COPYRIGHT © 2022 Jason Kotowski, All rights reserved.

**Iteration vs Recursion**

Jason Kotowski (jk234@uakron.edu)

**Introduction**

Iteration repeats a single block of code in a loop, and recursion calls the same function again in a loop. Recursion is often used to simplify code when dealing with the insertion and deletion of values into a data structure. While it appears smaller and cleaner than an iterative process, it comes at the cost of increased time complexity.[[1]](#footnote-0)

The purpose of our research is to determine the efficiency of using iteration to insert elements into a data structure compared to the efficiency of using recursion. Our goal is to determine if recursion has a significant effect on the speed at which a program runs. The experiment will be using two different data structures: arrays and linked lists.

**Discussion**

Hypothesis: Recursive insertion vs iterative insertion into a linked list will have a greater gap in program speed than recursive insertion vs iterative insertion into an array. Arrays are a very simple linear data structure. They contain a series of same-type elements that are individually referenced in contiguous memory locations. If the array is static, the size of the array is set during compilation. If the array is dynamic, the size of the array can change during the program. The experiment used in this paper uses a static array. On the other hand, a linked list is dynamic in size by default. They can grow based on the memory allocated, and the elements do not need to be manually shifted.[[2]](#footnote-1) The main problem comes with traversal. Arrays have the benefit of random access, while linked lists require one to start from the first node and work up to the head. In addition, changing the pointers is also more difficult with linked lists compared to arrays. One final concern about the data structures is the property of data locality or “cache-friendliness.” The data in an array is much closer to the cache, thereby, can be accessed more easily. This is also where linked lists fail. Linked lists not being contiguous blocks of memory causes the performance to suffer when trying to access the data.[[3]](#footnote-2)

Both data structures have their benefits and weaknesses. The thing we would like to test these data structures on is how well, or how poorly, they deal with recursion. Usually, with list-based data structures, such as arrays and linked lists, the program time is calculated linear time. The Big-O notation is stated as “O(n).” [[4]](#footnote-3) For this experiment, we will be using recursion in addition to iteration. Recursion is calculated in exponential time. The Big-O notation is stated as “O()(a > 1).”[[5]](#footnote-4) In terms of Big-O, exponential time takes way longer than linear time. A few questions arise when combining linear data structures with recursive traversal. How great of an effect will it have on the speed of the program? Is recursion preferable to iteration in any situation? This experiment may provide some answers to these questions.

**Specifications**

IDE: CodeBlocks 20.03

Device: Alienware 15 R3

Processor: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50 GHz

System Type: 64-bit operating system, x64-based processor

**Methodology/Code Samples**

The program is separated into several files. The main function, which builds the initial data structures, is located in the main.cpp file. The declarations for each data structure are located in their respective files (myArray.hpp, myLinkedList.hpp). Both the iterative and recursive versions of each data structure are included. The implementations for each data structure are also located in their respective files (myArray.cpp, myLinkedList.cpp). Finally, there are several other miscellaneous files that are used for timing the data structures (Complexity\_Runtime\_DS\_Example.cpp, Complexity\_Timer.hpp, Complexity\_Recorder.hpp). These files were provided by the University of Akron Data Structures class. They were essential to properly quantifying and outputting the execution time of each data structure.[[6]](#footnote-5)

For simplicity’s sake, the program only accepts integer values for every data structure. Every structure uses the same constant integer “MAX\_SIZE”:

const int MAX\_SIZE = 10;

The integer value defines how many elements are being inserted into the data structure. For example, since the value is 10, the integer values 1 through 10 will populate the data structure. The value can be modified by the programmer for whatever value they desire. The values used for this experiment will be “10, 25, 50, 100, 1000, 10000.”

The program starts with the construction of the data structure. The first structure is an array:

int myArray[MAX\_SIZE];

int \*ptr = myArray;

The program creates an integer-based array named “myArray” which has the size of the “MAX\_SIZE” value. In addition, the array also requires a pointer, so one has been created as well. This is later passed into the function to print the elements of the array.

At this point, a timer is created using the Complexity\_Timer/Recorder files:

class timer

{

protected:

double start, finish;

public:

std::vector<double> times;

void record() {

times.push\_back(time());

}

void reset\_vectors() {

times.erase(times.begin(), times.end());

}

void restart() { start = clock(); }

void stop() { finish = clock(); }

double time() const { return ((double)(finish - start))/CLOCKS\_PER\_SEC; }

};

The program calls the timer class to create a timer object for the main.cpp file. It begins to calculate the seconds passed since its construction, and it will stop once the program calls for it.

With the timer running, now we can populate the data structure:

for(int i = 0; i <= MAX\_SIZE; i++)

{

myArray[i] = MAX\_SIZE - i;

}

The populated array is used for both the iterative and recursive version of the data structure. The main aspect of this experiment deals with printing the elements to standard output through traversal:

cout << "Iterative Array: ";

printArr(ptr);

timer1.stop();

cout << timer1.time() << endl;

Before the timer is stopped and the data is printed, it calls the function “printArr(int)” from the myArray.cpp file. The traversal is simple, as it uses a for loop to scan the area and output each element. On the other hand, the recursive version (printArrRec(int, int)) works differently:

void printArrRec(int myArray[], int i)

{

if (i == MAX\_SIZE) {

cout << endl;

return;

}

cout << myArray[i] << " ";

i++;

printArrRec(myArray, i); // recursive call

}

It determines if the array has reached the end. If it does, then it exits the function. Otherwise, it will print the current position (int i) into standard output. Then, it makes the recursive call. The function loops until the if statement is satisfied.

The linked list is more complex. This program uses a linked list class, as well as a node class, to construct objects that will hold the elements. The class is located in the “myLinkedList.hpp” file:

class node{

public:

int num;

node \*next;

node(){

next = NULL;

}

};

class LinkedList{

node \*head;

public:

LinkedList(){

head = NULL;

}

void appendNode(int);

void printList();

node\* gethead()

{

return head;

}

void printListRec(node \*);

};

The class holds all the essential functions for appending a node and traversing the linked list. After creating a linked list object, the program uses a for loop, similar to the one used for populating an array. It then calls a function to insert a node on the head of the list:

void LinkedList::appendNode(int num){

node \*temp = new node();

temp->num = num;

temp->next = head;

head = temp;

}

After populating the list, the program can now print both versions of the list. The iterative version is very basic for a linked list class:

void LinkedList::printList(){

if (head == NULL){

cout << "List is empty" << endl;

}

else{

node \*temp = head;

while (temp != NULL){

cout << temp->num << " ";

temp = temp->next;

}

cout << endl;

}

}

The recursive function works as such:

void LinkedList::printListRec(node \*head)

{

if (head != NULL)

{

cout << head->num << " ";

printListRec(head->next);

}

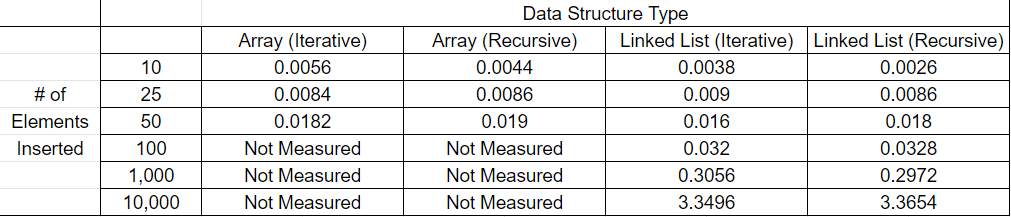
}

As can be seen above, the function is much more condensed and simplified than the iterative version. The final output, including the printed lists and the time it took to execute the functions, will be displayed in the console. This process would be repeated at least 10 times for each data structure type, and the average will be taken of those outputs.

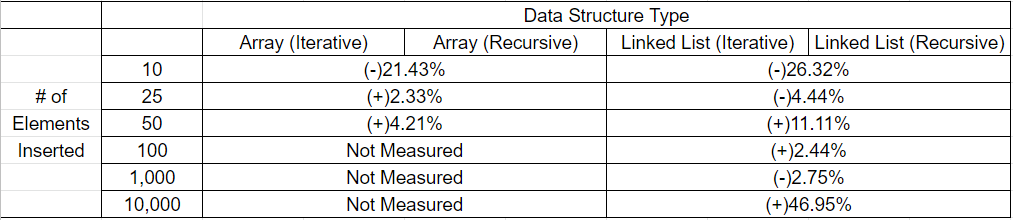
Note: various other aspects can affect the speed of the program, such as the device used and the optimization of the code. If the program was executed on a more advanced computer, or the code was better optimized, the program would execute faster, resulting in different output times.

**Results**

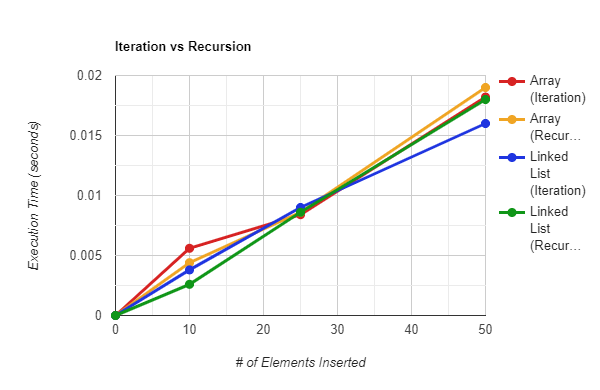
The final data is displayed in the chart below:

[[7]](#footnote-6)

Due to the restrictions of the array, the program could not handle 100 or more elements at a time. The linked lists, on the other hand, could handle a larger number of elements. Therefore, the only data for arrays that could be collected is between 10 and 50. Despite this limitation, the most interesting aspect of the results occur in the lower numbers of inserted elements. Recursion consistently had shorter execution times than iteration for less than 10 insertions. Recursion was about 21.43% faster for arrays and about 26.32% faster for linked lists. Program speed remained inconsistent beyond 10 inserted elements, even in the upper bounds. A chart calculating more time differences for iteration vs recursion is listed below:

[[8]](#footnote-7)

For most of the structures, recursion was slower than iteration. However, the correlation is very weak. The line graph below illustrates the differences in speed among the data structures between 10 and 50 elements:

[[9]](#footnote-8)

The differences in execution time nearly disappeared when testing 25 elements. The graph lines are so close together that the results barely diverge from each other.

**Conclusions**

The original hypothesis of recursive insertion vs iterative insertion into a linked list having a larger gap in program speed than recursive insertion vs iterative insertion into an array was not proven. The data proved to be too inconsistent to draw any clear conclusions on recursion vs iteration overall. The only semi-solid conclusion that can be made was from the data collected in the lower bounds. Recursion can benefit the speed of your program if you are dealing with a small number of inserted elements, but the benefits are minimal.

The experiment could benefit from more testing, including more variations of inserted elements, in hopes that more solid data can be obtained. Despite the data obtained, we can still use what we have to learn about how certain programmers might want to utilize recursion for their data structure. If you are a web designer, who does not rely too much on execution speed to make an effective program, you can stick with iteration. For someone who wants to optimize performance to the fullest extent, such as a game developer, highly-optimized recursion might help you, even if it does save mere milliseconds. It is up to the user and their specific program interests to decide whether or not recursion might be the best option for optimizing their code.

**Sources**

“Algorithms”. U1 ds03. Data Structures. College of Arts and Sciences, The University of Akron. slides 36-39.

“Lists Stacks Queues”. U1 ds04. Data Structures. College of Arts and Sciences, The University of Akron. slide 23.

*Project Artifacts*, Data Structures. College of Arts and Sciences, The University of Akron.

Roth, N., & Claesen, M. (2013, May 22). *What is a "cache-friendly" code?* Stack Overflow. Retrieved January 30, 2022, from https://stackoverflow.com/questions/16699247/what-is-a-cache-friendly-code

Shaffer, Clifford A. *Data Structures and Algorithm Analysis*, Edition 3.2 (C++ Version). P. 38, June 5, 2012.

1. Shaffer, Clifford A. *Data Structures and Algorithm Analysis*, Edition 3.2 (C++ Version). P. 38, June 5, 2012. [↑](#footnote-ref-0)
2. “Lists Stacks Queues”. U1 ds04. Data Structures. College of Arts and Sciences, The University of Akron. slide 23. [↑](#footnote-ref-1)
3. Roth, N., & Claesen, M. (2013, May 22). *What is a "cache-friendly" code?* Stack Overflow. Retrieved January 30, 2022, from https://stackoverflow.com/questions/16699247/what-is-a-cache-friendly-code [↑](#footnote-ref-2)
4. “Algorithms”. U1 ds03. Data Structures. College of Arts and Sciences, The University of Akron. slide 36. [↑](#footnote-ref-3)
5. “Algorithms”. U1 ds03. Data Structures. College of Arts and Sciences, The University of Akron. slide 39. [↑](#footnote-ref-4)
6. *Project Artifacts*, Data Structures. College of Arts and Sciences, The University of Akron. [↑](#footnote-ref-5)
7. **Chart 1:** *Chart tracking the time (in seconds) it takes to execute each data structure* [↑](#footnote-ref-6)
8. **Chart 2:** *Chart calculating the percentage differences between iteration and recursion (positive values indicate an increased time)* [↑](#footnote-ref-7)
9. **Graph 1:** *Graph plotting the execution time and number of elements inserted for each data structure* [↑](#footnote-ref-8)