CHRIS KATEERA
KTRKUD001
ASSIGNMENT 1
March 3, 2020

CSC2001 DATA STRUCTURES & JAVA

UNIVERSITY OF CAPE TOWN
IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

COMPARISON OF A BINARY SEARCH TREE & AN ARRAY DATA STRUCTURE

INTRODUCTION

Different data structures are organized differently and are optimized for certain operations. In this experiment is to compare the binary search tree with a traditional unsorted array data structure in terms of time complexity to find an element according to a unique key in java. A traditional array uses a linear search which has a linear complexity: O(n) – it's run time increases at an order of magnitude proportional to n if we assume that the key is equally likely to be in any of the n locations in the array. On the other hand, a binary search tree has a logarithmic time complexity O(logn) - the time is decreased at magnitude inversely proportional to n – the binary search looks through only half of an increasingly smaller data set per step. In this experiment i intend to prove this to you. Below i guide you through my procedure.

THE EXPERIMENT

In order to do this experiment I initially had to use the same dataset as a control for the experiment – the Cape Town Load shedding data. I made two apps: the ArrayApp and the LSBSTApp. I read input from a file into my storage data structures and I implemented a lookup algorithm for the the ArrayApp program and the LSBSTApp program. After testing and making sure they were working, I then instrumented both programs. This meant I could count the number of steps it took for me to get to any desired key. I then ran this with the original dataset, then made subsets of differently sized data (using a bash script) so that I could plot and get an indication of the correlation between steps taken to find the same key on both programs and the data size. I then plotted my results and discussed them.

PROGRAM DESIGN

I created a package with two new classes: the LSArrayApp and the LSBSTApp. I also added the BinarySearchTree classes to the same package: BinarySearchTree,BinaryTree, BinaryTreeNode, BTQueue, BTQueueNode.
The LSArrayApp used a traditional array as its storage datatype: It stored each line of the loadshedding data as an element in an array, while the LSBSTApp used a Binary Search Tree and inserted each element in different nodes in the tree- according to the Binary Search Tree algorithm.. The LSBSTApp uses insertion and lookup methods contained in the BinarySearchTree class. There are conting variables located in LSArrayApp which are accessed and used in LSBSTApp and in the find method of BinarySearchTree.
I implemented OOP by makig use of separate classes with working methods and variables – some of which I then re-used without compromising security.

TEST VALUES & RESULTS

```
#!/usr/bin/env bash
#PART 1&2
echo "Part 1 & 2 Execution: Check part1_2.txt"
: '
no parameters
3 working parameters
3 invalid parameters
'
(make run &&
make run params="2 29 00" &&
make run params="4 17 10" &&
make run params="8 10 22" &&
make run params="3 5 7" &&
make run params="4 7" &&
make run params="invalid text input") > part1_2.txt

#PART 3&4
echo "Part 3 & 4 Execution: Check part3_4.txt"

: '
no parameters
3 working parameters
3 invalid parameters
'
(make run2 &&
make run2 params="2 29 00" &&
make run2 params="4 17 10" &&
make run2 params="8 10 22" &&
make run2 params="3 5 7" &&
make run2 params="4 7" &&
make run2 params="invalid text input") > part3_4.txt
```

Bash script with the test values I used for part 2 and part 4.

```
java -cp ./bin LSArrayApp
------------START-OF-OUTPUT------------------
1_1_00 1
1_17_00 1
1_2_00 13
1_18_00 13
1_3_00 9
1_19_00 9
1_4_00 5
1_20_00 5
8_28_22 14, 6, 10, 2, 15, 7, 11, 3
8_13_22 15, 7, 11, 3, 12, 4, 8, 16
8_29_22 15, 7, 11, 3, 12, 4, 8, 16
8_14_22 15, 7, 11, 3, 12, 4, 8, 16
8_30_22 15, 7, 11, 3, 12, 4, 8, 16
8_15_22 15, 7, 11, 3, 12, 4, 8, 16
```

```
8_31_22 15, 7, 11, 3, 12, 4, 8, 16
8_16_22 15, 7, 11, 3, 12, 4, 8, 16
Value of opCount is:0

------------END-OF-OUTPUT-----------------
```

First 10 and last 10 lines printed when PrintAllAreas is invoked - when no parameters are passed.

```
java -cp ./bin LSArrayApp  2 29 00
------------START-OF-OUTPUT-----------------
Number of arguments: 3
Matching areas: 4,12
Value of opCount is:398

------------END-OF-OUTPUT-----------------
java -cp ./bin LSArrayApp  4 17 10
------------START-OF-OUTPUT-----------------
Number of arguments: 3
Matching areas: 6,
Value of opCount is:1273

------------END-OF-OUTPUT-----------------
java -cp ./bin LSArrayApp  8 10 22
------------START-OF-OUTPUT-----------------
Number of arguments: 3
Matching areas: 14,
Value of opCount is:2964

------------END-OF-OUTPUT-----------------
java -cp ./bin LSArrayApp  3 5 7
------------START-OF-OUTPUT-----------------
Number of arguments: 3
Areas not found
Value of opCount is:2976

------------END-OF-OUTPUT-----------------
java -cp ./bin LSArrayApp  4 7
------------START-OF-OUTPUT-----------------
Invalid number of arguments
Value of opCount is:0

------------END-OF-OUTPUT-----------------
java -cp ./bin LSArrayApp  invalid text input
------------START-OF-OUTPUT-----------------
Number of arguments: 3
Areas not found
Value of opCount is:2976

------------END-OF-OUTPUT-----------------
```

Test Results when parameters are passed: opCount is the operations count.

FINAL RESULTS & DISCUSSION

Dataset / oPCount	FREQUENCY FOR 300	FREQUENCY FOR 600	FREQUENCY FOR 900	FREQUENCY FOR 2700
1	18	35	53	186
2	94	188	279	879
3	138	275	412	1173
4	50	102	156	462

Table1: LSBSTApp opCount vs Frequency for some of the datasets

As seen above, the maximum opcount is 4. The minimum is 1. The various operation counts for the individual keys is statistically distributed as shown above. For each set, opCount 1 has the least frequency, followed by opCount 4, then opCount2 and finally opCount4.

In Figure 1 on the right a sample subset is presented summarizing how other sets performed. The number of operation counts was proportional to the key index for which it was looking up. This alludes to the constant time complexity we expected to see from the introduction. The time complexity of the algorithm is O(n). Best case opCount is 1 with the key element in the Array. The Average case opCount is the (n+1)/2 median key element in the list. The worst opCount case is n with the last key element in the list.

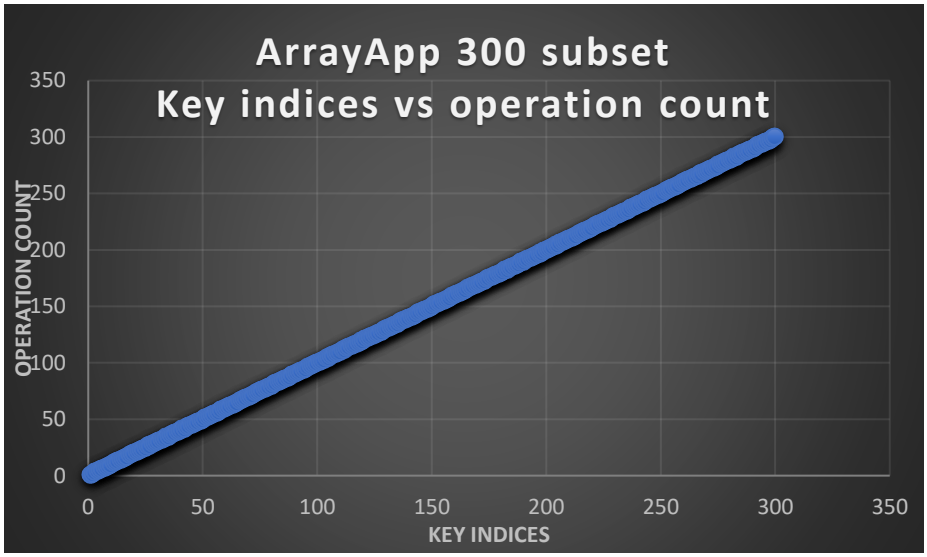
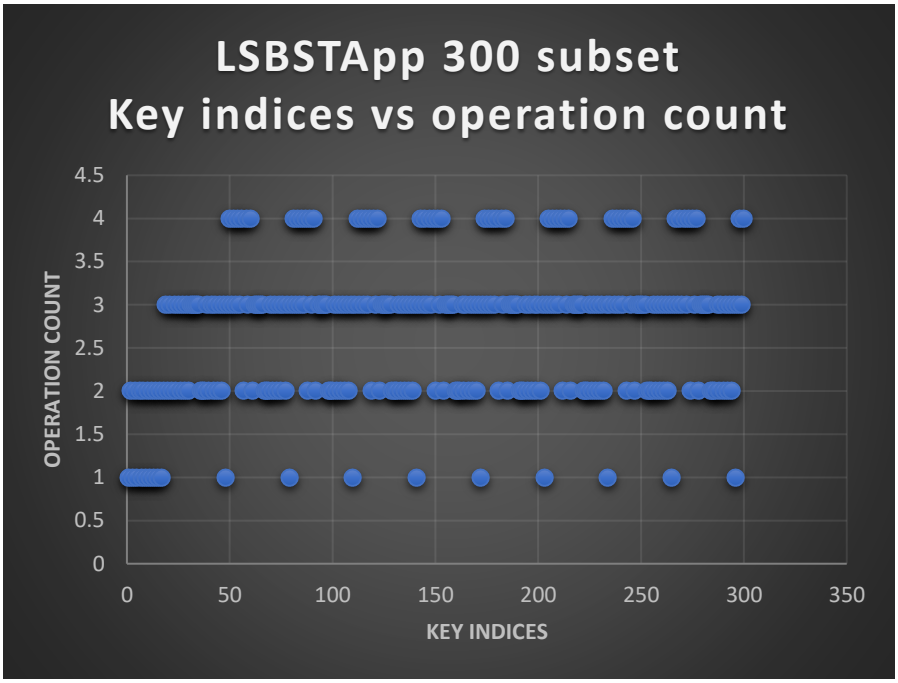
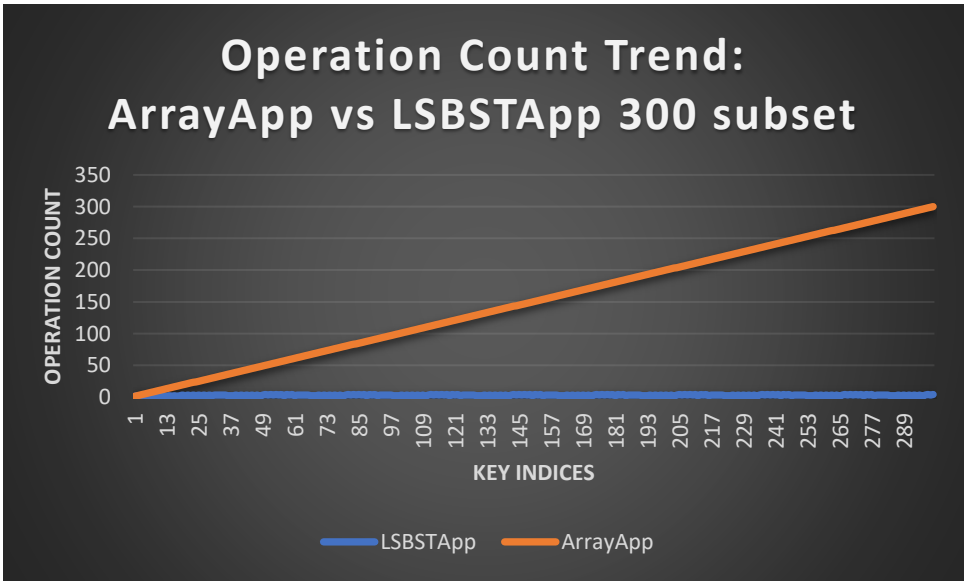


Figure 1: ArrayApp Graph

In Figure 2 on the right a sample subset is presented summarizing how other sets performed. The number of operation counts correspond to the height/level location of the node which is being looked up. The algorithm searches for the key knowing that the left node data is smaller than the right node data and this in turn halves the amount of data remaining to be searched each time. LSBSTApp follows a logarithmic time complexity O(logn). The best case opCount is 1 with the key element at the tree root. Average case opCount is for elements in the level of height/2. The worst case is for the elements at the bottom of the tree: at level height. It gets the complexity to O(n) because it takes steps proportional to the height of the tree.



This figure on the right finally compares the two Applications, evidently showing that the LSBSTApp is by far more efficient in terms of searching for a key. Extending this to the other sets we realize that a logarithmic fuction grows slowly and even for larger data sets, the steps still remain few. This points to a Binary Search Tree's advantage of scalability. However, for the ArrayApp, the number of steps grows linearly as we increase the datasets.



CREATIVITY

1. I used bash scripting to automate the 10 susbset creation in a file called susbset.sh.
2. I created a Makefile that accepts parameters in a string called params that would then be sent to my program. E.g. make run params="2 4 6"
3. I created a bash script to automate the testing and lookup of the subset values then printing the results to a file.
4. I counted the insertion steps when adding to a Binary Search Tree.
5. I used a bash script to do the testing of part2 and part4.

GIT SUMMARY STATISTICS

```
0: commit ab0ddc03d5fbd2986940d0bbf2e19e41c46ee908
1: Merge: 1058e6e f064d91
2: Author: Kudzai Kateera <ktrkud001@myuct.ac.za>
3: Date: Thu Mar 5 09:11:33 2020 +0200
4:
5: Merge branch 'master' of https://github.com/chris-kck/CS2
6:
7: commit f064d9114fa7bd2e511f724d1300db718fcc1315
8: Author: Chris Kay <kckateera@gmail.com>
9: Date: Thu Mar 5 09:11:34 2020 +0200
...
246: Author: Chris Kateera <kckateera@gmail.com>
247: Date: Mon Feb 24 23:08:37 2020 +0200
248:
249: Initial commit
250:
251: commit 251a837123e2f46009e9404a4327c9382e9608c5
252: Author: Kudzai Kateera <ktrkud001@nightmare.cs.uct.ac.za>
253: Date: Mon Feb 24 23:06:51 2020 +0200
254:
255: new file: test.txt
```

Git summary statistics

CONCLUSION

The experiment was a success and rigorously compared the traditional array to a binary search tree data structure. The results proved the various time complexities in the best, worst and average case scenarios. Finally, when it comes to searching, a binary search tree is faster and more scalable compared to an array.