BINUS INTERNATIONAL

DATA STRUCTURES

&

OBJECT-ORIENTED-PROGRAMMING

Final Project Report

Submitted by

Osten Antonio - 2802546115

Kevin Makmur Kurniawan - 2802547553

Nicholas Nixon Iswanto - 2802546664

Table of Contents

1. Background

Digital calendar applications have become an integral part of modern life, from managing work schedules to planning personal activities. A calendar application is one of the most ubiquitous applications made today, with it being often integrated into other applications, such as email clients, with microsoft outlook, and it is one of the most common applications that is preinstalled in digital devices such as phones [1].

Often if not almost, calendar applications always have the functionality of task scheduling, as well as the ability to visualize upcoming tasks in a given timespan. In addition to basic scheduling, many calendar apps offer additional features, such as customizable categories for tasks, different view options, and more. These calendar applications play a pivotal role in enhancing productivity and ensuring that users can manage their time effectively.

A calendar application comes with many advantages, such as better deadline management and time management. People often will schedule their upcoming activities to try to make more efficient use of their time, and a calendar application helps in achieving that. In a study, it was shown that people access their calendar five or more times per day, and it also argues that scheduling is becoming a default to manage their work, even their leisure activities [2].

The frequency of individuals that access their calendars highlights the importance of calendars in their daily routines. Studies and surveys have shown that the reliance on digital calendars to manage their life is 70% of adults, with 23.3% being on a desktop calendar, and 46.7% on mobile devices. This shows the importance of optimizing digital calendars [3]. In another study, it was shown that event calendars such as Google Calendar can consume up to 40% of the total simulation time, which means that nearly half of the computational power is dedicated to managing the calendar operations, such as insertion and deletion [4].

As such, this project will attempt to create a calendar application app, similar to Google calendar, which has the goals listed above, and also to provide a simple, and effective way to organize tasks while also focusing on user-friendliness. However, a question arises, of how to optimize an already established application, in the sense of technicality such as speed, and memory taken, which can affect user experience. To achieve this multiple data structures will be used, such as a priority queue, which is a standard data structure which has been researched a lot in this use case [4]-[7].

2. Problem

Calendar applications is one of the important daily driver and there has been multiple issues when implementing a calendar application, namely:

- **Recurring events**
  - Managing recurring events introduces multiple complications, specifically redundancies if not handled correctly [8]. Recurring events involve generating multiple occurrences based on defined rules, for example, “repeat x every two days for two weeks”. If more occurrences of those events than necessary were to be stored in the database or data structures, it could lead to performance issues [9].
- **Storage**
  - As stated above, if every task/event were to be stored in the data structure, then the storage for these data structures would grow out of proportion, however, there is also an additional problem, which is the main calendar GUI. The main calendar GUI involves filtering already existing tasks, copying and creating a new GUI element to be placed on the

calendar grid. With every navigation of the calendar, this process repeats, and this would add more GUI elements to the existing ones, which may cause storage issues.

- **Speed**
  - This issue is also related to storage as the more GUI elements and tasks/events the application has to process, the slower the process gets, while this is a given in many applications, the rate of processing speed depends on the data structures used, and how they are used. Similarly, as mentioned before, nearly half of the computation power is being used for the calendar operations (insertion, deletion, etc) [4]. Both issues of speed and storage can be illustrated in multiple github sources, an example [10].

3. Solution

In finding a solution to solve the problem, the implementation of the application must be done in a way that optimizes both data storage and processing speed, while also taking in consideration of recurring events, and one way to do this is by caching. In hardware terms, cache memory plays a crucial role in enhancing performance by bridging the speed gap between the processor and main memory, and this can be implemented here, by having one cache variable to store the GUI elements when navigating the calendar [11]. This way, it doesn't need to search for the tasks again when the user performs the same operation, e.g. navigating the calendar.

As for recurring events, an intuitive way to store it is to define a rule, such as ones defined in iCal, 'rrule' [12]. Through that, storing the rule in the database, we can parse each rule for the tasks, and then dynamically generate a new task/event object for every recurrence that exists within the constraints of the existing tasks (such as filtering within a month in the calendar) only when it is needed. The concept of this solution is also outlined in the article "An Ontology Design Pattern for Representing Recurrent Situations.", as the article proposes a structured framework for representing recurring events, and dynamically generating event instances based on the rules within a specific timeframe [13].

3.1 Data structures used

- **ArrayLists**

  ArrayLists are resizable arrays. Arrays are part of the linear data structures, where it is a sequence of *n* items of the data type, which are stored contiguously in the computer memory [14]. The items stored in this can be accessed by specifying the position or index of where the item is stored. In this case, each of the indexes will store the task object, Task(int priority, String title, String body, boolean status,...). The main reason why ArrayLists will be used is due to the ease of implementation, and it is very easy to understand. It also allows for automatic growth and shrinking, as well as an efficient access time.

- **AVL Tree**

  A tree is a type of graph that has no cycles. It contains similar properties to a graph, where each node contains a type of data, in this case, it will be the task object. For this project, an ordered tree will be implemented, where all the children of each vertex are ordered [14]. More specifically, a binary tree, where a left or right side of the child/tree is assigned to have a higher priority. However, what differs from a binary tree compared to an AVL tree is that it maintains a strict balance condition to ensure optimal performance. A tree can be useful in this as it can efficiently represent the priorities between the tasks, and it has a low time complexity for inserting and deleting, being O(logn) [15].

- **Queue (Priority Queue)**

  A queue is another linear data structure, however, elements can only be accessed from the front or the end of the data structure. It operates on a "first-in–first-out" (FIFO) basis, similar to real life queue [14]. Queues have been a common data structure to be used in this use case [6]-[7]. However, it was found that the performance of queues can degrade where events are unevenly distributed over time [6]. This project will see how the performance of queue can be compared to other data structures, regardless of the paper's findings. A priority queue will be used here due to the inherent sorted nature of it.

- **LinkedList**

  A linked list is a linear data structure where each element contains data and a reference to another node of the same list, the next one in sequence [14]. It allows for easier item modification (insertion, deletion) compared to arraylists or queues. However, the items cannot be accessed directly, rather the lists need to be iterated to find the item. A linked list may be more advantageous compared to other data structures due to being able to modify the nodes directly, changing its position, deletion, insertions.

4. Results

4.1 Data Collected

Below are the indicators used to rank the data structures are determined by summing their performance scores across each respective input size (10, 100, 500, 1000 tasks):

Excellent (4 points) > Good (3 points) > Poor (2 points) > Worst (1 point)

However, when two or more data structures are tied in points, the winner is determined by the number of excellent points followed by the number of worst points.

The following are the results gathered for the average time taken and the used space for each type of operations:

4.1.1 Loading

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**9**	194.1258	686.0376	2,117.4803	2,875.4614
Linked List	**6**	207.9906	719.0799	2,204.6857	3,090.5334
Priority Queue	**14**	101.9051	418.4208	1,214.9248	1,712.0885
Binary Tree	**12**	100.5543	394.5211	2,078.9241	3,397.5599

Table 1. Average loading time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**13**	29,188.3424	35,267.4704	2,117.4803	2,875.4614
Linked List	**7**	30,598.4176	37,139.9104	2,204.6857	3,090.5334
Priority Queue	**12**	31,230.2512	36,163.7712	1,214.9248	1,712.0885
Binary Tree	**6**	30,046.4704	37,437.6992	2,078.9241	3,397.5599

Table 2. Average loading space benchmark results

4.1.2 Searching

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**8**	13.1806	29.1684	55.9073	86.1859
Linked List	**8**	17.6613	27.2883	50.0736	97.2846
Priority Queue	**15**	10.6962	10.4318	40.1037	65.5201
Binary Tree	**9**	9.3351	30.4194	53.4714	87.9510

Table 3. Average searching time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**15**	34,482.4736	42,899.6560	107,112,662.4	189,643.1616
Linked List	**11**	35,468.6192	43,542.9616	109,679.6048	196,164.4112
Priority Queue	**7**	37,471.6320	64,772.9376	113,837.4352	215,297.2960
Binary Tree	**7**	38,170.8128	41,933.1744	118,008.1728	232,081.1648

Table 4. Average searching space benchmark results

4.1.3 Creating

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**11**	97.5517	64.3224	155.0229	184.8458
Linked List	**9**	168.1454	88.7092	150.1571	204.1241
Priority Queue	**16**	72.7637	40.2373	78.2460	133.3255
Binary Tree	**4**	3,080.0601	3,072.0745	3,212.0411	3,248.3736

Table 5. Average creation time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**10**	43,227.8352	114,992.1040	232,777,844.8	383,831.8624
Linked List	**9**	49,814.6832	135,386.4656	184,927.4368	370,440.5120
Priority Queue	**12**	43,365.0176	117,420.8224	201,017.2896	309,403.2416
Binary Tree	**9**	43,938.1696	113,312.2976	202,949.1104	418,721.6256

Table 6. Average creation space benchmark results

4.1.4 Updating

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**8**	174.8640	461.5411	1,454.5852	2,413.2668
Linked List	**5**	201.1070	516.1913	1,493.0204	2,259.8130
Priority Queue	**14**	118.5499	321.8885	850.3779	1,270.9172
Binary Tree	**12**	113.9072	367.6473	1,481.0472	2,198.7180

Table 7. Average updating time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**12**	34,647.5344	54,142.5456	137,081,500.8	247,114.7536
Linked List	**12**	34,311.8592	60,666.4480	132,243.3648	274,239.2704
Priority Queue	**8**	34,583.6224	82,313.8016	142,101.6160	258,350.7376
Binary Tree	**8**	34,535.5984	60,059.8688	168,033.6992	289,077.8624

Table 8. Average updating space benchmark results

4.1.5 Deleting

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**11**	79.2799	287.8414	697.3644	1,136.7393
Linked List	**9**	68.7068	304.3747	1,493.0204	1,214.5552
Priority Queue	**16**	61.6588	263.7813	510.3516	668.0038
Binary Tree	**4**	3,059.1682	3,190.2860	3,767.5418	4,317.3885

Table 9. Average deletion time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**11**	36,100.9216	113,248.2864	251,996,512	429,186.5232
Linked List	**8**	45,176.4240	72,922.4912	257,793.4208	552,527.3056
Priority Queue	**13**	38,401.5888	111,614.0128	246,319.7120	367,520.4736
Binary Tree	**8**	37,257.9520	144,469.4016	285,949.5072	397,508.7408

Table 10. Average deletion space benchmark results

4.1.6 Sorting

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**8**	141.0072	505.8250	1,260.4903	2,156.5502
Linked List	**7**	132.9752	515.9435	1,274.0086	2,430.6086
Priority Queue	**16**	53.7608	370.8342	901.4760	1,432.9508
Binary Tree	**10**	82.9481	389.9987	1,607.0515	1,794.9153

Table 11. Average sorting time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**10**	41,194.4928	53,790.2736	143,874,483.2	273,494.2720
Linked List	**8**	41,899.7536	54,551.1616	145,595.3376	270,940.1104
Priority Queue	**9**	39,864.1648	54,755.3536	145,869.1504	270,590.4832
Binary Tree	**13**	35,879.6992	62,272.0592	139,703.9168	259,766.1936

Table 12. Average sorting space benchmark results

4.1.7 Filter

Average time (ms)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**11**	31.5783	181.1652	503.5596	734.4294
Linked List	**7**	38.0176	233.4515	470.1785	787.9588
Priority Queue	**16**	25.2861	162.2649	280.1064	343.1944
Binary Tree	**6**	33.2123	193.5589	563.6282	889.3333

Table 13. Average filter time benchmark results

Average Space (kb)					
Data Structure	**Points**	**10**	**100**	**500**	**1000**
Array List	**11**	40,513.7328	75,818.9728	182,478,676.8	292,692.1344
Linked List	**7**	42,039.4032	105,929.6512	191,343.7088	312,230.4720
Priority Queue	**10**	43,328.2416	66,419.1872	163,652.7344	340,897.4816
Binary Tree	**12**	39,508.0752	73,055.5792	217,460.2192	281,270.9936

Table 14. Average filter space benchmark results

4.1.8 Calendar navigation and cache

For these results, only time taken will be accounted for as the main focus for this feature is to reduce the time loading of already visited months. However, for the space taken, as expected, the cache will take more as more space will be used to store the gui node of the already visited months to be reused later. For these specifically, the points system will not be considered, instead the average percentage change is the factor that will be taken account into, which is calculated by the following formula:

$$\frac{v_2 - v_1}{v_1} * 100$$

Where, v_2 is the new value and v_1 is the old value

Average calendar navigation time (ms)				
Data Structure	**10**	**100**	**500**	**1000**
Array List	1.3359	67.6606	108.2207	193.9912
Linked List	1.0882	42.5776	90.6712	180.8556
Priority Queue	1.1422	44.7892	54.9490	182.6191
Binary Tree	0.9097	25.3945	97.7373	176.9935

Table 15. Average calendar navigation time benchmark results

Average calendar navigation time with cache (ms)					
Data Structure	**Average % change**	**10**	**100**	**500**	**1000**
Array List	**80.4264**	0.8902	2.3598	4.7428	7.3476
Linked List	**67.8631**	1.2385	1.2362	7.3675	6.7038
Priority Queue	**77.0219**	0.9189	1.5749	2.9710	4.6373
Binary Tree	**86.2417**	0.3724	1.3246	5.4318	5.8813

Table 15. Average calendar navigation time with cache benchmark results

4.2 Time complexity graphs

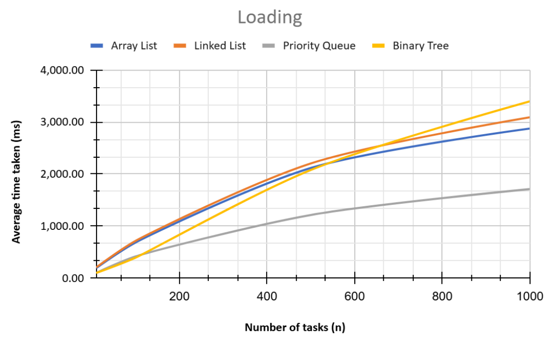

Figure 1. Average loading time result graph

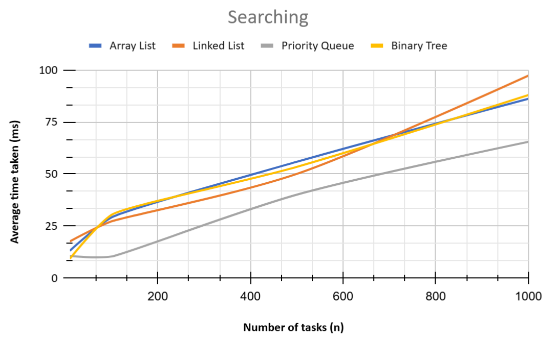

Figure 2. Average searching time result graph

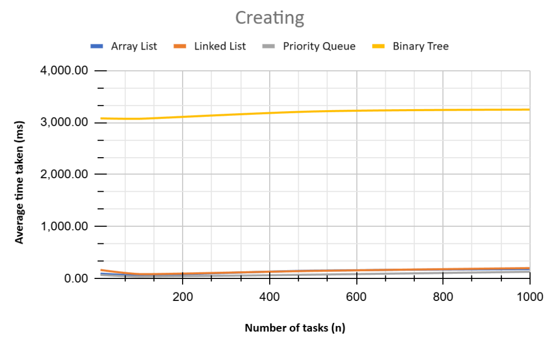

Figure 3. Average creation time result graph

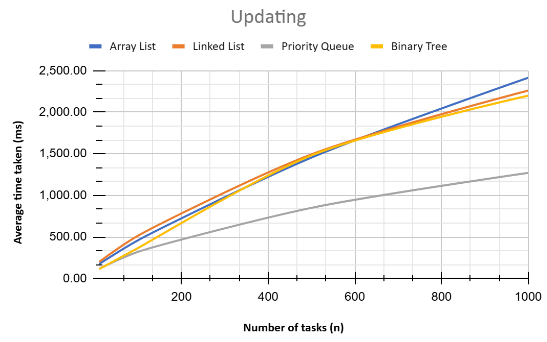

Figure 4. Average updating time result graph

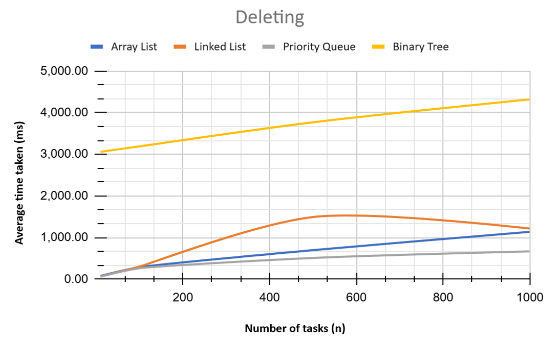

Figure 5. Average deleting time result graph

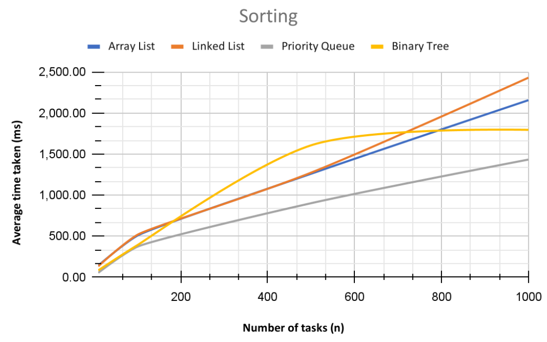

Figure 6. Average sorting time result graph

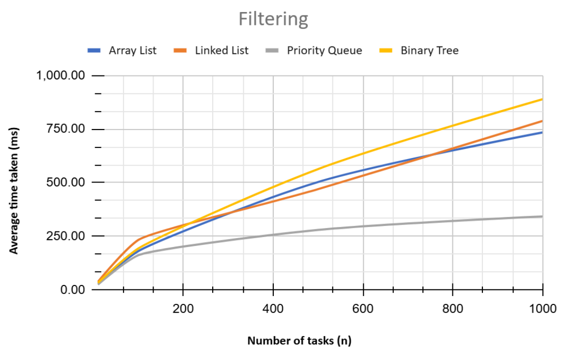

Figure 7. Average filtering time result graph

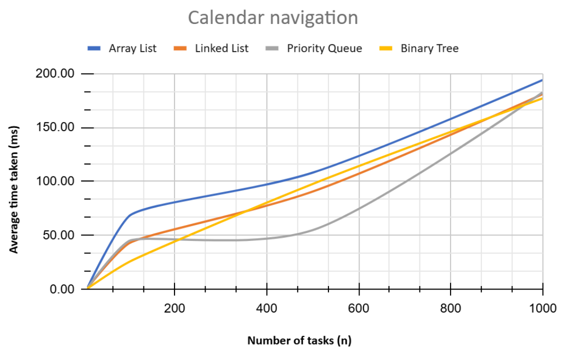

Figure 8. Average calendar navigation time result graph

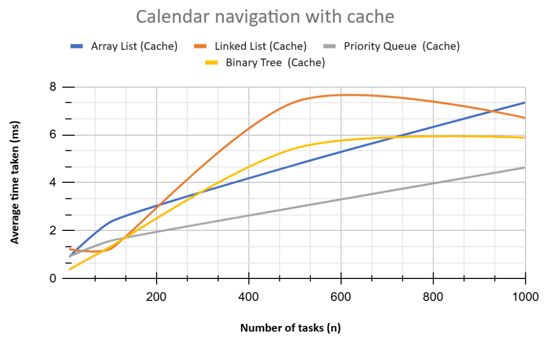

Figure 8. Average calendar navigation time with cache result graph

4.3 Data analysis and discussion

This section will be used to provide a closer look for each of the main calendar operations, comparing each data structure's theoretical time complexity in each of the operations with the actual performance gathered during testing. In addition, the average memory space used will also be discussed.

4.3.1 Loading

Theoretically, the time complexity for loading n tasks from the database should be o(n), however, the graph in figure 1 suggests a time complexity of o(log(n)). This may be due to the fact that for lower amounts of tasks, there is less time for the JVM to warm up, and by the time it warms up, there might be no more tasks to load [16]. This is evident in the supposed o(log(n)) shape of the graph as opposed to the o(n) shape. There might also be more optimization that the JVM uses such as the JIT (Just-In-Time) compiler, overheads, etc [17].

In terms of the result, priority queue has the lowest average amount of time spent to complete the operation, with it having 14 points. This can be due to the fact that priority queue uses another data structure called a heap [14], more specifically, by default, java uses a priority heap [18], which can be used to efficiently retrieve the highest priority item or least and insert according to the items priority. As such, the insertion's time complexity is way faster than an arraylist or linkedlist, which are o(n) [15]. Surprisingly, binary tree (AVL) also performs relatively well compared to array lists and linked lists, however, it falls short on higher input sizes, mostly due to the need of rebalancing or the fact that the dataset used is less optimized for the tree at higher input sizes.

Similarly, for space taken, both array list and priority queue performs relatively well compared to the linked list and binary tree as they only store the required item, and not another pointer to their data structure (another node).

4.3.2 Searching

The result of the search operation mirrors the loading operation, where the performance in terms of speed goes as; priority queue > binary tree > array list > linked list. This can also be pointed to the same reasoning, however, the theoretical time complexity and the resulting graph, figure 3, is more similar, with it being o(n). It is to be noted however, that binary tree's search can be optimized further.

As for the space taken however, both array list and linked list performs way better than priority queue and binary tree. Priority queues and binary trees consume more memory during search operations due to the internal structure (e.g., using binary tree nodes) requiring references for traversal and ordering. In contrast, ArrayList and LinkedList store tasks in a more linear and compact fashion, contributing to lower space usage.

4.3.3 Creating

For creating a new task, priority queue heavily out performs all data structure with it performing the best in all sizes of inputs, this can also be pointed to the fact that priority queue allows for the task to be inserted quickly at the correct position due to the priority heap. More notably, binary trees perform way worse due to the need for rebalancing. As for array list and linked list, they perform similarly, however, array list performs a bit better. The theoretical time complexity for both binary tree and priority queue is o(log(n)), and array list and linked list is o(n) and o(1) respectively [15]. However, it is hard to tell with the graph as binary tree is dominating the graph with its huge average time taken, however, from the shape it alongside priority queue closely resembles an o(log(n)) shape. As for array list and linked the graph appears to be more straight, which it can be inferred to be o(n), maybe even more due to the additional operations needed to create the GUI elements, however, again, it is hard to tell for this graph [15].

The result for the average space taken of the data structure also reflects the result for the average time taken, however, linked list performs worse compared to the performance of it in the average time taken. More notably, binary tree and linked list performed about the same, with them garnering the same amount of points, however, with linked list performing way worse at lower input sizes, and binary tree performing worse at higher input sizes. This result can be the result of JVM having different overhead protocols.

4.3.4 Updating

The algorithm for updating in this application looks through all of the loaded tasks, then if the task matches the old task to be updated, it will delete it, and then it will add the newly edited task. In theory the time complexity for that would be o(n) for looping through all of the task, for removing; priority queue and binary tree is o(log(n)), array list and linked list is o(n), and finally for adding; priority queue and binary tree is o(log(n)), array list is o(n), linked list is o(1), and priority queue and binary tree is o(log(n)) [15]. In total, the time complexity should be o(n). While taking account of all of the optimization that java has, the graph supports this, with it looking like the time complexity of o(log(n)), but less steeper. However, for binary trees specifically, there are some discrepancies, where it performs better here compared to deleting and inserting, even though in the code it is the combination of both of them. This issue can be partially explained by the update method for SQL being less expensive due to the need to not create or remove entire rows; rather, only changing some information in one row. As such, for the discussion, that issue will be ignored. Again, priority queue is among the fastest here due to the removal and addition being o(log(n)) [15]. While, linked list theoretically should have been the fastest, it may fail here due to the time complexity of the access being o(n) [15].

In terms of space, the results are close, with binary tree and priority queue being tied, and the same with array list and linked list. With Array list and linked list winning. The reasons may be the same reason as **4.3.2 Searching**, however to a different degree.

4.3.5 Deleting

The result for deleting mirrors the performance of the creation operation as conceptually, both of them are the same, with the only difference is that the creation is replaced with deletion. As such, the theoretical time complexity is o(n) for both array list and priority queue, again the graph might be different due to the jvm's optimization. Whereby, a theoretical o(n) performs as o(log(n)) [15]. Similar to creating, binary tree and priority queue's performance looks similar to a o(log(n)) graph. Again, binary tree perform way worse here due to the need for rebalancing.

The same also applies to the average space taken, where in creating, both linked list and binary tree tied, in deleting they also tied, the only difference is that in terms of deletion, binary tree only performed averagely worse (not having one excellent performance, but average performance).

4.3.6 Sorting

For sorting, the results here are self explanatory, with the inherently sorted data structures (priority queue and binary tree) performing the best and array list and linked list performing the worst. Both array list and linked list performed about the same with the only point difference being one. However, the difference between priority queue and binary tree is a huge 6 points, with priority queue being the best data structure for the sorting operation. For java, the average time complexity for the sort method is o(nlog(n)), and this is also reflected in the graph, with the graph showing o(n) (except for linked list, where it seems that it is increasing at a faster rate as it approaches 1000 input size), taking in account for the JVM optimization. More notably however, binary trees exhibit a graph resembling o(log(n)), making it the best in terms of the growth relative to the input size.

In terms of memory taken, binary tree performed the best, and priority queue performed the opposite of how it performed in the average time taken test. However, linked list is the worst in this case, in terms of average space and time taken. The result for space can also be reduced as both binary tree and priority queue already having a sorted structure, thus, less space is used to reheap or rearrange the tasks when sorting.

4.3.7 Filtering

For filtering, the theoretical time complexity is o(n) as each of the input needs to be checked if it matches defined by the user, and if it does, it will add it to the filtered data structure. The resulting graphs also support this, taking account of all of java's optimization. In terms of the best data structure, priority queue performed the best, with it having an excellent performance across all of the input sizes. The disparity between the performance of these data structures mainly comes from the insertion. This can be especially seen as the resulting performance perfectly mirrors **4.1.3 Creating**'s result.

However, as the function for filtering is bundled up with sorting, the order for average space taken remains the same with binary tree > array list > priority queue > linked list.

4.4 Cache performance

One of the main concerns brought up earlier was the need for speed especially in calendar operations, and based on prior work, one of the most heavy operations was navigating the calendar as the application would have to redraw the GUI elements contained for each day of the month, with addition of some of the trailing days and leading days, which in total consists of 35 total days to be checked, which could have GUI elements to be drawn over. While this may seem like it is not a significant amount, it has to be noted that this action is likely the most performed action as, firstly, in order to perform this action, the user only has to click one button, and secondly, it is one of the primary way for the user to visualise all of their tasks, in a comprehensive sense.

During the testing, it was found that the implementation of a cache dramatically helps with this, with it being able to bring down the operation that would take around 200 ms down to around 7 ms. That being said, the order of performance between these data structures would be: binary tree (86.2417% change) > array list (80.4264% change) > priority queue (77.0219% change) > linked list (67.8631% change).

4.5 Recurring events

While not benchmarked, recurring events were successfully implemented without any performance overhead, this is due to the use of "lazy loading", which is the technique to only load data only when needed, instead of loading everything at once [19]. In this case, it would be generating recurring tasks only when the month view range falls between the recurring tasks's end date and task's start date. For example, if a task is set to 'repeat x every two days for two weeks', new task instances will be generated every other day starting from the initial task date (or the last generated instance), **only** when it is needed, when a user moves to a new month in the calendar, etc. This also allows for the end date to be not defined, essentially allowing the user to have a task repeat infinitely, giving the user more flexibility.

5. Conclusion

Overall, priority queue seems to perform the best out of all of the data structures used in terms of average time taken to do the operations, while there are instances where other data structures outperforms priority queue in a certain number of tasks, priority queue still outperforms most of them. This is followed by binary tree, then array list and finally linked list. However, notably, binary tree gets out performed in some of the operations such as in creating and deleting due to needing to rebalance the tree. This is mostly negated in other operations where binary tree has an advantage due to the inherently sorted nature of it, similar to priority queue.

In terms of average space used, array list outperforms all other data structures, followed by priority queue, and finally with linked list and binary tree both tied. This can be pointed towards arraylist and priority queue only storing the required data, while priority queue needs to store additional information to maintain the ordering. As for linked list and binary tree it can be due to them needing to store the pointer to other nodes, or in other words another instance of their data structure, due to their recursive node base structure.

6. Implementation

6.1 UML class diagram

Higher resolution image can be found here

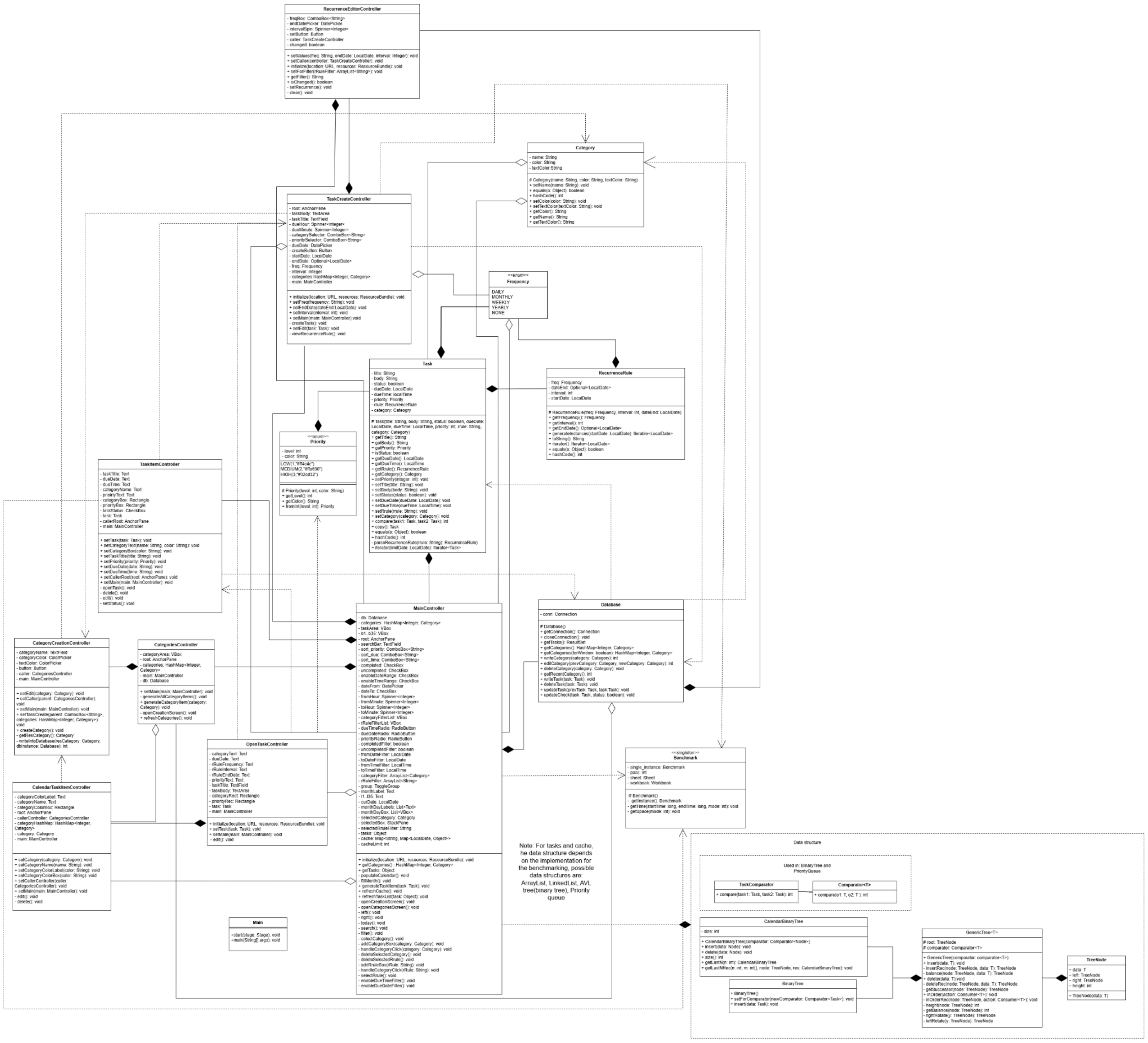

Figure 9. loCalendar UML diagram

6.2 Code implementation

All of the code specific parts can be found on the github repository. The benchmarked operation can be seen by the TODO comments on the main branch. In addition, specific data structures such as Linked list can be found on the branches of the repository.

6.3 Database schema

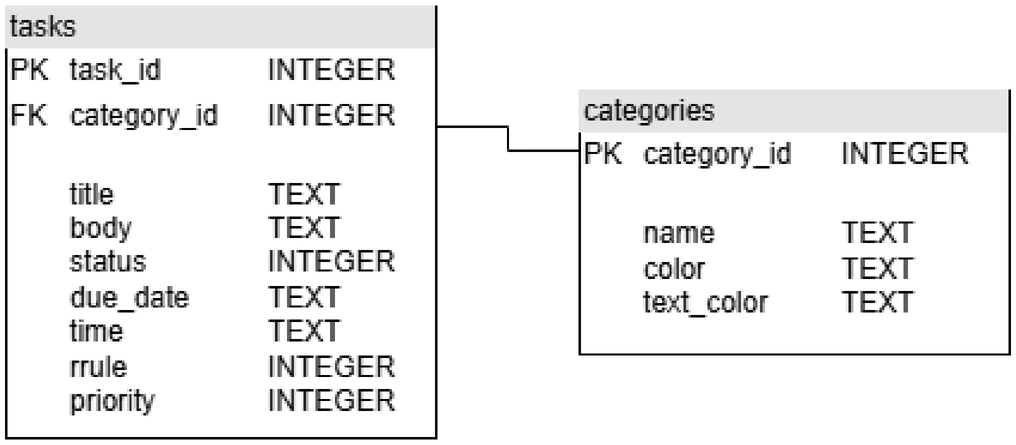

Figure 10. Database schema

Bibliography

[1] Manas Tungare, M. A. Pérez-Quiñones, and A. Sams, "An Exploratory Study of Calendar Use," arXiv (Cornell University), Jan. 2008, doi: https://doi.org/10.48550/arxiv.0809.3447.

[2] G. N. Tonietto and S. A. Malkoc, "The Calendar Mindset: Scheduling Takes the Fun Out and Puts the Work In," Journal of Marketing Research, vol. 53, no. 6, pp. 922–936, Dec. 2016, doi: https://doi.org/10.1509/jmr.14.0591.

[3] F. Duarte, "24+ Digital Calendar and Time Management App Stats (2024)," Exploding Topics, May 13, 2024. https://explodingtopics.com/blog/digital-calendar-market-summary

[4] K. Chung, J. Sang, and V. Rego, "A performance comparison of event calendar algorithms: An empirical approach," Software: Practice and Experience, vol. 23, no. 10, pp. 1107–1138, Oct. 1993, doi: https://doi.org/10.1002/spe.4380231005.

[5] M. Wimmer, D. Cederman, F. Versaci, T. J. Larsson, and P. Tsigas, "Data Structures for Task-based Priority Scheduling," arXiv (Cornell University), Jan. 2013, doi: https://doi.org/10.48550/arxiv.1312.2501.

[6] N. S. Oh and N. J. Ahn, "Dynamic calendar queue," pp. 20–25, Jan. 2003, doi: https://doi.org/10.1109/simsym.1999.766449.

[7] R. M. Brown, "Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem," *Communications of The ACM*, vol. 31, no. 10, pp. 1220–1227, Oct. 1988, doi: https://doi.org/10.1145/63039.63045.

[8] M. Fowler, "Recurring Events for Calendars." Accessed: Mar. 25, 2025. [Online]. Available: https://martinfowler.dev.org.tw/apsupp/recurring.pdf.

[9] loribean, "Recurring Calendar Events — Database Design. - loribean - Medium," Medium, Feb. 2024. https://medium.com/%40aureliadotlim/recurring-calendar-events-database-design-dc872fb4f2b5.

[10] MatthiasScherrerFGCZ, "Timeline: Performance issues with large datasets · Issue #5697 · primefaces/primefaces," GitHub, 2024. https://github.com/primefaces/primefaces/issues/5697.

[11] Sonia, A. Alsharef, P. Jain, M. Arora, Syed Rameem Zahra, and K. Dua, "Cache Memory: An Analysis on Performance Issues," International Conference on Computing for Sustainable Global Development, Mar. 2021, doi: https://doi.org/10.1109/indiacom51348.2021.00033.

[12] B. Desruisseaux, Ed., "Internet Calendaring and Scheduling Core Object Specification (iCalendar)," Sep. 2009, doi: https://doi.org/10.17487/rfc5545.

[13] V. A. Carriero, A. Gangemi, A. G. Nuzzolese, and V. Presutti, "Chapter 10. An Ontology Design Pattern for Representing Recurrent Situations," Studies on the semantic web, May 2021, doi: https://doi.org/10.3233/ssw210013.

[14] A. Levitin, Introduction to the Design & Analysis of Algorithms. Addison Wesley, 2003.

[15] geeksforgeeks, "Time complexities of different data structures," GeeksforGeeks, Dec. 29, 2020. https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/

[16] Don’t Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems, Papers presented at the 2005 workshop on Wireless traffic measurements and modeling. Berkeley, Ca Usenix Association, 2005. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[17] geeksforgeeks, “Just In Time Compiler,” GeeksforGeeks, Oct. 14, 2018. https://www.geeksforgeeks.org/just-in-time-compiler/

[18] Oracle, “PriorityQueue (Java Platform SE 8),” docs.oracle.com. https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

[19] N. S. Filho, “Lazy Loading in .NET: Best Practices and Precautions for Performance Optimization,” vol. 1, no. 15, pp. 1–4, Oct. 2024, doi: https://doi.org/10.5281/zenodo.13942844.

Appendix

1. **Program Manual**
    a. Make sure to download Java 17, and maven
    b. Choose one of the data structures from the branch on Github: https://github.com/osten-antonio/loCalendar
    c. Download the repository by clicking on the green button labeled “Code” and then selecting “Download ZIP”.
    d. Extract the contents of the ZIP File
    e. Download the maven dependencies
    f. Run the program through main.java

2. **Github Repository**

    Github Repository Link

3. **Presentation Link**

    LoCalendar Final Project Presentation