Behnam Khabazan

Professor Dimpsey

CSS 342

Project 6: Documentations and UML

Project 5 concludes of 5 working classes: Transaction class, Fund class, Account class, Binary Search Tree class, and finally the driver. Each class has different tasks and variants that they must hold which then gets used as a function in another class. In this specific method, the driver starts off the function by location and opening the specified file that holds the excitable codes, translates them to a list so that python can hold on to it, and adds it to a queue waiting to be executed. Then, the driver will go through the queue and its commands using the BinarySearchTree(BST) class to open accounts if they don’t exist, or complete the action specified. The BST class will have functions to open new nodes (Accounts) and call actions to those nodes if needed. Then, it will be able to store each account in a specific order accurately to be found later. The Account class is the main holder of funds and represents a client’s normal bank account. It will be where the client can withdraw, deposit, or print their history for 10 funds named differently. The funds in the account are instances of the Fund class where the money aspect is held privately with actions such as addition, set, or getters.

**Transaction:**

There should be one but idk what it does lol

**Fund**:

The fund class is the main basis of the project as it holds money within it. 10 instances of it will be made in account within the account class, and to keep them distinguished, the fund class will also have a variable that holds its name, and history of purchases:

Variables:

* money – represents the money instance
* history – a list of strings that represent the history of transactions
* name – a name to differentiate between the funds

Methods:

* add() and remove() which will add/reduce from the money instance
* getMoney() which will return the money
* getName() which will return the name
* set() that will set the money instance
* appendHistory() that will add to the history of that fund
* display() which will return the history of the function as a string

|  |
| --- |
| Class: Fund |
| name: str |
| money: str |
| history: list |
| Fund(name): None |
| add(money: int): None |
| remove(money: int): None |
| getMoney(): int |
| getName(): str |
| Set(money: int): None |
| appendHistory(input: str): None |

**Account:**

Account is the controller for funds, it holds a 4-digit ID, a list of funds, and the name of the holder. The funds are all named and held in the funds list in the constructor and initialized to 0.

Variables:

* ID – the 4-digit unique ID set for each account
* funds – the list of Funds that hold money
* firstName – first name of account holder
* lastName – last name of account holder

Methods:

* deposit(index, total) – must check invariants for index and total
* withdraw(index, total) – must check invariants for index and total, 0 and 1 are linked to one another, so are 2 and 3
* transfer(index, total, account, index2) – transfers money from indexed fund, to the new given account and its index after checking to see if index and total are correct numbers
* getId() – returns the ID set above
* getBalance() – prints the balance for all funds in the account
* printHistory() – prints the history for the accounts funds (all of them)
* printFundHistory() – prints the history for one fund

|  |
| --- |
| Class: Account |
| id: int |
| funds: list |
| firstName: str |
| lastName: str |
| Account(id: int, firstName: str, lastName: str): None |
| deposit(index: int, total: int): bool |
| Transfer(index: int, total: int, account: Account, index2: int): bool |
| Withdraw(index:int, total: int): bool |
| getBalance(): None |
| getId(): int |
| printHistory(): None |
| printFundHistory(): None |

**Node:**

The Node class is the inner class for the BinarySearchTree. It consists of a key, value, and the links to its child nodes.

Variables:

* key – a unique key that will allow the tree to sort and hold the nodes (account ID)
* value – the value is the correlating Account to the key
* leftChild – initialized at None, but will turn to the reference to the lower child
* rightChild – initialized at None, but will turn to the reference to the bigger than child

Methods:

* getLeftChild() – returns the none child if any exist
* getRightChild() – returns the right child if any exist
* setLeftChild(Node) – sets the left child (id or key have to be smaller than the current)
* setRightChild(Node) – sets the right child (id or key have to be bigger than the current)
* getValue() – returns the value set
* setValue(Object) – changes the value (not practical but can be useful)
* getKey() – returns the key or ID set for the Node
* setKey(Object) – changes the key or ID set for the Node
* isLeaf() – if no child nodes/references exist then it is a leaf

|  |
| --- |
| Class: Node |
| key: Object |
| value: Account |
| leftChild: Node |
| rightChild: Node |
| Node(key: int, value: Account): None |
| getLeftChild(): Node |
| getRightChild(): Node |
| setLeftChild(Node): None |
| setRightChild(Node): None |
| getValue(): Account |
| setValue(account: Account): None |
| getKey(): Object |
| setKey(input: Object): None |
| isLeaf(): bool |

**BinarySearchTree:**

The BinarySearchTree is a holder of the Account instances, it will keep them sorted by their IDs and keep track of whether they exist or not.

Variables:

* root: Node
* size: int

Methods:

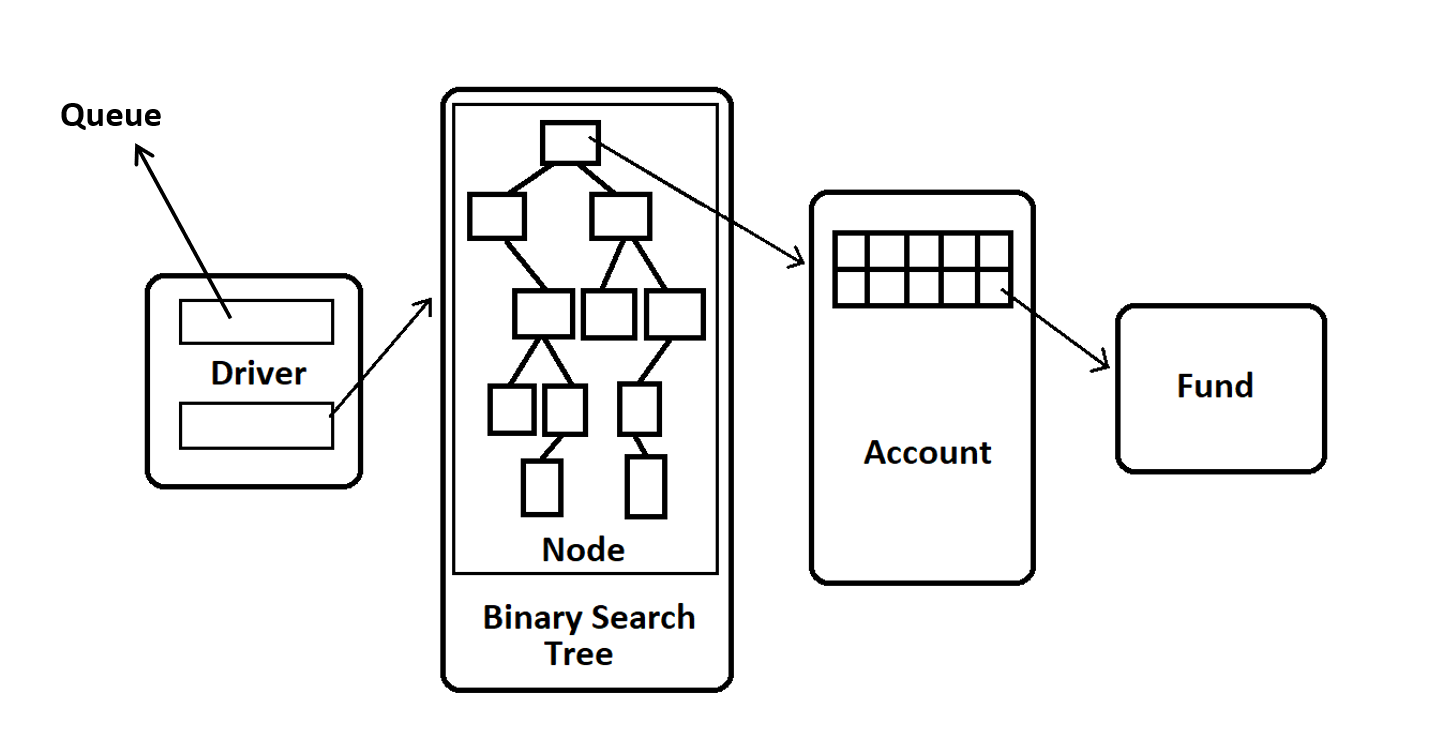
* size() – returns the size of the tree
* isEmpty() – checks to see if the size is 0
* put(object) – places a node in the correct spot by traversing to that position (checking IDs)
* get(key) – gets the value that the tree contains and returns it
* remove(key) – removes the node from the tree
* contains(key) – checks to see if the specified key exists within the tree

|  |
| --- |
| Class: BinarySearchTree |
| root: Node |
| size: int |
| BinarySearchTree(): None |
| size(): int |
| isEmpty(): bool |
| put(input: object, key: object): None |
| get(key: object): Node |
| remove(key: object): None |
| contains(key: object): bool |
| print(): str |

**Driver:**

Driver basically inputs the file that it reads from, into the queue, and then uses the queue and an instance of the BinarySearchTree to create and organize the Accounts.

|  |
| --- |
| Class: Driver |
| order: Queue |



File input example:

The file will be inputted by its name using File IO within the driver, and will be read line by line and the strings will be held within the queue named order. Then within the main method, the strings will be parsed, and passed through to an instance of the account class. Then the account class will be placed in the correct slot using the binary search tree (in order). If a withdraw comes through, the binary search tree will look for the intended account, and if found, it will complete the action.

Example:

File name BankTranIn.txt inputted, separated line by line, and placed in a queue.

The queue gets read one by one as the string bellow:

O Guy Ellies 6648

It will be parsed by each space, the first will determine the type of action, and from there it will vary.

If the first is an O, the 2nd and 3rd will complete the name and the 4th index will be the ID.

If its W or D the 2nd will be account ID and fund number and the 3rd will be amount.

If it’s T the 2nd will be account ID and fund number and the 3rd will be the amount and the 4th will be the second ID and fund number.

According the to the first index, the correct method will be called within the account, and will create a chain reaction to the Fund class. If needed, the binary search tree class will be used to see if account exists in the tree and will cause an error if needed.