ADVANCED DATA STRUCTURES
COP5536: Spring 2018

PROGRAMMING PROJECT REPORT

Job Scheduler

SUBMITTED BY
Sai Swetha Kondubhatla
UFID - 1175-9282
Email ID: skondubhatla@ufl.edu

Job Scheduler

Job Scheduler is used to schedule jobs in an operating system. The task of a scheduler is to schedule jobs according to the user requirements. It has been decided that when the processor becomes free, the scheduler will assign to it a job that has been run for the least amount of time so far. This job will run for the smaller of 5ms and the amount of remaining time it needs to complete. In case the job does not complete in 5ms it becomes a candidate for the next scheduling round.

Programming Environment

To implement this Job Scheduler, I have used JAVA programming environment. I used Eclipse IDE to run the code.

Function Prototypes and Structure:

Class Job.java

This class has all the variables related a specific job. A job contains the following fields:

- **Executed_time**: The time for which the job has ben scheduled by the CPU.
- **Total_time**: The total execution time of a job.
- **Remaining_time**: The total remaining time of a job after scheduling.

  Whenever this class is called, the job is inserted in both the Heap and the RedBlack tree respectively using the following methods:

- **Void Job(HeapImpl hp, int newId, long newExecTime, long newremaining):** This method inserts a job with the following values in the Heap

- **Void Job(RedBlackTree rbtree, int newId, long newExecTime, long newremaining):** This method inserts a job with the following values in the Red Black Tree.

Class RedBlackTree.java

This class uses the following methods to operate on a Red Black Tree. The red black tree is implemented with **Job_id** as its key.

- **Node closestValue(Node node, double j_id):** This method calculates the Next job for the given JobId j_id and returns node consisting of next job. This method uses the helper() method to calculate the NextValue and the closest value method is used in the NextJob() method.

- **Node closestValue_Prev(Node node, double j_id):** This method calculates the Previous job for the given job j_id and returns node with the previous job. This method uses the helper_prev() method to calculate the PreviousJob and the closestvalue_Prev method is used in the PreviousJob() method.

- **Boolean delete(Job key):** This method deletes a particular Job key from a Red Black Tree. If the job is deleted, then it returns true, else, it returns false.

- **Void deleteFixup(Node x):** This method rebalances the entire Red Black Tree after deletion of a node x. There are six cases for the deletion of a node in a red black tree. These are handled by this method thereby maintaining the balancing property.

- **Void deleteTree():** This method deletes the Entire Red Black Tree.

- **Node findNode(Node n1, Node n):** If node n1, if found in the tree, it returns n1. If not found, it returns null.

- **Void fixTree(Node node):** Whenever a node is inserted or deleted, the tree has to be balanced by performing rotate left or rotate right operations. This method helps handle these rotations and hence the balancing property of the tree is maintained.

- **Void insert(Job J):** This method Inserts a new job J into the Red Black Tree. After insertion, this method calls the fixTree method to maintain the rules of the red Black tree.

- **Void NextJob(int j_id):** This method prints the triplet (JobId, Executed_time, Total_time) for the job with smallest ID greater than a given jobID. It prints (0,0,0) if there is no such job. This method makes use of closestValue(Node node, double j_id) which further uses the helper method to calculate the next job.

- **Void PreviousJob(int j_id):** This method prints the triplet (JobId, Executed_time, Total_time) of the job with the greatest jobID that is less than to a given jobID. It prints (0,0,0) if there is no such job. This method makes use of closestValue_prev(Node node, double j_id) which further uses the helper_prev method node to calculate the next job.

- **Void Print(Node root, int j1, int j2):** This traverses all the nodes from root and prints all the triplets (JobId, Executed_time, Total_time) of the jobs which lie in the range [j1,j2].

- **Void printJob(int j_id):** This method prints the triplet (JobId, Executed_time, Total_time) of the job for given jobID j_id. It prints (0,0,0) if there is no such job.

- **Void printJob(int j1, int j2):** This method calls the helper function Print(Node root, int j1, int j2) to print he jobs in the range [j1, j2].

- **Void rotateLeft(Node node):** This method perform left rotation on the Red Black Tree.

- **Void rotateRight(Node node):** This method perform right rotation on the Red Black Tree.

- **Void transplant():** This method is called by the delete() method when the parent and child pointers are to be adjusted.

Class HeapImpl.java
This class consists of the following methods which help implement the heap. The heap is implemented with **executed_time** as the key.

- **Int getSize():** This method returns the size of the heap.

- **Void insert(Job J):** This method inserts a job J in the form of a node into the heap. If the newly inserted node is smaller, then it is swapped with the parent and it is made the current node and this continues until the minheapify property is satisfied.

- **Void minHeap():** This method forms a min heap using the minheapify() function.

- **Void minHeapify(int pos):** The node of the index pos is compared with its children. If the parent node is smaller than any of the children then it is swapped with that child. Then the minHeapify operation is performed on the exchanged child again to ensure Min heap property.

- **private void swap(int smallest, int index);**
  This function is called from Minheapify function. This function swaps both the parent node and the smallest child of the parent node to satisfy the min heap property.

- **Int parent(int pos):** This method takes in the position of the current node and returns the position of the parent of the current node.

- **Void print():** This method prints the heap in the order in which the values are inserted.

- **Job remove():** This method removes the Min Job i.e. the root of the heap.

- **Int remove_job(Job J):** This method removes a Job J from the heap and returns the index of the job.

Class jobscheduler.java
This class contains the main method, which drives the entire program.
This class takes in the input file (input.txt) and every line of the file is maintained in a string list. The arrival times of every operation is maintained in a timer (Int) list.
Each line of the input is analysed and a nested if and else loop is created where each operation of the scheduler is implemented. The following are the operations supported in the input.java file.

- **PrintJob (jobID, executed_time, total_time)** prints the triplet for a given jobID.
- **PrintJob (jobID, executed_time, total_time)** prints the triplets for all jobIDs in the range [low, high].
- **NextJob(jobID, executed_time, total_time)** prints the triplet for the job with smallest ID greater than a given jobID and the scheduler should print (0,0,0) if there is no such job.
- **PreviousJob(jobID, executed_time, total_time)** prints the triplet of the job with the greatest jobID that is less than to a given jobID and the scheduler should print (0,0,0) if there is no such job.
- **Insert (jobID, total_time)** inserts a new job into both the heap and RBT.

These above functions run in parallel with the scheduler, which schedules the jobs according to their executed_time.

Class Scheduler.java:
This class helps to schedule jobs based on their executed time and arrival time.
Every job has to run a min of 5ms provided the total_time of the job > 5ms. In case the job doesn't complete in 5ms, it becomes the next candidate for scheduling.
This class contains a global variable called **global_timer** which increments by 1ms as the job is scheduled. Every time a new job starts, the variable **jobrun_timer** is set to 0 and this variable keeps track of the time for which each job is run. If the job has run for 5ms, next job which has **lowest execution_time** will be scheduled. In case, more than one job has the same lowest execution time, the job with the least total_time will be executed as this can be done earlier and thereby removed from the heap.

- **Void scheduleHeap(HeapImpl hp, RedBlackTree rbt):** This method is used to schedule the jobs in such a way that it follows the above requirements. It makes use of the functions from both the RedBlackTree class and HeapImpl class in order to implement the scheduler.
- **Void scheduleHeap_remaining(HeapImpl hp, RedBlackTree rbt):** When the last function of the input file is run, there still must be pending jobs which

haven't been either scheduled or completed. All the remaining jobs, will be scheduled using this function.

Structure of the Program

1. The main function is present in jobscheduler.java file. When this file is executed, it takes the file given by the user as input. The arrival times of all the operations present in the file are added to a list (timer list). Based on these arrival times and the operations, the Jobs are inserted and executed. When the first insert operation comes in, thats when the global_timer starts and each operation calls Scheduler.java to schedule the jobs.

2. In Scheduler.java, a local time variable for each job(jobrun_time) is maintained and is incremented every second the job runs. When arrival_time = global_timer, thats when the scheduling and the operation mentioned will perform simultaneously.

3. The jobrun_timer variable keeps track of the time for which each job is run. When it exceeds 5ms, then it will go back to the heap, check if there is any eligible job for scheduling (based on lowest executed_time) and schedules the new job. If none, the same job will be scheduled. In case of more than one jobs with same execution time, total_time is considered and Job with lowest total_time will be scheduled. If the job is done executing, it is removed from both the heap and red black tree.

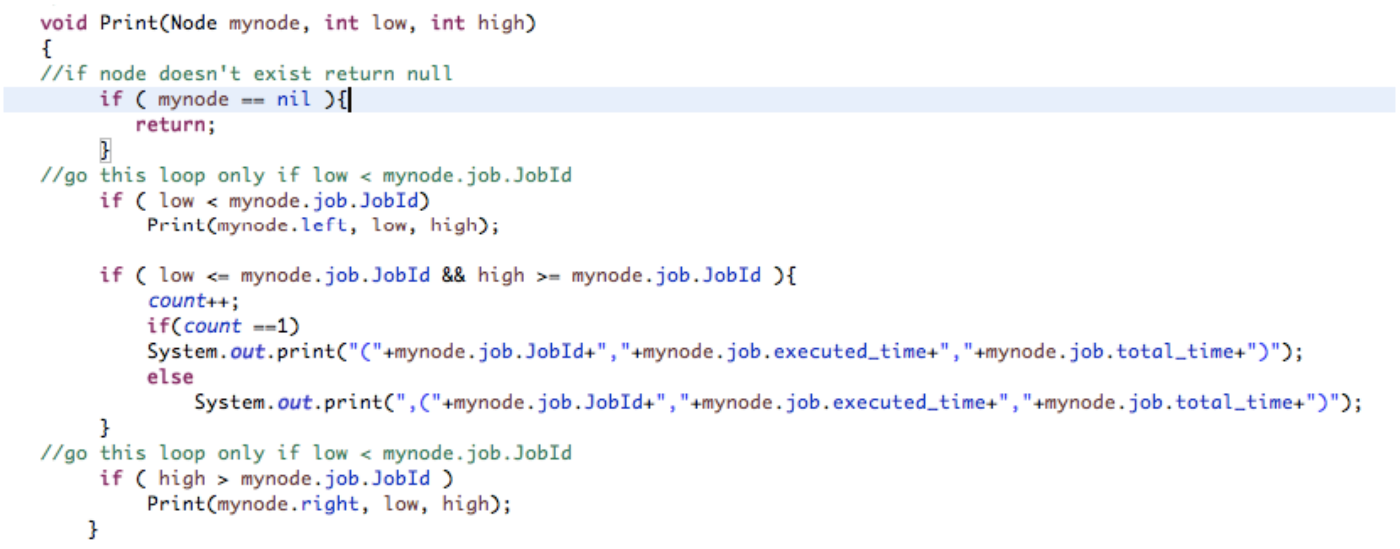
4. For getting a new job, deleting a job, maintaining the heap property, the methods present in HeapImpl.java and RedBlackTree.java are used. These files have methods that ensure the balancing of trees.

5. For the NextJob, PreviousJob, PrintJob the values are printed from the red black tree and these operations use the methods provided in the file RedBlackTree.java. While these operations come in, the scheduler also runs simultaneously.

Complexity:

- For PrintJob(jobID), NextJob(jobID), PreviousJob(jobID), the scheduler searches for a particular node in a red black tree and returns the Job with that jobID. We are essentially searching for the node in a tree. The search-time results from the traversal from root to leaf, and therefore a balanced tree of *n* nodes, having the least possible tree height, results in O(log *n*) search time.

- For PrintJob[low, high] range, 2 if loops have been written. The functions goes to the loop only if low < mynode.job.JobId or high > mynode.job.JobId.
  In best case, if either of this condition fails, it has to search for only half of the



```
void Print(Node mynode, int low, int high)
{
//if node doesn't exist return null
     if ( mynode == nil ){
        return;
     }
//go this loop only if low < mynode.job.JobId
     if ( low < mynode.job.JobId)
         Print(mynode.left, low, high);

     if ( low <= mynode.job.JobId && high >= mynode.job.JobId ){
         count++;
         if(count ==1)
         System.out.print("("+mynode.job.JobId+","+mynode.job.executed_time+","+mynode.job.total_time+")");
         else
             System.out.print(",("+mynode.job.JobId+","+mynode.job.executed_time+","+mynode.job.total_time+")");
     }
//go this loop only if low < mynode.job.JobId
     if ( high > mynode.job.JobId )
         Print(mynode.right, low, high);
    }
```

array thereby making the complexity O(log(n)). In case both of these conditions don't fail, it will take O(log(n) +S) complexity where S are the number of nodes within the range. In worst case, S = n, all the nodes in the red black tree lie within the range. Therefore, complexity = O(log(n) + (N-1)) = O(n). Hence, worst case complexity of this function is O(n) and best case is O(log(n)).