An Efficient Priority Queue Data Structure for Big Data Applications

James Rhodes* and Elise de Doncker

***Abstract*—We have designed and developed an efficient priority queue data structure that utilizes buckets into which data elements are inserted and from which data elements are deleted. The data structure leverages hashing to determine the appropriate bucket to place a data element based on the data element's key value. This allows the data structure to access data elements that are in the queue with an O(1) time complexity. Heaps access data elements that are in the queue with an O(log n) time complexity, where n is the number of nodes on the heap. Thus, the data structure improves the performance of applications that utilize a min/max heap. Targeted areas include big data applications, data science, artificial intelligence, and parallel processing. In this paper, we present results several applications. We demonstrate that the data structure when used to replace a min/max heap improves the performance applications by reducing the execution time. The performance improvement increases as the number of data elements placed in the queue increases. Also, in addition to being designed as a double-ended priority queue (DEPQ), the data structure can be configured to be a queue (FIFO), a stack (LIFO), and a set (which doesn't allow duplicates).**

***Index Terms*—Priority queue, Buckets data structure, big data, heap, performance**

I. Introduction

Many have abandoned the idea of hashing despite the fact that it affords an O(1) time complexity for accessing data elements. This is because of the possible collisions that can occur due to not having a unique slot for every key. However, many applications require only a limited and manageable number of keys. For example, using our proposed data structure, the 15-puzzle application for a given initial state required only 30 unique keys and the goal state was found in 1,056,405 less iterations than when a heap was used. Furthermore, many applications do not require every data element to have a unique key. A range of data elements can be grouped together with one key and experience only a slight amount of accuracy loss if any. For instance, using our data structure for a sample integral computed with adaptive multivariate integration, only 85 slots were generated when ranges of numbers were coalesced. The results were identical to using unique values out to 10 decimal places.

II. Heap Data Structure

Although priority queues can be implemented with arrays and linked lists, heaps are the most common data structure used to implement priority queues. Heap operations include insert, delete-min (delete-max), and find-min (find-max). Min heaps use the minimum key operations and max heaps use the maximum key operations. The structure of a heap is tree-based. There is a variety of types of heaps that have different operations and complexities.

A binary heap is based on a binary tree and is known as the standard heap. The heap property ensures that the keys of any parent node's children are not less than or greater than its key for a min heap or max heap, respectively [1]. In order to preserve the heap property, significant maintenance is required for both inserting and deleting data. We call this process "heapify". The heapify process is more costly from a time perspective when deleting from a heap than it is when inserting into a heap.

The insert and delete operations of a heap both have an O(logn) time complexity. This is adequate for most applications. However, some heap-intensive applications could experience performance degradation when the number of data elements on the heap is enormous. The priority queue data structure that we designed lowers the time complexity and produces comparable accuracy for these types of applications. By getting rid of the data manipulation required for heaps, our data structure improves the performance of applications by reducing the amount of CPU time required for applications to complete execution.

III. Buckets Data Structure

The priority queue data structure that we designed and developed is bucket-based. We investigated the idea of using buckets as the solution to eliminate the heapify process. We associate buckets with nodes on a heap and note a difference between a data element and a heap element. A data element is an individual node that is in a linked list. A heap element is the head node of a linked list and corresponds to a heap node that can be inserted or deleted. The nodes on a heap are buckets of nodes that have the same or similar key.

A bucket, which is a linked list, may have many nodes in it. However, the heap will only have the head node of the linked list on it. In particular, the nodes on the heap will be pointers to head nodes of linked lists. Although there could be millions of data elements on a heap, only a fraction of the nodes would have to be rearranged during the heapify process. When they are moved all the nodes in a bucket will be treated as one node.

The buckets used to store data elements are implemented as an array of pointers to the heads of linked lists. Since it is dynamic the array can grow and shrink. When a bucket no longer has any nodes because of the last node being deleted, the next non-null bucket with the minimum or maximum

Manuscript received October 20, 2022; revised December 7, 2022; accepted March 29, 2023.

The authors are with Western Michigan University, Kalamazoo, MI 49008 USA.

*Correspondence: james.rhodes@wmich.edu (J.R.)

value is selected. When a data element is inserted into a bucket that doesn't have any nodes, no data manipulation is required.

Buckets not only lower the time spent to heapify but also lower the frequency of calling the heapify process. When a data element is deleted from a bucket and there is still at least one node in it, a call to the heapify process is avoided. Furthermore, when a data element is inserted into a bucket that has at least one node in it, a call to the heapify process is avoided. A hashing function is used to determine the appropriate buckets with an O(1) time complexity. This allowed us to develop a priority queue data structure that alleviated the heapify process and its associated data manipulation.

IV. Background

We began our research with the objective to improve the performance of a multivariate integration application [2–9]. The application utilizes a heap and is driven by a parallel algorithm [10–15]. It implements a Compute Unified Device Architecture (CUDA) kernel to evaluate regions. The application deletes a node that represents the region with the largest error from the heap, divides it into 2, 4, 8, or 16 subregions, evaluates the subregions (computes their integral contribution and estimated error) and inserts them into the heap. The application continues until the total error is reduced below a user-prescribed level or a prescribed number of function evaluations have been performed. We identified the heap as a bottleneck when a huge number of data elements were inserted into the queue.

Originally, our goal was to improve the performance of the heap. We focused on reducing the number of calls to the heapify process. We identified that if ranges of data could be grouped together that would both decrease the number of nodes to be moved during the heapify process and decrease the number of calls to the heapify process.

We decided to use the error estimate of the regions as the key for a hashing function. The error estimate is declared as a double datatype which could not be used directly as an index into an array, so we used the exponent and mantissa of the number as indexes into a two-dimensional array. Our implementation is based on the IEEE Standard for Floating-Point Arithmetic (IEEE 754) representation and a computer program at [16]. After a several improvements of the data structure, we were ultimately able to eliminate the heap with its tree structure and recursive heapify process entirely. The data structure is now an array of pointers to heads of doubly linked lists.

V. Approach

We developed two versions of several applications one implemented with a heap and another with the buckets data structure. We executed both versions of the applications and analyzed and compared the results. The applications were executed on a system with the following specifications:

Dual 8C Intel Xeon E5-2670 @2.6GHz host, with 128GB of RAM

Kepler 20(m) GPU, with 2496 CUDA cores, 4.8GB global memory, and 956.8 GFLOPS double precision theoretical performance.

VI. Huffman Codes

Huffman codes constitute an application of binary trees with minimal weighted external path length that obtain an optimal set of codes for messages. The codes are binary strings that are used to transmit the corresponding messages. A decode tree is used at the receiving end to decode the message. The external nodes (leaves) of a decode tree represent messages. The branching that is necessary at each level of the decode tree to reach the correct external node is determined by the bits in the codeword for a message. A zero is interpreted as a left branch and a one is interpreted as a right branch. The bit strings from the root to an external node are called Huffman codes. The cost of decoding is proportional to the number of bits in the code, which is equal to the distance from the root to the external node. If f_i is the relative frequency in which a message will be transmitted and d_i is the distance from the root node to the external node for a message, then the expected decode time is $\sum_{i=1}^{h} f_i d_i$, where h is the height of the tree [17]. By choosing codewords resulting in a decode tree with minimal weighted external path length, the expected decode time is minimized. The expected decode time is the expected length of a transmitted message. Thus, the code that minimizes expected decode time, minimizes the expected length of a message code, as well.

A program that implements Huffman Codes usually implements a min heap as a priority queue. There are two main steps to the algorithm to generate the codes. The first is to load the priority queue with the data elements and their associated frequencies. The frequencies are used as the keys where the minimum key is given the highest priority and is therefore the next data element to be deleted from the priority queue.

After the data is loaded into the priority queue, the building process starts. This process consists of deleting two data elements with minimum frequencies, creating a new node that will be the parent of the two nodes that were deleted and become the new node's left and right children. The new node is inserted into the queue. This process continues until there is only one node resulting in a binary tree.

The final process of generating the codes by assigns a zero to the left child and a one to the right child of every parent node. However, since we are primarily concerned with the operations of the priority queue, we did not include the code assigning process in our analysis.

We developed a program that uses a min heap as a priority queue and used it to perform the load and build processes. We also developed a program that uses the buckets data structure as a priority queue and used it to perform the load and build processes. Fig. 1 presents the results as a plot with load, build, and total (load + build) times for both the heap program and the buckets program for a range of the number of data elements on the resulting tree. The plot shows that as the number of data elements increases, the efficiency of the

buckets data structure increases.

Fig. 1 shows that for Huffman Codes load times for the heap and the buckets data structure are similar as the two curves on the plot overlap. However, there is a substantial difference in the build times for the two data structures. The build time for the buckets data structure is 0.36 seconds at 1 million data elements. The build time for the heap is 2.61 seconds at 1 million data elements. Thus, the buckets data structure provided an 86.2% build time improvement as compared with the heap. The difference in the total times for the data structures is 2.25 (2.76 heap – 0.51 buckets) seconds which is an 81.5% improvement provided by the buckets data structure.

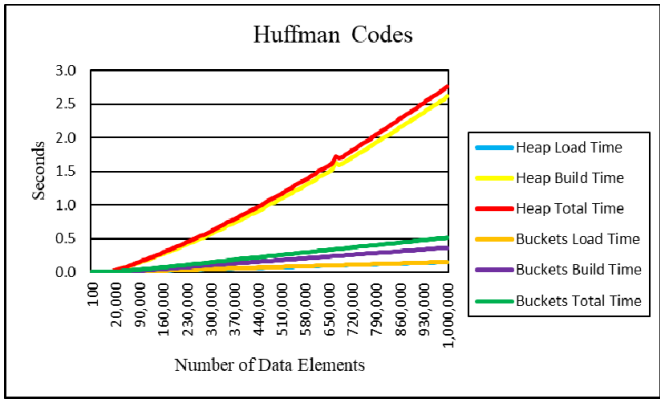

Fig. 1. Operation times for the Huffman Codes application.

In addition to a priority queue operation improvement, there is a total execution time improvement. The total execution time difference is 1 minute and 38 seconds (2 minutes and 3 seconds for the heap – 25 seconds for the buckets data structure), which is a 79.5% improvement achieved by the buckets data structure. The heapify process used by heaps to maintain the heap property causes the program to run almost five times longer than with the buckets data structure. This is because the heapify process is called recursively, and there is a vast amount of overhead associated with this recursion. Thus, the buckets data structure provides a decrease in CPU time over heaps due to both a priority queue operation improvement and a decrease in overall execution time. The execution improvement occurred with all the application instances we evaluated and analyzed. The resulting trees that were generated by both data structures were identical for every test case for the various numbers of data elements. Therefore, the buckets data structure did not cause any loss of accuracy.

VII. The Transportation Problem

The importance of transportation has a great role to play in society. The profits of companies from moving merchandise from one location on the globe to another are determined by transportation. Transportation costs and time can rival production costs and production time. Transportation theory is the study of optimal transportation and allocation of resources.

One type of transportation problem that is associated with Linear Programming (LP) deals with the physical transportation of products from sources to destinations, the objective being to minimize the cost of transportation.

The transportation problem consists of the following pieces of information [18]:

1) m = the number of sources
2) n = the number of destinations
3) The total quantity available at each source
4) The total quantity required at each destination
5) The cost of transportation of one unit from each source to each destination

In our analysis, m = n. Thus, the number of sources equals the number of destinations. The transportation problem has the following assumptions:

1) The total quantity available at all the sources is equal to the total quantity required at all the destinations. If they are not equal, an extra source or destination is added.
2) The unit transportation cost from a source to a destination is known.
3) The unit cost is independent of the number of products transported.
4) The objective is to minimize the transportation cost.

The transportation problem is formulated in matrix form. The rows are sources and the columns are destinations the supply amounts will be added as an extra separate column and the demand amounts will be added as an extra separate row. The cells of the matrix will have the cost of transporting one unit from and to the corresponding source and destination, respectively. There are three known initial basic feasible methods for solving the transportation problem:

1) North-West (N-W) Corner Rule
2) Least Cost Method
3) Vogel's Approximation Method

The North-West (N-W) Corner Rule starts with the north-west corner of the matrix and transports as much of the demand of the corresponding destination as the corresponding source can supply. Either the corresponding destination's demand will be met and the column will be eliminated, or the corresponding source's supplies will be exhausted and the row will be eliminated, or both. This process of choosing the north-west corner of the resulting matrix continues until all destinations' demands have been met and all sources' supplies have been exhausted.

The Least Cost Method starts with the cell that has the minimum cost and transports as much of the demand of the corresponding destination as the corresponding source can supply. If there is more than one cell with the minimum cost, any of the cells can be selected. Either the corresponding destination's demand will be met and the column will be eliminated, or the corresponding source's supplies will be exhausted and the row will be eliminated, or both. This process of choosing the cell that has the minimum cost of the resulting matrix continues until all destinations' demands have been met and all sources' supplies have been exhausted.

Vogel's Approximation Method is more complex than the above two methods but often provides better results. It

consists of first finding the difference between the two lowest cost values in each row and column. These differences are called penalties. The maximum value among all the penalties of all rows and columns is determined. If there is more than one row or column with the maximum penalty, any one of them can be selected. Find the cell that has the minimum cost in the row or column that was selected with the maximum penalty. If there is more than one cell with the minimum cost, any of the cells can be selected. Transport as much of the demand of the corresponding destination as the corresponding source can supply. Either the corresponding destination's demand will be met and the column will be eliminated, or the corresponding source's supplies will be exhausted and the row will be eliminated, or both. This process of finding the maximum penalty and choosing the cell that has the minimum cost of the resulting matrix continues until all destinations' demands have been met and all sources' supplies have been exhausted.

We propose a fourth initial feasible method to solve the transportation problem. It uses a greedy algorithm and two priority queues. This method consists of inserting all the sources with their associated costs into a minimum priority queue where cost is the priority. Delete a node from the minimum priority queue, which will provide one of the sources that have the lowest cost available. Use the source to find all the destinations to which the source transports at the minimum cost. Insert those destinations with their associated demands into a maximum priority queue where demand is the priority. Delete a node from the maximum priority queue, which will provide one of the destinations that have the highest demand available. Transport as much of the demand of the destination as the source can supply. Either the destination's demand will be met and it will be eliminated, or the source's supplies will be exhausted and it will be eliminated, or both. This process of finding the source with the minimum cost and the destination with the maximum demand continues until all destinations' demands have been met and all sources' supplies have been exhausted.

While an initial feasible solution may satisfy the requirements of all the sources and destinations, it may not provide the least cost possible. The solution that minimizes the transportation cost is the optimal solution. After obtaining the results from an initial feasible method, the solution must be verified with an optimality test. It will determine whether an initial feasible solution is the best solution and, if not, it will provide the optimal solution.

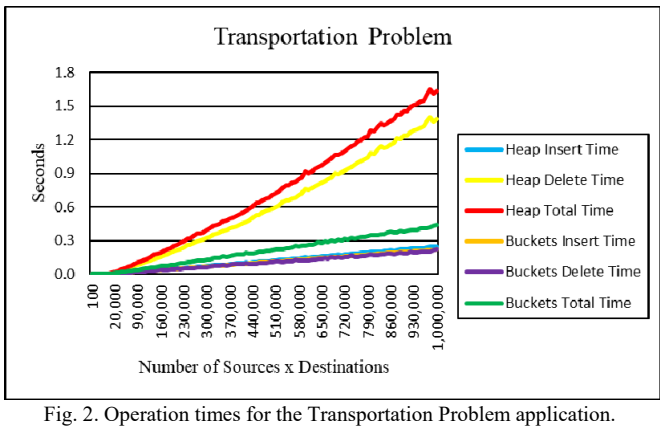

Fig. 2. Operation times for the Transportation Problem application.

We developed a program that uses a min heap and a max heap as priority queues and used them to perform the initial feasible solution that we proposed. We also developed a program that uses our buckets data structure as minimum and maximum priority queues and used them to perform the initial feasible solution that we proposed. Fig. 2 presents the results in the form of a plot with insert, delete, and total (insert + delete) times for both the heap program and the buckets program for a range of the number of sources and destinations. The plot shows that as the number of sources and destinations increases, the efficiency of the buckets data structure increases.

Fig. 2 shows that, for the Transportation Problem, insert times for the heap and insert and delete times for the buckets data structure are similar as the three lines on the plot overlap. However, the delete times for the heap are much longer than the other times. The delete time for the buckets data structure is 0.22 seconds at 1 million data elements. The delete time for the heap is over 1.39 seconds at 1 million data elements. Thus, the buckets data structure provided an 84.2% delete time improvement as compared with the heap. The difference in the total times for the data structures is 1.20 (1.64 heap – 0.44 buckets) seconds, which is a 73.2% improvement provided by the buckets data structure.

In addition to a priority queue operation improvement, there is a decrease in total execution time. The total execution time difference is 1 minutes and 11 seconds (2 minutes and 15 seconds for the heap – 1 minute and 4 seconds for the buckets data structure), which is a 52.4% improvement achieved by the buckets data structure. The heapify process to maintain the heap property causes the program to run more than twice as long as it does with the buckets data structure. The resulting minimized transportation costs that were generated by both data structures were identical for every test case with the various numbers of data elements. Therefore, the buckets data structure did not cause any loss of accuracy.

VIII. A* Search Algorithm

The A* Search algorithm is an extension of Dijkstra's algorithm and is useful for finding the lowest cost path between two vertices in a graph. The A* Search algorithm is a modification of Dijkstra's algorithm that is optimized for a single destination. It is a combination of Dijkstra's and best first search algorithms. It uses a cost function $f(n) = g(n) + h(n)$ where $g(n)$ is the cost of the path from the origin to vertex n and $h(n)$ is a heuristic function that is the cost of the path from n to the destination.

The A* Search algorithm is like the 15-puzzle and the REL relocation problem [19] since it has a cost $f(n) = g(n) + h(n)$ and 0 or 1 for blocked and unblocked cells, respectively, in a matrix that is used to represent graphs. The cost function [17] for the 15-puzzle application is $c(x) = f(x) + g(x)$ where $f(x)$ is the length of the path from the root of the state space tree to node x and $g(x)$ is a heuristic function representing the number of tiles that are not in the correct position in the matrix for the state of node x. The 0 and 1 in the A* Search algorithm determines the passage from one cell to an adjacent cell in the matrix. The 0 and 2 in REL determines the capacity from one vertex to an adjacent vertex in the graph.

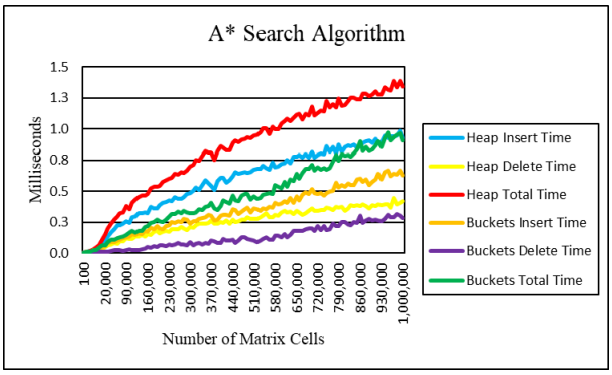

Fig. 3. Operation times for the A* Search algorithm application.

The A* Search algorithm uses a priority queue, which is a set – preventing duplicate values from being inserted into it. It also uses a stack to display the path in the correct order from origin to destination since the path is created from destination to origin. This prompted us to further design the buckets priority queue to be configured as a set or a stack (LIFO). It can also be configured to be a queue (FIFO). We implemented the A* Search algorithm using two buckets priority queues. One was configured to be a set and the other was configured to be a stack.

For testing and analysis, we developed a program that uses a min heap as a priority queue and used it to perform the A* Search algorithm. We also developed a program that implements the buckets data structure as a priority queue and used it to perform the A* Search algorithm. Fig. 3 presents the results in the form of a plot with insert, delete, and total (insert + delete) times for both the heap program and the buckets program for various matrix sizes. The plot shows that as the number of cells in the matrix that is searched increases, the efficiency of the buckets data structure increases.

Fig. 3 shows that, for the A* Search algorithm delete times for the heap are longer than but close to the delete times for the buckets data structure. However, the insert times for the heap are longer than the insert times for the buckets data structure. The insert time for the buckets data structure is 0.63 milliseconds at 1 million data elements. The insert time for the heap is 0.93 milliseconds at 1 million data elements. This is more than the buckets data structure's total time, which is 0.91 milliseconds. Thus, the buckets data structure provided a 32.3% insert time improvement when compared with the heap. The difference in the total times for the data structures is 0.44 (1.35 heap – 0.91 buckets) milliseconds which is a 32.6% improvement achieved by the buckets data structure.

In addition to a priority queue operation improvement, there is a decrease in total execution time to 2 seconds (4 seconds for the heap – 2 seconds for the buckets data structure), which is a 57.1% improvement achieved by the buckets data structure. The heapify process to maintain the heap property causes the program to run twice as long as it does with the buckets data structure. The resulting paths that were generated by both data structures were identical for every test case with the various numbers of data elements. Therefore, the buckets data structure did not cause any loss of accuracy.

IX. Queue or Stack Configuration

Since the buckets priority queue uses linked lists to store the nodes that are inserted into it, it did not require a lot of effort to design it to be configured as a queue or a stack. The required steps were to first make the linked lists doubly linked lists. The next step was to create attributes that determine whether to insert at the tail or head and whether to delete from the head or tail.

We determined through testing that with the buckets priority queue it is best to insert at the tail and delete from the head for a queue and insert at the head and delete from the head for a stack. Operating a queue and a stack in this fashion is more efficient from a time perspective than inserting at the head and deleting from the tail for a queue and inserting at the tail and deleting from the tail for a stack. There are two attributes, insert_end and delete_end, which determine where to insert and delete data elements, respectively. The insert_end attribute can have the following settings:

1) insert at the tail
2) insert at the head
3) 50% probability of inserting at the tail or head

The delete_end attribute can have the following settings:

1) delete from the head
2) delete from the tail
3) 50% probability of deleting from the head or tail

The option to randomly insert at the tail or head and delete from the head or tail with a 50% probability allowed the buckets data structure to find the goal state from an initial state of the 15-puzzle application in 1,056,405 less iterations than when a heap was used.

X. Set Configuration

The design and implementation phases of the buckets priority queue uncovered interesting aspects of a priority queue. The driving focus and reason for designing a buckets priority queue was to eliminate the overhead associated with heaps. The overhead was eliminated by putting the same or similar values into one bucket. This led to a developmental process of identifying duplicate values. The design handles the first insert into and the last delete from a bucket separately from all subsequent inserts and deletes. Thus, to configure the buckets priority queue into a set is as simple as preventing subsequent inserts into a bucket. An attribute (isASet) is used to configure the buckets priority queue as a set. If the buckets priority queue is configured to be a set, whenever a bucket is not empty, any attempts to add additional nodes into the bucket will not be allowed. For a heap to be used as a set, on average half of the nodes in the queue must be searched each time a node is inserted, to determine if the value being inserted is already in the queue. The buckets priority queue can perform this task with an O(1) time complexity.

XI. Contains Method

The buckets priority queue has a method, called "contains",

that returns TRUE or FALSE if a node with a certain value is in the queue. For a heap to have this method, on average half of the nodes on the queue must be searched to determine if a node with a certain value is in the queue. The buckets priority queue can perform this task with an O(1) time complexity.

XII. Double-Ended Priority Queue

The buckets priority queue was already designed to be configured to replace a min heap or a max heap. The data structure had an attribute (min_max) that was set to 0 to configure it to replace a max heap, or set to 1 to configure it to replace a min heap. There were places in the code where statements depended upon the value of min_max. For example, either the pointer to the bucket with the minimum value(s) or the maximum value(s) was updated dependent upon the value of min_max. The min_max attribute was eliminated, and the places in the code where it was used were changed to execute both statements to keep track of the buckets with the minimum and maximum value(s).

By adding this extra feature, the code became more efficient. A parameter (delete_end) is passed to the delete and top methods and is set to 0 to return the maximum value and set to 1 to return the minimum value. The effort required to implement and maintain a DEPQ with the buckets priority queue is less complex and more efficient than it is with a heap.

One benefit of a DEPQ that simultaneously keeps track of the minimum and maximum value(s) is that nodes that are determined to be unusable on either end of the queue can be deleted from the queue. When an application already has more nodes in the priority queue than will be processed, the excess nodes can be deleted to free up memory. This can be performed from either end of the priority queue.

After updating the code to convert the buckets data structure to be a DEPQ, thorough testing was done to ensure that after several inserts and deletes the pointers to the minimum and maximum value(s) were still valid. Nodes were inserted into the buckets priority queue and the proper updating of the pointers to minimum and maximum value(s) was verified. Then, nodes with both minimum and maximum value(s) were deleted from the buckets priority queue and the proper updating of the pointers to minimum and maximum value(s) was verified again. This procedure was executed several times to fully verify that the data structure was working as expected.

XIII. Stress Testing

Every task of this project and, in fact the whole pursuit of designing an efficient priority queue data structure, led to conducting stress testing. However, the idea didn't come to realization until converting the buckets priority queue into a DEPQ. After completing the DEPQ testing and verification process, there was a realization that all the applications, except for the transportation problem, have a common theme from a priority queue operations perspective. It consists of deleting a node from the queue, performing some processing, and inserting multiple nodes into the queue. The adaptive multivariate integration application that we used for testing in [20] deletes a region, divides it into 2, 4, 8, or 16 subregions, evaluates the subregions, and inserts them into the queue repeatedly until a tolerated error or maximum number of subdivisions is reached. The 15-puzzle application deletes a state, creates up to 4 new states, evaluates them, and inserts them into the queue repeatedly until the goal state is reached. The A* Search algorithm deletes a cell, identifies up to 8 new surrounding cells, evaluates them, and inserts them into the queue repeatedly until the destination is reached. The Huffman codes application has a similar priority queue process, but after the data elements are loaded into the queue, two nodes are deleted, joined and inserted into the queue. The transportation problem is the only application we implemented that does not have this type of continuous priority queue manipulation. Our proposed initial feasible solution for the transportation problem repeatedly inserts nodes into two priority queues and deletes all of the nodes from the priority queues before inserting more nodes into the priority queues.

We developed an application that isolates a priority queue's operations and analyzes its performance. The application deletes a node and then inserts a configured number of nodes, repeatedly until a configured number of data elements are inserted into the priority queue. The times spent inserting and deleting are accumulated separately and totaled.

We developed a program that uses a min heap as a priority queue and used it to perform stress testing. We also developed a program that uses the buckets data structure as a priority queue and used it to perform stress testing. The results follow in a series of figures with plots that have insert, delete, and total (insert + delete) times for both the heap program and the buckets program at various stages of the queue's capacity until the queue contains one million data elements. There is a plot for deleting one node and inserting 2, 4, 8 and 16 nodes, respectively. The plots show that as the number of data elements increases, the efficiency of the buckets data structure increases. The buckets priority queue yields roughly a 20% time improvement on inserting and a more than 90% time improvement on deleting as compared to the heap. We present the plots in decreasing order of the number of nodes that are inserted. For the heap, as the number of nodes inserted for stress testing decreases, the number of calls to heapify increases. Thus, the performance improvement achieved by the buckets data structure over the heap increases as well.

Fig. 4 shows that with deleting one node and inserting 16 nodes repeatedly until the queue has 1 million nodes, the delete time (0.04 deciseconds) for the program that utilizes the buckets data structure is a 91.1% improvement over the delete time (0.45 deciseconds) for the program that utilizes the heap. The insert time (0.69 deciseconds) for the program that utilizes the buckets data structure is a 18.8% improvement over the insert time (0.85 deciseconds) for the program that utilizes the heap. The difference in the total times for the data structures is 0.57 (1.30 heap – 0.73 buckets) deciseconds, which is a 43.8% improvement achieved by the buckets data structure.

In addition to a priority queue operation improvement,

there is a decrease in the total execution time when inserting a billion data elements. The total execution time difference is 1 minute and 23 seconds (4 minutes and 42 seconds for the heap – 3 minutes and 19 seconds for the buckets data structure), which is a 29.4% improvement provided by the buckets data structure. This execution time improvement will increase and become more substantial as the number of nodes inserted for stress testing decreases.

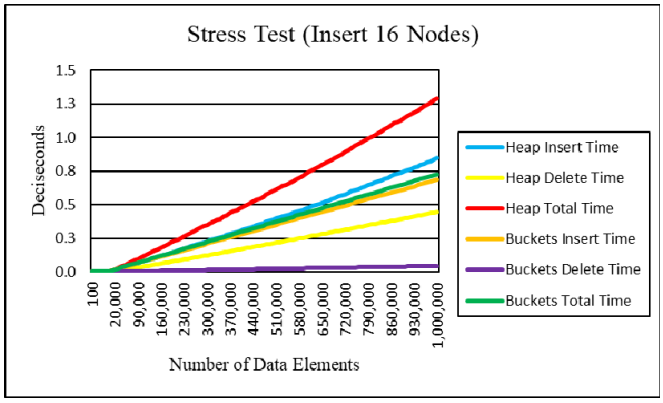

Fig. 4. Operation times for the Stress Test application (insert 16 nodes).

Fig. 5 shows that with deleting one node and inserting 8 nodes repeatedly until the queue has 1 million nodes, the delete time (0.09 deciseconds) for the program that utilizes the buckets data structure is a 90.1% improvement over the delete time (0.91 deciseconds) for the program that utilizes the heap. The insert time (0.76 deciseconds) for the program that utilizes the buckets data structure is a 19.1% improvement over the insert time (0.94 deciseconds) for the program that utilizes the heap. The difference in the total times for the data structures is 1.00 (1.85 heap – 0.85 buckets) deciseconds which is a 54.1% improvement provided by the buckets data structure.

In addition to a priority queue operation improvement, there is a total execution time improvement when inserting a billion data elements. The total execution time difference is 2 minutes and 40 seconds (6 minutes and 23 seconds for the heap – 3 minutes and 43 seconds for the buckets data structure) which is a 41.7% improvement provided by the buckets data structure.

Fig. 6 shows that when deleting one node and inserting 4 nodes repeatedly until the queue has 1 million nodes, the delete time (0.21 deciseconds) for the program that utilizes the buckets data structure is a 90.1% improvement over the delete time (2.12 deciseconds) for the program that utilizes the heap. The insert time (0.91 deciseconds) for the program that utilizes the buckets data structure is a 32.6% improvement of the insert time (1.35 deciseconds) over the program that utilizes the heap. The difference in the total times for the data structures is 2.35 (3.47 heap – 1.12 buckets) deciseconds which is a 67.7% improvement provided by the buckets data structure.

In addition to a priority queue operation improvement, there is a substantial decrease in the total execution time when inserting a billion data elements. The total execution time difference is 5 minutes and 46 seconds (10 minutes and 28 seconds for the heap – 4 minutes and 42 seconds for the buckets data structure) which is a 55.1% improvement achieved by the buckets data structure. The heapify process to maintain the heap property causes the program to run more than twice as long as with the buckets data structure.

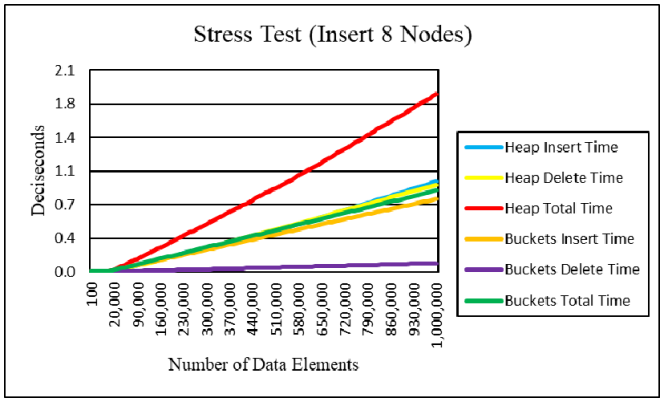

Fig. 5. Operation times for the Stress Test application (insert 8 nodes).

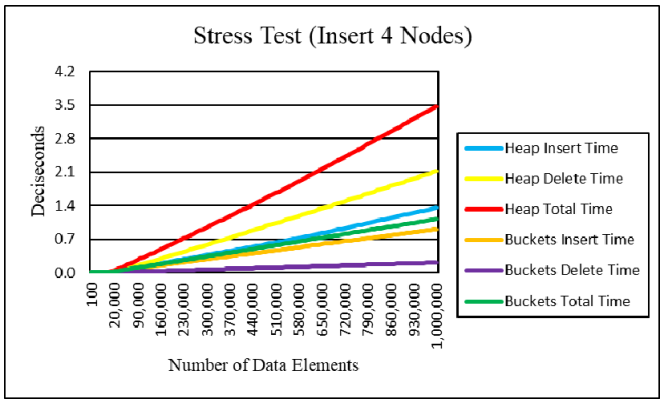

Fig. 6. Operation times for the Stress Test application (insert 4 nodes).

Fig. 7 shows that, with deleting 1 node and inserting 2 nodes repeatedly until the queue has 1 million nodes, the delete time (0.62 deciseconds) for the program that utilizes the buckets data structure is a 90.2% improvement over the delete time (6.30 deciseconds) for the program that utilizes the heap. The insert time (1.44 deciseconds) for the program that utilizes the buckets data structure is a 17.2% improvement of the insert time (1.74 deciseconds) over the program that utilizes the heap. The difference in the total times for the data structures is 5.98 (8.04 heap – 2.06 buckets) deciseconds which is a 74.4% improvement achieved by the buckets data structure.

In addition to a priority queue operation improvement, there is a substantial decrease in the total execution time when inserting a billion data elements, amounting to 15 minutes and 51 seconds (23 minutes and 53 seconds for the heap – 8 minutes and 2 seconds for the buckets data structure) which is a 66.4% improvement achieved by the buckets data structure. The heapify process to maintain the heap property causes the program to run almost three times as long as it does with the buckets data structure.

XIV. Summary

Table I shows insert, delete, and total (insert + delete) times for both the heap and the buckets data structures at 1 million data elements for all applications and configurations that we tested and analyzed and it shows the operation time

improvement that the buckets data structure provides.

Table II shows execution times for both the heap and the buckets data structure inserting 1 million (1 billion for stress testing) data elements for all applications and configurations that we tested and analyzed and it shows the execution time improvement that the buckets data structure provides. Table II also shows the number of direct and recursive heapify calls that were made in the program that utilizes the heap for every application and configuration that we tested and analyzed. Because we were able to omit the heapify process, all those calls to heapify were alleviated by the buckets data structure.

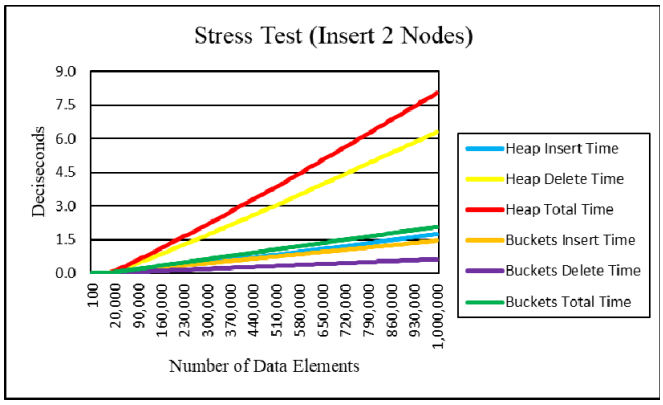

Fig. 7. Operation times for the Stress Test application (insert 2 nodes).

TABLE I: OPERATION TIMES AND IMPROVEMENT

Application	Insert Time		Delete Time		Total Time		Operation Time Improvement (%)
	Heap	Buckets	Heap	Buckets	Heap	Buckets	
Huffman Codes (s)	0.15	0.15	2.61	0.36	2.76	0.51	81.5
Transportation (s)	0.25	0.22	1.39	0.22	1.64	0.44	73.2
A* Search (ms)	0.93	0.63	0.42	0.28	1.35	0.91	32.6
Insert 16 Nodes (ds)	0.85	0.69	0.45	0.04	1.30	0.73	43.8
Insert 8 Nodes (ds)	0.94	0.76	0.91	0.09	1.85	0.85	54.1
Insert 4 Nodes (ds)	1.35	0.91	2.12	0.21	3.47	1.12	67.7
Insert 2 Nodes (ds)	1.74	1.44	6.30	0.62	8.04	2.06	74.4

TABLE II: EXECUTION TIMES AND IMPROVEMENT

Application	Execution Time (mins.)		Execution Time Improvement (%)	Heapify Calls	
	Heap	Buckets		Direct	Recursive
Huffman Codes	2.05	0.42	79.5	101,017,558	1,658,939,068
Transportation	2.25	1.07	52.4	151,542,810	1,167,808,107
A* Search	0.07	0.03	57.1	67,187	549,998
Insert 16 Nodes	4.70	3.32	29.4	66,666,666	1,849,418,575
Insert 8 Nodes	6.38	3.72	41.7	142,857,142	3,962,408,636
Insert 4 Nodes	10.47	4.70	55.1	333,333,332	9,244,380,216
Insert 2 Nodes	23.88	8.03	66.4	999,999,998	27,769,364,691

XV. CONCLUSION

We presented the results of an analysis where we compared the performance of a novel buckets priority queue data structure that we developed and a heap. We implemented several applications with both data structures. The results demonstrate that, for all applications implemented, our data structure outperforms heaps. In addition, the results demonstrate that the performance and efficiency of the buckets data structure increases as the number of data elements inserted into the queue increases. The new data structure can profoundly improve the performance of today's big data applications that rely on heaps. Furthermore, the performance improvement is obtained without any loss of accuracy of the application. We did not include applications that have extensive execution times. However, we demonstrated that the buckets data structure can accumulate a great deal of cost and time savings even for applications that have relatively short execution times.

XVI. FUTURE WORK

We will continue the implementation of real-world applications with the new data structure to showcase its innovation and use in the field of Computer Science both in the private and educational sectors. We are particularly interested in the areas of networking and IoT. We will further test the bucket data structure with applications that have extensive execution times to capitalize on more CPU-time savings.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

A conducted the research and analysis and developed the software; B provided direction and oversight; analyzed the data; A initially wrote the paper; B provided final edits; A, B approved the final version.

ACKNOWLEDGMENT

The authors would like to acknowledge Dr. J. Kapenga and Dr. D. Zeitler for their valuable comments.

REFERENCES

[1] J. W. J. Williams, "Algorithm 232: Heapsort," *Comm. ACM*, vol. 7, pp. 374–378, 1964.

[2] A. Genz and A. Malik, "An adaptive algorithm for numerical integration over an n-dimensional rectangular region," *Journal of Computational and Applied Mathematics*, vol. 6, pp. 295–302, 1980.

[3] R. Piessens, E. de Doncker, C. W. Überhuber, and D. K. Kahaner, "QUADPACK, a subroutine package for automatic integration," *Springer Series in Computational Mathematics*, vol. 1, Springer-Verlag, 1983.

[4] J. Berntsen, T. O. Espelid, and A. Genz, "An adaptive algorithm for the approximate calculation of multiple integrals," *ACM Trans. Math. Softw.*, vol. 17, pp. 437–451, 1991.

[5] J. Berntsen, T. O. Espelid, and A. Genz, "Algorithm 698: DCUHRE-an adaptive multidimensional integration routine for a vector of integrals," *ACM Trans. Math. Softw*, vol. 17, pp. 452–456, 1991.

[6] R. Cools and A. Haegemans, "Cubpack: Progress report," in *Numerical Integration, Recent Developments, Software and Applications, NATO ASI Series C: Mathematical and Physical Sciences*, T. O. Espelid and A. C. Genz, Eds., 1992, pp. 305–315.

[7] E. de Doncker, A. Genz, A. Gupta, and R. Zanny, "Tools for distributed adaptive multivariate integration on NOW's: PARINT1.0 release," *Supercomputing*, 1998.

[8] E. de Doncker, K. Kaugars, L. Cucos, and R. Zanny, "Current status of the ParInt package for parallel multivariate integration," in Proc. *Computational Particle Physics Symposium*, 2001, pp. 110–119.

[9] T. Hahn, "Cuba − a library for multidimensional numerical integration," *Comput. Phys. Commun.*, vol. 176, pp. 712–713, 2007.

[10] L. Jarzabek and P. Czarnul, "Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications," *J. Supercomputing*, vol. 73, pp. 5378–5401, 2017.

[11] O. E. Olagbemi and E. de Doncker, "Scalable algorithms for multivariate integration with ParAdapt and CUDA," in *Proc. 2019 Int. Conf. on Computer Science and Computational Intelligence*, 2019.

[12] E. de Doncker, F. Yuasa, A. Almulihi, N. Nakasato, H. Daisaka, and T. Ishikawa, "Numerical multi-loop integration on heterogeneous manycore processors," *The Journal of Physics: Conf. Series (JPCS)*, vol. 1525, 2019.

[13] E. de Doncker, F. Yuasa, and A. Almulihi, "Efficient GPU integration for multi-loop Feynman diagrams with massless internal lines," *Comp.*

and Exper. Simulations in Engineering, Mechanisms and Machine Science, vol. 75, pp. 737–747, 2019.
[14] E. de Doncker, F. Yuasa, O. Olagbemi, and T. Ishikawa, "Large scale automatic computations for Feynman diagrams with up to five loops," *Springer Lecture Notes in Computer science (LNCS)*, vol. 12253, pp. 145–162, 2020.
[15] E. de Doncker and F. Yuasa, "Self-energy Feynman diagrams with four loops and 11 internal lines," *Springer Lecture Notes in Computer science (LNCS)*, vol. 12953, pp. 160–175, 2021.
[16] Program for conversion of 32 bits single precision IEEE 754 floating point representation. (2021). [Online]. Available:https://www.geeksforgeeks.org/program-for-conversion-of-32-bits-single-precision-ieee-754-floating-point-representation/
[17] E. Horowitz, S. Sahni, and B. Rajasekaran, *Computer Algorithms/C++*, Computer Science Press, 1997.
[18] A. Babu, "Optimization in the transportation problem," *Medium, Towards Data Science*, July 5, 2020.
[19] D. Ratner and M. Warmuth, "The ($n^2 - 1$)-puzzle and related relocation problems," *Journal of Symbolic Computation*, vol. 10, pp. 111–137, 1990.
[20] J. Rhodes and E. de Doncker, "Design and implementation of an efficient priority queue data structure," in *Proc. 2022 Workshops onComputational Science and Its Applications*, Springer International Publishing, Cham, 2022, pp. 343–357.