Istanbul Technical University
Faculty of Computer and Informatics
Computer Engineering Department

BLG 335E
The LaTeX
Report

Analysis of Algorithms I, Project 3

Leminur Çelik - 150190085

December 30th, 2022

Contents

1 Description of Code

I used red black tree data structure in my implementation. I created an array to store elements given in the input file. I created process nodes with the properties of name, virtual running time, burst time, color, parent, right child and left child. I inserted the processes into the tree according to their arrival times. Every time the CPU time increases, then the virtual run time of the currently running task is increased by 1. By finding the minimum node, the process that will run now is selected. If there are nodes smaller than or equal to the run time of the minimum node, then delete the minimum node, update it by incrementing its virtual run time and check if it reaches its burst time. If it reaches the burst time, the process is completed. If not insert it into tree again.

If there are not any nodes smaller than or equal to the run time of the minimum node, then update the minimum node by incrementing its virtual run time and check if it reaches its burst time. If it reaches its burst time, then delete this node, because it is completed. If the given time runs out or all processes are processed, the simulation comes to an end. Red black tree is a binary tree, so dynamic set operations such as minimum, maximum, insert and delete takes O(lgn) time in the worst case. Red black tree has 5 property:

1. The root is black
2. Every node is either red or black
3. Every leaf is black
4. If a node is red, then both its children are black
5. The number of black nodes on each node's simple pathways to descendant leaves is constant.

I wrote 10 functions as shown below:

- Insert
- Delete
- Insert Fixup
- Delete Fixup
- RB Transplant
- Minimum
- Get Root
- Exist Minimum
- Left Rotate
- Right Rotate

1.1 Insert

When a new node is wanted to be added, its name, virtual run time and burst time are taken as input and added to the tree. If the red black tree property is lost, the fixup function is called.

Algorithm 1 Insert

```
function INSERT(name, vruntime, bursttime)
    Node * node ← newNode
    node.parent ← NULL
    node.vruntime ← vruntime
    node.name ← name
    node.bursttime ← bursttime
    node.left ← TNULL
    node.right ← TNULL
    node.color ← Red
    Node * y ← NULL
    Node * x ← Red
    while x ≠ TNULL do
        y ← x
        if node.vruntime < x.vruntime then
            x ← x.left
        end if
        if node.vruntime >= x.vruntime then
            x ← x.right
        end if
    end while
    node.parent ← y
    if y ← NULL then
        root ← node
    end if
    if node.vruntime < y.vruntime then
        y.left ← node
    end if
    else
    y.right ← node
    if node.parent ← NULL then
        node.color ← Black
        return
    end if
    if node.parent.parent ← NULL then
        return
    end if
    FIXUP-INSERT(node)
end function
```

1.2 Insert Fixup

When the red black tree feature is broken when node is added, it is fixed in the fixup function.

Algorithm 2 Fixup Insert

```
function FIXUP-INSERT(z)
    while z.parent.color ← Red do
        if z.parent ← z.parent.parent.left then
            y ← z.parent.parent.right
            if y.color ← Red then
                z.parent.color ← Black
                y.color ← Black
                z.parent.parent.color ← Red
                z ← z.parent.parent
            end if
            if z ← z.parent.right then
                z ← z.parent
                LEFT-ROTATE(z)
            end if
            z.parent.color ← Black
            z.parent.parent.color ← Red
            RIGHT-ROTATE(z.parent.parent)
        end if
        else(same as then clause with right and left exchanged)
    end while
    root.color ← Black
end function
```

1.3 Delete

When a node is wanted to be deleted, root node, run time and name of the node to be deleted are taken as input. After the necessary changes are made, the node is deleted. If the red black tree property is lost, the fixup function is called.

Algorithm 3 Delete

```
function DELETE(node, vruntime, name)
    Node * z ← TNULL
    Node * y
    Node * x
    while node ≠ TNULL do
        if node.vruntime ← vruntime AND node.name ← name then
            z ← x
        end if
        if node.vruntime <= vruntime AND node.name ← name then
            node ← node.right
        end if
        else
        node ← node.left
    end while
    if z ← TNULL then
        return
    end if
    y ← z
    original − color ← y.color
    if z.left ← TNULL then
        x ← z.right
        RB-TRANSPLANT(z, z.right)
    end if
    if z.right ← TNULL then
        x ← z.left
        RB-TRANSPLANT(z, z.left)
    end if
    else
    y ← MINIMUM(z.right)
    original − color ← y.color
    x ← y.right
    if y.parent ← z then
        x.parent ← y
    end if
    else
    RB-TRANSPLANT(y, y.right)
    y.right ← z.right
    y.right.parent ← y
    end else
    RB-TRANSPLANT(z, y)
    y.left ← z.left
    y.left.parent ← y
    y.color ← z.color
    delete z
    end else
    if original − color ← black then
        FIXUP-DELETE(x)
    end if
end function
```

1.4 Delete Fixup

When the red black tree feature is broken when node is deleted, it is fixed in the fixup function

Algorithm 4 Fixup Delete

function FIXUP-DELETE(x)
  while $x \neq root\ ANDxcolor \leftarrow Black$ **do**
    if $x \leftarrow x.parent.left$ **then**
      $s \leftarrow x.parent.right$
      if $s.color \leftarrow Red$ **then**
        $s.color \leftarrow Black$
        $x.parent.color \leftarrow Red$
        LEFT-ROTATE(x.parent)
        $s \leftarrow x.parent.right$
      end if
      if $s.left.color \leftarrow Black\ ANDs.right.color \leftarrow Black$ **then**
        $s.color \leftarrow Red$
        $x \leftarrow x.parent$
      end if
      if $s.right.color \leftarrow Black$ **then**
        $s.left.color \leftarrow Black$
        $s.color \leftarrow Red$
        RIGHT-ROTATE(s)
        $s \leftarrow x.parent.right$
      end if
      $s.color \leftarrow x.parent.color$
      $x.parent.color \leftarrow Black$
      $s.right.color \leftarrow Black$
      LEFT-ROTATE(x.parent)
      $x \leftarrow root$
    end if
    else (same as then clause with right and left exchanges)
  end while
  $root.color \leftarrow Black$
end function

1.5 RB Transplant

When a new element arrives, it is added to the heap in the insert function. There is no loops. Assignments get constant time. At the end of the function heap increase key function is called. Running time is O(lgn). Therefore, the execution time for heap insert is O(lgn).

Algorithm 5 RB Transplant

```
function RB-TRANSPLANT(u,v)
    if u.parent ← TNULL then
        root ← v
    end if
    if u ← u.parent.left then
        u.parent.left ← v
    end if
    else u.parent.right ← v     v.parent ← u.parent
end function
```

1.6 Minimum

Minimum function is used when finding the minimum node is needed. The leftmost node in the binary search tree will be small.

Algorithm 6 Minimum

```
function MINIMUM(x)
    while x.left ≠ TNULL do
        x ← x.left
    end while
    return x
end function
```

1.7 Get Root

Get Root function is used when trying to find root. Since the root node is kept in the red black tree class, it can be easily found.

Algorithm 7 Get Root

```
function GET-ROOT
    return this.root
end function
```

1.8 Exist Minimum

The virtual run time of the current running task is compared with that of other nodes. If it is still small, then it will continue to work. Otherwise it will be deleted from the tree.

Algorithm 8 Exist Minimum

```
function Exist-Minimum(x)
    if x ≠ root then
        if x.parent.vruntime <= x.vruntime then
            return true
        end if
        if x.right.name ≠ "" then
            if x.right.vruntime ← x.vruntime then
                return true
            end if
        end if
    end if
    return false
end function
```

1.9 Left Rotate

Left rotation may be required when adding or removing to preserve the red black tree property.

Algorithm 9 Left Rotate

```
function Left-Rotate(x)
    y ← x.right
    x.right ← y.left
    if y.left ≠ TNULL then
        root ← y
    end if
    if x ← x.parent.left then
        x.parent.left ← y
    end if
    if x.parent ← TNULL ANDx ≠ x.parent.left then
        x.parent.right ← y
    end if
    y.left ← x
    x.parent ← y
end function
```

1.10 Right Rotate

Right rotation may be required when adding or removing to preserve the red black tree property.

Algorithm 10 Right Rotate

function RIGHT-ROTATE(x)
  $y \leftarrow x.left$
  $x.left \leftarrow y.right$
  if $y.right \neq TNULL$ **then**
    $y.right.parent \leftarrow x$
  end if
  if $x \leftarrow x.parent.right$ **then**
    $root \leftarrow y$
  end if
  if $x.parent \neq TNULL\ ANDx \neq x.parent.right$ **then**
    $x.parent.left \leftarrow y$
  end if
  $y.right \leftarrow x$
  $x.parent \leftarrow y$
end function

2 Complexity Analysis

Left Rotate and Right Rotate functions run in O(1) time. Assignment operations take place in constant time. Only pointers are updated by a rotation.

Transplant function takes O(1) time. If else functions occur in constant time. Get Root function takes O(1) time, because it returns the root of tree.

In minimum function, the while loop iterates until it finds the minimum node and it takes O(lgn) time to find the minimum node since the height of the red black tree is always O(lgn).

Since red black tree is balance, its height is always O(lg n) where n is the number of nodes in the tree. Therefore the insertion and deletion operations run in O(lgn) time.

When case 1 happens in Insert Fixup, the while loop only repeats once, and the pointer goes two levels up the tree. The while loop can be executed in O(lgn) time. Therefore, Insert function takes total of O(lgn) time. In Delete Fixup, cases 1, 3 and 4 results in termination after completing a fixed number of color changes and a maximum of three rotations. While loop can be repeated only case 2. Then the pointer goes up the tree at most O(lgn) time doing no rotations. Therefore, Delete function takes total of O(lgn) time. Overall time complexity is O(lgn).

3 Food for Thought

Since the black red tree is balanced, its height is always O(lgn). In this way, search add and delete operations are performed in O(lgn) time. No matter how many processes are added, the height of the tree is maintained with rotations. Therefore, the upper bound is O(lgn).

CFS is used operating systems in the real world. Like many operating systems, Linux is a multitasking operating system. That's why it has a scheduler. Many processes are running simultaneously on the CPU. If there is one, it uses 100% of the process's power. If two, each uses 50%. This is ideal for the CPU. In the real world, the CPU can only perform one process, so other processes wait for their turn. However, this is not fair. CFS runs a fair clock and aims to reduce the processing and waiting time of processes fairly. The CPU selects the leftmost process in the red black tree and starts adding to the right as it processes. Thus, within a certain time, each process has a chance to be on the left.

4 Images of Outputs

After reading the arrival times and burst times of processes and run time from the input file, when the necessary operations are performed, first the run time, then the name of the currently running process, the virtual run time of currently running process, the minimum virtual run time, the tree structure and whether the processes are completed or not are printed. These values are separated by commas, while printing the structure of the tree, semicolons are used. Finally, how long it takes in ms, how many processes are completed and their order of completion are printed.

Figure 1: Output of 3 processes which 3 are completed

```
0,-,-,-,-,-
1,P1,0,0,P1:0-Black;Incomplete
2,P2,0,0,P2:0-Red;P3:0-Black;P1:1-Red;Incomplete
3,P3,0,0,P3:0-Red;P1:1-Black;P2:1-Red;Incomplete
4,P3,1,1,P3:1-Red;P1:1-Black;P2:1-Red;Complete
5,P1,1,1,P1:1-Black;P2:1-Red;Complete
6,P2,1,1,P2:1-Black;Completed

Scheduling finished in 0.166 ms.
3 of 3 processes are completed.
The order of completion of the tasks: P3-P1-P2
```

Figure 2: Output of 3 processes which 1 is completed

```
0,-,-,-,-,-,-
1,P1,0,0,P1:0-Black;Incomplete
2,P2,0,0,P2:0-Red;P3:0-Black;P1:1-Red;Incomplete
3,P3,0,0,P3:0-Red;P1:1-Black;P2:1-Red;Incomplete
4,P3,1,1,P3:1-Red;P1:1-Black;P2:1-Red;Complete

Scheduling finished in 0.142 ms.
1 of 3 processes are completed.
The order of completion of the tasks: P3
```

Figure 3: Output of 4 processes

```
0,-,-,-,-,-,-
1,P1,0,0,P1:0-Black;Incomplete
2,P2,0,0,P2:0-Black;P3:0-Black;P4:0-Red;P1:1-Black;Incomplete
3,P3,0,0,P3:0-Black;P4:0-Black;P1:1-Black;P2:1-Red;Incomplete
4,P4,0,0,P4:0-Black;P1:1-Black;P2:1-Black;P3:1-Red;Incomplete
5,P4,1,1,P4:1-Black;P1:1-Black;P2:1-Black;P3:1-Red;Complete
6,P1,1,1,P1:1-Black;P2:1-Black;P3:1-Black;Complete
7,P2,1,1,P2:1-Black;P3:1-Red;Complete
8,P3,1,1,P3:1-Black;Completed

Scheduling finished in 0.197 ms.
4 of 4 processes are completed.
The order of completion of the tasks: P4-P1-P2-P3
```