Project Team 7 - Binary Tree

Team Members

Serar Matloob, Michael Frieze, Tiantan Ma, Jesse Emelian

Project Objectives

- Construct 3 Binary Tree Data Structures using key value pairs of int[] arrays generated by Team 1
- Create Binary Tree traversal methods and search each tree for the first two odd integer data values between 900 and 9000
- Determine the most efficient method and record efficiency results using Big O notation and pass the results back to Team 1

Overview of Project Requirements

- **Construct 3 Binary Tree Data Structures by creating key value pairs using int[] arrays generated by Team 1**
    - Build jUnit test to traverse through tree, find the first 2 odd data value and add to an int[] array
    - Build Node generation helper class responsible for creating nodes for Binary Tree class
    - Build Array generation helper class responsible for generating int[] arrays to fill Binary Tree for testing
    - Build Binary Tree class responsible for generating and recursively searching each tree
- **Search each tree for the first two odd integer data values between 900 and 9000**
    - Generate Binary Tree using buildBinaryTree() method passing in values from the two generated arrays
    - Search Binary Trees using one of three traversal methods
        - In Order - Traverse left subtree recursively, returning to root, then traversing right subtree recursively
        - Pre Order - Start at root node, traverse left subtree recursively, then traverse right subtree recursibely
        - Post Order - Traverse left subtree recursively, then traverse right subtree recursively, returning to root node last
- **Determine the most efficient method and record efficiency results using Big O notation'**
    - We tested the binary tree using each traversal method, on 20 and 2000 node arrays in order and post order traversals were very similar. In 20 and 2000 value arrays In Order and Post Order traversal times were very similar. Larger arrays made Post Order 34% faster on average
    -

Java Code Overview

Helper Package

- Node.class
    - Generates the nodes to fill the tree
- ArrayListGen.class
    - Generates arrays used for testing the Binary Tree traversals

Production Package

- BinaryTree.java
    - Builds the Binary Tree using nodes generated in Node class
    - Traverses through Binary Tree using one of three methods
    - Finds and stores first two odd data values between 900 and 9000

Junit Test Package

- Setup
    - Initializes the test by creating the test arrays (Keys and Data) within a given size and passing them to the BinaryTree
- TestBinaryTree
    - In the test we call the buildBinaryTree() method to build the tree using the array values then call the searchOddPreOrder() method to traverse through the tree and find the first two odd data values in a given range and add it to an. Record the time the traversal took and print in nanoseconds.
    -

Integration Problems

- Different teams use different names for returned method
  - Solved by changing them to the standard names set by Team 1

- What should be the return value when no integers found
  - Talked with Team 1 and set return number ‘-1’ instead of the default ‘0’ when odds found are less than two.

Examples:

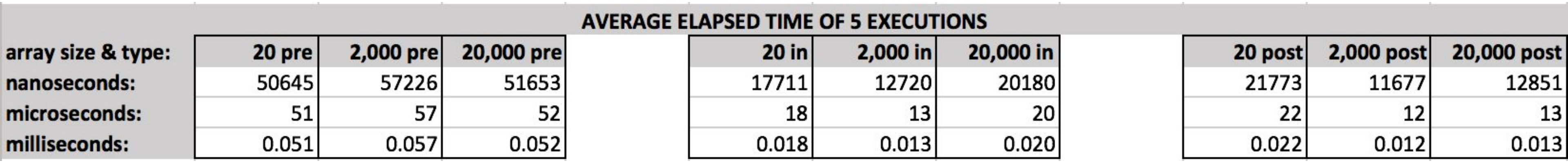
- Naming convention:
  - Before: getData()
  - After: getValues()

- Returned values:
  - Before: odds found: [567, 0]
  - After: odds found: [567, -1]

Internal Problems

- Algorithm in searching binary trees: since binary tree involves with recursion, need to figure out when should we break out the if loops.
    - Reviewed the lectures and revised the codes many times.
- Avoid “For ” loop in searching methods : want to use less time for searching
    - Create an arrayList and put the found odd numbers in it by using add() method.
- Searching time too small: the test results always become 1 or 0 because its searching too fast.
    - Use nanoseconds instead of milliseconds to count the time.

Performance Results



AVERAGE ELAPSED TIME OF 5 EXECUTIONS

array size & type:	20 pre	2,000 pre	20,000 pre	20 in	2,000 in	20,000 in	20 post	2,000 post	20,000 post
nanoseconds:	50645	57226	51653	17711	12720	20180	21773	11677	12851
microseconds:	51	57	52	18	13	20	22	12	13
milliseconds:	0.051	0.057	0.052	0.018	0.013	0.020	0.022	0.012	0.013

We used a preorder traversal to find the first 2 odd values within the range of 900 to 9,000.

- Worst time for pre-order traversal is O(N)
- Elapsed time for 20 element array was approximately 51 microseconds using preorder traversal.

We also compared the elapsed time of different traversal methods.

- The data in the chart above shows the elapsed time of each traversal method as well as the number of elements in each array.
- According to the data we collected, the inorder and postorder traversal found the odd integers quicker than a preorder traversal regardless of the array size.
- Although the elapsed time of inorder and postorder traversal was similar with smaller arrays, postorder traversal became quicker as the array size got larger.

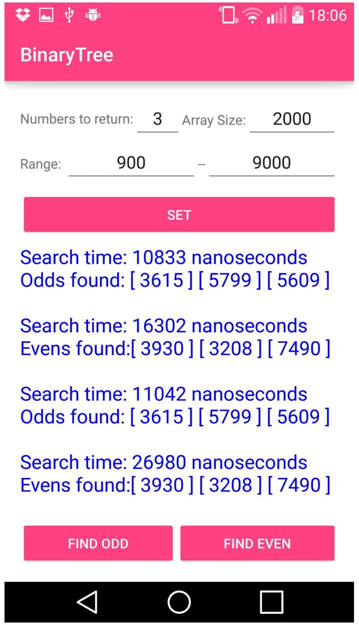
Summery

Lessons Learned:

- Teamwork:
  - Ability to work with teammates to build and organize the code.
  - Ability to work with other teams.
  - “There is no 1 hero in java programming!”
- Creating a project with future flexibility in mind.
- Building and using binary trees for efficient searching algorithms.
- Proper application testing using JUnit Test Case.
- Learned to use GitHub

Easy implementation example:

- An Android App created easily based on binary tree project our team has made.
  - Return options show how flexible the java code is.



Numbers to return: 3 Array Size: 2000

Range: 900 -- 9000

```
Search time: 10833 nanoseconds
Odds found: [ 3615 ] [ 5799 ] [ 5609 ]

Search time: 16302 nanoseconds
Evens found:[ 3930 ] [ 3208 ] [ 7490 ]

Search time: 11042 nanoseconds
Odds found: [ 3615 ] [ 5799 ] [ 5609 ]

Search time: 26980 nanoseconds
Evens found:[ 3930 ] [ 3208 ] [ 7490 ]
```