Abstract Data Types

Gallery

Represents a gallery with registered Users, Works and Auctions, and performs different operations on them.

**Attributes:**

* SepChainHashTable<String, User> users – stores all of the users (and artists), key is the unique login of each user. It is Hash Table because of the large amount of users and it provides us with quick access to each the user object by their login.
* SepChainHashTable<String, Work> works – stores all of the works in the system, key is the unique ID of each work. It is Hash Table because of the large amount of works and it provides us with quick acces to each work object by their ID.
* SepChainHashTable<String, Auction> Auctions – stores all the auctions in the system, key is the unique ID of each auction. It is Hash Table because of the large amount of auction and it provides us with quick access to each auction object by their ID;
* AVLTree<Integer, BinarySearchTree<String, Work>> WorksByVal – stores all the works that have been sold (with value larger than 0), key of the AVL tree is price, and the Value is Binary tree with name of the work as key and work object as value. We have chosen binary tree inside of AVL tree because AVLTree allows us to store work ordered by their price, but furthermore we need to sort them by their name, that is what we are using the Binary Tree for.

**Methods:**

* all the operations described in the section 3.

User

Each object of this class stores information about one user. User can either be artist or just normal user.

**Attributes:**

* String login (unique identifier)
* String name
* Int age
* String email
* Boolean isArtist (determine whether the user is artist)
* String artName (if user is artist)
* BinarySearchTree<String, Work> works (stores the works of the artist, Key is the name of the Work because it is sorted by the name in alphabetical order, Value is a reference to the work object)

**Methods:**

* String getInfo – returns the information about the user as a string (login + name + age + email)
* Every attribute have a setter and getter (except the artName, works)

Work

Each instance of this class stores information about one artwork in the gallery.

**Attributes:**

* String ID (unique identifier)
* String name (name of the artwork)
* Int year (year of creation)
* User author (reference to the author object)
* Int value (highest bided price on the work yet)

**Methods:**

* Every attribute have setter and getter
* String infoWork – returns information about the work as a string (id + name + year + value + author.login + author.getName)

WorkInAuction

Extends work, when a work is added to an auction, workInAuction with a reference to the work is created, storing extra information about the work related to this auction.

**Attributes:**

* Work attributes (because it extends work)
* Work work (reference to the work object it belongs to)
* Int minValue (minimal bidding value)
* DoubleList<Bid> bids (all bids for this work in auction)

**Methods:**

* Void addBid(Bid bid) – adds a bid to the bids
* Boolean close() – sets the highest bid as the value of the work, if there is higher bid than work current value. Return if works value has been changed.

Auction

Each instance of this class stores information and data about one auction in gallery.

**Attributes:**

* SepChainHashTable<String, WorkInAuction> worksInAuction (Hashtable of all the works in this auction, key is the ID of the work, value is the reference to work in auction object)
* DoubleList<Work> works(list of all works in auction, kept in double list because we need them on order they have been added to the auction)
* DoubleList<Bid> bids (List of all bids in the auction);
* String id (unique ID of the auction);

**Methods:**

* void addWorkAuction – adds work to both worksInAuction and works
* WorkInAuction getWorkFromID – returns work in auction based on the ID

Bid

Each instance of this class stores the information about one bid.

**Attributes:**

* Int value (value of the bid)
* User bidder (reference to the user that made the bid)

**Methods:**

* Every attribute has getter and setter

Operations in section 3

addUser:

Checks if user is older than 18, if the provided login is unique. Then creates a new user object and adds the user to the users HashTable. Returns the public user object.

**Time complexity**

(users)SepChainHashTable.find + (users)SepChainHashTable.insert

* + Best case: O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ)
  + Worst case: O(n) + O(n)

addArtist:

Checks if user is older than 18, if the provided login is unique. Then creates a new user object and adds the user to the users HashTable. Returns the public user object.

**Time complexity**

(users)SepChainHashTable.find + (users)SepChainHashTable.insert

* + Best case: O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ)
  + Worst case: O(n) + O(n)

removeUser:

Firstly, check if provided login refers to an existing user, then check if user has any active bids by iterating through all active auction and through every bid in each auction. If user is an artist, checks if he has any of his works in any active auctions, again, iterating through each auctions works. Then finally removes the user, if the user is artist also removes all of his works from works HashTable and from worksByVal AVL tree. Returns the removed public user object.

**Time complexity**

(users)SepChainHashTable.find + iterator creation + iterating through all auctions \* (iterating through all bids in auction + creating iterator)+ (users)SepChainHashTable.remove

If user is artist: + iterating through auctions \* (iterating through every work in auction \* (worksInAuction)SepChainHashTable.find)

If user has works: + iterating through user’s works \* ((works)SepChainHashTable.remove + (worksByVal)AVLTree.remove)

* + Best case: O(1) + O(1) + O(n) \* (O(n) + O(1)) + O(1)
  + Expected case: O(1+λ) + O(1) + O(n) \* (O(n) + O(1)) + O(n) \* (O(n) \* O(1+λ)) + O(n) \* (O(1+λ) + O(log n))
  + Worst case: O(n) + O(1) + O(n) \* (O(n) + O(1)) + O(n) \* (O(n) \* O(n)) + O(n) \* (O(n) + O(log n))

addWork:

Check if provided work ID is unique, if the login is valid, if the login corresponds to an artist,then adds the newly created work object in the works HashTable, also adds the object in the users BST of his works. Returns the public work object.

**Time complexity**

(users)SepChainHashTable.find + (works)SepChainHashTable.find + (works)SepChainHashTable.insert + (user.works)BST.insert

* + Best case: O(1) + O(1) + O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ) + O(1+λ) + O(log n)
  + Worst case: O(n) + O(n) + O(n) + O(n)

infoUser:

Checks if the provided login is valid. Returns the public user object. (for user.getInfo() to be called)

**Time complexity**

(users)SepChainHashTable.find

* + Best case: O(1)
  + Expected case: O(1+λ)
  + Worst case: O(n)

infoArtist:

Checks if the provided login is valid. Returns the public user object. (for user.getInfo() to be called)

**Time complexity**

(users)SepChainHashTable.find

* + Best case: O(1)
  + Expected case: O(1+λ)
  + Worst case: O(n)

infoWork:

Checks if the provided ID is valid. Return the public work object. (for work.infoWork() to be called)

**Time complexity**

(works)SepChainHashTable.find

* + Best case: O(1)
  + Expected case: O(1+λ)
  + Worst case: O(n)

createAuction:

Checks if the provided ID is unique, then creates new instance of auction object and adds it to auctions HashTable. Returns the new public auction object.

**Time complexity**

(auctions)SepChainHashTable.find + (auctions)SepChainHashTable.insert

* + Best case: O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ)
  + Worst case: O(n) + O(n)

addWorkAuction:

Check if the auction ID is valid, if the work ID is valid. If The work has been already added to the auction just returns the public work object. Otherwise creates the workInAuction object based on the existing work object, calls the auctions addWork() method. Returns the public work object.

**Time complexity**

(auctions)SepChainHashTable.find + (works)SepChainHashTable.find + (auction.worksInAuction)SepChainHashTable.find + (auction.worksInAuction)SepChainHashTable.insert + (auction.works)DoubleList.addLast

* + Best case: O(1) + O(1) + O(1) + O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ) + O(1+λ) + O(1+λ) + O(1)
  + Worst case: O(n) + O(n) + O(n) + O(n) + O(1)

bid:

Checks if the provided login is valid, if provided auction ID is valid, if the work is present in the stated auction, if the bid amount above minimum. Returns the public bid object.

**Time complexity**

(users)SepChainHashTable.find + (auctions)SepChainHashTable.find + (auction.worksInAuction)SepChainHashTable.find + (auction.works)DoubleList.addLast

* + Best case: O(1) + O(1) + O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ) + O(1+λ) + O(1)
  + Worst case: O(n) + O(n) + O(n) + O(1)

closeAuction:

Checks if provided auction ID is valid. Iterates through all the works in the auction calling workInAuction.close(), if the value has been changed it is removed and reinserted to the worksByVal AVL tree. If the work wasn’t in the tree yet it is just inserted. Also if the new value isn’t in the worksByVal tree yet, inserting it and creating a new BST tree for it.

Then creates a new DoubleList<WorkInAuction>, iterates through the works in the auction and adds the current work from worksInAuction in the newly created double list. (It has to iterate through the Double list because the worksInAuction isn’t ordered by the time added) Finally returns an iterator of the newly creted DoubleList.

**Time complexity**

(auctions)SepChainHashTable.find + (auction.worksInAuction)SepChainHashTable.iterator(iterator creation) + (iterating through SepChainHashTable \* (works)SepChainHashTable.find \* Worst and average case) + (auction)SepChainHashTable.remove + (creating)DoubleList + (creating)Iterator + iterating through every work(that was in auction ) \* DoubleList.AddLast

Worst and average case: + (worksByVal)AVLTree.find + (worksByVal)AVLTree.remove + (worksByVal)AVLTree.find + (worksByVal)AVLTree.insert + (worksByVal)AVLTree.find + (worksByVal)AVLTree.insert

* + Best case: O(1) + O(1) + (O(n) \* O(1)) + O(1) + O(1) + O(1) + O(n) \* O(1)
  + Expected case: O(1+λ) + O(1) + (O(n) \* O(1+λ) \* X) + O(1+λ) + O(1) + O(1) + O(n) \* O(1)
  + Worst case: O(n) + O(1) + (O(n) \* O(n) \* X) + O(n) + O(1) + O(1) + O(n) \* O(1)

X = O(log n) + O(log n) + O(log n) + O(log n) + O(log n) + O(log n)

listAuctionWorks:

Check if the provided ID of an auction is valid, if the auction has any works in it. And then returns an iterator of the auction’s works. (auction.works.iterator)

**Time complexity**

(auctions)SepChainHashTable.find + (auction.works)DoubleList.iterator(iterator creation)

* + Best case: O(1) + O(1)
  + Expected case: O(1+λ) + O(1)
  + Worst case: O(n) + O(1)

listArtistWorks:

Checks if the provided login is an existing one, if the login belongs to an artist and if he has any works. Returns an iterator of his works. (user.works.iterator)

**Time complexity**

(users)SepChainHashTable.find + (user.works)BST.iterator(iterator creation)

* + Best case: O(1) + O(1)
  + Expected case: O(1+λ) + O(1)
  + Worst case: O(n) + O(1)

listBidsWork:

Checks is provided auction ID is valid, if work is in the auction, if work has any bids. Then return an iterator of the work’s bids. (work.bids.iterator)

**Time complexity**

(auctions)SepChainHashTable.find + (auction.worksInAuction)SepChainHashTable.find + (work.bids)DoubleList.iterator(iterator creation)

* + Best case: O(1) + O(1) + O(1)
  + Expected case: O(1+λ) + O(1+λ) + O(1)
  + Worst case: O(n)+ O(n) + O(1)

listWorkByValue:

Checks if any works have been sold yet. It iterates trough the worksByVal tree and each BLT tree inside, adding the works inside a DoubleList. Returns an iterator of these works. (doublelist.iterator)

**Time complexity**

(creation)DoubleList + AVLTree.iterator(iterator creation) + iterating through the AVLTree \* (BST.iterator(iterator creation) + iterating through BST \* (DoubleList.AddLast))) + DoubleList.iterator(iterator creation)

* + Best case: O(1) + O(1) + (O(n) \* (O(1) + O(n) \* O(1))) + O(1)
  + Expected case: O(1) + O(1) + (O(n) \* (O(1) + O(n) \* O(1))) + O(1)
  + Worst case: O(1) + O(1) + (O(n) \* (O(1) + O(n) \* O(1))) + O(1)

Space complexity

SepChainHashTable<String, User> users (size of)= A

SepChainHashTable<String, Work> works (size of) = B

SepChainHashTable<String, Auction> auctions (size of)= C

AVLTree<Integer, BinarySearchTree<String, Work>> worksByVal (size of) = D

Total space complexity:

O(A) + O(B) + O(C) + O(D) + O(A \* (BST.size(user.works))

+ O(C \* (Dictionary.size(auction.worksInAuction)) + DoubleList.size(auction.works) + (auction.bids)DoubleList.size))

+ O(C \* ((auction.worksInAuction)Dictionary.size \* (workInAuction.bids)DoubleList.size))

+ O(D \* (BST.size \* (workInAuction.bids)DoubleList.size))

Peter Duska and Johanka Jakubove