Cpt\_S 223

Group: Freya Varez, James Ponce

**Report – PA4**

**Problem Statement**:

The goal of the exercise was to design and implement a ‘board simulator’ program with the following characteristics:

* The ‘board’ is an square matrix (where ). Note that the implementation of the board is purely conceptual within the program (see *algorithm design)*.
* Within the simulator there are players (where and ). is specified by the user and may change during runtime. Each player has a unique integer *ID*. Similarly, each player construct occupies a unique cell within the bounds of the game board mentioned above.

During runtime, the simulator must support the following functions:

* **Insert():** Loads a new player into the simulator, taking care to follow the above characteristics of the board – meaning an appropriate error message will print to the console when a players insertion fails.
* **Remove():** Deletes a player with a specified ID provided by the user, again printing an appropriate error message in the case of a failed removal.
* **Find():** Searches for a player with the specifies ID provided by the user, returning true is the player is found, false otherwise.
* **MoveTo():** Searches for a player specifies by the user, and moves them to a location on the board (again, specified by the user), granted this location is in the same row, column or diagonal to the players initial location.

(Note: if a player is moved to an occupied space, the occupying player is deleted and an appropriate message displayed through the console).

A program with the above specifications was designed and implemented to optimize runtime and memory usage; taking into account that the size of the board () and player count () may be especially large (tested up to ).

**Experimental Setup:**

The program was compiled and experimentally tested using a Unix system through the EECS servers and using a g++ compiler.

The algorithm design for search() was efficient enough that for 1000000 the runtime was <1 microsecond (for 1000000) and thus could not be measured. This was contrary to the theoretical time complexity, however this may have been due to the algorithms design (see *Algorithm Design*).

Due to being unable to test the runtime complexity of search(), the runtime of printByID() was used instead – being that it implements a compilation of similar functions (see *figure* 1). The printByID() experiment was run 56 times on various inputs of and (*see table 1* and respective *figure 2*).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| size m | 100 | 500 | 1000 | 5000 | 10000 | 100000 | 1000000 |
| 1 | 14 | 46 | 39 | 74 | 117 | 727 | 6387 |
| 2 | 12 | 19 | 33 | 65 | 70 | 576 | 5654 |
| 3 | 28 | 124 | 38 | 62 | 82 | 608 | 5734 |
| 4 | 29 | 20 | 28 | 112 | 77 | 592 | 5649 |
| 5 | 18 | 57 | 25 | 76 | 88 | 584 | 5734 |
| 6 | 16 | 80 | 30 | 62 | 79 | 586 | 5720 |
| 7 | 11 | 27 | 32 | 100 | 73 | 600 | 5769 |
| 8 | 11 | 56 | 87 | 80 | 86 | 585 | 6674 |

Table 1: PrintByID() samples (time in microseconds)

*Figure 2: Testing runtime of printByID() (time in microseconds)*

*Figure 3: Smaller sample sizes of table 1 (time in microseconds)*

**Algorithm Design:**

The overall program was designed with the possibility of large game board and number of players in mind. Thus, all algorithms had to balance an optimized runtime and memory usage. However, the program also takes into account that in most cases the number of players () is much less than .

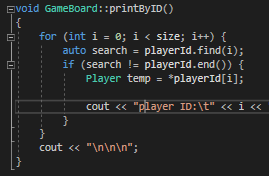
Thus, the game board was implemented conceptually in the form of integer coordinates within each player node. This was to avoid the memory strain of loading a full game board (likely through the use of a 2D array) and also to optimize the search() algorithm below. Similarly, each node was placed into a std∷map container by both ID and position. The balancing mechanisms for each map was done through the built in algorithms; these are implementation dependent, but usually rely on Red-Black or AVL tree algorithms.[[1]](#footnote-1)

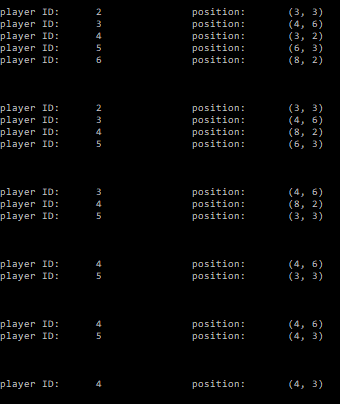
* **Insert():** The insert algorithm was basic and relied upon the search() algorithm below to determine if the new nodes destination was currently occupied.
* **Remove():** Again, the design of the deletion algorithm was simple, since no direct board was implemented, a node could be deleted without any effect.

*Note:* The *key* for the above map implementation was based on the (x,y) position within the board. This takes into account that each player holds a unique position.

* **Find():** Find was optimized using a map, which requires a to search based on ID as a search key.
* **MoveTo():** This algorithm simply implemented the remove() and insert() algorithm above with the slight change that upon insert() into an occupied position – the occupying player is in turn deleted (as opposed to simply printing an error message to the user and cancelling the move).

*Figure 1: PrintByID() algorithm design*





*Figure 3: Test output*

1. *Std::map Insert() Algorithm*, n.d., http://www.cplusplus.com/reference/map/map/. [↑](#footnote-ref-1)