**Task B**

**Test & Analyse Report for Two Tree Implementations**

1. **Introduction**

First, we read all the tokens in the BSP\_combined .text file and output the basic information for the given data. The given data totals 2000 lines, 5998 data points, of which 3999 are different data points, from 1 to 40,000. Which means we have 3999 nodes in the tree created by the given dataset.

We first test with the given data, and then generate different data sets of different sizes to test the time of different functions in different tree depths. The data set size is from 100 to 10\*\*6, in each dataset we increase the size by multiple 10. At the same time, due to the random generation of data, we will randomly generate data sets 10 times in the same tree depth and run it multiple times for each data set to eliminate interference caused by different data or device.

In the following of the report, we compare different functions running times in different implementation trees. We use ten times average running time for comparation. For finding nodes and its parent/children function, we use ten times average running time for calling 1000 times certain function. The function calling time can be reset to any number in our code, we choose 1000 times because it’s large enough to show the average performance but not too large to take large amount of time.

The sampled data for testing finding node function are all exist in the tree, this will make the code never print error message. Cause input and output may take a lot amount of time, and the random generated data can’t control the IO time well, so we excluded the influence by IO, only focus on the algorithm running time. And also, all generated data only exist in the function, we don’t want cost time writing and reading a file.

We can run the code in analysis.java, the running time result will print in terminal. The time result maybe different cause different random generated data.

1. **Data tables in different tree node sizes**
2. **Given data set -- 4000 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 41023990 | 60483860 |
| Find node | 8444790 | 44740430 |
| Find node’s parent | 10093030 | 45448940 |
| Find node’s children | 7519930 | 50700560 |
| Inorder traverse | 4342730 | 4706000 |
| Preorder traverse | 5172380 | 2924030 |
| Postorder traverse | 8897830 | 3303010 |

1. **100 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 256170 | 238060 |
| Find node | 1108210 | 3197940 |
| Find node’s parent | 2778560 | 3896630 |
| Find node’s children | 4125220 | 3180000 |
| Inorder traverse | 1441110 | 449620 |
| Preorder traverse | 435460 | 310760 |
| Postorder traverse | 324130 | 535000 |

1. **10\*\*3 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 4217700 | 6329000 |
| Find node | 6360500 | 14235530 |
| Find node’s parent | 7299360 | 16256630 |
| Find node’s children | 7323830 | 16310570 |
| Inorder traverse | 5898650 | 3311510 |
| Preorder traverse | 5802150 | 1408670 |
| Postorder traverse | 2308590 | 1912150 |

1. **10\*\*4 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 128669160 | 261465790 |
| Find node | 34459500 | 107718330 |
| Find node’s parent | 29864490 | 100870640 |
| Find node’s children | 28115100 | 85029710 |
| Inorder traverse | 5022630 | 85029710 |
| Preorder traverse | 24882230 | 2241420 |
| Postorder traverse | 2954230 | 3762550 |

1. **10\*\*5 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 19271621870 | 23420722050 |
| Find node | 358815670 | 1248273110 |
| Find node’s parent | 339690540 | 1285545780 |
| Find node’s children | 331253920 | 1160576120 |
| Inorder traverse | 46555840 | 47507240 |
| Preorder traverse | 53886710 | 29376450 |
| Postorder traverse | 34420970 | 42428230 |

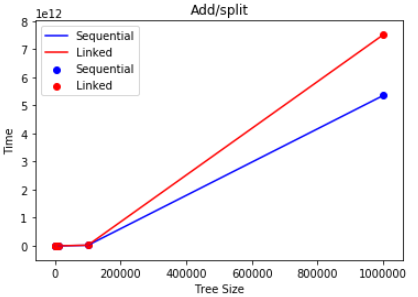
1. **10\*\*6 nodes in the tree**

|  |  |  |
| --- | --- | --- |
| Tree implementation | Sequential | Linked |
| Add/split time | 5350463159470 | 7512737061990 |
| Find node | 10851936880 | 26505799620 |
| Find node’s parent | 11086871860 | 26348957300 |
| Find node’s children | 10792107600 | 26501157660 |
| Inorder traverse | 524346010 | 452824680 |
| Preorder traverse | 420905040 | 322145990 |
| Postorder traverse | 460473030 | 443181660 |

1. **Comparison**

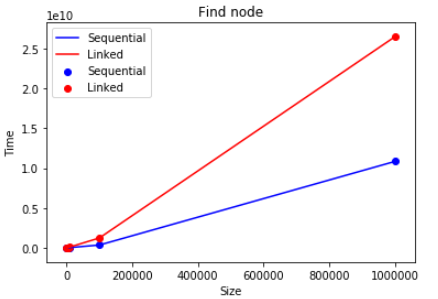
Based on the table, we made some graphs to clearer display of the speed relationship between the two algorithms:

1. Add/split



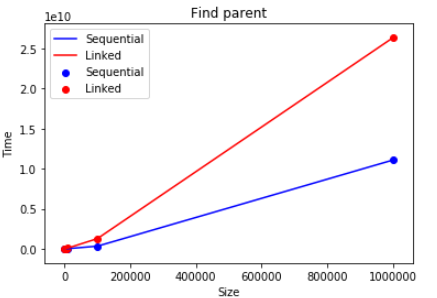
We can see on the plot that as the tree size become larger, the time cost of two algorithms all increased significantly. However the speed of construction of the tree by add/split function of sequential tree is faster.

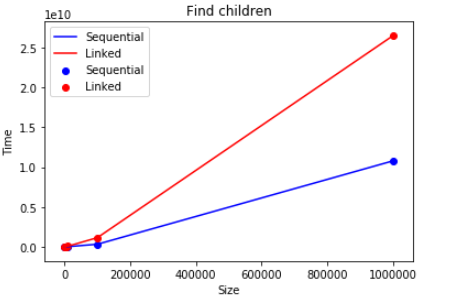
1. Find node



From the plot, sequential tree runs faster in any size of tree for finding a node.

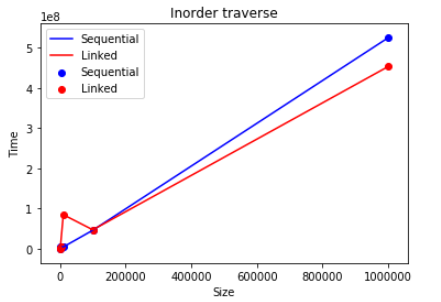
1. Find node’s parent & find node’s children





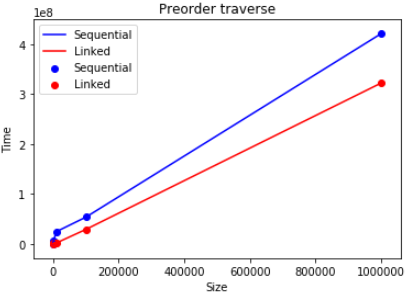
From the plot we can see that, in find node’s parent/children function, the sequential tree also preforms better, even the tendency is similar as finding node. This also proved that in finding node’s parent/children function, the main cost is still finding the node itself.

1. Inorder traverse



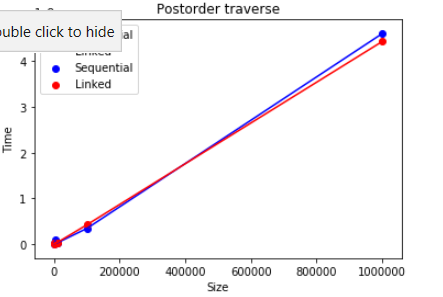
For inorder traverse function, when tree size is relatively small, the sequential tree performs better, however when tree size is large, the link tree performs better.

1. Preorder traverse



Link tree performs better all the time.

1. Postorder traverse



When tree is small, the sequential tree performs better slightly, however when tree is large link tree perform better slightly.

1. **Summary**

Based on the plots above we can clearly see the running efficiency of two implementations in different scenarios. Through the comparison we should know that, when we want construct a large tree by add/split function or find a node or its parent/children, the sequential tree is better. However, if we want to traverse a large tree in any order, the link tree performs better. If the tree is relatively small, sequential tree could do better in Inorder and Postorder traverse, so we may choose sequential tree in this condition.

**Task C**

**Question C1:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | O(1) | O(log(n)) | O(n) | O(n!) |
| Find parent |  |  | Yes |  |
| Find a node |  |  | Yes |  |
| Print all nodes |  |  | Yes |  |

For finding a node, the worst case is to search the whole tree. There are total n nodes in the tree, so the complexity of the worst case is O(n).

For finding a node’s parent, we need find the node firstly, the worst case complexity for finding the node on O(n) as we talked above. Then for linked tree, we only need O(1) complexity to go the node’s parent cause we have reference to each node’s parent. The total complexity is O(n) + O(1) which is O(n). For sequential tree, finding the node itself is the same O(n), for finding its parent, cause we have the node’s index, we can simply calculate its parent index and go to the node’s parent’s index in Array with O(1) complexity, same as linked tree. So the worst case complexity of finding parent is O(n).

For printing all nodes, no matter what condition it is, we need go each node one by one. So the average complexity and worst case complexity are the same. As total n nodes, the worst case complexity is O(n).

**Question C2:**

Worst case: 14 billion seconds

Average: 7 billion seconds

In the worst case, all the questions we ask are incorrect except the very last one. So we need ask 14 billion questions which cost 14 billion seconds.

In the average case, any question could be correct, and the probability of any question to be correct is the same, so we add the time\_of\_correct\_questions\*probability\_of\_correct\_question together, then we can have average\_questions = 1/14billion + 2/14billion + 3/14bilolion + … + 14billion/14billion, which is 7billion. So in average case we need ask 7billion questions, and each question cost 1 second, so in average astronaut need 7billion seconds before having a drink.

Worst case: Log(14billion)

Average: Log(14billion)

In this way of asking questions, no matter what age the alien is, we need keep asking questions until the last correct answer left. In each question, we can exclude half of numbers, for example, if the question is “Are you at most 7 billion years old?”, if the answer is yes, numbers larger than 7 billion excluded, if answer is no, number less than 7billion excluded. We keep asking until only one number N left, then we ask, “Are you N years old?”. Then we can get the right answer. So in both cases we need ask log(7billion) questions.