a String Array
   * result:  updates private variable args, sets debug and timeit to false
   */
  public Driver(String [] a)
  {
    this.args = a;
    this.debug = false;
    this.timeit = false;
  }

  /* purpose: print out usage aka menu with options
   * input:   none
   * results: prints menu in CLI
   */
  private void usage()
  {
    System.out.println("Usage: [-d|-t] -n <numbers> | -r <count>");
  }


  /*
   * purpose: process the user's command
   * input:   none, uses private String array
   * result:  sorts a list of user-enetered numbers or of n random numbers.
   *          It is optional to time how long it took and to print the sorting p
rocess
   */
  public void parse()
  {
    if (args.length == 0)
    {
      usage();
      return;
    }

    int i = 0;

    if ("-d".equals(args[i]))    // args[0].equals("-d")
    {
      //print function from tree
      this.debug = true;
      i++;

      if (args.length > i && "-t".equals(args[i]))
      {
        this.timeit = true;
        i++;
      }
    }
```

```
    else if ("-t".equals(args[0]))
    {
      timeit = true;
      i++;

      if (args.length > i && "-d".equals(args[i]))
      {
        debug = true;
        i++;
      }
    }

    // -r and -n args[i]

    if (args.length > i && "-n".equals(args[i]))
    {
      i++;
      List<Integer> list = new LinkedList<Integer>();

      while(args.length > i)
      {
        int n = Integer.parseInt(args[i]);
    list.add(n);
    i++;
      }

      Tree<Integer> tree = new Tree<Integer>(list, this.debug);

      // calculates time taken to sort list
      if(timeit == true)
      {
        long startTime_ms = System.currentTimeMillis();

        tree.sort();

        long endTime_ms = System.currentTimeMillis();
    System.out.println("-> Sorting took " + (endTime_ms - startTime_ms) + " milliseconds"
);

    long memoryUsed = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().
freeMemory();
        System.out.println("It used "+ memoryUsed/(1024*1024) + "Mb");
      }
     else
       tree.sort();
    }

    else if(args.length > i && "-r".equals(args[i]))
    {
      i++;
      int count = Integer.parseInt(args[i]);

      List<Integer> list = new LinkedList<Integer>();
      Random r = new Random();
      int j = 0;
      while(j < count)
      {
        list.add(r.nextInt());
    j++;
      }

      Tree<Integer> tree = new Tree<Integer>(list, this.debug);

      // calculates time taken to sort list
      if(timeit == true)
      {
        long startTime_ms = System.currentTimeMillis();

    tree.sort();
```



```
        long endTime_ms = System.currentTimeMillis();
        System.out.println("-> Sorting took " + (endTime_ms - startTime_ms) + " millisec
onds");

    long memoryUsed = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().
freeMemory();
        System.out.println("It used "+ memoryUsed/(1024*1024) + "Mb");
      }
     else
       tree.sort();
    }

    else
    {
      usage();
      return;
    }

    System.out.println("debug = " + debug + " time it = " + timeit);
  }

  /*
   * purpose: run the program
   * input:   command from the user (taken from the command line)
   * result:  perfomrs task depending on the command entered by user
   */
  public static void main(String [] args)
  {
    Driver d = new Driver(args);
    d.parse();

  }

}
```