Evaluation of Real-Time Recursion and Iteration

Science Research, Dr. Nerurkar

Matt Nappo

January 19, 2020

1 Background

It is a fact that software cannot exist without data. All programs, in one form or another, manipulate data. So, it is extremely important that programs are able to maintain organization of that data. Organization of data means that certain functions on that data are efficient, such as reading data, writing data, changing data, searching for data, and deleting data. As a result, the study of *data structures and algorithms* is the true core of computer science. Data structures store data in an efficient manner, and algorithms operate on a data structure to do some function with the data. The algorithms are typically specialized for each structure. This research project investigates two ubiquitous data structures: the linked list and the binary search tree, and evaluates the effectiveness of implementing their algorithms in a recursive versus an iterative fashion.

1.1 The Binary Search Tree

A binary search tree has the following criteria [1]:

1. All nodes must have no more than two children.
2. Let `N` be a node where `N.left` is the left child of `N`, `N.right` is the right child of `N`, and `N.val` is the value of `N`. All binary search trees must satisfy the following for all nodes in a tree:
   (a) `N.left.val < N.val`
   (b) `N.right.val > N.val`

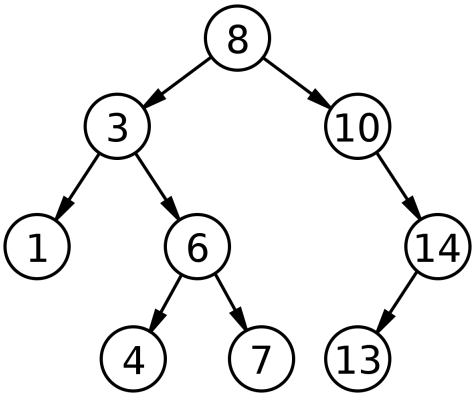

Figure 1: A binary search tree (BST)

1.2 The Linked List

Nodes in a singly-linked list contain data, and a reference to the next link of the list. A singly-linked list is a collection of these nodes, where each node points to the next.

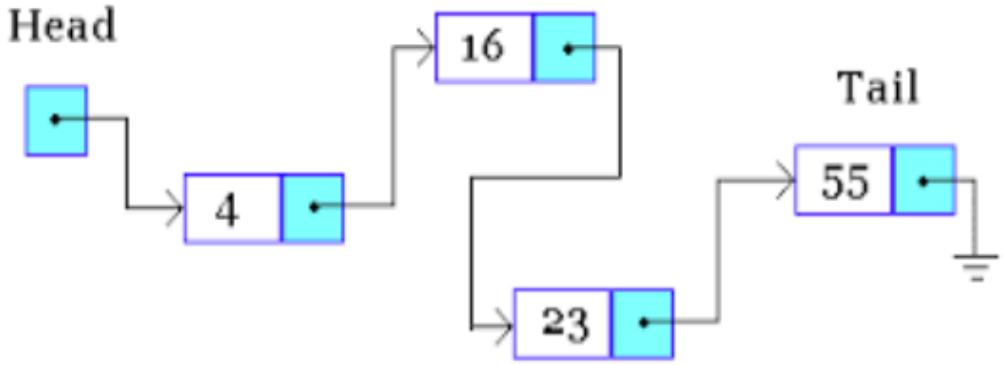

Figure 2: A singly-linked list

1.3 Algorithms

An algorithm is an ordered set of instructions to complete a task. The following algorithms (from now referred to as methods or functions) are implemented on both the binary search tree and the singly-linked list:

- insert: add an element to the structure.
- delete: remove an element from the structure.
- search: find an item in the structure given its value.
- sort: sort the data in the structure in ascending order.
- get (list only): return the node at the given index.

There are many ways to implement the same algorithm. Two main paradigms include recursion and iteration. A recursive method calls itself on a smaller substructure of the parent structure. A recursive algorithm is similar to a recursive definition in mathematics, such as the Fibonacci sequence defined as

$$F_n = F_{n-1} + F_{n-2}.$$

Note that with a recursive definition, there must be a set of base conditions. In the case of the Fibonacci sequence, the base condition is that $F_0 = 0$, $F_1 = 1$. Similarly, all recursive algorithms must have a *base case*, a condition that, when satisfied, will signal that the algorithm is complete.

Iterative algorithms execute all instructions in a completely linear fashion, directly *iterating* through all of the required steps composing the algorithm.

Both the binary search tree and the linked list are *recursive data structures*, which means that each sub-structure is of the same type as its parent structure. For example, all sub-trees of a binary search tree are also binary search trees. Each sub-list of a list is also a list. Refer to Figure 1 and confirm for yourself that this statement is true: each BST node holds the properties of an entire BST, and each linked-list node holds the properties of an entire list. Recursive methods can therefore break the structure into smaller pieces, and operate on those smaller pieces as if it were the complete structure. Iterative methods operate on the structure as a whole.

This paper seeks to explore the differences in speed between both types of algorithms: iterative and recursive.

2 Research Methods

2.1 Implementation Techniques

All source code for this project is wrtten in the C programming language. C is the best choice for analysis like this because it gives the programmer high control over the system's memory, is very fast, provides a fairly linear translation to assembly code, and has a very tiny runtime. This is important, because it means that the data collected will be truly representative of the algorithm's speed, without the noise of a garbage collector or similar runtime process running in the background. This will help greatly in isolating the independent variable (recursion vs. iteration).

The codebase contains an iterative implementation of a linked list, a recursive implementation of a linked list, an iterative implementation of a binary search tree, and a recursive implementation of a binary search tree.

The codebase is also fairly consistent. The same level of programming effort was used to implement both version of both data structures. There are no optimizations done to any of the implementations. This is done to isolate the effects of the independent variable: recursion versus iteration. To view the complete code for this project, see https://github.com/xoreo/algorithms.

2.2 Testing Framework

The codebase also contains a small testing framework to collect timing data on these data structures' methods. This framework acts like the "lab" for this experiment, making it easy to measure the amount of real time it takes to run a function. It also runs a method millions / hundreds of thousands of times to ensure that the data collected is accurate.

The interface for using this framework is very simple. It contains just one method:

```
double test_time (void (*func)(void));
```

Note, `func` should be a function that thoroughly tests all methods of the given data structure.

For example, to run the `insert` method of a binary tree 10 million times, only four lines of code are required:

```
void test_insert(void) {
    for (int i = 0; i < 10000000; i++)
        insert(tree, item);
}
test_time(test_insert);
```

In addition to this, the framework includes additional Bash Scripts to ensure the greatest level of consistency in how data is collected. The scripts execute memory safety tests (via Valgrind) to detect memory leaks, and make sure that the code is compiled in a consistent way (with equal compiler flags).

3 Data

This is the data that was collected on both the iterative and recursive implementations of both data structures:

Binary Search Tree: 50,000,000 insertion operations of 50,000,000 random 32-bit unsigned integers, and 1 in-order sort of all 50,000,000 entries. This process was repeated 11 times.

Linked list: 100,000 insertion operations of 100,000 random 32-bit unsigned integers, and 100,000 get operations of each of the 100,000 nodes in the list. This process was repeated 11 times.

The linked list tests are on a smaller scale (100,000 compared to 50,000,000) because lists are considerably slower than binary search trees. It would take an extreme amount of time to insert and search 50,000,000 elements in a singly-linked list 11 times.

All data is measured in seconds.

3.1 Raw Data & Averages

Recursion vs. Iteration			
Iterative BST	Recursive BST	Iterative list	Recursive list
114.547	105.087	12.473	28.485
102.834	106.676	13.032	28.263
103.298	103.726	12.908	28.061
100.983	104.380	12.924	28.396
102.800	103.933	12.956	27.402
100.699	104.544	12.960	26.080
102.190	107.717	13.085	27.198
104.079	105.517	12.993	28.706
103.879	106.345	13.067	25.657
103.983	103.091	12.730	25.923
104.354	95.540	13.045	25.546

Average speed			
Iterative BST	Recursive BST	Iterative list	Recursive list
103.968	104.232	12.925	27.247

3.2 Bar Graphs

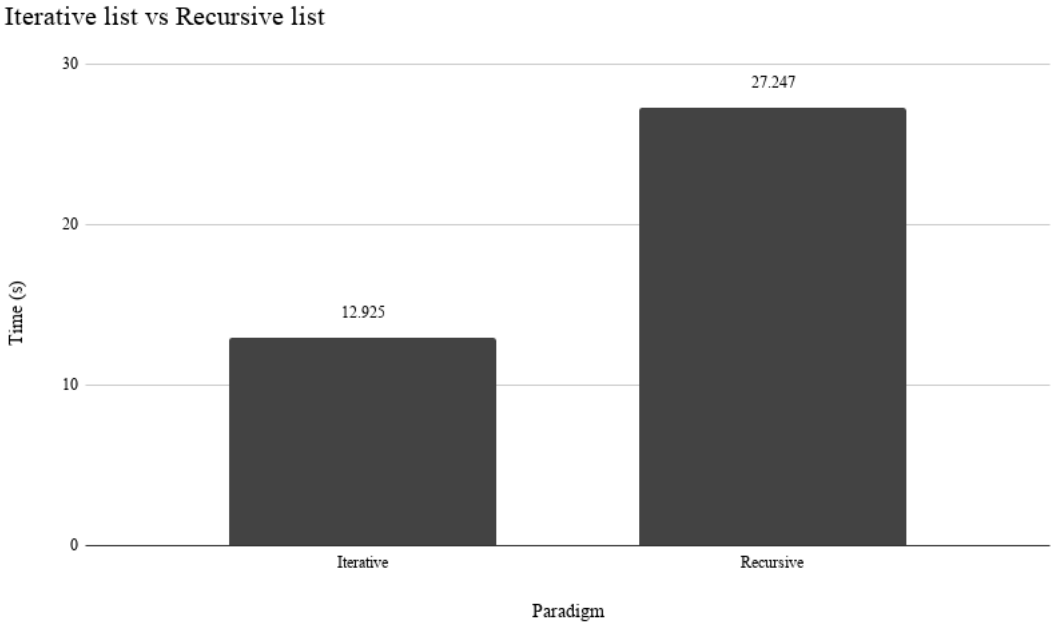

Figure 3: Average speed (list)

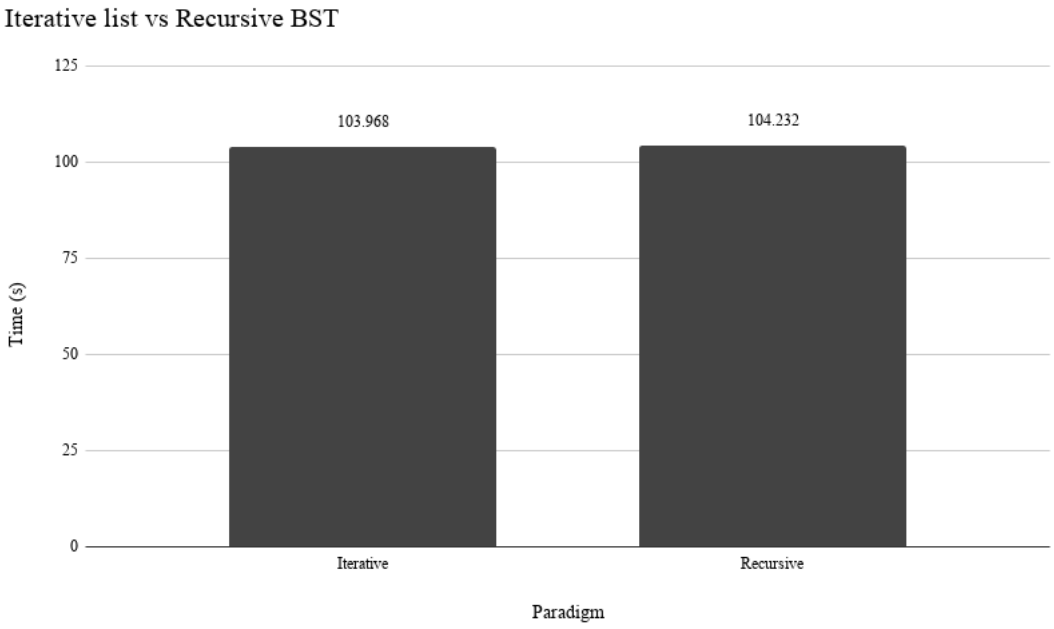

Figure 4: Average speed (BST)

4 Analysis

The iterative linked list is 212 % faster than the recursive linked list. Recall that this is for an input size of 100,000. For the BST, both the recursive and iterative implementations are very similar. On average, the iterative BST is only 264ms faster than the recursive tree.

Binary search trees are $O(\log n)$ worst-case time for insertion and searching, whereas linked lists are $O(n)$ for these operations [1]. So, it will require a larger input size for the slow effects of recursion to be visible like they were in the recursive linked list. Even $n = 50,000,000$ operations is not large enough for this to be the case, as logarithmic time is extremely fast. Perhaps an input size of $1,000,000,000$ would be large enough to see a greater difference between the two implementations, but it would take too much time for such an input size to execute on the hardware available to me. Because the linked list is $O(n)$, it did not take a very large input (only 100,000) to see a large difference between the recursive and iterative implementations.

4.1 Bias

The only bias I can see in this experiment is that my insertion method for the linked list is really an append function; it appends nodes to the end of the list. This is an $O(n)$ operation. A head-first insertion (prepend) would be $O(1)$ constant time. However, this should not have a large effect, or any effect, on the difference between recursion and iteration, because both implementations used an $O(n)$ append function.

4.2 Low-Level Assembly Analysis

Why is recursion slower? Recall that a recursive function repeatedly calls itself. But invoking a function is not an atomic operation; there are many operations that must be done to simply invoke a function:

1. The arguments of the function are pushed onto the stack.
2. The memory location of the function's return address is allocated and pushed onto the stack.

3. The function is called, and the arguments are popped from the stack.

4. The function runs to completion, and the return value is popped from the stack.

This amounts to several additional *expensive* assembly instructions that are not needed for an iterative method, because iterative methods do not need to repeatedly call additional functions; they simply run the function code linearly.

Here is a comparison of the assembly generated by GCC for the iterative and recursive insertion method of a linked list:

```
Iterative Insert:

insert:
.LFB9:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    movq    %rsi, -32(%rbp)
    movq    -24(%rbp), %rax
    movq    (%rax), %rax
    movq    %rax, -16(%rbp)
    jmp .L10
.L11:
    movq    -16(%rbp), %rax
    movq    (%rax), %rax
    movq    %rax, -16(%rbp)
.L10:
    movq    -16(%rbp), %rax
    movq    (%rax), %rax
    testq   %rax, %rax
    jne .L11
    movl    $16, %edi
    call    malloc@PLT
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    -32(%rbp), %rdx
    movq    %rdx, 8(%rax)
    movq    -8(%rbp), %rax
    movq    $0, (%rax)
    movq    -16(%rbp), %rax
    movq    -8(%rbp), %rdx
    movq    %rdx, (%rax)
    movq    -24(%rbp), %rax
    movl    8(%rax), %eax
    leal    1(%rax), %edx
    movq    -24(%rbp), %rax
    movl    %edx, 8(%rax)
```

```
Recursive Insert:

insert_:
.LFB9:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    movq    %rsi, -32(%rbp)
    movq    -24(%rbp), %rax
    movq    (%rax), %rax
    testq   %rax, %rax
    jne .L10
    movl    $16, %edi
    call    malloc@PLT
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    -32(%rbp), %rdx
    movq    %rdx, 8(%rax)
    movq    -8(%rbp), %rax
    movq    $0, (%rax)
    movq    -24(%rbp), %rax
    movq    -8(%rbp), %rdx
    movq    %rdx, (%rax)
    jmp .L9
.L10:
    movq    -24(%rbp), %rax
    movq    (%rax), %rax
    movq    -32(%rbp), %rdx
    movq    %rdx, %rsi
    movq    %rax, %rdi
    call    insert_
```

As seen in the assembly code above, the recursive method contains many expensive instructions, such as `jne`, `jmp`, and `call`. The `call` is especially expensive, as it calls itself, repeating the entire function call process described above. The iterative assembly code consists almost entirely of `movq` instructions, which are very fast. There is one loop (subroutine `.L11`), which has three fast `movq` instructions, and no expensive function calls. This is exactly why iteration is faster than recursion. Although both methods are of the same time complexity, the computer needs to do much more work to execute a recursive function. The CPU needs to execute all of those extra stack operations which manage functions being called. Recursion is not a zero-cost abstraction. If a list has n elements, then the recursive insertion method will have to execute those extra instructions n times (in the worst-case). The iterative insertion will only require those instructions one time, no matter how large n is. This is because iterative functions are only called *once*.

5 Conclusion

This experiment has proven that recursion is slower than iteration for binary search trees and singly-linked lists. Although this research only studied two data structures, the results can probably be applied to other similar recursive structures. The extra assembly instructions required for recursion will be required for any recursive function, not just those of BSTs and lists.

This research has also shown that implementation is just as important as theory. Two algorithms with the same time complexity can execute dramatically differently in real time. For example, appending to a linked list is an $O(n)$ time operation. However, the data in the above table shows that the $O(n)$ recursive append was 2.12 times slower than the $O(n)$ iterative append.

Although usually slower, recursion still has some benefits. While programming, I found the recursive solution to be *much* more elegant than the iterative solution. A binary search tree is so clearly and beautifully a recursive structure, so implementing recursive methods on a recursive structure is most logical; the structure is set up perfectly for recursion. For example, consider the recursive binary tree insertion:

```
if (data < node->data) {
    if (node->left != NULL)
        insert_(node->left, data);
    else
        node->left = init_node(data);
}
else if (data >= node->data) {
    if (node->right != NULL)
        insert_(node->right, data);
    else
        node->right = init_node(data);
}
```

This code is so clean and symmetrical, whereas the iterative implementation of the same method is the exact opposite. Recursive functions harnesses the power stored in the recursion of the structure.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.