Software Testing HW#2

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Monday 3rd June, 2024

Dept. of Computer Engineering

University of Tehran

Contents

1 Question 1

1.1 Wrtie Tests

We have written multiple scenarios for each method of the target class ”Gedcom_Service”.

1.1.1 Read And Parse File

For these method since it has too many nested if conditions, we consider 19 test cases and different scenarios as follow:

- Test 1: Pass an empty file to the method.

```
@Test
public void readAndParseFileEmptyFile() throws Exception{
    String path = "src/test/java/edu/stevens/ssw555/test1.ged";
    Gedcom_Service gcd = new Gedcom_Service();
    gcd.readAndParseFile(path);
    assertTrue(gcd.dupInd.isEmpty(), "Expected dupInd to be empty");
    assertTrue(gcd.dupFam.isEmpty(), "Expected dupFam to be empty");
}
```

- Test 2: Pass non existance file’s path.

```
@Test
public void readAndParseFileNoExistanceFIle(){
    Gedcom_Service gcd = new Gedcom_Service();
    String nonExistentFilePath = "path/to/nonexistent/file.ged";
    assertThrows(IOException.class, () -> {
        gcd.readAndParseFile(nonExistentFilePath);
    }, "Expected readAndParseFile to throw IOException due to non-
       ↪ existent file");
}
```

- Test 3: Write a test file (test2.ged) and check the individuals would be well-initialized.

```
@Test
public void readAndParseFileWithSingleInd() throws Exception{
    String path = "src/test/java/edu/stevens/ssw555/test2.ged";
    Gedcom_Service gcd = new Gedcom_Service();
    gcd.readAndParseFile(path);
    assertTrue(gcd.dupInd.isEmpty(), "dupInd should be empty");
    assertEquals(1, gcd.individuals.size(), "dupInd should have one
       ↪ individual");
    Individual individual = gcd.individuals.get("@I1@");
    assertNotNull(individual, "The individual entry should not be null");
    assertEquals("@I1@", individual.getId(), "Check the individual's ID")
       ↪ ;
```

```
        assertEquals("John Doe", individual.getName(), "Check the individual's name");
        assertEquals("M", individual.getSex(), "Check the individual's sex");
        assertEquals("@F1@", individual.spouseOf, "Check the individual's sex ");
        assertEquals("@F2@", individual.getChildOf(), "Check the individual's  sex");

        assertEquals("06/15/1975", individual.getBirth(), "Check the individual's birth date and place");
        assertEquals("08/20/2050", individual.getDeath(), "Check the individual's death date and place");
        assertEquals(1, gcd.individuals.size(), "The size of individuals should be 1");
    }
```

- Test 4: Check if families would be well-initialized.

```
    @Test
    public void readAndParseFileWithSingleFamily() throws Exception{
        String path = "src/test/java/edu/stevens/ssw555/test3.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertTrue(gcd.dupFam.isEmpty(), "dupFam should be empty");
        assertEquals(1, gcd.families.size(), "dupInd should have one individual");
        Family family = gcd.families.get("@F1@");
        assertNotNull(family, "The individual entry should not be null");
        assertEquals("@F1@", family.getId(), "Check the family's ID");
        assertEquals("@I1@", family.getHusb(), "Chekc the family's Husband");
        assertEquals("@I2@", family.getWife(), "Chekc the family's Wife");
        ArrayList<String> children = new ArrayList<>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(children, family.child, "check the child array");
        assertEquals("06/12/1995", family.getMarriage(), "Check the marriages  date");
        assertEquals("08/15/2020", family.getDivorce(), "check the divorce date");
    }
```

- Test 5: Check the branch if in INDI works correctly.

```
    @Test
    public void readAndParseFileWrongIND() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test5.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
```

```
        assertEquals(gcd.individuals.size(), 0, "Size of individuals is 0");
    }
```

- Test 6: Check FAM's branch works correct.

```
    @Test
    public void readAndParseFileWrongFAM() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test6.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.families.size(), 0, "Size of individuals is 0");
    }
```

- Test 7: Check NAME's brach works correct.

```
    @Test
    public void readAndParseFileWithNullName() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test7.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.name, "Expecting to be null");
    }
```

- Test 8: Check SEX's branch works correct.

```
    @Test
    public void readAndParseFileWithNullSex() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test8.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.sex, "Expecting to be null");
    }
```

- Test 9: Check FAMS's branch works quite well.

```
    @Test
    public void readAndParseFileWithNullFAMS() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test9.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.spouseOf, "Expecting to be null");
    }
```

- Test 10: Check FAMC's branch works well.

```
    @Test
    public void readAndParseFileWithNullFAMC() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test10.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.getChildOf(), "Expecting to be null");
    }
```

- Test 11: Check BIRT's branch works.

```
    @Test
    public void readAndParseFileWithNullBIRT() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test11.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.getBirth(), "Expecting to be null");
    }
```

- Test 12: Check DEAT's branch:

```
    @Test
    public void readAndParseFileWithNullDEAT() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test12.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 0");
        Individual ind = gcd.individuals.get("@I1@");
        assertNull(ind.getDeath(), "Expecting to be null");
    }
```

- Test 13:Check if duplicate individual would be detected.

```
    @Test
    public void readAndParseFileDupInd() throws Exception{
        String path = "src/test/java/edu/stevens/ssw555/test4.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.dupInd.isEmpty(), "dupInd should not be empty");
        assertTrue(gcd.dupFam.isEmpty(), "dupFam should be empty");
        assertEquals(gcd.individuals.size(), 1, "Size of individuals is 1");
        assertEquals(gcd.dupInd.size(), 1 , "dupInd size should be 1");
        assertFalse(gcd.individuals.isEmpty(), "Individuals must be consist
            ↪ one person");
```

```
        Individual ind = gcd.individuals.get("@I1@");
        assertNotNull(ind, "ind must not be NULL");
        assertEquals(ind.getName(), "John Doe", "Name must be matched");
        assertEquals(ind.getBirth(), "01/1/1990", "Date of birth must be
            ↪ matched");
        assertEquals(ind.getSex(), "M", "Sex would be matched");
        ind = gcd.dupInd.get(0);
        assertNotNull(ind, "Must not be null");
        assertEquals(ind.getName(), "Jane Smith", "Second Person with
            ↪ duplicate ID's name must matched");
        assertEquals(ind.getSex(), "F", "Second Person must be female");
        assertEquals(ind.getBirth(), "02/1/1990", "Second person's birth day
            ↪ must matched");
    }
```

- Test 14: Check HUSB's branch.

```
    @Test
    public void readAndParseFileWrongHUSB() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test13.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        Family fam = gcd.families.get("@F1@");
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertNull(fam.getHusb(), "Husband must be null");
        assertEquals(fam.getMarriage(), "06/12/1995", "mariage data must be
            ↪ matched");
        assertEquals(fam.getDivorce(), "08/" +  "15" + "/" + "2020", "divord
            ↪ data must match");
        assertEquals(fam.getWife(), "@I2@", "Wife must match");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

- Test 15: Check WIFE's branch.

```
    @Test
    public void readAndParseFileWrongWIFE() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test14.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        Family fam = gcd.families.get("@F1@");
```

```
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I1@", "Husband must match");
        assertEquals(fam.getMarriage(), "06/12/1995", "mariage data must be
            ↪ matched");
        assertEquals(fam.getDivorce(), "08/" +  "15" + "/" + "2020", "divord
            ↪ data must match");
        assertNull(fam.getWife(), "Wife must be null");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

- Test 16: Check CHILD's branch:

```
    @Test
    public void readAndParseFileWrongCHILD() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test15.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        Family fam = gcd.families.get("@F1@");
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I1@", "Husband must match");
        assertEquals(fam.getMarriage(), "06/12/1995", "mariage data must be
            ↪ matched");
        assertEquals(fam.getDivorce(), "08/" +  "15" + "/" + "2020", "divord
            ↪ data must match");
        assertEquals(fam.getWife(), "@I2@", "Wife must match");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

- Test 17: Check MARR's branch.

```
    @Test
    public void readAndParseFileWrongMARR() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test16.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        Family fam = gcd.families.get("@F1@");
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I1@", "Husband must match");
        assertNull(fam.getMarriage(), "mariage data must be null");
```

```
        assertEquals(fam.getDivorce(), "08/" +  "15" + "/" + "2020", "divord
            ↪ data must match");
        assertEquals(fam.getWife(), "@I2@", "Wife must match");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

- Test 18: Check DIV's branch

```
    @Test
    public void readAndParseFileWrongDIV() throws Exception {
        String path = "src/test/java/edu/stevens/ssw555/test17.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);
        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        Family fam = gcd.families.get("@F1@");
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I1@", "Husband must match");
        assertEquals(fam.getMarriage(), "06/12/1995", "mariage data must be
            ↪ matched");
        assertNull(fam.getDivorce(), "divord data must be null");
        assertEquals(fam.getWife(), "@I2@", "Wife must match");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

- Test 19: If duplicate families were in the file, will the code correctly add it to duplicate families array?

```
   @Test
    public void readAndParseFileDupFam() throws Exception{
        String path = "src/test/java/edu/stevens/ssw555/test18.ged";
        Gedcom_Service gcd = new Gedcom_Service();
        gcd.readAndParseFile(path);

        assertFalse(gcd.families.isEmpty(), "Families is not empty");
        assertEquals(gcd.families.size(), 1, "Families have 1 in the list");
        assertEquals(gcd.dupFam.size(), 1, "One duplicate");

        Family fam = gcd.families.get("@F1@");
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I1@", "Husband must match");
        assertEquals(fam.getMarriage(), "06/12/1995", "mariage data must be
            ↪ matched");
```

```
        assertEquals(fam.getDivorce(), "08/" +  "15" + "/" + "2020" ,"divord
            ↪ data must match");
        assertEquals(fam.getWife(), "@I2@", "Wife must match");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I3@");
        children.add("@I4@");
        assertEquals(fam.getChild(), children, "Childeren must matched");

        fam = gcd.dupFam.get(0);
        assertEquals(fam.getId(), "@F1@", "Id must matched");
        assertEquals(fam.getHusb(), "@I4@", "Husband must match");
        assertEquals(fam.getMarriage(), "06/15/1995", "mariage data must be
            ↪ matched");
        assertEquals(fam.getDivorce(), "08/" +  "20" + "/" + "2020" ,"divord
            ↪ data must match");
        assertEquals(fam.getWife(), "@I5@", "Wife must match");
        children = new ArrayList<String>();
        children.add("@I6@");
        children.add("@I7@");
        assertEquals(fam.getChild(), children, "Childeren must matched");
    }
```

Note that for each scenarios we consider to build a ged files where can be seen in the ”main/java/test/edu/stevens/ssw555” directory.

1.1.2 Create Output File

We have written just one test for this since the code recursively call himself in a particular branch we avoid to test this.
For this purpose, we have utilized the ”TempDir” from junit library for creating temp file. See if the file will be created with this method, we have written a test:

```
@Test
public void testCreateOutputFileWithValidPath() throws Exception {
    String validPath = tempDir.toString();
    InputStream sysInBackup = System.in;
    ByteArrayInputStream in = new ByteArrayInputStream(validPath.getBytes());
    System.setIn(in);
    Gedcom_Service.createOutputFile();
    System.setIn(sysInBackup);
    Path outputPath = Paths.get(Gedcom_Service.fileName);
    assertTrue(Files.exists(outputPath), "Expected output file to be created");
    Files.deleteIfExists(outputPath);
}
```

1.1.3 Write To File

We have implemented 5 test cases for this method with different scenarios:

- Test 1: Is our content would be written to the file?

```
    @Test
    public void testWriteToFile() throws Exception {
        Gedcom_Service.fileName = tempDir.resolve("GedcomService_output.txt")
            ↪ .toString();
        String content = "Test output content";
        Gedcom_Service.writeToFile(content);
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");
        String fileContent = new String(Files.readAllBytes(outputPath));
        assertTrue(fileContent.contains(content), "Expected content to be
            ↪ written to the file");
    }
```

- Test 2: What if our content has more than one line?

```
    @Test
    public void testWriteMultipleLinesToFile() throws Exception {
        String content1 = "First line of content";
        String content2 = "Second line of content";
        Gedcom_Service.fileName = tempDir.resolve("GedcomService_output.txt")
            ↪ .toString();
        Gedcom_Service.writeToFile(content1);
        Gedcom_Service.writeToFile(content2);
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");
        String fileContent = new String(Files.readAllBytes(outputPath));
        assertTrue(fileContent.contains(content1), "Expected first line of
            ↪ content to be written to the file");
        assertTrue(fileContent.contains(content2), "Expected second line of
            ↪ content to be written to the file");
    }
```

- Test 3: Does it write an empty content?

```
    @Test
    public void testWriteToFileWithEmptyContent() throws Exception {
        String content = "";
        Gedcom_Service.fileName = tempDir.resolve("GedcomService_output.txt")
            ↪ .toString();
        Gedcom_Service.writeToFile(content);
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");
        String fileContent = new String(Files.readAllBytes(outputPath));
```

```
        assertTrue(fileContent.contains(content), "Expected empty content to
            ↪ be written to the file");
    }
```

- Test 4: Will this write in a file even file's path is null?

```
@Test
public void testWriteToFileWithNullFileName() {
    Gedcom_Service.fileName = null;
    assertThrows(NullPointerException.class, () -> {
        Gedcom_Service.writeToFile("This should fail");
    });
}
```

This test case must throw an exception.

1.1.4 Test Birth Before Death

For this method we have consider multiple scenarios as well:

- Test 1: Tests the scenario where the birth date is before the death date, expecting no errors.

```
    @Test
    public void testBirthBeforeDeathValidDates() throws Exception {
        Individual indi = new Individual("@I1@");
        indi.setBirth("01/01/1980");
        indi.setDeath("01/01/2000");
        indi.setId("@I1@");
        indi.setName("John Doe");
        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(indi.getId(), indi);
        Gedcom_Service.birthBeforeDeath(individuals);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ valid dates");
        }
    }
```

- Test 2: Tests the scenario where the birth date is after the death date, expecting an error message.

```
    @Test
    public void testBirthBeforeDeathInvalidDates() throws Exception {
        Individual indi = new Individual("@I1@");
        indi.setBirth("01/01/2000");
        indi.setDeath("01/01/1980");
```

```
        indi.setId("@I1@");
        indi.setName("John Doe");
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        Files.createFile(outputPath);
        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(indi.getId(), indi);

        Gedcom_Service.birthBeforeDeath(individuals);
        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR:INDIVIDUAL: User Story US03: Birth
            ↪ Before Death";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ invalid dates");
    }
```

- Test 3: Tests the scenario where the death date is null, expecting no errors.

```
    @Test
    public void testBirthBeforeDeathNullDeathDate() throws Exception {
        Individual indi = new Individual("@I1@");
        indi.setBirth("01/01/1980");
        indi.setId("@I1@");
        indi.setName("John Doe");
        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(indi.getId(), indi);

        Gedcom_Service.birthBeforeDeath(individuals);
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪  valid dates");
        }
    }
```

- Tests the scenario where the birth date is after the death date, expecting the error message to be written to the file.

```
    @Test
    public void testBirthBeforeDeathFileOutput() throws Exception {
        Individual indi = new Individual("@I1@");
        indi.setBirth("01/01/2000");
        indi.setDeath("01/01/1980");
        indi.setId("@I1@");
        indi.setName("John Doe");
        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(indi.getId(), indi);
        Gedcom_Service.birthBeforeDeath(individuals);
```

```
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");
        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR:INDIVIDUAL: User Story US03: Birth
            ↪ Before Death";
        assertTrue(fileContent.contains(expectedOutput), "Expected error
            ↪ message to be written to the file");
    }
```

1.1.5 Marriage Before Divorce

Now let's proceed into creating some tests for "marriageBeforeDivorce" methods.The test scenarios are as follow:

- Test 1: Tests the scenario where the marriage date is before the divorce date, expecting no errors.

```
    @Test
    public void testMarriageBeforeDivorceValidDates() throws Exception {
        Individual husb = new Individual("@I1@");
        husb.setId("@I1@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I1@");
        fam.setWife("@I2@");
        fam.setMarriage("01/01/1980");
        fam.setDivorce("01/01/2000");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        Files.createFile(outputPath);

        Gedcom_Service.Marriagebeforedivorce(individuals, families);

        String fileContent = new String(Files.readAllBytes(outputPath));
        assertFalse(fileContent.contains("ERROR"), "Expected no error for
            ↪ valid dates");
```

```
    }
```

- Test 2: Tests the scenario where the marriage date is after the divorce date, expecting an error message.

```
    @Test
    public void testMarriageBeforeDivorceInvalidDates() throws Exception {
        Individual husb = new Individual("@I1@");
        husb.setId("@I1@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I1@");
        fam.setWife("@I2@");
        fam.setMarriage("01/01/2000");
        fam.setDivorce("01/01/1980");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);

        Gedcom_Service.Marriagebeforedivorce(individuals, families);

        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR:FAMILY: User Story US04: Marriage
            ↪ Before Divorce";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ invalid dates");
    }
```

- Test 3: Tests the scenario where the divorce date is null, expecting no errors.

```
    @Test
    public void testMarriageBeforeDivorceNullDivorceDate() throws Exception {
        Individual husb = new Individual("@I1@");
        husb.setId("@I1@");
        husb.setName("John Doe");
```

```
        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I1@");
        fam.setWife("@I2@");
        fam.setMarriage("01/01/1980");
        fam.setDivorce(null);

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        Files.createFile(outputPath);

        Gedcom_Service.Marriagebeforedivorce(individuals, families);

        String fileContent = new String(Files.readAllBytes(outputPath));
        assertFalse(fileContent.contains("ERROR"), "Expected no error for
            ↪ null divorce date");
    }
```

- Test 4: Verifies the content of the output file for invalid dates.

```
    @Test
    public void testMarriageBeforeDivorceFileOutput() throws Exception {
        Individual husb = new Individual("@I1@");
        husb.setId("@I1@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I1@");
        fam.setWife("@I2@");
        fam.setMarriage("01/01/2000");
        fam.setDivorce("01/01/1980");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husb.getId(), husb);
```

```
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);

        Gedcom_Service.Marriagebeforedivorce(individuals, families);

        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR:FAMILY: User Story US04: Marriage
            ↪ Before Divorce";
        assertTrue(fileContent.contains(expectedOutput), "Expected error
            ↪ message to be written to the file");
    }
```

- Test 5: Tests the scenario where an invalid date format is provided to trigger a ParseException.

```
    @Test
    public void testMarriageBeforeDivorceParseException() throws Exception {
        Individual husb = new Individual("@I1@");
        husb.setId("@I1@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I1@");
        fam.setWife("@I2@");
        fam.setMarriage("invalid-date");
        fam.setDivorce("01/01/2000");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        ByteArrayOutputStream errContent = new ByteArrayOutputStream();
        System.setErr(new PrintStream(errContent));

        Path outputPath = Paths.get(Gedcom_Service.fileName);
```

```
28
29          Gedcom_Service.Marriagebeforedivorce(individuals, families);
30
31          String errorOutput = errContent.toString();
32          assertTrue(errorOutput.contains("java.text.ParseException"), "
               ↪ Expected ParseException to be printed to stderr");
33          System.setErr(System.err);
34      }
```

1.1.6 Birth Before Marriage

Write some scenarios to test ”birthBeforeMarriageOfParent” method.

- Test 1: Tests the scenario where all dates are valid and no errors are expected.

```
1       @Test
2       public void testBirthBeforeMarriageOfParentValidDates() throws Exception
           ↪ {
3           Individual child = new Individual("@I1@");
4           child.setId("@I1@");
5           child.setName("Child Doe");
6           child.setBirth("01/01/1985");
7
8           Individual husb = new Individual("@I2@");
9           husb.setId("@I2@");
10          husb.setName("John Doe");
11
12          Individual wife = new Individual("@I3@");
13          wife.setId("@I3@");
14          wife.setName("Jane Smith");
15
16          Family fam = new Family("@F1@");
17          fam.setHusb("@I2@");
18          fam.setWife("@I3@");
19          fam.setMarriage("01/01/1980");
20          fam.setDivorce("01/01/1990");
21          fam.setChild(new ArrayList<>(Arrays.asList("@I1@")));
22
23          HashMap<String, Individual> individuals = new HashMap<>();
24          individuals.put(child.getId(), child);
25          individuals.put(husb.getId(), husb);
26          individuals.put(wife.getId(), wife);
27
28          HashMap<String, Family> families = new HashMap<>();
29          families.put(fam.getId(), fam);
30
31          Gedcom_Service.birthbeforemarriageofparent(individuals, families);
32
```

```
        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ valid dates");
        }
    }
```

- Test 2: Tests the scenario where a child is born before the parents' marriage date, expecting an error message.

```
    @Test
    public void testBirthBeforeMarriageOfParentBirthBeforeMarriage() throws
        ↪ Exception {
        Individual child = new Individual("@I1@");
        child.setId("@I1@");
        child.setName("Child Doe");
        child.setBirth("01/01/1975");

        Individual husb = new Individual("@I2@");
        husb.setId("@I2@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I3@");
        wife.setId("@I3@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I2@");
        fam.setWife("@I3@");
        fam.setMarriage("01/01/1980");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I1@");
        fam.setChild(children);

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child.getId(), child);
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        Gedcom_Service.birthbeforemarriageofparent(individuals, families);

        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");
        String fileContent = new String(Files.readAllBytes(outputPath));
```

```
        String expectedOutput = "ERROR: User Story US08: Birth Before
            ↪ Marriage Date";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ birth before marriage date");
    }
```

- Test 3: Tests the scenario where a child is born after the parents' divorce date, expecting an error message.

```
    @Test
    public void testBirthBeforeMarriageOfParentBirthAfterDivorce() throws
        ↪ Exception {
        Individual child = new Individual("@I1@");
        child.setId("@I1@");
        child.setName("Child Doe");
        child.setBirth("01/01/1995");

        Individual husb = new Individual("@I2@");
        husb.setId("@I2@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I3@");
        wife.setId("@I3@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I2@");
        fam.setWife("@I3@");
        fam.setMarriage("01/01/1980");
        fam.setDivorce("01/01/1990");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I1@");
        fam.setChild(children);

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child.getId(), child);
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);

        Gedcom_Service.birthbeforemarriageofparent(individuals, families);

        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

```

```
        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR: User Story US08: Birth After Divorce
            ↪ Date";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ birth after divorce date");
    }
```

- Test 4: Verifies the content of the output file for the case where a child is born before the parents' marriage date.

```
    @Test
     public void testBirthBeforeMarriageOfParentFileOutput() throws Exception
         ↪ {
        Individual child = new Individual("@I1@");
        child.setId("@I1@");
        child.setName("Child Doe");
        child.setBirth("01/01/1975");

        Individual husb = new Individual("@I2@");
        husb.setId("@I2@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I3@");
        wife.setId("@I3@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I2@");
        fam.setWife("@I3@");
        fam.setMarriage("01/01/1980");
        fam.setDivorce("01/01/1990");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I1@");
        fam.setChild(children);

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child.getId(), child);
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Path outputPath = Paths.get(Gedcom_Service.fileName);

        Gedcom_Service.birthbeforemarriageofparent(individuals, families);

```

```
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR: User Story US08: Birth Before
            ↪ Marriage Date";
        assertTrue(fileContent.contains(expectedOutput), "Expected error
            ↪ message to be written to the file for birth before marriage date
            ↪ ");
    }
```

- Test 5: Tests the scenario where an invalid date format is provided, expecting a ParseException.

```
    @Test
    public void testBirthBeforeMarriageOfParentParseException() throws
        ↪ Exception {
        Individual child = new Individual("@I1@");
        child.setId("@I1@");
        child.setName("Child Doe");
        child.setBirth("invalid-date");

        Individual husb = new Individual("@I2@");
        husb.setId("@I2@");
        husb.setName("John Doe");

        Individual wife = new Individual("@I3@");
        wife.setId("@I3@");
        wife.setName("Jane Smith");

        Family fam = new Family("@F1@");
        fam.setHusb("@I2@");
        fam.setWife("@I3@");
        fam.setMarriage("01/01/1980");
        ArrayList<String> children = new ArrayList<String>();
        children.add("@I1@");
        fam.setChild(children);

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child.getId(), child);
        individuals.put(husb.getId(), husb);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        ByteArrayOutputStream errContent = new ByteArrayOutputStream();
        System.setErr(new PrintStream(errContent));

```

```
        Gedcom_Service.birthbeforemarriageofparent(individuals, families);

        String errorOutput = errContent.toString();
        assertTrue(errorOutput.contains("java.text.ParseException"), "
            ↪ Expected ParseException to be printed to stderr");

        System.setErr(System.err);
    }
```

1.1.7 Male Last Name

Through different scenarios, we encounter a bug in this method that different last names wouldn't correctly behave like we want.The test scenarios for this:

- Test 1: Tests the scenario where all male family members have the same last name, expecting no errors.

```
    @Test
    public void testMaleLastNameAllSame() throws Exception {
        Individual child1 = new Individual("@I1@");
        child1.setId("@I1@");
        child1.setName("John Doe");
        child1.setSex("male");

        Individual child2 = new Individual("@I2@");
        child2.setId("@I2@");
        child2.setName("Mike Doe");
        child2.setSex("male");

        Family fam = new Family("@F1@");
        fam.setChild(new ArrayList<>(Arrays.asList("@I1@", "@I2@")));

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child1.getId(), child1);
        individuals.put(child2.getId(), child2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Gedcom_Service.Malelastname(families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪  same last names");
        }
    }
```

- Test 2: Tests the scenario where male family members have different last names, expecting an error message.(This test will fail cause the code has bug)

```
    @Test
     public void testMaleLastNameDifferent() throws Exception {
         Individual child1 = new Individual("@I1@");
         child1.setId("@I1@");
         child1.setName("John Doe");
         child1.setSex("male");

         Individual child2 = new Individual("@I2@");
         child2.setId("@I2@");
         child2.setName("Mike Smith");
         child2.setSex("male");

         Family fam = new Family("@F1@");
         fam.setChild(new ArrayList<>(Arrays.asList("@I1@", "@I2@")));

         HashMap<String, Individual> individuals = new HashMap<>();
         individuals.put(child1.getId(), child1);
         individuals.put(child2.getId(), child2);

         HashMap<String, Family> families = new HashMap<>();
         families.put(fam.getId(), fam);

         Gedcom_Service.Malelastname(families);

         Path outputPath = Paths.get(Gedcom_Service.fileName);
         assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

         String fileContent = new String(Files.readAllBytes(outputPath));
         String expectedOutput = "ERROR: User Story US16:Male last name";
         assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ different last names");
     }
```

- Test 3: Tests the scenario where there are no male family members, expecting no errors.

```
     @Test
     public void testNoMaleMembers() throws Exception {
         Individual child1 = new Individual("@I1@");
         child1.setId("@I1@");
         child1.setName("Jane Doe");
         child1.setSex("female");

         Individual child2 = new Individual("@I2@");
         child2.setId("@I2@");
```

```
        child2.setName("Emily Doe");
        child2.setSex("female");

        Family fam = new Family("@F1@");
        fam.setChild(new ArrayList<>(Arrays.asList("@I1@", "@I2@")));

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child1.getId(), child1);
        individuals.put(child2.getId(), child2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Gedcom_Service.Malelastname(families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ no male members");
        }
    }
```

- Test 4: Verifies the content of the output file for the case where male family members have different last names.(This test will not pass since this this method has bug!)

```
    public void testMaleLastNameFileOutput() throws Exception {
        Individual child1 = new Individual("@I1@");
        child1.setId("@I1@");
        child1.setName("John Doe");
        child1.setSex("male");

        Individual child2 = new Individual("@I2@");
        child2.setId("@I2@");
        child2.setName("Mike Smith");
        child2.setSex("male");

        Family fam = new Family("@F1@");
        fam.setChild(new ArrayList<>(Arrays.asList("@I1@", "@I2@")));

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(child1.getId(), child1);
        individuals.put(child2.getId(), child2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(fam.getId(), fam);

        Gedcom_Service.Malelastname(families);
```

```
23
24          Path outputPath = Paths.get(Gedcom_Service.fileName);
25          assertTrue(Files.exists(outputPath), "Expected output file to be
                ↪ created");
26
27          String fileContent = new String(Files.readAllBytes(outputPath));
28          String expectedOutput = "ERROR: User Story US16:Male last name";
29          assertTrue(fileContent.contains(expectedOutput), "Expected error
                ↪ message to be written to the file for different last names");
30      }
```

1.1.8 Aunts and Uncles Name

The hardest test creating was for this method.The different test case scenrarios are as follow:

- Test 1: Tests the scenario where there is no incest, expecting no errors.

```
1       @Test
2       public void testAuntsAndUnclesNameNoIncest() throws Exception {
3           Individual father = new Individual("@I1@");
4           father.setId("@I1@");
5           father.setName("John Doe");
6           father.setChildOf("@F3@");
7
8           Individual mother = new Individual("@I2@");
9           mother.setId("@I2@");
10          mother.setName("Jane Smith");
11          mother.setChildOf("@F2@");
12
13          Individual child = new Individual("@I3@");
14          child.setId("@I3@");
15          child.setName("Child Doe");
16          child.setChildOf("@F1@");
17          child.setSpouseOf("@F4@");
18
19          Individual aunt = new Individual("@I4@");
20          aunt.setId("@I4@");
21          aunt.setName("Aunt Doe");
22
23          Individual uncle = new Individual("@I5@");
24          uncle.setId("@I5@");
25          uncle.setName("Uncle Smith");
26
27          Individual grandmother = new Individual("@I6@");
28          grandmother.setId("@I6@");
29          grandmother.setName("Grandmother Smith");
30
31          Individual grandfather = new Individual("@I7@");
```

```
        grandfather.setId("@I7@");
        grandfather.setName("Grandfather Smith");

        Individual grandmother2 = new Individual("@I8@");
        grandmother2.setId("@I8@");
        grandmother2.setName("Grandmother Doe");

        Individual grandfather2 = new Individual("@I9@");
        grandfather2.setId("@I9@");
        grandfather2.setName("Grandfather Doe");

        Family family = new Family("@F1@");
        family.setHusb("@I1@");
        family.setWife("@I2@");
        family.setChild(new ArrayList<>(Arrays.asList("@I3@")));

        Family motherFamily = new Family("@F2@");
        motherFamily.setHusb("@I9@");
        motherFamily.setWife("@I8@");
        motherFamily.setChild(new ArrayList<>(Arrays.asList("@I2@", "@I4@")))
            ↪ ;

        Family fatherFamily = new Family("@F3@");
        fatherFamily.setHusb("@I7@");
        fatherFamily.setWife("@I6@");
        fatherFamily.setChild(new ArrayList<>(Arrays.asList("@I1@", "@I5@")))
            ↪ ;

        Family childFamily = new Family("@F4@");
        childFamily.setHusb("@I3@");
        childFamily.setWife("@I6@");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(father.getId(), father);
        individuals.put(mother.getId(), mother);
        individuals.put(child.getId(), child);
        individuals.put(aunt.getId(), aunt);
        individuals.put(uncle.getId(), uncle);
        individuals.put(grandmother.getId(), grandmother);
        individuals.put(grandfather.getId(), grandfather);
        individuals.put(grandmother2.getId(), grandmother2);
        individuals.put(grandfather2.getId(), grandfather2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family.getId(), family);
        families.put(motherFamily.getId(), motherFamily);
        families.put(fatherFamily.getId(), fatherFamily);
```

```
        families.put(childFamily.getId(), childFamily);

        Gedcom_Service.individuals = individuals;

        assertEquals(Gedcom_Service.individuals, individuals);

        Gedcom_Service.AuntsandUnclesname(families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ no incest");
        }
    }
```

- Test 2: Tests the scenario where a child is married to their aunt, expecting an error message.

```
    @Test
    public void testAuntsAndUnclesNameIncest() throws Exception {
        Individual father = new Individual("@I1@");
        father.setId("@I1@");
        father.setName("John Doe");
        father.setChildOf("@F3@");

        Individual mother = new Individual("@I2@");
        mother.setId("@I2@");
        mother.setName("Jane Smith");
        mother.setChildOf("@F2@");

        Individual child = new Individual("@I3@");
        child.setId("@I3@");
        child.setName("Child Doe");
        child.setChildOf("@F1@");
        child.setSpouseOf("@F4@");

        Individual aunt = new Individual("@I4@");
        aunt.setId("@I4@");
        aunt.setName("Aunt Doe");

        Individual grandmother = new Individual("@I6@");
        grandmother.setId("@I6@");
        grandmother.setName("Grandmother Smith");

        Individual grandfather = new Individual("@I7@");
        grandfather.setId("@I7@");
        grandfather.setName("Grandfather Smith");

```

```
        Individual grandmother2 = new Individual("@I8@");
        grandmother2.setId("@I8@");
        grandmother2.setName("Grandmother Doe");

        Individual grandfather2 = new Individual("@I9@");
        grandfather2.setId("@I9@");
        grandfather2.setName("Grandfather Doe");

        Family family = new Family("@F1@");
        family.setHusb("@I1@");
        family.setWife("@I2@");
        family.setChild(new ArrayList<>(Arrays.asList("@I3@")));

        Family motherFamily = new Family("@F2@");
        motherFamily.setHusb("@I9@");
        motherFamily.setWife("@I8@");
        motherFamily.setChild(new ArrayList<>(Arrays.asList("@I2@", "@I4@")))
            ↪ ;

        Family fatherFamily = new Family("@F3@");
        fatherFamily.setHusb("@I7@");
        fatherFamily.setWife("@I6@");
        fatherFamily.setChild(new ArrayList<>(Arrays.asList("@I1@")));

        Family childFamily = new Family("@F4@");
        childFamily.setHusb("@I3@");
        childFamily.setWife("@I4@"); // Child is married to Aunt

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(father.getId(), father);
        individuals.put(mother.getId(), mother);
        individuals.put(child.getId(), child);
        individuals.put(aunt.getId(), aunt);
        individuals.put(grandmother.getId(), grandmother);
        individuals.put(grandfather.getId(), grandfather);
        individuals.put(grandmother2.getId(), grandmother2);
        individuals.put(grandfather2.getId(), grandfather2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family.getId(), family);
        families.put(motherFamily.getId(), motherFamily);
        families.put(fatherFamily.getId(), fatherFamily);
        families.put(childFamily.getId(), childFamily);

        Gedcom_Service.individuals = individuals;
        Gedcom_Service.families = families;

```

```
        Gedcom_Service.AuntsandUnclesname(families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR: User Story US20: Aunts and Uncles";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ incest");
    }
```

- Test 3: Verifies the content of the output file for the case where a child is married to their aunt.

```
    @Test
    public void testAuntsAndUnclesNameFileOutput() throws Exception {
        Individual father = new Individual("@I1@");
        father.setId("@I1@");
        father.setName("John Doe");
        father.setChildOf("@F3@");

        Individual mother = new Individual("@I2@");
        mother.setId("@I2@");
        mother.setName("Jane Smith");
        mother.setChildOf("@F2@");

        Individual child = new Individual("@I3@");
        child.setId("@I3@");
        child.setName("Child Doe");
        child.setChildOf("@F1@");
        child.setSpouseOf("@F4@");

        Individual aunt = new Individual("@I4@");
        aunt.setId("@I4@");
        aunt.setName("Aunt Doe");

        Individual grandmother = new Individual("@I6@");
        grandmother.setId("@I6@");
        grandmother.setName("Grandmother Smith");

        Individual grandfather = new Individual("@I7@");
        grandfather.setId("@I7@");
        grandfather.setName("Grandfather Smith");

        Individual grandmother2 = new Individual("@I8@");
        grandmother2.setId("@I8@");
        grandmother2.setName("Grandmother Doe");

```

```
        Individual grandfather2 = new Individual("@I9@");
        grandfather2.setId("@I9@");
        grandfather2.setName("Grandfather Doe");

        Family family = new Family("@F1@");
        family.setHusb("@I1@");
        family.setWife("@I2@");
        family.setChild(new ArrayList<>(Arrays.asList("@I3@")));

        Family motherFamily = new Family("@F2@");
        motherFamily.setHusb("@I9@");
        motherFamily.setWife("@I8@");
        motherFamily.setChild(new ArrayList<>(Arrays.asList("@I2@", "@I4@")))
            ↪ ;

        Family fatherFamily = new Family("@F3@");
        fatherFamily.setHusb("@I7@");
        fatherFamily.setWife("@I6@");
        fatherFamily.setChild(new ArrayList<>(Arrays.asList("@I1@")));

        Family childFamily = new Family("@F4@");
        childFamily.setHusb("@I3@");
        childFamily.setWife("@I4@"); // Child is married to Aunt

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(father.getId(), father);
        individuals.put(mother.getId(), mother);
        individuals.put(child.getId(), child);
        individuals.put(aunt.getId(), aunt);
        individuals.put(grandmother.getId(), grandmother);
        individuals.put(grandfather.getId(), grandfather);
        individuals.put(grandmother2.getId(), grandmother2);
        individuals.put(grandfather2.getId(), grandfather2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family.getId(), family);
        families.put(motherFamily.getId(), motherFamily);
        families.put(fatherFamily.getId(), fatherFamily);
        families.put(childFamily.getId(), childFamily);

        Gedcom_Service.individuals = individuals;
        Gedcom_Service.families = families;

        Gedcom_Service.AuntsandUnclesname(families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
```

```
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR: User Story US20: Aunts and Uncles";
        assertTrue(fileContent.contains(expectedOutput), "Expected error
            ↪ message to be written to the file for incest");
    }
```

- Test 4: Tests the scenario where an individual referenced in the family does not exist in the individuals map, expecting a NullPointerException.

```
    @Test
    public void testAuntsAndUnclesNameParseException() throws Exception {
        Individual father = new Individual("@I1@");
        father.setId("@I1@");
        father.setName("John Doe");
        father.setChildOf("@F3@");

        Individual mother = new Individual("@I2@");
        mother.setId("@I2@");
        mother.setName("Jane Smith");
        mother.setChildOf("@F2@");

        Individual child = new Individual("@I3@");
        child.setId("@I3@");
        child.setName("Child Doe");
        child.setChildOf("@F1@");
        child.setSpouseOf("@F4@");

        Family family = new Family("@F1@");
        family.setHusb("@I1@");
        family.setWife("@I2@");
        family.setChild(new ArrayList<>(Arrays.asList("@I3@")));

        Family spouseFamily = new Family("@F4@");
        spouseFamily.setHusb("@I3@");
        spouseFamily.setWife("@I5@"); // @I5@ does not exist

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(father.getId(), father);
        individuals.put(mother.getId(), mother);
        individuals.put(child.getId(), child);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family.getId(), family);
        families.put(spouseFamily.getId(), spouseFamily);

        Gedcom_Service.individuals = individuals;
```

```
        Gedcom_Service.families = families;

        ByteArrayOutputStream errContent = new ByteArrayOutputStream();
        PrintStream originalErr = System.err;
        System.setErr(new PrintStream(errContent));

        try {
            Gedcom_Service.AuntsandUnclesname(families);
        } catch (NullPointerException e) {
            e.printStackTrace();
        }

        String errorOutput = errContent.toString();
        assertTrue(errorOutput.contains("NullPointerException"), "Expected
            ↪ NullPointerException to be printed to stderr");
        System.setErr(originalErr);
    }
```

1.1.9 Unique Family Name By Spouses

Different scenarios for testing this method has been considered where can categorized as follow:

- Test 1: Tests the scenario where families have different spouses or marriage dates, expecting no errors.

```
    @Test
    public void testUniqueFamilyNameBySpousesNoDuplicate() throws Exception {
        Individual husband1 = new Individual("@I1@");
        husband1.setId("@I1@");
        husband1.setName("John Doe");

        Individual wife1 = new Individual("@I2@");
        wife1.setId("@I2@");
        wife1.setName("Jane Smith");

        Individual husband2 = new Individual("@I3@");
        husband2.setId("@I3@");
        husband2.setName("Michael Johnson");

        Individual wife2 = new Individual("@I4@");
        wife2.setId("@I4@");
        wife2.setName("Emily Davis");

        Family family1 = new Family("@F1@");
        family1.setHusb("@I1@");
        family1.setWife("@I2@");
        family1.setMarriage("01/01/2000");

        Family family2 = new Family("@F2@");
```

```
        family2.setHusb("@I3@");
        family2.setWife("@I4@");
        family2.setMarriage("02/02/2010");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husband1.getId(), husband1);
        individuals.put(wife1.getId(), wife1);
        individuals.put(husband2.getId(), husband2);
        individuals.put(wife2.getId(), wife2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family1.getId(), family1);
        families.put(family2.getId(), family2);

        Gedcom_Service.uniqueFamilynameBySpouses(individuals, families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ unique families by spouses");
        }
    }
```

- Test 2: Tests the scenario where two families have the same spouses and marriage dates, expecting an error message.

```
    @Test
    public void testUniqueFamilyNameBySpousesDuplicate() throws Exception {
        Individual husband = new Individual("@I1@");
        husband.setId("@I1@");
        husband.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family family1 = new Family("@F1@");
        family1.setHusb("@I1@");
        family1.setWife("@I2@");
        family1.setMarriage("01/01/2000");

        Family family2 = new Family("@F2@");
        family2.setHusb("@I1@");
        family2.setWife("@I2@");
        family2.setMarriage("01/01/2000");

        HashMap<String, Individual> individuals = new HashMap<>();
```

```
        individuals.put(husband.getId(), husband);
        individuals.put(wife.getId(), wife);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family1.getId(), family1);
        families.put(family2.getId(), family2);

        Gedcom_Service.uniqueFamilynameBySpouses(individuals, families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        assertTrue(Files.exists(outputPath), "Expected output file to be
            ↪ created");

        String fileContent = new String(Files.readAllBytes(outputPath));
        String expectedOutput = "ERROR: User Story US24: Unique Families By
            ↪ Spouse";
        assertTrue(fileContent.contains(expectedOutput), "Expected error for
            ↪ duplicate families by spouses");
    }
```

- Test 3: Tests the scenario where families have the same spouses but different marriage dates, expecting no errors.

```
    @Test
    public void testUniqueFamilyNameBySpousesDifferentDates() throws
        ↪ Exception {
        Individual husband = new Individual("@I1@");
        husband.setId("@I1@");
        husband.setName("John Doe");

        Individual wife = new Individual("@I2@");
        wife.setId("@I2@");
        wife.setName("Jane Smith");

        Family family1 = new Family("@F1@");
        family1.setHusb("@I1@");
        family1.setWife("@I2@");
        family1.setMarriage("01/01/2000");

        Family family2 = new Family("@F2@");
        family2.setHusb("@I1@");
        family2.setWife("@I2@");
        family2.setMarriage("02/02/2001");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husband.getId(), husband);
        individuals.put(wife.getId(), wife);

```

```
        HashMap<String, Family> families = new HashMap<>();
        families.put(family1.getId(), family1);
        families.put(family2.getId(), family2);

        Gedcom_Service.uniqueFamilynameBySpouses(individuals, families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪ families with different marriage dates");
        }
    }
```

- Test 4: Tests the scenario where a husband has different wives in different families, expecting no errors.

```
    @Test
    public void testUniqueFamilyNameBySpousesDifferentSpouses() throws
        ↪ Exception {
        Individual husband1 = new Individual("@I1@");
        husband1.setId("@I1@");
        husband1.setName("John Doe");

        Individual wife1 = new Individual("@I2@");
        wife1.setId("@I2@");
        wife1.setName("Jane Smith");

        Individual wife2 = new Individual("@I3@");
        wife2.setId("@I3@");
        wife2.setName("Emily Davis");

        Family family1 = new Family("@F1@");
        family1.setHusb("@I1@");
        family1.setWife("@I2@");
        family1.setMarriage("01/01/2000");

        Family family2 = new Family("@F2@");
        family2.setHusb("@I1@");
        family2.setWife("@I3@");
        family2.setMarriage("01/01/2000");

        HashMap<String, Individual> individuals = new HashMap<>();
        individuals.put(husband1.getId(), husband1);
        individuals.put(wife1.getId(), wife1);
        individuals.put(wife2.getId(), wife2);

        HashMap<String, Family> families = new HashMap<>();
        families.put(family1.getId(), family1);
```

```
        families.put(family2.getId(), family2);

        Gedcom_Service.uniqueFamilynameBySpouses(individuals, families);

        Path outputPath = Paths.get(Gedcom_Service.fileName);
        if (Files.exists(outputPath)) {
            String fileContent = new String(Files.readAllBytes(outputPath));
            assertFalse(fileContent.contains("ERROR"), "Expected no error for
                ↪  families with different spouses");
        }
    }
```

1.1.10 Get Month

In the last part we are going to create some tests for the ”GetMonth” function.

- Test 1: Tests all valid month abbreviations to ensure the correct month number is returned.

```
    @Test
    public void testGetMonthValid() {
        assertEquals("01", Gedcom_Service.getMonth("JAN"), "Expected month
            ↪ number for JAN is 01");
        assertEquals("02", Gedcom_Service.getMonth("FEB"), "Expected month
            ↪ number for FEB is 02");
        assertEquals("03", Gedcom_Service.getMonth("MAR"), "Expected month
            ↪ number for MAR is 03");
        assertEquals("04", Gedcom_Service.getMonth("APR"), "Expected month
            ↪ number for APR is 04");
        assertEquals("05", Gedcom_Service.getMonth("MAY"), "Expected month
            ↪ number for MAY is 05");
        assertEquals("06", Gedcom_Service.getMonth("JUN"), "Expected month
            ↪ number for JUN is 06");
        assertEquals("07", Gedcom_Service.getMonth("JUL"), "Expected month
            ↪ number for JUL is 07");
        assertEquals("08", Gedcom_Service.getMonth("AUG"), "Expected month
            ↪ number for AUG is 08");
        assertEquals("09", Gedcom_Service.getMonth("SEP"), "Expected month
            ↪ number for SEP is 09");
        assertEquals("10", Gedcom_Service.getMonth("OCT"), "Expected month
            ↪ number for OCT is 10");
        assertEquals("11", Gedcom_Service.getMonth("NOV"), "Expected month
            ↪ number for NOV is 11");
        assertEquals("12", Gedcom_Service.getMonth("DEC"), "Expected month
            ↪ number for DEC is 12");
    }
```

- Test 2: Invalid inputs are tested.

```
    @Test
    public void testGetMonthNullPointerException() {
        assertThrows(NullPointerException.class, () -> {
            Gedcom_Service.getMonth(null);
        }, "Expected NullPointerException for null input");
    }
```

- Test 3: Pass null as an input for this function.

```
    @Test
    public void testGetMonthInvalidInput() {
        assertNull(Gedcom_Service.getMonth("ABC"), "Expected null for invalid
            ↪  month abbreviation ABC");
        assertNull(Gedcom_Service.getMonth(""), "Expected null for empty
            ↪ string");
    }
```

In overall, we have implemented 53 tests for the implemented code. The results have been shown in the figure 1:

Figure 1: Test Results

Note that 2 tests wouldn't pass due to having bug in the male last names method!

1.2 Mutant Testing

First we are going to add PIT to plugins and configure our "pitest":

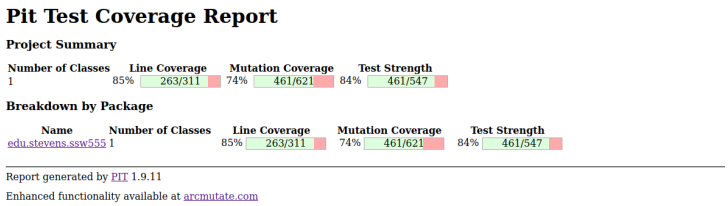
```
    id 'info.solidsoft.pitest' version '1.15.0'
```

Then we should configure our "pitest":

```
pitest {
    failWhenNoMutations = false
    verbose = true
    junit5PluginVersion = '1.2.1'
    pitestVersion = '1.9.11'
    targetClasses = ['edu.stevens.ssw555.Gedcom_Service']
    targetTests = ['edu.stevens.ssw555.Gedcom_ServiceTest']
    mutators = ['ALL']
    outputFormats = ['XML', 'HTML']
```

```
    threads = 4
}
```

(We consider all kind of mutators).
Now based on these we will create mutant for our Gedcom_Service with provided tests.With ”./gradlew pitest” command we can easily perform mutant testing. The results of mutant testing has been illustrated in figure 2:



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	85%	263/311	74%	461/621	84%	461/547

Breakdown by Package

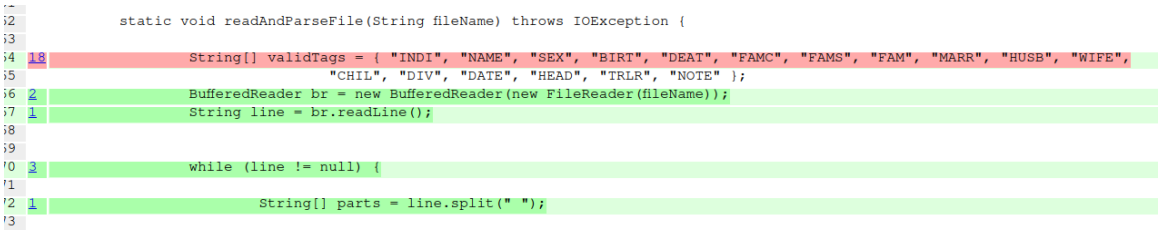
Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
edu.stevens.ssw555	1	85%	263/311	74%	461/621	84%	461/547

Report generated by PIT 1.9.11

Enhanced functionality available at arcmutate.com

Figure 2: Results of Mutant

According to figure 2, 74% of mutants are killed therefore we have reach this coverage. The test strength is also indicating that we have created strong tests.
In order to answer some provided question, we will use this report and XML foramt.
Q1: Since all of mutators are either ”killed”, ”Time-out” or survived we have no mutants that did not compiled.But for generating mutants that did not compiled, we could create a custom mutator where can not be compiled.But to be honest these mutants are not very helpful for us since they are not specify locations that needed to be tested.Thus we avoid to this but keep in mind that for such a purpose we can create some custom mutator where can remove semicolons or change the return format. For example if a method is void, we can change it to String to expect to return something.
Q2: Some mutants will change the original code in a way that they are equivalent to it.Furthermore, this mutant’s behavior are as same as original code.Assume that a branch in the code return True as boolean function.If we already change it True the code will already has same behavior right?(It is good to know that PIT had a bug in RETURN_TRUE that fixed!) And also some combined mutators can mutanct some different locations can lead to same behaviour of code.(Like change the if else branches and with in their containg branch!)
Q3:Some of this must be compiled but they are not helpful and can indeed killed by approximately all of the test cases for that method! Even we have a mutant that behave like this look at the figure 3: As you can see even if we



```
        static void readAndParseFile(String fileName) throws IOException {

18              String[] validTags = { "INDI", "NAME", "SEX", "BIRT", "DEAT", "FAMC", "FAMS", "FAM", "MARR", "HUSB", "WIFE",
                                       "CHIL", "DIV", "DATE", "HEAD", "TRLR", "NOTE" };
2               BufferedReader br = new BufferedReader(new FileReader(fileName));
1               String line = br.readLine();


3               while (line != null) {

1                       String[] parts = line.split(" ");
```

Figure 3: Compiled mutant but not helpful

change the condition in the while loop in ”readAndParseFile” method, almost all of the test cases can kill it. Since if we negate this function, we wont get what we want! Each file has some inputs that this method must behave like we wanted it to.Thus almost all of the test cases will kill this mutant.(Another examples are each statement with in either FAM or INDI nested if conditions. Since approximately 10 out of 19 test cases would detect these mutants).

```
families.put(fam.getId()
} else {
dupFam.add(fam);
}
```

Figure 4: Useful Mutant

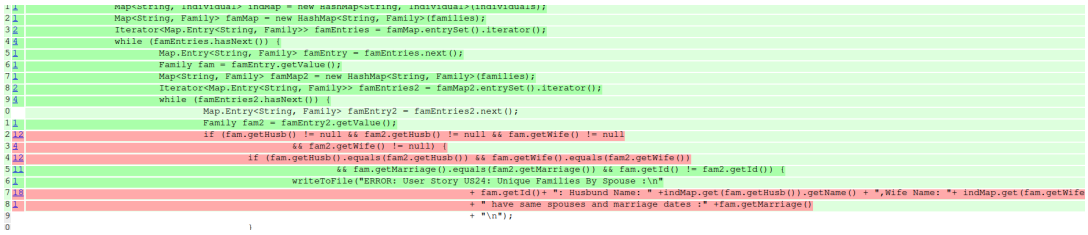


Figure 5: Live Mutants

Q4: Some of this mutants are indeed useful for us.In order to clarify this let's take give you an illustrative example from our code in figure 4: As depicted in figure 4, this mutant remove the add! Only one test will detect this for us and that is "readAndParseFileDupFam" test case. Remember we have 19 test case for this method meaning that only 1 out of 19 test cases killed this useful mutant.This example will give you a brief explantion of useful mutant.Another example for this would be "dupInd" in the "readAndParse" method.
Q5:Live mutants are those are not detected by any test cases where lead us to create some tests for! For these purpose we want to kill the following mutants in figure 5: Now we will write some tests for killing live mutants where can seen in the last line of codes.As you can see in figure 6 we resolve some of them: **Q7**:The answer they can appear through different mutants but they also differ in techniques they yiels. And they can indeed help us more bugs.

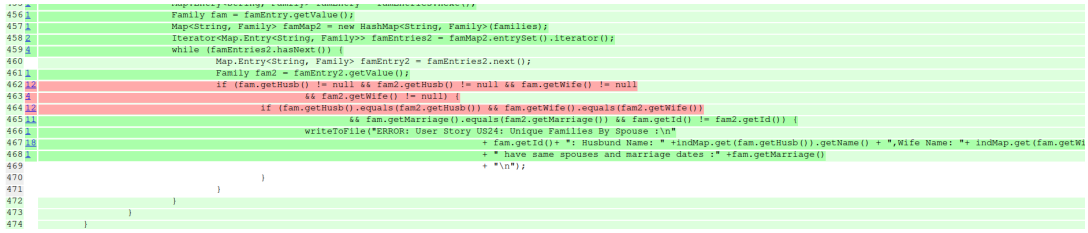


Figure 6: Kill some live mutants

2 Question 2

2.1 Concolic Testing

Steps of Concolic testing:

- Generate concrete inputs, each taking different program path
- On each input, execute program both concretely and symbolically
- Both cooperate with each other.Concrete execution guides symbolic execution where enables it to overcome incompleteness of theorem prover.Symbolic execution guides generation of concrete inputs in order to increase program code coverage.

```
1  public static void h(int x, int y) {
2      // pre-cond: x, y >= 0
3      if (x <= 6 * 31) {
4          System.out.println((x / 31 + 1) + " " + (x % 31 + 1));
5      } else {
6          x -= 6 * 31;
7          if (x <= 5 * 30) {
8              System.out.println((7 + x / 30) + " " + (1 + x % 31));
9          } else {
10             x -= 5 * 30;
11             boolean leap = isLeapYear(y);
12             if ((leap && x <= 30) || (!leap && x <= 29)) {
13                 System.out.println(12 + " " + x);
14             } else {
15                 throw new RuntimeException();
16             }
17         }
18     }
19 }
```

Assume the first phase as follow:(We will use table to show the each iteration of algorithem)

Concrete State	Symbolic State	Path Condition
$x = 93$ $y = 1400$	$x = x_0$ $y = y_0$	$x_0 \leq 6 * 31$

Table 1: Phase 1

We should solve $x_0 > 6 * 31$. We can set $x = 6 * 31 + 1$ for the second iteration of concolic algorithm:

Concrete State	Symbolic State	Path Condition
$x = 187$ $y = 1400$	$x = x_0$ $y = y_0$	$x_0 > 6 * 31$ $x_0 - 6 * 31 \leq 5 * 30$

Table 2: Phase 2

Now we should solve unequal equation $x_0 > 336$. Now based on this we can set x to 337.

Concrete State	Symbolic State	Path Condition
$x = 337$	$x = x_0$	$x_0 > 6 * 31$
$y = 1400$	$y = y_0$	$x_0 - 6 * 31 > 5 * 30$
leap = True	leap = isLeapYear(y_0)	$((\text{leap}_0 \wedge (x_0 - 6 * 31 - 5 * 30 \leq 30) \vee (\neg leap \wedge x_0 - 6 * 31 - 5 * 30 \leq 29$

Table 3: Phase 2

Since we check this path we should If we solve this $(x_0 - 6 * 31 - 5 * 30) > 30)$ we would get have set x to be 367. But to be more precise we should solve negation of the last conditional path. Therefore there will be two possible answers for this:

1. First assign $(x_0 - 6 * 31 - 5 * 30) > 30)$ with leap year.
2. Second possible answer of negate the last condition is $(x_0 - 6 * 31 - 5 * 30) > 30)$ and not leap year.

2.2 Force to see Not leap year

For this purpose we can do the following change:

```
public static void h(int x, int y) {
    // pre-cond: x, y >= 0
    if (x <= 6 * 31) {
        System.out.println((x / 31 + 1) + " " + (x % 31 + 1));
    } else {
        x -= 6 * 31;
        if (x <= 5 * 30) {
            System.out.println((7 + x / 30) + " " + (1 + x % 31));
        } else {
            x -= 5 * 30;
            boolean leap = isLeapYear(y);
            if (!leap) {
                if (x <= 29) {
                    System.out.println(12 + " " + (x + 1));
                } else {
                    throw new RuntimeException();
                }
            } else {
                if (x <= 30) {
                    System.out.println(12 + " " + (x + 1));
                } else {
                    throw new RuntimeException();
                }
            }
        }
    }
}
```

With the code above we can force the algorithem that see the branc of (!leap). Note that the algoritehm will stop when each computation tree has been seen.