

**PA5 Part 2 Report**

By

**Andrew Rimpici & Kevin Eaton**

**CSI 281**

**Data Structures & Algorithms**

**Fall 2017-18**

**1.** **Introduction**

The purpose of this experiment is to test the variation in speed of insertion, deletion, merging, and searching between four different data structures: Linked List, Doubly Linked List, Dynamic Array, and Dynamic ArrayList. All of these tests will be tested on three different list sizes: 1,000 10,000 and 100,000. For merging, it would be two lists of 1,000 10,000 and 100,000 respectively. Three iterations of the test will run and have their times averaged together for more accurate results. We predict it will take longer to insert into and remove elements from the dynamic array and dynamic arraylist as these structures will have to shift the other elements in the array over after each insertion or deletion. We also predict it will be faster to merge two lists together than it would be to add elements one by one into a structure.

**2.** **Background**

***Linked List***

A Linked List is a data structure made up of objects known as nodes that are linked together through a series of pointers. Each node holds a value and a pointer to the next node in the list. In every linked list, there is an mHead that specifies the first node in the list and in some cases, mostly for convenience, there is an mTail that specifies the last node. A positive to using a linked lists is that insertion and deletion of nodes is faster and less CPU intensive than it would be with an array [1]. Another advantage to linked lists over arrays is that the size of the structure is flexible and does not need a fixed size to be set at runtime [1]. In addition, there is no memory wastage when resizing the linked list compared to a dynamic array. Dynamic arrays often have empty slots at the end of the list due to resizing at runtime, while linked lists can scale using only the amount of memory needed [1]. A disadvantage to linked lists is that they take up more memory per element since each node holds the element and a pointer [1]. Another disadvantage is that you have to traverse through the whole list to access an element unlike where arrays have random access capabilities [1]. Finally, another disadvantage for single linked lists is that they cannot traverse the list in reverse. This means in order to get to the previous element, you must traverse the whole list again instead of simply stepping back once [1].

***Doubly Linked List***

A Doubly Linked List is a data structure very similar to a linked list. The only difference is that each node in the list now has an extra pointer that points to the previous node in the list. This allows for the advantage of reverse traversion throughout the list [2]. Another advantage to using a doubly linked list instead of a regular linked list is that when deleting we now have access to the previous node through the previous pointer. With a regular linked list we would have to keep track of the previous node [2]. A disadvantage to using a doubly linked list is that it takes up more space than a linked list per element because each node now has to hold a pointer to the previous node in addition to a pointer to the next node [2]. Finally, another disadvantage is the additional requirement of maintaining an extra previous pointer [2].

***Dynamic Array***

A Dynamic Array allows for the efficiency of an array combined with the flexibility of a resizable structure. An advantage of arrays is that all of the memory is blocked together which allows for the random access of elements in the list [3]. Random access is not possible with linked lists. Another advantage is that is uses less memory per element compared to linked lists as it does not need to store pointers for each index [3]. Another advantage dynamic arrays have over linked lists is that it allows for fast sorting since it uses block memory [3]. A disadvantage to arrays is that takes more computation time to insert or remove elements from the structure. When adding or removing a new element, the array must shift all of the other elements down by one in the list. In linked lists, adding or removing an element is as simple as adding another node and updating the pointers to include the node. This version of the dynamic array is resized every time an element is inserted or removed. This may slow down insertion and deletion times, but it will allow for the array to only use the necessary amount of memory for its size.

***Dynamic ArrayList***

The main difference between the dynamic array and the dynamic arraylist is how each structure handles insertion, deletion, and capacity. This version of the dynamic array is modeled after the java version of arraylists. This structure uses a capacity variable to keep track of the underlying array size while having another variable, size, to keep track of how many elements in the list are being used. This allows for faster insertion and deletion times since the underlying array does not need to be resized every time. An advantage of this structure over the other dynamic array is that it will only resize if the size variable reaches the capacity, or if the array becomes small enough to where the capacity should shrink. In this version, the capacity scales by 10 each time it needs to grow or shrink. Example: An arraylist of size 9 will have a capacity of 10, while an arraylist of size 11 will have a capacity of 100. While it allows for faster insertion and deletion, a clear disadvantage of the arraylist is that it takes up a lot of unused memory. A list of size 11 has 89 empty spots waiting to be used which is memory that could be used for other things.

**3.** **Implementation Detail**

***Insertion***

Insertion testing will yield results that show which list structure is more efficient when adding new elements to the list. It will consist of three tests, one test where elements will be prepended to the list, one test where elements will be appended to the back of the list, and one test where elements will be inserted into the middle of the list (index equal to: list size / 2).

***Deletion***

Deletion testing will yield results that show which list structure is more efficient when removing elements from the list. It will consist of the same three tests except instead of inserting elements, elements will be removed. One test will repeatedly remove the first element from the list until the list is empty, another test will repeatedly remove the middle element from the list until the list is empty, and finally, another test will repeatedly remove the ending element from the list until the list is empty.

***Sequential Search***

Search testing will yield results that show which structure is most efficient when searching sequentially. Each list structure will use sequential search on ordered lists in ascending order. Each list is populated in ascending order with the value of the current index spot. The search key value will be the same for each structure, and will be half the size of the list itself.

***Merging***

Merge testing will yield results that show which structure is best at merging two of its lists together. Two of the same structure, in ascending order and equal length, will be merged together. The time it takes to merge the lists will then be compared to the other merges and also how long it takes to insert elements into a list one by one.

Overall, our experiment is to analyze how these four data structures differ when handling insertion, deletion, sequential search, and merging.

**4.** **Experimentation Detail**

|  |  |
| --- | --- |
| **System Type** | Microsoft Windows 10 Home 64bit |
| **Processor Type** | Intel® Core™ i7-6700 4 Core(s), 8 Logical Processor(s) |
| **CPU Speed** | CPU @ 3.4GHz |
| **On Board Memory** | 32.0 GB |

**Summary Data** *(All data is measured in seconds)*

**Linked List Averages**

|  |  |  |  |
| --- | --- | --- | --- |
| Linked List Insertion Average | | | |
|  | Front | Middle | End |
| 1000 | 2.59E-04 | 1.20E-03 | 2.49E-04 |
| 10000 | 2.03E-03 | 1.81E-01 | 1.82E-03 |
| 100000 | 1.99E-02 | 2.16E+01 | 2.15E-02 |
| Linked List Deletion Average | | | |
|  | Front | Middle | End |
| 1000 | 2.27E-04 | 1.13E-03 | 2.24E-03 |
| 10000 | 1.91E-03 | 1.59E-01 | 2.70E-01 |
| 100000 | 2.02E-02 | 2.14E+01 | 2.98E+01 |

|  |  |
| --- | --- |
| Linked List Search Average | |
|  | Sorted Search where search key = (SIZE / 2) |
| 1000 | 3.51E-06 |
| 10000 | 5.01E-05 |
| 100000 | 4.58E-04 |
| Linked List Merge Average | |
|  | Merging of two equally sized and ascendingly sorted lists |
| 1000 | 5.07E-04 |
| 10000 | 4.60E-03 |
| 100000 | 4.08E-02 |

**Doubly Linked List Averages**

|  |  |  |  |
| --- | --- | --- | --- |
| Doubly Linked List Insertion Average | | | |
|  | Front | Middle | End |
| 1000 | 2.33E-04 | 1.30E-03 | 2.63E-04 |
| 10000 | 2.00E-03 | 1.51E-01 | 1.98E-03 |
| 100000 | 1.91E-02 | 1.74E+01 | 2.00E-02 |
| Doubly Linked List Deletion Average | | | |
|  | Front | Middle | End |
| 1000 | 2.45E-04 | 1.02E-03 | 2.11E-03 |
| 10000 | 2.00E-03 | 1.53E-01 | 2.63E-01 |
| 100000 | 1.89E-02 | 1.88E+01 | 2.91E+01 |

|  |  |
| --- | --- |
| Doubly Linked List Search Average | |
|  | Sorted Search where search key = (SIZE / 2) |
| 1000 | 5.31E-06 |
| 10000 | 8.69E-05 |
| 100000 | 7.47E-04 |
| Doubly Linked List Merge Average | |
|  | Merging of two equally sized and ascendingly sorted lists |
| 1000 | 5.85E-04 |
| 10000 | 4.59E-03 |
| 100000 | 4.25E-02 |

**Dynamic Array Averages**

|  |  |  |  |
| --- | --- | --- | --- |
| Dynamic Array Insertion Average | | | |
|  | Front | Middle | End |
| 1000 | 1.93E-03 | 1.74E-03 | 1.73E-03 |
| 10000 | 1.43E-01 | 1.31E-01 | 1.26E-01 |
| 100000 | 1.34E+01 | 1.29E+01 | 1.22E+01 |
| Dynamic Array Deletion Average | | | |
|  | Front | Middle | End |
| 1000 | 2.84E-03 | 2.12E-03 | 1.60E-03 |
| 10000 | 2.12E-01 | 1.59E-01 | 1.07E-01 |
| 100000 | 2.07E+01 | 1.53E+01 | 9.91E+00 |

|  |  |
| --- | --- |
| Dynamic Array Search Average | |
|  | Sorted Search where search key = (SIZE / 2) |
| 1000 | 3.31E-06 |
| 10000 | 2.79E-05 |
| 100000 | 2.63E-04 |
| Dynamic Array Merge Average | |
|  | Merging of two equally sized and ascendingly sorted lists |
| 1000 | 1.36E-05 |
| 10000 | 1.07E-04 |
| 100000 | 9.78E-04 |

**Dynamic ArrayList Averages**

|  |  |  |  |
| --- | --- | --- | --- |
| Dynamic ArrayList Insertion Average | | | |
|  | Front | Middle | End |
| 1000 | 1.18E-03 | 6.39E-04 | 4.85E-05 |
| 10000 | 9.91E-02 | 4.83E-02 | 4.51E-04 |
| 100000 | 9.54E+00 | 4.77E+00 | 3.61E-03 |
| Dynamic ArrayList Deletion Average | | | |
|  | Front | Middle | End |
| 1000 | 3.60E-02 | 1.95E-02 | 3.57E-02 |
| 10000 | 1.06E-01 | 5.60E-02 | 1.06E-01 |
| 100000 | 1.07E+01 | 5.33E+00 | 1.07E+01 |

|  |  |
| --- | --- |
| Dynamic ArrayList Search Average | |
|  | Sorted Search where search key = (SIZE / 2) |
| 1000 | 1.90E-06 |
| 10000 | 1.58E-05 |
| 100000 | 1.36E-04 |
| Dynamic ArrayList Merge Average | |
|  | Merging of two equally sized and ascendingly sorted lists |
| 1000 | 2.23E-04 |
| 10000 | 2.37E-03 |
| 100000 | 2.14E-02 |

**5.** **Discussion and Conclusion**

The data collected shows very interesting results and reaffirms our predictions. After looking at the data, the Dynamic Array structure is the fastest of all the structures when merging two lists together. Although the Dynamic Array was quicker than the Dynamic ArrayList at merging, the Dynamic ArrayList was quicker when inserting and removing elements from the structure. This is because the list did not have to resize every time an element was added or removed.

Linked Lists and Doubly Linked Lists were extremely fast inserting at the front and back of the list, but much slower when inserting data into the middle. This is most likely because these structures do not have random access, like arrays, and must traverse to the middle to insert the node each time. The trend continues with the linked lists when deleting nodes as well. Deleting nodes were fast from the front and back of the list, but slower in the middle likely for the same reason as inserting.

All the results from sequentially searching through each structure are very similar. It took close to the same amount of time for each structure to find the search key. This may have been because of how we implemented our test in that each list was sorted and the key was in the middle of the list for each test.

An interesting fact I noticed is that it took noticeably longer for the arraylist to delete its nodes from the end than to add them to the list, but this may have been caused by how our implementation is coded.

Merging for linked lists and doubly linked lists does not yield any noticeable time improvements, but for the dynamic array, merging was much faster than inserting. The dynamic arraylist was faster inserting one-by-one at the end of the list rather than merging but this could be due to how it was implemented.

On average, the best lists for insertion and deletion are the linked list and the doubly linked list. The dynamic arraylist is very fast when adding and deleting from the end of the list, but slows down when adding/deleting anywhere else in the list. Linked lists and doubly linked lists are faster than the arrays overall across the three sections: front insertion, middle insertion, and end insertion.

Overall, there is a very nice linear increase in the time it takes to run all of these tests on 1,000 10,000 and 100,000 sized lists which shows that the time complexity stays linear, as the lists scale up.

**6.** **References**

[1] The Crazy Programmer. (2017). *Advantages and Disadvantages of Linked List - The Crazy Programmer*. [online] Available at: https://www.thecrazyprogrammer.com/2016/11/advantages-disadvantages-linked-list.html [Accessed 19 Oct. 2017].

[2] GeeksforGeeks. (2017). *Doubly Linked List | Set 1 (Introduction and Insertion) - GeeksforGeeks*. [online] Available at: http://www.geeksforgeeks.org/doubly-linked-list/ [Accessed 19 Oct. 2017].

[3] Tutorialcup.com. (2017). *Advantages and Disadvantages of Array in C Programming*. [online] Available at: https://www.tutorialcup.com/cprogramming/array-advantages-disadvantages.htm [Accessed 19 Oct. 2017].

*Link to the PA5 GitHub*

*https://github.com/kevineaton603/ListStructurePreformance/*

**7.** **Appendix**

The following pages hold all of the raw data charts from the experiment.

**INSERTION DATA** *(All data is measured in seconds)*

**Linked List Insertion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked List Front Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.40E-04 | 2.48E-04 | 2.89E-04 | 2.59E-04 |
| 10000 | 1.90E-03 | 2.16E-03 | 2.03E-03 | 2.03E-03 |
| 100000 | 2.02E-02 | 1.95E-02 | 2.00E-02 | 1.99E-02 |
| Linked List Middle Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.12E-03 | 1.37E-03 | 1.10E-03 | 1.20E-03 |
| 10000 | 1.87E-01 | 1.78E-01 | 1.77E-01 | 1.81E-01 |
| 100000 | 1.99E+01 | 2.48E+01 | 2.00E+01 | 2.16E+01 |
| Linked List End Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.75E-04 | 2.42E-04 | 2.30E-04 | 2.49E-04 |
| 10000 | 1.88E-03 | 1.82E-03 | 1.77E-03 | 1.82E-03 |
| 100000 | 2.20E-02 | 2.05E-02 | 2.20E-02 | 2.15E-02 |

**Doubly Linked List Insertion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Doubly Linked List Front Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.16E-04 | 2.31E-04 | 2.52E-04 | 2.33E-04 |
| 10000 | 2.10E-03 | 2.05E-03 | 1.84E-03 | 2.00E-03 |
| 100000 | 1.93E-02 | 1.86E-02 | 1.94E-02 | 1.91E-02 |
| Doubly Linked List Middle Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.59E-03 | 1.11E-03 | 1.21E-03 | 1.30E-03 |
| 10000 | 1.48E-01 | 1.54E-01 | 1.49E-01 | 1.51E-01 |
| 100000 | 1.75E+01 | 1.77E+01 | 1.70E+01 | 1.74E+01 |
| Doubly Linked List End Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.51E-04 | 2.27E-04 | 3.13E-04 | 2.63E-04 |
| 10000 | 1.87E-03 | 2.22E-03 | 1.85E-03 | 1.98E-03 |
| 100000 | 1.99E-02 | 1.98E-02 | 2.04E-02 | 2.00E-02 |

**Dynamic Array Insertion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic Array Front Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.73E-03 | 1.76E-03 | 2.28E-03 | 1.93E-03 |
| 10000 | 1.57E-01 | 1.37E-01 | 1.34E-01 | 1.43E-01 |
| 100000 | 1.36E+01 | 1.35E+01 | 1.32E+01 | 1.34E+01 |
| Dynamic Array Middle Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.76E-03 | 1.80E-03 | 1.67E-03 | 1.74E-03 |
| 10000 | 1.35E-01 | 1.31E-01 | 1.28E-01 | 1.31E-01 |
| 100000 | 1.30E+01 | 1.29E+01 | 1.27E+01 | 1.29E+01 |
| Dynamic Array End Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.76E-03 | 1.73E-03 | 1.70E-03 | 1.73E-03 |
| 10000 | 1.27E-01 | 1.27E-01 | 1.24E-01 | 1.26E-01 |
| 100000 | 1.25E+01 | 1.21E+01 | 1.21E+01 | 1.22E+01 |

**Dynamic ArrayList Insertion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic ArrayList Front Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.08E-03 | 1.07E-03 | 1.40E-03 | 1.18E-03 |
| 10000 | 9.91E-02 | 9.90E-02 | 9.92E-02 | 9.91E-02 |
| 100000 | 9.66E+00 | 9.49E+00 | 9.47E+00 | 9.54E+00 |
| Dynamic ArrayList Middle Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 6.24E-04 | 7.07E-04 | 5.87E-04 | 6.39E-04 |
| 10000 | 4.83E-02 | 4.90E-02 | 4.75E-02 | 4.83E-02 |
| 100000 | 4.84E+00 | 4.74E+00 | 4.74E+00 | 4.77E+00 |
| Dynamic ArrayList End Insertion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 5.14E-05 | 4.48E-05 | 4.93E-05 | 4.85E-05 |
| 10000 | 3.71E-04 | 4.85E-04 | 4.97E-04 | 4.51E-04 |
| 100000 | 3.50E-03 | 3.57E-03 | 3.75E-03 | 3.61E-03 |

**DELETION DATA** *(All data is measured in seconds)*

**Linked List Deletion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked List Front Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.20E-04 | 2.70E-04 | 1.92E-04 | 2.27E-04 |
| 10000 | 1.81E-03 | 2.01E-03 | 1.90E-03 | 1.91E-03 |
| 100000 | 1.85E-02 | 2.20E-02 | 2.00E-02 | 2.02E-02 |
| Linked List Middle Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.06E-03 | 1.31E-03 | 1.00E-03 | 1.13E-03 |
| 10000 | 1.56E-01 | 1.64E-01 | 1.56E-01 | 1.59E-01 |
| 100000 | 1.84E+01 | 1.83E+01 | 2.75E+01 | 2.14E+01 |
| Linked List End Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.17E-03 | 2.19E-03 | 2.37E-03 | 2.24E-03 |
| 10000 | 2.61E-01 | 2.76E-01 | 2.72E-01 | 2.70E-01 |
| 100000 | 2.88E+01 | 2.94E+01 | 3.11E+01 | 2.98E+01 |

**Doubly Linked List Deletion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Doubly Linked List Front Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.36E-04 | 2.66E-04 | 2.32E-04 | 2.45E-04 |
| 10000 | 2.28E-03 | 1.92E-03 | 1.81E-03 | 2.00E-03 |
| 100000 | 1.85E-02 | 1.88E-02 | 1.92E-02 | 1.89E-02 |
| Doubly Linked List Middle Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 9.95E-04 | 9.83E-04 | 1.07E-03 | 1.02E-03 |
| 10000 | 1.53E-01 | 1.54E-01 | 1.53E-01 | 1.53E-01 |
| 100000 | 1.86E+01 | 1.87E+01 | 1.90E+01 | 1.88E+01 |
| Doubly Linked List End Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.12E-03 | 1.96E-03 | 2.25E-03 | 2.11E-03 |
| 10000 | 2.58E-01 | 2.64E-01 | 2.66E-01 | 2.63E-01 |
| 100000 | 2.91E+01 | 2.91E+01 | 2.92E+01 | 2.91E+01 |

**Dynamic Array Deletion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic Array Front Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.67E-03 | 2.72E-03 | 3.12E-03 | 2.84E-03 |
| 10000 | 2.08E-01 | 2.16E-01 | 2.12E-01 | 2.12E-01 |
| 100000 | 2.04E+01 | 2.09E+01 | 2.08E+01 | 2.07E+01 |
| Dynamic Array Middle Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.11E-03 | 2.12E-03 | 2.13E-03 | 2.12E-03 |
| 10000 | 1.58E-01 | 1.62E-01 | 1.57E-01 | 1.59E-01 |
| 100000 | 1.51E+01 | 1.56E+01 | 1.52E+01 | 1.53E+01 |
| Dynamic Array End Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.65E-03 | 1.56E-03 | 1.60E-03 | 1.60E-03 |
| 10000 | 1.07E-01 | 1.08E-01 | 1.07E-01 | 1.07E-01 |
| 100000 | 9.82E+00 | 1.01E+01 | 9.84E+00 | 9.91E+00 |

**Dynamic ArrayList Deletion Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic ArrayList Front Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.14E-03 | 1.16E-03 | 1.20E-03 | 1.17E-03 |
| 10000 | 1.06E-01 | 1.08E-01 | 1.05E-01 | 1.06E-01 |
| 100000 | 1.06E+01 | 1.08E+01 | 1.06E+01 | 1.07E+01 |
| Dynamic ArrayList Middle Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 6.25E-04 | 5.97E-04 | 7.54E-04 | 6.58E-04 |
| 10000 | 5.70E-02 | 5.64E-02 | 5.46E-02 | 5.60E-02 |
| 100000 | 5.30E+00 | 5.40E+00 | 5.30E+00 | 5.33E+00 |
| Dynamic ArrayList End Deletion | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.16E-03 | 1.18E-03 | 1.35E-03 | 1.23E-03 |
| 10000 | 1.05E-01 | 1.09E-01 | 1.04E-01 | 1.06E-01 |
| 100000 | 1.06E+01 | 1.08E+01 | 1.06E+01 | 1.07E+01 |

**SELECTION SEARCH DATA** *(All data is measured in seconds)*

**Linked List Selection Search Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked List Sorted Selection Search for key equal to (SIZE / 2) | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 5.11E-06 | 3.61E-06 | 1.80E-06 | 3.51E-06 |
| 10000 | 6.10E-05 | 7.09E-05 | 1.83E-05 | 5.01E-05 |
| 100000 | 6.65E-04 | 5.27E-04 | 1.81E-04 | 4.58E-04 |

**Doubly Linked List Selection Search Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Doubly Linked List Sorted Selection Search for key equal to (SIZE / 2) | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 4.81E-06 | 3.91E-06 | 7.21E-06 | 5.31E-06 |
| 10000 | 9.52E-05 | 9.40E-05 | 7.15E-05 | 8.69E-05 |
| 100000 | 5.96E-04 | 1.06E-03 | 5.86E-04 | 7.47E-04 |

**Dynamic Array Selection Search Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic Array Sorted Selection Search for key equal to (SIZE / 2) | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.50E-06 | 1.50E-06 | 6.91E-06 | 3.31E-06 |
| 10000 | 1.11E-05 | 1.11E-05 | 6.16E-05 | 2.79E-05 |
| 100000 | 1.08E-04 | 1.10E-04 | 5.69E-04 | 2.63E-04 |

**Dynamic ArrayList Selection Search Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic ArrayList Sorted Selection Search for key equal to (SIZE / 2) | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.10E-06 | 1.80E-06 | 1.80E-06 | 1.90E-06 |
| 10000 | 1.17E-05 | 1.83E-05 | 1.74E-05 | 1.58E-05 |
| 100000 | 1.12E-04 | 1.81E-04 | 1.14E-04 | 1.36E-04 |

**MERGE DATA** *(All data is measured in seconds)*

**Linked List Merge Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linked List | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 5.37E-04 | 5.14E-04 | 4.70E-04 | 5.07E-04 |
| 10000 | 4.68E-03 | 4.56E-03 | 4.57E-03 | 4.60E-03 |
| 100000 | 4.04E-02 | 4.12E-02 | 4.08E-02 | 4.08E-02 |

**Doubly Linked Merge Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Doubly Linked List | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 6.31E-04 | 5.22E-04 | 6.02E-04 | 5.85E-04 |
| 10000 | 4.72E-03 | 4.86E-03 | 4.19E-03 | 4.59E-03 |
| 100000 | 4.20E-02 | 4.31E-02 | 4.26E-02 | 4.25E-02 |

**Dynamic Array Merge Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic Array | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 1.35E-05 | 1.26E-05 | 1.47E-05 | 1.36E-05 |
| 10000 | 1.04E-04 | 1.11E-04 | 1.07E-04 | 1.07E-04 |
| 100000 | 1.00E-03 | 9.51E-04 | 9.82E-04 | 9.78E-04 |

**Dynamic ArrayList Merge Data**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dynamic ArrayList | | | | |
|  | Run 1 | Run 2 | Run 3 | Average |
| 1000 | 2.08E-04 | 2.27E-04 | 2.35E-04 | 2.23E-04 |
| 10000 | 2.24E-03 | 2.25E-03 | 2.61E-03 | 2.37E-03 |
| 100000 | 2.09E-02 | 2.16E-02 | 2.18E-02 | 2.14E-02 |