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 an array of nodes, which I'll call A, where each node N contains a Zombie, its x and y values, and a link to another node. Each index of the array represents a range of n values, each separated by a distance of d. For example, a Zombie with $x = x_n$ would be put into $A[\lfloor x_n/d \rfloor]$. In addition, each node stored in an array index is at a higher position that all nodes it links to. The reasoning for this is that we need to have some list component so adding and removing elements is easier but this abstract data type also allows us to jump to a given x position, which must be done for a number of the operations required. Sorting by height in each index allows us to cut down on the runtime of Bomb.

2 **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. Select the array index that a zombie with this x-value would be stored in. Refer to the node (or null)
   stored in this index as $N$.
2. while $N$ exists and $N_y \geq y$
    (a) if $N_x = x$ and $N_y = y$ then return $N_{zombie}$
    (b) else $N \leftarrow N_{next}$
3. return $null$
```

We know this algorithm works because it cycles through every zombie not below the specified height, in the array index that we know will store the zombie due to its x index. Average case runtime is $O(\frac{z}{n})$, where z is the number of zombies alive and n is number of array indices chosen.

3 **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 $z = null$ then return an error
2. Select the array index that a zombie with this x-value would be stored in, or $A[\lfloor x/d \rfloor]$. Refer to the
   node (or null) stored in this index as $N$.
3. if $N = null$ then $N \leftarrow$ a new node where $zombie \leftarrow z$, $x \leftarrow z_x$,$y \leftarrow z_y$, and $next \leftarrow null$.
4. loop until break
    (a) if $N_x = x$ and $N_y = y$ then return an error
    (b) if not ($N_{next}$ exists and $N_{next_y} \geq y$) then break
    (c) else $N \leftarrow N_{next}$

5. $N'_{next} \leftarrow$ a new node where $zombie \leftarrow z$, $x \leftarrow z_x$,$y \leftarrow z_y$, and $next \leftarrow N_{next}$.
```

We know insert works because it looks in the correct array index to store x and and cycles through every zombie above or equal to y to ensure there is not another zombie at (x,y). The infinite loop and break statement works to reduce repeated code which could not be abstracted away. Average case runtime is $O(\frac{z}{n})$, where z is the number of zombies alive and n is number of array indices chosen.

4 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. Select the array index that a zombie with this x-value would be stored in, or $A[\lfloor x/d \rfloor]$. Refer to the
   node (or null) stored in this index as $N$.

2. if $N_x = x$ and $N_y = y$
    (a) $Z \leftarrow N_{zombie}$
    (b) $N \leftarrow N_{next}$
    (c) return $Z$

3. loop until break
    (a) if not ($N_{next}$ exists and $N_{next_y} \geq y$) then break
    (b) if $N_{next_x} = x$ and $N_{next_x} = y$
        i. $Z \leftarrow N_{zombie}$
        ii. $N_{next} \leftarrow N_{next_{next}}$
        iii. return $Z$
    (c) else $N \leftarrow N_{next}$

4. return an error
```

We know delete works for the same reason insert works, as much of the loop works the same way. Average case runtime is $O(\frac{z}{n})$, where z is the number of zombies alive and n is number of array indices chosen.

5 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. $i_{left} \leftarrow \lfloor (x_P - \frac{d}{2})/d \rfloor$

2. $i_{right} \leftarrow \lceil (x_P - \frac{d}{2})/d \rceil$

3. while $A[i_{left}] = null$ and $A[i_{right}] = null$
    (a) if $i_{left} = 0$ and and $i_{right} = |A| - 1$, then return null
    (b) if $i_{left} > 0$ then decrement $i_{left}$
    (c) if $i_{right} < |A| - 1$ then increment $i_{right}$

4. $N_{left} \leftarrow A[i_{left}]$, $N_{right} \leftarrow A[i_{right}]$

5. if $N_{left}$ exists then $N_{closest} \leftarrow N_{left}$ else $N_{closest} \leftarrow N_{right}$
```

```
6. while $N_{left}$ exists
    (a) if $N_{left_x}$ is closer to $x_P$ than $N_{closest_x}$ then $N_{closest} \leftarrow N_{left}$
    (b) $N_{left} \leftarrow N_{left_{next}}$
7. while $N_{right}$ exists
    (a) if $N_{right_x}$ is closer to $x_P$ than $N_{closest_x}$ then $N_{closest} \leftarrow N_{right}$
    (b) $N_{right} \leftarrow N_{right_{next}}$
```

Javelin works by finding the left and/or right closest not-null columns to x_P and finding the closest node out of both columns. Average case runtime is $O(\frac{z}{n})$, if we expect an even distribution of zombies, but worst case runtime would be $O(z * n)$.

6 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. let $m$ and $n$ be two indices in A.
2. if direction = left then $m \leftarrow 0$ and $n \leftarrow |A|$
3. else $m \leftarrow |A|$ and $n \leftarrow 0$
4. for all integers $i$ in $m$ to $n$
    (a) if $A[i]$ exists
        i. $N_{closest}$,N$\leftarrow A[i]$
        ii. while $N$ exists
            A. if $N_x$ is closer to $(m * d)$ than $N_{closest_x}$ then $N_{closest} \leftarrow N$
            B. $N \leftarrow N_{next}$
        iii. return the coordinates of $N_{closest}$
5. return *null*
```

Arrow works by starting at the furthest left/right column and moving in. The first not null column it finds, it finds the furthest from the center node and returns its coordinates. Average case runtime is $O(\frac{z}{n})$, if we expect an even distribution of zombies, but worst case runtime would be $O(z * n)$.

7 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. $N_{highest} \leftarrow null$
2. for all integers $i$ between $\lfloor (x_P - r)/d \rfloor$ and $\lceil (x_P + r)/d \rceil$
    (a) if $A[i]$ exists
        i. $N \leftarrow A[i]$
        ii. while $|x_P - N_x| > r$ and $N$ exists, $N \leftarrow N_{next}$
        iii. if $N$ exists and either ($N_{highest} = null$ or $A[i]_y \geq N_{highest_y}$) then $N_{highest} \leftarrow A[i]$
3. return the coordinates of $N_{highest}$
```

Bomb works by cycling through the range of all columns which could include nodes inside of the affected range and finding the one with the highest node. This is easy since the top stored node is always the highest node, but we also have to ensure that the top node is actually in the range. The average case runtime is roughly $O(r)$.