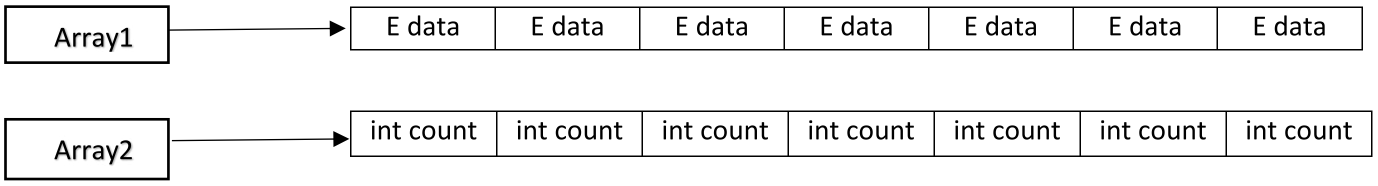
CSE 2105 – DATA STRUCTURES
2016 – 2017 FALL SEMESTER PROJECT REPORT
THE BAG ADT

Before explanation of a code, I want to discuss about which data structure is a proper one to implement the BAG ADT. There are many way to build a BAG ADT. In this report, you can find more than one solution to implement BAG ADT, but the question is which solution is more efficient?

SOLUTION PROPOSALS

1) **With Array**

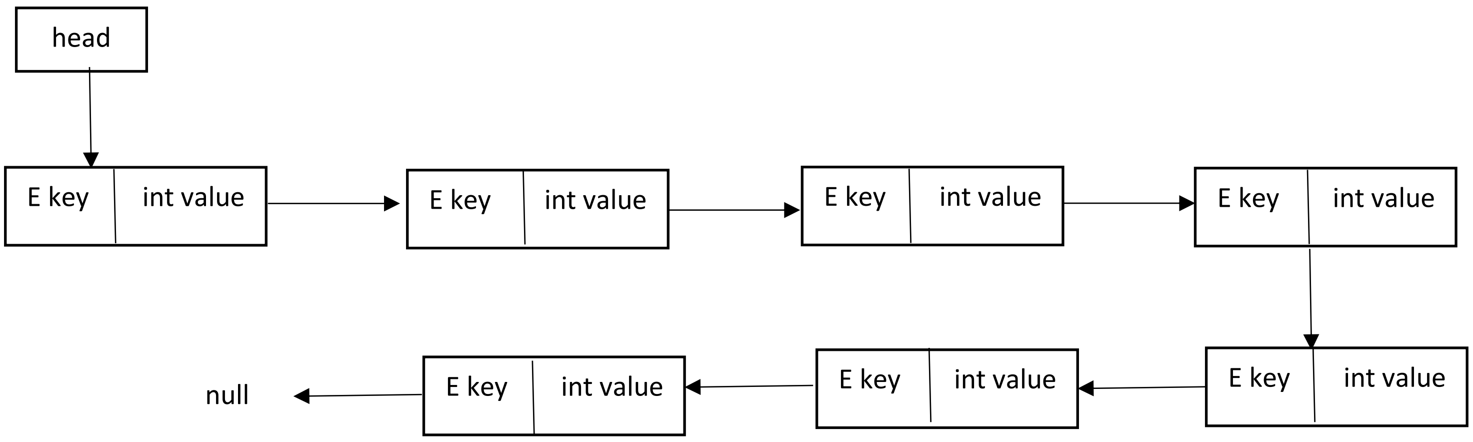


I think we can implement the BAG by using 2 arrays. First array (Array1) can hold the data such as string, integer, double or float. Because data type 'E' is type parameter which is an identifier that specifies a generic type name and it allows us to write a single method that can handle with string, integer or any type objects. Second array (Array2) can hold the counter for the data which is located the same index in Array1. We can reach out the data and its value by using same index.

Why I did not use this structure?

- The size of the arrays is fixed: We do not know the upper limit on the number of elements in advance.
- Inserting and deletion a new element in an array of elements is expensive: We have to Delete the Element from given Location and then Shift All successive elements up by 1 position

2) **With Linked List**



I think we can implement the BAG by using linked list. We can hold the key and its value in the same Node.

<u>Why I used this structure?</u>

- Dynamic size
- Ease of insertion/deletion: we have to Just Change the Pointer address field (Pointer), So Insertion and Deletion Operations are quite easy to implement.

Examining my first project: BAG ADT with Linked List

Class Node<E>

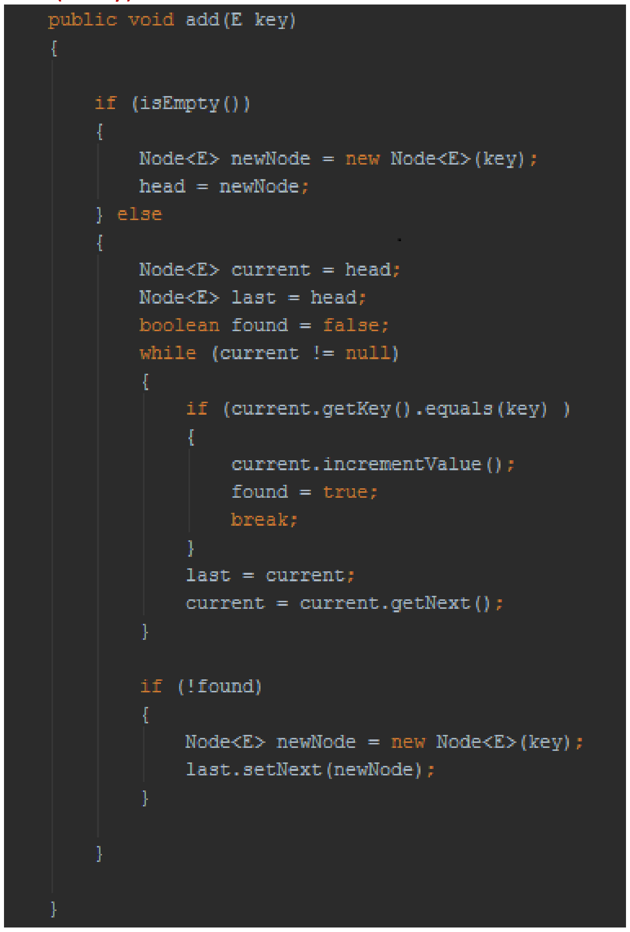
I created a Node<E> class which has a 3 instances variables **E key, int value, Node<E> next**. I wrote a setter and getter methods of instances variables to reach them and also, 2 more method to increase and decrease the value of the key, which is amount of how many element there are in the BAG.I checked the 'value' to make a decision for should element be deleted from BAG or not in remove method.

Class BagADT<E>

I created **Node<E> head** to refer to the first node of the list.

<u>Method explanation</u>

add (E key):



```
public void add(E key)
{

    if (isEmpty())
    {
        Node<E> newNode = new Node<E>(key);
        head = newNode;
    } else
    {
        Node<E> current = head;
        Node<E> last = head;
        boolean found = false;
        while (current != null)
        {
            if (current.getKey().equals(key) )
            {
                current.incrementValue();
                found = true;
                break;
            }
            last = current;
            current = current.getNext();
        }

        if (!found)
        {
            Node<E> newNode = new Node<E>(key);
            last.setNext(newNode);
        }

    }

}
```

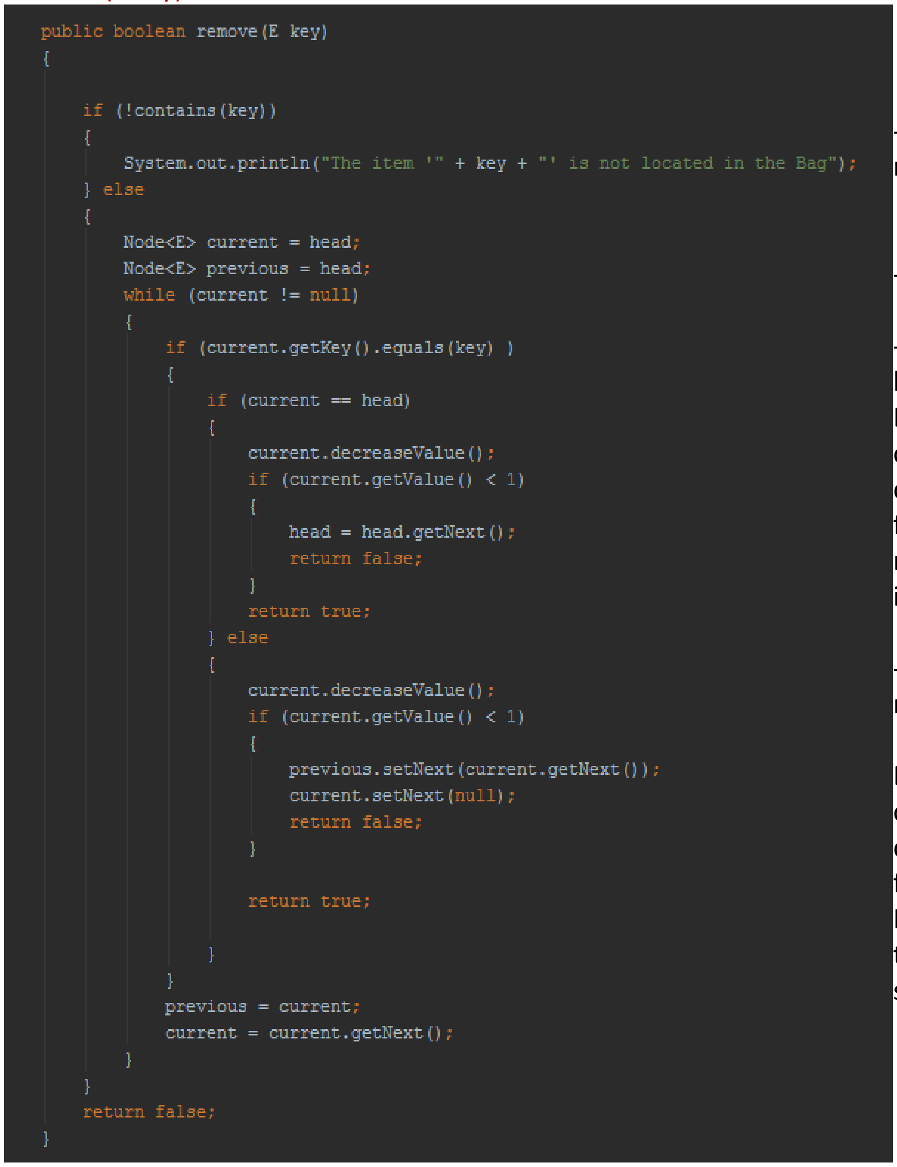
->if the bag is empty, create a new node and assign that is head.

->loop to traverse through Bag.

->comparison, does new element exist in the Bag? If yes, increase its value, assign true to found and break the loop.

If no, create a new node which has the new element and add this node to Bag.

remove (E key):



```
public boolean remove(E key)
{

    if (!contains(key))
    {
        System.out.println("The item '" + key + "' is not located in the Bag");
    } else
    {
        Node<E> current = head;
        Node<E> previous = head;
        while (current != null)
        {
            if (current.getKey().equals(key) )
            {
                if (current == head)
                {
                    current.decreaseValue();
                    if (current.getValue() < 1)
                    {
                        head = head.getNext();
                        return false;
                    }
                    return true;
                } else
                {
                    current.decreaseValue();
                    if (current.getValue() < 1)
                    {
                        previous.setNext(current.getNext());
                        current.setNext(null);
                        return false;
                    }

                    return true;

                }
            }
            previous = current;
            current = current.getNext();
        }
    }
    return false;
}
```

->if the item we try to remove is not located in the bag.

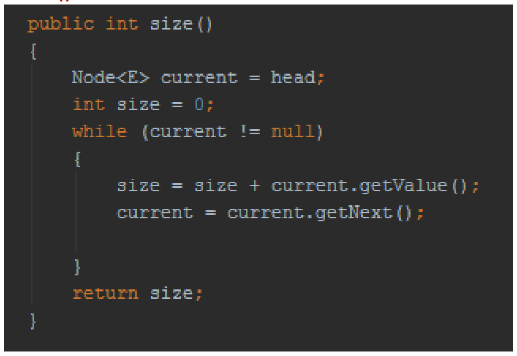
->loop to traverse through Bag.

->if the item we try to remove is head
Decrease the value of item and check if its value less than 1, then delete it. If it is deleted, return false. If just its value decreased, return true which means that the item still in the bag

->if the item we try to remove is not head

Decrease the value of item and check if its value less than 1, then delete it. If it is deleted, return false.
If just its value decreased, return true which means that the item still in the bag

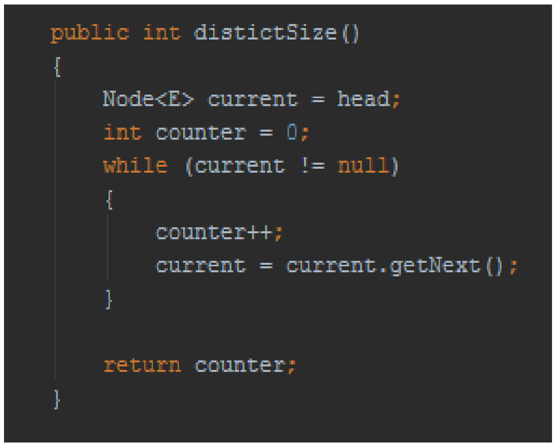
size ():



```
public int size()
{
    Node<E> current = head;
    int size = 0;
    while (current != null)
    {
        size = size + current.getValue();
        current = current.getNext();

    }
    return size;
}
```

->loop to traverse through Bag.
->for every node, sum its value. Because 'value' holds the data which refers to how many instances of element is located in the Bag.

->return the total of instances are located in the Bag distictSize ():

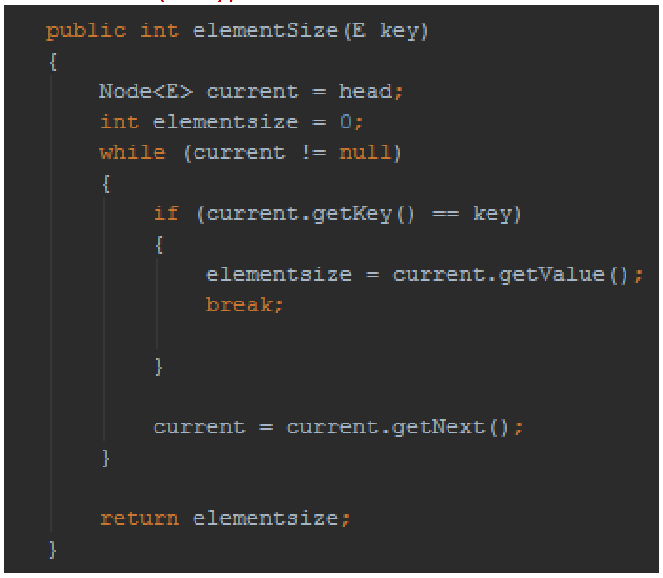


```java
public int distictSize()
{
    Node<E> current = head;
    int counter = 0;
    while (current != null)
    {
        counter++;
        current = current.getNext();
    }

    return counter;
}
```

->loop to traverse through Bag.

->count the number of node in Bag. Because every node is distinct.

elementSize (E key):



```java
public int elementSize(E key)
{
    Node<E> current = head;
    int elementsize = 0;
    while (current != null)
    {
        if (current.getKey() == key)
        {
            elementsize = current.getValue();
            break;

        }

        current = current.getNext();
    }

    return elementsize;
}
```

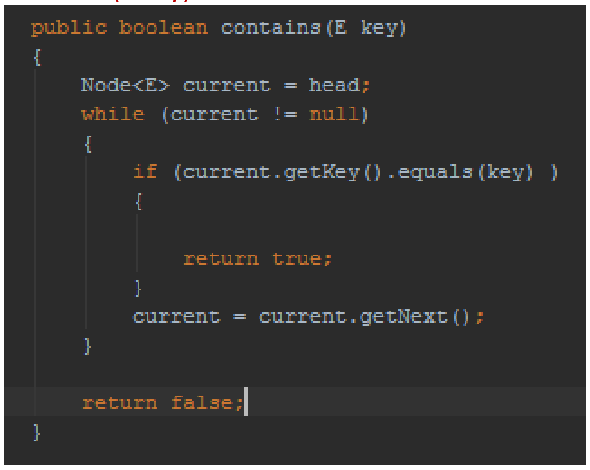
->loop to traverse through Bag.

->the item is found

->assign the value to variable elementsize and break the loop.

->return the elementsize.

contains (E key):



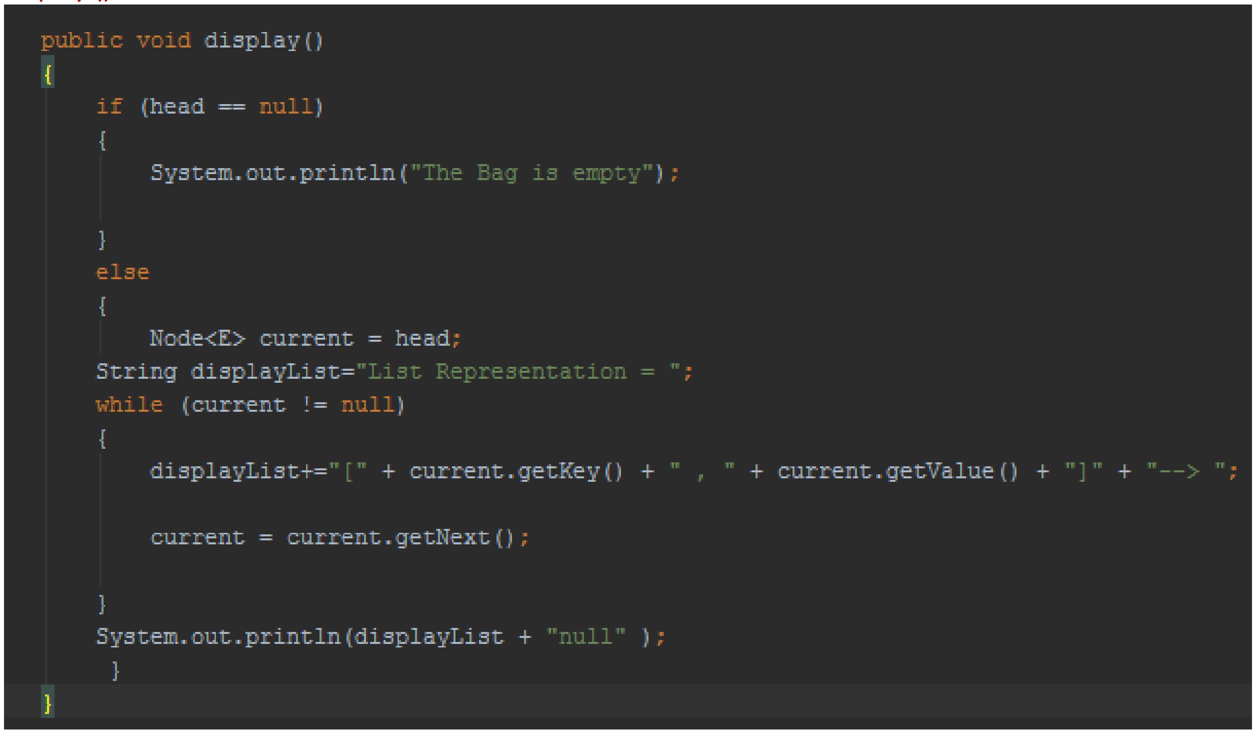
```java
public boolean contains(E key)
{
    Node<E> current = head;
    while (current != null)
    {
        if (current.getKey().equals(key) )
        {

            return true;
        }
        current = current.getNext();
    }

    return false;
}
```

->loop to traverse through Bag.

->If the item is found, return true.

-> If the item is not found, return false.

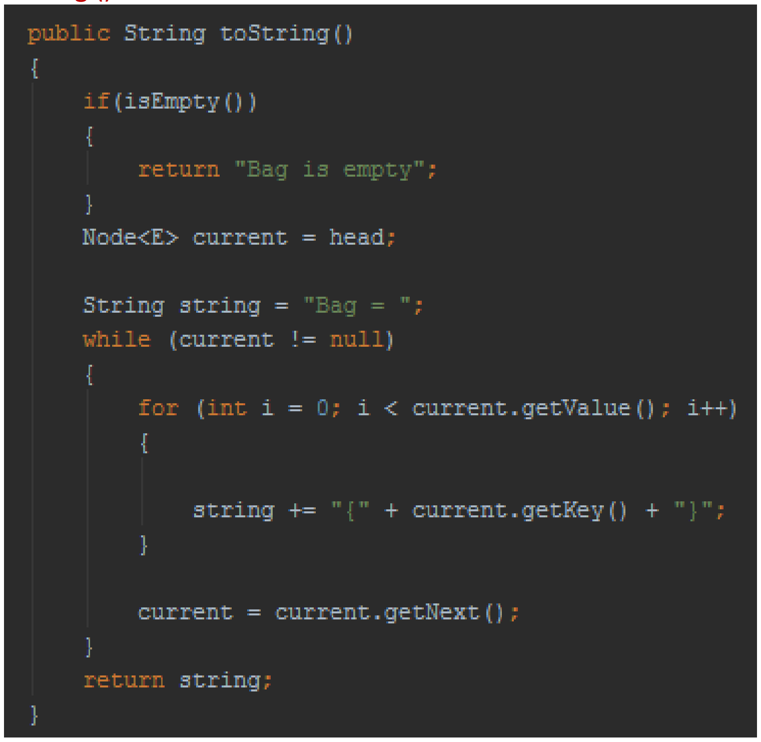
display ():



```
public void display()
{
    if (head == null)
    {
        System.out.println("The Bag is empty");

    }
    else
    {
        Node<E> current = head;
    String displayList="List Representation = ";
    while (current != null)
    {
        displayList+="[" + current.getKey() + " , " + current.getValue() + "]" + "--> ";

        current = current.getNext();

    }
    System.out.println(displayList + "null" );
     }
}
```

I wrote display method to show how the nodes are linked as a linked list.

toString ():

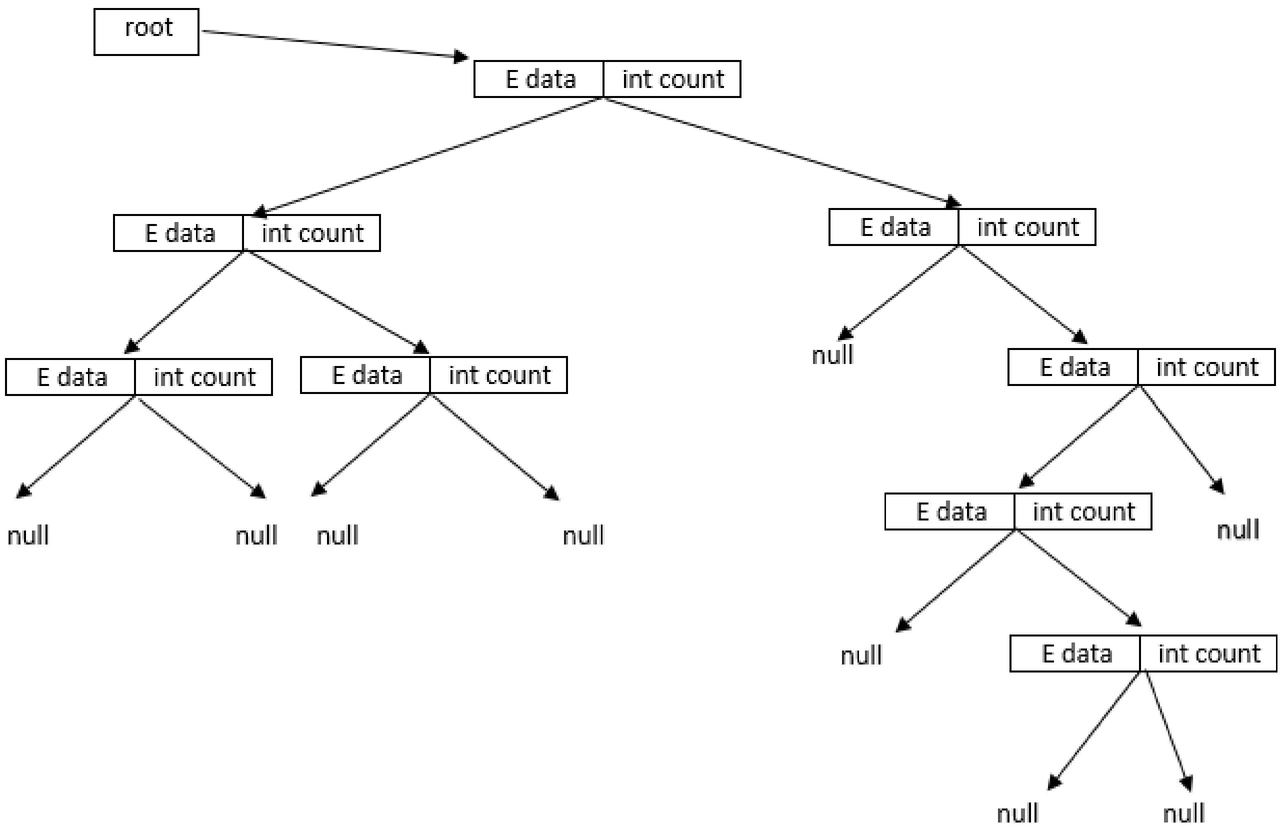


```
public String toString()
{
    if(isEmpty())
    {
        return "Bag is empty";
    }
    Node<E> current = head;

    String string = "Bag = ";
    while (current != null)
    {
        for (int i = 0; i < current.getValue(); i++)
        {

            string += "{" + current.getKey() + "}";
        }

        current = current.getNext();
    }
    return string;
}
```

->create a new string.

->loop executes as many as instances of the same item.

3) **With Binary Search Tree**



I think we can implement the BAG by using binary search tree. We can hold the data and its count in the same Node.

<u>Why I used this structure?</u>

- Dynamic size
- Searching in a binary tree becomes faster. It allows to make add and remove method faster.

Examining my second project: BAG ADT with Binary Search Tree

Class Node<E>

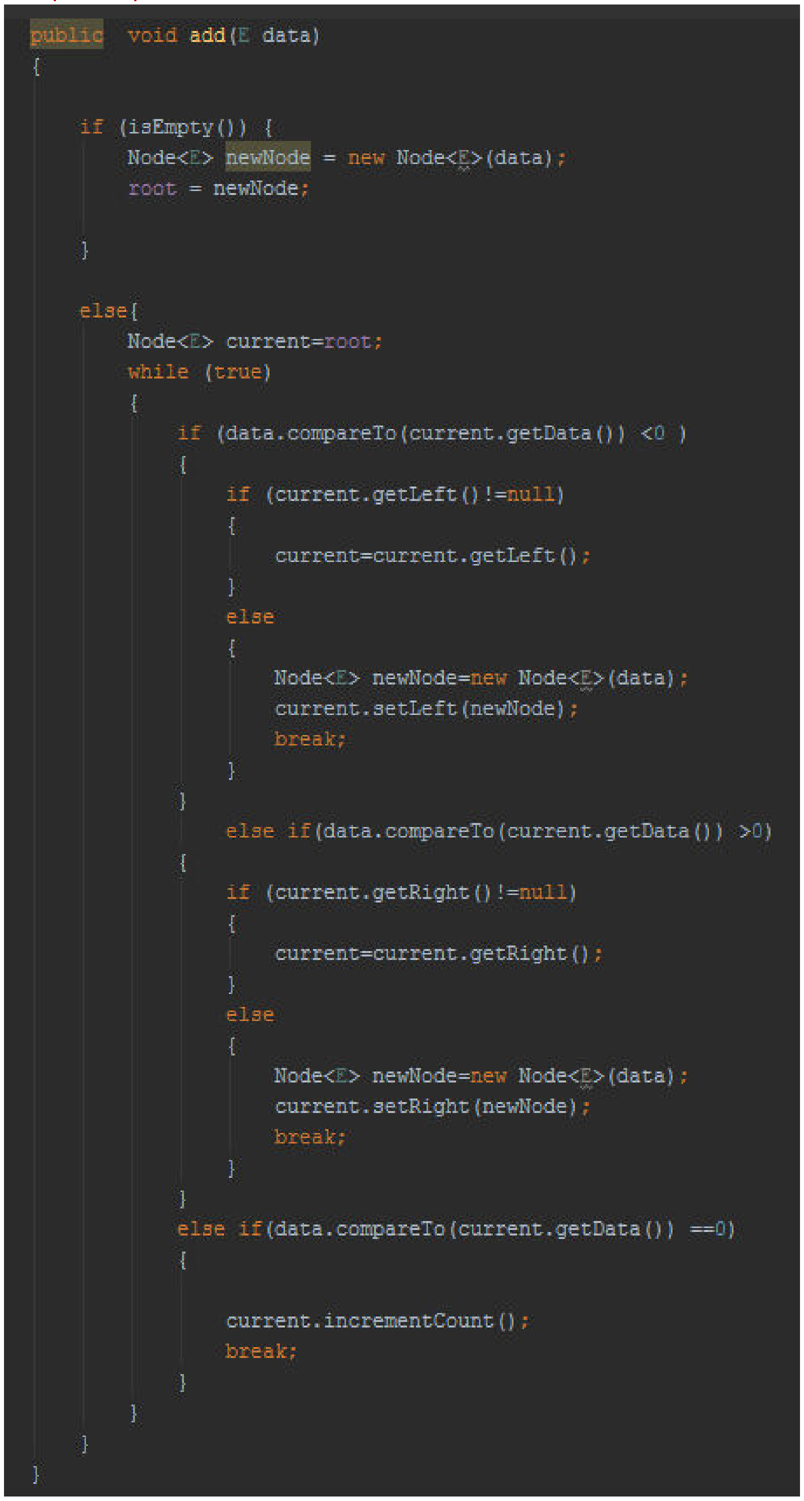
I created a Node<E> class which has a 4 instances variables **E data, int count, Node<E> left, Node<E> right**. I wrote a setter and getter methods of instances variables to reach them and also, 2 more method to increase and decrease the count of the data, which is amount of how many element there is in the BAG.I checked the 'count' to make a decision for should element be deleted from BAG or not in remove method. 2 more method to control that does node has a left or left child in the tree.

Class Bag <E extends Comparable<E>>

I created **Node<E> root** to refer to the root of the tree. I had to **extend Comparable<E>** to compare two object while inserting, searching or deleting object from tree.

Method explanation

add(E data)



```java
public  void add(E data)
{

    if (isEmpty()) {
        Node<E> newNode = new Node<E>(data);
        root = newNode;

    }

    else{
        Node<E> current=root;
        while (true)
        {
            if (data.compareTo(current.getData()) <0 )
            {
                if (current.getLeft()!=null)
                {
                    current=current.getLeft();
                }
                else
                {
                    Node<E> newNode=new Node<E>(data);
                    current.setLeft(newNode);
                    break;
                }
            }
                else if(data.compareTo(current.getData()) >0)
            {
                if (current.getRight()!=null)
                {
                    current=current.getRight();
                }
                else
                {
                    Node<E> newNode=new Node<E>(data);
                    current.setRight(newNode);
                    break;
                }
            }
            else if(data.compareTo(current.getData()) ==0)
            {

                current.incrementCount();
                break;
            }
        }
    }
}
```

->if the bag is empty, create a new node and assign that is root.

->comparison, if new data is less than the data of current root node,
->if left child is not null, go left sub tree.

->if left child is null, create a new node and set it as a left child of current root and break the loop.

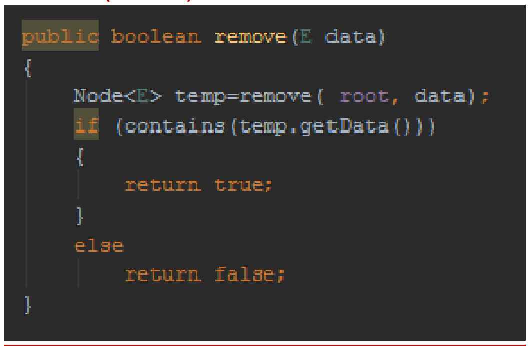
->comparison, if new data is greater than the data of current root node,

->if right child is not null, go right sub tree.

->if right child is null, create a new node and set it as a right child of current root and break the loop.

->if the item we try to add to tree is already exist in the tree, just increase the count and break the loop.

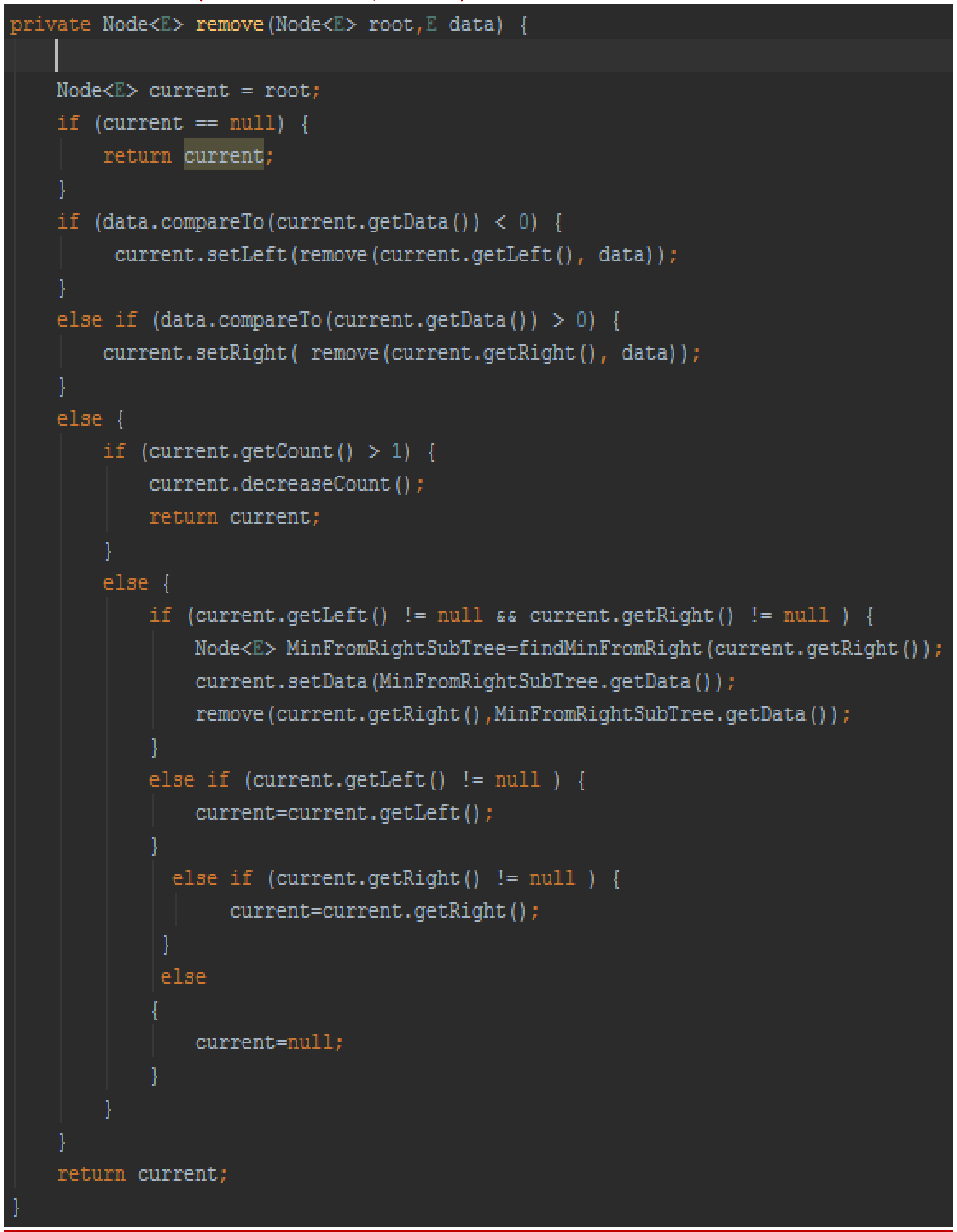
remove (E data)



```java
public boolean remove(E data)
{
    Node<E> temp=remove( root, data);
    if (contains(temp.getData()))
    {
        return true;
    }
    else
        return false;
}
```

->I needed a new remove method, Because I decided to use recursion in the method, so I used to remove method which takes 2 parameter one is data and other is root of tree. Other remove method returns the node which is deleted, and check the returned node. If it is still in the bag, return true, otherwise return false.

remove (Node<E> root , E data)



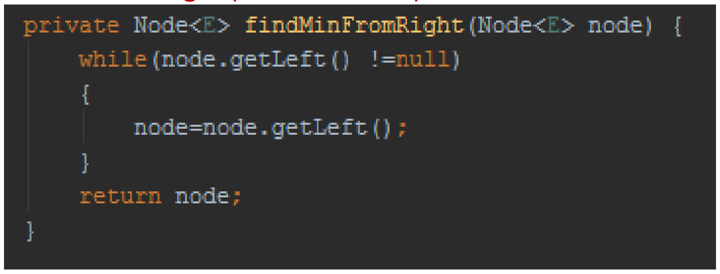
```java
private Node<E> remove(Node<E> root,E data) {

    Node<E> current = root;
    if (current == null) {
        return current;
    }
    if (data.compareTo(current.getData()) < 0) {
         current.setLeft(remove(current.getLeft(), data));
    }
    else if (data.compareTo(current.getData()) > 0) {
        current.setRight( remove(current.getRight(), data));
    }
    else {
        if (current.getCount() > 1) {
            current.decreaseCount();
            return current;
        }
        else {
            if (current.getLeft() != null && current.getRight() != null ) {
                Node<E> MinFromRightSubTree=findMinFromRight(current.getRight());
                current.setData(MinFromRightSubTree.getData());
                remove(current.getRight(),MinFromRightSubTree.getData());
            }
            else if (current.getLeft() != null ) {
                current=current.getLeft();
            }
             else if (current.getRight() != null ) {
                current=current.getRight();
             }
            else
            {
                current=null;
            }
        }
    }
    return current;
}
```

->comparison, if data is less than the data of current root node, recall the method by giving a left child as a parameter.
->comparison, if data is greater than the data of current root node, recall the method by giving a right child as a parameter.
->we found the correct node
->if the count of item is greater than 1 which means that if there are more than 1 instances of the same item, decrease the count of item.
->if the node to be deleted has left and right children, find the minimum item in the right sub tree and copy its value and recall method to delete minimum node from right sub tree.
-> if the node to be deleted has only left child.

-> if the node to be deleted has only right child.

-> if the node to be deleted does not have a child(Leaf node).

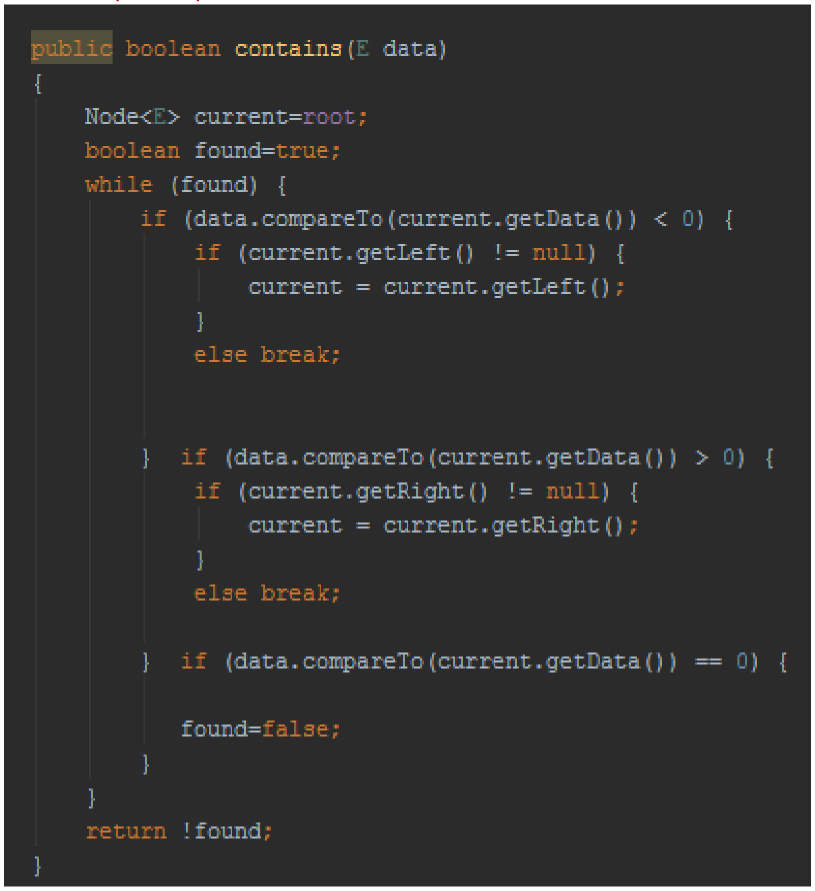
findMinFromRight (Node<E> node)



```java
private Node<E> findMinFromRight(Node<E> node) {
    while(node.getLeft() !=null)
    {
        node=node.getLeft();
    }
    return node;
}
```

->return the minimum item in the tree.

contains (E data)



```java
public boolean contains(E data)
{
    Node<E> current=root;
    boolean found=true;
    while (found) {
        if (data.compareTo(current.getData()) < 0) {
            if (current.getLeft() != null) {
                current = current.getLeft();
            }
            else break;


        }  if (data.compareTo(current.getData()) > 0) {
            if (current.getRight() != null) {
                current = current.getRight();
            }
            else break;

        }  if (data.compareTo(current.getData()) == 0) {

           found=false;
        }
    }
    return !found;
}
```

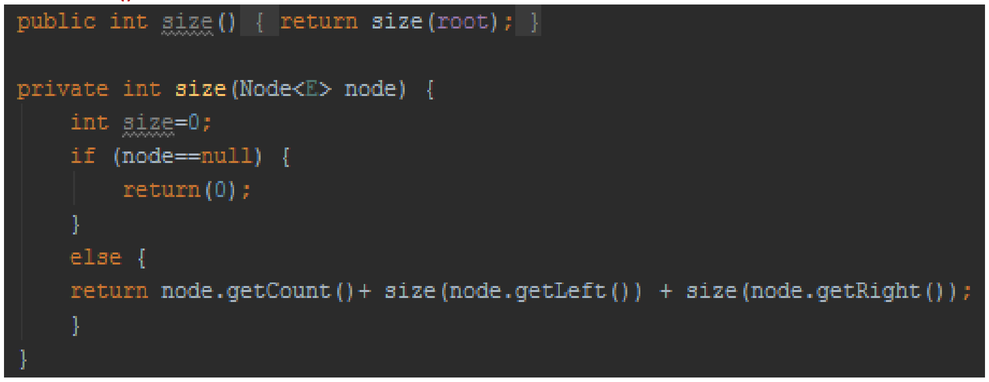
->comparison, if data is less than the data of current root node, go left.
->if it does not have left child break the loop.

->comparison, if data is greater than the data of current root node, go right.
->if it does not have right child break the loop.

->if data is equal to the data of current root node, assign false to found variable.

->return !found.

size ()

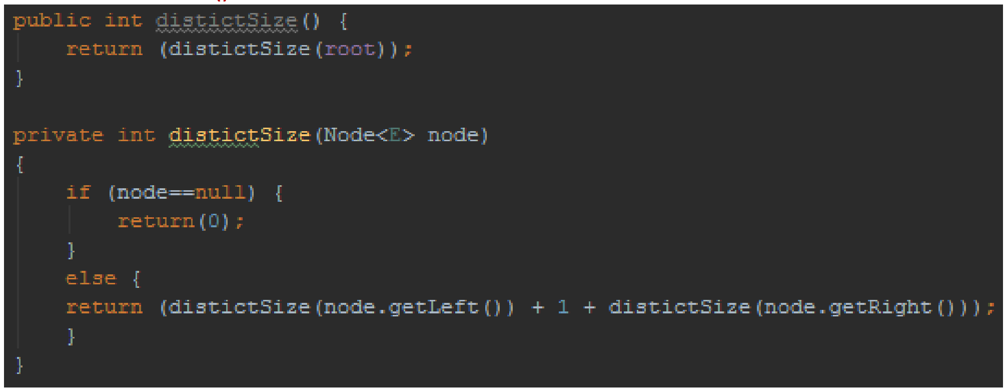


```java
public int size() { return size(root); }

private int size(Node<E> node) {
    int size=0;
    if (node==null) {
        return(0);
    }
    else {
    return node.getCount()+ size(node.getLeft()) + size(node.getRight());
    }
}
```

->I needed a new size method, Because I decided to use recursion in the method to traverse tree.

->sum the count of each item and return it.

distictsize ()

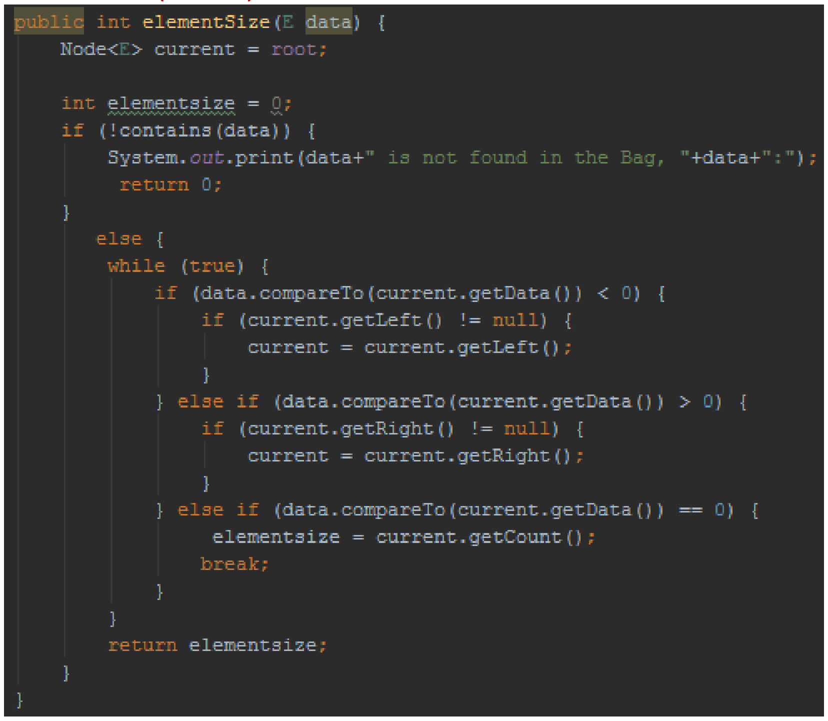


```java
public int distictSize() {
    return (distictSize(root));
}

private int distictSize(Node<E> node)
{
    if (node==null) {
        return(0);
    }
    else {
    return (distictSize(node.getLeft()) + 1 + distictSize(node.getRight()));
    }
}
```

->I needed a new distictSize method, Because I decided to use recursion in the method to traverse tree.

->sum the number of node in the tree and return it. Nodes are unique in the tree.

elementSize (E data)



```java
public int elementSize(E data) {
    Node<E> current = root;

    int elementsize = 0;
    if (!contains(data)) {
        System.out.print(data+" is not found in the Bag, "+data+":");
         return 0;
    }
       else {
        while (true) {
            if (data.compareTo(current.getData()) < 0) {
                if (current.getLeft() != null) {
                    current = current.getLeft();
                }
            } else if (data.compareTo(current.getData()) > 0) {
                if (current.getRight() != null) {
                    current = current.getRight();
                }
            } else if (data.compareTo(current.getData()) == 0) {
                 elementsize = current.getCount();
                break;
            }
        }
        return elementsize;
    }
}
```

->if the item is not located in Bag.

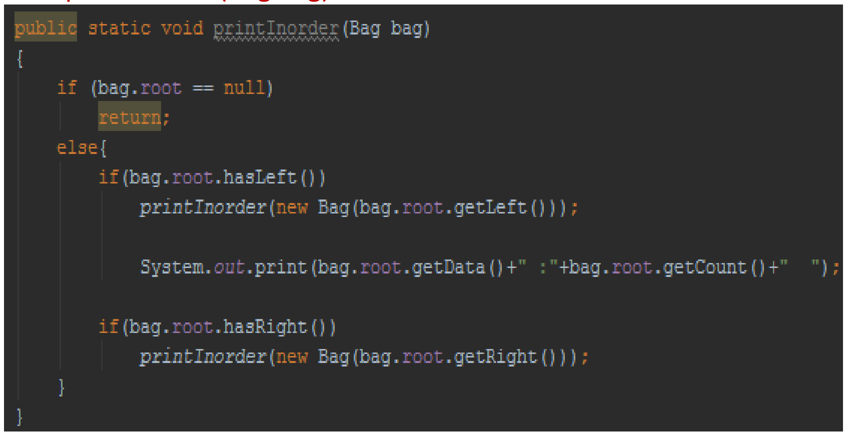
->if the item is located in Bag.

->comparison, if data is less than the data of current root node, go left.

->comparison, if data is greater than the data of current root node, go right.

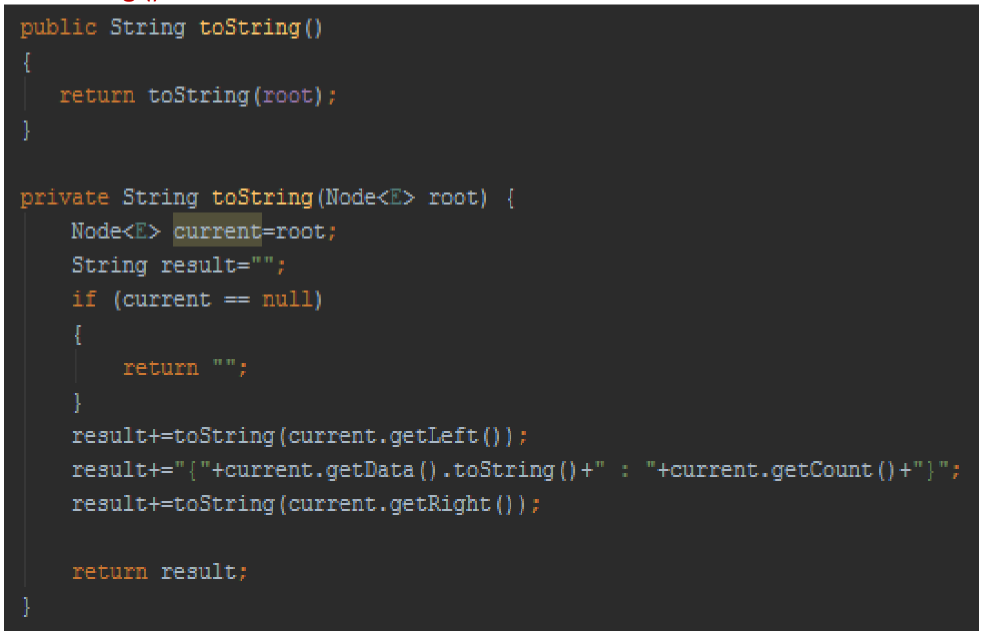
->found the correct node, assign the count of item to elementsize and break the loop.
->return elementsize.

printInorder (Bag bag)



```java
public static void printInorder(Bag bag)
{
    if (bag.root == null)
        return;
    else{
        if(bag.root.hasLeft())
            printInorder(new Bag(bag.root.getLeft()));

            System.out.print(bag.root.getData()+" :"+bag.root.getCount()+"  ");

        if(bag.root.hasRight())
            printInorder(new Bag(bag.root.getRight()));
    }
}
```

->display the Bag by using in-order traversal toString ()



```java
public String toString()
{
    return toString(root);
}

private String toString(Node<E> root) {
    Node<E> current=root;
    String result="";
    if (current == null)
    {
        return "";
    }
    result+=toString(current.getLeft());
    result+="{"+current.getData().toString()+" : "+current.getCount()+"}";
    result+=toString(current.getRight());

    return result;
}
```

->I needed a new toString method, Because I decided to use recursion in the method to traverse tree.

->traverse the tree and display the Bag by using in-order traversal

BERAT GÜMÜŞ
140315040