GIT Department of Computer Engineering

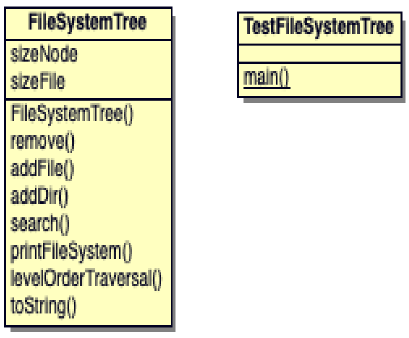
CSE 222/505 - Spring 2020
Homework 5
Report

Dilara KARAKAŞ

171044010

Q1

1.Class Diagram



2. Problem Solution Approach

A general tree is a tree that does not have the restriction that each node of a tree has at most two subtrees. So nodes in a general tree can have any number of subtrees.

Directories can have more than one child. These can be directory or file. File cannot have children. So the arraylist is always empty, and the arraylist.size () function always returns 0.

The addDir function takes the String parameter. If the parameter contains the file name and finds a directory that does not exist among the given ones, it throws an exception.

The addFile function takes the parameter String. It throws an exception if there is more than one file name in the parameter or if there are no file names and there is a directory name that does not exist among the given ones.

The remove function takes a string as a parameter. It throws an exception if there is no directory or file name in the received parameter.

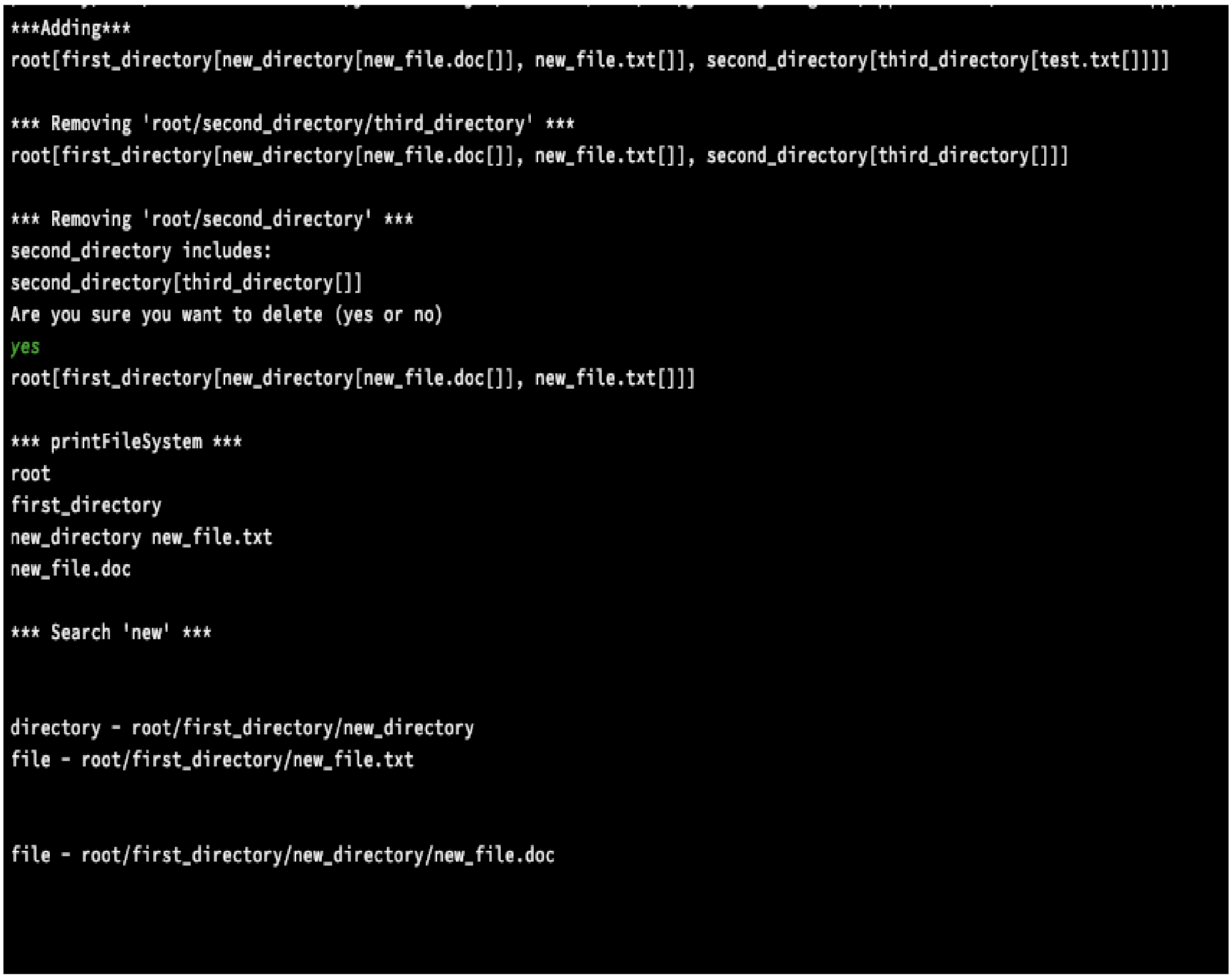
The search function takes a string as a parameter. This parameter is the word to be searched. This makes the toString2 () method in the FileNode class. Checks if all children have words

Returns the path and type (directory or file) we added in the addDir () and addFile () functions, if any.

3. Test Cases

```
tree.addDir("root/first_directory");
tree.addDir("root/second_directory");
tree.addDir("root/first_directory/new_directory");
tree.addFile("root/first_directory/new_file.txt");
tree.addDir("root/second_directory/third_directory");
tree.addFile("root/second_directory/third_directory/test.txt");
tree.addFile("root/first_directory/new_directory/new_file.doc");
*** Removing 'root/second_directory/third_directory' ***"
tree.remove("root/second_directory/third_directory/test.txt");
*** Removing 'root/second_directory' ***
tree.remove("root/second_directory");

** printFileSystem ***
tree.printFileSystem();
*** Search 'new' ***;
tree.search("new");
```

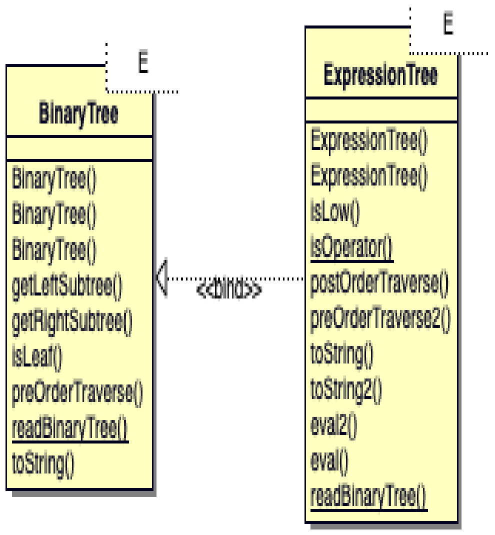
4. Results



```
***Adding***
root[first_directory[new_directory[new_file.doc[]], new_file.txt[]], second_directory[third_directory[test.txt[]]]]

*** Removing 'root/second_directory/third_directory' ***
root[first_directory[new_directory[new_file.doc[]], new_file.txt[]], second_directory[third_directory[]]]

*** Removing 'root/second_directory' ***
second_directory includes:
second_directory[third_directory[]]
Are you sure you want to delete (yes or no)
yes
root[first_directory[new_directory[new_file.doc[]], new_file.txt[]]]

*** printFileSystem ***
root
first_directory
new_directory new_file.txt
new_file.doc

*** Search 'new' ***


directory - root/first_directory/new_directory
file - root/first_directory/new_file.txt


file - root/first_directory/new_directory/new_file.doc
```

Q2

1. Class Diagram



2.Problem Solutions Approach

Just as for a linked list, a node consists of a data part and links (references) to successor nodes. So that we can store any kind of data in a tree node, we will make the data part a refer- ence of type E. Instead of having a single link (reference) to a successor node as in a list, a binary tree node must have links (references) to both its left and right subtrees.

Class Node<E> is nested within class BinaryTree<E>. Note that it is declared **protected** and its data fields are all **protected**. Later, we will use the BinaryTree<E> and Node<E> classes as superclasses. By declaring the nested Node<E> class and its data fields protected, we make them accessible in the subclasses of BinaryTree<E> and Node<E>.

The constructor for class Node<E> creates a leaf node (both left and right are **null**). The toString method for the class just displays the data part of the node.

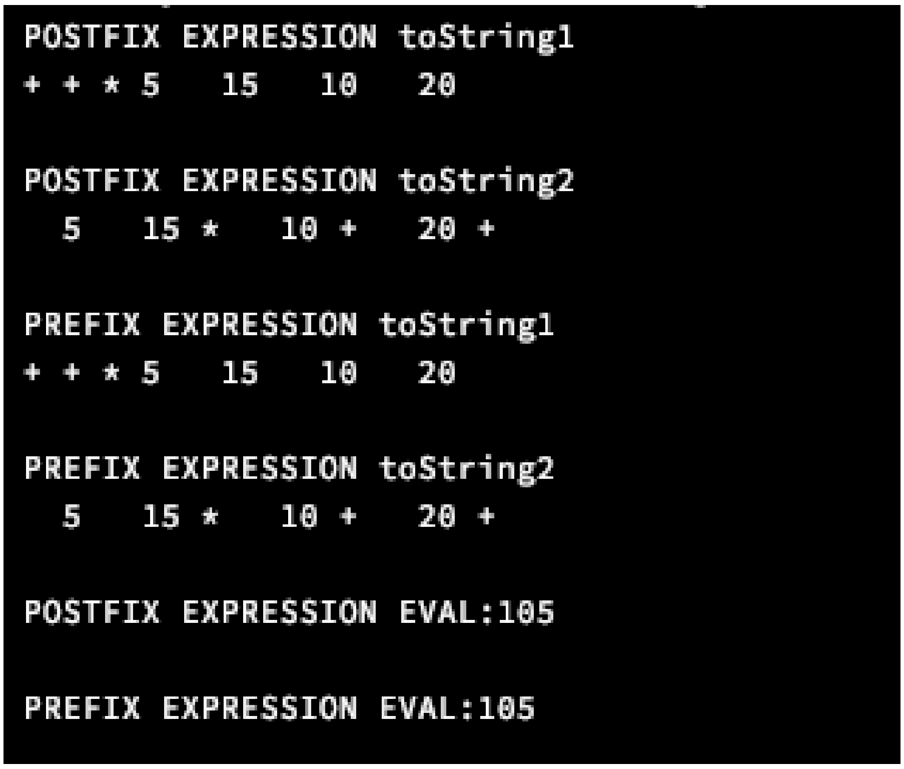
Both the BinaryTree<E> class and the Node<E> class are declared to implement the Serializable interface. The Serializable interface defines no methods; it is used to provide a marker for classes that can be written to a binary file using the ObjectOutputStream and read using the ObjectInputStream.

Expression Tree distinguishes prefix and postfix by looking at the first character in the string. The first character is prefix if the aritmetic operator, otherwise postfix. All the remaining functions carry out their operations as prefix and postfix.

3.Test Cases

```
ExpressionTree<String> tree = new ExpressionTree<>("10 5 15 * + 20 +");

System.out.println("POSTFIX EXPRESSION toString1");

System.out.println(tree.toString());

System.out.println("\nPOSTFIX EXPRESSION toString2");

 System.out.println(tree.toString2());

ExpressionTree<String> tree2 = new ExpressionTree<>("+ + 10 * 15 5 20");

System.out.println("\nPREFIX EXPRESSION toString1");

System.out.println(tree2.toString());

System.out.println("\nPREFIX EXPRESSION toString2");

System.out.println(tree2.toString2());

System.out.println("\nPOSTFIX EXPRESSION EVAL:" + tree.eval());

System.out.println("\nPREFIX EXPRESSION EVAL:" + tree2.eval());
```

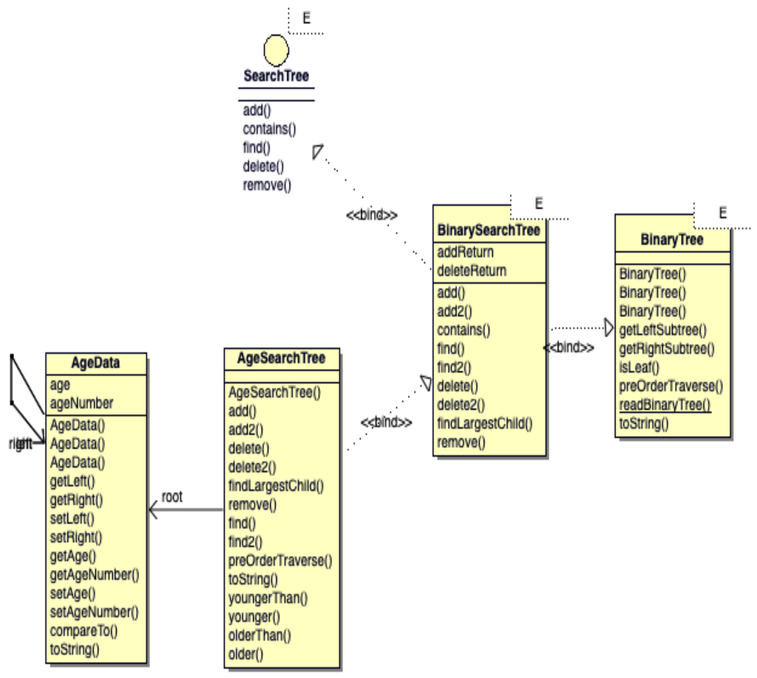
4. Results



```
POSTFIX EXPRESSION toString1
+ + * 5   15   10   20

POSTFIX EXPRESSION toString2
  5   15 *   10 +   20 +

PREFIX EXPRESSION toString1
+ + * 5   15   10   20

PREFIX EXPRESSION toString2
  5   15 *   10 +   20 +

POSTFIX EXPRESSION EVAL:105

PREFIX EXPRESSION EVAL:105
```

Q3

1.Class Diagram



2.Problem Solution Approach

In AgeSearchTree, which is extended from BinarySearchTree, if there is a different element in the add () function, we increase the number of people by one. If there is no one, we create a new node.

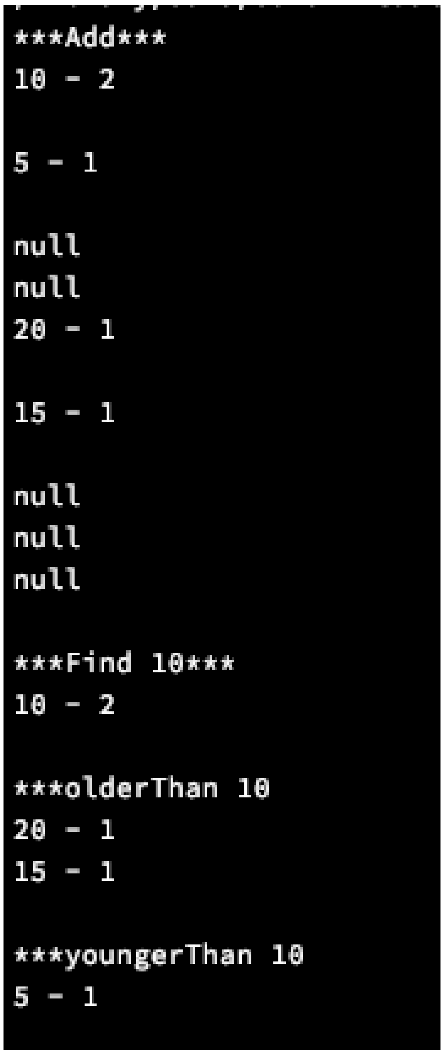
If there is the same element in the remove () function in the same way, we reduce the number of people by one. We delete the node when it is zero.

The find () method is changing the arm according to the age of the searched age from the age of the node. If the seeker is older, he goes to the left boy, if he is younger to the right boy.It works the same as youngerThan () and olderThan().

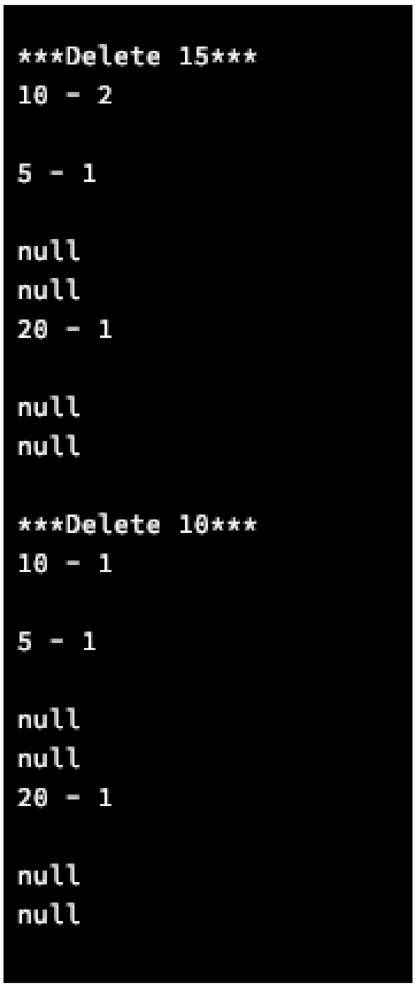
3.Test Cases

```
AgeSearchTree tree = new AgeSearchTree();
System.out.println("***Add***");
tree.add(new AgeData(10));
tree.add(new AgeData(20));
tree.add(new AgeData(5));
tree.add(new AgeData(15));
tree.add(new AgeData(10));
System.out.println(tree.toString());
System.out.println("***Find 10***");
System.out.println(tree.find(new AgeData(10)).toString());
System.out.println("***olderThan 10");
System.out.println(tree.olderThan(10));
System.out.println("***youngerThan 10");
System.out.println(tree.youngerThan(10));
System.out.println("***Delete 15***");
tree.delete(new AgeData(15));
System.out.println(tree.toString());
System.out.println("***Delete 10***");
tree.remove(new AgeData(10));
System.out.println(tree.toString());
```

4.Results



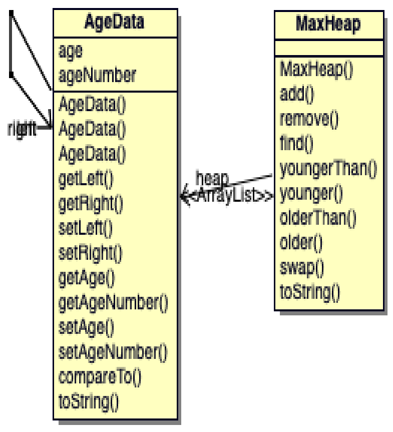
```
***Add***
10 - 2

5 - 1

null
null
20 - 1

15 - 1

null
null
null

***Find 10***
10 - 2

***olderThan 10
20 - 1
15 - 1

***youngerThan 10
5 - 1
```



```
***Delete 15***
10 - 2

5 - 1

null
null
20 - 1

null
null

***Delete 10***
10 - 1

5 - 1

null
null
20 - 1

null
null
```

Q4

1.Class Diagram



2.Problem Solution Approach

In AgeSearchTree, which is extended from BinarySearchTree, if there is a different element in the add () function, we increase the number of people by one(Check if any changes in heap is needed since the key is “number of people”). If there is no one, we create a new node.

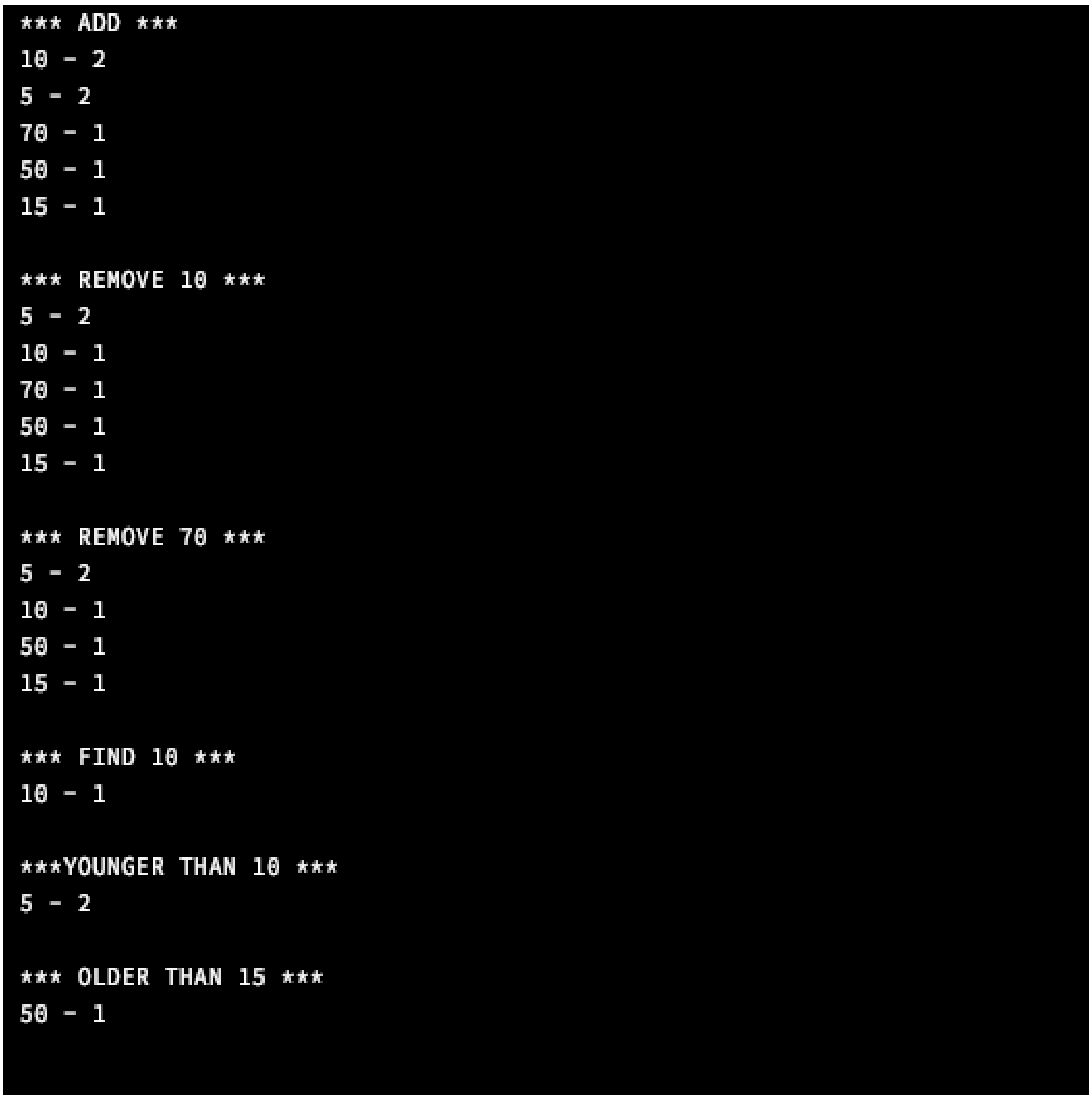
If there is the same element in the remove () function in the same way, we reduce the number of people by one (Check if any changes in heap is needed since the key is “number of people”). We delete the node when it is zero.

The find () method is changing the arm according to the age of the searched age from the age of the node. If the seeker is older, he goes to the left boy, if he is younger to the right boy.It works the same as youngerThan () and olderThan().(leftChild to (2 * parent) + 1 and rightChild to leftChild + 1.)

3.Test Cases

```
MaxHeap heap = new MaxHeap();

System.out.println("*** ADD ***");

heap.add(new AgeData(10));

heap.add(new AgeData(5));

heap.add(new AgeData(70));

heap.add(new AgeData(10));

heap.add(new AgeData(50));

heap.add(new AgeData(5));

heap.add(new AgeData(15));

System.out.println(heap.toString());

System.out.println("*** REMOVE 10 ***");

heap.remove(new AgeData(10));

System.out.println(heap.toString());

System.out.println("*** REMOVE 70 ***");

heap.remove(new AgeData(70));

System.out.println(heap.toString());

System.out.println("*** FIND 10 ***\n" + heap.find(10).toString());

System.out.println("***YOUNGER THAN 10 ***\n" + heap.youngerThan(10));

System.out.println("*** OLDER THAN 15 ***\n" + heap.olderThan(15));
```

4.Results



```
*** ADD ***
10 - 2
5 - 2
70 - 1
50 - 1
15 - 1

*** REMOVE 10 ***
5 - 2
10 - 1
70 - 1
50 - 1
15 - 1

*** REMOVE 70 ***
5 - 2
10 - 1
50 - 1
15 - 1

*** FIND 10 ***
10 - 1

***YOUNGER THAN 10 ***
5 - 2

*** OLDER THAN 15 ***
50 - 1
```