

Name : Mustafa Cem ONAN

ID : 180315064

DATA STRUCTURES – SEMESTER ASSIGNMENT: BAG ADT REPORT

1) Project Summary:

This project is about to create a Bag Abstract Data Type Application with the features explained in the assignment.

This project contains 4 classes such as “Node.java”, “Bag.java” “Car.java” “Test.java”. Source codes can be found in the “src” directory in the given .zip file.

Node class represents the data units that is used by Bag ADT. And the Bag is the Abstract Data Type that is actually based on a “Binary Search Tree”. Bag class has all the 10 required methods they are written in the assignment, and also it has some extra methods.

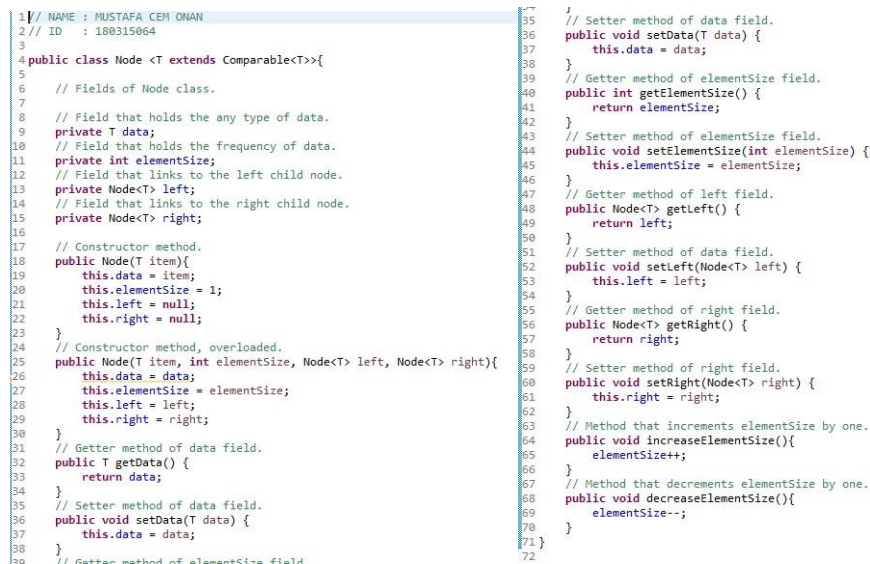
Test class has the main method, and all the tests of required methods has done in this class. And the Car class is for only generic data testing.

Various features required are mentioned below sequence wise along with the explanation of where those features can be found in the source code.

2) Node Class:

A. Introduction and Explanation of Basic Usage of Node Class:

- First of all, an abstract data type should be based on a data structure. In this project, it's preferred to use a Binary Search Tree as the base data structure. And this also needs a Node class in order to hold the given data, and link to the another nodes.



```
1// NAME : MUSTAFA CEM ONAN
2// ID : 180315064
3
4public class Node <T extends Comparable<T>>{
5
6    // Fields of Node class.
7
8    // Field that holds the any type of data.
9    private T data;
10    // Field that holds the frequency of data.
11    private int elementSize;
12    // Field that links to the left child node.
13    private Node<T> left;
14    // Field that links to the right child node.
15    private Node<T> right;
16
17    // Constructor method.
18    public Node(T item){
19        this.data = item;
20        this.elementSize = 1;
21        this.left = null;
22        this.right = null;
23    }
24    // Constructor method, overloaded.
25    public Node(T item, int elementSize, Node<T> left, Node<T> right){
26        this.data = data;
27        this.elementSize = elementSize;
28        this.left = left;
29        this.right = right;
30    }
31    // Getter method of data field.
32    public T getData() {
33        return data;
34    }
35    // Setter method of data field.
36    public void setData(T data) {
37        this.data = data;
38    }
39    // Getter method of elementSize field.
40    public int getElementSize() {
41        return elementSize;
42    }
43    // Setter method of elementSize field.
44    public void setElementSize(int elementSize) {
45        this.elementSize = elementSize;
46    }
47    // Getter method of left field.
48    public Node<T> getLeft() {
49        return left;
50    }
51    // Setter method of left field.
52    public void setLeft(Node<T> left) {
53        this.left = left;
54    }
55    // Getter method of right field.
56    public Node<T> getRight() {
57        return right;
58    }
59    // Setter method of right field.
60    public void setRight(Node<T> right) {
61        this.right = right;
62    }
63    // Method that increments elementSize by one.
64    public void increaseElementSize(){
65        elementSize++;
66    }
67    // Method that decrements elementSize by one.
68    public void decreaseElementSize(){
69        elementSize--;
70    }
71 }
72
```

Image 1: Screenshot of the Node class.

B. Class Header

- Since it's unknown which type of data will be stored in this ADT, Java Generics has been used to get any type of data as arguments.

- Also, in order to place nodes as right or left child, there must be operated a compare algorithm. So, in diamond brackets, used T to represent any type, and it's also inherited from the Java's Comparable interface to operate .compareTo() method.

C. Fields

- There are 4 fields in Node class such as: data, elementSize, left and right.
- data field holds the any type of data.
- elementSize field holds the frequency of a data.
- left field links to the left-child node.
- right field links to the right-child node.

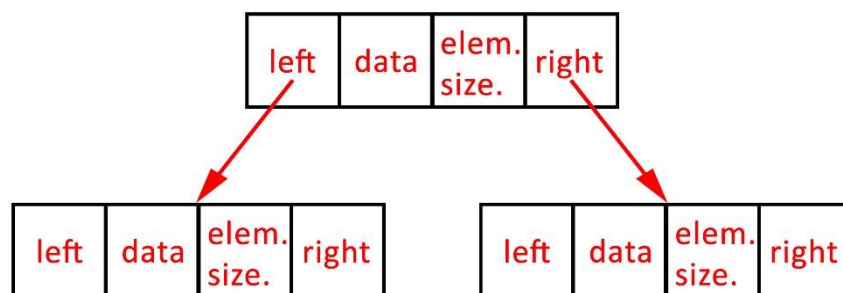


Image 2: A representation of the Node Structure.

D. Methods

- There are two constructor methods in this class. First one only takes the any type of data as the argument, and sets the elementSize to 1, rest is setted to null. And another constructor (method overload) takes all 4 states as the arguments, and sets it's fields to this given arguments.

```

14      // Constructor method.
15      public Node(T item){
16          this.data = item;
17          this.elementSize = 1;
18          this.left = null;
19          this.right = null;
20      }
21      // Constructor method, overloaded.
22      public Node(T item, int elementSize, Node<T> left, Node<T> right){
23          this.data = item;
24          this.elementSize = elementSize;
25          this.left = left;
26          this.right = right;
27      }
  
```

Image 3: A screenshot of constructor methods of the Node class.

- There are getter and setter methods of all the fields, and also there are 2 extra methods to increment and decrement elementSize by one.

3) Bag Class:

A. Introduction and Explanation of Basic Usage of Bag Class:

- Bag class is an ADT, that is actually based on a Binary Search Tree algorithm. An empty bag has just one root node and it links to null. Every nodes can have maximum 2 child nodes such as left or right. And also greater child nodes are placed on the right sub-tree of their parent's, and the less child nodes are placed at the left sub-tree of their parent's
- There are many other data structures could be used to implement this ADT such as linked lists, array lists, AVL trees etc. But it's preferred to use Binary Search Tree in this project, reasons of this choice is explained along this section:
- Lists are easy to implement, but they have $O(n)$ linear time cost on removing and traversal operations. Since nodes are placed on the left sub-tree when they are less than it's parent, and placed on the right-sub tree when they are greater than it's parent, it's possible to operate traversal operations with $O(\log n)$ time cost on Binary Search Trees.
- When it comes to compare Binary Search Trees with AVL trees, AVL trees are more balanced in practise. So it can be said that, both Binary Search Trees and AVL trees has $O(\log n)$ traversal time cost but, since AVL trees has less levels than the Binary Search trees, AVL trees has less traversal time cost. Because they are balancing themselves after every adding and removing operation. But, when the collection reach a large number of nodes, balancing operations would have a significant extra time cost. Because of this, it's preferred to use Binary Search Tree as the main structure rather than an AVL tree in this project.

B. Class Header

- Since it's unknown which type of data will be stored in this ADT, Java Generics has been used to get any type of data as arguments.
- Also, in order to place nodes as right or left child, there must be operated a compare algorithm. So, in diamond brackets, used T to represent any type, and it's also inherited from the Java's Comparable interface to operate .compareTo() method.

```
1 // NAME : MUSTAFA CEM ONAN
2 // ID   : 180315064
3
4 public class Bag <T extends Comparable<T>> {
5
```

Image 4: A screenshot of the class header of Bag.

C. Fields

- This class has only one field that is named as “root”. This field has the type of Node and represents the root node of the bag.

```
4
5 //Fields of Bag<T extends Comparable<T>>
6 private Node<T> root;
7
```

Image 5: A screenshot of the fields of Bag.

D. Methods

- There is only one constructor method for this class. And it's references to another method which named as “clear()”.

```
8 //Constructor of Bag<T extends Comparable<T>>
9 public Bag(){
10     clear();
11 }
12
```

Image 6: A screenshot of the constructor method of the Bag.

- And the clear method is sets the root to null. So, since there is no handler for rest of the nodes, all the nodes will be destroyed by the Java's Garbage Collector, automatically when clear method is invoked.

```
38
39 //Method that resets the Bag.
40 public void clear(){
41     root = null;
42 }
```

Image 7: A screenshot of the clear method.

- There are two add methods in Bag class. First one takes only an any type of data as the argument and has the return type of void, and the another one takes a Node and an any type of data as the arguments and has the return type of Node. So there is used a method overloading.
- Actual adding process are handling by the second add method. First one is used only implemented for encapsulating purposes. First one only takes data argument from the end user, and invokes the second add method as it's starting point is root node. Finally, sets the root to the returned node from the second add method.

```
13 //Required methods starts
14
15 //Method that adds a data to the Bag.
16 public void add(T item){
17     root = add(root, item);
18 }
19
```

Image 8: A screenshot of the first add method.

- It's going to be examined the second add method along this section of the report. This method is simply, takes an argument which named as current and it has the type of Node. This represents the starting point of adding operations. And the another argument which named as item, has the generic type, represents the given data which will be holded by the new added node.
- This method, first checks if the current node is null or exists, if it is null, then it references to create a new Node with the given item, and returns the current node.

```
--
21 //Method that adds a data to the Bag.
22 private Node<T> add(Node<T> current, T item){
23     if (current == null){
24         current = new Node<>(item);
25         return current;
26     }
```

Image 9: A screenshot of the topside of the second add method.

- And if the current node is not null, then the method checks the 3 conditions. Recursively invokes itself in those conditions and returns the current node until it find a node that holds the given item or until if the bag doesn't contains a node that holds the given item after a compared traversal. If there exists a node that holds the given item, then it's only incremented it's element size by one. But if there are no nodes in the bag that holds the given item, then it creates a new node with the given item in it's constructor, and links this to the proper place on the bag.
- First, method compares given item and the data of the current node. If current node's data is less than the given item, then invokes the add method recursively for the right child of the current node.
- Else if current node's data is greater then the given item, then it invokes the add method recursively for the left child of the current node.
- As it can be seen, there are two base cases for this recursive structure:
- It's compared and recursively invoked again and again until the current node is null or current node's data and given item are equal to each other. Finally, if current is null, it means there are no nodes in the bag that holds the given item. So as it's mentioned above; it references the current node to create a new node with the given item and then returns current node. If current is not null, it means there is a node exists in the bag that holds the given item. In this case, it's incremented the current node's element size by invoking increaseElementSize() method, then returns the current node.

```
--
21 //Method that adds a data to the Bag.
22 private Node<T> add(Node<T> current, T item){
23     if (current == null){
24         current = new Node<>(item);
25         return current;
26     }
27     if (current.getData().compareTo(item) < 0){
28         current.setRight(add(current.getRight(), item));
29     }
30     else if (current.getData().compareTo(item) > 0){
31         current.setLeft(add(current.getLeft(), item));
32     }
33     else if (current.getData().compareTo(item) == 0){
34         current.increaseElementSize();
35     }
36     return current;
37 }
```

Image 10: A screenshot of the second add method.

- Next, it's going to be examined the contains() method. This method, makes a traversal on the Bag, and if there exists a Node that holds the given item, it returns true. Else, it returns false. There is also made a method overloading for contains() methods. First one takes only one argument which named as item and it has generic type. And it only returns the result by invoking the second contains() method. Actual search operations are made in the second method.

```
//Method that returns true if given data exists in the Bag.
public boolean contains(T item) {
    return contains(root, item);
}
```

Image 11: A screenshot of the first contains() method

- Second method, takes two arguments such as first one is named as current and has the type of Node, and the other one is named as item and has the generic type. current, represents the starting node of traversal. Similarly to add method, it compares current node's data and given item and until it is found that they are equal or until there are no more nodes that holds the given item after a traversal. If there exists a node that holds the given item in it's data field, then it returns true. Else, it returns false.

```
49 //Method that returns true if given data exists in the Bag.
50 private boolean contains(Node<T> current, T item) {
51     if (current != null) {
52         if (current.getData().compareTo(item) == 0) {
53             return true;
54         }
55         else if (current.getData().compareTo(item) > 0) {
56             return contains(current.getLeft(), item);
57         }
58         else if (current.getData().compareTo(item) < 0) {
59             return contains(current.getRight(), item);
60         }
61     }
62     return false;
63 }
```

Image 12: A screenshot of the second contains() method.

- Next it's going to be examined the distinctSize() method. This method, makes a complete in-order traversal (LNR) on the bag, and counts the node number. Finally, it returns the count number after traversal completed.
- There are two distinctSize() methods so there is made a method overloading. First method does not take any argument and has the return type of int. It only invokes the second distinctSize() method with the argument as root node, and returns it's result.

```
64
65 //Method that returns the distinct element size of the Bag.
66 public int distinctSize(){
67     return distinctSize(root);
68 }
69
```

Image 13: A screenshot of the first distinctSize() method.

- Actual operations, made by the second method. It takes an argument which is named as current and has the type of Node. And this method has the return type of int.
- First, it defines an integer which is named as i. This integer is initialized to 0. This integer is defined in order to count each nodes on traversal.
- Then it checks if the current node is null, if it is returns 0. It means if the bag is empty, if it is, since there are no nodes exist in bag, there are no size or distinct size can be different from zero.
- After that, it makes a complete in-order traversal, and increments the i by one on each recursive iteration. Finally, returns the i as the distinct size of the bag.

```

70 //Method that returns the distinct element size of the Bag.
71 private int distinctSize(Node <T> current) {
72     int i = 0;
73
74     if (current == null){
75         return 0;
76     }
77     else{
78         i += distinctSize(current.getLeft());
79         i += 1;
80         i += distinctSize(current.getRight());
81     }
82     return i;
83 }

```

Image 14: A screenshot of the second distinctSize() method.

- Next, it's going to be examined the equals() method now. Since each class is inherited from the Object class in Java, and the each class has the equals() method by the Object has it, it should be overridden by the implementer. Else, equals method would not work properly.
- equals() method, takes an argument which is named as obj and has the type of Object, and method has the return type of boolean. Basicly, it compares the given object with the this bag. If they are equal, returns true, else, returns false.
- To do this comparison, first given object must checked if it is an instance of the Bag class. At the first condition, it's checked.
- Then, there is an if statement that checks if the string outputs of those 2 bags are equal to each other or not, by using equals() and toString() methods. It should be noted that, the toString method is already overridden for Bag class. It will be examined at the further paragraphs. If the string outputs of those 2 bags are equal to each other, then it returns true. Else, it returns false.
- This is a tricky operation. Here is the logic behind comparing string outputs for equality check : There could be made a complete traversal and checking each element's element size by recursively for those 2 bags. But it would be cost a significant time cost. Instead of this, if those 2 bags are equal, then it means the string outputs must be equal to each other too.

- As it's mentioned in assignment, order of elements is insignificant but the number of instances is significant on equality of 2 bags. Since if those bags have same instance numbers on the same elements, and since they are the Binary Search Trees, whether if order is same or not, the string outputs of those bags will be the same on "in-order" traversal. Both string outputs are ordered from the least element to greatest element.

```

83
84 //Method that returns true if the given Bag equals to this Bag.
85 @Override
86 public boolean equals(Object obj){
87     if (obj instanceof Bag) {
88         if (this.toString().equals(obj.toString())) {
89             return true;
90         }
91     }
92     return false;
93 }
94

```

Image 15: A screenshot of the equals() method.

- Next, it's going to be examined the elementSize() method now. This method takes an argument named as item has the generic type and the method has the return type of int. This method basically, searches for an item, if a node exists in bag that holds the given item, then returns it's element size by invoking the getElementSize() method which is defined in the Node class. If there is no such given element in bag, then it returns 0.
- In order to do this, this method references to another method which is named as search(). Detailed explanation of the search() method can be found in the further paragraphs.

```

94 //Method that returns the frequency of the given data.
95 public int elementSize(T item){
96     Node<T> current = search(item);
97     if (current == null)
98         return 0;
99     else
100         return current.getElementSize();
101 }
102

```

Image 16: A screenshot of the elementSize() method.

- Next, it's going to be examined the isEmpty() method now. This method takes no arguments and the method has the return type of boolean. Simply, checks if the root is null, returns true. Else, returns false.

```

103 //Method that returns true if the Bag is empty.
104 public boolean isEmpty(){
105     if (root == null){
106         return true;
107     }
108     return false;
109 }
110

```

Image 17: A screenshot of the isEmpty() method.

- Next, it's going to be examined the remove() method now. There are two remove methods in this class. It's applied a method overloading for encapsulating. First one takes an argument named as item and has the generic type, and the return type of first method is boolean. Simply, check if the node exists in the bag, that holds the given item, if bag does not contain that, then returns false with an error message. Else if bag contains a corresponding node, then it invokes the second remove method which actually operates the recursive removing operations, and then links the root node to returned node by the invoking second remove() method. After that, checks if the size of bag after removing and comparing it with the size of bag before removing. If new size is less than before the removing, then method returns true, else, it returns false.

```

110
111 //Method that removes the given item from the Bag.
112 public boolean remove(T item){
113     if (this.contains(item) == false){
114         System.out.println("Item could not found.");
115         return false;
116     }
117     int beforeRemoving = this.size();
118     root = remove(root, item);
119     if (this.size() < beforeRemoving){
120         return true;
121     }
122     return false;
123 }

```

Image 18: A screenshot of the first remove() method.

- Second remove() method, operates the removing instructions recursively until it find the corresponding node that holds the given item. There are two base cases:
- If the corresponding node's element size is greater than 1, it's only decreasing it's element size by one. And returns.
- Else if the corresponding node's element size is equals to 1, than it's changing it's data and element size by the taking maximum element of left-subtree or the minimum element of the right-subtree. If corresponding node has both left and right childs, then it checks the maximum depth of it's left sub-tree and right sub-tree. So replacing the node from the sub-tree which has the greater depth than the other one, in order to obtain a shorter, more balanced tree structure at the end. In order to do this, it references to methods their names such as : leftMost(), rightMost() and maxDepth(). Detailed explanations of those methods can be found at further paragraphs.

```

127 //Method that removes the given item from the Bag.
128 private Node<T> remove(Node<T> current, T item){
129     if (current == null){
130         return null;
131     }
132     if (current.getData().compareTo(item) > 0){
133         current.setLeft(remove(current.getLeft(), item));
134     }
135     else if (current.getData().compareTo(item) < 0){
136         current.setRight(remove(current.getRight(), item));
137     }
138     else if (current.getData().compareTo(item) == 0){
139         if (current.getElementSize() > 1){
140             current.decreaseElementSize();
141             return current;
142         }
143         else {
144             if (current.getLeft() == null && current.getRight() == null){
145                 if (current == root){
146                     clear();
147                 }
148                 return null;
149             }
150             else if (current.getLeft() == null){
151                 current = current.getRight();
152             }
153             else if (current.getRight() == null){
154                 current = current.getLeft();
155             }
156             else {
157                 if (maxDepth(current.getRight()) > maxDepth(current.getLeft())){
158                     Node<T> leftMostOfRightOfCurrent = leftMost(current.getRight());
159                     current.setData(leftMostOfRightOfCurrent.getData());
160                     current.setElementSize(leftMostOfRightOfCurrent.getElementSize());
161                     leftMostOfRightOfCurrent.setElementSize(1);
162                     current.setRight(remove(current.getRight(), leftMostOfRightOfCurrent.getData()));
163                 }
164             }
165         }
166     }
167     else if (current.getData().compareTo(item) == 0){
168         if (current.getElementSize() > 1){
169             current.decreaseElementSize();
170             return current;
171         }
172         else {
173             if (current.getLeft() == null && current.getRight() == null){
174                 if (current == root){
175                     clear();
176                 }
177                 return null;
178             }
179             else if (current.getLeft() == null){
180                 current = current.getRight();
181             }
182             else if (current.getRight() == null){
183                 current = current.getLeft();
184             }
185             else {
186                 if (maxDepth(current.getRight()) > maxDepth(current.getLeft())){
187                     Node<T> leftMostOfRightOfCurrent = leftMost(current.getRight());
188                     current.setData(leftMostOfRightOfCurrent.getData());
189                     current.setElementSize(leftMostOfRightOfCurrent.getElementSize());
190                     leftMostOfRightOfCurrent.setElementSize(1);
191                     current.setRight(remove(current.getRight(), leftMostOfRightOfCurrent.getData()));
192                 }
193                 else {
194                     Node<T> rightMostOfLeftOfCurrent = rightMost(current.getLeft());
195                     current.setData(rightMostOfLeftOfCurrent.getData());
196                     current.setElementSize(rightMostOfLeftOfCurrent.getElementSize());
197                     rightMostOfLeftOfCurrent.setElementSize(1);
198                     current.setLeft(remove(current.getLeft(), rightMostOfLeftOfCurrent.getData()));
199                 }
200             }
201         }
202     }
203     return current;
204 }

```

Image 19: Screenshots of the second remove() method.

- Next, it's going to be examined the size() method now. There are 2 size() methods in Bag class. First one takes no argument and it has the return type of int. It only, invokes the second size() method which actually operates the recursive traversal operations. So there is a method overloading for encapsulation purposes.

```

176 //Method that returns the total element number of the Bag.
177 public int size(){
178     return size(root);
179 }
180

```

Image 20: A screenshot of the first size() method.

- Second size method takes an argument named as current and has the type of Node, and the method has the return type of int. Similarly to distinctSize() method, it makes a complete in-order traversal on the bag, and counts “element sizes” of the nodes. After the traversal completed, it returns the counter variable.

```

182 //Method that returns the total element number of the Bag.
183 private int size(Node <T> current) {
184     int i = 0;
185
186     if (current == null){
187         return 0;
188     }
189     else {
190         i += size(current.getLeft());
191         i += current.getElementSize();
192         i += size(current.getRight());
193     }
194     return i;
195 }

```

Image 21: A screenshot of the second size() method.

- Next, it's going to be examined of the toString() method now. Since every object inherits the toString() method from the Object class, it should be overridden that the toString method in user-made classes to print them properly. So there are 2 toString() methods in this class.
- First one checks if the bag is empty, if it is empty, then returns the string that “This is an empty Bag.” Else, it only invokes the second toString() method by giving root as its argument. So, there is a method overloading also for encapsulation purposes. Actual traversal operations handled by the second toString() method.

```

195
196 //Method that returns a string representation of the Bag.
197 @Override
198 public String toString(){
199     if (this.isEmpty())
200         return "This is an empty Bag.";
201     return toString(root);
202 }

```

Image 22: A screenshot of the first toString() method.

- Second method takes an argument named as current and its type is Node. And the method has the return type of String. First, this method defines a variable s has the type of String. s is initialized to empty string. Until there are no more nodes to travel, it recursively makes a complete in-order traversal on the Bag and concatenates the s with the current node's data and element size. After the traversal completed, it returns the s.

```

204 //Method that returns a string representation of the Bag.
205 private String toString(Node<T> current){
206     String s = "";
207
208     if (current == null){
209         return "";
210     }
211     else {
212         s += toString(current.getLeft());
213         s += "[" + current.getData() + " x " + current.getElementSize() + " ";
214         s += toString(current.getRight());
215     }
216     return s;
217 }
218

```

Image 23: A screenshot of the second toString() method.

- Required methods and all the features expected in the assignment has explained so far. Along these paragraph, it will be explained the extra methods those exists in the Bag class. Those methods are implemented in order to speed up the workflow, and avoid the spaghetti coding.
- Now, it's going to be examined the leftMost() method. This takes an argument named as current, and it's type is Node. And the method has the return type of Node. This, simply, invokes itself recursively, until it reach to the left most node of the given node. After the recursion completed, it returns the leftmost node. This is used in remove() method to find minimum node of the right-sub trees.

```

219
220 //User methods:
221 //Method that returns the left-most element after the given node.
222 private Node<T> leftMost(Node<T> current){
223     if (current.getLeft() != null){
224         return leftMost(current.getLeft());
225     }
226     return current;
227 }

```

Image 24: A screenshot of the leftMost() method.

- Now it's going to be examined the rightMost() method. This method has the same logic with the leftMost() method unlikely, it returns the rightmost node. This is also used in remove() method to find maximum node of the left-sub trees.

```

229 //Method that returns the right-most element after the given node.
230 private Node<T> rightMost(Node<T> current){
231     if (current.getRight() != null){
232         return rightMost(current.getRight());
233     }
234     return current;
235 }

```

Image 25: A screenshot of the rightMost() method.

- Now it's going to be examined the maxDepth() method. This method takes an argument named as current and it's type is Node. The method has the return type of int. Basicly, this method recursively invokes itself and calculates the right and left sub tree depths of the given node until the current reaches to null. After the recursion completed, it checks if the left or right depth is greater, and returns the greater one. This method is used for remove operations in remove() method.

```

237 //Method that returns the maximum depth of the given node.
238 private int maxDepth(Node<T> current){
239     if (current == null){
240         return 0;
241     }
242     else {
243         int leftDepth = maxDepth(current.getLeft());
244         int rightDepth = maxDepth(current.getRight());
245
246         if (leftDepth > rightDepth){
247             return (leftDepth + 1);
248         }
249         else {
250             return (rightDepth + 1);
251         }
252     }
253 }

```

Image 26: A screenshot of the maxDepth()

- Finally the last method, search() is examined along this section. There are 2 search methods in this class for encapsulation purposes. First one takes an argument named as item and it's type is generic. And the first method's return type is Node. It only invokes the second search() method by the starting node argument as root and returns the node that came by this invoke. Actual traversal operations are made by second search() method.

```

255 //Method that returns the node if it has the given data.
256 private Node<T> search(T item) {
257     return search(root, item);
258 }

```

Image 27: A screenshot of the first search() method.

- The second search() method takes an argument named as current and it's type is Node. The method has the return type of Node. search() method has the same logic with the contains() method. Basically, it recursively makes a complete in-order traversal from the starting point as the given (current) node until the corresponding node that contains the given item in the bag or until there are no more nodes to travel. If a corresponding node exists, it returns the node. Else, it returns null. This method used in elementSize() method in order to find the given data.

```

260 //Method that returns the node if it has the given data.
261 private Node<T> search(Node<T> current, T item) {
262     if (current == null)
263         return null;
264     else {
265         if (current.getData().compareTo(item) == 0) {
266             return current;
267         }
268         else if (current.getData().compareTo(item) > 0) {
269             return search(current.getLeft(), item);
270         }
271         else if (current.getData().compareTo(item) < 0) {
272             return search(current.getRight(), item);
273         }
274     }
275     return current;
276 }
277
278 }

```

Image 28: A screenshot of the second search() method.

4. Car Class :

- Car class is only for showing that the Bag adt can work with the any type of user-written objects. There are two fields for a car : String name and int price. And car class overrides the compareTo method in order to sort Car objects. First, it checks the prices. If prices are different, then sorts them by their prices. Else if prices are same, then sorts them by their names (Not case sensitive.).


```

1// NAME : MUSTAFA CEM ONAN
2// ID   : 180315064
3
4public class Car implements Comparable<Car>{
5    private String name;
6    private int price;
7
8    public Car(String name, int price) {
9        this.name = name;
10       this.price = price;
11    }
12
13    public String getName() {
14        return name;
15    }
16
17    public int getPrice() {
18        return price;
19    }
20
21    @Override
22    public String toString() {
23        return "" + name + ", Price : " + price;
24    }
25
26    @Override
27    public int compareTo(Car anotherCar) {
28        if (this.price == anotherCar.getPrice()) {
29            return this.name.compareToIgnoreCase(anotherCar.getName());
30        }
31        return this.price - anotherCar.getPrice();
32    }
33 }

```

Image 29: A screenshot of the Car class.

5. Test Class :

A. Examination of This Class:

- Test class is created for the testing all 10 required methods, those are expected to work properly from the implementer in the assignment.
- This class contains the main method, and 4 of Bag objects. Rest, all the informations of the operations has done in this class are written as the comment lines.


```

1 // NAME : MUSTAFA CEM ONAN
2 // ID : 180315064
3
4 public class Test {
5
6     public static void main(String[] args) {
7         Bag<Integer> bag1 = new Bag<>();
8         Bag<Integer> bag2 = new Bag<>();
9         Bag<Integer> bag3 = new Bag<>();
10
11         System.out.println("bag1: " + bag1);
12
13         System.out.println("Adding some data to the bag1");
14
15         //Testing out the add() method
16         bag1.add(10);
17         bag1.add(3);
18         bag1.add(12);
19         bag1.add(12);
20         bag1.add(5);
21         bag1.add(10);
22         bag1.add(10);
23         bag1.add(100);
24
25         //Testing toString() method
26         System.out.println("bag1: " + bag1);
27         //Testing size() method
28         System.out.println("Size of the bag1 = " + bag1.size());
29         //Testing distinctSize() method
30         System.out.println("Distinct Size of the bag1 = " + bag1.distinctSize());
31
32         System.out.println();
33         System.out.println("_____");
34         System.out.println();

```

```

34         System.out.println();
35
36         //Testing clear() method
37         System.out.println("bag1 will now be cleared.");
38         bag1.clear();
39
40         System.out.println("bag1: " + bag1);
41
42         System.out.println();
43         System.out.println("_____");
44         System.out.println();
45
46         System.out.println("Equality test : adding [0-9] digits to bag1 in ordered, adding same digits to bag2");
47         System.out.println("bag1 must be equal to bag2");
48         //Adding [0-9] digits to the bag1
49         for (int i = 9; i >= 0; i--){
50             bag1.add(i);
51         }
52
53         //Adding [0-9] digits IN REVERSE ORDER to the bag2
54         for (int i = 0; i <= 9; i++){
55             bag2.add(i);
56         }
57
58         System.out.println("bag1: " + bag1);
59         System.out.println("bag2: " + bag2);
60
61         //Testing equals() method (Should be equal)
62         if (bag1.equals(bag2))
63             System.out.println("(YES) bag1 equals to the bag2.");
64         else
65             System.out.println("(NO) bag1 IS NOT equal to the bag2.");
66
67         System.out.println();

```

```

67     System.out.println();
68     System.out.println("_____");
69     System.out.println();
70
71     System.out.println();
72     System.out.println("Now adding an another 0 to the bag1 and testing equality one more time.");
73     System.out.println("bag1 must NOT be equal to bag2");
74     //Adding an another 0 to the bag1 and testing equals() one more time
75     bag1.add(0);
76
77     System.out.println("bag1: " + bag1);
78     System.out.println("bag2: " + bag2);
79
80     //Testing equals() method (Should NOT be equal)
81     if (bag1.equals(bag2))
82         System.out.println("(YES) bag1 equals to the bag2.");
83     else
84         System.out.println("(NO) bag1 IS NOT equal to the bag2.");
85
86
87     System.out.println();
88     System.out.println("_____");
89     System.out.println();
90
91     System.out.println("Element size, isEmpty() test :");
92     //Reset bag1 and bag2.
93     bag1.clear();
94     bag2.clear();
95
96     //Add 5 times 10 to the bag1.
97     for (int i = 0; i < 5; i++)
98         bag1.add(10);
99
100    //Print bag1.
101
102    //Print bag1.
103    System.out.println("bag1: " + bag1);
104
105    //Testing elementSize() method.
106    System.out.println("Element size of the 10 is: " + bag1.elementSize(10));
107
108    //Testing isEmpty()
109    if (bag1.isEmpty())
110        System.out.println("bag1 is empty");
111    else
112        System.out.println("bag1 IS NOT empty");
113
114    //Clearing bag1
115    bag1.clear();
116    System.out.println("bag1 is now cleared");
117    System.out.println("bag1 : " + bag1);
118
119    //Testing isEmpty() again
120    if (bag1.isEmpty())
121        System.out.println("bag1 is empty");
122    else
123        System.out.println("bag1 IS NOT empty");
124
125    System.out.println();
126    System.out.println("_____");
127    System.out.println();
128
129    System.out.println("Remove test:");
130    //Testing remove() function
131    bag1.clear();
132    bag2.clear();
133    bag3.clear();
134
135    bag1.add(10);

```

```
133     bag1.add(10);
134     bag1.add(10);
135     bag1.add(5);
136     bag1.add(5);
137     bag1.add(5);
138     bag1.add(15);
139     bag1.add(15);
140     bag1.add(15);
141
142     System.out.println(bag1);
143     System.out.println("Size : " + bag1.size());
144     System.out.println("Distinct size : " + bag1.distinctSize());
145
146     System.out.println("Removing 5");
147     bag1.remove(5);
148
149     System.out.println(bag1);
150     System.out.println("Size : " + bag1.size());
151     System.out.println("Distinct size : " + bag1.distinctSize());
152
153     System.out.println("Removing 10");
154     bag1.remove(10);
155
156     System.out.println(bag1);
157     System.out.println("Size : " + bag1.size());
158     System.out.println("Distinct size : " + bag1.distinctSize());
159
160     System.out.println("Removing 10 again");
161     bag1.remove(10);
162
163     System.out.println(bag1);
164     System.out.println("Size : " + bag1.size());
165     System.out.println("Distinct size : " + bag1.distinctSize());
166
```

```
166
167     System.out.println("Removing 5");
168     bag1.remove(5);
169
170     System.out.println(bag1);
171     System.out.println("Size : " + bag1.size());
172     System.out.println("Distinct size : " + bag1.distinctSize());
173
174     System.out.println("Removing 15");
175     bag1.remove(15);
176
177     System.out.println(bag1);
178     System.out.println("Size : " + bag1.size());
179     System.out.println("Distinct size : " + bag1.distinctSize());
180
181     System.out.println("Removing 15 again");
182     bag1.remove(15);
183
184     System.out.println(bag1);
185     System.out.println("Size : " + bag1.size());
186     System.out.println("Distinct size : " + bag1.distinctSize());
187
188     System.out.println("Removing 15 again");
189     bag1.remove(15);
190
191     System.out.println(bag1);
192     System.out.println("Size : " + bag1.size());
193     System.out.println("Distinct size : " + bag1.distinctSize());
194
195     System.out.println("Removing 5");
196     bag1.remove(5);
197
198     System.out.println(bag1);
199     System.out.println("Size : " + bag1.size());
```



```

199 System.out.println("Size : " + bag1.size());
200 System.out.println("Distinct size : " + bag1.distinctSize());
201
202 System.out.println("This project is implemented by :");
203 System.out.println("Name : Mustafa Cem ONAN");
204 System.out.println("ID : 180315064");
205
206 System.out.println();
207 System.out.println("_____");
208 System.out.println();
209
210 System.out.println("Any Type Test:");
211 System.out.println("Adding Car objects to a bag.");
212
213 //Car object has already overridden compareTo method it's inside. And it's sorting
214 //car objects to their prices. If the prices of 2 car is equal, then sorts alphabetically.
215
216 Car car1 = new Car("astra", 30000);
217 Car car2 = new Car("vectra", 40000);
218 Car car3 = new Car("passat", 300000);
219 Car car4 = new Car("e220d", 550000);
220 Car car5 = new Car("sahin", 15000);
221 Car car6 = new Car("corsa", 55000);
222 Car car7 = new Car("corsa", 40000);
223 Car car8 = new Car("520d", 300000);
224 Car car9 = new Car("dogan", 15000);
225 Car car10 = new Car("w140", 300000);
226
227 Bag<Car> carBag = new Bag<Car>();
228
229 carBag.add(car1);
230 carBag.add(car2);
231 carBag.add(car3);
232 carBag.add(car4);
233
234
235
236
237
238
239
240
241
242 }
243 }
244
245
246
247
248

```

Image 30: Screenshots of Test class.

UML USE CASE DIAGRAM

