EE516 : Homework 4

Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

Make a program to construct red-black tree based on the virtual run time of each process.
Assume:

int pid;
int vrt[pid] = {27, 19, 34, 65, 37, 7, 49, 2, 98};

Remove the leftmost node and re-structure the red-black tree until all vrt[] nodes are removed from the tree.

```c
int main(int argc, const char *argv[])
{
    int pid;
    int vrt[] = {27, 19, 34, 65, 37, 7, 49, 2, 98};

    struct rb_tree *tree;
    tree = rb_create();

    struct rb_node *leftmost;

    for (pid = 0; pid < (sizeof(vrt) / sizeof(vrt[0])); pid++) {
        rb_insert(tree, vrt[pid]);
    }

    rb_print(tree);

    leftmost = find_leftmost(tree);
    while (leftmost != NULL) {
        info("DEL : %d", leftmost->vrt);
        rb_delete(tree, leftmost);
        leftmost = find_leftmost(tree);
    }

    free(tree->root);
    free(tree->nil);
    free(tree);

    return 0;
}
```

1. Create a new RB tree
2. Insert given virtual run-times as keys
3. Print RB tree using in-order traversal
4. Iteratively delete leftmost node until there is no node left

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/HW04 (master) $ ./task01
2
7
19
27
34
37
49
65
98
DEL : 2
DEL : 7
DEL : 19
DEL : 27
DEL : 34
DEL : 37
DEL : 49
DEL : 65
DEL : 98
```

In-order traversal of RB-Tree. Since RB-Tree is inherently a BST, the in-order traversal prints all elements in sorted order

Deleting leftmost node (least virtual run-time) is exploited in Completely Fair Scheduler (CFS) of Linux. The CFS keeps a cache of leftmost node & thus can schedule it in O(1) time

Importance of a Red-Black (RB) Tree

- A Binary Search Tree (BST) supports basic operations like SEARCH, INSERT, DELETE, PREDECESSOR, SUCCESSOR, MINIMUM and MAXIMUM in O(h) time
- A RB Tree guarantees:
  $h \le 2 \log (n+1)$

Thus all RB Tree operations are guaranteed to be O(log(n))

RB Node Attributes

struct **rb_node** {

color	... enum {**RED** / **BLACK**}
key	... Any comparable (< / = / >) data type
left_child	... struct rb_node *
right_child	... struct rb_node *
parent	... struct rb_node *

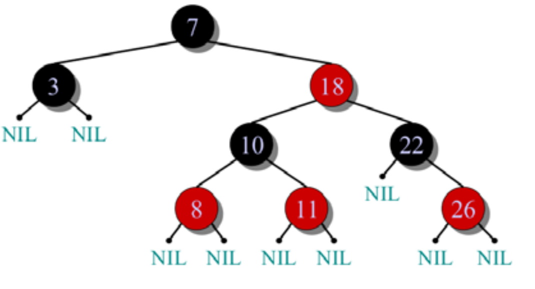
}

Concept of Black Height

- For a Node n:
  Number of black nodes on any simple path from (but not including node n) to any leaf (NIL)
  - Denoted by bh(n)
- For a RB Tree:
  bh(<root node>)

RB Tree Properties

- A RB Tree is a BST by definition with exactly 2 children for internal nodes and exactly 0 children for NIL (sentinel) nodes
- Every node is either **RED** or **BLACK**
- The root is **BLACK**
- Every leaf (NIL) is **BLACK**
- A **RED** node must have a **BLACK** parent
- All simple paths from a node to NIL contains the same number of **BLACK** nodes
- Since a RB Tree is inherently a BST, the leftmost node will always be the one with minimum value of key



References:

[1] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/lecture-10-red-black-trees-rotations-insertions-deletions/#vid_transcript

[2] http://web.mit.edu/~emin/Desktop/ref_to_emin/www.old/source_code/red_black_tree/index.html