

Report for Algorithms & Analysis Assignment 1

Written by



Nhi Huynh s3652578

Quan Fan s3516995

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

Contribution Sheet

For each of member in your team, please write your name, student number, contribution percentage to assignment and your signature. The percentages should sum to 100%.

Name	Student Number	Contribution Percentage	Signature
Nhi Huynh	s3652578	50%	
Quan Fan	s3516995	50%	

Task B - Experimental Setup

Data Scenarios

We use three different sizes for this analysis: splitting nodes 100 times, 1000 times, and 2000 times. The data we use to grow the Binary Partitioning Tree (BSP) is provided in the "BSPTree_combined.txt" file on Canvas.

As such, we have six BSP trees in total: three sequential trees of three different sizes and another three linked trees of three different sizes. Having two representations of BSP trees allows us to compare the time complexity of both data structures as required by the assignment brief.

Rationale for our data scenarios

We choose 100, 1000, and 2000 because we think these numbers represent a small-sized tree, a medium-sized tree and a large-sized tree. We also think that the order of growth provided by 100, 1000 and 2000 is sufficient to differentiate the time complexity needed for this analysis.

Indeed, the final time data generated provide clear distinction between the complexity for sequential tree and linked trees in terms of splitting nodes, searching for nodes and traversing through nodes. More detailed discussion will be provided below.

How we generated BSP trees

We generated the linked representation and the sequential representation by reading the "BSPTree_combined.txt" file line by line. The first line is used to set the root of the tree. The rest of the file continues to be read until we achieve the order of growth we desire.

A snippet of code below shows how we grow the BSP trees from empty to the desired order of growth.

```
public class TimeAnalysis
{
    String dataFile = "C:\\Users\\uyenn\\eclipse-workspace-oxygen\\AA-A1\\src\\BSP_combined.txt";

    protected SequentialRepresentation<String> seqTree100 = new SequentialRepresentation<String>();
    protected SequentialRepresentation<String> seqTree1000 = new SequentialRepresentation<String>();
    protected SequentialRepresentation<String> seqTree2000 = new SequentialRepresentation<String>();

    protected LinkedRepresentation<String> linkedTree100 = new LinkedRepresentation<String>();
    protected LinkedRepresentation<String> linkedTree1000 = new LinkedRepresentation<String>();
    protected LinkedRepresentation<String> linkedTree2000 = new LinkedRepresentation<String>();

    //Measure time taken to split nodes n times in a BSP Tree
    public void timeSplitNodes(BSPTree<String> tree, int times) {
        BufferedReader csvReader;
        String row = null;
```

```

int count = 0;
long startTime;
long endTime;
double estimatedTime;

try {
    //Read BSP_combined.txt
    csvReader = new BufferedReader(new FileReader(this.dataFile));
    row = csvReader.readLine();

    //Start counting time
    startTime = System.nanoTime();

    //Set the first string as root node
    tree.setRootNode(row);

    //Continue to split nodes n times
    while (count <= times) {
        row = csvReader.readLine();

        if (row != null) {
            String[] data = row.split(" ");
            String srcLabel = data[0];
            String leftChild = data[1];
            String rightChild = data[2];

            tree.splitNode(srcLabel, leftChild, rightChild);
        }
        count += 1;
    }

    //Record endtime and compute estimated time
    endTime = System.nanoTime();
    estimatedTime = ((double) (endTime - startTime))/Math.pow(10, 9);

    //Print result and close the file reader
    System.out.println("\nTime taken to split " + (String.valueOf(times)) + " times = " + estimatedTime +
" sec ");

    csvReader.close(); } //The rest of the code is left out...

```

How we performed timing BSP trees

We categorise our data based on these criteria

1. Linked representation vs. Sequential representation
2. BSP with distinct sizes after being split 100 times, 1000 times, or 2000 times
3. The time measured for each of these basic functions: splitting the nodes, searching for nodes (including searching for the node itself, its parent node, or its children nodes), and traversing through all nodes (preorder, inorder, or postorder)

Therefore, we performed 42 unit tests in total. We use

```
long startTime = System.nanoTime();  
long endTime = System.nanoTime();  
double estimatedTime = ((double) (endTime - startTime))/Math.pow(10, 9);
```

to measure the time taken for each test.

The timing result collected is as follows.

Table 1. Splitting nodes

	sequential trees	linked trees
split 100 times	0.0010485 sec	0.0788537 sec
split 1000 times	0.0081949 sec	2.8158585 sec
split 2000 times	0.0120031 sec	16.8980055 sec

Table 2. Finding node, children, parent (node 1020) for BSP trees split 100 times

	sequential trees size 100	linked trees size 100
search for node 1020	0.0000098 sec	0.0002581 sec
search for children nodes of node 1020	0.0000581 sec	0.0003885 sec
search for parent node of node 1020	0.0000639 sec	0.000256 sec

Table 3. Finding node, children, parent (node 3110) for BSP trees split 1000 times

	sequential trees size 1000	linked trees size 1000
search for node 3110	0.0000748 sec	0.0061365 sec
search for children nodes of node 3110	0.0000914 sec	0.0061218 sec
search for parent node of node 3110	0.0000994 sec	0.0059947 sec

Table 4. Finding node, children, parent (node 3960) for BSP trees split 2000 times

	sequential trees size 2000	linked trees size 2000
search for node 3960	0.0000688 sec	0.0325966 sec
search for children nodes of node 3960	0.0001522 sec	0.0276579 sec
search for parent node of node 3960	0.0002437 sec	0.0291946 sec

Table 5. Inorder traversal for BSP trees split 100 times, 1000 times, or 2000 times

	sequential trees	linked trees
size 100	0.0002326 sec	0.0001688 sec
size 1000	0.007928 sec	0.0076756 sec
size 2000	0.0255303 sec	0.0372302 sec

Table 6. Postorder traversal for BSP trees split 100 times, 1000 times, or 2000 times

	sequential trees	linked trees
size 100	0.0002046 sec	0.0002288 sec
size 1000	0.007167 sec	0.0089465 sec
size 2000	0.0274274 sec	0.0348541 sec

Table 7. Preorder traversal for BSP trees split 100 times, 1000 times, or 2000 times

	sequential trees	linked trees
size 100	0.0004317 sec	0.0001129 sec
size 1000	0.0077055 sec	0.0135831 sec
size 2000	0.0289836 sec	0.0257531 sec

Evaluation

Scenario 1: Splitting nodes

We found that linked BSP trees were more than 70 times more inefficient than sequential BSP trees when it came to growing the trees (see Table 1).

Interestingly, the more times we split the nodes, the more inefficient linked BSP trees get compared to the sequential trees. The time taken escalates up to 1400 times larger when it comes to splitting 1000 nodes.

We hypothesise the reason for this is that assigning elements through array indexing is much faster than assigning new nodes through defining node references.

Scenario 2: Finding nodes, children nodes, parent node

We found that linked BSP trees were up to 143 times more inefficient than sequential BSP trees when it came to finding a node (see Table 2, 3 and 4).

Similar to the trend we found with splitting nodes, as the number of nodes in the tree increases, a linked BST tree takes even more time to find a node compared to a sequential BST tree with the same number of nodes. The time inefficiency ratio increases from 26x, 82x to 143x for BST trees being split 100, 1000 and 2000 times (see Table 2, 3 and 4).

The same pattern applies for finding children nodes and parent node of a node (see Table 2, 3 and 4).

Scenario 3: Inorder, postorder, preorder traversal

It is fascinating to find out that despite the significant time advantage sequential BSP trees have over linked BSP trees when it comes to splitting and searching for nodes, it takes approximately the same amount of time for both representations to traverse through every node.

The order of traversal does not seem to affect the time taken for both sequential and linked BST trees to traverse inorder, postorder, and preorder. As seen in table 5, 6 and 7, the average time for each BST tree to traverse are almost identical (around 0.00023 sec).

Recommendation

Taking every scenario into consideration, we strongly recommend using sequential representation for faster computation of a large number of nodes. Having fast-accessing arrays as its core implementation, sequential BSP tree ensures higher speed of 3D graphics computation, which is crucial for enhanced gaming experience.

Task C - Theory analysis

Question 1

	$O(1)$	$O(\log(n))$	$o(n)$	$o(n!)$
Find parents	No	No	Yes	No
Find a node	No	No	Yes	No
Print all nodes	No	No	Yes	No

The worst case of finding a node would be the $O(n)$. This happens when the target node is not in the tree, or it is in the very last position of the tree. In sequential or linked representation, we need to go through every node of the tree, hence $O(n)$ complexity.

Given a target node as a function parameter, once we find the target node, its parent can be easily accessed by `node.parent`. This takes constant time ($O(1)$) once the target node is found. In total, the time it takes is $O(n) + O(1)$, which makes it $O(n)$ in the worst case.

Printing all the nodes also requires traversing through every single node. As the pointer needs to go through all the nodes and print each one out, the complexity for printing all nodes is $O(n)$.

Question 2

2.1

This situation is equivalent to using sequential search to go through all the ages from 1 till the 14 billion. If the alien answers no, the pointer will move to the next node. When the alien says yes, and the pointer will stop and return the values.

So, the worst case of the sequential search will be $O(n)$. With $N = 14$ billion, it takes 14 billion seconds, or $14000000000/60/60/24/365 = 444$ years for the astronaut to guess the right age **in the worst case**.

In the average case, the complexity is $O(n/2)$. This makes $444/2 = 222$ years to guess the right age of the alien.

2.2

The worst case of the binary search tree: $O(n)$.

The average case of the binary search tree: $O(\log n)$.

By using the binary search, we use the $\log N$ to get the height of the binary tree. So the height of the binary tree will be the $\log_2(N) = \log_2(14000000000) = 30.38 = 31$ times. The average case will be 16 times.

By using the binary tree to solve the problem, the astronaut will need to ask about 31 questions to guess the alien age in the worst case. And about 16 question to guess the alien age in average case.