malloc(): Through the Looking Glass*

Samuel Diamantstein[1], Zachary Stroud - Taylor[1], Alex Trostanovsky[1]

Abstract

malloc() is a general purpose memory allocating function for C and C++. Using malloc() and free(), programmers allocate and deallocate memory dynamically. The capacity for C and C++ to dynamically allocate and deallocate memory has resulted in their use in hardware implementations and algorithms that benefit from specialized memory management techniques. In this report, we evaluate pre-built alternative memory-allocators with respect to time complexity to determine the efficiency of malloc(). After establishing a common measure of efficacy, we provide use cases for the different dynamic memory allocators.

Keywords: malloc(), Custom Memory Allocators, Benchmarks

*This work was not supported by any organization. The testing platform detailed in this report is freely available at https://github.com/ZacASTaylor/Memory_Allocators

Email addresses: samueldiamantstein@cmail.carleton.ca (Samuel Diamantstein), zacharystroudtaylor@cmail.carleton.ca (Zachary Stroud - Taylor), alextrostanovsky@cmail.carleton.ca (Alex Trostanovsky)

Contents

1. INTRODUCTION

1.1. Context

Computer program memory can be segmented into four blocks:

1. **Text**, also known as the code segment, contains executable instructions.
2. **Data** contains any global or static variables which have a pre-defined value.
3. **Heap**, used for dynamic memory allocation.
4. **Stack** contains the function call stack, and is used for static memory allocation.

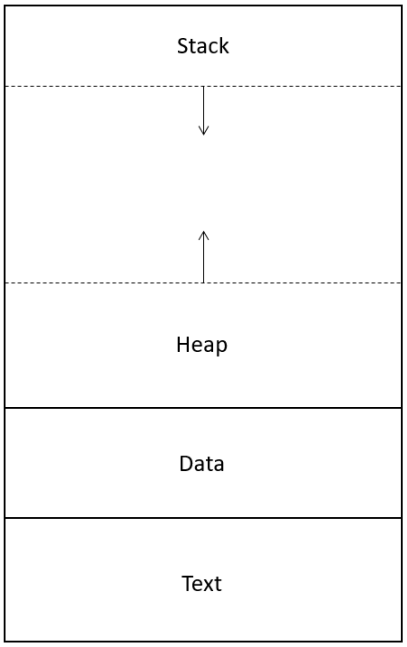

Figure 1: **Program Memory Model**

A graphical representation of the memory allocated to a process. From bottom to top: The text segment, which contains the executable instructions. The data segment, which contains global and static variables. The heap, which grows upwards as memory is dynamically allocated at runtime. The stack, grows downwards (towards the heap) with stack frames as functions are called.

Computer applications manage their memory in two broad categories: Compile time (static) allocation and Dynamic allocation. Static memory allocation entails predefining and capping the amount of memory that a program can use. Dynamic allocation enables programs to allocate memory as needed at run time. Dynamic memory addresses are stored on the heap using the predefined function `malloc()` in C and C++ which allocates the requested memory of a given size and returns a pointer to it.

In the attempt to explore an alternate means of memory allocation other than `malloc()`, we tested four types of custom memory allocators (Linear, Stack, Free List, and Free Red-Black Tree)

to explore their relative strengths and weaknesses. To further examine the possible efficiency gains of our four custom allocators, we used two data structures: Red-Black Trees and Doubly linked lists to store the memory allocated.

1.2. Problem Statement

The problem that motivates the application of the four listed custom memory allocators in conjunction with the two listed data structures is the goal of outperforming `malloc()`. `malloc()` is a general purpose memory allocator and does not offer the most efficient means of memory allocation for all use cases. We define our success in solving the problem of efficient memory allocation in terms of optimizing runtime, as measured by elapsed time for calls to allocate/deallocate

1.3. Outline

The rest of this report is structured as follows:

(i) Section 2 presents background information relevant to this project, in which we review in detail the concepts and previous works related to our project. This includes describing the behavior of the five linear allocators as well as the two abstract data types and their theoretical runtimes.

(ii) Section 3 describes the obtained results.

(iii) The results are evaluated in Section 4.

(iv) We conclude our report in Section 5. We summarize the work performed, discuss the relevance of the work in contemporary terms, and reflect on possible future work to extend the work already performed.

2. BACKGROUND INFORMATION

2.1. Data Structures

Before we delve into the implementational details of the different memory allocation algorithms, we will review the abstract data structures upon which `malloc()` and the custom algorithms are based.

2.1.1. Doubly Linked List

A doubly-linked list is a linked data structure that consists of a set of sequential nodes. Each node contains data (or payload) and pointers to its predecessor and successor. All doubly-linked lists contain a reference to their head (the first node in the list), and in some cases, a reference to their tail (the last node in the list).

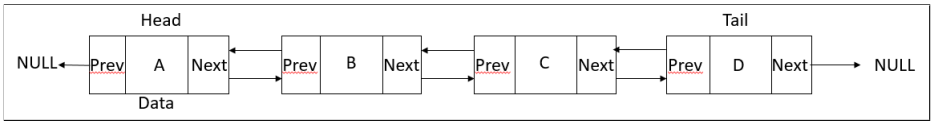

Figure 2: **A Doubly Linked List**

A doubly linked list, showing the first or head node with a previous pointer to null, a pointer reserved for the contents of the node, and a next pointer pointing to the next internal node. There are two internal nodes which are built in the same fashion. The final node is the tail node which has a previous pointer to the second last node (the last internal node), data, and a next pointer to null. The pointers pointing to null denote the point at which the data structure begins or ends.

A doubly-linked list implements the `get(i)`, `set(i, x)`, `add(i, x)`, and `remove(i)` operations (where i is the index of the list and x is the data to be set/added) in:

$O(1+\min\{i, n-i\})$ **time per operation.**[1]

2.1.2. Red-Black Tree

A red-black tree is a binary search tree in which every node has a colour that is either red or black. All red-black trees must satisfy the following two properties: [1]

- There are the same number of black nodes on every root to leaf path.
- No two red nodes are adjacent.

Adherance to these mandatory properties makes implementing Red-Black trees notoriously difficult, but it is due to these properties that Red-Black trees support a sorted set interface and allow for `add(x)`, `remove(x)`, and `find(x)` operations in:

$O(\log n)$ **in *worst-case* time per operation.** [1]

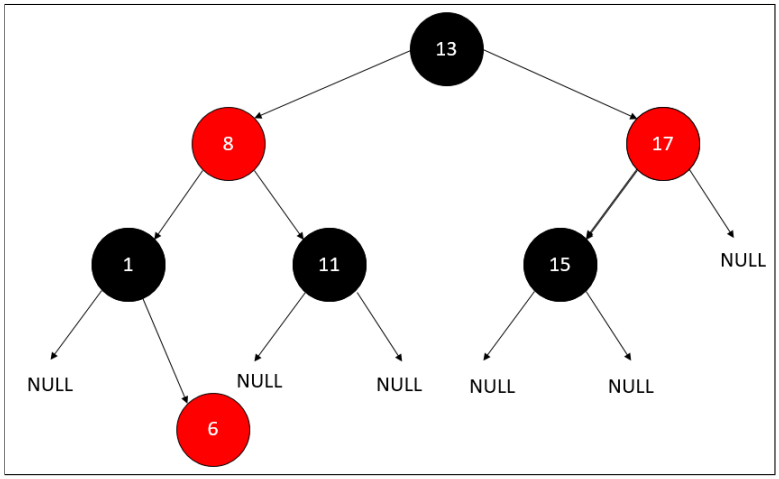

Figure 3: A Red-Black Tree

2.2. *Allocator Types*

The driving purpose behind the use of custom memory allocators is their capacity to specialize in certain forms of allocation, as well as the ability to base them on systematic data structures in an attempt to make the allocation algorithm more efficient than the traditional `malloc()`. Implementing any memory allocator requires us to keep track of which bytes in the heap are currently allocated and which are still available for use. Information is stored about the block's size as part of the block of memory that we allocate, allowing us to keep track of the total memory currently allocated.

We will describe the nature and behavior of the custom allocators and secondary data structures we use. We conclude this section with an explanation of the algorithms used by `malloc()` and `free()`.

2.2.1. *Linear*

The linear allocator is the most basic allocator to possibly construct. The linear allocator keeps track of two positions in the heap:

1. the starting point of the block of memory being allocated,
2. the ending point or the offset, which gets shifted as new blocks of memory are added in sequential order.

The linear allocator offers the benefit of maintaining low fragmentation of heap memory thanks to its simple functioning. Heap fragmentation is an inefficient and scattered use of the heap which is a common issue affecting many dynamic memory allocation algorithms. The linear allocator lacks some of the functionality offered in the following sophisticated allocators. The complexity associated with an allocation of memory is O(1) , and given the simplicity of the linear allocator, freeing memory is not possible and thus its complexity cannot be accounted for. The simplicity of the linear allocator therefore makes it an ideal means for memory allocation for simple tasks that do not require re-writing of memory in the heap.

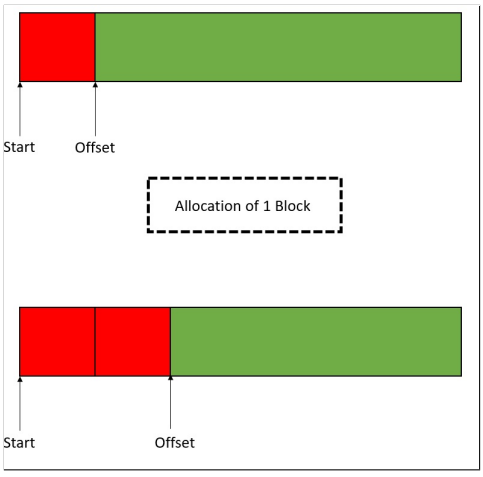

Figure 4: **Linear allocator, adapted from [6]**

An allocation occuring in a segment of memory going from one allocated portion to two allocated portions according to the prescribed fashion of a linear allocator.

2.2.2. *Stack*

A stack allocator is in principle a linear allocator. In addition to storing blocks of memory, a stack allocator also maintains headers for each block. This provides the functionality of freeing memory from the top of the stack in a Last-In/First-Out (LIFO) manner. The size of each allocated block of memory is stored in the header structure (similar to a node in a linked list).

In order to allocate memory, the offset is moved forward from its initial position to reflect the new amount of used memory, and a header is added to the block allocated to indicate its size.

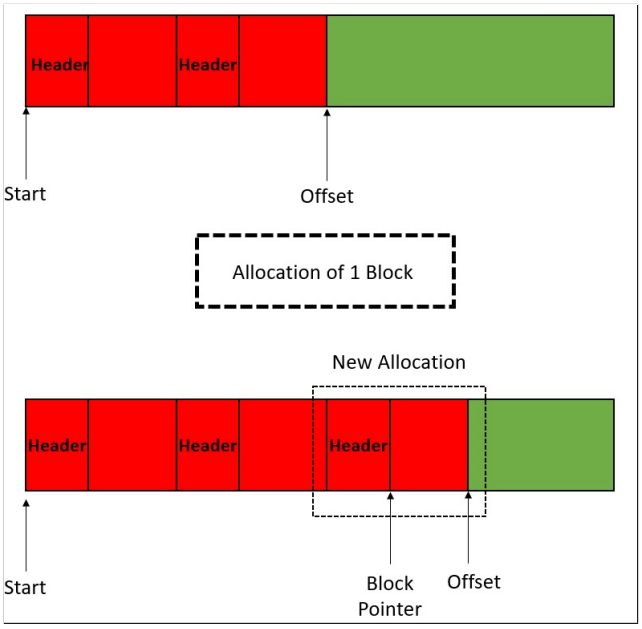

Figure 5: **Stack allocator: allocation, adapted from [6]**

An allocation occuring in a segment of memory going from two allocated portions to three allocated portions according to the prescribed fashion of a stack allocator.

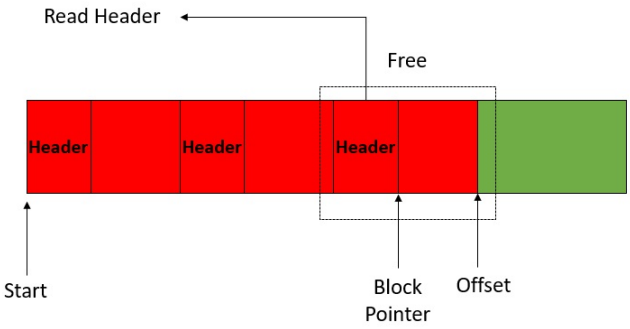

Figure 6: **Stack allocator: deallocation, adapted from [6]**

A deallocation occuring in a segment of memory going from three allocated portions to two allocated portions according to the prescribed fashion of a stack allocator

In order to free blocks of memory using the stack allocator, the header for the block is read, returning the block size. Memory is freed by moving the offset backwards to the location in memory as specified in the header of the block to be freed. Lastly, the header is removed.

The benefit of using the stack allocator is that it allows for freeing of memory, not just allocation. Additionally, the stack allocator retains the simplicity of the linear allocator while having deallocation functionality.

2.2.3. Pool

The pool allocator acts in a manner quite unlike the prior two custom allocators. A linked list is used to partition the heap into smaller segments of equal size. The linked list data structure (head, tail, nodes, and `prev` and `next` pointers per node) is located within each block of memory in order to keep track of the address of adjacent free blocks of memory. Allocation and deallocation can be thought as resembling the `pop` and `push` operations (as implemented in a stack).

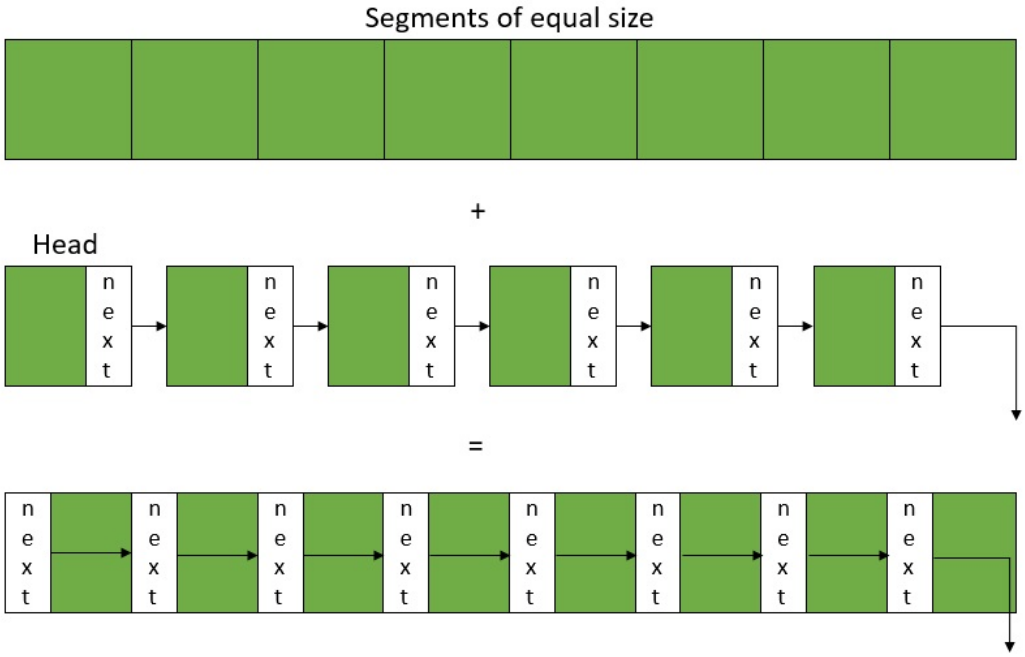

Figure 7: Segmenting the heap with a Pool allocator, adapted from [6]

When a block of memory is allocated, the first free block pointed to on the linked list block is popped.

Freeing a block of memory entails pushing the freed block onto the linked list as its first element.

Lastly, it is worth noting that the linked list is not necessarily sorted by size. The structure of the list is determined by the order of allocations and deallocations.

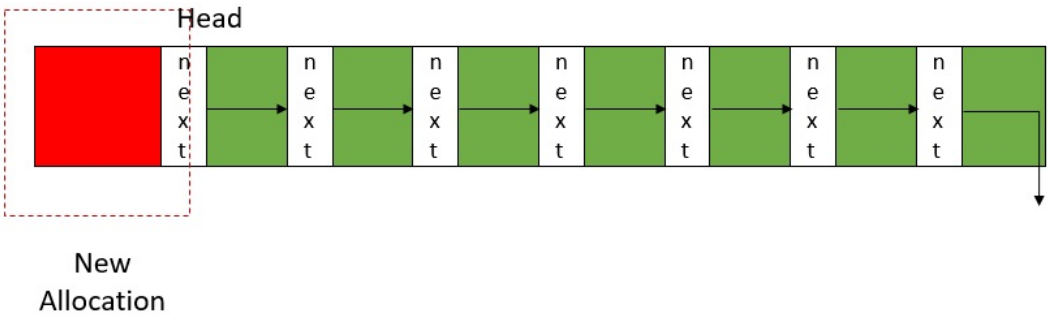

Figure 8: Allocation with a Pool allocator, adapted from [6]

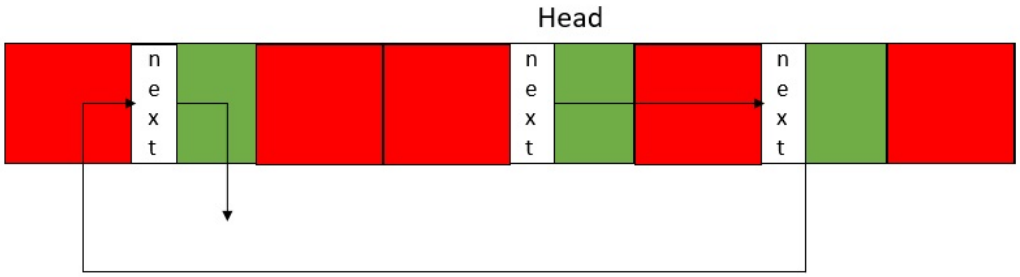

Figure 9: The Heap with a Pool allocator, adapted from [6]

2.2.4. Free list

The free list allocator is a general puprpose allocator that, opposed to the allocators above, does not impose any restriction on the order of allocations/deallocations. There are three types of free list allocators [5]:

1. **Implicit free list**
   An implicit free list is a linked list which links all blocks (free **and** in use) on the heap. Each node typically contains information regarding:

   (i) the size of each node's payload

   (ii) a pointer to the payload itself (the user data which is allocated on the heap)

   (iii) a bit which flags whether that node is allocated or free.

   Since the size of each node is known, all nodes are able to accurately point to (using `next`) their successors. The `prev` reference is not required, but is typically included to allow for forward and backward traversal while searching for freed segments on the heap.

   In addition, a reference to each node's predecessor enables a crucial operation to be performed in specific cases. When two adjacent nodes are *both* free, both memory addresses are `fused` to produce one combined, larger memory address. This greatly reduces **heap fragmentation**.

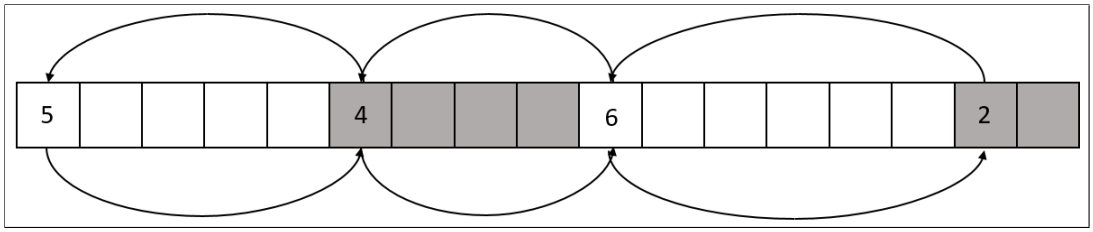

Figure 10: An implicit free list links all blocks

2. **Explicit free list**
   An explicit free list optimizes the traversal of the heap in search of a free memory address which can fit the requested byte size by the user. The explicit free list does so by *explicitly* connecting only the free nodes in the heap.

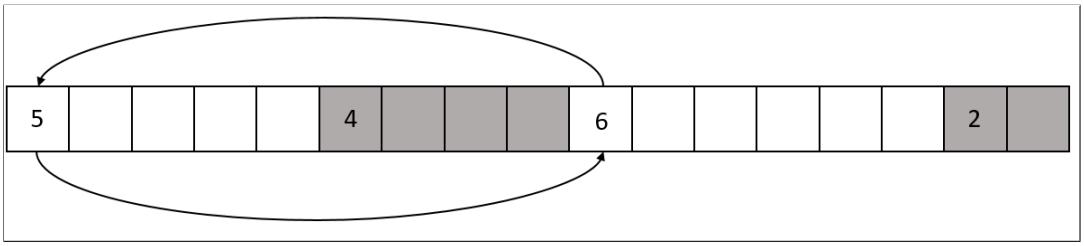

Figure 11: An explicit free list links only free blocks

3. **Segregated free list** A segregated free list further optimizes the search for a suitable free memory address by using separate explicit free lists for different size classes (See Figure 14., in Section 2.3. `malloc()`).

2.2.5. Searching algorithms

The search for a free address space on the heap can be done in two ways:

1. **First-fit**: Pick the first available free segment that can accomodate the request.
2. **Best-fit**: Iterate through the entire list, and allocate the node with the smallest block of memory in which the request can fit.

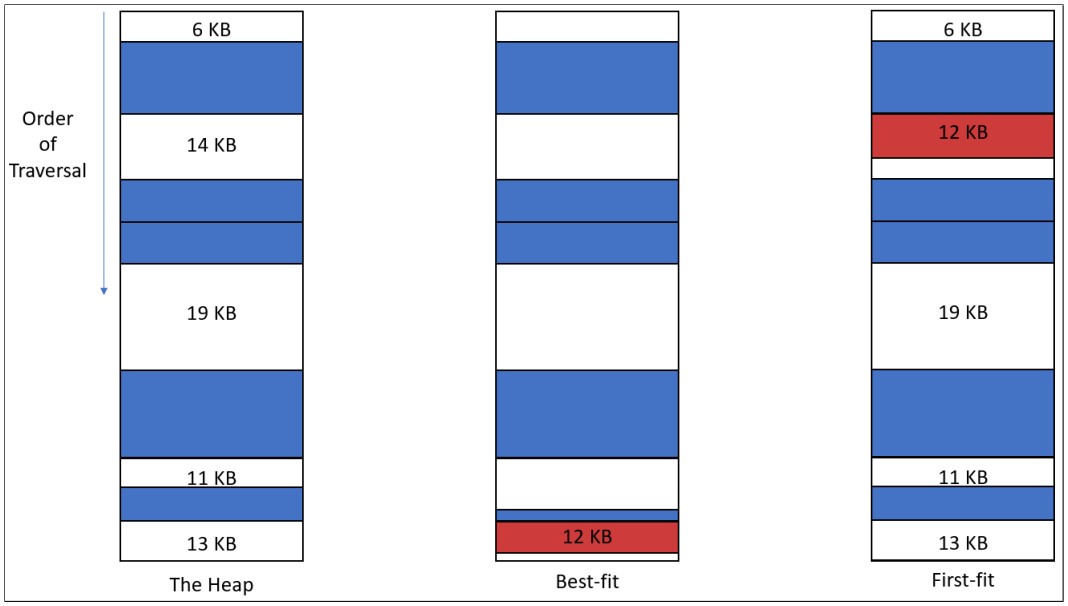

Figure 12: Best-fit vs. First-fit, adapted from [3]

The usage of a First-fit searching algorithm may be theoretically faster than Best-fit, but after numerous calls to a First-fit Memory allocator, the heap would become highly fragmented.

Best-fit memory allocation algorithms have been shown to produce the least fragmentation on real loads [4]. However, using a Best-fit search algorithm requires that an explicit free list be searched in its entirety ($O(n)$ time) to find the appropriate node.

Therefore, if the heap is segmented by way of a more efficient (sorted) data structure, the time for a Best-fit search could be potentially optimized.

2.2.6. Free Red-Black Tree allocator

A Free Red-Black Tree allocator uses a red-black tree (Section 2.1.2) structure to store, connect, and sort all free segments of the heap. Using a Red-Black tree to partition the heap should reduce Best-fit search times to $O(\log n)$.

2.3. `malloc()`

`void *malloc(size_t size)` is a C library function that allocates the requested memory (`size`) and returns a pointer to it (`void *`).

`void free(void *ptr)` is a C library function that deallocates the memory address pointed to by `ptr`.

At its core, the `malloc()` algorithm makes use of an array of doubly-linked lists. Specifically, each array index (called a **bin**) contains an *explicit free list* of allocated memory addresses. Each node in these lists contains the size information (how much size is allocated with a specific call to `malloc()` of **both** previous and next address on the heap. [2]

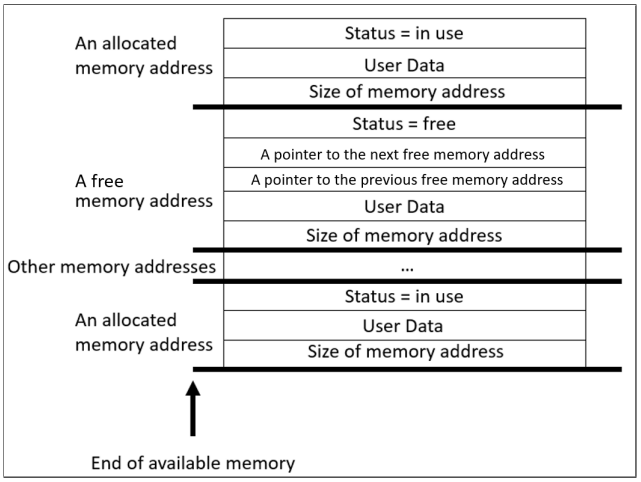

Figure 13: A single explicit free list, adapted from [2]

These *explicit free lists* are organized into **bins** which are grouped by size. Traditional implementations of `malloc()` employ 128 fixed size bins, where bins for sizes that are less than 512 bytes each hold *exactly* one size.

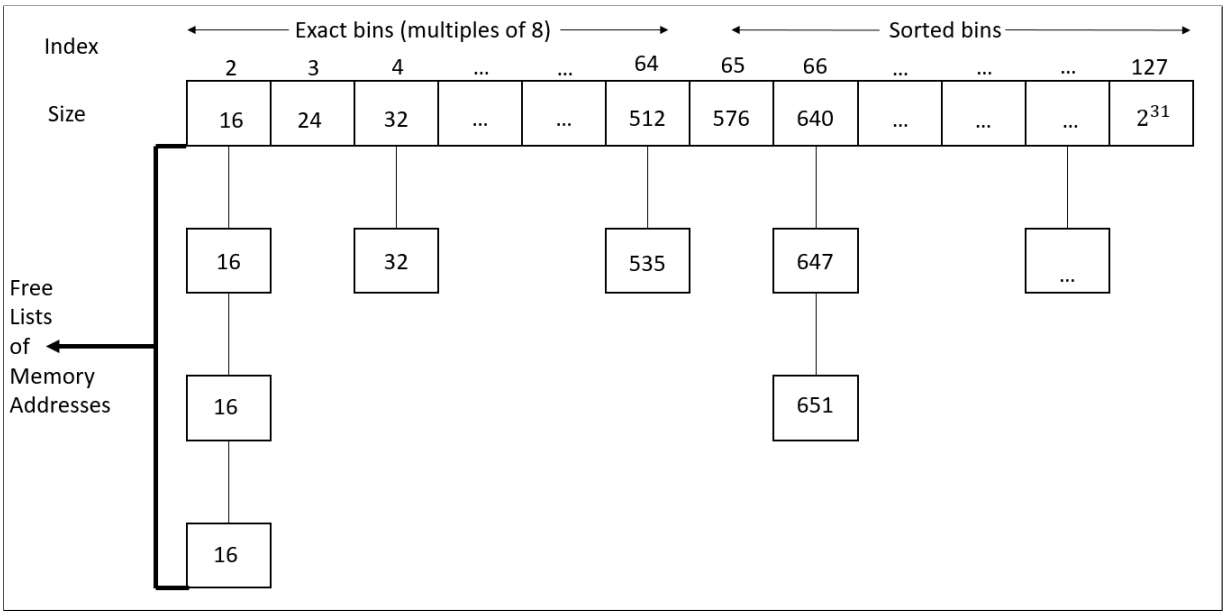

Figure 14: Available chunks in the heap, adapted from [2]

To summarize, we will use **Figure 14.** to trace a sample call to `malloc(651)`.

1. Since 651 $>$ 512, `malloc(651)` will not access the *exact bins*, but instead, look for the appropriate bin out of the *sorted array of bins*.

2. `malloc(651)` will access `index 66`

3. `malloc(651)` will perform a *best-fit* search through the *explicit free list* and reach its tail, and allocate the memory address at `node 651`

2.3.1. Impetus for Current Research

We have mentioned (Section 2.1.1) that a search through a doubly-linked list can be done in $O(1+\min\{i,n-i\})$ time. Similarly, `malloc()`, and any custom free-list based memory allocator, should be expected to run in $O(n)$ time. This is due to the fact that a free list is not sorted, so a Best-fit search through the list **must** iterate through the list in its entirety, before picking the node with the smallest block of memory in which the request can fit.

However, it is worth noting that n refers to the length of a free-list. In the case of `malloc()`, we have seen that the heap is divided into (typically) 128 such explicit free lists, which in turn, are sorted by ascending size (See Figure. 14). So, practically, a search for a free memory address on the heap by `malloc()` would access the appropriate bin, and traverse an explicit free list to find the best-fit location for allocation.

In our literature review, we have encountered sources which claim that `malloc()` is *general purpose* (i.e. must work in a variety of use cases), and so, is inefficient in use cases where the allocation requirements are specific. [6]

In addition, some sources claim that `malloc()` performs *slowly* in use cases where additional memory is required from the system, which necessitates a switch from user to kernel mode. [6]

We have decided to inspect these claims, and provide two testing suites which compare:

1. Linear and Stack allocators
2. Implicit and Explicit Free List, Free Red-Black Tree, and `malloc()` allocators

Specifically, we were interested to compare the performance of a Free Red-Black Tree allocator and `malloc()`.

2.4. Testing Methodology

All testing was done on an Intel®Core™i5-4690K CPU @ 3.50GHz processor, in an Ubuntu 16.04.4 LTS Operating System, using a GNU GCC Compiler and Development Environment. All custom allocators (Implicit Free List [7], Explicit Free List [10], Free Red-Black Tree[11]), testing source code, and documentation are available for review and replication purposes at [7]. All figures were obtained by using RStudio Version 1.1.442 ©2009-2018 RStudio, Inc.

2.4.1. Linear and Stack allocator Benchmarks

Test cases were adapted from [6].
A sequence of 10 allocations for each of the following $\{64,256,512,1024,2048,4096\}$ byte sizes were called. Elapsed time was calculated using `time.h` C library. Specifically, the use of `setTimer()` [8] and `struct timespec` [9].

2.4.2. Implicit and Explicit Free List, Free Red-Black Tree, and `malloc()` *allocators*

We describe the test suite developed and implemented by the authors for the purposes of comparing benchmarks between the three custom memory allocators, and `malloc()`:

1. A BASH terminal `for` loop executed calls to each allocator with an integer command line argument (within the range of: 10-10000) which specified how many allocation calls are to be executed.

2. Each such call to the allocator was a request for a random size between 1 - 1024 bytes.

3. All pointers that were returned from the allocation request were stored in an address array.

4. Once all allocations were completed, another `for` loop randomly freed half of the addresses in the array.

Elapsed time was calculated in the exact manner as in Section 2.4.1.

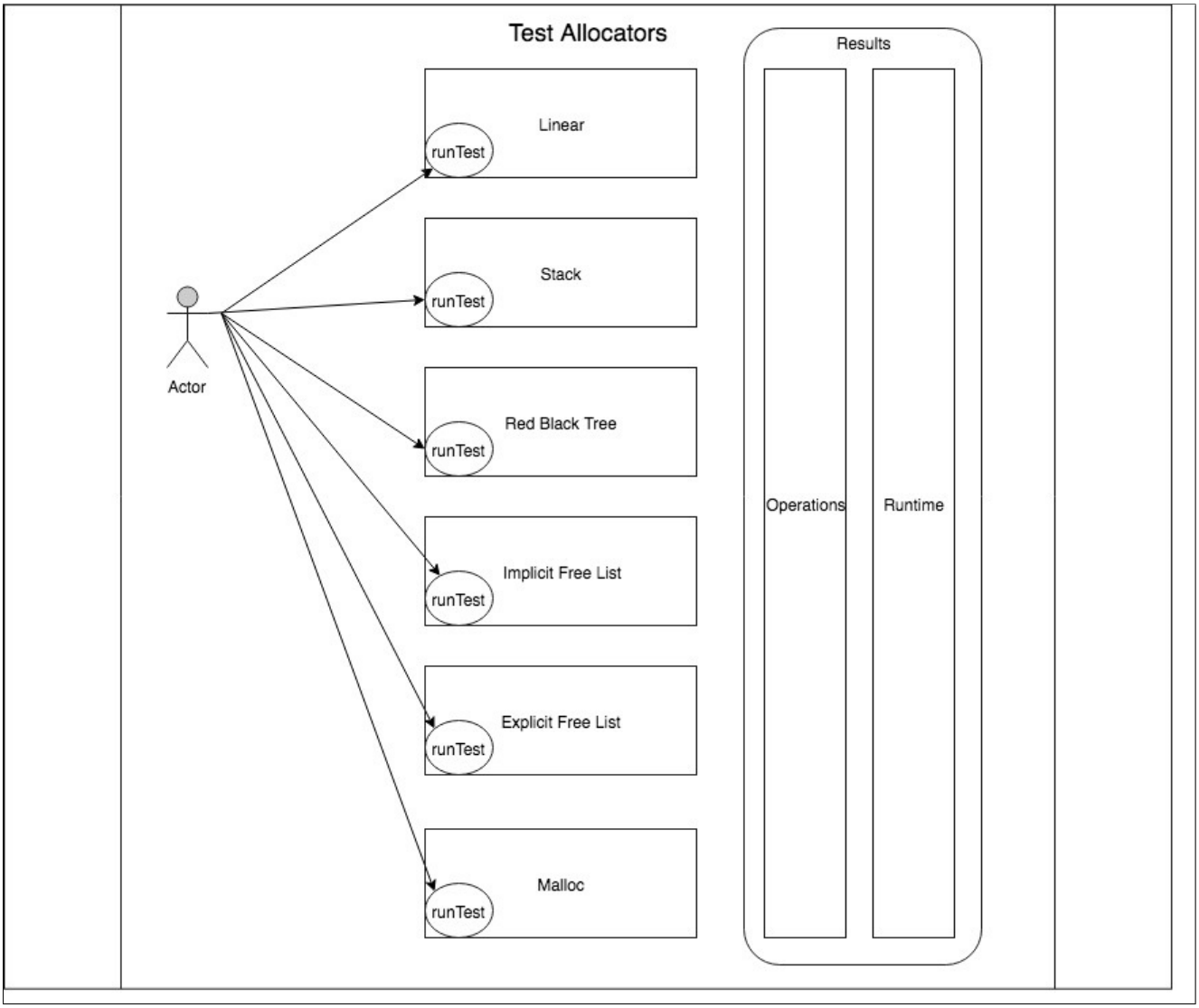

Figure 15: **Testing Platform Use Case Diagram**

A use case diagram representing a user's interaction with our testing platform. `runTest` *performs memory allocations of various sizes and records each allocators' performance as measured by the number of operations and elapsed time (ms)*

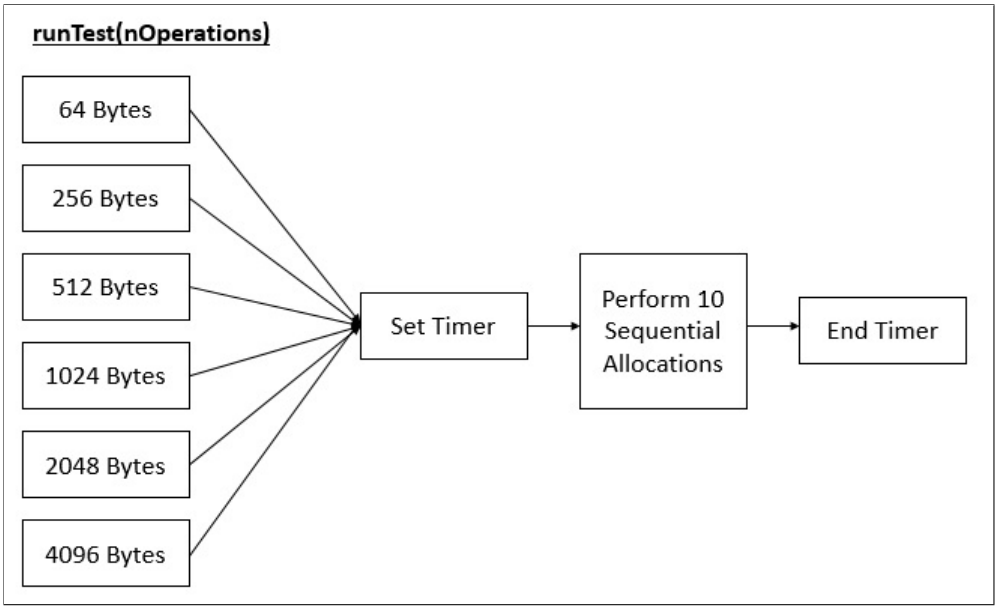

Figure 16: `runTest` **Algorithm**

A flow-chart detailing the testing algorithm used to assess the performance of the Linear and Stack allocators

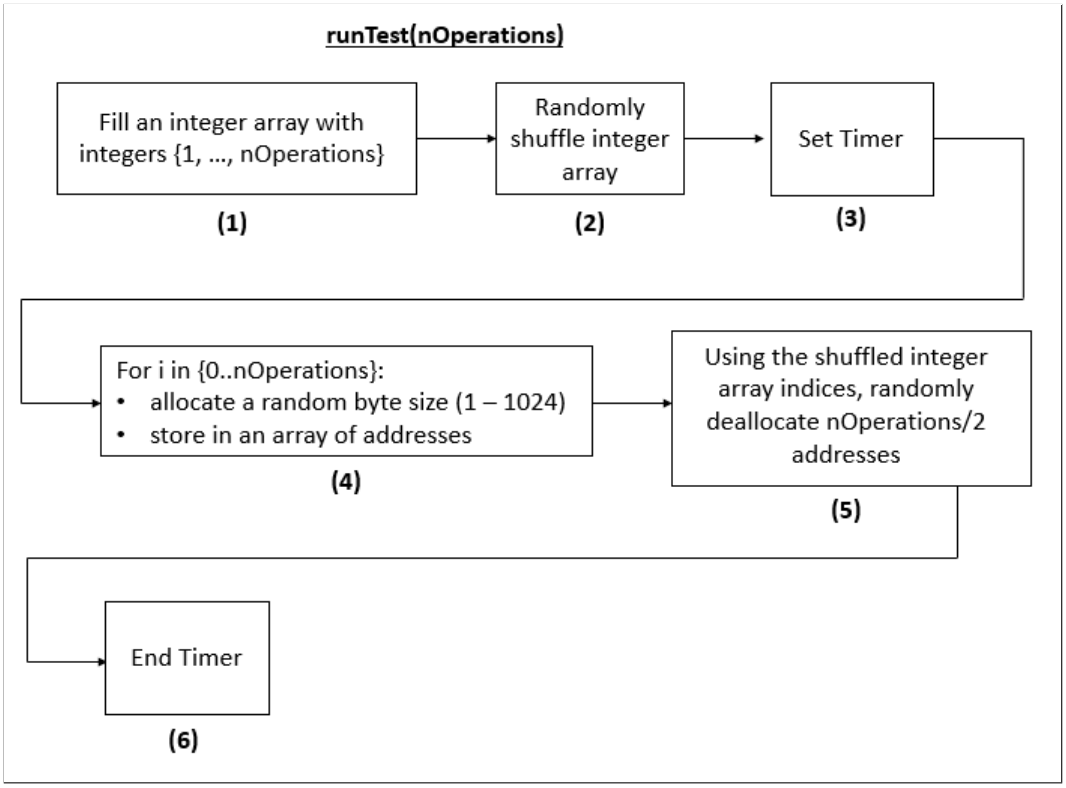

Figure 17: `runTest` **Algorithm**

A flow-chart detailing the testing algorithm used to assess the performance of `malloc()`*, Implicit/Explicit Free List, and Free Red-Black Tree allocators.*

3. RESULTS

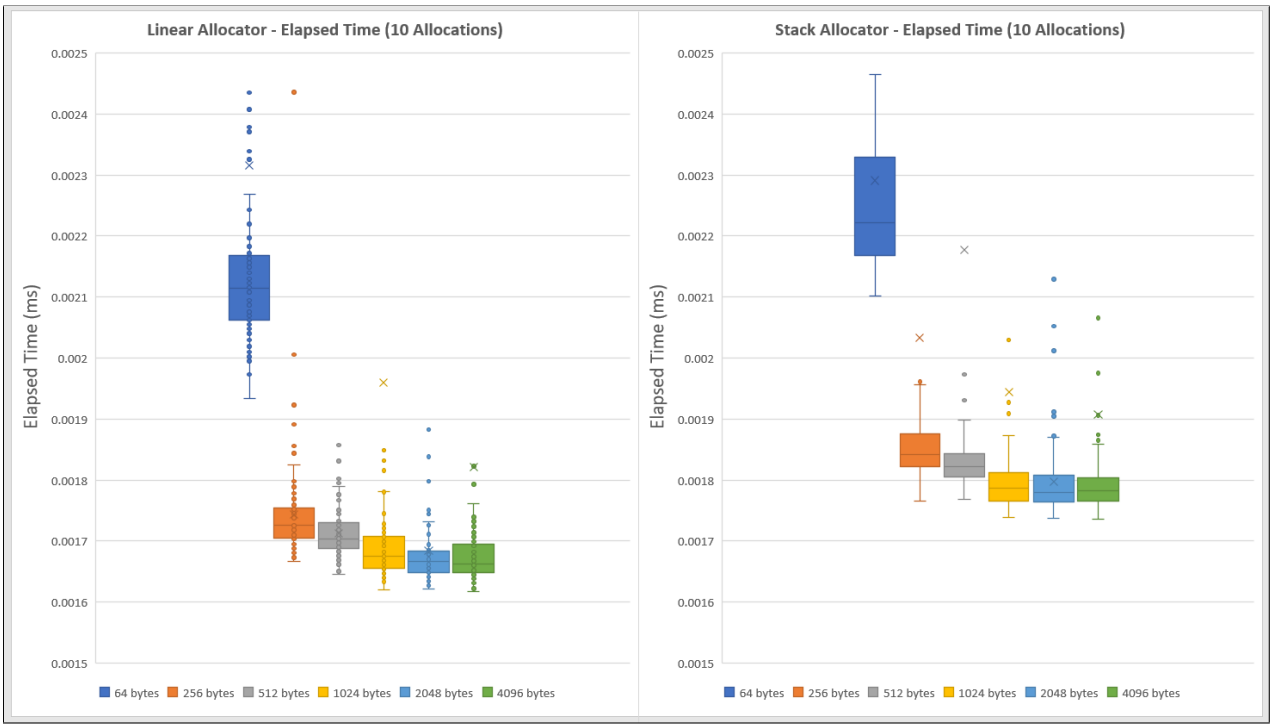

Figure 18: **Linear and Stack allocator Benchmarks**

As expected, the Linear allocator consistently performed faster than the Stack allocator on benchmarks in all size parameters (64 - 4096 bytes). However, it is worth reiterating that the Linear allocator's efficiency stems from its minimal implementation - it does not offer any deallocation functionality.

The first allocation (64 bytes) in our series of tests took significantly longer to complete. This is due to the allocation algorithm of both the Linear and Stack allocators. That is, the first call to the allocators requires that a location on the heap be established as the starting point for all subsequent block allocation. Once this location is established, all following allocations need only refer to this existing reference on the heap, and increment/decrement from it.

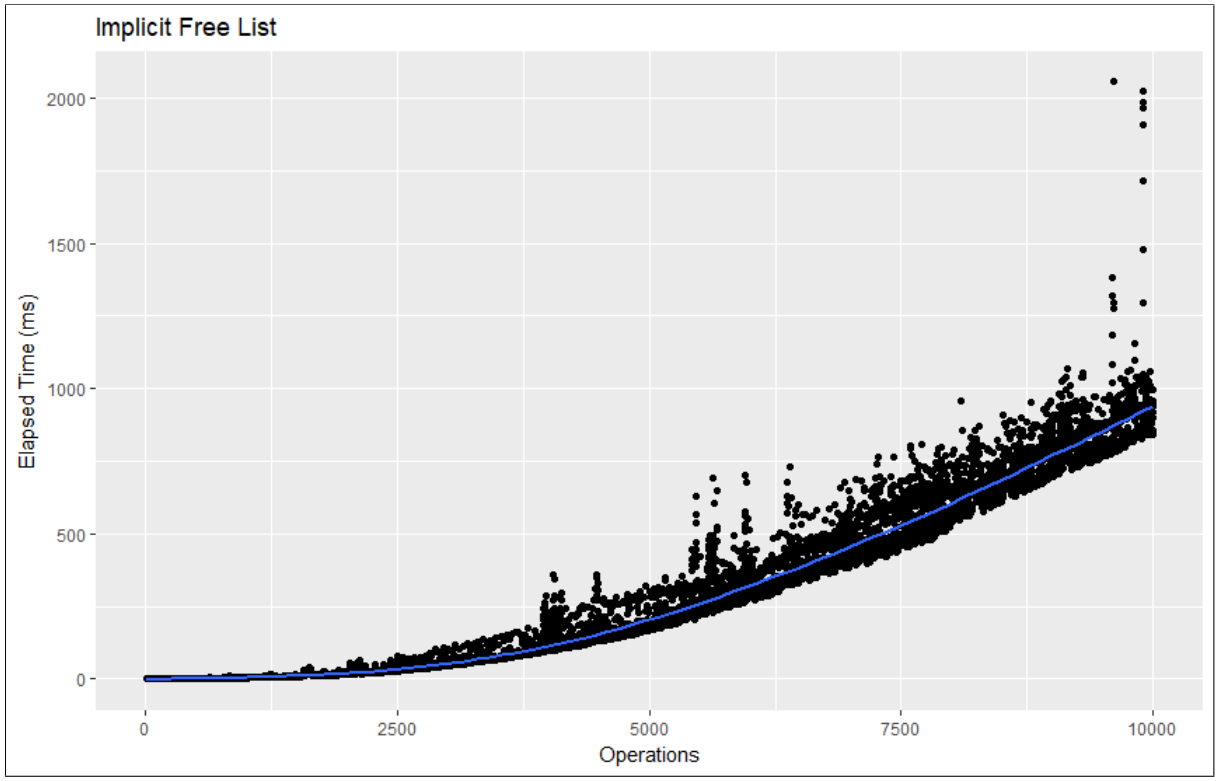

Figure 19: **Implicit Free List Benchmarks**

The Implicit Free List allocator was the slowest allocator of the three custom memory allocators. We attribute this performance defecit to the fact that this allocator maintains an implicit doubly linked list which connects all segments (both free and allocated) on the heap. Therefore, when a call to allocate was made to the Implicit Free List allocator, the algorithm would traverse **all** *nodes in the heap to perform its first-fit search.*

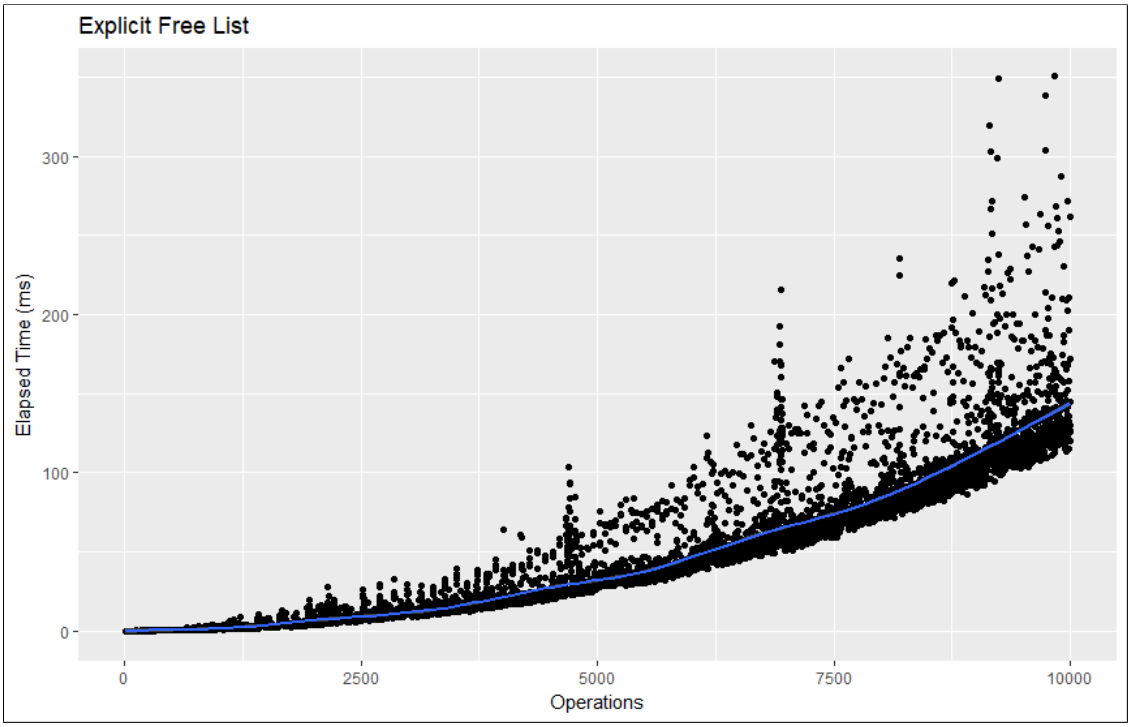

Figure 20: **Explicit Free List Benchmarks**

Predictably, the explicit linkages between the free nodes on the heap used by the Explicit Free List allocator allowed it to perform significantly faster than the Implicit Free List allocator. That is, when searching the heap for a free segment, an Explicit Free List allocator need only traverse the doubly-linked list connecting all free segments of the heap, which drastically reduces its search times for both: finding an address for an allocation and storing the freed address in the explicit linked list following a call to `free()`.

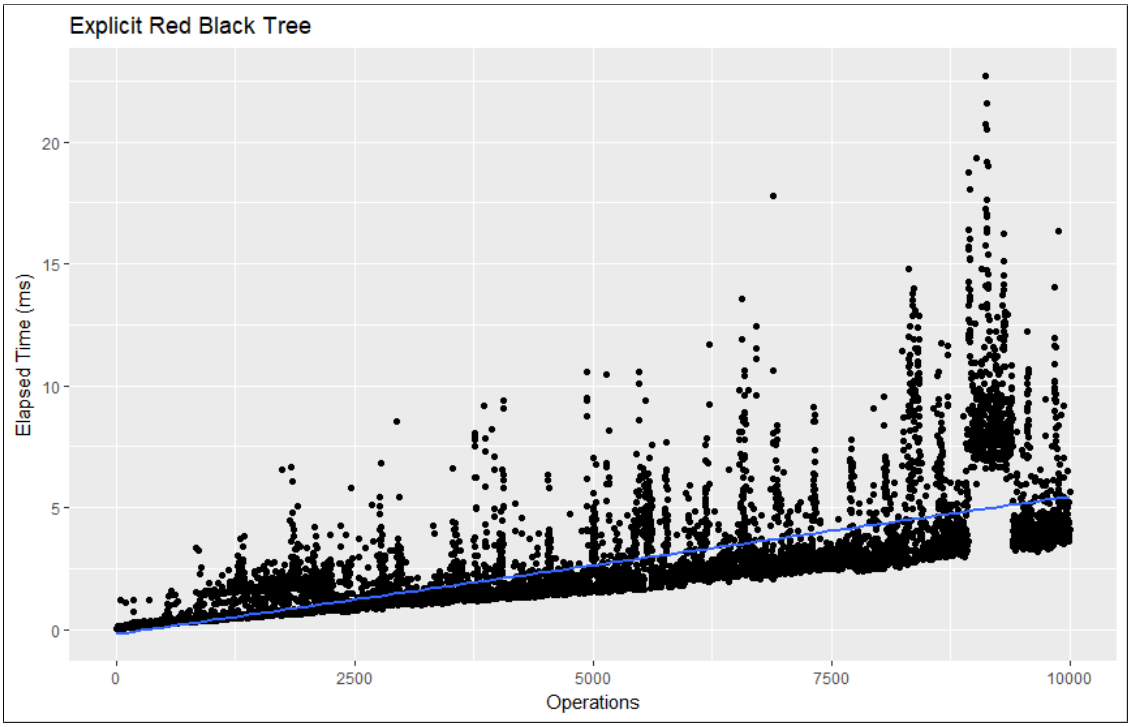

Figure 21: **Explicit Red-Black Tree Benchmarks**

Interestingly, the Free Red-Black Tree allocator and `malloc()` *displayed a linear trend in the growth of elapsed time to perform with respect to the number of operations performed. To compare, the Implicit and Explicit Free List allocators both exhibited a parabolic growth of elapsed time.*

In addition, an anomaly was noted in the time measurements of tests in the range of $\approx 9000 - 9500$ *operations. The authors attribute these outlying entries to an unsterile testing environment. Specifically, since all tests were performed on one of the author's personal computer and desktop environment, it is possible that personal applications interacted with the heap during data collection and affected the accuracy of the final benchmarks.*

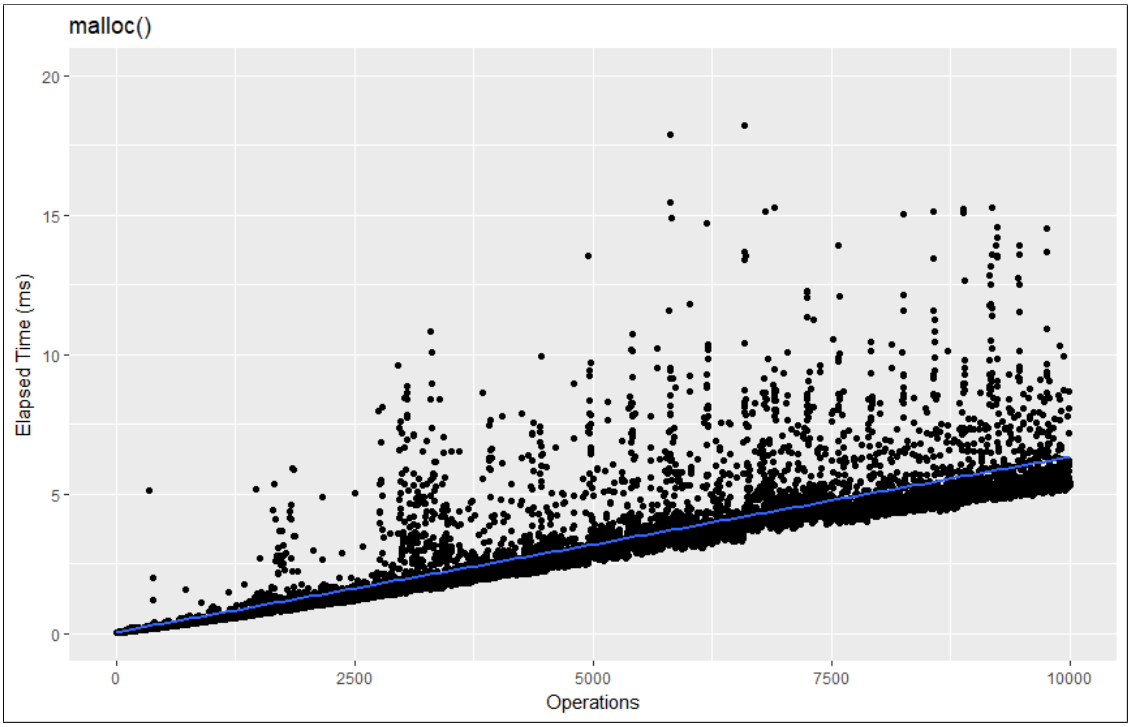

Figure 22: `malloc()` Benchmarks

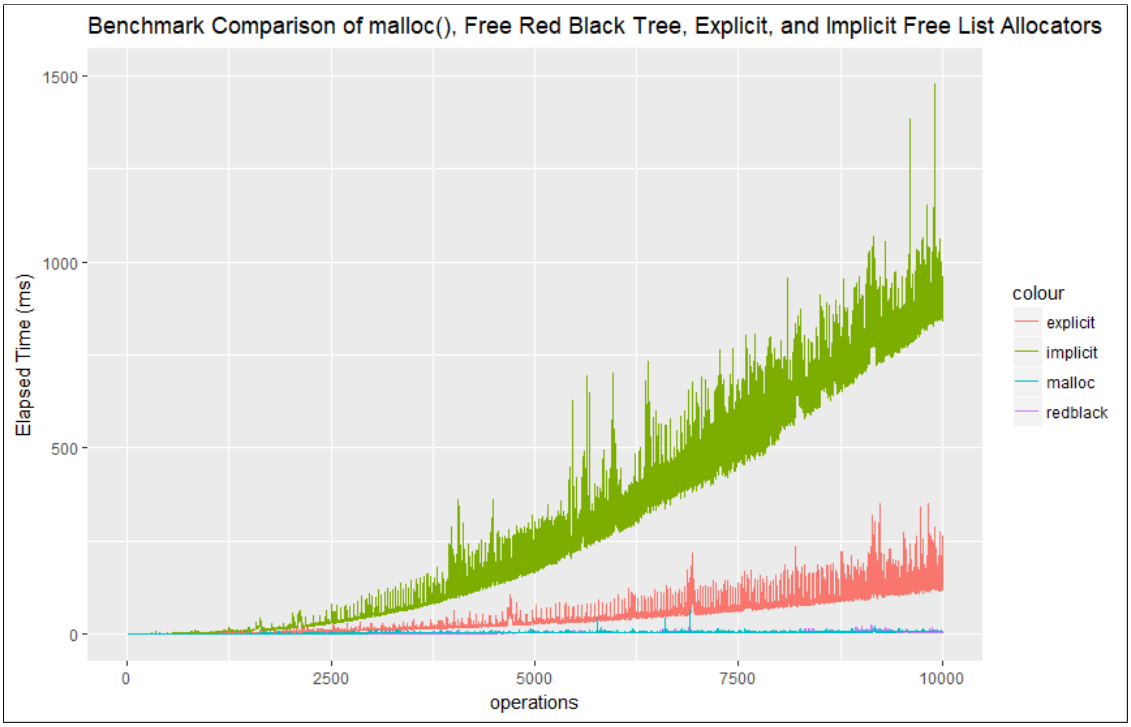

Figure 23: **Benchmark Comparison of** `malloc()`, `Free Red-Black Tree`, `Explicit`, and `Implicit Free List allocators`

A comparison of all custom allocators and `malloc()` *articulates the results mentioned in Figures 19-21.* `malloc()` *and the Free Red-Black Tree allocator completed the testing benchmarks in very similar runtimes, when compared to the slower Explicit Free List allocator, and the much slower Implicit Free List allocator.*

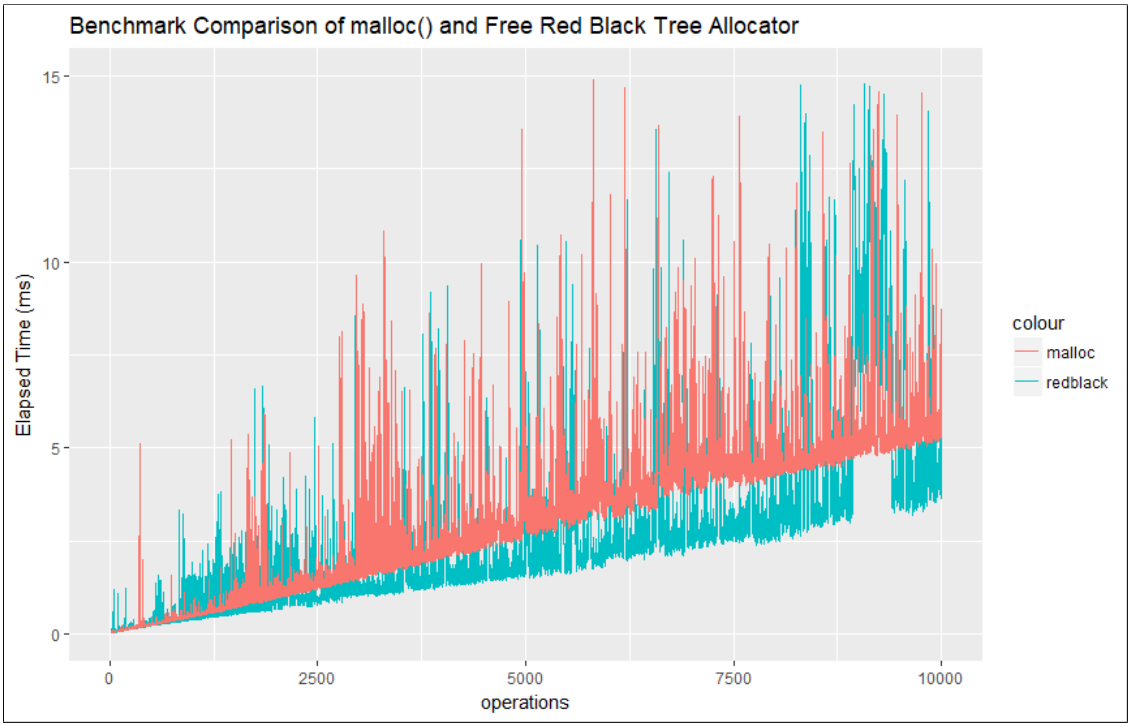

Figure 24: **Benchmark Comparison of** `malloc(), Free Red-Black Tree allocator`

Surprisingly, the Free Red-Black Tree allocator performed consistently faster than `malloc()` *on all Elapsed Time Benchmarks. Furthermore, as the number of operations increased, the gap between the Elapsed Time Benchmarks of* `malloc()` *and the Free Red-Black Tree allocator grew, indicating that the Free Red-Black Tree allocator handled high-volume requests of allocation optimally.*

4. EVALUATION AND QUALITY ASSURANCE

After learning about the algorithms and data structures that made up `malloc()`, we set out to compare the efficiency and runtime of `malloc()` to other custom-built memory allocators. Extensive testing produced clear evidence that the speed and efficiency of `malloc()` was underestimated. A greater variety of testing formats would have produced a more complete set of results. For example, our tests could have included:

- unusually large allocation requests
- allocation/deallocation request patterns which are optimally handled by custom allocator algorithms

Such benchmarks would have lended to the creation of a robust testing suite and a more definitive bottom line regarding the efficacy of all custom allocators described, as well as `malloc`. `malloc`'s performance can be attributed to a structural segmentation of the heap. `malloc` stores its free segments in an array in which each index (or bin) contains an explicit free list of allocated memory addresses. Traditionally, `malloc` uses 128 bins that are sequentially ordered by size. We have shown that this implementation of a memory allocator produces minimal heap-search times.

The tests performed did not account for an analysis of space complexity. The efficacy of the custom allocators we described is dependant on the use of the searching, insertion, and deletion algorithms which are supported by, in some cases, complex data structues (i.e. Red-Black Trees, Arrays of Explicit Free Lists). These data structures require an additional implementational level of header structures (such as nodes, a head, tail, root, or leaves) which require additional space. Including benchmarks which analyzed the space complexity of the custom allocators in this report would have produced a thorough analysis of memory allocation algorithms.

Finally, we will note that even though the Free Red-Black Tree allocator was shown to exhibit faster running times when compared to `malloc()`, the use of the Free Red-Black Tree allocator will likely be restricted to implementations that are *extremely* time-sensitive. Therefore, we believe that `malloc()`'s widespread use stems from its thorough documentation, availability, and, as was shown in this report, excellent time efficiency and speed.

5. CONCLUSION

5.1. Summary

We set out to compare different custom memory allocators to `malloc()`. Our tests revealed that:

- The only allocator that had a lower runtime than `malloc()` was the Free Red-Black Tree allocator.
- Both `malloc()` and the Free Red-Black Tree allocator appear to have linear runtimes (with respect to the number of operations).
- The Free Red-Black Tree allocator demonstrated a lower growth rate coefficient (approximately 25% faster than `malloc()` as the number of allocations increased).

- The Explicit Free List allocator and the Implicit Free List allocator both demonstrated considerably slower runtimes which were best modeled by parabolic growth.
- The Stack allocator ran in linear runtime and proved useful in specific cases where memory allocations are all done sequentially. (This is due to the nature of a stack which dictates that memory must be freed in a Last-in, First-Out (LIFO) fashion.)

5.2. Relevance

`malloc()` was shown to be a time efficient, multi-purpose memory allocation function with only moderate space overhead for the storing of its backing data structure. However, the Red-Black Tree backed allocator proved to be noticeably more efficient. This applies to the COMP 3000 - Operating Systems curriculum as it shows that relying on default implementations of memory allocation algorithms is safe for time sensitive low-level/system-programming applications that require dynamic memory.

In addition, we claim (based on the reported benchmarks) that the Free Red-Black Tree allocator is more suited for system-programming applications in which time complexity is a concern.

Finally, for certain cases (such as very little room for overhead space or consistent patterns of sequentially called allocations), a Stack allocator may also provide a viable solution for systems programmers. We provide a concise review of use cases and time complexity analyses of the custom allocators discussed in this report.

5.3. Synopsis

5.3.1. Linear allocator

Use Cases:

- Programs which follow an allocation pattern that does not require memory deallocation at the pointer level
- Reuse of addresses only after a total reset of all allocated memory upon `free()`

Allocation/Deallocation Time Complexity : $O(1)$

5.3.2. Stack allocator

Use Cases:

- Allocation patterns which require a sequence of allocations to be made, which are then expected to be deallocated in reverse order. [12]
- Certain allocations which stay active for a certain lifespan, that require additional subseqeuent allocations. [12]

Allocation/Deallocation Time Complexity : $O(1)$

5.3.3. Pool allocator

Use Cases:

- Instances where objects of a constant size need to be created and destroyed dynamically, an example being bullets for weapons in video games. The pool allocator also guards against heap fragmentation, and in the case of medical image processing where images remain a fixed size, the ability to process the images loaded in memory without fragmentation is essential to efficient image processing. [14]

Allocation/Deallocation Time Complexity : $O(1)$

5.3.4. (Implicit/Explicit) Free List allocator

Use Cases:

- Intended for use as a reserve allocation strategy in low memory situations, when no memory is available for the data structures which are required by other, more complex allocators (e.g. `malloc()`, Free Red-Black Tree allocator). [13]

Allocation/Deallocation Time Complexity : $O(1+\min\{i, n-i\})$

5.3.5. Free Red-Black Tree allocator

Use Cases:

- Faster, more efficient allocation runtime (as compared to default `malloc()` renders this allocator as an optimal choice in allocation patterns which are highly constrained in terms of elapsed time per operation requirements.

Allocation/Deallocation Time Complexity : $O(\log(n))$

5.4. Future Work

A future investigation into optimizing the space and time complexity of a Free Red-Black Tree allocator is a suggestion for future research. One possible way to accomplish this would be to apply a binning system which would contain a range of frequently used allocation sizes (known in advance).

Finally, it is well known that Skiplists [16] achieve the same efficient search runtime as Red-Black Trees, while being much easier to implement [1]. Therefore, the authors believe that future research should focus on using Skiplists as the backing data structure for heap traversal. Specifically, an implementation of a binning system similar to `malloc()`'s, but with Skiplists instead of explicit free lists. Submitting such an allocator to benchmark testing similar to the tests described in this report would yield a highly informative comparison to `malloc()`. During our literature review, we have encountered similar technical algorithms being described for use in the latest massively parallel dynamic memory allocation algorithms for GPU's [15].

Since Dynamic Memory Allocation is an integral part of all processing from the kernel to the user level, we hope that this report has provided the reader with an analytical framework with which to critically assess the computational needs of programs in development, and be able to choose, or implement, a suitable memory allocator to meet those needs in the most optimal manner.

6. REFERENCES

[1] Morin, Pat. Open Data Structures: An Introduction. Vol. 2. Athabasca University Press, 2013. Retrieved from http://opendatastructures.org/

[2] Doug Lea. (2000, April 4). A Memory Allocator. Retrieved from http://gee.cs.oswego.edu/dl/html/malloc.html

[3] Virginia Tech University, Online CS Modules: Memory Allocation. Retrieved from https://courses.cs.vt.edu/csonline/OS/Lessons/

[4] Wilson, P. R., Johnstone, M. S., Neely, M., & Boles, D. (1995). Dynamic storage allocation: A survey and critical review. In Memory Management (pp. 1-116). Springer, Berlin, Heidelberg.

[5] CSE351 - Inaugural Edition - Spring 2010. Paul G. Allen School of Computer Science and Engineering. Memory Allocation Online Lecture. Retrieved from https://courses.cs.washington.edu/courses/cse351/

[6] mtrebi. (2017, April 4). Memory Allocators. GitHub repository, Retrieved from https://github.com/mtrebi/memory-allocators

[7] Zachary Stroud-Taylor. (2018, April 12). Memory_Allocators. Github repository, Retrieved from https://github.com/ZacASTaylor/Memory_Allocators

[8] setitimer(2) - Linux man page. Retrieved from https://linux.die.net/man/2/setitimer

[9] The Single UNIX ®Specification, Version 2. Copyright ©1997 The Open Group. Retrieved from pubs.opengroup.org/onlinepubs/7908799/xsh/time.h

[10] Max Gillett. (2013, June 24). Coursera - Hardware Software Interface. Github repository, Retrieved from https://github.com/maxgillett/coursera/tree/master/Hardware%20Software%20Interface/course-materials/lab5

[11] Sebastien Chapuis. (2015, February 21). Malloc / Free / Realloc / Calloc implementation using a red-black tree. Github repository, Retrieved from https://github.com/sebastiencs/malloc

[12] Molecular Musings. (2012, August 27). Memory allocation strategies: a stack-like (LIFO) allocator. Retrieved from https://blog.molecular-matters.com/2012/08/27/

[13] Memory Pool System 1.117.0 Documentation. (2018). Free List Allocator. Retrieved from https://www.ravenbrook.com/project/mps/master/manual

[14] EASTL – Electronic Arts Standard Template Library. (2007, April 27). Appendix 26 - Memory allocators used in game software. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html#Appendix_26

[15] Steinberger, M., Kenzel, M., Kainz, B., & Schmalstieg, D. (2012, May). Scatteralloc: Massively parallel dynamic memory allocation for the GPU. In Innovative Parallel Computing (InPar), 2012 (pp. 1-10). IEEE.

[16] Carl Kingsford. School of Computer Science, Carnegie Mellon University. Skip Lists - CMSC 420. Retrieved from https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf

7. CONTRIBUTIONS

7.1. Samuel Diamantstein

- Report Writing: Introduction, Background Information: Linear/Pool/Stack Sections/Figures, Evaluation and Quality Assurance, Synopsis
- Use case diagram
- Report Revision
- Linear Allocation Implementation and testing (unsuccessful)

7.2. Zachary Stroud - Taylor

- Testing
- Graphing
- Report Writing: Conclusion
- Figure descriptions
- Github Repository
- Stack Allocator Implementation
- Process memory extension (unsuccessful)
- Python scripting for graphing (deprecated)

7.3. Alex Trostanovsky

- Development and Implementation of Testing Platform
- Implementation of Custom Allocators within Testing Platform
- Segregated Fits, Pool Allocator Implementation
- Adaptation of Benchmark Testers from [6]
- Figure descriptions
- Data manipulation
- Graph development, and Graphing
- Report Writing: Introduction, Background Information, Results, Evaluation and Quality Assurance, Conclusion.
- Figures: 1-14, 16-23.
- Report editing, revision, citations.
- LaTeX document creation.