PROJECT REPORT:
PERSONAL FINANCE TRACKER

Managing personal finances effectively requires both tracking your income and expenditures and setting clear financial goals. The **Personal Finance Tracker** is a powerful yet user-friendly application designed to help individuals achieve financial stability. It allows users to track their savings, set financial goals, categorize expenses, and visualize spending patterns through a pie chart representation.It also provides a feature to convert one currency to another which helps user to convert currencies on the spot. By combining goalsetting features with advanced category management , spending analysis,and currency converter this tool becomes a one-stop solution for all personal finance needs.

This **Personal Finance Tracker** is ideal for anyone who wants to efficiently manage their finances, stay motivated to save, and gain valuable insights into their spending patterns. Whether planning for a significant financial milestone or managing day-to-day expenses, this tool empowers users to take control of their financial journey.

KEY FEATURES OF THE APPLICATION

The program includes the following functionalities:

0.User login system:

- Upon entering the program the user can login or signup and continue ahead to the program.

1. Income after tax deduction and Savings Tracking

- Users can input their initial income, which serves as the baseline for financial planning.The income is stored after deducting the tax from it.
- Savings can be updated incrementally, and the current balance is displayed after each update.

2. Financial Goals Management

- Users can set financial goals, specifying:
    - A **goal name** (e.g., “Vacation Fund,” “Emergency Savings”). ○

      A **target amount** to achieve.

      A **timeframe in months** for completing the goal.

- The system monitors progress and congratulates the user upon reaching the goal.

3. Expense Categorization

- Users can create categories to organize their expenses (e.g., Food, Entertainment, Utilities, Transportation).
- Income can be allocated to these categories, ensuring funds are set aside for specific purposes.
- Categories provide clarity on spending habits and make budgeting more effective.

4. Spending Funds in Categories

- Users can spend from specific categories, and the application automatically deducts the spent amount from the allocated funds.
- If insufficient funds are available in a category, the system alerts the user.

5. Pie Chart Visualization of Spending

- The application generates a **pie chart** that visually represents the percentage of income spent on each category.
- This feature provides an instant overview of spending habits and helps users identify areas for improvement.

6. Detailed Financial Overview

- A consolidated view of income, savings, goal progress, category allocations, and expenses is available at any time.
- Users can track progress toward financial goals and analyze spending habits side by side

7. Currency converter

- User can convert currencies from one to another by telling how much he wants to convert.
- This feature helps the user to do all from one place.

HOW TO USE THE PROGRAM

Upon launching the application, users are guided step-by-step through the following workflow:

Step 0:Set Up your account

- Upon entering the program the user is urged to sign up if he doesn't have an account already and if he has the user can enter his login details and proceed.

Step 1: Set Up Income

- Enter the total income for financial planning at the start of the month. For instance, if the user enters "2000," this becomes the available amount for savings and category allocation.The income is stored after the tax is deducted from the income.

Step 2: Allocate Funds to Categories

- Create categories such as "Food," "Rent," "Entertainment," and assign specific amounts to each. For example:
    - Food: $500 ○
    - Rent: $1000 ○
    - Entertainment: $300
    - Miscellaneous: $200
- The system ensures that allocations do not exceed the total income.

Step 3: Spend from Categories

- Select a category and specify the amount spent. For example:
    - Spending $50 on Food deducts $50 from the Food category.
- The application alerts the user if the remaining funds in the category are insufficient.

Step 4: Set Financial Goals

- Define a financial goal with a name, target amount, and duration in months. For instance:
    - Goal: "New Car," Target Amount: $5000, Duration: 6 months.
- Track progress as savings grow toward the target.

Step 5: Monitor Progress and Spending

- View goal progress updates in real-time. The program informs the user of how much has been saved and how much more is needed.
- Check category-wise spending visually through a pie chart, which shows the breakdown of expenses. For instance:

- Food: 25%
- Rent: 50%
- Entertainment: 15%
- Miscellaneous: 10%

Step 6: Analyze and Adjust

- Based on spending habits shown in the pie chart, adjust category allocations or spending patterns to better manage finances.

FUTURE ENHANCEMENTS

While the current version of the **Comprehensive Personal Finance Tracker** provides robust features to manage finances effectively, we have identified several areas for future development to further enhance its functionality:

1. Real-Time Implementation

- Currently, the program uses an advanced day and month simulation method to implement timebased functionalities, such as tracking goal progress and monthly allocations.
- In the future, we aim to integrate real-time tracking using system clocks, enabling users to experience seamless, real-time updates for transactions, goal progress, and category resets.

2. Database Integration

- The current system supports a single user per session. We plan to integrate a database to support multiple users.
- The database will store:
  - User profiles and login credentials.
  - Individual user details, including category allocations, savings progress, and financial goals.
  - Transaction history for each user, enabling personalized tracking and analytics.

3. Filing for Transaction History

- To ensure that users can review their financial activities over time, we will add a filing system.
- At the end of each month, the system will:
    - Automatically store transaction history and category-wise spending in structured files.
    - Provide options for users to view or export their monthly data for analysis or recordkeeping.

4. Real time fetching of current currency conversion rates

- The program can access real time conversion rates to stay updates with the continuously changing exchange rate.

These enhancements will make the application more versatile, allowing users to manage their finances more efficiently while retaining historical records for long-term planning and analysis.

COMPLEX COMPUTING PROBLEM ATTRIBUTES IN THE PROJECT

1. Range of Conflicting Requirements

- **Definition**: Involves wide-ranging or conflicting technical, computing, and other issues.
- **Application in the Project**: ○ The project addresses various user needs, including fund allocation, savings tracking, goalsetting, and spending visualization. Balancing these requirements led to conflicting demands.
    - For example, implementing time simulation for monthly resets conflicted with the need for real-time tracking. A simulated time advancement method was chosen for simplicity and feasibility, but future enhancements aim to integrate real-time updates.
    - Another conflict arose in presenting detailed visualizations (pie charts) versus maintaining simplicity in the user interface. The solution was modularity, where users access features sequentially.

---

2. Depth of Analysis Required

- **Definition**: Has no obvious solution and requires conceptual thinking and innovative analysis to formulate abstract models.
- **Application in the Project**:
    - Abstract concepts like goal progress, monthly resets, and fund allocation were translated into structured algorithms. ○ The system models categories as nodes in a binary tree and transactions as linked lists for efficient access and storage.
    - Pie chart visualization required mapping the financial data dynamically to percentages, ensuring accurate representation regardless of user inputs.

---

3. Depth of Knowledge Required

- **Definition**: Requires in-depth computing or domain knowledge and an analytical approach based on well-founded principles.
- **Application in the Project**:
    - The project combines concepts from multiple domains:
        - **Finance**: Budgeting, savings, and expense management.
        - **Data Structures**: Binary trees and linked lists for category and transaction handling.
        - **Visualization**: Generating pie charts to represent data visually.

      Designing the interaction between savings, goals, and categories required a solid understanding of both programming and financial principles.

---

4. Familiarity of Issues

- **Definition**: Involves infrequently encountered issues.
- **Application in the Project**:
    - Integrating a time simulation method with financial tracking posed unique challenges. This approach, while simpler than real-time tracking, required careful consideration to align time-based resets with category updates and goal progress.
    - Dynamically updating pie charts to reflect category spending, especially after each transaction, involved advanced synchronization of visualization and back-end data.

---

5. Level of Problem

- **Definition**: Outside problems encompassed by standards and standard practice for professional computing.
- **Application in the Project**:
    - The system required non-standard solutions for problems like:
        - Simulating time advancement for monthly resets.
        - Managing interdependent modules (e.g., category updates affecting savings and goal progress).
        - Providing real-time pie chart visualizations that dynamically adapt to user transactions.
    - These problems go beyond typical budget-tracking tools, demanding innovative designs and abstractions.

---

6. Extent of Stakeholder Involvement and Conflicting Requirements

- **Definition**: Involves diverse groups of stakeholders with widely varying needs.
- **Application in the Project**: ○ Potential users include individuals, families, and small business owners, each with unique financial tracking needs.

- Balancing conflicting requirements—like offering simple budgeting features for casual users versus advanced goal-setting and analysis for power users—required a flexible, modular approach.
- The system allows users to add only the features they need (e.g., goal tracking is optional).

---

7. Consequences

- **Definition**: Has significant consequences in a range of contexts.
- **Application in the Project**:
  - Inaccurate financial tracking could lead users to mismanage their budgets or fail to achieve goals, directly impacting their financial health.
  - The project's robust design ensures accuracy and clarity, promoting informed financial decisions. ○ Visualization tools (pie charts) allow users to identify overspending quickly, enabling course correction.

---

8. Interdependence

- **Definition**: Is a high-level problem possibly including many component parts or sub-problems.
- **Application in the Project**:
  - The project integrates multiple interdependent modules:
    - **Savings Management**: Tracks and updates user savings.
    - **Goal Tracking**: Monitors progress toward specific financial goals.
    - **Category Management**: Allocates funds and tracks expenses.
    - **Visualization**: Generates dynamic pie charts.
  - Each module interacts with others. For example:
    - Spending in a category affects pie chart visualization. ▪ Savings updates directly influence goal progress.

---

9. Requirement Identification

- **Definition**: Identification of a requirement or the cause of a problem is ill-defined or unknown.
- **Application in the Project**: ○ The initial requirements were loosely defined: users needed a financial tracker, but specifics like categorization, goal management, and visualization were added incrementally.
  - Iterative refinement identified user needs, such as monthly resets, detailed transaction tracking, and category-specific expense visualization.
  - This dynamic evolution of requirements led to a system adaptable to a broad range of use cases.

-

CODE OVERVIEW FUNCTION BY FUNCTION(EXCLUDING THE UI IMPLEMENTATION)

1.concise breakdown of the headers:

```
#include "mainwindow.h"
#include "./ui_mainwindow.h"
#include <QtCharts/QChartView>
#include <QtCharts/QPieSeries>
#include <QtCharts/QPieSlice>
#include <iostream>
#include <string>
#include <iomanip>
#include <map>
```

```
#include <vector>
#include <algorithm>
#include <QtCore/QCoreApplication>
#include <QMessageBox>
#include <QLayout>
#include <QString>
#include <QTableWidget>
```

- **QChartView, QPieSeries, QPieSlice**: Used to create and display pie charts for visualizing financial data (like expense categories).
- **iostream, string, iomanip**: Standard C++ libraries for input/output and string manipulation.
- **map, vector, algorithm**: C++ containers for storing and manipulating data like categories and transactions.
- **QCoreApplication**: Manages the application's main event loop.
- **QMessageBox**: Displays dialog boxes for messages or warnings.
- **QLayout**: Manages the layout of widgets in the GUI.
- **QString**: Handles Unicode text strings in Qt applications.
- **QTableWidget**: Displays tabular data, likely for transaction history or financial details.

2.Goal Struct:

```
struct Goal {
    string name;
double targetAmount;
int targetMonths;
double goalProgress;
bool isCompleted;

    Goal(string goalName, double targetAmt, int months)
        : name(goalName), targetAmount(targetAmt), targetMonths(months),
goalProgress(0), isCompleted(false) {}
     void setGoal(double targetAmt, int months) {         targetAmount =
targetAmt;         targetMonths = months;         goalProgress = 0;
isCompleted = false;         cout << "Goal set successfully! Target: " <<
targetAmount << " in " << targetMonths << " months.\n";
    }      bool checkGoalProgress(int
savingsBalance) {




        goalProgress = savingsBalance;  // Track the savings balance         if
(goalProgress >= targetAmount) {             isCompleted = true;             cout
<< "🚀 Congratulations! You have achieved your goal: " << name <<
"!\n";             return true;  // Indicate the goal is achieved and should
be reset
        }         return false;  // Goal not
yet achieved
    }
};
```

This Goal struct represents a financial goal in the tracker. Here's a breakdown of its components:

1. **Attributes**:
    - name: The name of the goal (e.g., "Vacation Fund"). ○ targetAmount: The target amount to be saved for the goal. ○ targetMonths: The number of months in which the goal should be achieved. ○ goalProgress: The current progress toward the target, updated with savings.
    - isCompleted: A flag indicating whether the goal has been achieved.

2. **Constructor**:
    - Initializes the goal with a name, target amount, and target months, while setting the initial progress to 0 and marking the goal as incomplete.
3. **Methods**:
    - setGoal(double targetAmt, int months): Allows the user to set or update the goal's target amount and duration. It resets the progress and marks the goal as incomplete.
    - checkGoalProgress(int savingsBalance): Compares the current savings balance to the target amount. If the savings meet or exceed the target, it marks the goal as completed and prints a congratulatory message. Returns true if the goal is completed, otherwise false.

This structure helps in managing financial goals by tracking the user's progress and notifying them upon goal completion.

3.DateTime Struct:

```
struct DateTime {     int day, month, year, hour, minute;
```

```
    DateTime(int d = 1, int m = 1, int y = 2023, int h = 0, int min = 0)
        : day(d), month(m), year(y), hour(h), minute(min) {}
     bool isLeapYear() const {         return (year % 4 == 0 && (year % 100 != 0
|| year % 400 == 0));     }      int daysInMonth() const {         const int
daysInEachMonth[12] = {31, (isLeapYear() ? 29 : 28), 31, 30, 31,
30, 31, 31, 30, 31, 30, 31};
return daysInEachMonth[month - 1];
    }      void advance(int days, int hours, int
minutes) {         minute += minutes;         hour
+= hours + minute / 60;         day += days + hour /
24;         minute %= 60;         hour %= 24;
         while (day > daysInMonth())
{             day -= daysInMonth();
month++;             if (month > 12)
{                 month = 1;
year++;
            }
        }
    }      void display() const {         cout <<
setw(2) << setfill('0') << day << "/"
             << setw(2) << setfill('0') << month << "/"
             << year << " "
             << setw(2) << setfill('0') << hour << ":"
             << setw(2) << setfill('0') << minute;
    }
    bool operator<(const DateTime& other) const {         if
(year != other.year) return year < other.year;         if
(month != other.month) return month < other.month;         if
(day != other.day) return day < other.day;         if (hour
!= other.hour) return hour < other.hour;         return
minute < other.minute;
    }
};
```

The DateTime struct represents a specific point in time and includes several methods for managing and manipulating date and time values. Here's a breakdown:

Attributes:

- **day**: The day of the month.
- **month**: The month (1-12).
- **year**: The year.
- **hour**: The hour of the day (0-23).
- **minute**: The minute (0-59).

Constructor:

- **DateTime(int d, int m, int y, int h, int min)**: Initializes the date and time with given values. Default values are set for day, month, year, hour, and minute.

Methods:

- **isLeapYear()**: Returns true if the year is a leap year (i.e., February has 29 days), and false otherwise. It checks if the year is divisible by 4, but not 100, unless divisible by 400.
- **daysInMonth()**: Returns the number of days in the current month. It uses an array that holds the number of days for each month, adjusting February's days if the year is a leap year.
- **advance(int days, int hours, int minutes)**: Advances the date and time by a specified number of days, hours, and minutes. It adjusts the day, month, and year as needed (handles overflow for minutes, hours, and days).
- **display()**: Prints the date and time in the format DD/MM/YYYY HH:MM, ensuring the day, month, hour, and minute are displayed with two digits.
- **operator<**: Overloads the < operator to allow comparison between two DateTime objects. It compares year, month, day, hour, and minute in order, returning true if the current object is earlier than the other.

4.Transaction and Category nodes :

```
struct Transaction {     int amount;
    string type;
string category;
DateTime dateTime;
    Transaction* next;

    Transaction(int amt, string t, DateTime dt, string c) : amount(amt), type(t),
dateTime(dt), next(nullptr), category(c) {}
};  struct
CategoryNode {
string name;     int
balance;
    Transaction* transactions;
    CategoryNode* left;
    CategoryNode* right;

    CategoryNode(string n) : name(n), balance(0), transactions(nullptr),
left(nullptr), right(nullptr) {}
     void addTransaction(int amount, string type, DateTime dateTime, string category)
{
        Transaction* newTransaction = new Transaction(amount, type,
dateTime, category);         newTransaction->next = transactions;
transactions = newTransaction;
    }      void resetBalance() {         balance
= 0; // Reset the category balance
    }
    void displayTransactions() {
        Transaction* temp = transactions;
while (temp) {
            cout << "Type: " << temp->type << ", Amount: " << temp->amount << ", Date:
";
            temp->dateTime.display();
cout << endl;             temp =
temp->next;
        }
    }
};
```

The Transaction and CategoryNode structs are used to represent financial transactions and categories within a personal finance tracking system. Here's an explanation of their components and functionality:

Transaction Struct:

This struct represents a single financial transaction.

- **amount**: The amount of money involved in the transaction.
- **type**: The type of the transaction (e.g., "expense" or "income").
- **category**: The category of the transaction (e.g., "Food", "Rent").
- **dateTime**: A DateTime object representing when the transaction occurred.
- **next**: A pointer to the next Transaction in a linked list. This allows multiple transactions to be linked together for a particular category.

Constructor:

- Initializes a new Transaction with the given amount, type, date, and category. Sets the next pointer to nullptr.

CategoryNode Struct:

This struct represents a financial category (e.g., "Food", "Entertainment").

- **name**: The name of the category (e.g., "Food").
- **balance**: The total balance of the category. It tracks the available amount in that category.
- **transactions**: A linked list of transactions for this category, allowing for easy tracking of all transactions in that category.
- **left** and **right**: Pointers used for a binary search tree (BST) structure. These allow the categories to be stored in a hierarchical structure (e.g., sorted alphabetically).

Constructor:

- Initializes a CategoryNode with the given name, and sets its balance to 0 and the transactions list to nullptr.

Methods:

1. **addTransaction(int amount, string type, DateTime dateTime, string category)**:
    - Creates a new Transaction and adds it to the transactions linked list of the CategoryNode. The new transaction is added at the head of the list.
2. **resetBalance()**:

- Resets the balance of the category to 0, essentially clearing out any previous calculations of the remaining funds in that category.

3. **displayTransactions()**:

   - Iterates through the linked list of transactions and displays each one, showing its type, amount, and the date/time of the transaction. The DateTime object's display() method is called to format the date and time.

5.Currency Converter :

```
class CurrencyConverter { private:
    std::map<std::string, double> conversionRates;
 public:
    CurrencyConverter() {
conversionRates["USD"] = 1.0;
conversionRates["PKR"] = 270.0;
conversionRates["GBP"] = 0.76;
    }

    // Conversion function     double convertCurrency(double amount, const
std::string& fromCurrency, const std::string& toCurrency) {         if
(conversionRates.find(fromCurrency) == conversionRates.end() ||
conversionRates.find(toCurrency) == conversionRates.end()) {
std::cout << "Error: Unsupported currency." << std::endl;             return -1;
        }
         double amountInUSD = amount / conversionRates[fromCurrency];
return amountInUSD * conversionRates[toCurrency];     }

    // Getter for conversion rates     const std::map<std::string,
double>& getConversionRates() const {         return conversionRates;
    }

    // Display conversion rates in a QListWidget     static void
displayConversionRates(QListWidget* listWidget, const
std::map<std::string, double>& rates) {         listWidget->clear();

        // Retrieve individual rates
double usdToPkr = rates.at("PKR");
double usdToGbp = rates.at("GBP");
double usdToUsd = rates.at("USD");

        // PKR to others         double
pkrToUsd = 1.0 / usdToPkr;         double
pkrToGbp = usdToGbp / usdToPkr;
```

```
        // GBP to others
        double gbpToUsd = 1.0 / usdToGbp;
double gbpToPkr = usdToPkr / usdToGbp;

        // Add items manually          listWidget->addItem("1 USD = " +
QString::number(usdToPkr, 'f', 2) + " PKR");          listWidget->addItem("1 USD = " +
QString::number(usdToGbp, 'f', 2) + " GBP");
         listWidget->addItem("1 PKR = " + QString::number(pkrToUsd, 'f', 4) + "
USD");          listWidget->addItem("1 PKR = " + QString::number(pkrToGbp, 'f', 4) + "
GBP");
         listWidget->addItem("1 GBP = " + QString::number(gbpToUsd, 'f', 2) + " USD");
listWidget->addItem("1 GBP = " + QString::number(gbpToPkr, 'f', 2) + " PKR");     }
};
```

The CurrencyConverter class is designed to handle currency conversion operations and display the available conversion rates. Here's a breakdown of its components:

Private Members:

- **std::map<std::string, double> conversionRates;**:
  - A map to store the conversion rates between different currencies (e.g., USD, PKR, GBP). The key is the currency code (e.g., "USD"), and the value is the exchange rate relative to USD.

Public Methods:

1. **Constructor (CurrencyConverter())**:
   - Initializes the conversion rates for the currencies. It sets:
     - USD to USD as 1.0.
     - PKR to USD as 270.0.
     - GBP to USD as 0.76.
2. **convertCurrency(double amount, const std::string& fromCurrency, const std::string& toCurrency)**:
   - Converts a given amount from one currency to another based on the exchange rates.
   - First, it checks if both the source (fromCurrency) and target (toCurrency) currencies exist in the conversionRates map. If either is not found, it returns -1 and prints an error message.

- It converts the given amount to USD first (using the fromCurrency rate), and then converts it from USD to the target currency (toCurrency).

3. **getConversionRates()**:
    - Returns a reference to the map holding the conversion rates. This can be used to access or update the rates.
4. **static void displayConversionRates(QListWidget* listWidget, const std::map<std::string, double>& rates)**:
    - A static method that displays the conversion rates in a QListWidget (a Qt widget for displaying a list of items).
    - It calculates and formats the conversion rates between the supported currencies (USD, PKR, GBP) and adds them as items in the listWidget.
    - The conversion is done in both directions for each pair (e.g., USD to PKR and PKR to USD).
    - Uses QString::number() to format the conversion rates to a desired decimal precision for display.

6.Finance Tracker Class :

```
class FinanceTracker {

private:
    CategoryNode* root;
int totalIncome;
    CategoryNode* savings;
    CategoryNode* emergencyFunds;
    CategoryNode* sinkingFunds;
    DateTime currentDate;
    CurrencyConverter converter;
    Goal* currentGoal = nullptr;  // Pointer to dynamically manage goal lifecycle
    CategoryNode* insertCategory(CategoryNode* node, string name) {
if (!node) return new CategoryNode(name);
        if (name < node->name) node->left = insertCategory(node->left, name);
else if (name > node->name) node->right = insertCategory(node->right, name);
return node;
    }

    CategoryNode* searchCategory(CategoryNode* node, string name) {
if (!node || node->name == name) return node;
        if (name < node->name) return searchCategory(node->left, name);
return searchCategory(node->right, name);
    }


    // Gather all categories' balances    void
 gatherCategoryBalances(CategoryNode* node, QPieSeries* series) {
```

```
        if (!node) return;         gatherCategoryBalances(node->left,
series);         series->append(QString::fromStdString(node->name), node-
>balance);         gatherCategoryBalances(node->right, series);
    }      void resetUserCategories(CategoryNode*
node) {         if (!node) return;         node-
>resetBalance();         resetUserCategories(node-
>left);         resetUserCategories(node->right);
    }                  void   sortTransactionsByDate(vector<Transaction*>&
sortedTransactions) {
        // Sort the vector of transactions by date
        sort(sortedTransactions.begin(), sortedTransactions.end(), [](Transaction* a,
Transaction* b) {                return a->dateTime < b->dateTime; // Compare using
DateTime's overloaded operator<
        });
    }

    // Function to calculate tax and deduct it from the income     void
calculateTax(double& income, QLabel* taxdeduct) {         double taxRate = 0.1; //
10% tax rate         double tax = income * taxRate;         income -= tax;
taxdeduct->setText("Tax calculated and deducted: " + QString::number(tax) +
"\n");
    }
public:
    FinanceTracker(int income) : root(nullptr), totalIncome(income), currentDate(1,
1, 2023) {
        savings = new CategoryNode("Savings");
        emergencyFunds = new CategoryNode("Emergency Funds");
sinkingFunds = new CategoryNode("Sinking Funds");     }
    void addCategory(string name) {
root = insertCategory(root, name);
    }
    void displayCategoryBalances(QTableWidget* table, CategoryNode* savings,
CategoryNode* emergencyFunds, CategoryNode* sinkingFunds, CategoryNode* root) {
```

```
        // Clear any existing rows in the table
```

```
        table->setRowCount(0);

        // Add the three hardcoded categories to the table
```

```
        int rowCount = 0;

        // Add Savings         table->insertRow(rowCount);         table-
>setItem(rowCount, 0, new QTableWidgetItem("Savings"));         table-
>setItem(rowCount, 1, new QTableWidgetItem(QString::number(savings-
>balance)));
rowCount++;

        // Add Emergency Funds         table->insertRow(rowCount);
table->setItem(rowCount, 0, new QTableWidgetItem("Emergency Funds"));
table->setItem(rowCount, 1, new
QTableWidgetItem(QString::number(emergencyFunds->balance)));
rowCount++;

        // Add Sinking Funds         table->insertRow(rowCount);
table->setItem(rowCount, 0, new QTableWidgetItem("Sinking Funds"));
table->setItem(rowCount, 1, new
QTableWidgetItem(QString::number(sinkingFunds->balance)));
rowCount++;

        // Recursive lambda to add user-defined categories
std::function<void(CategoryNode*)> addCategories = [&](CategoryNode* node) {
if (!node) return;

            // Add left subtree
categories
addCategories(node->left);
            // Add current node's data             table-
>insertRow(rowCount);             table->setItem(rowCount, 0,
new
QTableWidgetItem(QString::fromStdString(node->name)));
            table->setItem(rowCount, 1, new QTableWidgetItem(QString::number(node-
>balance)));
            rowCount++;

            // Add right subtree categories
addCategories(node->right);
        };

        // Add user-defined categories starting from the root
addCategories(root);
```

```
}
void viewCategoriesAndBalances(QTableWidget* table) {
```

```
        displayCategoryBalances(table, savings, emergencyFunds, sinkingFunds, root);
}

    CategoryNode* getCategoryRoot(){
return root;
    }

    Goal* getCurrentGoal() const {
return currentGoal;
    }      int getSavingsBalance() const {         return savings->balance;
// Assuming savings is a predefined category     }

    DateTime    getCurrentDate()    const    {
return currentDate;
    }      bool allocateFunds(const std::string& name, int
amount) {
        // Validate amount         if (amount <= 0) {
std::cout << "Error: Amount must be greater than zero.\n";
return false;
        }

        // Find the appropriate category
        CategoryNode* category = (name == "Savings" ? savings :
                                  (name == "Emergency Funds" ? emergencyFunds :
(name == "Sinking Funds" ? sinkingFunds :
                                                searchCategory(root, name))));
        // Check if the category exists and if there is enough income
if (category && amount <= totalIncome) {
            // Allocate funds
            category->balance += amount;
totalIncome -= amount;
            // Log the transaction
            category->addTransaction(amount, "add", currentDate, name);

            std::cout << "Successfully allocated " << amount << " to " << name <<
".\n";
            return true;
        }
```

```
// Provide meaningful feedback for failure if (!category) {
std::cout << "Error: Category " << name << " not found.\n";
```

```
        } else if (amount > totalIncome) {            std::cout << "Error:
Insufficient income to allocate " << amount <<
".\n";         }
return false;
    }
    bool spendFunds(const string& name, int amount) {
        CategoryNode* category = (name == "Savings" ? savings :
                                  (name == "Emergency Funds" ? emergencyFunds :
(name == "Sinking Funds" ? sinkingFunds :
                                                searchCategory(root, name))));
         if (category) {             if (category->balance >= amount) {
category->balance -= amount;                  category->addTransaction(-
amount, "Spent", currentDate, name);                  std::cout << "Spent " <<
amount << " from " << name << "\n";                return true;
            } else {                std::cout << "Insufficient funds in " <<
name << " category.\n";                return false;
            }
        }
         cout << "Category " << name << " not
found.\n";         return false;
    }
    void setGoal(const string& goalName, double targetAmount, int months) {
if (currentGoal) {             cout << "A goal already exists. Resetting the
previous goal.\n";             delete currentGoal;
        }
        currentGoal = new Goal(goalName, targetAmount,
months);     }
    void checkGoalProgress(int savingsBalance) {
if (currentGoal) {
            if (currentGoal->checkGoalProgress(savingsBalance)) {
// Goal achieved, reset the currentGoal pointer                delete currentGoal;
currentGoal = nullptr;                cout << "The goal has been achieved and
reset. You can now set a new
```

```
goal.\n";
```

```
        } else {
            cout << "Goal progress: " << currentGoal->goalProgress << " / " <<
```

```
currentGoal->targetAmount << "\n";
            }         } else {             cout <<
"No goal is currently set.\n";
        }
    }      void startNewMonth(double newIncome, QLabel* taxdeduct, QLabel*
newincome) {         transferCategoryBalancesToSavings();
calculateTax(newIncome, taxdeduct);  // Deduct tax from new income
totalIncome = newIncome;

        //          Advance          to          the          next          month
currentDate.advance(currentDate.daysInMonth(), 0, 0);

printTransactionHistory();
resetUserCategories(root);
        newincome->setText("New Income Entered (After Tax Deduction): " +
QString::number(totalIncome) + "\n");
    }      void startNewDay() {
currentDate.advance(1, 0, 0);
    }      void transferCategoryBalancesToSavings() {
savings->balance += root->balance;         root-
>resetBalance();         emergencyFunds->balance +=
sinkingFunds->balance;         sinkingFunds-
>resetBalance();
    }      void
printTransactionHistory() {
        vector<Transaction*> allTransactions;

        // Gather transactions from all categories
gatherCategoryTransactions(root, allTransactions);
gatherCategoryTransactions(savings, allTransactions);
gatherCategoryTransactions(emergencyFunds, allTransactions);
gatherCategoryTransactions(sinkingFunds, allTransactions);
        // Sort transactions by date
        sortTransactionsByDate(allTransactions);

        // Print the sorted transaction history         cout <<
"Transaction History (Sorted by Date & Time):\n";
```

```
for (auto& transaction : allTransactions) {
cout << "Amount: " << transaction->amount
         << ", Type: " << transaction->type
```

```
                << ", Date: ";
transaction->dateTime.display();
cout << endl;
        }
    }      void showMonthlyTransactions(QTableWidget*
tableWidget) {
        // Clear the existing data in the table
tableWidget->clearContents();         tableWidget-
>setRowCount(0);

        // Gather all transactions from categories (using
std::vector)         std::vector<Transaction*> allTransactions;
gatherCategoryTransactions(root, allTransactions);
gatherCategoryTransactions(savings, allTransactions);
gatherCategoryTransactions(emergencyFunds, allTransactions);
gatherCategoryTransactions(sinkingFunds, allTransactions);
        // Sort transactions by date (same logic as
printTransactionHistory)         sortTransactionsByDate(allTransactions);
        // Set up the table headers         tableWidget->setColumnCount(4);
// Only 3 columns: Amount, Type, Date         tableWidget-
>setHorizontalHeaderLabels({"Amount", "Type", "Date", "Category"});

        // Populate the table with all transactions
        for (int row = 0; row < allTransactions.size(); ++row) {
            Transaction* transaction = allTransactions[row];
            // Insert a new row
            tableWidget->insertRow(row);
            // Set data for each column (use std::string for conversion)
tableWidget->setItem(row, 0, new
QTableWidgetItem(QString::fromStdString(std::to_string(transaction->amount))));
tableWidget->setItem(row, 1, new
QTableWidgetItem(QString::fromStdString(transaction->type)));
            tableWidget->setItem(row, 2, new QTableWidgetItem(QString::fromStdString(
std::to_string(transaction-
>dateTime.day) + "-" +
                                                std::to_string(transaction-
>dateTime.month) + "-" +
std::to_string(transaction>dateTime.year)
```

```
                                                    )));
        tableWidget->setItem(row, 3, new
QTableWidgetItem(QString::fromStdString(transaction->category)));
```

```
        }

        // Resize columns to fit the content         tableWidget-
>resizeColumnsToContents();
    }      void gatherCategoryTransactions(CategoryNode* node,
vector<Transaction*>& allTransactions) {
        if (!node) return;

        // Add transactions from the current
node          Transaction* temp = node-
>transactions;         while (temp) {
allTransactions.push_back(temp);
temp = temp->next;
        }

        // Recurse into left and right subtrees
gatherCategoryTransactions(node->left, allTransactions);
gatherCategoryTransactions(node->right, allTransactions);     }
    void displayPieChart() {
        QPieSeries* series = new QPieSeries();
        // Add predefined categories to the pie chart
series->append("Savings", savings->balance);         series-
>append("Emergency Funds", emergencyFunds->balance);
series->append("Sinking Funds", sinkingFunds->balance);
        // Gather user-added categories and add them to the pie chart
gatherCategoryBalances(root, series);

        // Create a chart and add the series
QChart* chart = new QChart();         chart-
>addSeries(series);
        chart->setTitle("Income Distribution by Category");
        // Create a chart view and display it
        QChartView* chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);
chartView->resize(600, 400);         chartView->show();
    }
```

```
    ~FinanceTracker() {     if (currentGoal) {
    delete currentGoal;  // Ensure no memory leaks
        }
    }
};
```

This code defines a FinanceTracker class, which represents a system for managing personal finances, categorizing expenses, allocating funds, setting goals, tracking transactions, and generating reports. Here's a summary of the key features and components of the class:

Key Components:

1. **Category Management**:
    - The system uses a binary search tree (BST) to store categories. The categories can include predefined ones like "Savings", "Emergency Funds", and "Sinking Funds", as well as userdefined categories. ○ The insertCategory and searchCategory methods are used for adding and retrieving categories from the BST.
2. **Transaction Management**:
    - Transactions are tracked within each category using a linked list (Transaction pointer). Transactions include details like the amount, type (e.g., "add", "spent"), date, and associated category.
    - Transactions are sorted by date for viewing or reporting purposes.
3. **Fund Allocation & Spending**:
    - Funds can be allocated to or spent from various categories. The system ensures there is enough income before proceeding with any transactions.
4. **Tax Calculation**:
    - The calculateTax method deducts a tax from the income at a fixed rate (10%).
5. **Goals**:
    - The setGoal and checkGoalProgress methods allow users to set financial goals (e.g., saving a target amount within a specified number of months) and track their progress towards meeting these goals.

6. **Monthly and Daily Operations**:

- At the start of a new month, the system transfers balances from user categories to "Savings", calculates tax, and updates the total income.
- The startNewDay function advances the current date by one day.

7. **Transaction History & Reporting**:
   - The printTransactionHistory method prints all transactions sorted by date.
   - The showMonthlyTransactions method displays transactions in a table view in a Qt widget, with columns for amount, type, date, and category.
8. **Pie Chart Visualization**:
   - The displayPieChart method generates a pie chart using QPieSeries to visualize the distribution of funds across different categories.

7.User Struct :

```
struct User {
string username;
string password;
    FinanceTracker Tracker;

    User(const string &u, const string &p, int income) : username(u),
password(p), Tracker(income) {}
};
vector<User> users;
User* currentUser = nullptr; // Global pointer to the currently logged-in user
```

The User struct and the accompanying global variables users and currentUser are part of a simple system for handling user accounts within the FinanceTracker application. Here's a breakdown of the components:

Components: 1.

User Struct:

- The User struct stores information about each user, including:
  - username: A string representing the user's name.
  - password: A string representing the user's password (which should ideally be stored in a hashed format for security).

- Tracker: An instance of the FinanceTracker class, initialized with the user's income.

2. **users Vector**:
    - vector<User> users: A vector to store multiple User objects. This is essentially a collection of all registered users.
3. **currentUser Pointer**:
    - User* currentUser = nullptr: A global pointer to the currently logged-in user. It starts as nullptr and is later set to point to a valid User object when a user successfully logs in.

GROUP MEMBERS

1.DT-002 Asiya Sohail

2.DT-003 Amna Altaf

3.DT-027 Asma Javed