**Report for Algorithms & Analysis Assignment 1**

Akshay Sunil Salunke- s3730440

Nitish Bhat- s3733170

|  |
| --- |
| We certify that this is all our own original work. If we took any parts from elsewhere, then they were non-essential parts of the assignment, and they are clearly attributed in my submission. We agree to this honor code by typing "Yes": Yes. |

**TASK B:**

**Experimental Setup**

Data Generation - For generating trees of different sizes, with various depths or the number of nodes, we made a program DataGenerator.java which takes a max value as the argument for generating random data with tree like structure within the specified limits.

Sizes of data - We generated different files for 10^2, 10^3, 10^5, 10^6 number of nodes as our datasets for comparison. We used these sizes because the size of 100 nodes allows us to contrast the performance for 100000 nodes for finding and splitting node operations for the Sequential and Linked data structures.

Generation of scenarios - We decided to analyse all 4 text files whose data was generated randomly for both, Sequential Representation and Linked Representation. Complexities chosen for all trees were the same for getting better results. Though we decided to conduct analysis on all 4 generated files, the file with 10^6 nodes took nearly 2 hours for data generation and as a result we decided to discard this file for our further analysis. Now considering 3 files, all the readings for Sequential Representation were taken first.

Scenario 1

Node addition was done from the interactive mode (after loading initial tree in memory) where the function splitNode() was given a recursive call. The node was split further to grow the tree, thus increasing the height of the tree. As we wanted to test the implementations on the nodes farthest from the root node, the nodes having maximum depth and no children were chosen for analysis.

* Leftmost leaf node
* Rightmost leaf node

Scenario 2

As the complexity of initial tree was varied, we decided to do both the functions of Finding Node (FN) and Finding Parent/Children (FP/FC) on the same node. We observed that the traversals take the longest time to reach the leaf nodes located at the bottom of the tree. The tree was first printed in InOrder to locate the leftmost and rightmost child nodes on the tree. We also found out that both implementations take their shortest time respectively to reach the root node so we considered it as one of our test cases along with the other two.

* Leftmost leaf node
* Rightmost leaf node
* Root node

Scenario 3

Traversing is used to visit every node of the tree. The main benefit of not using linear data structures is that it can be traversed in different ways according to one’s need. All 3 traversals were run on one implementation before proceeding to the second. The average of 3 test runs was calculated.

* PreOrder
* InOrder
* PostOrder

Timing - The time considered in all scenarios is in nanoseconds as it gives the most accurate results showing clear comparisons. Parameter settings for all datasets was same and 3 calculations were recorded for each scenario taking the average across these runs.

**Evaluation**

In sequential representation, for representing a binary tree with ‘n’ nodes, only a 1D array with a maximum size of 2^(n)-1 is needed. In linked representation, elements are linked using references or pointers. A node represents an element in linked list which have some data and a pointer pointing to the next node.

Scenario 1

|  |  |  |
| --- | --- | --- |
| **set4.txt (10^5 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |
| **Scenario 1** |  |  |
| SP 9277 A B (rightmost) | 14158638 | 7269886 |
| SP 16857 C D (leftmost) | 2178225 | 32858 |

|  |  |  |
| --- | --- | --- |
| **set1.txt (10^2 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |
| **Scenario 1** |  |  |
| SP 42 101 102 (rightmost) | 476616 | 340697 |
| SP 31 103 104 (leftmost) | 155079 | 145595 |

In Scenario 1, the observations suggest that for adding a node and growing the tree further, linked representation outperforms the sequential one. Even if the densities of the trees were varied, the results remained constant. This is because in an array, we need to check if there’s enough space to save left (2n+1) and right (2n+2) children for node n, if not, we need to make a new bigger array and copy the old array into it:

**myTree = Arrays.copyOf(myTree, myTree.length \* 2);**

whereas in a linked list, we just have to update the address present in the pointer of the node.

Scenario 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **set1.txt (10^2 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |  |  |
| **Scenario 2** | Finding Node |  | Finding Parent/Child |  |
| 42 (rightmost) | 11300 | 7800 | 301701 | 14300 |
|  | 7300 | 8000 | 236600 | 14100 |
|  | 7801 | 7600 | 280801 | 9800 |
|  | **8800.333333** | **7800** | **273034** | **12733.33333** |
|  |  |  |  |  |
| 74 (root) | 800 | 2699 | 499 | 6400 |
|  | 501 | 2501 | 501 | 3401 |
|  | 500 | 2900 | 400 | 3301 |
|  | **600.3333333** | **2700** | **466.6666667** | **4367.333333** |
|  |  |  |  |  |
| 31 (leftmost) | 11701 | 4658 | 119031 | 6400 |
|  | 12315 | 4453 | 169228 | 5412 |
|  | 24614 | 4997 | 154641 | 5692 |
|  | **16210** | **4702.666667** | **147633.3333** | **5834.666667** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **set4.txt (10^5 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |  |  |
| **Scenario 2** | Finding Node |  | Finding Parent/Child |  |
| 9277 (rightmost) | 532800 | 1647300 | 864500 | 1422000 |
|  | 592300 | 1645700 | 864500 | 1422000 |
|  | 592300 | 1690000 | 840600 | 1653600 |
|  | **572466.6667** | **1661000** | **856533.3333** | **1499200** |
|  |  |  |  |  |
| 75565 (root) | 700 | 900 | 1400 | 3400 |
|  | 810 | 1000 | 500 | 3200 |
|  | 601 | 900 | 700 | 3400 |
|  | **703.6666667** | **933.3333333** | **866.6666667** | **3333.333333** |
|  |  |  |  |  |
| 16857 (leftmost) | 2464975 | 1969 | 2514163 | 5950 |
|  | 1763781 | 1721 | 2117676 | 4951 |
|  | 1478185 | 2095 | 1516630 | 4708 |
|  | **1902313.667** | **1928.333333** | **2049489.667** | **5203** |

This was the most complex scenario for the analysis. As we varied the densities of the tree, we found that irrespective of the depth of the tree, the root node is found and searched for much faster by sequential representation than linked [Best case complexity for linear search = O (1)]. As the code in our LinkedRepresentation.java searches for the left tree first,

**return search (n.leftNode, nodeLabel) || search(n.rightNode, nodeLabel);**

we can see that linked representation finds the leftmost node faster. This is because in sequential representation, the array has to traverse through all elements in order to reach the desired node. However, we found a contradicting case for the rightmost node, where for tree with 10^2 nodes linked representation was faster while for 10^5 nodes it was the other way around. For tree with 10^5 nodes, sequential is faster because the time required to search rightmost node using recursive call (rightmost node is the last node in recursive call-stack) is larger than a linear array search. This might be because even though the theoretical complexity is the same for both, recursive call might have large overhead as compared to sequential representation.

Scenario 3

|  |  |  |
| --- | --- | --- |
| **set2.txt (10^3 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |
| **Scenario 3** |  |  |
| TP (Pre) | 531600 | 478599 |
|  | 303200 | 351400 |
|  | 208700 | 248200 |
|  | **347833.3333** | **359399.6667** |
|  |  |  |
| TI (In) | 383600 | 416800 |
|  | 223799 | 275600 |
|  | 116700 | 115300 |
|  | **241366.3333** | **269233.3333** |
|  |  |  |
| TS (Post) | 421200 | 392899 |
|  | 233501 | 283000 |
|  | 190800 | 231201 |
|  | **281833.6667** | **302366.6667** |

|  |  |  |
| --- | --- | --- |
| **set4.txt (10^5 nodes)** | **SeqTree (nano sec)** | **LinkTree (nano sec)** |
|  |  |  |
| **Scenario 3** |  |  |
| TP (Pre) | 184269100 | 229843200 |
|  | 140898500 | 213923900 |
|  | 179540600 | 174750400 |
|  | **168236066.7** | **206172500** |
|  |  |  |
| TI (In) | 383600 | 172661300 |
|  | 223799 | 164670500 |
|  | 116700 | 169080100 |
|  | **241366.3333** | **168803966.7** |
|  |  |  |
| TS (Post) | 421200 | 200282700 |
|  | 233501 | 194663600 |
|  | 190800 | 204673500 |
|  | **281833.6667** | **199873266.7** |

For Scenario 3, we observed a very close difference for printing all 3 traversals between both the implementations. There was a similarity seen for both where the InOrder printing was the fastest followed by PostOrder and PreOrder respectively.

**Recommendation**

For scenario 1, for all the datasets we would recommend using linked representation based on our analysis. The problem for growing a tree with sequential representation is that as it requires copying of an array, resizing is costlier and hence we often need to estimate the size of an array beforehand.

Scenario 2 analysis suggests that the depth of the tree does matter when we wish to find the node and its parent/child. As our find node function involves searching the left side of the tree first, the leftmost node is found first by linked tree. For the rightmost node, sequential is faster. The code of finding the farthest node by traversing will matter to see if the tree traverses to its right side first or the left side. We recommend using sequential representation for larger trees and using linked for smaller as searching of an array is often slower than linked.

All three depth first search traversals of all orders take comparatively less time for sequential representation. We recommend using this approach as in a linear data structure, it is very easy to move from a child to its parents and vice versa. Since arrays provide fast random access, they are slightly faster in traversing the whole tree in contrast to linked representation (which uses recursive call).

**TASK C:**

**Question C1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | O(1) | O(log(n)) | O(n) | O(n!) |
| Find parent | NO | NO | YES | NO |
| Find a node | NO | NO | YES | NO |
| Print all nodes | NO | NO | YES | NO |

Find node explanation: For Linear search, the worst case happens when the element to be searched (n in the above code) is not present in the array. When n is not present, the search() functions compares it with all the elements of arr[] one by one. For linked representation, we use recursive search on both subtrees(since this is a binary tree and not a binary search tree). Therefore, the worst-case time complexity would be O(n).

Find node and print all nodes explanation: For both the cases, we need to traverse through and visit every node of the tree once which is O(n).

**Question C2**

Strategy 1

For this method, the astronaut needs to do an exhaustive search for all n’s. As n=14 billion, the astronaut needs to ask 14 billion questions before knowing the right age of the alien will be the worst case. As for average case, the astronaut will know the right age after asking the alien n/2 i.e. 7 billion questions.

Strategy 2

This is an example of BST approach where the element in the right node is always bigger and left node is smaller than the parent node. For Binary search the worst case and average case time complexity is O(log n). As n= 14 billion, the astronaut will need to ask O(log 14000000000) ≈ 33 questions.