Carsten Hood
CSCE 221H-200

Program 2 Report
Binary Tree Traversals

Introduction

This report presents the setup and results of experiments intended to demonstrate the complexity in space and time of traversals of a simple implementation of a binary tree. Postorder, inorder, and preorder tours are examined, and recursive and iterative implementations are compared for different input tree structures. All tour algorithms return a double-ended queue representing all nodes in the tree ordered as they were traversed.

Theoretical Analysis

An inorder traversal is implemented iteratively using a loop and temporary deque. Beginning with the root, a loop examines a node *r* so long as the deque has elements or *r* is not null. In the loop if *r* represents a null value, it is added to the temporary deque and set to its left child. Otherwise it is set to the last element in the deque, which is removed, added to the output sequence, and then *r* is set to the right child. In this way every left chain is stored temporarily so that it can be visited in reverse order to satisfy an inorder tour, shown here in pseudo code:

```
Algorithm iterative-inorder (Node r, Deque dq)
    Deque tempdq
    if r is not null
        add r to back of tempdq
        set r to left child of r
    else
        set r to back of tempdq
        remove element at back of tempdq
        add r to output sequence dq
        set r to right child of r
```

The algorithm visits each node twice at most, once when it is stored temporarily and again when it is added to the output sequence. Each visit takes $O(1)$ time, performing simple operations. We have $O(2n)O(1) = O(n)$. Whether more nodes are visited once or twice depends on the structure of the tree; however, structure will not affect the complexity of the algorithm.

All recursive tree traversals are implemented using an euler tour. Only one of the three visit actions is used as appropriate; for example, the inorder tour implements the bottom-visit action of the euler tour, as shown by the following pseudo code:

```
Algorithm recursive-inorder (Node p, deque dq)
    if p is null
        return
    else if p is external
        perform external visit: add p to dq
    else
        perform left visit: do nothing
        call recursive-inorder on left child of p
        perform bottom visit: add p to dq
        call recursive-inorder on right child of p
        perform right visit: do nothing
```

Each of *n* nodes is visited 3 times, but an action is performed only one of these visits. The visit action involves adding the node to a DE queue, a constant time operation. We have:

$$3n \times O(1) = 3O(n) = O(n)$$

Therefore, recursive tours run in *O(n)* time regardless of the structure of the tree, as exactly one action will be performed per node. Because of this, *O(n)* is the best, worst, and average case complexity of all recursive traversals. Since preorder, inorder, and postorder implementations have identical complexities, a postorder traversal is not tested but is represented by the preorder traversal.

Space complexity, however, is dependent on tree structure in the recursive algorithms. Function overhead builds up in chains of a tree when a node recursively calls the tour algorithm on its children. When an external node is reached, memory is freed as the algorithm returns to the method called on a previously visited node. Therefore, recursive depth is limited by the height, or depth, of the tree. The best case occurs in a complete binary tree when the height *h* is *O(logn)*. Then the space complexity is *O(logn).* However, the worst case is *O(n)* in a tree consisting of a single chain, where the depth equals *n*, the number of nodes. Then the space complexity is *O(n)* because *n* function calls will build up before any space is freed.

The big-O constants *C* and n_0 can be found to satisfy the equation: $0 \le f(n) \le Cg(n)$ for all $n \ge n_0$, where *f(n)* represents the time used in a traversal algorithm for an input size *n*. *C* represents the maximum time required for each increase in *n*, and can be used to assign an experimental value to the algorithm's time. The constant n_0 marks the size of the tree at which an algorithm's plot fits under the equation *Cg(n)*. It is affected by operations in the algorithm that occur independent of the size *n,* which may be more noticeable for small values of *n*. These values are determined through experimentation.

Experimental Setup

The experiments described in this report were performed on a Mac computer running OS X 10.9.1 with a 2.7 GHz Intel Core i7 processor and 8GB RAM memory. C++ code was compiled using Apple LLVM 5.0.

The standard library ctime was used to time the operations. Executions were repeated at least ten times as necessary and averaged, following the formula: *iterations* = 10 + (*max_size**32)/*size* where *max_size* = 65536. Using this formula, over a million executions are averaged for the smallest input size of $n = 2$ to compensate for clock inaccuracy.

Two types of trees were generated to test the extremes of the complexities of all functions: a complete binary tree with minimal depth, and a chain tree resembling a linked list with a depth equal to its size. These two tree types were chosen to represent the range of input structures, being the best and worst cases for a tree structure. They were initialized prior to timing with consecutive integer values and sizes ranging over powers of two from 2 to 65536, the maximum size permitted by the system due to the limited space of the stack. A recursive preorder traversal and both iterative and recursive forms of an inorder traversal were tested. This allows for a direct comparison between iterative and recursive algorithms using the different inorder traversals. The recursive algorithm will use more space on the stack, but I also expect that it will be more efficient. Comparison between the recursive inorder and preorder traversals is expected to reveal their similarity, even between different input structures. A simple search function implemented using the traversals was also tested by repeatedly generating a random value within the bounds of the tree and finding it. Averaging the time of searching many random values can be expected to represent the average time of a search for a value known to be in the tree. The search algorithm represents a practical application of the traversals, but is a less accurate means of analyzing them and is of limited importance in this report.

The file used to test time complexity was designed so that both tree types and all functions concisely share timing code. The test function is passed a number of iterations, a tree to be tested, and a string, *which_test*, which determines the function to be called on the given tree. Because timing code is shared, the differences in results can be relied on to depend primarily on the difference in the tree structures or traversal implementations.

Space complexity of certain functions was determined less efficiently by repeatedly running one method at a time with increasing input sizes of powers of 10 on a linux server using Valgrind; the resulting files were analyzed to acquire the maximum space used in the stack while the method was running, which represents the depth of the recursive functions. I expect the space efficiencies of the recursive functions to be nearly identical, but notably greater for the complete tree structure than the chain tree structure because the best and worst cases have different complexities.

Experimental Results

Figure 1 plots the running time of preorder and inorder recursive traversals of both complete and chained trees. It is intended to demonstrate the similarity between all recursive tours, regardless of tree structure or order. All four functions are nearly identical even though the two input structures represent extreme cases. This demonstrates experimentally that the best and worst case of the recursive traversals share a complexity of $O(n)$.

Figures 2 and 3 compare iterative and recursive traversals between trees of identical structure. Size is divided over the expected time, n, and plotted against time to produce linear graphs. This allows for determining experimental big-O constants from the equation $0 \leq f(n) \leq Cg(n)$ for all $n \geq n_0$. Both plots have arrows representing these values and the resulting bounding lines, $Cg(n)$. The algorithms begin with a higher cost for low sizes before settling into a linear path, which can be explained by inefficiencies in timing and certain overhead operations that require a time independent of size. Because of this, the tree structures grow more efficient on average for greater sizes. Skipping the small input sizes can tighten the gap between the bounding plots and the functions' plots. The chain tree structure's algorithms even out for input sizes greater than $n = 8$. After this small size, the iterative inorder function is bounded by the horizontal line where y = 2.05E-7, so C = 2.05E-7 and $n_0 = 8$. Similarly, the big-O constants observed for the recursive preorder function are represented by the pair (8, 9.43E-08) of the form (n_0 , C). These values of n_0 are determined by observing where the plots even out into a constant path: where the line that bounds a plot intersects the plot. In figure 3 there is a similar pattern for the same algorithms executed on a complete tree. By this method the big-O constants for the iterative inorder function are determined to be (8, 2.00E-7), and (8, 9.00E-8) for the recursive preorder function. The similarity in all four pairs of constants allows for simple comparison of the algorithms' efficiencies. As expected by theoretical analysis, the recursive traversal is clearly more efficient than the iterative traversals for both tree types. The iterative method requires operating on each node up to two times, while the recursive tours visit each node once. It is also apparent that both traversal implementations are more efficient for the complete tree than the linked-list tree, as expected. The chain tree structure represents the most inefficient model of a tree. This may be due to each node having at least one null pointer; time and operations are wasted on these null children. In a complete tree, most internal nodes are full and the tree is efficiently navigated. However, even though these two trees represent extreme tree structures and a difference is observable, the difference is not very large, even for high input sizes. The complete tree provides a slightly superior performance. It appears that tree structure is barely significant for the efficiency of these traversal algorithms. This agrees with the analysis that the best-case and worst-case times for traversals share a complexity of $O(n)$.

Figure 4 plots the maximum stack memory used by a recursive preorder traversal for increasing sizes of complete and linked-list tree structures. The plot represents the depth reached by recursive traversals for varying sizes, and allows for comparison of the memory efficiencies of complete tree structures and chain structures for equal values of n. Tree sizes from 1 to 100,000 for powers of ten were used; beyond 100,000 the maximum stack memory is reached and an error occurs. It is apparent that the recursive depth of the complete structure is less than that of the chain. This difference appears slight due to the misleading nature of the log-log plot but is significant and grows with every increase in size so that the graphed functions diverge. This confirms experimentally that the best case memory efficiency, when operating on a complete tree, is $O(\log n)$, while the worst case, using a single-chained tree, is $O(n)$. The function plots scale at different rates. Recursive depth is limited by the height of the tree, which is $O(\log n)$ for a complete tree and $O(n)$ for a linked-list structure.

Summary

This report uses theory and experimentation to compare and analyze the efficiencies of tree structures and their traversal algorithms in relation to space and time. The experimental results, including data and resulting plots, confirmed the theoretical behavior of a simple binary tree structure without any contradictions. The efficiencies of three simple recursive algorithms, preorder, inorder, and postorder tours, are shown to be nearly equivalent. Though I did expect the recursive algorithms to be more efficient than an iterative implementation, I noted that the recursive inorder algorithm is nearly twice as efficient as an iterative form; while recursion may be limited by space on the stack, it is clearly a superior method for speed. This was important to me because it demonstrated the practical usefulness of recursion, which I have doubted in the past in relation to the extent the concept is emphasized in coursework.

The programming required in this assignment was not challenging but was useful in enforcing an understanding of binary tree structures. Most effort was concentrated on gathering experimental data, primarily data concerning memory use. Though I would expect it to be fast and simple to obtain the maximum memory use of the stack in a simple C++ program, doing so over a range of input sizes requires a span of uninvolving time as files and commands are navigated through a console system much older than me, with which I have little experience, spoiled as I am by modern integrated developer environments. However, the assignment certainly cemented my understanding of the workings of the binary tree structure and its traversal algorithms.

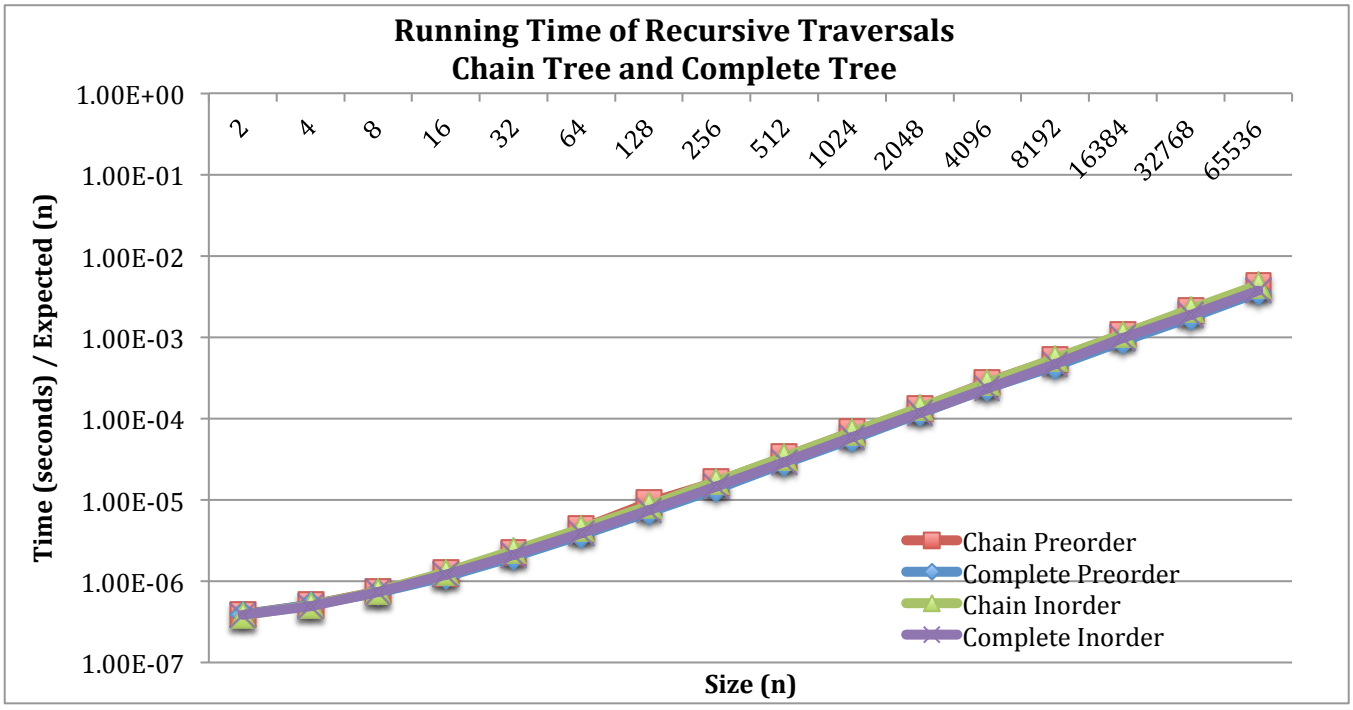

Figure 1: Execution time in seconds vs. tree size for recursive preorder and inorder traversal functions, tested twice each with a chained tree and a complete tree.

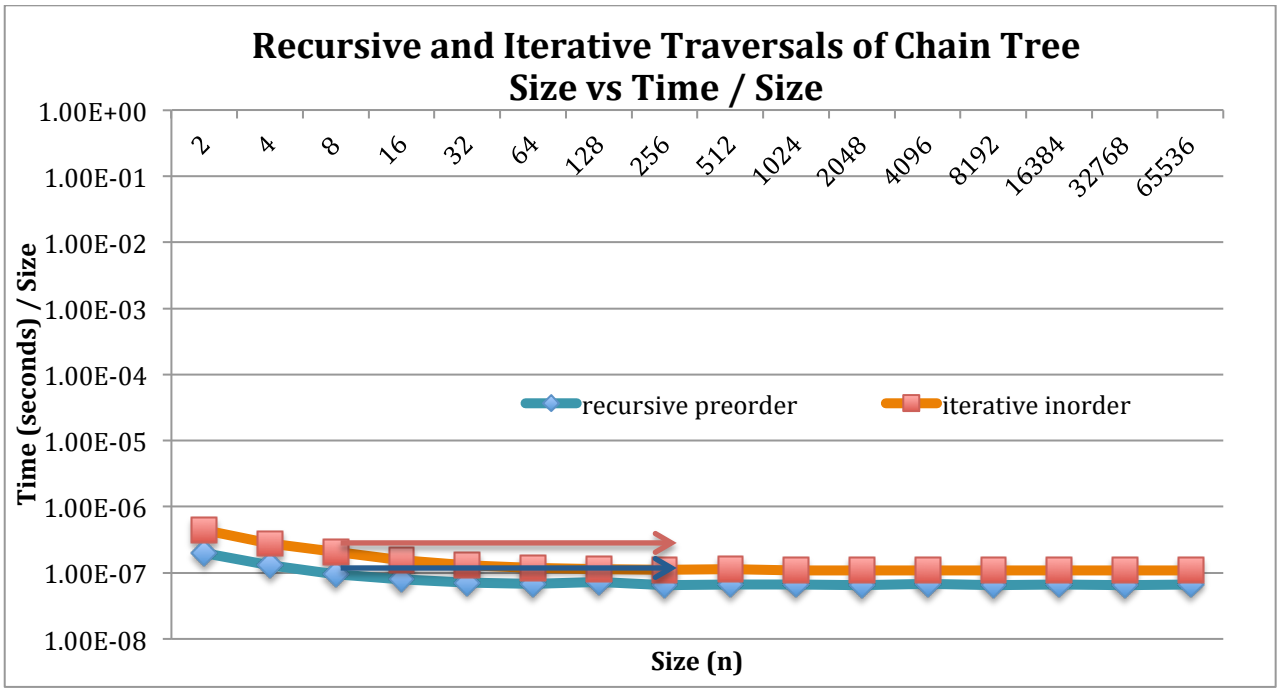

Figure 2: Time/size vs. size of recursive preorder and iterative inorder traversals of a binary tree consisting of a single chain. Horizontal arrows show the bounding big-O functions, the arrows' start representing n_0 and their levels representing C.

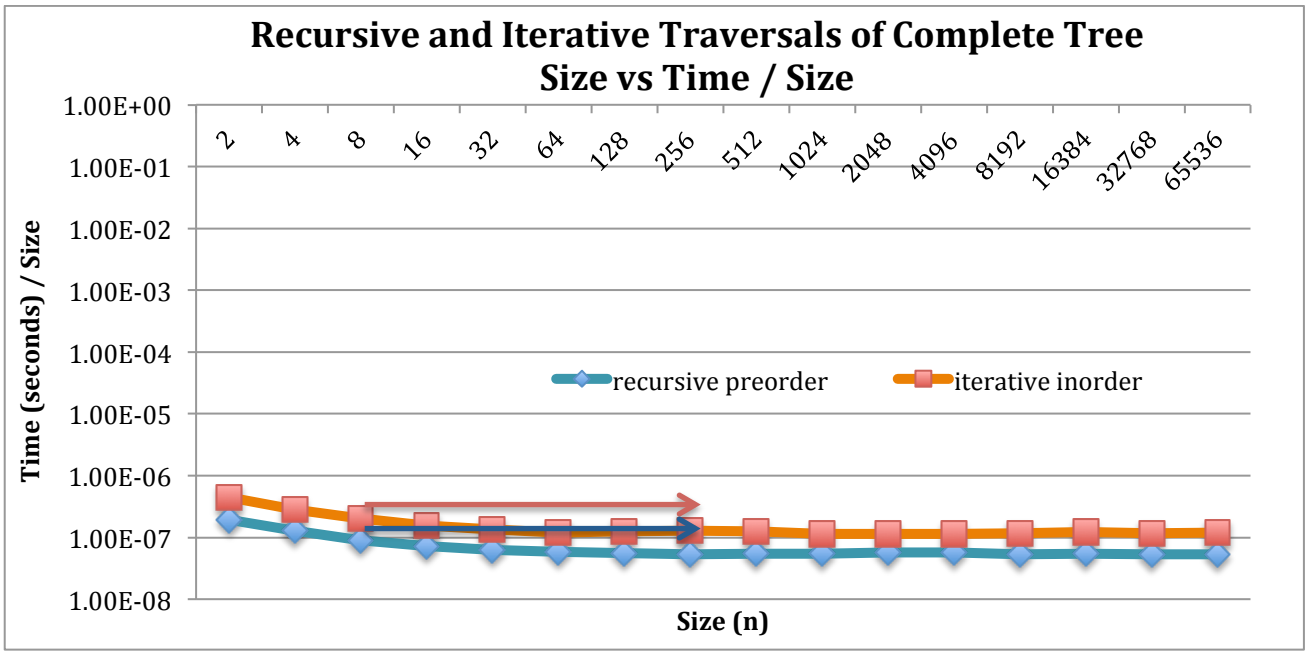

Figure 3: Time/size vs. size of recursive preorder and iterative inorder traversals of a complete binary tree on a log-log plot. Horizontal arrows show the bounding big-O functions, the arrows' start representing n_0 and their levels representing C.

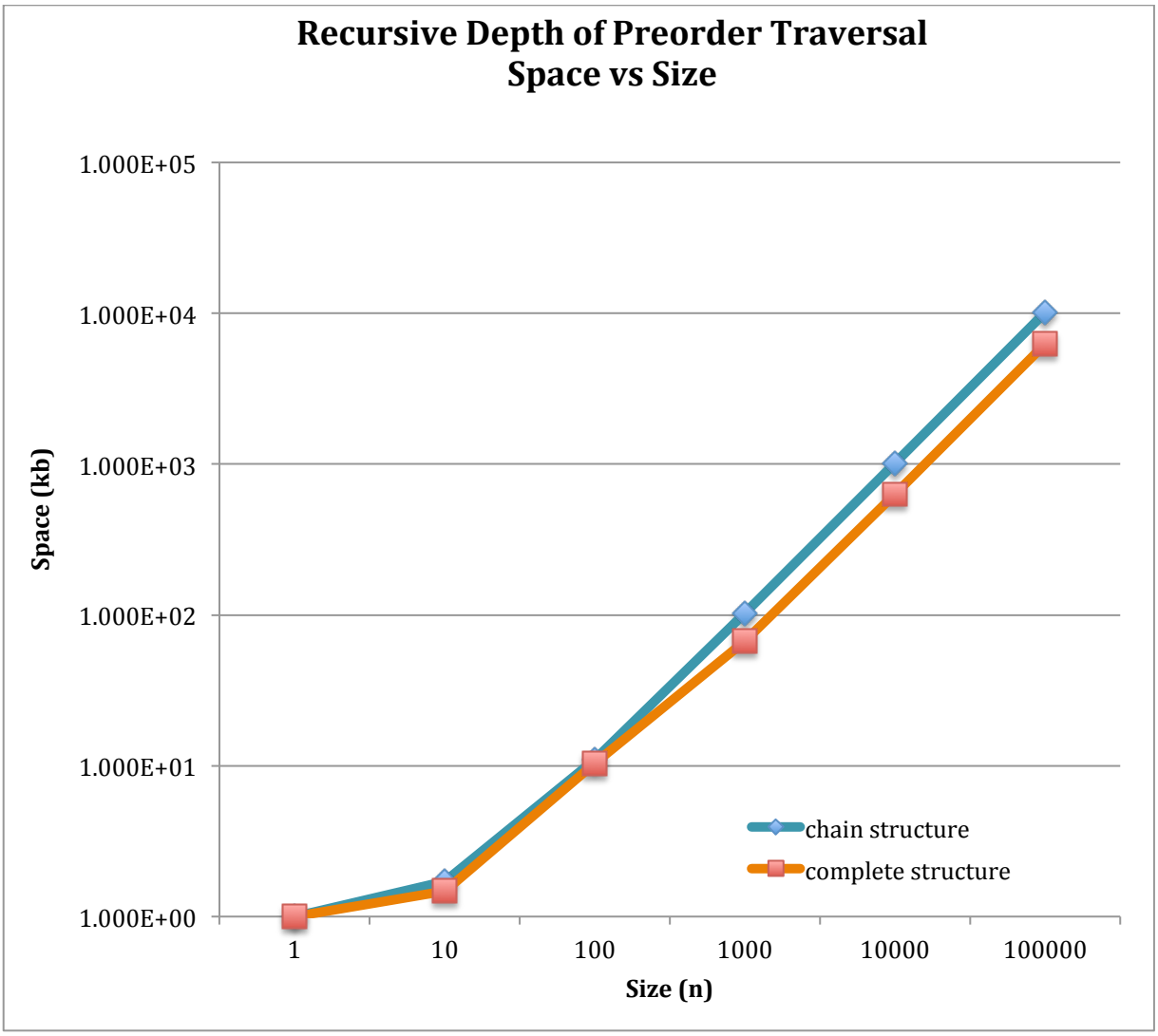

Figure 4: Maximum space used on the stack during recursive function calls vs. the tree size on a log-log plot. A complete binary tree and a binary tree consisting of a single chain are used. Space usage represents maximum recursive depth.