CSE 5311: Design and Analysis of Algorithms

Project: Task Scheduler using Red-Black Trees

Spring 2018

Instructor: Dr. Gautam Das
TA: Shohedul Hasan

Group No.: 8

Kaustubh Agnihotri (1001554290)

Karthik Venkatasivareddy (1001518082)

Supreeth Javalli (1001518038)

Introduction

The Completely Fair Scheduler (CFS) is a process scheduler which handles CPU allocation for executing processes and aims at maximizing overall CPU utilization while also maximizing interactive performance. CFS uses the concept of “sleeper fairness”, which considers the sleeping or waiting tasks equivalent to those on the run queue. This means that interactive tasks which spend most of their time waiting for user input or other events get a comparable share of CPU time when they need it.

A "maximum execution time" is calculated for each process. This time is based upon the idea that an "ideal processor" would equally share processing power amongst all processes. Thus, the maximum execution time is the time the process has been waiting to run, divided by the total number of processes, or in other words, the maximum execution time is the time the process would have expected to run on an "ideal processor".

Completely Fair Scheduler Algorithm

When the scheduler is invoked to run a new process, the operation of the scheduler is as follows:

1. The left most node of the scheduling tree is chosen (as it will have the lowest spent execution time), and sent for execution.
2. If the process simply completes execution, it is removed from the system and scheduling tree.
3. If the process reaches its maximum execution time or is otherwise stopped it is reinserted into the scheduling tree based on its new spent execution time.
4. The new left-most node will then be selected from the tree, repeating the iteration. If the process spends a lot of its time sleeping, then its spent time value is low, and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.

Refer here

CFS Implementation

For implementing the completely fair scheduler we made some adjustments to the algorithm but did not make any changes that would affect the core of the algorithm. We implemented the algorithm in the following way:

1. Read the input file and sort it according to arrival time into an array
2. The first process (whenever tree is empty) is executed for quantum slice. While this process executes, other processes might arrive and wait
3. After execution is completed, if the process still has time left, it is inserted into the tree
4. Now all the processes which have arrived while 1st process was executing are also inserted into the tree
5. The unfairness score is calculated for each process based upon the arrival time and spent time as unfairnessScore = arrivalTime + spentTime
6. This ensures that the process with the least spent time goes to the left most leaf
7. Now we extract the left most process and execute it by calculating the maximum_execution time as stated above (waitingTime / no. of processes)
8. Thus, every time the process which has waited the most (or spent least time using the CPU) is selected

Data Structures and Programming Languages

As stated in the project description, we have used following two types of data structures:

1. Red Black Tree
2. Min Heap (Using priority queue)

We have implemented the CFS algorithm using the two data structures in java. We have then created a wrapper in python to call that algorithm. All the test cases have been designed and run using the python wrapper. We have used python since it was easier to plot graphs and generate random inputs for testing purposes using python. User inputs are also accepted through the python UI.

Test Cases and Results

We performed many tests on the developed code and found out some unexpected results. Heap performed better than red black tree data structure for most of the times. The main reason behind this was that retrieving the minimum element from the heap is possible in O(1) time, whereas for the red black tree it took O(log n) time. Another reason for this could be that the maximum height of red black tree can be 2 log(n) whereas that of a heap can be log(n). Even though in big O notations we would refer as O(log n), this can be responsible for the slight better performance of heap than red black tree.

Let us go through some analysis using graphs that we generated from the test results.

Inputs provided: Table1 shows the inputs that were provided for certain test cases. Using these inputs, we ran the code and retrieved the execution time taken by RBT and Heap scheduler individually. We compared these results and Figure 1 displays the comparison graph.

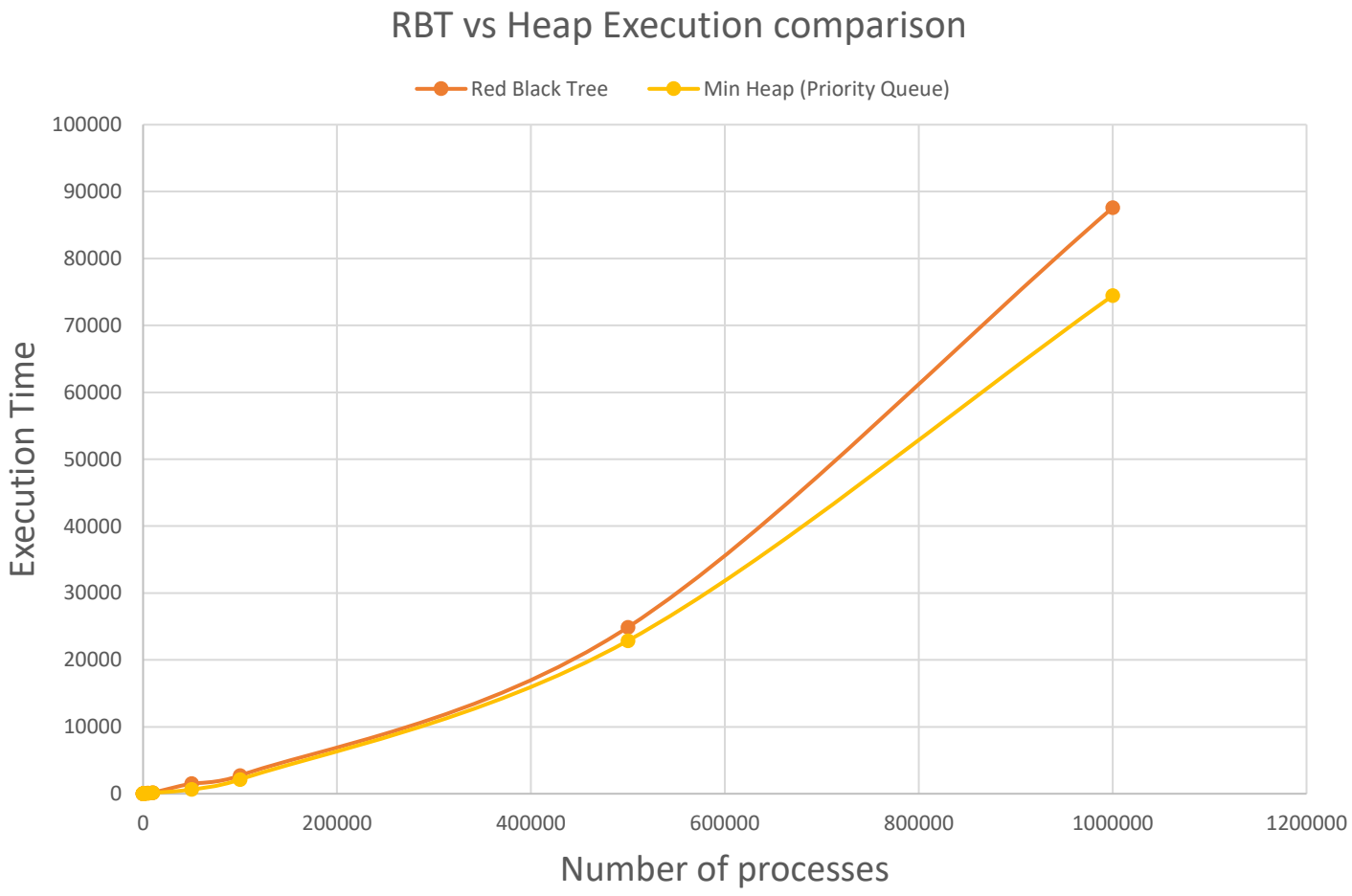
After this we executed the algorithm with different input sizes for 10 iterations each and compared the execution time taken by RB Tree and Min Heap for every input size. While generating the inputs for these test cases, we included the following two scenarios:

1. Same input file for all 10 iterations i.e. the size as well as the content remains same
2. Different input file for all 10 iterations i.e. the size remains the same, but the content varies

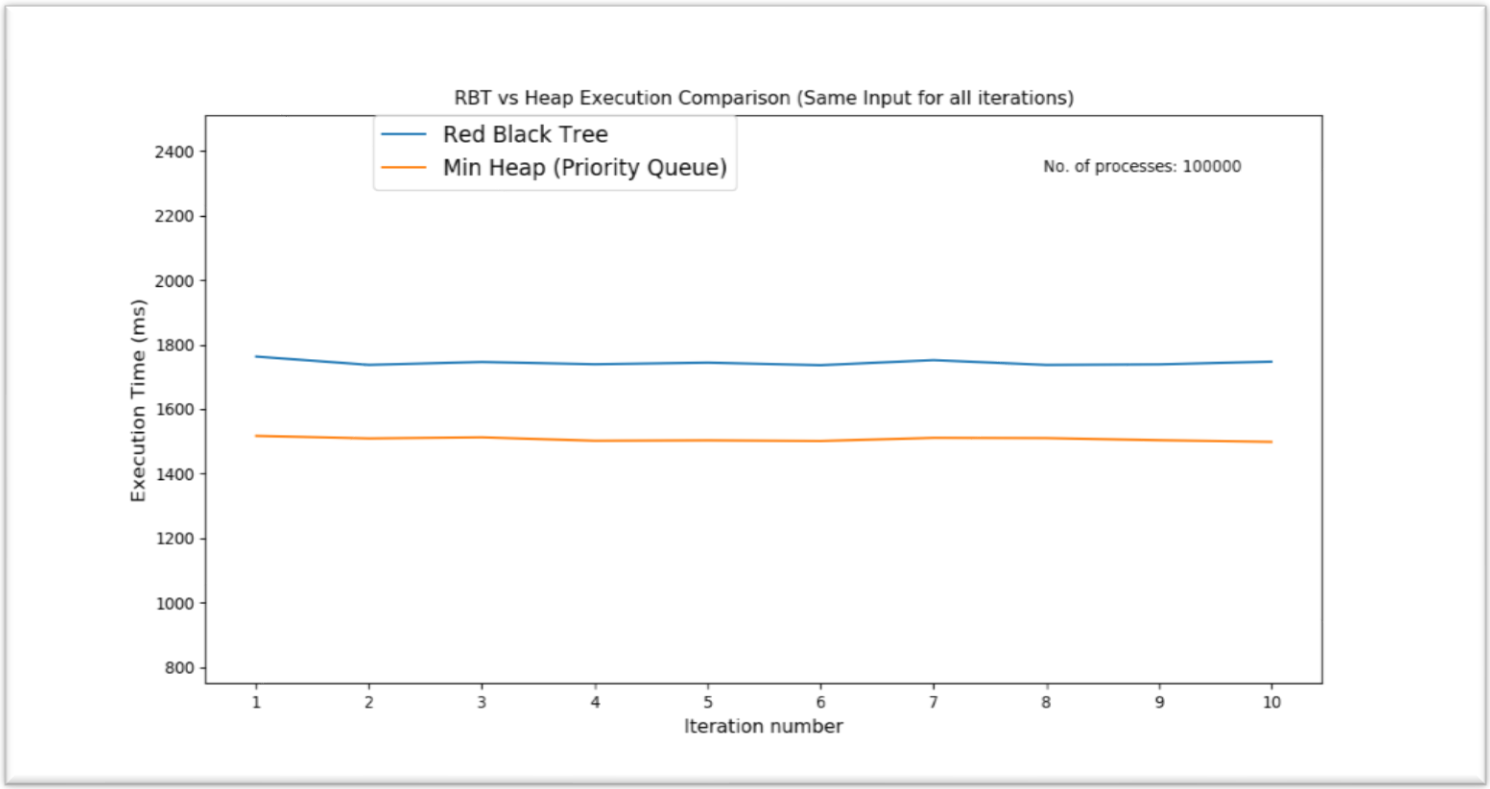
The results for these cases have been displayed below in Figure 2, Figure 3, Figure 4, Figure 5, Figure 6 and Figure 7.

1 Inputs provided

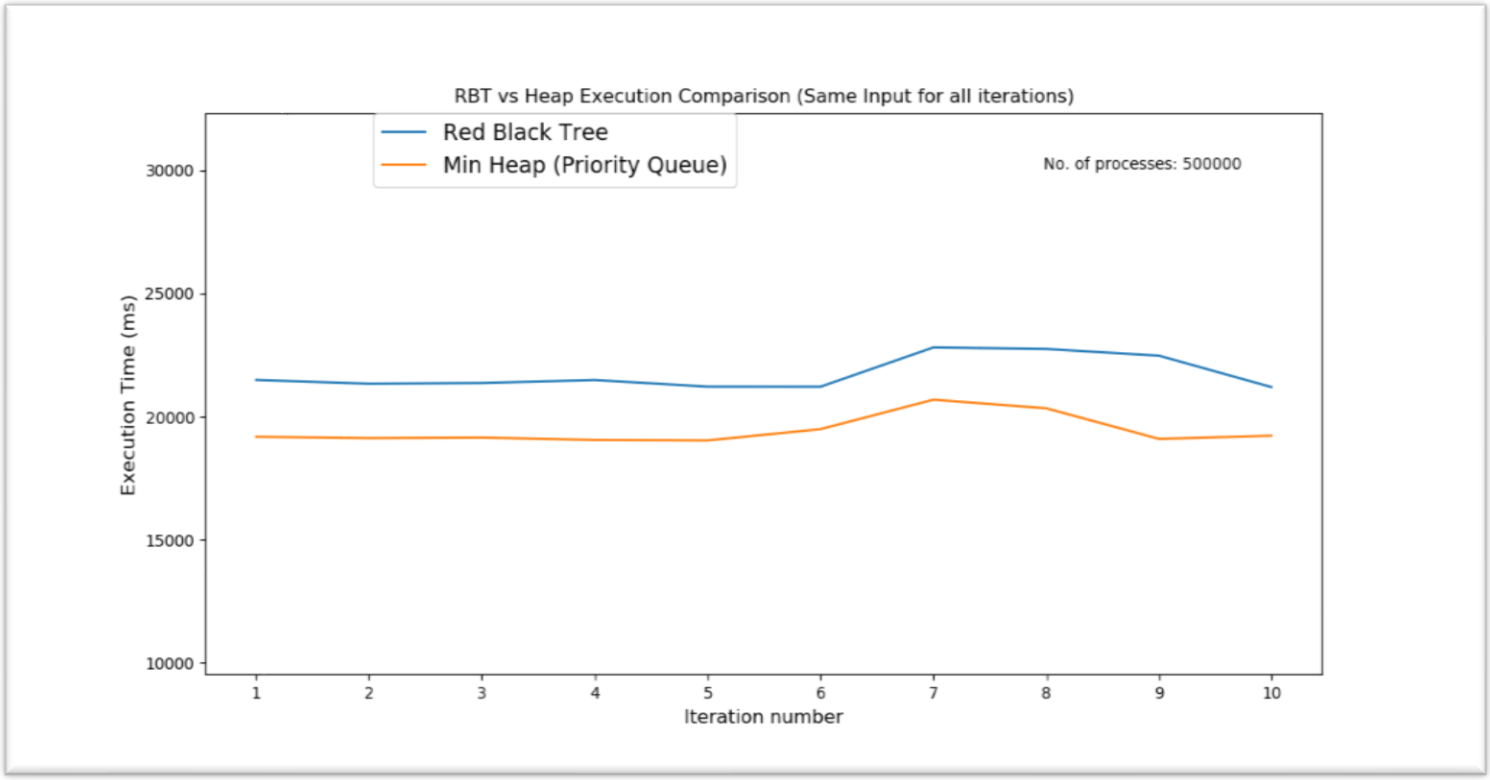
Quantum Slice	No. of processes	RBT Execution time (ms)	Heap Execution time (ms)
4	5	1.245865	1.203199
8	10	1.383536	1.187839
12	50	4.044228	2.319358
16	100	5.058556	3.067446
20	500	13.829108	10.187654
40	1000	17.836927	13.820575
60	5000	88.697666	68.661985
80	10000	161.868662	135.595119
100	50000	1528.531461	640.667329
120	100000	2721.570655	2132.328562
200	500000	24880.85144	22870.75936
320	1000000	87586.74979	74442.86256



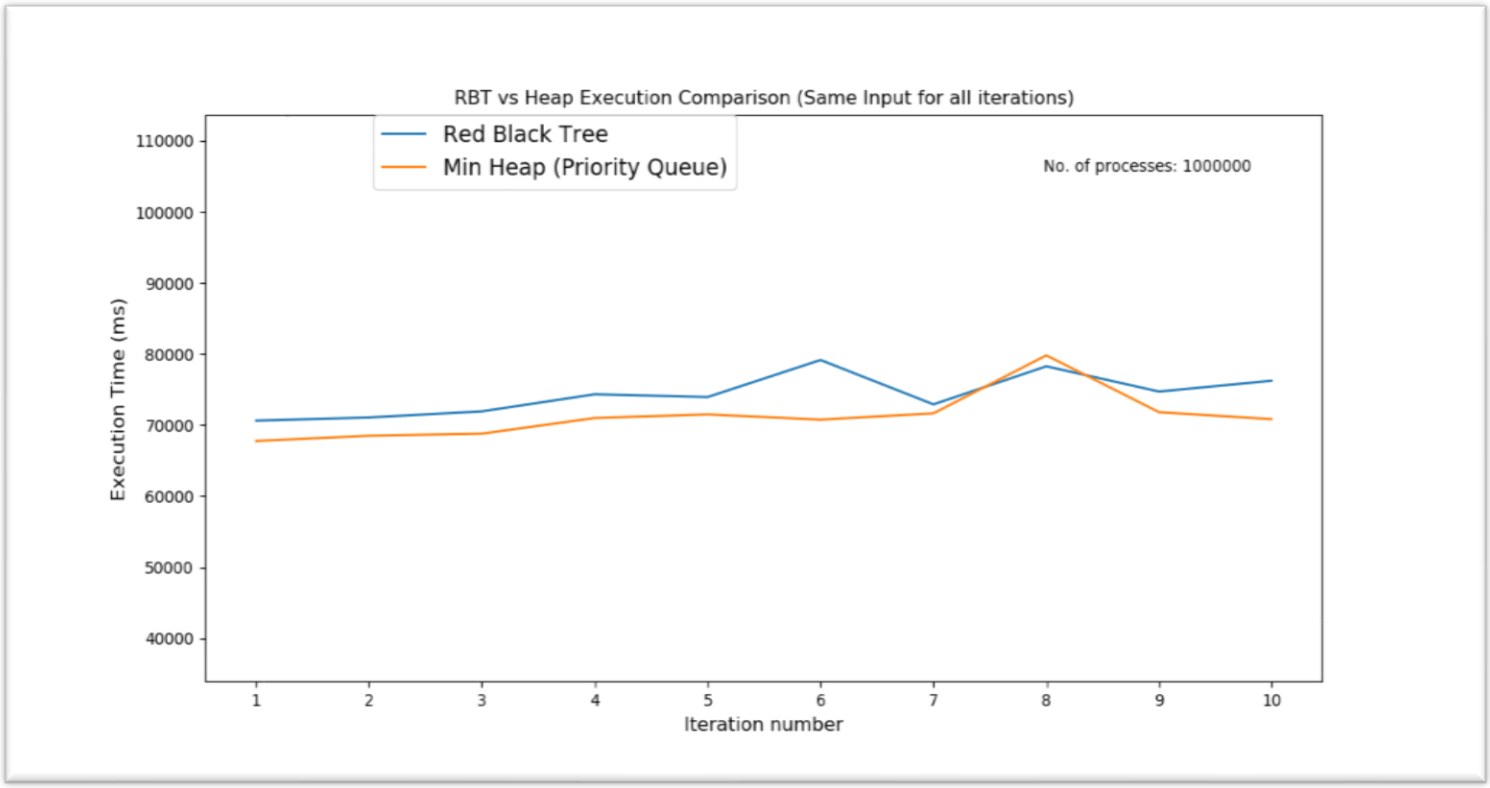
1 RBT vs Heap Comparison



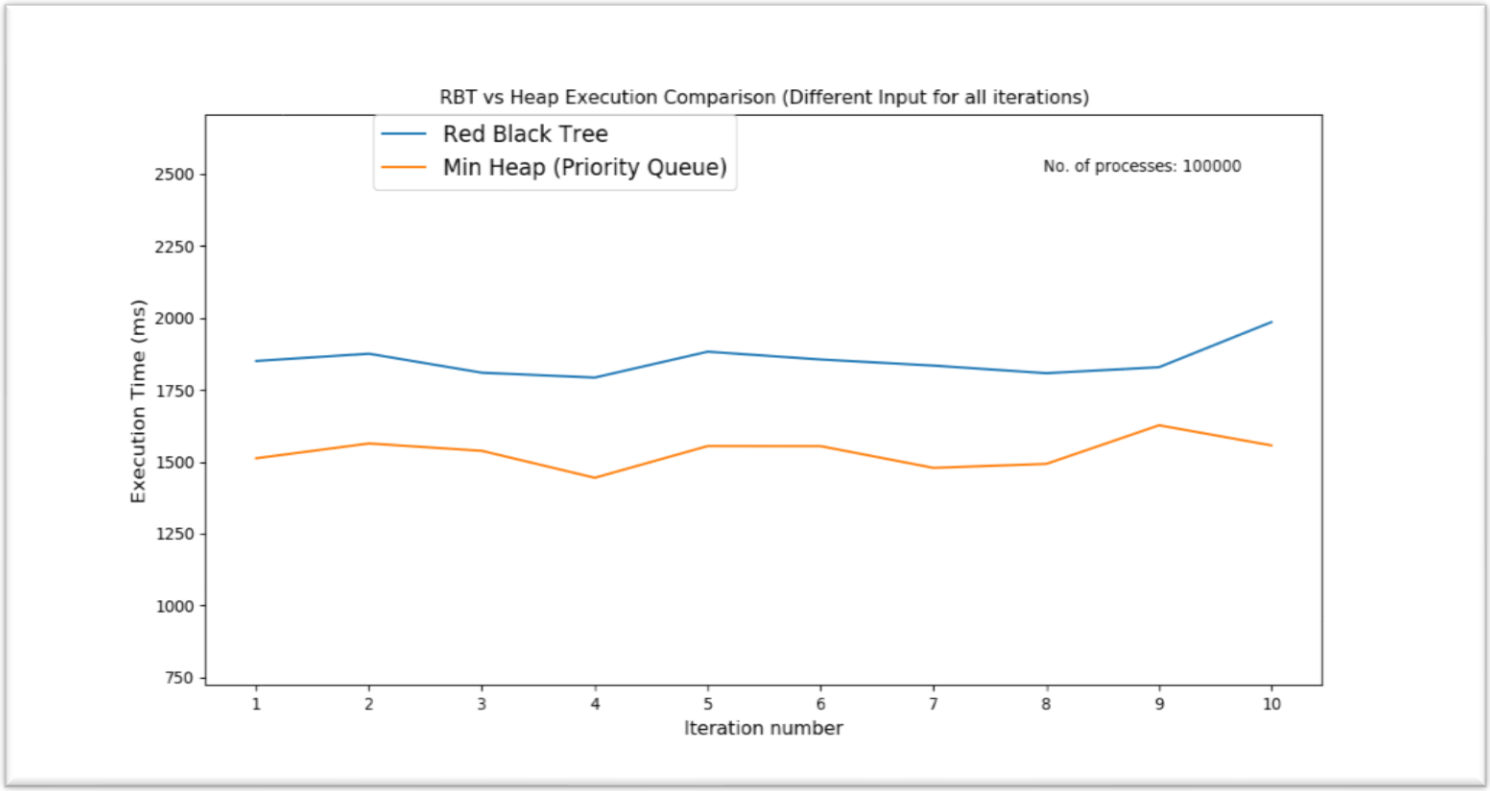
2 RBT vs Heap (100000 inputs, 10 iterations)



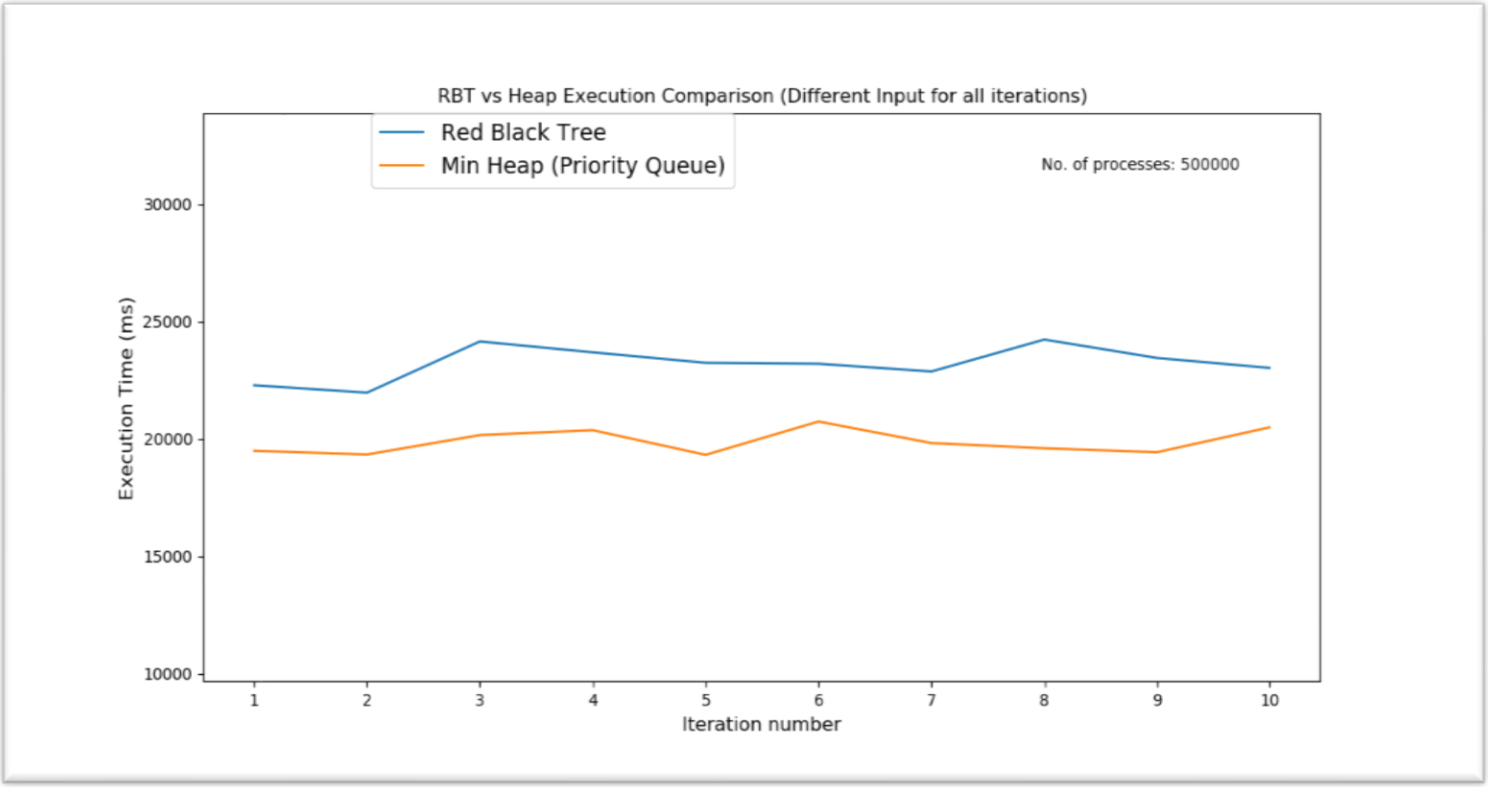
3 RBT vs Heap (500000 inputs, 10 iterations)



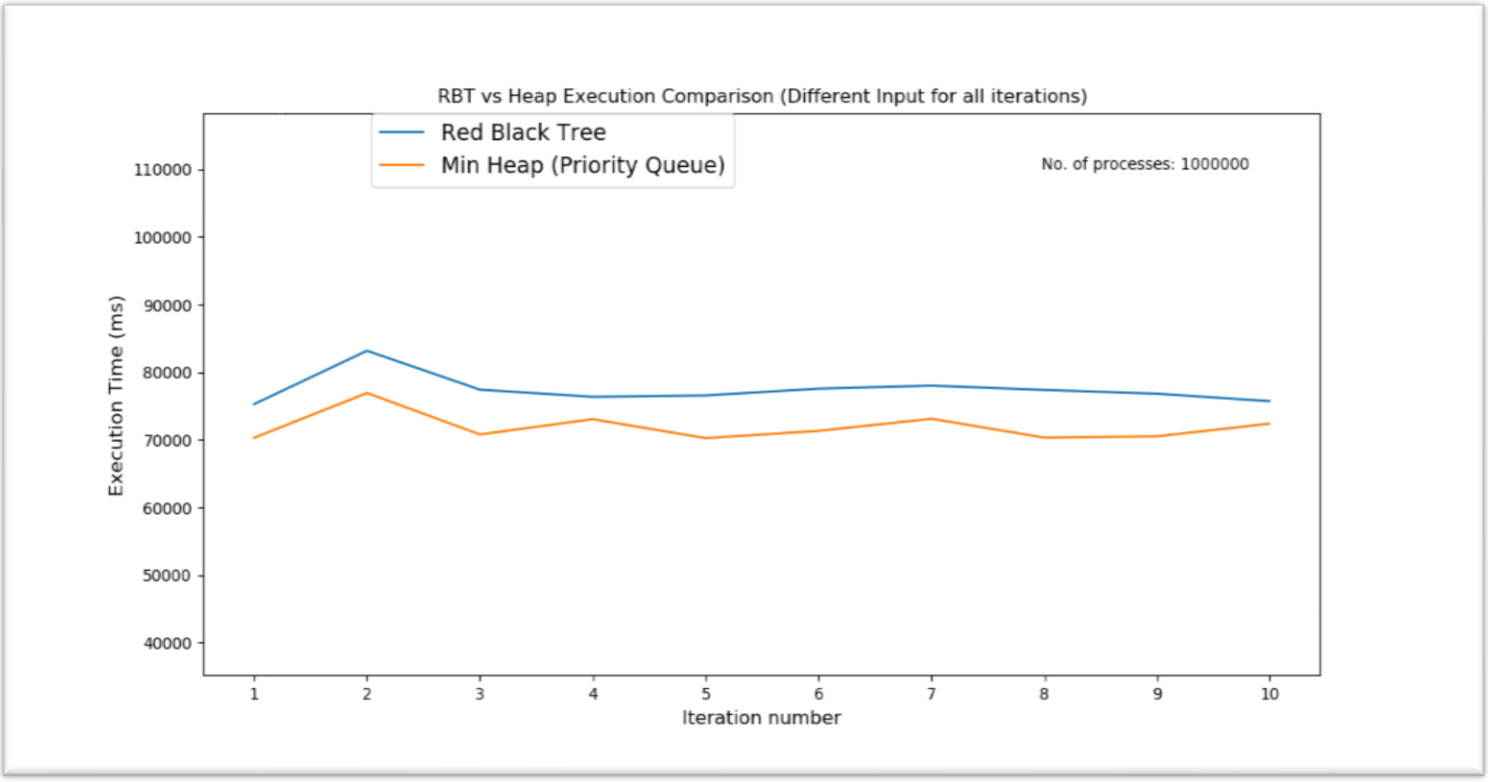
4 RBT vs Heap (1000000 inputs, 10 iterations)



5 RBT vs Heap (100000 inputs, 10 iterations)

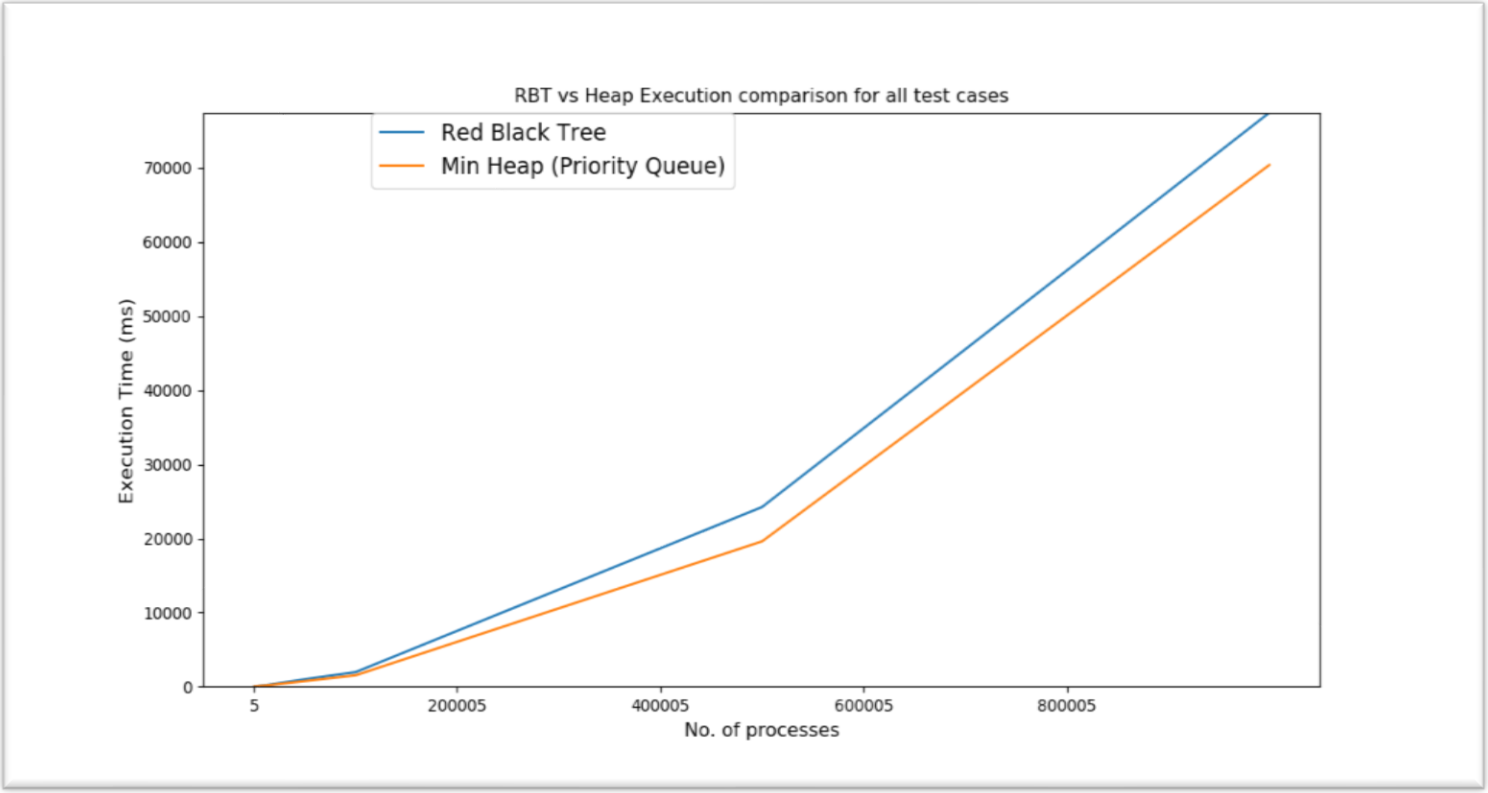


6 RBT vs Heap (500000 inputs, 10 iterations)



7 RBT vs Heap (1000000 inputs, 10 iterations)

We also computed the execution times from each test case when the difference between the execution time of RBT and Heap was maximum. Plotting these against the number of processes used for the input gave us a similar and consistent result as that in Figure 1. The results for this test case have been displayed in Figure 8.



8 RBT vs Heap (1000000 inputs, 10 iterations)

Innovative Experiment:

In order to test whether the algorithm is actually able to get comparable share of CPU time and are executing in the manner expected, we decided to check with the following input:

Quantum Slice = 5

<0, 0, 10>, <1, 0, 10>, <2, 0, 10>, <3, 0, 10>, <4, 0, 10>

All the processes arrived at 0 and had a total execution time of 10 units. For a sequential execution the processes would have completed with following waiting time (Table 2):

2 Sequential Execution

Process ID	Waiting Time
P0	20
P1	25
P2	30
P3	35
P4	40

Now with the CFS scheduler we got the following results (Table 3):

3 CFS Execution

Process ID	Waiting Time
P0	32.307636
P4	35.985420
P3	38.545704
P2	39.792778
P1	40.00000

From the above table we can see that the CPU tried to give equal opportunity to each and every process. The difference between the waiting times process finishing first and process finishing last reduced significantly. Whereas, the maximum waiting time i.e. the waiting time of the process finishing last remained the same. This shows us that the resources of CPU were shared fairly amongst all the processes by using the CFS scheduler. Please find the output for the CFS scheduler for above test case in the following file: Same Arrival Equal Exec Output.txt

Evaluation:

1. Compare this implementation with that of a heap

➔ As seen from the tests above, heap performed better than the red black tree. The reasons as stated above for this performance direct us to the probable cause being the O(1) complexity for retrieving the minimum element from the heap which is a lot less than O(log n) required by red black tree. Also the maximum height of red black tree can be 2 log(n) where as that of a heap is going to be log(n).

2. How fair is the scheduler actually?

➔ An ideal scheduler would allocate equal execution time to each process and there by maintain an equal waiting time for each process. The CFS scheduler achieves this goal to a fair amount and ensures that all the processes receive equal share of the CPU resources when they come for execution.

3. Does this scheduler maximize throughput?

➔ The scheduler maximizes throughput, especially when multi-programming comes into picture. Since all the processes are able to get a fair share of execution, interactive processes which spend most of their time waiting for some action, receive comparable share as soon as they need it.

Why CFS uses Red Black Tree if Heap performed better:

Implementing heap in Linux requires contiguous memory allocation which becomes tough ask as the multi-programming grows. Thus, space requirement is a hurdle in implementing heap. (Refer this)

References:

1. https://en.wikipedia.org/wiki/Completely_Fair_Scheduler
2. https://www.alouche.net/2012/02/04/linux-cfs-algorithm-and-virtual-runtime/
3. https://www.linuxjournal.com/node/10267
4. https://stackoverflow.com/questions/33191110/reason-why-cfs-scheduler-using-red-black-tree?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa