Data Structures

Final Assignment

Name : Nour Waled Ali Ali

ID : 18012006

الاسم : نور وليد علي علي

الرقم الجامعي : 18012006

المقرر : CSE22 Data structur I

القسم : حاسبات

الفرقة : الأولى

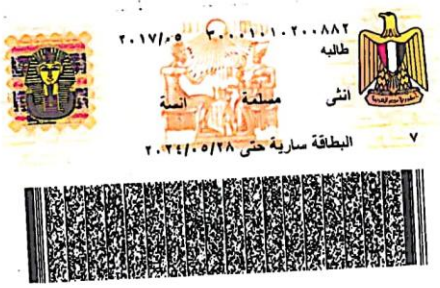
تاريخ الاختبار : 7 - 6 - 2020

أقر أنا الطالب الموضح اسمه ورقم جلوسه على هذا النموذج بأنني قمت بنفسي بالاطلاع على هذا النموذج وقمت بالحصول على نسخة الاختبار المرفق بالنموذج وقمت بحل الاختبار بنفسي طبقا للقواعد المعلنة ولم اتلقى أي مساعدة من أحد بالمخالفة لقواعد الاختبارات وأنني المسئول عن الاجابات التي تم تسجيلها من هذا الحساب الالكتروني واتحمل كامل المسئولية القانونية في حال مخالفتي لقواعد الاختبار

الاسم : نور وليد علي علي

التوقيع : نور وليد علي


جمهورية مصر العربية
بطاقة تحقيق الشخصية
نور
وليد على على فرج الله
٥ش السلطان حسين
العطارين - الاسكندرية
٣٠٠٠١٠١٠٢٠٠٨٨٢
HA6498671


٣٠٠٠١٠١٠٢٠٠٨٨٢
طالبه
انثى
مسلمة
انسة
٢٠١٧/٠٥
البطاقة سارية حتى ٢٠٢٤/٠٥/٢٨

```
import java.util.Arrays;

public class FinalAssignmentSolution {

	public class MiscUtils implements IMiscUtils {

	/**
	 * Inserts an integer value in a BST of integers.
	 * Result should be a valid BST.
	 * BST has no duplicates.
	 *
	 *@param root: BST root, a valid BST of integers
	 *@param element: integer value to insert.
	 *
	 * */
	@Override
	public BinaryTreeNode insert(BinaryTreeNode root, int element) {
			if(root == null)
				return new BinaryTreeNode(element);

	if(element < (Integer)root.element)
		{ root.left = insert(root.left,element);}

	else if(element > (Integer)root.element)
		{root.right = insert(root.right,element);}

	else{
		// element = root.element
		  return root;
		}
		  return root;
			}
			/**
			* Returns the sum of the elements in the tree in
			* the specified range [low, high] inclusive.
			*
			*@param root: BST root, a valid BST of integers.
			*@param low: range lower limit.
			*@param high: range upper limit.
			*
			* */
	@Override
	public int sumRange(BinaryTreeNode root, int low, int high) {

				int sum = 0;
				if(root == null || high<low)
					return sum;

				Stack s = new Stack();

				s.push(root);

				while(!s.isEmpty()){
				BinaryTreeNode current = (BinaryTreeNode) s.pop();
  if((Integer)current.element >= low && (Integer)current.element <= high){
					sum += (Integer)current.element;
				 }
		  if((Integer)current.element > low && current.left != null){
```

```
                        s.push(current.left);
                    }

        if((Integer)current.element< high && current.right !=null){
                        s.push(current.right);
                    }
                  }
                 return sum;
            }
             /**
              * Returns true if the input is a valid BST, false otherwise
              * @param root: Tree root.
              *
              * */
    @Override
    public boolean isValidBST(BinaryTreeNode root) {
                    if(root == null){ return true; }
        return isBstValid(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
             /**
              * Returns true if the input is a valid BST, false otherwise
              * @param root: Tree root.
              * @param minValiue = -2147483648
              * @param minValiue =  2147483648
              *
              * */
 private boolean isBstValid(BinaryTreeNode root, Integer minValue, Integer
maxValue) {

        if(root == null) return true;
    if((Integer)root.element > minValue && (Integer)root.element < maxValue
     && isBstValid(root.left, minValue, (Integer) root.element)
        && isBstValid(root.right, (Integer)root.element, maxValue)) {
                        return true;
                } else {
                        return false;
                    }
                }
              /**
              * Given an array of integers, return an array containing
              * the indices of the next smaller number of every number
              * or -1 if the next smaller number does not exist. .
              * @param array: array of numbers.
              * @throws: throws an exception in the input array is null.
              * */
    @Override
    public int[] nextSmallerNumber(int[] array) {
                    int size = array.length;
            if(array == null || size == 0) {
                    throw new RuntimeException();
            }
            Stack index = new Stack(); // To push in new array
            int[] arr = new int[size]; //  return
                 index.push(0);  // push first element in array [0]
            for (int i = 1; i < size; i++) {

                    if (index.isEmpty()) { // true
                        index.push(i);
                        continue;
```

```
                }

        while (!index.isEmpty() && array[(int) index.peek()] > array[i]) {
                                        // index
                        arr[(int) index.pop()] = i;
                }
                    index.push(i);
            }
                    while (!index.isEmpty()) {
                             // -1
                        arr[(int) index.pop()] = -1;
                    }
                    return arr;
                    }
            }
        public class HashTableDictionary<K,V> implements IDictionary<K,V> {

                 private class both{
                         K key;
                         V value;
                    both(K key,V value){
                    this.key = key;
                    this.value = value;
                    }
                }
                    private int N; // size of array
                    private both[] arr; // array of hash table
                    private int counter; // check is Empty

      public HashTableDictionary(int max_size) {
                   this.N = max_size;
                    Object[] temp = new Object[max_size];
                  this.arr =Arrays.copyOf(temp,max_size,both[].class);
                    this.counter = 0;
            }

  private int get_index(K key) {

   return (Math.abs(key.hashCode())& 0x7fffffff) % N;  // get index in array
         }
              /**
               * Retrieves the value corresponding to the specified key.
               * Returns null if the key doesn't exist in the dictionary.
               * @param key: key.
               * @throws: Throws Exception if the key is null.
               * */
        @Override
        public V get(K key) {

              if(key == null)
                 throw new RuntimeException();

              int index = get_index(key); // index of array

                   while(arr[index] != null) {
                      if(arr[index].key.equals(key)) {
                           return arr[index].value;
                      }
                      index = (index+1)%N;
```

```
                    }
                return null;
            }
        /**
         * Inserts a new pair of the key and value in the dictionary.
         * If the key already exists, the old value is returned.
         * If the key doesn't exist the, null is returned,
         * @param key: key.
         * @param value: value.
         * @throws: Throws an exception if the key or the value is null.
         * */
    @Override
    public V set(K key, V value) {
                if(key == null || value == null) {
                    throw  new RuntimeException();
                    }
                    int index = get_index(key);
                    int check = index;
                    do {
                     if(arr[check] == null) {
                     both t = new both(key,value);
                           arr[check] = t;
                           counter++;
                           return value;
                     }
          if(arr[check].key.equals(key)) { //key already exists
                           V s = arr[check].value;
                        arr[check].value = value;
                           return s;
                     }
                    check = (check+1) % N;
                    }
                while(check != index);

                   return null;
                }
                /**
                * Removes the key and returns its value.
                * @param key: key
                * @throws: throws exception if the key is null
                *
                * */
        @Override
        public V remove(K key) {
                    if(key == null)
                           throw new RuntimeException();

                int index = get_index(key);

                while(arr[index] != null) {
                      if(arr[index].key.equals(key)) {
                           V r = arr[index].value;
                           arr[index] = null;
                           counter--;
                           return r;
                      }
                      index = (index+1)%N;
                  }
                    return null;  // not found
```

```
                }
        /**
         * Returns true if the dictionary is empty and false otherwise.
         * */
                @Override
                public boolean isEmpty() {
                        return counter == 0;
                }
    }
public class TreeDictionary<K, V> implements IDictionary<K,V>{

                private int counter = 0; // check size
                private Node root;
                private class Node{
                  private K key;
                  private V value;
                  private Node left,right;
                  public Node(K key,V value) {
                        this.key=key;
                        this.value=value;
                   }
                 }
               /**
                * Retrieves the value corresponding to the specified key.
                * Returns null if the key doesn't exist in the dictionary.
                * @param key: key.
                * @throws: Throws Exception if the key is null.
                * */
               @Override
               public V get(K key) {
                    if(key == null)
                       throw new RuntimeException();

                        Node x = root;
      while (x != null){
int check = ((Comparable) key).compareTo((K)x.key); // check >> key in root
                    if(check < 0) {
                        x = x.left;
                        }
                    else if(check > 0) {
                        x = x.right;
                        }
                    else if (check == 0) {
                        return x.value;
                           }
                        }
                  return null;
                }
    /**
     * Inserts a new pair of the key and value in the dictionary.
     * If the key already exists, the old value is returned.
     * If the key doesn't exist the, null is returned,
     * @param key: key.
     * @param value: value.
      * @throws: Throws an exception if the key or the value is null.
      * */
                @Override
                public V set(K key, V value) {
                        if(key == null || value == null) {
```

```
                    throw  new RuntimeException();
                }
                if(root == null) {
                root = new Node(key,value);
                counter++;
                return value;
                  }
        return put(root,key,value);
        }
/**
* Inserts a new pair of the key and value in the dictionary.
* If the key doesn't exist the, null is returned,
* @param key: key.
* @param value: value.
* @param Node: n
* return value in node
* */
private V put(Node n , K key ,V value) {
    int check = ((Comparable) key).compareTo(n.key);

                if (check < 0) {
                    if (n.left == null) {
                        n.left = new Node(key, value);
                        counter++;
                        return value;
                    } else {
                        return put(n.left, key, value);
                    }
                }
                if (check > 0) {
                    if (n.right == null) {
                        n.right = new Node(key, value);
                        counter++;
                        return value;
                    } else {
                        return put(n.right, key, value);
                    }
                }
                V p = n.value;
                n.value = value;
                return p;
          }
        /**
        * Removes the key and returns its value.
        * @param key: key
        * @throws: throws exception if the key is null
        *
        * */
        @Override
        public V remove(K key) {
               if(key == null) {
                     throw new RuntimeException();
                     }
                Node removed = this.findKey(key,root);
                 if (removed==null) {
                     return null;
                 } else {
                     V val = removed.value;
                     counter--;
```

```
                root = remover(key,root);
                return val;
            }
        }

        public Node remover(K key, Node r) {
          if (r==null) {
              return r;
          }
          int check = ((Comparable) key).compareTo(r.key);
          if (check < 0) {
              r.left = remover(key, r.left);
          } else if (check > 0) {
              r.right = remover(key, r.right);
          } else {
              if (r.left==null) {
                  r = r.right;
              } else if (r.right==null) {
                  r = r.left;
              } else {
                  Node min = r.right;
                  while (min.left != null) {
                      min = min.left;
                  }
                  r.key = min.key;
                  r.value = min.value;
                  r.right = remover(min.key, r.right);
              }
          }
          return r;
      }
        private Node findKey(K key, Node r) {
          if (r==null) {
              return r;
          }
          int check = ((Comparable) key).compareTo(r.key);
          if (check < 0) {
              return findKey(key, r.left);
          } else if (check > 0) {
              return findKey(key, r.right);
          } else {
              return r;
          }
      }

        @Override
        public boolean isEmpty() {

            return counter == 0;
        }

    }           public class Stack {

        public class Node {

            public Node next = null;
            public Object value;

            public Node(Object element) {
```

```
                                this.value = element;
                        }
                    }
                Node top = null;
                int len = 0;
        /**
         * Removes the element at the top of stack and returns that element
         * @return top of stack element, or through exception if empty
         */
              public Object pop() {
                    if(len>0) {
                          Object p = top.value; // get value
                          top = top.next;  // remove this element
                          len--;
                          return p;}
                    else {
                          throw new RuntimeException(); // stack is Empty
                          }
                    }
       /**
        * Get the element at the top of stack without removing it from stack.
        * @return top of stack element, or through exception if empty
        */
        public Object peek() {
                    if(len>0) {
              Object p = top.value;     // get value without removing it
                    return p;
              }
              else {
                    throw new RuntimeException(); // stack is Empty
              }
                }
                    /**
                    * Pushes an item onto the top of this stack.
                    * @param element
                    * to insert
                    */
             public void push(Object element) {
                          Node i = new Node(element);
                          i.next = top;
                          i.value = element; // Take the value
                          top = i ;  // Add it
                          len ++;
                     }
                    /**
                    * Tests if this stack is empty
                    * @return true if stack empty
                    */
                    public boolean isEmpty() {
                          if(len==0) { // Empty
                                return true;
                           }
                          else { //Not Empty
                          return false;
                    }
              }
        }
}
```