CELAL BAYAR UNIVERSITY COMPUTER ENGINEERING

CSE-2105 DATA STRUCTURES

2019-2020 FALL SEMESTER PROJECT REPORT

Name: Ege Kutay

Surname: YÜRÜŞEN

Number: 180316017

Department: Computer Engineering

Subject: The Bag Abstract Data Type

Subject is separated according to:

1) Node class

2)The Comparable Interface

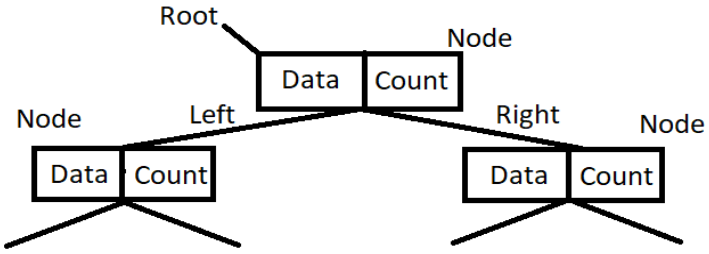
3)Bag class

4)Test class

5)Appendix

1) Node Class

Firstly, the Node Class is created for the Bag ADT which is structed by 'Binary Search Tree (BST)'. I used Binary Search Tree because it is one of the ways to implement the Bag ADT without using any external packages from the java library. Furthermore, BSTs are much more efficient than linked lists and BSTs has dynamic size. The bag has structure like:



Due to picture of above, each node contains the data of an item and contains the data of how many is the same item contained in the node as integer. As the result, each node keeps singular data of items which

makes it simpler to process distinct size and size for the Bag ADT. On the other hand, makes the program more efficient for the Bag ADT operations sincerely you don't have to add node into the BST for the same item.

Some of the fields in the Node class and Bag class are generic type. Generic types allow you to implement the data with different data types. In my project the generic type parameter is called "AnyType".

```
public class Node<AnyType>
{
    private int quantity;
    private AnyType item;
    private Node<AnyType> left;
    private Node<AnyType> right;
    public Node(AnyType stuff)
    {
    item=stuff;
    setLeft(null);
    setRight(null);
    quantity=1;
    }
```

The Node class has 4 fields which are:

AnyType 'item': This field used for containing the item's data, the item's data is inserted from the user.

Integer 'quantity': This field used for containing the information of quantity of an item.

Node (AnyType) 'left, right': These fields are used for reach to the root's left child and root's right child.

The methods of the Node class are mostly setters and getters for the fields. Except for these setters and getters, the Node class has these methods:

```
public void decreaseQuantity()
{
quantity--;
}
public void increaseQuantity()
{
quantity++;
}

public boolean hasLeft()
{
    if(left!=null)
        return true;
    else
        return false;
}
public boolean hasRight()
{
    if(right!=null)
        return true;
    else
        return false;
}
```

increaseQuantity(): Increases the quantity of the item.

decreaseQuantity(): Decreases the quantity of the item.

hasLeft(): Checks if left node is null or not, if the right node isn't null then it returns true else it returns false

hasRight(): Checks if right node is null or not, if the right node isn't null then it returns true else it returns false.

2)The Comparable Interface

Sincerely my Bag project has the BST structure the program needs to compare two nodes which belongs to the bag. In order to compare nodes I used comparable interface which is located in java.lang package.

This interface contains only one method which is called compareTo(). With compareTo() method you can only sort the elements basis of single data member only. For example: it maybe roll no, name, age etc. With compareTo() method we can sort elements of:

- String objects
- Wrapper class objects
- User defined-class objects

The compareTo() method returns :

- Positive integer, if the current root object is greater than the specified object.
- Negative integer, if the current root object is lower than the specified object.
- Zero, if the current root object is equal to the specified object.

In the Bag ADT project describe there is an example which is the sequence of strings: 'to', 'be'

'or', 'not', 'to', 'be'.

Come to think of, how does compareTo() method compares two string objects with each other?

The compareTo() method compares two string object lexicographically (dictionary order). As the result compareTo() method returns:

- Positive integer, if the current root string object comes first in dictionary than the specified string object.
- Negative integer, if the current root string object comes later in dictionary than the specified string object.
- Zero, if the current root string object equals to the specified string object.

If the string object formed as letter or word, it orders letters and words according to English Dictionary.

3) The Bag Class

The Bag class extends comparable interface, sincerely the bag has BST structure some of the bag operations need to make some comparisons in order to execute.

```
public class Bag<AnyType extends Comparable<AnyType>>
{
private int size;//returns size
private int distictsize;//returns distict size
private int i=0;// used for equals() method
private Node<AnyType> root;
public Bag() {
size=0;
distictsize=0;
root=null;
}
```

The bag class has 4 fields which are:

- int size: Contains the integer data of the bag's size
- int distict size: Contains the integer data of the bag's distinct size.
- int i: This is used for equal method, it's going to mentioned later in the equals() class.
- Node<AnyType> root: 'root' is referring to root of the tree.

The Bag Class Methods:

A) public void add(AnyType item)

```
public void add(AnyType item) {

    size++;
    root= add(item,root);//A1
}
```

The add method AnyType parameter is entered by the user.

Whenever the add method invoked it's increases the bag's size. Because every time we invoke the add method, we add an item.

- A1: Invokes the add method with 'item' and 'root' (root of the bag) parameters and assigns them to root itself. Sincerely root is a node type object the right side of the equation should be a node type object. So, I a need a new add method that returns node type object. On the other hand, I needed a new add method because I decided to use recursion in the method.

```
private Node<AnyType> add(AnyType Item,Node<AnyType> temp){
    if(temp==null)//A2
    {
    distictsize++;
    return new Node<AnyType>(Item);
    }
    else
    {
        if(Item.compareTo(temp.getItem()) <0)//A3
        {

            temp.setLeft(add(Item,temp.getLeft()));
        }
        else if(Item.compareTo(temp.getItem())>0)//A3
        {
            temp.setRight(add(Item,temp.getRight()));

        }
        else if(Item.compareTo(temp.getItem())==0)//A4
        temp.increaseQuantity();
        return temp;
    }
}
```

The method is private sincerely it's not necessary to access from outside of the bag class. The private add method has these properties:

- A2: If the temporary node is null creates new node and returns the new node. Sincerely every node contains singular data of items the statement increases bag's distinct size.
- A3: Comparison, if new item's data('item') is less than the item's data of temporary node('temp), recall the method with new item's data and current node's left child paramaters.
- Second A3: Comparison, if new item's data('item') is more than the item's data of temporary node('temp'), recall the method with new item's data and current node's right child paramaters.
- A4) Comparison, if the new data equals to the temporary node's data, the private add method increases quantity of item.

Conclusion of the add operation: A3 statements are work for roaming inside of the bag to find the correct place to set the new item's data. When the correct place is found in the bag; it creates new node and it adds the new node to the bag. If the new item's data is already existing in the bag the item's quantity is increased by one.

B) public boolean Remove(AnyType item)

```
public boolean remove(AnyType item) {
if(!contains(item)) //B1
{
    System.out.println("The ["+item+"] item is not contained in the bag");
    return false;
}
else //B2
{
    size--;
    root = remove(item,root);
    return true;
}
}
```

The AnyType parameter is set by the user.

If statement: If the new item's data doesn't contain in the bag, it returns false with a message.

Else statement: Invokes a new remove method with ‘item’ and ‘root’ (root of the bag) parameters and assigns a new remove method to root itself. Sincerely root is a node type the right side of the equation should be a node type object. So, I a need a new remove method that returns node type. The other reason that I need a new remove method, I decided to use recursion in the method.

When remove method invoked and program gets into the else statement, it reduces size of the bag by one sincerely bag size reduced when an item is removed from the bag despite what the item it is.

```
private Node<AnyType> remove(AnyType item,Node<AnyType> temp){

    if(item.compareTo(temp.getItem())<0)//B1
        temp.setLeft(remove(item,temp.getLeft()));
    else if(item.compareTo(temp.getItem())>0)//B2
        temp.setRight(remove(item,temp.getRight()));
    else //B3
    {
         if(temp.getQuantity()>1) //B4
        {
          temp.decreaseQuantity();
          return temp;
        }
         else //B5
        {
             distictsize--;
             if(temp.getLeft()==null && temp.getRight()==null)//B6
            {
             temp=null;
            }
             else if(temp.getLeft()==null) //B7
                return temp.getRight();
             else if(temp.getRight()==null) //B8
                 return temp.getLeft();
             else //B9
             {
                 Node<AnyType> minfromright=findMinFromRight(temp.getRight());
                 temp.setItem(minfromright.getItem());
                 temp.setRight(remove(minfromright.getItem(),temp.getRight()));
             }
        }
 }
return temp;
}
```

- B1: Comparison, if new item's data(‘item’) is less than the item's data of temporary node(‘temp’), recall the method by giving new item's data and temporary node’s left child as parameters.
- B2: Comparison, if new item's data(‘item’) is more than the item's data of temporary node(‘temp’), recall the method by giving new item's data and temporary node’s right child as parameters.
- B3: The program found the correct item in the bag that the user entered for.
- B4: If the quantity of the item is more than one reduces the item’s quantity.
- B5: The program enters this statement if the quantity of the item is equal to one. Sincerely the program removing singular data of the item, the method reduces distinct size of the bag.
- B6: If the right child and left child of the temporary(‘temp’) root node is empty, it assigns temporary as null.
- B7: If the left child of the temporary root node is null, it returns right node of the temporary node.
- B8: If the right child of the temporary root node is null, it returns left node of the temporary node.

- B9: If both child of the temporary node isn't null; it finds the minimum value from right subtree then set the minimum value to the temporary node. After this process it removes the minimum value from right subtree.

B10: private Node<AnyType> findMinFromRight(Node<AnyType> node)

```
private Node<AnyType> findMinFromRight(Node<AnyType> node) {//B10
    while(node.getLeft()!=null)
    {
        node=node.getLeft();
    }
    return node;
}
```

The method returns the minimum value from right subtree of the given parameter.

Conclusion of the remove operation: If the entered item is not existing in the bag, the method returns false else it returns true and the program access to the private remove method. B1 and B2 statements are works for roaming inside of the bag to find the correct item. When correct item is found, it checks left child and right child of the temporary node (the correct item's node) are null. If both are null, the remove method empties the temporary node. If the only one of children is null, the remove method returns the children that isn't null. If both child of the temporary root node isn't null, the program finds the minimum value from right subtree of the temporary root node and set it into the temporary root node. Then remove method recalled for the minimum value of the right subtree in order to remove the minimum value. If the correct item quantity is bigger than one it reduces it's quantity by one else the method reduces distinct size one sincerely every item's data is singular for each node.

C) public int Size()

Returns the size of the bag as an integer.

```
public int Size() {
    return size;
}
public int DistictSize() {
    return distictsize;
}
```

D) public int DistictSize()

Returns distinct size of the bag as an integer.

E) public boolean Contains(AnyType item)

```
public boolean Contains(AnyType item){
    if(root==null)//E1
        return false;
Node<AnyType> current=root;
boolean exist=false;
while(!exist)//E2
{
 if(item.compareTo(current.getItem())<0)//E3
 {
     if(current.getLeft()!=null)
     current=current.getLeft();
     else
     break;
 }
 else if(item.compareTo(current.getItem())>0)//E4
 {
     if(current.getRight()!=null)
         current=current.getRight();
         else
         break;
 }
 else if(item.compareTo(current.getItem())==0)//E5
 {
     exist=true;
 }
}
return exist;
}
```

- E1: Checks if the bag is empty. If it is, returns false.
- E2: the while loop is terminated when the item turns true or it terminates the while loop when the bag does not contain the item's data after roaming inside of the bag.
- E3: Comparison, if the item's data is less than the current root node item's data go to left. If the left child is null, break the loop.
- E4: Comparison, if the item's data is more than the current root node item's data go to right. If the right child is null, break the loop.
- E5: Comparison, if the item's data is equal to the current root node item's data terminate the the loop.

After the while loop terminated it returns 'exist' as boolean type.

Conclusion of contains operation: If the root is empty there is no way that the bag is contains any item, so it returns false. E1 and E2 statements work for roaming to find the correct place of item's data. If the item's data is found in the bag it returns true else, it returns false.

F) public boolean equals(Object obj)

I needed a new equals method because I decided to use recursive method. I used the private equals method which takes 2 parameter one is entered from the user (‘bag’) and the other is current root bag’s root (‘root’).

```
public boolean equals(Object obj) {
    Bag bag=(Bag) obj;//F1
    equals(bag,root);
    if(bag.DistictSize()!=DistictSize()|| bag.Size()!=Size())//F2
        return false;
    else if(i>0)//F3
        return false;
     else
         return true;
}
private void equals(Bag bag,Node<AnyType>root) {
    if(!bag.Contains(root.getItem()))//F4
        i++;
     if(root.hasLeft())
    {
        equals(bag,root.getLeft());//F5

    }
     if(root.hasRight()) //F6
    {
        equals(bag,root.getRight());
    }
}
```

- F1: Down casts the object as bag object type.
- F2: If the entered bag’s distinct size and size are not equal to the other bag the method returns false.
- F3: If the integer ‘i’ is bigger than zero, returns false else returns true.
- F4: If the entered bag doesn’t contain the other bag’s current root’s item data it increases ‘i' by 1.
- F5: If the entered bag’s current root has left child recalls the private equals method with entered bag and the current root left child.
- F6: If the entered bag’s current root has right child recalls the private equals method with entered bag and the current root right child.

Conclusion of the equals operation:If the entered bag’s distinct size and size is not equal to the other bag’s, there is no way that two bags are equal. Furthermore, even if bags size and distinct size are equal, we need to check if every item of the entered bag contained in the other bag. The F5 and F6 statements works for preorder traversing inside of the bag, when the whole bag is preorder traversed and if the F4 statement does not come true while traversing, the actual equal method returns true sincerely ‘i’ is 0 else, the equals method returns false.

G) public int ElementSize(AnyType item)

```
public int ElementSize(AnyType item) {
    Node<AnyType> current = root;
    boolean found=false;
    if(!Contains(item))//G1
        return 0;
    else
 {
    while(!found)
    {
        if(item.compareTo(current.getItem())<0) //G2
        {
        current=current.getLeft();
        }
        else if(item.compareTo(current.getItem())>0) //G3
        {
        current=current.getRight();
        }
        else if(item.compareTo(current.getItem())==0) //G4
        {
            found=true;
        }
    }
    return current.getQuantity();//G5
```

- G1: If the bag doesn't contain the entered item the method returns 0.
- G2: Comparison, if entered item data is less than the data of current root node, go left.
- G3: Comparison, if entered item data is more than the data of current root node, go right.
- G4: Comparison, if entered item data is equal to current root node, terminate the loop.
- G5: Returns the quantity of the current root node.

Conclusion of the Element Size operation:

The program check if the entered item contained in the bag. If the bag doesn't contain the entered item, it returns 0 else the program enters to the while loop. G2 and G3 are works for roaming in the bag in order to find the placement of the entered item. When the program finds it, the method returns the quantity of the item in the bag.

H) public String toString()

```
public String toString() {
return toString(root);//H1
}
private String toString(Node<AnyType> root) {
if(isEmpty()) //H2
{
return "Bag is empty";
}
else {//H3
    Node<AnyType> current = root;
    String result = "";
    if (current == null)//H4
    {
        return "";
    }
    result += "[" + current.getItem().toString() + "" + "("+current.getQuantity()+ ")"+ "]";
    result += toString(current.getLeft());
    result += toString(current.getRight());
    return result;
    }
}
}
```

- H1: I needed a new toString() method because I decided to use recursion in the method. So I used to toString() method which takes one parameter which is the root of the tree.
- H2: If the bag is empty is returns a message that reports bag is empty.
- H3: If the bag is not empty the program is preorder traversing and printing all nodes in the bag.

I) public boolean isEmpty()

If the root is null, method returns true else the method returns false.

```
public boolean isEmpty() {
    if(root==null)
        return true;
    else
        return false;
}
public void clear() {
    size=0;
    distictsize=0;
    root=null;
}
```

J) public boolean clear()

Makes size, distinct size 0 and assigns root as null. If the root is null, you can't access to the other elements, so the bag becomes empty.

4) Test Class

The code:

```
public static void main(String [] args)
{
    Bag bag1=new Bag();
    bag1.add("to");
    bag1.add("be");
    bag1.add("or");
    bag1.add("not");
    bag1.add("to");
    bag1.add("be");
System.out.println(" 'to', 'be','or','not','to' ,'be' added to the bag1:");
System.out.println(bag1);
System.out.println("Size of the bag1 : "+bag1.Size());
System.out.println("----------------------");
System.out.println("Distinct size of our bag1 : "+bag1.DistictSize());
System.out.println("----------------------");
System.out.println("Element size of 'to' : "+bag1.ElementSize("to"));
System.out.println("----------------------");
System.out.println("Checking the bag if the bag1 contains 'or' : "+bag1.Contains("or"));
System.out.println("----------------------");
System.out.println("Clearing the bag1...");
bag1.clear();
System.out.println("is the bag1 empty?" +bag1.isEmpty());
```

Result:

```
 'to', 'be','or','not','to' ,'be' added to the bag1:
[to(2)][be(2)][or(1)][not(1)]
Size of the bag1 : 6
----------------------
Distinct size of our bag1 : 4
----------------------
Element size of 'to' : 2
----------------------
Checking the bag if the bag1 contains 'or' : true
----------------------
Clearing the bag1...
is the bag1 empty? true
----------------------
```

The code:

```
bag1.add("to");
bag1.add("be");
bag1.add("or");
bag1.add("not");
bag1.add("to");
bag1.add("be");
System.out.println("----------------------");
System.out.println(" 'to', 'be','or','not','to','be' added to the bag1 :");
System.out.println(bag1);
Bag bag2=new Bag();
bag2.add("not");
bag2.add("be");
bag2.add("or");
bag2.add("to");
bag2.add("be");
bag2.add("to");
System.out.println("----------------------");
System.out.println("'not' 'be' 'or' 'to' 'be' 'to' added to bag2");
System.out.println(bag2);
System.out.println("----------------------");
System.out.println("Checking if bag1 equals bag2 : "+bag1.equals(bag2));
System.out.println("----------------------");
System.out.println("Removing 'or' from our bag1");
System.out.println("Is 'or' removeable from bag1 ? : "+bag1.remove("or"));
System.out.println("Result: "+bag1);
System.out.println("----------------------");
System.out.println("Trying to remove 'or' from our bag1:");
System.out.println("Is 'or' removeable from bag1 ? : "+bag1.remove("or"));
System.out.println("----------------------");
System.out.println("Checking if  bag1 and bag2 equals after removing 'or' from bag1 : "+bag1.equals(bag2));
System.out.println("----------------------");
```

Result:

```
 'to', 'be','or','not','to','be' added to the bag1 :
[to(2)][be(2)][or(1)][not(1)]
----------------------
'not' 'be' 'or' 'to' 'be' 'to' added to bag2
[not(1)][be(2)][or(1)][to(2)]
----------------------
Checking if bag1 equals bag2 : true
----------------------
Removing 'or' from our bag1
Is 'or' removeable from bag1 ? : true
Result: [to(2)][be(2)][not(1)]
----------------------
Trying to remove 'or' from our bag1:
The [or] item is not contained in the bag
Is 'or' removeable from bag1 ? : false
----------------------
Checking if  bag1 and bag2 equals after removing 'or' from bag1 : false
```

Code:

```
Bag bag3=new Bag();
bag3.add("b");
bag3.add("b");
bag3.add("or");
bag3.add("to");
bag3.add("to");
bag3.add("not");
System.out.println("'not' 'be' 'or' 'to' 'be' 'to' added to bag2");
System.out.println(bag2);
System.out.println("Size of bag2 : "+bag2.Size());
System.out.println("Distict size of bag2 : "+bag2.DistictSize());
System.out.println("----------------------");
System.out.println("'b','b','or','to','to','not' added to bag3");
System.out.println(bag3);
System.out.println("Size of bag3 : "+bag3.Size());
System.out.println("Distinct size of bag3 : "+bag3.DistictSize());
System.out.println("----------------------");
System.out.println("Checking if bag2 and bag3 equals each other : "+bag2.equals(bag3));
System.out.println("----------FOR INTEGERS-----------");
Bag ibag=new Bag();
ibag.add(4);
ibag.add(6);
ibag.add(7);
ibag.add(2);
ibag.add(2);
ibag.add(3);
System.out.println("'4','6','7','2','2','3' added to the integer bag1: "+ibag);
System.out.println("");
Bag ibag2=new Bag();
ibag2.add(3);
ibag2.add(2);
ibag2.add(6);
ibag2.add(2);
ibag2.add(7);
ibag2.add(4);
System.out.println("'3','2','6','2','7','4' added to the integer bag2: "+ibag2);
System.out.println("Checking if ibag and ibag2 equal each other :"+ibag.equals(ibag2));
System.out.println("Removing '4' from integer bag1:"+ibag.remove(4)+"\n"+ibag);
System.out.println("Checking if  integer bag1 and bag2 are equal after removing '4' from integer bag1: "+ibag.equals(ibag2));
System.out.println(ibag);
```

Result:

```
'not' 'be' 'or' 'to' 'be' 'to' added to bag2
[not(1)][be(2)][or(1)][to(2)]
Size of bag2 : 6
Distict size of bag2 : 4
----------------------
'b','b','or','to','to','not' added to bag3
[b(2)][or(1)][not(1)][to(2)]
Size of bag3 : 6
Distinct size of bag3 : 4
----------------------
Checking if bag2 and bag3 equals each other : false
----------FOR INTEGERS-----------
'4','6','7','2','2','3' added to the integer bag1: [4(1)][2(2)][3(1)][6(1)][7(1)]

'3','2','6','2','7','4' added to the integer bag2: [3(1)][2(2)][6(1)][4(1)][7(1)]
Checking if ibag and ibag2 equal each other :true
Removing '4' from integer bag1:true
[6(1)][2(2)][3(1)][7(1)]
Checking if  integer bag1 and bag2 are equal after removing '4' from integer bag1: false
[6(1)][2(2)][3(1)][7(1)]
```

5)Appendix

Sources:

https://moodle.cbu.edu.tr/mod/resource/view.php?id=904 //Binary Search Tree lab source codes

https://beginnersbook.com/2013/12/java-string-compareto-method-example/

https://www.javatpoint.com/Comparable-interface-in-collection-framework

https://www.java2novice.com/java-interview-programs/delete-node-binary-search-tree-bst/

https://www.geeksforgeeks.org/generics-in-java/

https://github.com/beratgumus/The-Bag-ADT