Humans vs Zombies Pseudocode

Jeremy Novak (jrn50)

October 2018

1 Data Structure Design

To implement the humans vs zombies code, I decided to store Zombie objects in a red-black tree, which is a variety of binary tree. The benefits of doing this is that it natively supports almost all required functions in $O(logN)$ time (just not bomb). For the pseudo code, I will not be covering the innerworking of a red-black binary tree since this is well documented elsewhere. I will assume that it has the following functions: *get*, *put*, *remove*, $first$, *last*, *ceiling*, $floor$, and *subtree*. These functions are easy to implement in Binary trees and as such can be found in almost every implementation of red-black trees, such as Java's $TreeMap$.

2 Compare (a, b)

For any binary tree, we need to have some way to compare keys. In this case, all functions that look for a zombie other than by searching for a specific coordinate, involve the x-axis. Thus, it makes since to have the red-black tree primarily sort by x-axis, and secondarily by the y-axis.

```
1. if $A_x = B_x$ then return which has the higher y value
2. else return whichever has the higher x value
```

3 Zombie(x, y)

Zombie(x, y) returns the zombie at location (x, y) on the wall or null if there is no such zombie.

```
1. Call get on red-black tree for a key of $(x, y)$ and return value.
2. if no value is found then return null
```

We know this algorithm works because searching is a native function of every red-black tree ADT. Runtime is $O(log(n))$.

4 Insert(z, x, y)

Insert(z, x, y) inserts zombie z at location (x, y) on the wall. It returns an error if a zombie is already present at (x, y) or if z is null.

```
1. if $Zombie(x, y)$ doesn't equal null, return an error
2. else Call put on red-black tree for a key of $(x, y)$ and a value of $z$
```

We know insert works because it is a native feature of every red-black tree ADT. Runtime is $O(log(n))$.

5 Delete(x, y)

Delete(x, y) deletes and returns the zombie at (x, y). It returns an error if there is no zombie at that location.

```
1. $Z \leftarrow Zombie(x, y)$
2. **if** $z = null$, **return** an error
3. **else**
   (a) Call *remove* on red-black tree for a key of $(x, y)$
   (b) **return** Z
```

We know delete works for the same reason insert works, as its code functions in the same way and *remove* is a native feature of red-black tree ADTs. Runtime is $O(log(n))$.

6 Javelin(x_P)

Javelin(x_P) returns the (x, y) coordinates of the zombie that would be killed by the javelin, or null if there is no zombie.The argument represents the player's position. The javelin kills the zombie whose horizontal position is closest to that of the player, irrespective of height.

```
1. $left \leftarrow$ get the key of a $floor$ function on value $(x_P)$
2. $right \leftarrow$ get the key of a *ceiling* function on value $(x_P)$
3. **if** both $left$ and $right$ are null **then return** *null*.
4. **if** one out of $left$ and $right$ is null **then return** the other.
5. **return** the closer of $left$ and $right$
```

We know Javelin works because the closest key will either be greater than, less than, or equal. Ceiling and floor functions find the closest greater than or equal key and the closest less than or equal to key. Thus, the left and right keys represent the two possible closest keys, and then we just manually check the closer of the two. Runtime is $O(log(n))$.

7 Arrow(*direction*)

Arrow(*direction*) returns the (x, y) coordinate of the zombie that would be killed by the arrow. The direction is either left or right. It returns null if there is no zombie. The arrow kills the zombie that is furthest to the left (or the right, at the player's choosing), irrespective of his height.

```
1. **if** direction = left **then return** furthest left key in tree
2. **else return** furthest right key
```

Arrow works because finding the least and greatest keys in a tree is natively supported by red-black trees, as it just constitutes repeatedly taking the left or right node until one reaches the bottom. As a result, runtime is $O(log(n))$.

8 Bomb(x_P, r)

Bomb(x_P, r) returns the (x, y) coordinate of the zombie that would be killed by a bomb of range r launched by a player at coordinate x_P. It returns *null* if there is no zombie in range. The bomb kills the zombie that is the highest on the wall among all of the zombies within a given horizontal distance r from the player's position. In other words, if the player horizontal position is x_P, the bomb kills the highest zombie whose horizontal coordinate is between $x_P - r$ and $x_P + r$.

```
1. call subtree function to get all keys greater than $x_P - r$ and less than $x_P + r$ in the tree.
2. Convert subtree to a set $S$ than can be iterated through
3. $H \leftarrow null$
4. for each key $K$ in $S$
    (a) if $K_y > H_y$ then $H \leftarrow K$
5. return $H$
```

We know bomb works because it simply compares every key in the range of the bomb and selects the one with the highest y-value. Runtime on this function is the slowest, but is bounded by a worst case runtime of $O(n)$.