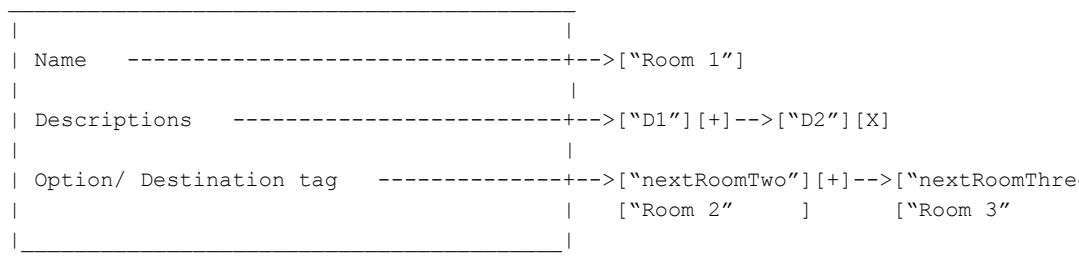
David Sun
1357057
CMPS 12B/M
12/7/2014
design.pdf
This file includes the explanation for my implementation of the cyoa program.

For my implementation of the cyoa project, I decided to use an array to store all the rooms, a linked queue of type string to store all description tags, and a linked queue of type String[] of size 2 to store all options and destination tags. I also implemented a stack in order to account for the undo option.

For an arbitrary Room object, with name "Room 1," description tags "D1" and D2," and finally option nextRoomTwo followed by Room 2, and option nextRoomThree followed by Room 3, can be visualized with the diagram below:



When cyoa is run, the program first reads how many rooms there are in the file given by the number of "r [roomName]" commands. An array of rooms is created based on how many rooms there are.

The program then scans through the entire file, generating new rooms. For each "r [roomName]" command, the room array index is incremented, assigning a new room.

For each "d [description]" command, the program appends the description to the description queue.

For each "o [option]" followed by "t [tag]" command, the program creates a String array of size 2, and adds the option to the first index, and adds the tag to the second index. Finally, the String array is appended to the Option/Tag queue.

It takes O(1) steps to append a new room to the room data structure since we are inserting into an unordered array at first. After all the rooms have been read in without error, a Merge-Sort algorithm is applied to the room array. The rooms are sorted into lexicographical order based on their names. The best, average, and worst case of Merge-Sort is O(n log n). This way, searching for room indexes can have a guaranteed O(log n) search time in the worst case.

Though binary search trees may very well have yielded great insertion and search results, suppose the adventure file had names ordered in such a way that each following room had a name that is lexicographically greater than its previous room. Then, the tree would degrade into essentially a linked list whose search and insertion would have a O(n) running time. Also, since we are given a constant number of rooms in the adventure file, there is no need to worry about increasing the size of the array.

I thought about this assignment in terms of large data sets. Suppose we were given an adventure with a million rooms. Initially it would take relatively longer to sort all the rooms. After sorting, however, we need not sort the array again. Searching for rooms would require O(log n) time in the worst case, and since we're doing multiple searches, a running

time of O(log n) is excellent compared to that of an algorithm which takes O(n) steps. If we had implemented a binary search tree instead, that tree would have had the potential to degrade into a linked list, which would require O(n) operations for insertion AND searching, and therefore take much longer than a O(n log n) running time followed by guaranteed O(log n) searching.

I found that queues were appropriate for description tags as we merely insert the description without performing any actions on it later.

In my opinion, queues were also appropriate for Options/Destinations because each room may have AT MOST 12 Options/Destinations, and it takes minimal time to linearly search through twelve elements.

Finally, to account for the undo command, I implemented a stack. After the user enters a command that corresponds to an option, the index of the current room in the array gets pushed onto the stack. This way, if the user decides to undo, the stack data structure calls the pop method and the room index returns to the popped value, or the index of the previous room. If the stack is empty and the users tries to undo, a NoSuchElementException is thrown, which is caught by the main program.