Assignment 4

Subhalingam D

October 3, 2019

1 Tries

1.1 `Person`

Contains the name and phone number of a person. The `getName()` returns the name of the person and `toString()` is overridden to return a string of required format.

1.2 `TrieNode`

The children are stored in `ArrayList` of size 95. It also stored the value at the leaf. It has some trivial methods and a constructor:

- `TrieNode()`: Initialises the value and all children to null
- `setValue()`: Sets the value
- `getChild()`: Sets the child of the given character
- `makeChild()`: Make a child of the given character
- `hasChild()`: Returns if it has a child (when there is a child that is not null)
- `hasChildOtherThan()`: Checks if there is a child other than given child (index)
- `killChild()`: Sets the child (index) to null
- `removeValue()`: Removes the value
- `getChildByIndex()`: Gets the child by given index (number)
- `getValue()`: Gets the value

Note: When character is bassed, it's index is determined by a simple mapping: $h(i) = i - 32$..

1.3 `Trie`

The `root` is the beginning of a `Trie`.

1.3.1 `insert()`

Maintain a `curr` node initialised to `root`. For $0 \leq i < size$ (where $size$ is the length of the string to be added), if `curr` does not have the child for character at $(i+1)^{th}$ position in the word, make one. Then make `curr` point to that child. After $size$ iterations, the loop is terminated and `curr` has reached the end for this word. Now, if there is a value already at this place, return false; else set the value and return true.

1.3.2 `delete()`

Start from `root`. For $0 \leq i < size$ ($size$ being the length of the string), if there isn't a corresponding child, then the given word doesn't exist. If there is a value or other children while travelling down, make a note of this (we will be needing the last such occurrence). After the loop gets terminated, check if there is some `value` at this node-if not there, return not found. If there is/are further child(ren) for this node, just remove the `value` present; else remove the corresponding child from the node whose occurrence we had noted above.

1.3.3 `search()`

Starting from `root`, travel down the `Trie` for length of the search string. While travelling, if a corresponding child is not present, it means that our search string isn't present. Return accordingly. If after the end of travel, if there is no `value` at this Node, then the search string is still not there. If everything has been passed, then return the `value`.

1.3.4 `startsWith()`

Similar to `search()`, travel down and return the matching node (of the last character of the string) or `null`. Then `printTrie()` is called.

1.3.5 `printTrie()`

Do a recursive on this function with the Node `trieNode`, as follows:

- if `trieNode` is `NULL`, return
- if `trieNode.getValue()` is `NOT NULL`, print this `value`.
- Call `printTrie()` on each of the children.

It is a basic Depth-First Search.

1.3.6 `print()`

Maintain two `ArrayList` of `TrieNode`, say t_1 and t_2 and another `ArrayList`, say c of characters. Add `root` to $t2$ and have a loop until t_2 becomes empty. Print "Level x" on console ($x = 1$ at the beginning and incremented at each loop. Make $t_! = t_2$ and t_2 to new object. For $0 \leq j < 95, 0 \leq it_2.size()$: Get $(j+1)^{th}$ child of $(i+1)^{th}$ index of t_1. If `NOT NULL`, add it in t_2 and add the ASCII value of $j + 32$ to c (if $j \neq 0$). After the loop gets terminated, print all the elements in c by seperating them with commas. Reset c and print a new line.

1.3.7 `printLevel()`

Do a similar thing as `print()` but add `x` $\leq$ `l` (l is the l^{th} level we need) along with $t_2.size() > 0$, in the loop condition. Also print only the level required, rather than printing all the levels.

2 Red-Black Tree

2.1 Introduction

We use the real world blood relations to say the relation between two people in the tree. For example, two children of the same parent are sibling, the parent's brother is uncle and so on. A Red-Black Binary tree is balanced by definition. Some of it's properties are:

- Every Node has a colour either Red or Black
- Root is always black
- There are NO adjacent Red Nodes,i.e., both parent and it's child can't be Red
- The black depth remains the same in any path from the root to leaf node/`NULL`

To preserve these properties, we need to do some operations namely recolouring (as the name says, changing colours) and restructuring (via Left/Right Rotation(s))

2.1.1 Left Rotation

Let G be a node, P it's right child, X it's right-right grandchild (assuming everyone exists or can be obtained by a similar approach for other cases). By Left Rotation at G, we make the left child of P (if it exists) as the right child of G, G as the left child of P and G's ex-parent as the parent of P. This is the same result as seen by visualisation.

2.1.2 Right Rotation

Let G be a node, P it's left child, X it's left-left grandchild (assuming everyone exists or can be obtained by a similar approach for other cases). By Right Rotation at G, we make the right child of P (if it exists) as the left child of G, G as the right child of P and G's ex-parent as the parent of P. This is the same result as seen by visualisation.

2.2 `RedBlackNode`

This is the Node for the `Red-Black Tree`. Each `RedBlackNode` has a key, list of values, reference to left child, right child and parent, the colour (Red/Black). It has the following (trivial) functions:

- `addData()` adds data at the key
- `setColor()` sets the specified colour
- `chnageParent()` changes the parent
- `makeLeftChild()` makes a left child
- `makeRightChild()` makes a right child
- `getKey()` gets the key
- `getParent()` gets the parent
- `getLeftChild()` gets the left child
- `getRightChild()` gets the right child
- `getColor()` gets the color
- `getValue()` gets the first value from the list of values
- `getValues()` gets the list of values present

2.3 `RBTree`

2.3.1 `insert()`

We first perform a normal BST insertion.

- if root is empty, insert at node, make it *Black* and we are done
- until you find an empty node, keep going down depending on the key given, i.e., if key is smaller than the node we are at, then we go left, if it's greater, then right. If we find a key match while traversing, we just add the value at that node and we are done.
- We should have reached a null node by this time. So insert the new node as the current parent's left (if key given is smaller than parent) or right child.

Now there is possibility of having adjacent nodes *Red* (which is a violation of the property of Red-Black Tree, listed in 2.1). Basically the following cases are possible.

	Grandparent	Parent	Uncle	Resolution
i	ANY	Black	ANY	No problem and we are done
ii	Black	Red	Red	Recolouring
iii	Black	Red	Black	Restructuring (by rotation)
	Red	Red	ANY	NOT POSSIBLE (violation of property)

Note that `NULL` nodes are coloured *Black*.

Case ii: Change colour of Parent and Uncle (to *Black*), Grandparent to *Red*. Continue check (Recurrence) from Grandparent (i.e., assume it is newly inserted and follow the table again) until we arrive at the root or the first generation (child(ren) of root) of this family.

Case iii: Let X be the child who is creating problems, P be his parent and G his grandparent. The following cases arise:

- P **is Left child of** G**,** X **is Left child of** P (Left-Left): Right Rotation at G.
- P **is Left child of** G**,** X **is Right child of** P (Left-Right): Left Rotation at P, then Right Rotation at G.
- P **is Right child of** G**,** X **is Left child of** P (Right-Left): Left Rotation at P, then Right Rotation at G.
- P **is Right child of** G**,** X **is Right child of** P (Right-Right): Left Rotation at G

After rotations, make the oldest person *Black* and the middle generation *Red*. The problem will be solved in just one restructuring unlike the recolouring case (which can take upto $\log n$ steps in loose terms).
Note that Left/Right Rotation has been defined already in 2.1.1, 2.1.2. At then end, we recolour the root (just incase) to `Black` to maintain the property that root is always `Black`

2.3.2 `search()`

Start at root. Go all the way down till `NULL` is met. We go left if query key is smaller than the current node or right if its larger. If the key matched at some point during traversal, return the values at that point iself and we are done. If the loop gets terminated after meeting the `NULL`, it means that the key isn't there and we are done.

3 Priority Queue

3.1 `Student`

Test class that stores name and marks of the Student.

- `compareTo()` compares the marks of the student with the marks of the student object that is being passed and returns negative value if the student's marks is less than that of the passed object, 0 if equal or a positive value if greater
- `toString()` is overridden to print output in desired format.
- `getName()` returns the name of the student.

3.2 `MaxHeap`

`token` is used to set timestamp (explained below) and a count of number of elements in the `MaxHeap` is also maintained (initialised to 0). `ArrayList` is used to maintain the queue.
Let i denote the index in the `ArrayList` (starting from zero). The following observations can be made:

- The index of it's parent (other than for root) is $[\frac{i-1}{2}]$ (where [.] is floor function)
- The index of it's left child, if exists, is $2i + 1$
- The index of it's right child, if exists, is $2i + 2$

3.2.1 The class `Wrsp`

`Wrap` encapsulates the element's class and a timestamp (to store the order of insertion and perform operations on those objects with same priority accordingly.
`compareTo()` compares two objects of given object, returns negative number if the object has less priority than the passed object, positive number if greater. In case of equality, the timestamps are compared and the one with lower timestamp (came first) is said to have higher priority. A total order is hence maintained.

3.2.2 `insert()`

Insert the given element after the last element in the queue and increase the size of queue and the token number (used as timestamp). Until the inserted element becomes root or the parent is has a (strictly) higher priority than the inserted element, keep swapping both of them and travel upwards. The Max-Heap property will be preserved after the loop is terminated.

3.2.3 `extractMax()`

This method returns the element with the largest priority (the root). To preserve the left-complete property of heap, we replace the root with the last element and preserve the Max-Heap property by travelling downwards.

- if there is only one child to the current node and the node has a lower priority than it's child, the MaxHeap priority property is violated; so we swap them and move down. Else (i.e., the child has a lower priority than the current one), we are done.
- else (there are two children). Get the child with higher priority (among the two children); compare them with the current node (that was replaced to the root). If it has a lower priority, swap it with the child with higher priority. Otherwise, we are done.

We follow the above steps until the node has reached the maximum depth (or has breaked out of the loop).
(Note: Since a total order is being maintained, no two elements can have same priority)

3.2.4 `isEmpty()`

Returns true if queue is empty; false otherwise.

4 Project Management

4.1 `User`

Every `User` has a name. `compareTo()` makes a lexicographical comparison between the names of another user (passed) and returns negative if the object's name is before the passed object's name (in dictionary), 0 if equal, positive integer otherwise.

4.2 `Project`

Every `Project` has a name, budget and priority. The more is the priority value, the more preference is given to it for completing a `Job` in it.
`addBudget()` adds (more) budget to the `Project`

4.3 `Job`

Every `Job` has a name, the `User` who has made it, the `Project` to which it belongs to and a runtime. Along with these, it also has a unique ID which preserves the order in which the `Jobs`s were added.
The `compareTo()` returns -1 if the object has a lower priority than the one that is passed, 1 if it's higher and in case of equal priority, a total order is maintained by returning 1 if ID of the object is lesser than the one passed (i.e., the former came first) and -1 otherwise.

4.4 `Scheduler`

All the `Projects` are stored in a `Trie`, `Job` as a `MaxHeap` (Priority Queue) and a `Trie`, list of `User`s in a `RedBlack Tree`, expansive (and hence, unfinished) `Job`s and completed jobs in a Queue. A ID Generator is used to generate unique IDs to a `Job` and a count of Pending `Job`s are also stored.

4.4.1 `handle_user()`

The given `User` is inserted to the Tree maintained.

4.4.2 `handle_project()`

Insert the given `Project` in the `Trie`.

4.4.3 `handle_job()`

Check if the given `Project` and `User` for the given `Job` is present. If not, give an error. Otherwise, add the `Job` to the `MaxHeap` and the `Trie`.

4.4.4 handle_query()

Check if the given Job is present in the JobList (Trie). If it is not even found, then the Job doesn't exist. Otherwise, check if it is present in the Queue of Completed Jobs-if yes, then the Job has been completed, else not completed.

4.4.5 handle_add()

Check for the Project in the Trie. If it doesn't even exist, print it and return. Make a new MaxHeap and a new Queue. We then remove all expensive Jobs and add those Jobs that belong to the Project whose budget has been updated to a new MaxHeap and the rest to the new Queue. All the other pending Jobs (which are already in a MaxHeap) is removed and added to the new MaxHeap created. The new Queue created above becomes the Queue of expensive Jobs and the new MaxHeap becomes the Priority Queue of all jobs that has to be executed.

4.4.6 schedule()

Each time the schedule() is called, the following happens:

- Remove the Job with the highest priority
- if there is no Job to be executed, then leave it
- if the runtime of this Job is more than the budget of the Project (to which it belongs to) , add this jobs to the list of ExpensiveJobs and go to Step 1 again
- else reduce the budget of the Project and add the Job to the list of CompletedJobs and return

4.4.7 handle_empty_line()

When an empty line is encountered, one Job is scheduled through schedule() method.

4.4.8 run_to_completion()

If we have reached the EOF (End of File) of the Input file, all the remaining Jobs are executed by calling schedule() continuously until MaxHeap containing the Jobs becomes empty.

4.4.9 print_stats()

Print the stats of various Jobs. First print all the completed Jobs and end time (by adding the runtime of each Job). Then print all the unfinished Jobs (easy to see that these are the ExpensiveJobs). These Jobs do not have a endtime as it wasn't completed.

5 Time Complexity

5.1 Tries

The insert(), delete() and search() all have a time complexity of $O(k)$ where k is the length of the key used as key. The startsWith() is also $O(k)$ but for printTrie(), which is a Depth-First Search would be $O(hn)$ where h is the average height (average number of characters in a key) and n being the number of keys and hence prefix-searching has the same order of printTrie(). print() is a breadth-first traversal. It's time complexity would also be $O(hn)$. By a similar argument for printLevel(), it will be similar; in fact, $O(in)$ where i is the i^{th} level if $i \leq max(height)$ or $max(height)$ otherwise, to be more precise.

5.2 Red-Black Tree

Due to the properties of Red Black Trees, it is balanced and it's height of the tree is $O(\log n)$ (n is total number of nodes). The insertion would take place at $O(2 \log n)$ in the worst case (one $O(logn)$ for going down and another for coming up till root while recolouring) in the worst case and expected to be $O(\log n)$. Restructuring is $O(1)$ steps. Searching is also $O(\log n)$ as we have to traverse along the height while searching.

5.3 Max Heap

A Max-Heap is also balanced. So height would be $O(\log n)$. For insertion, we have to insert at the last position and travel up by swapping to maintain the properties. Hence `insert()` is $O(2\log n)$ (n being number of nodes in the Heap) and expected to be $O(\log n)$. By a similar argument, for `extractMax()`, the worst case time complexity would be $O(2\log n)$ as this involves making the last node replaced at root travel downwards and expected to be ($O(\log n)$.

6 Interesting Findings

- Tries have wastage of spaces but are very efficient due to it's $O(lengthofkey)$ time complexity. They are better than Hashes as the latter can have a bad worse case time complexity.
- Tries enable easy prefix-searching which might be difficult to implement in Hash Tables and BSTs.

7 Threading

7.1 Implementation

We have the following objects:

- `allUsers` stored in a Red-Black Tree
- `allProjects` stored in a Trie
- `allJobs` (unfinished jobs) stored in Max Heap
- `jobList` with the history of all the jobs (completed and pending) stored in a Trie
- `expensiveJobs` which have higher budget requirement in a Queue
- `completedJobs` stored in a Queue

7.1.1 `handle_user()`

We lock `allUsers` which is sufficient for performing operations in this method.

7.1.2 `handle_project()`

We lock `allProjects` and perform the normal operations.

7.1.3 `handle_job()`

We first check if the `Project` and `User` provided in the `Job` is already present. For this we lock `allProjects` first, check for `Project`s, then lock `allUsers` and check for `User`s. Since we donot have a *delete* operation, we can release these locks as soon as the corresponding checks are over. Then we have to lock `jobList` and `allJobs` and insert the `Job`.

7.1.4 `handle_query()`

We first lock `jobList` to see if the `Job` is present and release it after the check is complete. Then we lock `completedJobs` while we check if the `Job` has been completed (and release it after the check). If not, the `Job` has not been finished yet.

7.1.5 `handle_add()`

Lock `allProjects` to get the `Project` details from the `Trie` first and release it after fetched. It is necessary to lock `allJobs` as we don't want other Threads to execute a `Job` while `ADD` is encountered (we may loose the execution of a `Job` with higher priority). It is not necessary to lock `expensiveJobs`, which is being used at `handle_add()` and `schedule()` while Threads get implemented. We perform the remaining operations in 4.4.5.

7.1.6 handle_empty]$_{l}ine()$

We call `schedule()`, which operates on `allJobs`, `expensiveJobs` and `completedJobs`. We lock `allJobs` and `completedJobs` and call `schedule()` As mentioned in 7.1.5, `expensiveJobs` need not be locked as `allJobs` is being locked.

7.1.7 main()

For each command, we create Threads and run them. We wait for all the Threads to complete their execution before calling `run_to_completion()` followed by `print_stats()`
All other methods remains unchanged.

7.2 Deadlocks

All the Threads are Thread-safe and won't encounter in deadlock because of the following reasons:

- No other methods other than `handle_job()` and `handle_empty_line()` has lock on one object at the time which will be released at some point of time.
- In `handle_job()`, there are locks on `allJobs` and `jobList`, simultaneously, and `handle_empty_line()` on `allJobs` and `completedJobs`. Clearly they are not interdependent in the two cases.