מיכאל כהן-וונג 230423790
פראג פראג 213857725

Data Structures used:

AVLNode<T> members:

```
AVLNode<T>* leftNode;  // Left child
AVLNode<T>* rightNode; // Right child
int count;             // Number of nodes in the subtree rooted at this node
int height;            // Height of the node
T val;                 // Value stored in the node
int key;
```

Functions:

```
AVLNode(int valueId, T value);
~AVLNode();
void setLeftNode(AVLNode* left);
void setRightNode(AVLNode* right);
int findBalanceFactor() const;
int updateHeight() const;
void fixValues(); // Used to update values after rotation
AVLNode<T>* rotateLeft(); // LL rotation
AVLNode<T>* rotateRight(); // RR rotation
```

Highlight of important functions and members in AVLNode<T> with their time complexity:

int val: holds an object of some type of value, we will use this later to hold ships and pirates

int height: The height of each node is used to calculate the BF of the node which can allow us to figure out if we need to use LL rotation or RR rotation to balance it (because an AVL tree is a self balancing tree).

rotateLeft(): rotates the node LL as taught in class. O(1) complexity

rotateRight(): rotates the node RR as taught in class. O(1) complexity

fixValues(): When we implement the acual AVL tree we'll have to fix the values in the path we went down to figure out which rotations we need to do in response to the new changes. O(1) complexity

Constructor/ initialization: AVLNode(int valueId, T value): Assigning the values of a set number of members happens at O(1) because all the objects it holds are also O(1).

Destructor ~AVLNode(): delete O(1) objects that are deleted at O(1) so O(1) total. (we will see later that the node only holds objects that can be deleted at O(1))

SPACE COMPLEXITY OF AVLNode:

The space complexity of the node is O(1) because it will always hold the same number of objects.

AVLTree<T>, which is composed of AVLNode<T>

This object is designed to be an AVLTree like we learned in class.
Members:
AVLNode<T>* root;
constructor:
AVLTree()
Functions

```
  void insert(AVLNode<T>* node);
  void erase(int key);
  void erase(T check);
  bool empty() const;
  AVLNode<T>* find(int key) const;
  AVLNode<T>* getMaximum() const;
  AVLNode<T>* getMinimum() const;
```

External function: balance()

We'll start with a shortened proof that the height of a balanced tree (which an AVLTree is) has complexity logn:
An AVLTree of height h has a max of two children per node, so its bounded above by a full tree of height h. The number of nodes in a full tree of height h is
1+2+3+4+... + 2^h = n, so we get according to the geometric sum formula

$$n = \frac{1-2^h}{1-2} = 2^h - 1$$

so $h = O(logn)$. As we saw in lecture 4 an AVL tree is bounded by below by a fibonnaci tree, which as we proved in the lecture the height of a fibonacci tree is O(logn) (according to the golden ratio seen through recursion). We'll assume the height of our tree is O(logn) when proving the complexity of the rest of the functions.

Highlights of imporatant functions and members
Initialization/constructor AVLTree(): This sets the AVLNode<T>* root to nullptr. This happens at O(1) time complexity.
Destructor ~AVLTree<T> deletes an AVLTree with n nodes. Each node can be destroyed at O(1) so total destructor complexity is O(n).
AVLNode<T>* root: This is an AVLNode<T> that holds the root of the tree.
void erase(T check); Recursively goes down a path to find the node and removes it, then uses recursion to climb back up the path to update the values and use the balance() function to rotate the tree (meaning it uses the rotate function in the node). (longest path down according to AVL tree invariants is the height of the tree which is O(log n) so we get logn operations to climb down then on the way up logn * complexity of rotate (O(1)) so total is O(logn)

void insert(AVLNode<T>* node): inserts node then balances in similar way to erase function. (longest path down according to AVL tree invariants is log n (proved in class) so we get logn * complexity of rotate = O(logn))
balance(); Uses the get method for BF in each node and determines which rotation is needed (if needed at all) for that particular node.

AVLTree<T> space complexity:
An AVLTree can hold n AVLNodes, where each node has a space complexity of O(1). This means that the space complexity of the AVLTree is O(n), where n is the number of nodes.

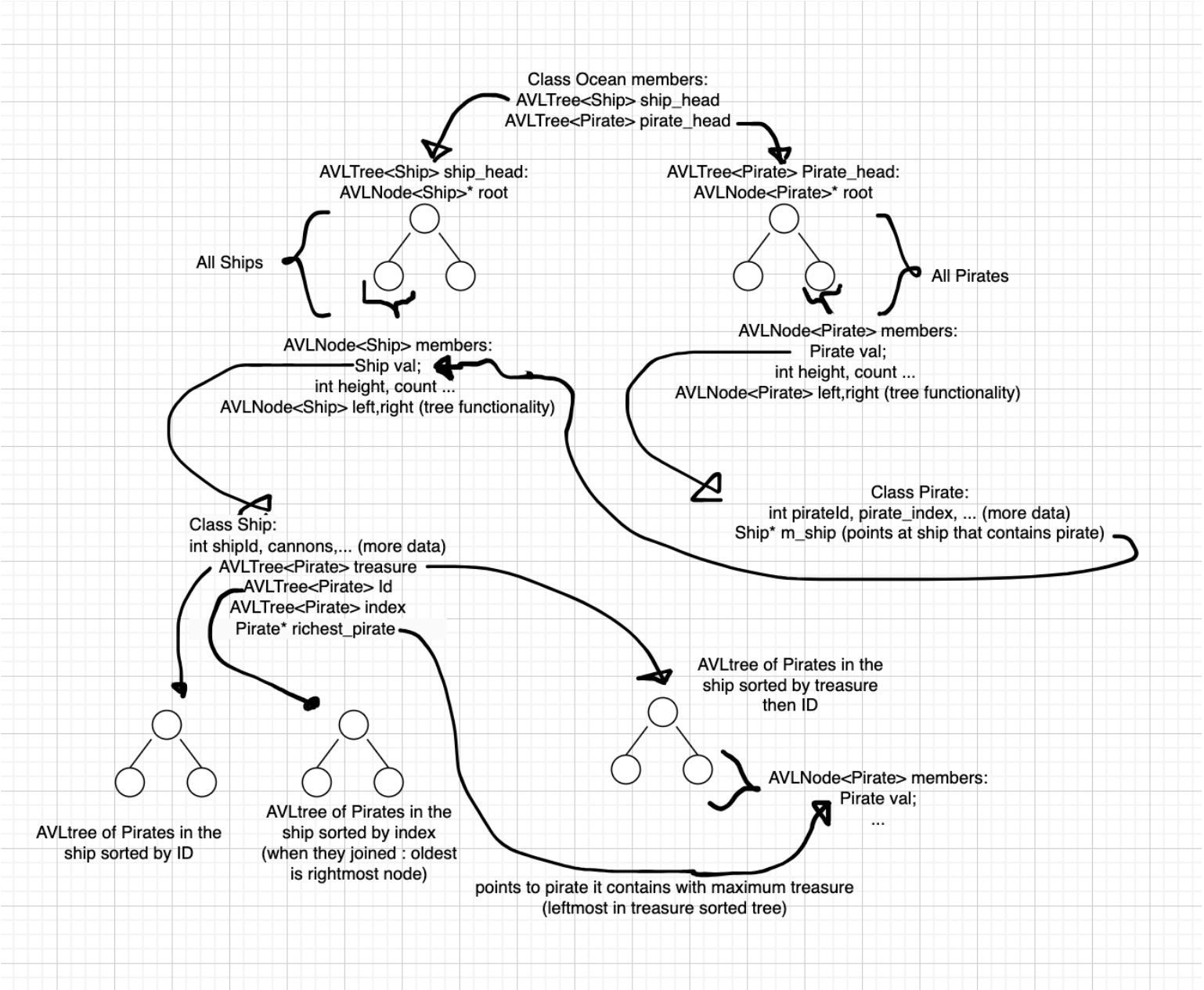
Classes used:
Members of each class:
class Pirate:
//contains the id of the pirate
int m_pirateId;
//contains the amount of treasure a pirate has
int m_treasure;
//contains the index of the pirate in the ship (when it joined the ship)
int m_pirateIndex;
//contains the Id of the ship
int m_shipId;
//contains a pointer to the owner ship
Ship * m_ship;
Space complexity: O(1) (set number of members).
Initialization time: O(1)
Destructor time: O(1)

class Ship
//counts the number of pirates in the ship
int m_counter=0;
//stores shipId
int m_shipId;
//stores number of cannons in the ship
int m_cannons;
//AVL tree of pirates in the ship sorted by m_pirate_index (member of pirate class)
AVLTree<Pirate> pirates_index;
//AVL tree of pirates in the ship sorted by m_pirate_id (member of)
AVLTree<Pirate> pirates_id;
//AVL tree of pirates in the ship sorted by m_treasure () then if treasure is the same is sorted by ID
AVLTree<Pirate> pirates_treasure;

//pirate pointer to richest pirate in the ship
Pirate* m_richestPirate;
//Gives an index to a pirate who just joined
int current_index=-10000;
Space complexity: O(1) (set number of members).
Initialization time: O(1)
Destructor time: Clearly O(n) where n is the number of nodes in each of the trees (all the same because each pirate is inserted into each of the trees). (derived from the AVLTree destructor we described earlier)

class Ocean
//Contains AVLtree of all the ships that is sorted shipIds (m_shipId is a member of Ship)
AVLTree<Ship> ship_head;
//Contains AVLtree of all the ships that is sorted pirateIds (m_pirateId is a member of Pirate)
AVLTree<Pirate> pirate_head;

//drawing of the structure:



OCEAN COMPLEXITY:

Note: For this section we use what we calculated in the AVLTree section: ~AVLTree = O(size of tree), and AVLTree() = O(1)

Space complexity: For this complexity we'll assume that there are n pirates and m total ships. Note that there are total n pirates **split** among with **no pirate appearing on more than one ship** m ships. We'll now count the total amount of space. First theres the AVLTree of pirates, which is pretty straightforward and is O(n) space complextiy as we saw in the AVLTree section (there are n pirates so n total nodes). Next we'll look at the AVLTree of ships.

Here we have an AVLTree of ships, and each ship holds 3 AVLTrees of the pirates it contains. Note that no pirate appears in more than 1 ship, so if we take the sum of the nodes AVLTrees in the ships we'll get 3 times the total number of pirates, which is n (all the ships put together will hold the n total pirates, and each ship holds 3 trees of its pirates so total 3n). Then all that's left is to delete the AVLTree of pirates, which is m. So tota we get O(3n + n + m) = O(n+m)

Constructor Ocean(): This initializes two default AVLTrees (O(1) each) which is total O(1)

Destructor ~Ocean(): As we described in the space complexity, the total amount of deleting in all the trees is O(3n + n + m), because as we mentioned no pirate appears in two ships. So we'll delete the AVLTree of pirates at n complexity, then the sum of the deletions of AVLTrees of pirates in the ships will be 3n. Then finally deleting the AVLTree of all the ships is O(m). So total is O(n + m).

Function implementation and complexity:
Note: In some of the functions we check the validity of inputs and check for allocation failures. These checks amount to constant time so they will not be included in the function calulations.
Note: Due to the structure of an AVL tree, as we proved in the lectures, searching by key happens at O(logn) where n is the number of nodes in the tree.

Note: We'll use the fact that erasing and inserting into a tree (including the balancing) is O(logn) as we proved above and we may omit the explanation in the following explanations.
StatusType add_ship(int shipId, int cannons)
Ocean contains an AVL tree of ships so add ship is simply insert in an AVL tree which is log(m) (m is the total number of ships). Then, we also initialized ship and its members which is constant time so total we get **O(log m).**

StatusType remove_ship(int shipId)
First we need to check that the ship is empty. Then we need to find the Ship in the AVLTree of ships in Ocean (which is sorted by shipId and we are searching through shidId) which is O(logm), so total is **O(logm)**

StatusType add_pirate(int pirateId , int shipId, int treasure)
First we use the given shipId in the AVL tree of ships in Ocean to find and get access to the Ship we needed. This costs O(logm). Afterwards we add one to the counter of the ship, then we take the currentIndex of the ship and put it into the pirates arguments when initializing. Along with the currentIndex we also put the pirateId, treasure, and a pointer to the ship with the given shipId into the constructer of the pirate and they become members of pirate. Then we incremented currentIndex of the ship by 1 so that we don't have repeating indexses. Then

we inserted the pirates into the AVL tree of pirates in ocean (O(logn)), then into the 3 AVL trees in the Ship, which is also O(logn). Then we set the richest_pirate pointer of the ship to the max of the pirate treasure AVLTree (which is O(logn) because the complexity is the height of the tree). So total we got **O(logm + log n).**

StatusType remove_pirate(int pirateId)
First, we searched for the pirateId in the AVL tree of pirates in ocean. Finding the pirate happens at O(logn) because we're searching by the key. After finding the pirate we use the ship pointer it contains (that points to the ship that is holding him) to get to the ship that holds pirate. This happens at constant time. Then using the pointer we an access the 3 AVL trees of pirates that the ship holds and remove them by using the pirates' members as keys to search in the respective trees. We also added comparison operators that allow us to remove the exact pirate from the treasure tree despite more than 1 pirate potentially having the same amount of treasure. Collectively, erasing the pirate from the trees in ship is O(logn). Then, we can remove the pirate from the AVL tree in Ocean (O(logn)). Afterwards we'll set the richest_pirate pointer of the ship (which we have access to from the initial pirate we found) and point it to the max node in the ship's pirate_treasure tree (this search is O(logn) because its simply the height of the tree which we proved was O(log(n))). So in total **O(log n)**

StatusType treason(int sourceShipId, int destShipId)

We use the sourceShipId to find the ship in the AVL tree of ships in Ocean (O(logm)). Afterwards we search for the pirate that entered first by searching for the minimum key in the AVL tree of pirates sorted by index in ship O(logn). This will return the first pirate that joined because the index increases every time a new pirate comes in so a later pirate would have a higher index. We look at the pirates' members and use them as keys to remove the pirate from all the AVL trees in the ship O(logn). Then we add the battleWinnings member in ship to the pirates' treasure (O(1)). Afterwards, we subtract the destShipId's battleWinnings from the incoming pirate O(1) and then insert it into destShip O(log n). We then also give the pirate a new index depending on the current index of the destShip (all the actions above are encapsulated in an add_pirate and remove_pirate). This leaves us with a total of **O(logn + logm)** complexity.

StatusType update_pirate_treasure(int pirateId , int change)
We start by searching for the pirateID in the AVLTree of pirates in Ocean (O(logn)). Then we update its "m_treasure" member by the amount "change" argument. Then we use the ship pointer member in the pirate to go to the ship in O(1) time, then we use the pirates' members to find its instances in the AVLTrees in the ship and change the treasure of the pirate there too (O(logn)). Afterwards, we compare the updated pirates treasure to the current m_richestPirate in the ship, if the updated pirate is now richer than the old richest then we make m_richestPirate point to the new pirate. If the updated pirate has less money then we

do nothing. If they have the same amount then we set m_richestPirate to be a pointer to the pirate with the higher pirateId. Total **O(logn)**

output_t < int > get_treasure(int pirateId)
We start by searching for the pirateId in the AVLtree of pirates in Ocean (O(logn)). Then we use the ship pointer member (O(1) to access its data) to find out the battleWinnings of the ship. Then we return the pirates' written treasure in the Ocean AVLTree and return that plus the battleWinnings of the ship that the pirate belongs to. Total **O(logn)**

output_t < int > get_cannons(int shipId)
We search for the shipId in the AVLtree of ships in Ocean which is O(log(m)). Then we simply get the ship member m_cannons and return that along with a success. Total: **O(log(m)).**

output_t < int > get_richest_pirate(int shipId)
We search for the shipId in the AVLtree of ships in Ocean (O(logn)). Then we look at the m_richestPirate member in ship and return the pirateId of that pirate O(1). Total **O(logm)**

StatusType ships_battle(int shipId1, int shipId2)
We first find the two ships using the given shipId's in the AVLtree of ships in Ocean (O(logm)).
Then we use the members of the ships "m_counter" and "m_cannons" (O(1)) to compute each ship's battle power. After comparing the two battle powers we find out which ship won (or if they tied). In the case that ship1 won (assuming without loss of generality) we add the number of pirates in ship2 to ship1's battleWinnings (O(1)), and we subtract the number of pirates in ship1 from ship2's battleWinnings (O(1)). This is all done in **O(logm).**