CELAL BAYAR UNIVERSITY COMPUTER ENGINEERING

CSE-2105 DATA STRUCTURES

2020-2021 FALL SEMESTER PROJECT REPORT

NAME: Hüsamettin

SURNAME: Demirtaş

NUMBER: 190315074

GitHub Link: https://github.com/MrIronstone/DataStructuresProject

This project has been developed by only me and you can observe from my GitHub commit log how did I progress throughout the development process.

I've decided that if I put my report in pieces, it would be easier to understand.

My project has 5 different class. I will explain their purpose of creation and what they do. We can list it as follows:

1. Student
2. Comparable Interface
3. MyArrList
4. MyBinarySearchTree
5. MySeperateChainingHashTable
6. MainTest

1) STUDENT CLASS

First, despite its essential role in this project of this class, it has been created due to project requirements and this project's abstract data types generically creation allows you manipulate with this class as you want.

Fields of this class are name, surname and they have been encapsulated. While name and surname are string, id is integer. The reason why I chose id as integer is when I need to compare two separate students, I will compare by their IDs.

To compare those classes instances when needed, I implemented the Comparable Interface and overridden its method. I also overridden the object class's toString method and I rearranged the output string. Before my rearrangement it was printing like "Student@49097b5d" but after my rearrangement it is printing like [Hüsamettin Demirtaş 190315074].

2) COMPARABLE INTERFACE

My project has the Binary Search Tree ADT that needs to compare two nodes which belongs to the Student. In order to compare nodes I used comparable interface which is located in java.lang package.

To compare two generic objects, I had to extend the comparable interface to generic types while I implemented the Comparable directly to whole class on Student class. So I used generic types THAT has compareTo method on my BST and Node types.

```
1  public class BinarySearchTree<T extends Comparable<? super T>>
2  {
3
4      private class Node<T extends Comparable<? super T>>
5      {
6          private Node<T> left;
7          private T data;
8          private Node<T> right;
9
```

This allows me to compare two separate generic object that has overridden the compareTo method. Without that java cannot compare two separate generic objects.

```
1   public class Student implements Comparable<Student>
2       private String name;
3       private String surname;
4       private int studentID;
```

The compareTo method that I implemented to my student class returns positive one if the current student's id is bigger than the compared student's id, returns zero if equals but this cannot be happened, and if the current student's id is smaller than the compared student's id, it returns negative one. So, this implementation basically helps me to compare two separate my own creation classes like java's primitive classes etc. integer.

```
@Override
public int compareTo(Student o) {
    if( this.getID() < o.getID())
        return -1;
    else if( this.getID() == o.getID())
        return 0;
    else
        return 1;
}
```

3) MyArrList CLASS

In this class I used array implementation of List Type. The reason why I used this implementation is I thought like I'm building basic student system and students are only removed and added once a year, and only their information is read throughout the year. So, I thought if I use array implementation which has fast read and change benefits rather than node implementation, it would be better.

```
public class MyArrList<T>
{
    private static final int DEFAULT_CAPACITY = 10;

    private T[] theItems;
    private int theSize;
```

We can basically say that in this class, I have generic type of array that has 10 capacity when this class is created. It has size counter

rather than capacity. While capacity displays the size of the array that in background, size is displaying the actual spatial fullness of the array. This class's constructor method simply calls clear method and clear method equals the size to 0 and create new array with default capacity.

```
public MyArrList()
{
    clear();
}
```

```
public void clear()
{
    theSize = 0;
    theItems = (T[]) new Object[DEFAULT_CAPACITY];
}
```

This class has get method that gets the element at the given index.

```
public T get( int idx )
{
    if( idx < 0 || idx >= theSize )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + theSize );

    return theItems[ idx ];
}
```

It also has a method that changes the value of the given index.

```
public T set( int idx, T newVal )
{
    if( idx < 0 || idx >= theSize )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + theSize );

    T old = theItems[ idx ];
    theItems[ idx ] = newVal;

    return old;
}
```

It has a method that adds an element at the end which calls a method that adds to a given index, when adding at the and method called it simple calls other adding method and gives the size as index to other method. Naturally increases the size when an element is added.

```
public boolean add(T x)
{
    add(theSize, x);
    return true;
}

public void add(int idx, T x)
{
    if( idx < 0 || idx > theSize )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + theSize );

    if(theItems.length == theSize)
        ensureCapacity(theSize * 2);

    for(int i=theSize; i>idx; i--)
        theItems[i] = theItems[i - 1];

    theItems[idx] = x;

    theSize++;
}
```

It has remove method that removes the element at given index.

```
public void remove(int idx)
{


    if( idx < 0 || idx >= theSize )
        throw new ArrayIndexOutOfBoundsException( "Index " + idx + "; size " + theSize );

    T removedItem = theItems[ idx ];

    for(int i=idx; i<theSize-1; i++)
        theItems[i] = theItems[i + 1];

    theSize--;

}
```

It has a method that returns its size, a method that checks its emptiness.

```
public int size()
{
    return theSize;
}
```

```
public boolean isEmpty()
{
    return theSize == 0;
}
```

When the size is equals to capacity, the Class has a method that ensures new capacity, two times of older. It creates new array of

generic elements with double size of older array and copies shallowly elements to new array.

```
private void ensureCapacity(int newCapacity)
{
    T[] old = theItems;

    theItems = (T[]) new Object[newCapacity];

    for( int i = 0; i < theSize; i++ )
        theItems[ i ] = old[ i ];
}
```

And of course, when we want to print this array, we have to override the toString method from Object Class to print properly and we have to rearrange it.

```
@Override
public String toString()
{
    String rStr = "[ ";

    for(int i=0; i < theSize; i++)
        rStr = rStr + theItems[i] + " ";

    rStr = rStr + "]";

    return rStr;
}
```

4) MyBinarySearchTree

In this class I've created Binary Search Tree ADT. To build proper binary search tree, comparisons must be done. To do that in generic structure, I have extended generic types from generic comparable structures.

```
public class MyBinarySearchTree<T extends Comparable<? super T>>
{
```

This tree consists of nodes which is another ADT. Each node has 3 different fields.

```
private class Node<T extends Comparable<? super T>>
{
    private Node<T> left;
    private T data;
    private Node<T> right;
```

First one is data of course, Nodes are generic as well, so data is generic. Left and right handlers “pointers” of its own type. Nodes also has encapsulation, constructer methods.

```
public Node(Node<T> l, T d, Node<T> r)
{
    left = l;
    data = d;
    right = r;
}
```

BST has root handler as a field and has lots of methods to make it run like real BST. It has a function named insertIterativly that adds elements iteratively.

```
public void insertRecursively(T newValue)
{
    root = insertRecursively(newValue, root);
}

private Node<T> insertRecursively(T newValue, Node<T> tempNode)
{
    if( tempNode == null )
        return new Node<>(null, newValue, null);

    if( newValue.compareTo(tempNode.getData()) > 0)
        tempNode.setLeft(insertRecursively(newValue, tempNode.getLeft()));
    else
        tempNode.setRight(insertRecursively(newValue, tempNode.getRight()));

    return tempNode;
}
```

```
public void insertIteratively(T newValue)
{
    Node<T> newNode = new Node<>(null, newValue, null);

    if(isEmpty())
        root = newNode;
    else
    {
        Node<T> tempNode = root;

        while(true)
        {
            // newValue < tempNode.getData()
            if(newValue.compareTo(tempNode.getData()) < 0)

            {
                if(tempNode.getLeft() == null)
                {
                    tempNode.setLeft(newNode);
                    break;
                }
                else
                    tempNode = tempNode.getLeft();
            }
            else
            {
                if(tempNode.getRight() == null)
                {
                    tempNode.setRight(newNode);
                    break;
                }
                else
                    tempNode = tempNode.getRight();
            }
        }
    }
}
```

It creates a new node with given data and enters to infinite all nodes it encounters and place and connects the node at the right position.

It also has an insertRecursively method that adds recursively. It does the same job with previous, but its recursion property makes its code simpler. In this method, it simply compares the given data and temp data, if given is bigger than temp, it forwards to right of the temp and vice versa.

Connects nodes properly and it can compare generic type datas thanks to comparable extension.

```
private static final int DEFAULT_TABLE_SIZE = 10;

private LinkedList<T>[] theLists;
```

It has deletion method named deleteRecursively, it simple deletes given node and rearrange the disorder caused by removing a node by reordering with the right's minimum node. To decide which node to replace, it has other method named as inOrderSuccessor. It has method that checks its emptiness and of course it has printing methods in three different ways; preorder, inorder and postorder. Each traversal has a unique printing method named same with their traversal style.

```
public Node<T> deleteRecursively(Node<T> root, T value) {

    if (root == null)
        return root;
    if (root.data.compareTo(value) > 0) {
        root.left = deleteRecursively(root.left, value);
    } else if (root.data.compareTo(value) < 0) {
        root.right = deleteRecursively(root.right, value);
    } else {

        if (root.left == null) {
            return root.right;
        } else if (root.right == null)
            return root.left;

        root.data = inOrderSuccessor(root.right);
        root.right = deleteRecursively(root.right, root.data);
    }
    return root;
}
```

```
public T inOrderSuccessor(Node<T> root) {
     T minimum = root.data;
    while (root.left != null) {
        minimum = root.left.data;
        root = root.left;
    }
    return minimum;
}
```

5) MySeperateChainingHashTable CLASS

In this class I've created a Hash Table that has a fields as different generic lists' handler and constant integer 10(ten) as a guidance for first creation. It has constructer method that creates the lists in a for loop for handler with constant integer 10. Thus, hash table has been created and waiting for inputs.

```
public MySeparateChainingHashTable()
{
    theLists = new LinkedList[DEFAULT_TABLE_SIZE];

    for(int i=0; i<theLists.length; i++)
        theLists[i] = new LinkedList<>();
}
```

```
private int myHash(T x)
{
    return (x.hashCode() % theLists.length);
}
```

When I need to store an element, I get that element's hash code and take the result from mod list's length. This gives me an integer index number.

I put the element appropriate position. To do that, class has lots of methods. To insert it has method named insert.

```
public void insert(T x)
{
    LinkedList<T> whichList = theLists[myHash(x)];

    if(!whichList.contains(x))
        whichList.add(x);
}
```

It first sends an element to hashing function method named as myHash, in that method, like I mentioned it gets its hash code, take result from mod by list's length and this returns integer number that is an index. In insert method, when this method returns the index, method puts this index and checks first if this element is already in the chain list. If not, it adds the element to the chain. This is done by "contains" method. This method, this takes the element, hashes it and checks if it is already in the list. If yes returns true, if not return false.

```
public boolean contains(T x)
{
    LinkedList<T> whichList = theLists[myHash(x)];

    return whichList.contains(x);
}
```

Removing is doing the same thing just at checking stage, it checks IF is contains and if yes, it removes the element from the chain. Removing and adding stages are done by java's pre-written List class.

```
public void remove(T x)
{
    LinkedList<T> whichList = theLists[myHash(x)];

    if(whichList.contains(x))
        whichList.remove(x);
}
```

It has a clearing method named as makeEmpty and it simply sweep all generic lists with for loop.

```
public void makeEmpty()
{
    for(int i=0; i<theLists.length; i++)
        theLists[i].clear();
}
```

To print it readable, I had to override the object's toString method and I used double for loop now it can be read crystal clear.

```
public void printHashTable()
{
    LinkedList<T> whichList;

    for(int i=0; i<theLists.length; i++)
    {
        whichList = theLists[i];

        System.out.print("|" + i + "|" + " --> ");

        for(int j=0; j < whichList.size(); j++)
            System.out.print(whichList.get(j) + " --> ");

        System.out.println();
    }
}
```

6) MainTest CLASS

This class is the things gets working. In this class I have configured the console that user will encounter as our instructor wanted. The project description says that console must have 8 different part starts from 0 to 7.

First, to use our abstract data types we must initialize them and Scanner object first.

```
public class MainTest {

    public static void main(String[] args) {

        MyArrList<Student> arrList1 = new MyArrList<>();
        MyBinarySearchTree<Student> Tree1 = new MyBinarySearchTree<>();
        MySeparateChainingHashTable<Student> HashTable1 = new MySeparateChainingHashTable<>();

        Scanner input = new Scanner(System.in);
```

When the user starts the program, it must print 8 option to console and user must select one of them. To manage this, I used switch case. Due to option at 0 is about exit command, I've created infinite loop and if user want to exit can select 0. As I mentioned there are only 8 different options and their numbers are written on the console. If user types different from them, switch case will run the default case which warns user about wrong input and will go back to start due to infinite loop. Instead of ending the program unexpectedly, loop the user endlessly is better option.

Let's start with case 0. In case 0, user will be asked if user really want to exit to be sure and user must type specified character to the screen shown by the program. In case of typing wrong character and ending unexpectedly, here I used infinite loop again. In this scenario, if user is sure about exit, must type Y. If so, program will end. If not, program will go back to previous infinite loop which is intended. Thus, if wrongly pressed to 0, program won't close.

```
case "0" -> {

    while (true) {
        // infinite loop if user doesn't input correctly
        System.out.println("Do you really want to exit? (Y/N) ");
        String exitAnswer = input.nextLine();
        if (exitAnswer.equals("Y"))
            System.exit(0);
        if (exitAnswer.equals("N"))
            break;
        else
            System.out.println("Wrong Input, try again");
    }
}
```

In case 1, user can add student. I get the name, surname and id of student from user. On student system, there can't be two students with same ID number, so first I check from arraylist structure if that ID already exist if not, create a new Student and add to all structures by using their add command.

```
case "1" -> {
    /*
    This option adds the student to the list, tree and hash structures. You should get from
    the user student id, name and surname before adding student. New student should be
    added to the appropriate positions of the structures according to its student id (i.e
    student id is the key).
    */

    System.out.println("Please enter name");
    String name = input.nextLine();

    System.out.println("Please enter surname");
    String surname = input.nextLine();

    System.out.println("Please enter id");
    int id = Integer.parseInt(input.nextLine());


    boolean isContain = false;
    for (int i = 0; i < arrList1.size(); i++) {
        if (arrList1.get(i).getID() == id) {
            isContain = true;
            break;
        }
        isContain = false;
    }
    if (isContain) {
        System.out.println("This ID already exist");
    } else {
        Student newStudent = new Student(name, surname, id);
        arrList1.add(newStudent);
        Tree1.insertIteratively(newStudent);
        HashTable1.insert(newStudent);
    }
}
```

In case 2, program will get the student id from the user and (if found) delete this student from all structures. To do that, checks the ID from the arraylist if exist or not, if doesn't exit warns user by printing "This ID doesn't exist". If exist, it deletes the student from all data structures.

```
case "2" -> {
    /*
    First get the student id from the user and (if found) delete this student from all
    structures.
    */
    System.out.println("Please enter the ID that you want delete");
    int wantedToDeleteID = Integer.parseInt(input.nextLine());
    int index = 0;
    boolean isFound = false;
    for (int i = 0; i < arrList1.size(); i++) {
        if (arrList1.get(i).getID() == wantedToDeleteID) {
            isFound = true;
            break;
        }
        isFound = false;
        index++;
    }
    if (isFound) {
        // The place where the deletion is done from all data structures
        arrList1.remove(index);
        Tree1.deleteRecursively(Tree1.root,arrList1.get(index));
        HashTable1.remove(arrList1.get(index));
    } else {
        System.out.println("The ID doesn't exist");
    }
}
```

In case 3, program ask user the ID that he/she wants to print information of. Program search the all students' id and if the id user

typed is not existing, warns the user as “This ID doesn’t exist”. But if found, prints information of this student to the screen.

```
case "3" -> {
    /*
    First get the student id from the user and (if found) and print this student's properties
    to the screen. Also print the number of search levels (how many hops) that you found
    the student from the list, tree and hash.
    */
    System.out.println("Please enter the ID that you want print");
    int wantedToPrint = Integer.parseInt(input.nextLine());
    int index2 = 0;
    boolean isFound2 = false;
    for (int i = 0; i < arrList1.size(); i++) {
        if (arrList1.get(i).getID() == wantedToPrint) {
            isFound2 = true;
            break;
        }
        isFound2 = false;
        index2++;
    }
    if (isFound2) {
        //
        System.out.print("[");
        System.out.print(arrList1.get(index2).getID());
        System.out.print(", ");
        System.out.print(arrList1.get(index2).getName());
        System.out.print(", ");
        System.out.print(arrList1.get(index2).getSurname());
        System.out.println("]");

    } else {
        System.out.println("The ID doesn't exist");
    }
}
```

In case 4, program ask user from which data structure he/she want to operate and print all the student’s information according to selection by traversing structure. I used infinite loop here again to prevent unexpected program endings. If user selects BST, another infinite loop stars and ask user which traversal he/she wants; preorder, inorder and postorder.

```
case "4" -> {
    /*
    If the user selects option 4, first ask him from which data structure he/she want to
    operate and print all the student's information according to selection by traversing
    related structure.
    */
    while (true) {
        int flag1 = 0;
        System.out.println("Which Data Structure do you want to print ?" +
                "\n 1 For List" +
                "\n 2 For Binary Search Tree" +
                "\n 3 For Hash");

        int answerForPrint = Integer.parseInt(input.nextLine());
        switch (answerForPrint) {
            case 1 -> {

                System.out.println(arrList1.toString());
                flag1++;
            }
            case 2 -> {
                while (true) {
                    int flag2 = 0;

                    System.out.println("Which traversal do you want to use to print Tree?" +
                            "\n 1 For PreOrder" +
                            "\n 2 For InOrder" +
                            "\n 3 For PostOrder");

                    int answerForTreeTraversal = Integer.parseInt(input.nextLine());
                    switch (answerForTreeTraversal) {
                        case 1 -> {
                            Tree1.printPreorder();
                            flag2++;
                        }
                        case 2 -> {
                            Tree1.printInorder();
                            flag2++;
                        }
                        case 3 -> {
                            Tree1.printPostorder();
                            flag2++;
                        }
                        default -> System.out.println("You Typed Wrong, Try again");
                    }
                    if(flag2 ==1 ) break;
                }
                flag1++;
            }
            case 3 -> {
                HashTable1.printHashTable();
                flag1++;
            }
            default -> System.out.println("You Typed Wrong, Try Again");
        }
        if(flag1 == 1) break;
    }
}
```

In case 5, program prints to the screen all the distinct “names” (use only names, not surnames) from your data structures. To do that,

I created a string array that has a size of a arraylist's array size and assign all names to it with for loop. I switch from array to list by java's pre-written list and array classes. First to show the difference it prints all names and after that takes that list and throw it to the HashSet. The reason of that HashSet can't have same object so, same ones will be removed. It prints the HashSet and shows the unique names.

```
case "5" -> {
    /*
    Print to the screen all the distinct "names" (use only names, not surnames) from your
    data structures. (i.e. not list duplicate names to the screen).
     */
    String[] names = new String[arrList1.size()];
    for (int i = 0; i < arrList1.size(); i++) {
        names[i] = arrList1.get(i).getName();
    }


    List<String> list = Arrays.asList(names);

    System.out.println("All Names : " + list);

    HashSet<String> set = new HashSet<>(list);

    System.out.println("No duplicates : " + set);

}
```

In case 6, program shows that how many of each "name" included in the data structures. To do that I thought that using TreeMap is the easiest way. I created a string array named tokens by collecting names from array by for loop. And then I iterated the tokens with string token by enhanced for. To prevent case sensitivity, first I lowercased the token and checks if this token is already in the TreeMap. Names are key and their count is value in this TreeMap. If there is none, adds the key to TreeMap but if not, just updates the value by increasing one. When this for loop is completed. It is ready to print the names and their counts.

```
case "6" -> {
    /*
    Show that how many of each “name” included in the data structures. (use only names,
    not surnames). For example you can print like following format:
    emre : 2
    sevcan : 3
    zeynep : 1
     */
    TreeMap<String, Integer> myMap = new TreeMap<>();

    //tokenize the input
    String[] tokens = new String[arrList1.size()];
    for (int i = 0; i < arrList1.size(); i++) {
        tokens[i] = arrList1.get(i).getName();
    }

    int count;

    for (String token : tokens) {
        String word = token.toLowerCase();

        if (myMap.containsKey(word)) //is word in map?
        {
            count = myMap.get(word);
            myMap.put(word, count + 1);
        } else
            myMap.put(word, 1); //add new word
    }

    System.out.print("Name Counts : ");
    System.out.println(myMap);

}
```

In case 7, project wants from us to print author’s information. When this case is activated it just prints my name surname and id to the console.

```
case "7" -> {
    /*
    7. Print the author(s) of the program (your student id, name and surname
     */
    System.out.println("\nThe author of this program is \n" +
            "Hüsamettin Demirtaş who studies at Celal Bayar University \n" +
            "on Computer Engineer Department\n");
}
```

RESULTS

```
0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
3
Please enter the ID that you want print
10
[10, Hüsamettin, Demirtaş]
```

```
0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
4
Which Data Structure do you want to print ?
 1 For List
 2 For Binary Search Tree
 3 For Hash
3
|0| -->
|1| -->
|2| -->
|3| --> [Emre Şatır 100] --> [Zeynep Çipiloğlu 5] --> [Muhammet Cinsdikici 15] -->
|4| --> [Didem Abidin 45] -->
|5| --> [Hüsamettin Demirtaş 55] -->
|6| --> [Hüsamettin Demirtaş 10] -->
|7| -->
|8| --> [Hüsamettin Demirtaş 9] --> [Emre Şatır 999] -->
|9| --> [Turan Göktuğ Altundoğan 40] -->
```

```
0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
4
Which Data Structure do you want to print ?
 1 For List
 2 For Binary Search Tree
 3 For Hash
2
Which traversal do you want to use to print Tree?
 1 For PreOrder
 2 For InOrder
 3 For PostOrder
2
[Zeynep Çipiloğlu 5] [Hüsamettin Demirtaş 9] [Hüsamettin Demirtaş 10] [Muhammet Cinsdikici 15] [Turan Göktuğ Altundoğan 40] [Didem Abidin 45] [Hüsamettin Demirtaş 55] [Emre Şatır 100] [Emre Şatır 999]
```

```
0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
4
Which Data Structure do you want to print ?
 1 For List
 2 For Binary Search Tree
 3 For Hash
1
[ [Hüsamettin Demirtaş 10] [Hüsamettin Demirtaş 9] [Emre Şatır 100] [Didem Abidin 45] [Zeynep Çipiloğlu 5] [Muhammet Cinsdikici 15] [Turan Göktuğ Altundoğan 40] [Emre Şatır 999] [Hüsamettin Demirtaş 55] ]
```

```
0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
5
All Names : [Hüsamettin, Hüsamettin, Emre, Didem, Zeynep, Muhammet, Turan Göktuğ, Emre, Hüsamettin]
No duplicates : [Hüsamettin, Turan Göktuğ, Muhammet, Zeynep, Didem, Emre]

0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
6
Name Counts : {didem=1, emre=2, hüsamettin=3, muhammet=1, turan göktuğ=1, zeynep=1}

0. Exit
1. Add student
2. Delete student
3. Find student
4. List all students
5. List distinct names
6. List name counts
7. About
Enter your selection:
7

The author of this program is
Hüsamettin Demirtaş 190315074 who studies at Celal Bayar University
on Computer Engineer Department
```