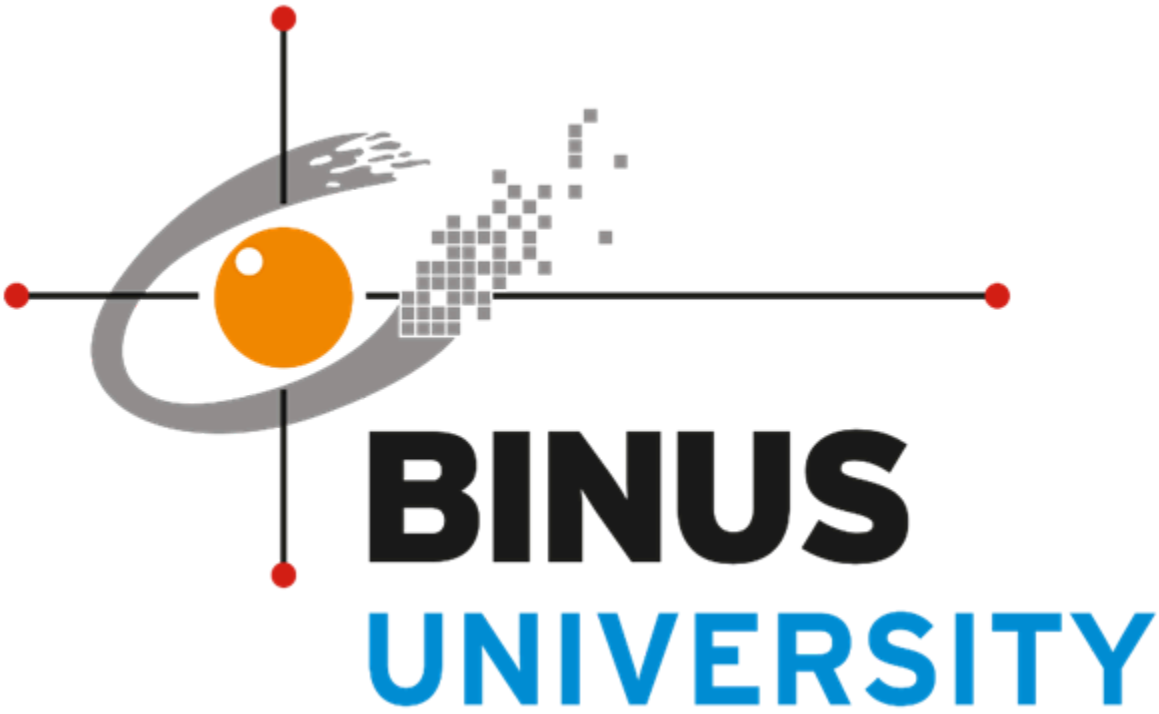
Final Project

Object Oriented Programming & Data Structures



Project name: “Enhanced Budgeting App”

Student names: Agie Winata (2902642726), Darrus, William Ekapanna Rusmana (2802523111)

Class L2CC

Table of contents

1. Background

In today's digital age, the proper and effective management of data has become essential in many applications, especially in fields regarding finance management. There is an increasing need for individuals and organizations to properly keep track of their expenses efficiently. A clear solution to this problem is using technology to keep track of any and all financial records. Expense management systems allow users to store, view, update, and analyze their spending habits more effectively, allowing for better personal and businesses financial decisions

As the amount of financial data grows, so does the complexity of managing these records. Traditional manual bookkeeping is difficult to navigate, as well as being time consuming, businesses that choose to embrace cloud‑based accounting report processing time reductions of around 60 % and higher data accuracy[1]. Similarly, mobile finance apps that integrate real‑time tracking, notifications, and AI‑driven insights significantly enhance transparency and decision‑making[2].

Automating expense monitoring and categorization has proven tangible benefits: one study found users reduced unnecessary expenditures by approximately 20 % within three months of using budgeting tools, and 60 % fewer late payments occurred when bills were automated [3]. Features like instant notifications and intuitive visual dashboards boost user engagement and adherence to financial plans [4].

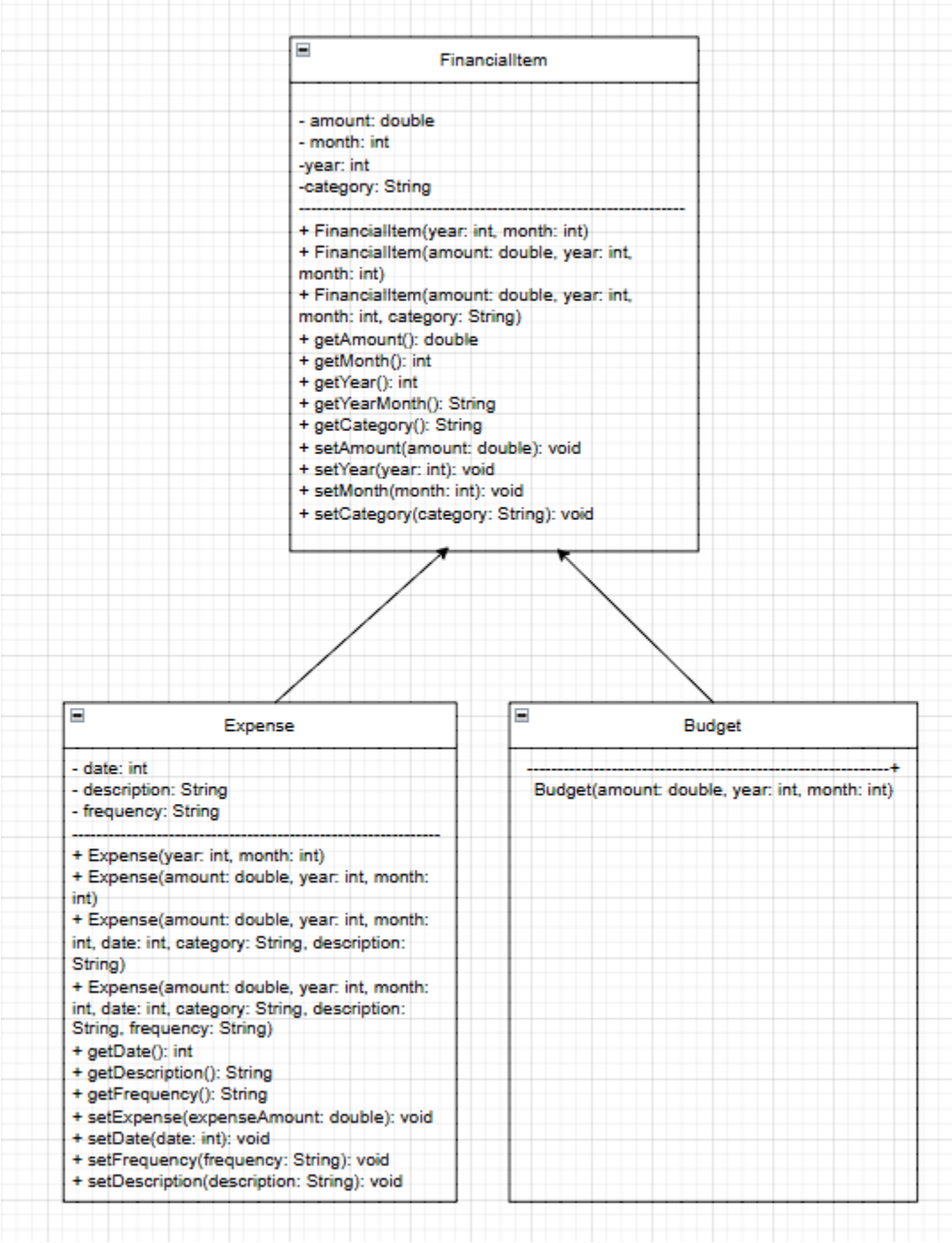
Security, scalability, and regulatory compliance are fundamental for finance apps. Attendance with robust encryption, multi‑factor authentication, and adherence to frameworks like GDPR ensures user trust and data integrity [5]. Similarly, solutions designed to scale—capable of handling transaction surges without performance degradation—support both individual users and businesses [6]. The adoption of AI and automation (e.g. robotic process automation, anomaly detection) further accelerates processing, reduces human workload, and improves accuracy [7].

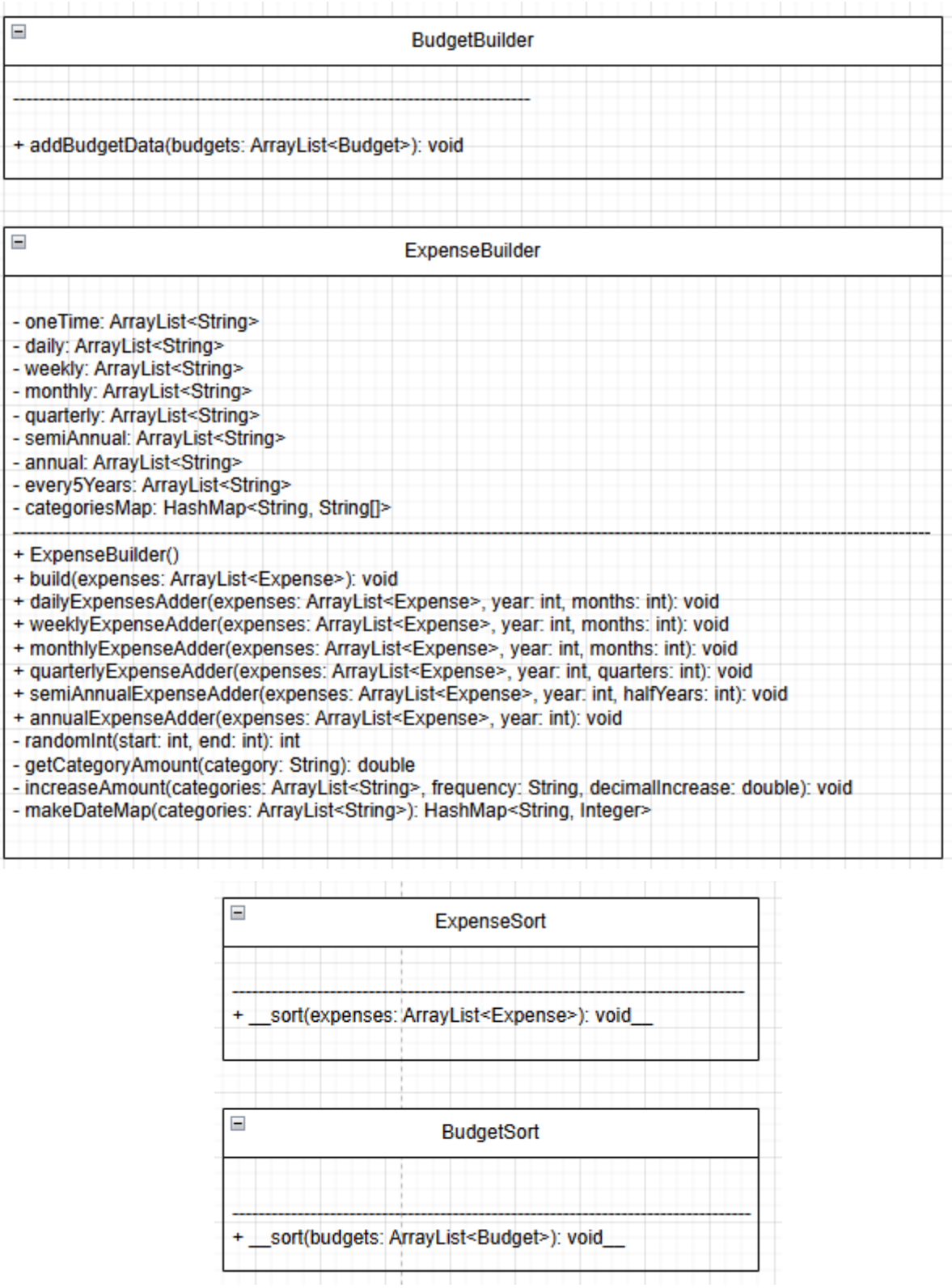
Research consistently shows that digital finance applications enhance financial wellbeing. Users of such tools report higher satisfaction, improved money management habits, and better engagement with their finances [8].

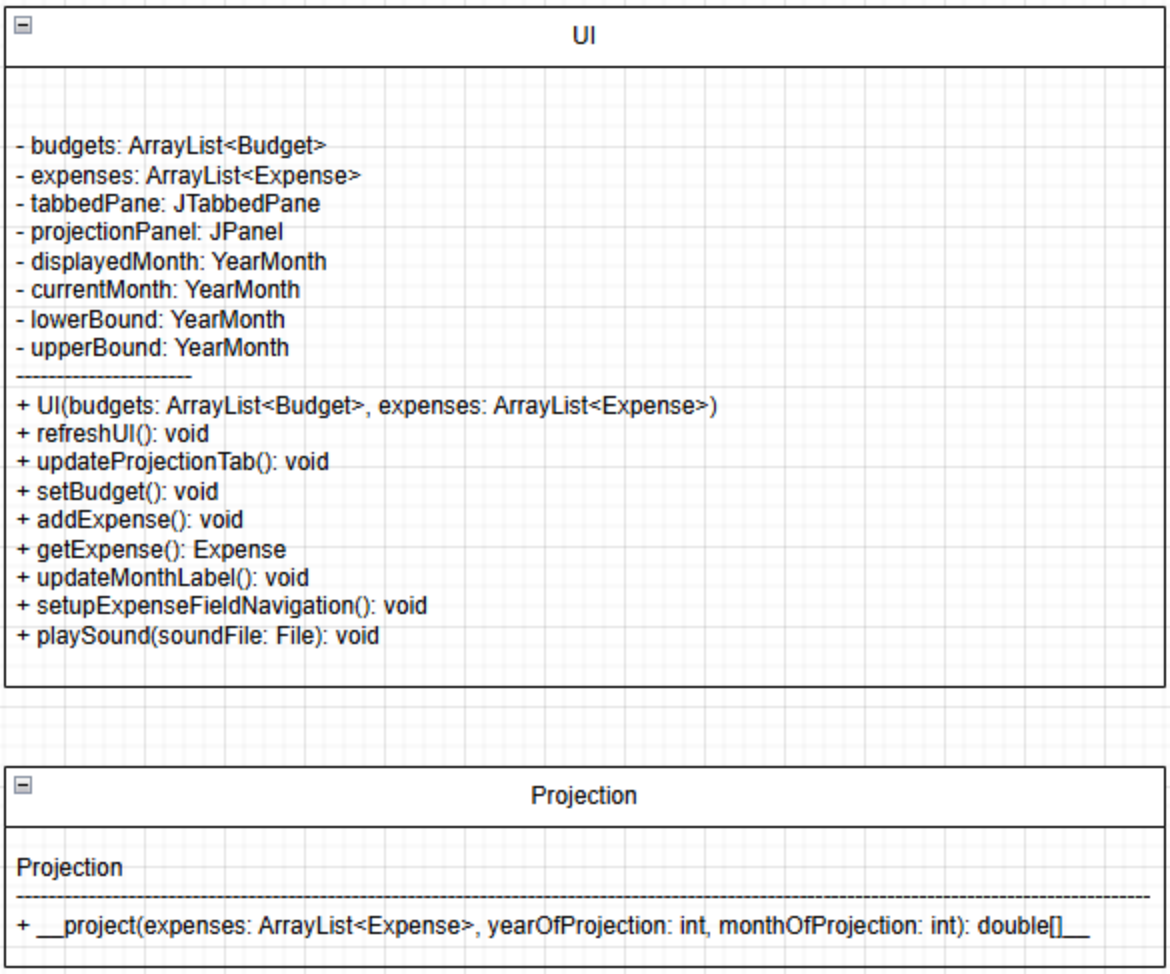
Technology has revolutionized financial record-keeping by making it more efficient, accurate, secure, and engaging. These systems not only automate tedious tasks but actively support better financial choices and outcomes. As financial data continues to expand, integrating intelligent expense management tools becomes increasingly essential for personal and business success.

2. System Design

2.1 UML design




BudgetBuilder
+ addBudgetData(budgets: ArrayList<Budget>): void
ExpenseBuilder
- oneTime: ArrayList<String>
- daily: ArrayList<String>
- weekly: ArrayList<String>
- monthly: ArrayList<String>
- quarterly: ArrayList<String>
- semiAnnual: ArrayList<String>
- annual: ArrayList<String>
- every5Years: ArrayList<String>
- categoriesMap: HashMap<String, String[]>
+ ExpenseBuilder()
+ build(expenses: ArrayList<Expense>): void
+ dailyExpensesAdder(expenses: ArrayList<Expense>, year: int, months: int): void
+ weeklyExpenseAdder(expenses: ArrayList<Expense>, year: int, months: int): void
+ monthlyExpenseAdder(expenses: ArrayList<Expense>, year: int, months: int): void
+ quarterlyExpenseAdder(expenses: ArrayList<Expense>, year: int, quarters: int): void
+ semiAnnualExpenseAdder(expenses: ArrayList<Expense>, year: int, halfYears: int): void
+ annualExpenseAdder(expenses: ArrayList<Expense>, year: int): void
- randomInt(start: int, end: int): int
- getCategoryAmount(category: String): double
- increaseAmount(categories: ArrayList<String>, frequency: String, decimalIncrease: double): void
- makeDateMap(categories: ArrayList<String>): HashMap<String, Integer>
ExpenseSort
+ __sort(expenses: ArrayList<Expense>): void__
BudgetSort
+ __sort(budgets: ArrayList<Budget>): void__



3. Problem Description

As stated in the background, the use of budgeting and expense-tracking applications has significantly increased, especially with the growing emphasis on personal finance management in the digital era. With more users depending on these applications to track daily, monthly, and yearly expenses, these systems are expected to process large volumes of expense data efficiently without compromising speed or memory usage. However, managing a constantly growing list of expenses—including adding, updating, deleting, and searching transactions—can result in performance bottlenecks and higher memory consumption if the underlying data structure is not optimal.

To address this problem, this project will implement and evaluate four different data structures: ArrayList, LinkedList, HashMap, and TreeMap. Each of these structures offers different advantages in terms of access time, memory usage, and manipulation efficiency. The goal is to determine which data structure is the most effective for handling expense data in a budgeting application.

Specifically, the project will measure the runtime and memory usage for the following operations:

1. Generating a list of expenses (simulating user input of financial transactions)
2. Adding an expense to the list
3. Deleting an expense from the list
4. Updating an expense's details (amount, description, etc.)
5. Searching for an expense based on its amount
6. Viewing all expenses (displaying the full list of expenses)
7. Sorting expenses (e.g., by amount or date)

After performing these operations using all selected data structures, the average runtime speed and memory consumption will be calculated. The data structure with the best balance of speed and low memory usage will be considered the most suitable for implementing in an efficient and scalable expense management system.

4. Proposed Solution

4.1 AVL Tree

generateExpense

-The method first checks whether the expenses LinkedList contains any items.

-If the list is empty:

-It prints the message:

"No expenses to sort."

This lets the user know there are no expenses available for sorting.

-The method then immediately exits without performing any further action.

addExpense

-A new Expense object is created using UtilityMethod.initialiseAddedExpense().

-The expense's amount is set to the amount specified by the user.

-The expense is then inserted into the AVL Tree using insertRec(), which maintains the AVL balance.

C. removeExpense

-Prints a message explaining that AVL Trees do not use indexes like lists or arrays and that expenses are removed based on their amount value.

-Deletes the expense with the specified amount (amountToRemove) from the AVL Tree using the recursive deleteRec method.

-Prints a message confirming the removal of the expense with the specified amount.

D. updateExpense

-Prints a message explaining that AVL Trees do not use indexes like arrays or lists.

Therefore, any index or position passed to this method will be ignored.

-Only the amountToUpdate value will be used to locate and update an existing expense.

-Calls the UpdateExpenseRec method to recursively search for the expense with the specified amount and perform the update operation.

E. searchExpense

-Displays a message showing which expense amount is being searched.

-Searches the AVL Tree for a node containing an expense with the specified amount using the recursive searchRec method.

-If an expense with the specified amount is found, it prints details of the found expense

-If no such expense is found:

-Prints: "No expense found with amount: [amount]".

F. viewExpenses

-Checks if the AVL Tree is empty by verifying if the root is null.

-If it is empty, prints: "The AVL Tree is empty. No expenses to display." and exits.

-If the tree is not empty:

-Prints a message: "Viewing expenses in AVL Tree (max 50 entries):".

-Performs an in-order traversal of the tree to display the expenses in sorted order.

-Limits the output to a maximum of 50 expenses.

-If more than 50 expenses exist, prints a note indicating that additional expenses are present but not shown.

G. sortExpenses

-Searches the tree for an expense with the given amount.

-If found, replaces the entire expense with new details provided by UtilityMethod.initialiseUpdatedExpense().

-The updated expense's count is reset to 1.

-If no such expense exists, the user is informed.

-AVL Tree properties remain unaffected because the amount (which determines tree structure) remains unchanged.

1. ArrayList

   A. generateExpense

   -Generates a specified number of random expenses (numValues) and adds them to the ArrayList.

   -Each expense is created using the utility method initialiseExpense().

-This method keeps adding expenses until the required number is reached.

B. addExpense

-Checks if the index is within the valid range (between 0 and the size of the ArrayList).

-If invalid, prints: "Invalid index. Please enter a valid index." and exits the method.

-If the index is valid:

-Creates a new random expense using the utility.initialiseExpense() method.

-Adds the new expense at the specified index in the ArrayList.

-Prints a message showing the index and details of the added expense.

C. removeExpense

-Checks if the index is within the valid range (between 0 and the size of the ArrayList minus one).

-If the index is invalid, prints: "Invalid index. Please enter a valid index." and exits the method.

-If the index is valid:

-Removes the expense at the specified index from the ArrayList.

-Prints the details of the removed expense, including the index and the expense information.

D. updateExpense

-Checks if the index is within the valid range (between 0 and list size minus one).

-If the index is invalid, prints an error message and exits.

-If valid:

-Creates a new expense using the utility method initialiseExpense().

-Replaces the expense at the specified index with this newly created expense.

-Prints the details of the updated expense.

E. searchExpense

-Checks if the index is within the valid range (between 0 and list size minus one).

-If the index is invalid, prints an error message and exits.

-If valid:

-Retrieves the expense at the specified index.

-Prints the details of the found expense.

F. viewExpenses

-Checks if the ArrayList is empty.

-If empty, prints a message indicating that there are no expenses to display.

-If not empty:

-Prints the details of up to 50 expenses in the ArrayList.

-For each expense printed: Displays the index, amount, year, month, day, description, and frequency.

G. sortExpenses

-Checks if the ArrayList is empty.

-If empty, prints a message indicating there are no expenses to sort.

-If not empty:

-Sorts the expenses in ascending order based on their amount using Java's built-in Collections.sort() method with a comparator that compares the amount field of each expense.

-Prints a message stating that the expenses have been sorted by amount.

2. HashMap

A. generateExpense

-A UtilityMethod object is created to generate random expenses.

-A loop runs numValues times:

-A random Expense object is generated.

-The generated expense is stored in the HashMap with key i.

B. addExpense

-A UtilityMethod object is created to generate an expense to add.

-A new expense is initialized using initialiseAddedExpense().

-Checks if the specified index does not exist in the HashMap:

-If not present, adds the expense directly at that index and prints a confirmation.

-If the index already exists:

-Shifts all existing expenses (from the last index down to the given index) one position to the right to make space.

-Inserts the new expense at the specified index.

C. removeExpense

-Checks if the specified index exists in the HashMap:

-If exists, removes the expense at that index and prints a confirmation.

-If does not exist, prints a message saying no expense was found at that index.

D. updateExpense

-Checks if the specified index exists in the HashMap:

-If exists, updates the expense at that index with a new Expense object and prints the updated details.

-If does not exist, prints a message saying no expense was found at that index.

E. searchExpense

-Checks if the specified index exists in the HashMap:

-If exists, retrieves and prints the Expense object at that index.

-If does not exist, prints a message saying no expense was found at that index.

F. viewExpenses

-Checks if the HashMap is empty:

-If empty, prints a message saying no expenses to display.

-If not empty, iterates through the entries in the HashMap and prints each expense's details.

-Limits the display to 50 expenses maximum.

G. sortExpenses

-Converts the HashMap into a List of entries to allow sorting.

-Sorts the list by each expense's amount in ascending order.

-Prints the details of each expense from the sorted list.

3. LinkedList

A. generateExpense

-Generates and adds a specified number of random expenses into the LinkedList.

-A loop runs from 0 to the given number (numValues).

-In each loop iteration:

-A new random expense is created using the utility method.

-The expense is added to the LinkedList.

-After adding all expenses, the method finishes.

B. addExpense

-Checks if the given index is valid (between 0 and LinkedList size).

-If invalid, prints an error message and exits.

-If valid:

-Initializes a new expense using the utility method.

-Adds the new expense at the specified index in the LinkedList.

-Prints the details of the added expense.

C. removeExpense

-Checks if the given index is valid (between 0 and LinkedList size minus one).

-If invalid, prints an error message and exits.

-If valid:

-Removes the expense at the specified index from the LinkedList.

-Prints the details of the removed expense.

D. updateExpense

-Checks if the given index is valid (0 to size - 1).

-If invalid, prints an error message and exits.

-If valid:

-Creates a new random expense using the utility method.

-Replaces the expense at the specified index with this new expense.

-Prints the updated expense details.

E. searchExpense

-Verifies if the given index is within valid range (0 to size - 1).

-If invalid, prints an error message and exits.

-If valid:

-Retrieves the expense at the specified index from the LinkedList.

-Prints the details of the found expense.

F. viewExpenses

-Checks if the LinkedList is empty:

-If yes, prints "No expenses to display." and exits.

-If not empty:

-Prints "Expenses in LinkedList:".

-Iterates through the LinkedList and prints each expense.

-Limits the output to the first 50 expenses to avoid excessive printing.

-Measures and prints the time and memory used for the operation.

G. sortExpenses

-Checks if the LinkedList is empty:

-If yes, prints "No expenses to sort." and exits.

-If not empty:

-Uses Collections.sort() along with a comparator to sort the expenses based on their amount in ascending order.

-Prints "Expenses sorted by amount." after sorting.

-Measures and prints the time and memory used for the sorting operation.

4. Stack

A. generateExpense

-Measures memory usage and execution time before starting the operation.

-Loops numValues times to generate expenses:

-Calls utility.initialiseExpense() to create a new random Expense object.

-Pushes the generated Expense onto the top of the expenseStack.

-Measures memory usage and execution time after the operation.

-Prints the memory consumed and time taken for generating the expenses.

B. addExpense

-Measures memory usage and execution time before starting the operation.

-Checks if the provided index (indexToAdd) is valid:

-If the index is less than 0 or greater than the size of the stack, prints "Invalid index. Please enter a valid index." and exits the method.

-If the index is valid:

-Creates a temporary stack (tempStack) to assist in the insertion process.

-Pops elements from expenseStack and pushes them into tempStack until reaching the specified indexToAdd position.

-Generates a new Expense using utility.initialiseExpense() and pushes it onto the main stack (expenseStack) at the correct position.

-Moves all elements back from tempStack to expenseStack, restoring the original order with the new element inserted.

-Prints the message: "Expense added at index X" along with the details of the newly added expense.

-Measures memory usage and execution time after the operation.

-Prints the memory consumed and time taken for the addition operation.

C. removeExpense

-Measures memory usage and execution time before starting the operation.

-Checks if the provided index (indexToRemove) is valid:

-If the index is less than 0 or greater than or equal to the stack size, prints "Invalid index. Please enter a valid index." and exits the method.

-If the index is valid:

-Creates a temporary stack (tempStack) to assist in the removal process.

-Pops elements from expenseStack and pushes them into tempStack until reaching the element at the specified indexToRemove.

-Pops (removes) the targeted Expense from expenseStack.

-Prints the message: "Expense removed at index X" along with the details of the removed expense.

-Moves all elements back from tempStack to expenseStack, restoring the original order minus the removed element.

-Measures memory usage and execution time after the operation.

-Prints the memory consumed and time taken for the removal operation.

D. updateExpense

-Checks if the index is invalid (less than 0 or greater than/equal to stack size):

-If yes, prints "Invalid index. Please enter a valid index." and exits.

-If valid:

-Pops elements into a temporary stack until the target index is reached.

-Removes the old expense and pushes a newly initialized expense.

-Restores the original stack by pushing back elements from the temporary stack.

-Prints the updated expense details.

-Measures and prints the time and memory used for the operation.

E. searchExpense

-Checks if the index is invalid (less than 0 or greater than/equal to stack size):

-If yes, prints "Invalid index. Please enter a valid index." and exits.

-If valid:

-Pops elements into a temporary stack until reaching the target index.

-Peeks at the top element to get the desired expense.

-Prints the found expense details.

-Restores the original stack by pushing back elements from the temporary stack.

-Measures and prints the time and memory used for the operation.

F. viewExpenses

-Checks if the stack is empty:

-If yes, prints "No expenses to display." and exits.

-If not empty:

-Pops each element to a temporary stack while printing its details (index, amount, year, month, day, description, frequency).

-Restores the original stack by pushing elements back from the temporary stack.

-Measures and prints the time and memory used for the operation.

G. sortExpenses

-Checks if the stack is empty:

-If yes, prints "No expenses to sort." and exits.

-If not empty:

-Copies stack elements into a temporary list.

-Sorts the list by expense amount using a comparator.

-Clears the original stack and pushes the sorted elements back.

-Prints "Expenses sorted by amount."

-Measures and prints the time and memory used for the sorting operation.

5. Results

Data Structures Runtime Table (Nanoseconds) (Input Size = 1,000)						
Data Structures	**Data Generation**	**Insertion**	**Deletion**	**Searching**	**Updating**	**Sorting**
ArrayList	416,000	49,047	20,230	27,667	48,167	1,230,400
LinkedList	583,300	59,973	21,400	12,420	47,023	1,559,800
HashMap	712,600	485,769	33,500s	39,567	59,967	2,729,400
Stack	607,500	743,200	419,500	351,772	368,700	1,621,800.00

AVL Tree	700,200	95,300	48,900	758,700	99,067	(Already Sorted)

Data Structures Memory Usage Table (Bytes) (Input Size = 1000)

Data Structures	Data Generation	Insertion	Deletion	Searching	Updating	Sorting
ArrayList	74,408	2,882	0	0	2,675	454,760
LinkedList	123,408	4,402	0	0	2,672	12,128
HashMap	128,458	19,810	1,760	1,760	1,760	466,792
Stack	71,928	9,792	6,243	69,490	6,861	465,328
AVL Tree	74,520	2,648	2,648	2,672	9,648	(Already Sorted)

Data Structures Runtime Table (Nanoseconds) (Input Size = 10,000)

Data Structures	Data Generation	Insertion	Deletion	Searching	Updating	Sorting
ArrayList	1,689,600	68,300	41, 663	21,667	43,367	10,422,300
LinkedList	2,175,400	92,400	95,400	96,400	80,000	19,882,200
HashMap	2,445,500	3,753,500.	66,700	43, 867	82,657	14,862,600
Stack	3,583,400	3,868,000	3,987,971	1,670,756	1,513,876	19,451,600
AVL Tree	1,978,800	145,333	102.928	699,067	161,467	(Already Sorted)

Data Structures Memory Usage Table (Bytes) (Input Size = 10,000)

Data Structures	Data Generation	Insertion	Deletion	Searching	Updating	Sorting
ArrayList	673,672	2,933	0	0	4,075	6,291,456
LinkedList	797,480	9,334	0	0	3,291	5,782,656
HashMap	1,145,893	196,430	1,760	1,760	1,760	7,098,288
Stack	521,512	47,426	46,243	42, 821	42,861	5,782,656
AVL Tree	587,440	2,648	10, 342	2,672	2,648	(Already Sorted)

Data Structures Runtime Table (Nanoseconds) (Input Size = 100,000)

Data Structures	Data Generation	Insertion	Deletion	Searching	Updating	Sorting
ArrayList	3,482,400	256,955	63,600	26,800	42,367	82,904,400
LinkedList	4,800,000	329,033	179,200	338,533	372,200	69,061,000
HashMap	7,404,000	11,491,833	45,367	46,833	81,657	56,781,600
Stack	5,591,700	10,747,652	5,987,971	8,599,333	6,513,876	60,725,032
AVL Tree	19,812,900	77,400	90.928	571,533	90,467	(Already Sorted)

Data Structures Memory Usage Table (Bytes) (Input Size = 100,000)

Data Structures	Data Generation	Insertion	Deletion	Searching	Updating	Sorting
ArrayList	4,830,288	256,955	0	0	504,075	103,604,400
LinkedList	7,402,912	269,800	0	0	503,291	74,061,600
HashMap	9,745,984	1,778,805	391,80	591,216	452,955	62,781,600
Stack	4,890,144	836,426	1,046,243	1,111,867	962,831	61,868,032
AVL Tree	4,591,700	269,648	494,584	893,400	679,440	(Already Sorted)

The tables above contain the runtime and memory usage of different data structures on executing the operations in the first row. The data is gathered through executing these operations on different parts of the data structure -beginning, middle, and end for three times and taking their measurements. Afterward, using all the data gathered, we find their averages using the mean. Once found, all data would be inputted into the table, and the lowest figure for each of the operations is highlighted. All content displayed within the table is measured in Nanoseconds for runtime and Bytes for memory usage. The subsequent section will then explain the underlying causes for the results displayed.

5.1. ArrayList

From the table above, we can see that ArrayList is commonly the fastest when it comes to executing the operations tested. This superior performance could be due to how the ArrayList is implemented. For example, as observed from the data, ArrayList is commonly the fastest at executing, searching, and updating data, which could be attributed to how it handles memory. All data inside the arraylist is stored consecutively in one block and indexed, thus the program does not need to traverse the whole list to find the specified elements, and can just access it through its

index (Kumar, 2024). This gives ArrayList a time complexity of O(1), which is in line with our results.

However, it is worth noting that it is very difficult to accurately measure the runtime and memory usage of programs; hence, it is entirely possible to get data that does not reflect the theoretical time and space complexities of the data structure. One example of such is the runtime of insertion and deletion for ArrayLists in the table. Theoretically speaking, ArrayList has a runtime of O(N) for deletion and insertion (GeeksforGeeks, 2016), yet our data suggests that it has a runtime of O(1). We do not have a definite, concrete reason for such a phenomenon; however, we speculate that this occurs due to the JVM's dead code elimination. Code that is not used and has no real impact on the program is susceptible to being optimised by the JVM through dead code elimination, where some part of the operation is skipped (GeeksforGeeks, 2023). In our case, since we do not use the result from inserting and deleting the elements, the JVM could skip some internal operations of the add and remove methods, resulting in a time complexity of O(1). Apart from that, ArrayList also measures 0 bytes for memory usage when measuring searching and deletion, which does not reflect the expected space complexities of ArrayList. We theorised that the dead code deletion caused this for our deletion method. Furthermore, since no major operation is done in the searching method, the change in memory before and after could be so insignificant that it is not registered in the program.

Aside from that, all performance metrics recorded for ArrayList are in line with its theoretical time complexities. Sorting in an arraylist takes a lot of time and memory, since all elements would have to be inserted at different positions, the arraylist would need to move all of the objects inside and reconnect them with one another, all of which would take a long time and a lot of memory.

5.2. LinkedList

The result from the table above neatly mirrored the theoretical space and time complexities of LinkedList. For example, from the table above, we can see that LinkedList uses, on average uses more memory than its ArrayList counterpart. This is due to the fact that LinkedList, unlike ArrayList, stores data in nodes, where each node points to a different node (GeeksforGeeks, 2024). This means that, upon generation, the LinkedList would have to create

nodes and pointers to each node next to it, all of which use a ton of memory. Furthermore, due to the lack of indexes, the inserting, deleting, searching, and updating data would take O(n) time, as the program would need to go through the list one node at a time, until the correct position is found.

In contrast, there are times when our data suggests that LinkedList is faster than ArrayList at searching and updating data. All of which does not reflect the theoretical O(n) time complexity of LinkedList, and also discredits the O(1) time complexity of ArrayList (GeeksforGeeks, 2024). This discrepancy, however, can be explained by the fact that operations done in front and at the back of a LinkedList have an O(1) time complexity, since a LinkedList directly points to the most front and furthest back nodes in its list (GeeksforGeeks, 2024). This means that any operation done on these nodes would yield a time complexity of O(1) as the program does not need to traverse the entirety of the LinkedList. All of this entails that, in our tests for input size 1000, the LinkedList constantly performs its operation, on its first and last node, at a blazing speed. Which, when taking into account for the mean of each operation, produces a slightly lower figure than its ArrayList counterparts.

Aside from that, another anomaly can be seen within the graph, which is the fact that the memory usage for deleting and searching data is zero bytes. We also hypothesize that this phenomenon happens due to the same reason as the ArrayList, where the JVM's dead code deletion property might've skipped some steps for the deletion method, and the searching method does not produce enough noticeable memory changes.

5.3. HashMap

From the table above, we can conclude that the data almost accurately represents the theoretical time complexities of a hash map. We can see that operations such as deletion, searching, and updating yielded a constant O(1) time complexity across all input sizes, perfectly demonstrating the main quality of a hashmap. This occurrence could be attributed to how a

hashmap is implemented, as when you implement a hashmap, the program would immediately associate the value that you put, with the corresponding key that you put alongside the value. This means that inserting (DINESH, 2024), deleting, searching, and updating data would supposedly have a time complexity of O(1), since the program could just directly access the value through keys, and does not need to traverse the entire data structure.

However, there is a discrepancy between our data and theory when we measure the runtime for the data insertion of a hashmap. Our table's data suggests that the time complexity for data insertion in our hashmap is O(N). This occurrence is due to the nature of our programming for our method, which adds expenses. Normally, when a collision occurs in a hash map, one could use linear probing to solve the collision. Where the program would look for an empty space for the expense to be added to. However, the method used to add an object into the hashmap is programmed in such a way that it would increment every key larger than the soon-to-be added object's key by one. Afterward, the soon-to-be added object is then placed in that empty slot, all of which turns the supposed O(1) time complexity into O(N). This demonstrates that a hashmap is not suitable for keeping track order of insertion for data.

Other than that, the other data clearly reflects the theoretical time complexity of a hash map. For example, the high memory usage for the hashmap reflects its O(N) space complexity (GeeksforGeeks, 2017), since the hashmap would need to make more buckets and resize itself to accommodate the increasing amount of data. furthermore

5.4. Stack

The data above does not paint a clear picture of the properties of stacks. Runtime for operations that align with the stack's Last-In, First-Out (LIFO), such as generating data, scales linearly with input sizes, giving it a time complexity of O(N). This is due to the fact that, though Stack's push and pop methods have a time complexity of O(1) (GeeksforGeeks, 2022), executing it N amount of times scales it to O(N). The same pattern could also be observed for its space complexity.

However, an even bigger flaw is exposed when a stack is utilised for an operation that does not follow its LIFO principle. When executing inserting, deleting, searching, and updating data, stacks continue to exhibit an O(N) time complexity. This could be attributed to the fact that in order for each operation to be done at the correct index, the program would need to remove every element after the specified index into a new stack. Not only that, after the operation is performed, the program would then need to re-add the moved element into the original stack. All of this caused stacks to be very slow and required a lot of memory to execute operations that do not follow its LIFO principle. Furthermore, sorting requires even more space and time to execute, as the data inside the stack would need to be copied into a list, which would then be sorted. Afterward, the content of the list would be copied back into the stack. All of this makes stacks very inefficient and unsuitable to be use in this project.

5.5. AVL Tree

The data above correctly reflects an AVL Tree property. For core operations, such as inserting, deleting, searching, and updating data, the AVL tree continues to exhibit a time complexity of O(logN) across all input sizes. This could be explained by the tree's structure, as when an operation is performed, the structure automatically cuts the amount of repetition needed to find the specific place for the operation by two. This process happens again and again, resulting in a time complexity of O(logN). This makes AVL trees very reliable for all sorts of operations with large amounts of data, as when implemented properly, it guarantees a time complexity of O(LogN) (GeeksforGeeks, 2023). On top of that, each data point inserted into the tree is already sorted, due to the tree's property, meaning that the programme does not have to spend additional time and space to sort the objects (Mohith J, 2023).

However, AVL trees do contain multiple drawbacks, one of which is the fact that it has a space complexity of O(N). This is due to the fact that, for each of the data points that is inserted into the AVL tree, the tree would need to create a separate node and two pointers for it (Mohith J, 2023). Hence, the memory usage of the structure increases linearly with the input size. Another drawback of the AVL tree is its complicated balancing and deleting logic; thus, it is unsuitable for beginners to implement.

6. Conclusion

7. Implementation

8. References and Attributed

8.1 References

[1]https://xmcasia.com/how-technology-simplifies-financial-record-keeping/?utm_source=chatgpt.com

[2]https://moldstud.com/articles/p-finance-management-mobile-app-development?utm_source=chatgpt.com

[3]https://moldstud.com/articles/p-the-importance-of-a-financial-management-app-in-today-world?utm_source=chatgpt.com

[4]https://moldstud.com/articles/p-finance-management-mobile-app-development?utm_source=chatgpt.com

[5]https://happay.com/blog/features-of-expense-management-software/?utm_source=chatgpt.com

[6]https://happay.com/blog/features-of-expense-management-software/?utm_source=chatgpt.com

[7]https://www.researchgate.net/publication/376427913_Digital_Transformation_in_Financial_Management_Security_and_Efficiency?utm_source=chatgpt.com

[8]https://rsisinternational.org/journals/ijriss/articles/usage-of-digital-finance-applications-and-its-impact-on-financial-well-being-a-conceptual-framework/?utm_source=chatgpt.com

8.2 Attributed

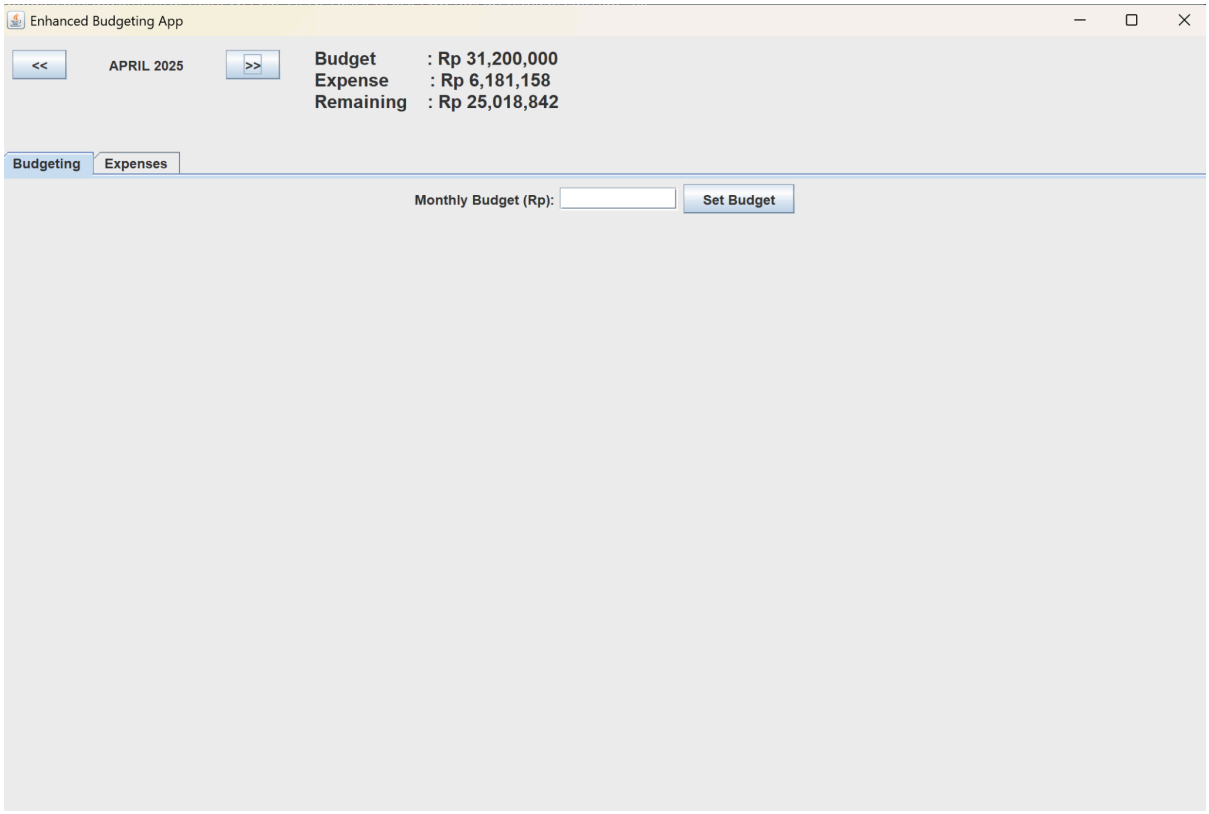
1. Pixabay. (n.d.). *Walkman button* [Sound effect]. Pixabay. https://pixabay.com/sound-effects/walkman-button-272973/

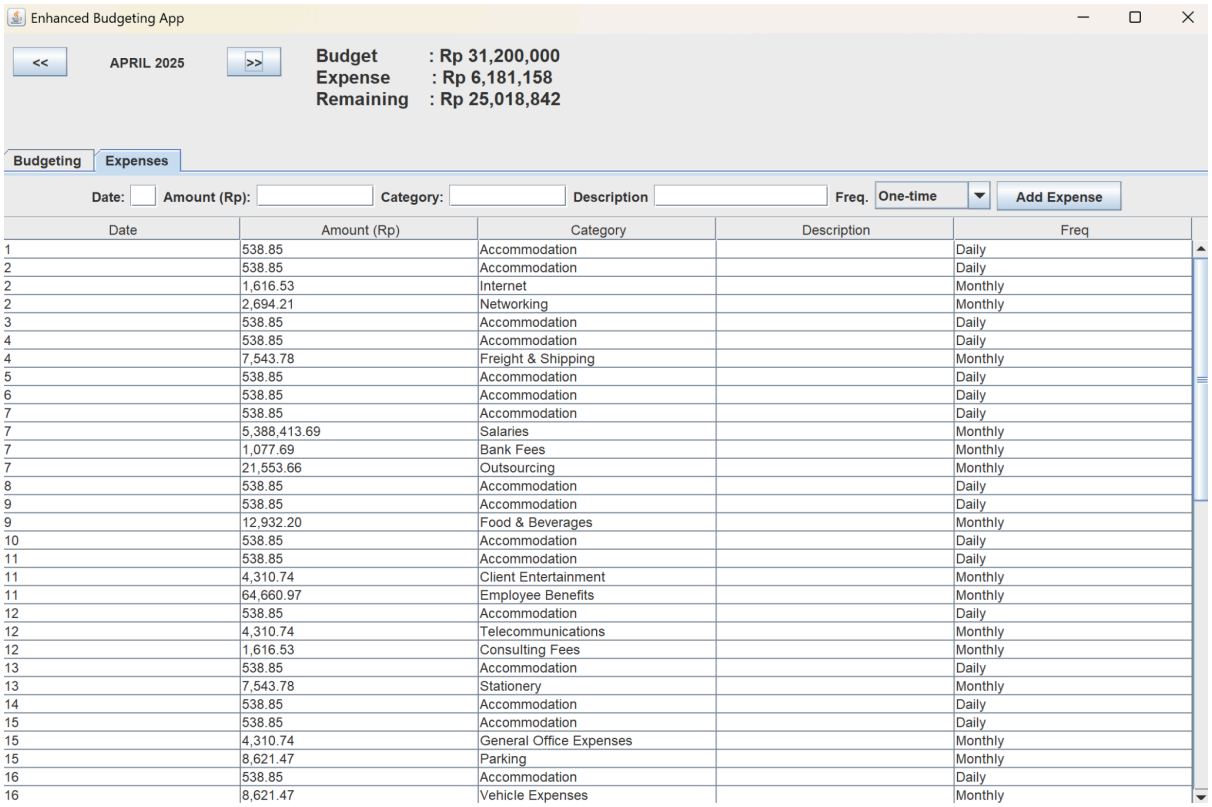
2. Windows error sound https://pixabay.com/sound-effects/windows-error-sound-effect-35894/

9. Appendix

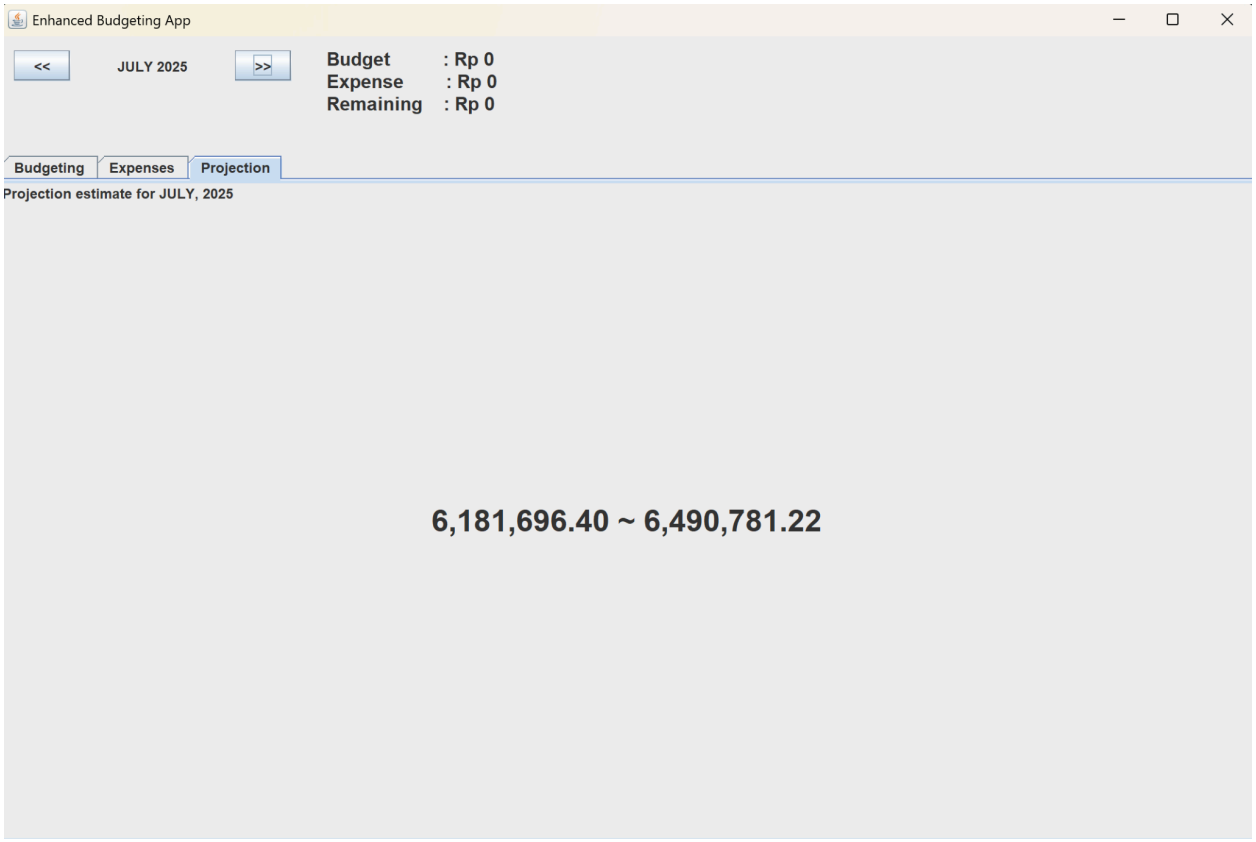
9.1 Manual for Data Structure Testing Program

9.2 App snapshots





Date	Amount (Rp)	Category	Description	Freq
1	538.85	Accommodation		Daily
2	538.85	Accommodation		Daily
2	1,616.53	Internet		Monthly
2	2,694.21	Networking		Monthly
3	538.85	Accommodation		Daily
4	538.85	Accommodation		Daily
4	7,543.78	Freight & Shipping		Monthly
5	538.85	Accommodation		Daily
6	538.85	Accommodation		Daily
7	538.85	Accommodation		Daily
7	5,388,413.69	Salaries		Monthly
7	1,077.69	Bank Fees		Monthly
7	21,553.66	Outsourcing		Monthly
8	538.85	Accommodation		Daily
9	538.85	Accommodation		Daily
9	12,932.20	Food & Beverages		Monthly
10	538.85	Accommodation		Daily
11	538.85	Accommodation		Daily
11	4,310.74	Client Entertainment		Monthly
11	64,660.97	Employee Benefits		Monthly
12	538.85	Accommodation		Daily
12	4,310.74	Telecommunications		Monthly
12	1,616.53	Consulting Fees		Monthly
13	538.85	Accommodation		Daily
13	7,543.78	Stationery		Monthly
14	538.85	Accommodation		Daily
15	538.85	Accommodation		Daily
15	4,310.74	General Office Expenses		Monthly
15	8,621.47	Parking		Monthly
16	538.85	Accommodation		Daily
16	8,621.47	Vehicle Expenses		Monthly



9.3 Repository and Resources

Link to GitHub repository
https://github.com/WILLIAM-RUSMANA/Enhanced-Budgeting-App

Link to Demo video and poster
https://drive.google.com/drive/folders/1X8SVb_UyPilcT84En5GifwhlYZ26B77g