Tries:
Implementaion: Each node contains an arraylist, each slot corresponding to a character as a node.

1. insert() :
-> returns true if the object doesn't exist already and hence inserted else return false
-> implemented iteratively traversing down from root along the characters of the key string
-> time complexity = O(String_length)

2. delete() :
-> returns true if delete is successful else it returns false.
-> implemented iteratively by first finding the node storing the object (if it exists) else returning false. It is then traversed upwards towards the root to delete the corresponding nodes if they don't effect existence of other objects/ nodes.
-> time complexity = O(String_length)

3. startswith():
-> returns the node associated with the last character of the given prefix.
-> implemented iteratively by traversing down from the root using characters of the given string.
-> time complexity = O(String_length)

4. printTrie():
-> prints all words associated with the given node as prefix

-> implemented recursively by using the fact that printTrie(prefix) <=> printTrie(startswith(prefix + ch)) where ch is any possible character (including empty character). Recursion is terminated if no further character isn't possible ahead of the prefix.
-> time complexity = O(no. of nodes)

5. printlevel():
-> prints a given level of trie assuming node to be at level 0
-> implemented using dfs of trie. Each node with level equal to the given level is concatenated to a ArrayList, which is finally sorted and printed.
-> time complexity = O(no. of nodes)

6. print():
-> prints all levels of trie.
-> implemented by iteratively calling printlevel until the last level is reached.
-> time complexity = O(no. of nodes ^ 2)

7. search():
-> searches for the object with given key
-> implemented iteratively going down from the root along the characters of the string and checking for existence of the required object at the node.
-> time complexity = O(String_length)

RedBlack Trees:
Implementation: Each node has a characteristic colour, associated key, associated values related to the key (in an arraylist).
Follows all the rules discussed in the lectures (equal black depth, red parent no red child, null being black and black root)

1. insert():
-> inserts a new key, value pair in the tree
-> Done iteratively by first finding the spot to be inserted at in case of a new key, a value is simply added to the node for an existing key.
New node is coloured red by default (unless it is the root). Three cases are then considered
Black parent: no further steps needed
Red parent - Red uncle: one tri node restructure done (using a helper function)
Red parent - Black uncle: recoloured the parent, uncle and grand-parent. The three cases are then checked again for grand parent unless we reach the root.
-> time complexity = O(log(no. of nodes))

2. search():
-> Searches for elements with a given key.
-> Implemented iteratively by going down the root using comparison of the key with each node visited until either the required node is found or null is reached.
-> time complexity = O(log(no. of nodes)

PriorityQueue/ Heap:
Implementation: Using an arraylist such that it follows that children_indeces = 2*parent_index + {1 or 2}
Priority for each element has been modified by maintaining a count of the no. of elements inserted before the given node. This ensures unique priority for all elements.

1. insert():
-> inserts a new element with a given priority.
-> implemented iteratively by first adding the new element at last and then bubbling it up by comparison with the parent until a greater parent is found or index = 0 is reached.
-> time complexity = O(log(no. of elements))

2. extractmax():
-> returns the element with the highest available priority.
-> implemented iteratively by first returning the element at index = 0 (if any else terminated) and replacing it by the last element. Size of arraylist is then decreased. This element is then compared with larger of its children and exchanged in case of mismatch until appropriate position is found or last index is reached.
-> time complexity = O(log(no. of elements))

JOB SCHEDULER:
Implementation: Using the given functions and driver class.
Several data structures created, namely jobs -> a heap (remaining jobs) and 2 array lists (finished and unfinished jobs) and an RBTree (all jobs)

(priority in the heap is determined by first the corresponding project priority and then the input time

in case of a conflict. This time is maintained separately in the driver class, incremented after every call to handle_job(), and assigned to the concerned job object.)

projects -> a trie

users -> a trie

1.handle_project():
-> inserts a project.
-> implemented using an allProjects trie in which insertion is possible only with uniquely named projects.

2.handle_user():
-> inserts a user.
-> inserts user into the trie if it doesn't exist already.

3.handle_job():
-> inserts a job.
-> implemented by inserting the job into the heap and the RBtree(job name as key) after checkong existence of corresponding user and project.

4.handle_query():
-> prints the status of queried job
-> implemented using allJobs RBtree, BST search is used in particular.

5.schedule():
->helper function which extracts elements from the heap until one which can be done is found.
->The elements which can't be done are added to the unfinishedJobs list.

6.handle_empty_line():
-> executes the next job with highest priority (given that it can be done).
-> implemented using the heap, extracting elements until one which can be done is encountered. (used schedule() as helper)

7.handle_add():
-> adds to the budget of a given project
-> all the unfinished jobs corresponding to the given project are brought back to the priority queue (as they may be doable now that the budget has increased)

8.run_to_completion():
-> removes all the remaining elements from the heap and puts them in finished/ unfinished lists as needed.
-> heap is now empty with all jobs in either of the two lists (besides each being in the RBtree)

9.print_stats():
-> prints all stats of the system at termination.
-> finished list already consists all the elements in the order in which they were executed, hence giving the corresponding stats.
For the unfinished elements, they are inserted back in the heap(which was initially empty) and then extracted one at a time, in order to print them according to their priority and time of inclusion.

END, 2018EE10433, ACHINT KUMAR AGGARWAL
